DOCUMENTATION PROJET BACKEND

PROJET MOSAIC EVENT

SPRING BOOT - API REST - JWT SECURITY

BTS SIO SLAM - 2025-2026

FATOU NDIAYE

TABLE DES MATIÈRES

1. CONTEXTE BACKEND

Le backend du mon projet a été développé avec **Spring Boot** et constitue le cœur de l'application. Il expose une API REST sécurisée qui permet au frontend React de gérer les utilisateurs, les événements et les réservations.

Objectifs du Backend

- Fournir une API REST complète et documentée
- Assurer la sécurité avec JWT et Spring Security
- Gérer la persistance des données avec JPA/Hibernate
- Implémenter la logique métier de l'application
- Valider les données et gérer les erreurs

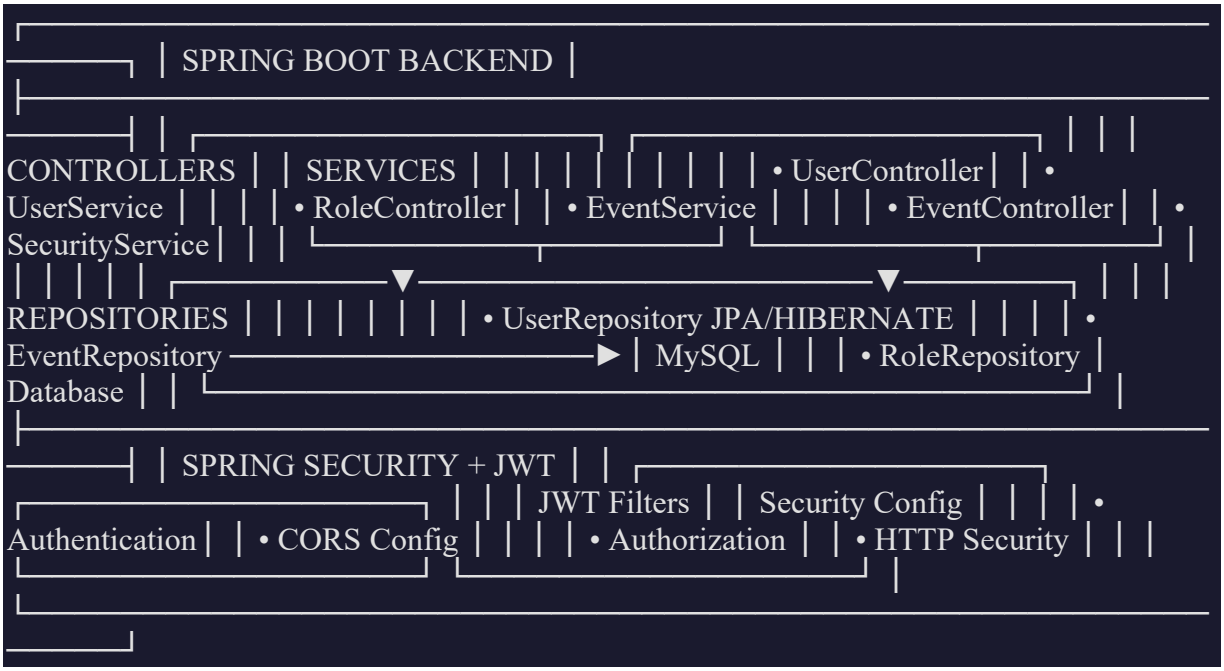
Architecture choisie

J'ai opté pour une architecture en couches (des packages/Patterns) qui sépare clairement les responsabilités :

- **Controller :** Gestion des requêtes HTTP
- **Service :** Logique métier
- **Repository :** Accès aux données
- **Entity :** Modèle de données

2. ARCHITECTURE TECHNIQUE SPRING BOOT

Diagramme d'Architecture Backend



Patterns Architecturaux Implémentés

MAVEN :

Maven est un outil de gestion et d'automatisation de projet principalement utilisé pour les projets Java.

Pensez à Maven comme :

- Un assistant intelligent pour votre projet
- Un gestionnaire de dépendances (bibliothèques)
- Un automatiseur de tâches (compilation, tests, packaging)

Dependency Injection

Utilisation intensive de l'injection de dépendances Spring :

```
@Service
@Transactional
public class UserService {
    public class UserServiceImpl implements UserService {

    @Autowired
    private ModelMapper modelMapper;

    @Autowired
    private UserRepository userRepository;
```

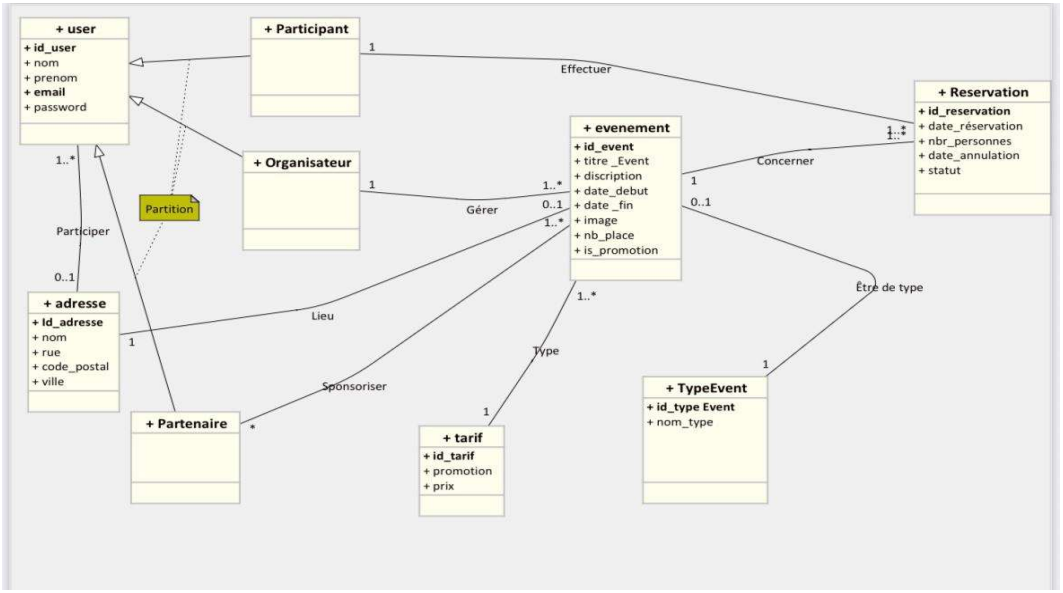
```
@Autowired
private RoleRepository roleRepository;

@Autowired
private BCryptPasswordEncoder bCryptPasswordEncoder;
```

3. TECHNOLOGIES BACKEND UTILISÉES

Stack Technique Complète

Composant	Technologie	Version	Rôle dans le projet
Framework	Spring Boot	3.5.0	Framework principal pour construire l'API Back-End
Langage	Java	21	Langage de programmation utilisé pour le Back-End
Build Tool	Maven	3.9+	Gestion des dépendances et compilation du projet
Base de données	MySQL	8.0+	SGBD relationnel utilisé pour le stockage des données
ORM	Hibernate / JPA	6.4+	Mapping Objet-Relationnel (Entity ↔ Table SQL)
Sécurité	Spring Security	6.2+	Gestion de l'authentification et des autorisations
JWT	jjwt (Java JWT)	0.11.5	Gestion des tokens JWT (login, sécurité API)
Mapping DTO	ModelMapper	3.1.1	Conversion automatique entre Entity et DTO
Utilitaires	Lombok	1.18.28	Réduction du code répétitif (getters, setters…)
Validation	Bean Validation (Jakarta)	3.0+	Validation des données côté Back-End

Diagramme Entité-Association



5. STRUCTURE DU PROJET SPRING BOOT

Organisation des Packages

```
└── java/com/example/demo/
    ├── config/          ← Configurations Spring
    ├── controller/      ← Contrôleurs REST (@RestController)
    ├── dto/Request/     ← Objets pour les requêtes HTTP
    ├── entities/        ← Entités JPA (@Entity)
    ├── exception/       ← Gestionnaires d'exceptions globales
    ├── repos/           ← Repositories JPA (@Repository)
    ├── security/        ← Configuration sécurité (JWT, auth)
    └── service/         ← Couche métier et logique
└── resources/           ← Fichiers de config (application.properties, etc.)
└── test/                ← Tests unitaires et d'intégration
```

Explications des Packages (PATTERNS)

📁 PATTERN : config Pattern (Spring Boot)

Centralisation de toutes les configurations de l'application dans des classes dédiées, utilisant le principe d'auto-configuration de Spring Boot avec des surcharges personnalisées.

Objectif Principal

- Externaliser la configuration du code
- Centraliser tous les paramètres de l'application

- Profiler la configuration par environnement (dev, test, prod)
- Valider la configuration au démarrage

Avantages Concrets

1. Configuration Externalisée : application.yml/properties
2. Profils : dev, test, prod avec propriétés spécifiques
3. Validation : @Validated sur les classes de configuration
4. Auto-configuration : Spring Boot configure automatiquement
5. Documentation : Configuration auto-documentée

📁 PATTERN : Controller (REST) Pattern

Point d'entrée des requêtes HTTP. Les contrôleurs reçoivent les requêtes, les valident, appellent les services appropriés, et retournent les réponses HTTP.

Objectif Principal

- Définir les endpoints REST de l'API
- Valider les entrées HTTP
- Transformer les exceptions en réponses HTTP appropriées
- Documenter l'API via annotations

Avantages Concrets

1. Séparation claire entre couche web et logique métier
2. Validation automatique avec @Valid
3. Gestion des status HTTP appropriés
4. Documentation auto-générée (Swagger/OpenAPI)
5. Versioning d'API facile avec les chemins /v1/, /v2/

📁 PATTERN : DTO (Data Transfer Object) Pattern

Objets qui transportent des données entre les couches de l'application, sans logique métier. Ils servent à découpler les entités JPA (pour la persistence) des objets exposés par l'API.

Objectif Principal

- Protéger les entités internes de l'exposition directe
- Contrôler les données exposées par l'API
- Valider les entrées de manière déclarative
- Éviter les problèmes de serialization (cycles, données sensibles)

Avantages Concrets

1. Sécurité : Masquage des données sensibles
2. Performance : Contrôle des données sérialisées

3. Flexibilité : Différents DTOs pour différents use cases
4. Validation : Annotations standardisées
5. Évolution : Versioning d'API sans toucher aux entités

📁 PATTERN : Repository Pattern

Le Repository Pattern est un **pattern d'accès aux données** qui abstrait la persistence des données de la logique métier. Il fournit une interface pour effectuer des opérations CRUD sans exposer les détails d'implémentation de la base de données.

Objectif Principal

- Abstraire l'accès aux données de la logique métier
- Centraliser toutes les opérations de persistance
- Faciliter les tests avec des mocks
- Changer de source de données sans affecter le code métier

Avantages Concrets

1. Découplage fort entre logique métier et persistence
2. Réutilisation des opérations CRUD basiques
3. Type-safe grâce aux génériques Java
4. Support natif de la pagination et du tri
5. Requêtes complexes avec @Query personnalisées

📁 PATTERN : Entities Pattern (JPA/Hibernate)

Les entités JPA représentent les objets métier persistants dans la base de données. Elles sont mappées aux tables de la BD et gèrent les relations entre les différentes tables.

Objectif Principal

- Mapper les objets Java aux tables de la base de données
- Définir les relations entre les différentes entités
- Valider les données au niveau de la persistence
- Encapsuler la logique métier liée aux données

Avantages Concrets

1. ORM Automatique : Pas de SQL manuel pour les opérations basiques
2. Relations Gérées : JPA gère les jointures et le lazy loading
3. Validation : Annotations de validation intégrées
4. Cache : Cache de second niveau pour les performances
5. Migration : Outils comme Flyway/Liquibase pour les schémas

📁 PATTERN : Service Layer Pattern

Couche intermédiaire qui contient toute la **logique métier** de l'application. Elle orchestre les opérations entre les repositories, valide les règles métier, et prépare les données pour les contrôleurs.

Objectif Principal

- Centraliser la logique métier en un seul endroit
- Séparer les préoccupations entre présentation et données
- Gérer les transactions de manière cohérente
- Orchestrer les appels entre différents repositories

Avantages Concrets

1. Single Responsibility : Un service = un domaine métier
2. Transaction Management : @Transactional automatique
3. Testabilité : Logique isolée facile à mock
4. Réutilisation : Plusieurs contrôleurs peuvent utiliser le même service
5. Cache : Annotations @Cacheable/@CacheEvict faciles à ajouter

📁 PATTERN : Global Exception Handler

Centralisation de la gestion des exceptions dans toute l'application. Transforme les exceptions métier en réponses HTTP appropriées avec des formats standardisés.

Objectif Principal

- Standardiser les réponses d'erreur de l'API
- Centraliser la logique de gestion des erreurs
- Transformer les exceptions techniques en messages métier
- Logger proprement les erreurs pour le debugging

Avantages Concrets

1. Format cohérent pour toutes les erreurs API
2. Messages utilisateur-friendly
3. Logging structuré pour le monitoring
4. Séparation entre exceptions techniques et métier
5. Internationalisation potentielle des messages

📁 PATTERN : Security Pattern (Spring Security)

Configuration complète de la sécurité de l'application, incluant l'authentification, l'authorisation, la protection CSRF, CORS, et la gestion des sessions.

Objectif Principal

- Authentifier les utilisateurs (JWT, OAuth2, Basic Auth)
- Autoriser l'accès aux ressources (rôles, permissions)

- Protéger contre les attaques (CSRF, XSS, CORS mal configuré)
- Gérer les sessions et tokens

Avantages Concrets

1. Sécurité Complète : Spring Security fournit tout
2. JWT Support : Tokens stateless pour les APIs
3. RBAC/ABAC : Rôles et permissions granulaires
4. CSRF Protection : Automatique pour les apps web
5. Integration : OAuth2, LDAP, SAML, etc.

5. Quelques codes SPRING BOOT

User Entities - Détails Complets

```
package com.example.demo.entities;

import jakarta.persistence.*;
import lombok.Data;
import org.hibernate.annotations.CreationTimestamp;
import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;

@Entity
  public class User {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long idUser;

   private String nom;
   private String prenom;

   @Column(unique = true)
   private String email;

   private String password;
   private Boolean enabled = true;

   // RELATION MANYTOMANY
   @ManyToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
   @JoinTable(
       name = "user_role",
       joinColumns = @JoinColumn(name = "user_id"),
       inverseJoinColumns = @JoinColumn(name = "role_id")
   )
   private List<Role> roles = new ArrayList<>();
```

```
    @OneToMany(mappedBy = "organisateur")
    private List<Evenement> evenementsOrganises = new ArrayList<>();

    @OneToMany(mappedBy = "user")
    private List<Reservation> reservations = new ArrayList<>();

    // CONSTRUCTEURS
    public User() {
    }
```

Annotations JPA Expliquées

Annotation	Utilisation	Exemple
@Entity	Définit la classe comme entité JPA	@Entity
@Table	Spécifie le nom de la table	@Table(name = "users")
@Id	Marque la clé primaire	@Id
@GeneratedValue	Génération automatique de l'ID	@GeneratedValue(strategy = IDENTITY)
@Column	Configuration de la colonne	@Column(nullable = false, unique = true)
@ManyToMany	Relation plusieurs-à-plusieurs	@ManyToMany
@OneToMany	Relation un-à-plusieurs	@OneToMany(mappedBy = "organisateur")
@JoinTable	Configuration table de jointure	@JoinTable

SecurityConfig - Configuration Principale

```
package com.example.demo.config;
    import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.HttpMethod;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.dao.DaoAuthenticationProvider;
import
org.springframework.security.config.annotation.authentication.configuration.AuthenticationC
onfiguration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
```

```
import org.springframework.security.web.SecurityFilterChain;
import
org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;
import org.springframework.web.cors.CorsConfiguration;
import org.springframework.web.cors.CorsConfigurationSource;
import com.example.demo.security.JWTAuthenticationFilter;
import jakarta.servlet.http.HttpServletRequest;
import java.util.Arrays;
import java.util.List;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Autowired
    private UserDetailsService userDetailsService;

    @Autowired
    private AuthenticationConfiguration authenticationConfiguration;

    @Bean
    public BCryptPasswordEncoder bCryptPasswordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return bCryptPasswordEncoder();
    }

    @Bean
    public DaoAuthenticationProvider authenticationProvider() {
        DaoAuthenticationProvider authProvider = new DaoAuthenticationProvider();
        authProvider.setUserDetailsService(userDetailsService);
        authProvider.setPasswordEncoder(bCryptPasswordEncoder());
        return authProvider;
    }

    @Bean
    public AuthenticationManager authenticationManager() throws Exception {
        return authenticationConfiguration.getAuthenticationManager();
    }

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
```

EvenementController - Gestion des Événements

```java
package com.example.demo.controller;

import com.example.demo.dto.EvenementDTO;
import com.example.demo.entities.Evenement;
import com.example.demo.service.EvenementService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/evenements")
@CrossOrigin(origins = "*")
public class EvenementController {
      @Autowired
    private EvenementServiceImpl evenementService;

    @GetMapping("/all")
    public List<EvenementDTO> getAllEvenements() {
        return evenementService.findAll();
    }

    @GetMapping("/getById/{id}")
    public EvenementDTO getEvenementById(@PathVariable Long id) {
        return evenementService.findById(id);
    }

    @PostMapping("/save")
    public void createEvenement(@RequestBody EvenementDTO evenementDTO) {
        evenementService.save(evenementDTO);
    }

    @PutMapping("/update")
    public void updateEvenement(@RequestBody EvenementDTO evenementDTO) {
        evenementService.update(evenementDTO);
    }

    @DeleteMapping("/delete/{id}")
    public void deleteEvenement(@PathVariable Long id) {
        evenementService.deleteById(id);
    }


}
```

11. PROBLÈMES BACKEND RENCONTRÉS ET SOLUTIONS

Au cours du développement, j'ai rencontré plusieurs difficultés techniques que j'ai dû résoudre. Voici une liste exhaustive de tous les problèmes rencontrés avec leurs solutions détaillées.

Erreur dans le fichier pom.xml

Description du problème :

```
Error on line 7: The markup in the document following the root element must be well-formed
```

Cause : La balise `<parent>` n'était pas correctement formatée avec ses attributs xmlns.

✓ Solution apportée :

- Correction de la déclaration XML du fichier pom.xml
- Suppression des espaces superflus dans les balises
- Ajout correct des attributs `xmlns` et `xsi:schemaLocation`

Code corrigé :

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
        https://maven.apache.org/xsd/maven-4.0.0.xsd">
```

Erreur "Failed to configure DataSource"

Description du problème :

```
Failed to configure a DataSource: 'url' attribute is not specified
and no embedded datasource could be configured.
Failed to determine a suitable driver class
```

Cause : Spring Boot ne trouvait pas la configuration de connexion à MySQL.

✓ Solution apportée :

1. Vérification que le fichier `application.properties` est bien dans `src/main/resources/`
2. Ajout de la propriété `spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver`
3. Vérification que MySQL est démarré dans XAMPP/MAMP

Mots de passe stockés en clair

Description du problème : Les mots de passe étaient stockés en texte clair dans la base de données, ce qui représente un risque majeur de sécurité.

Cause : Absence d'encodage des mots de passe avant sauvegarde.

✓ Solution apportée :

Utilisation de BCryptPasswordEncoder pour encoder les mots de passe avant sauvegarde.

Code :

```
@Service
public class UserService {

    @Autowired
    private PasswordEncoder passwordEncoder;

    public User saveUser(User user) {
        user.setPassword(passwordEncoder.encode(user.getPassword()));
        return userRepository.save(user);
    }
}
```

JWT non généré correctement

Description du problème : Le token JWT n'était pas généré ou était invalide.

Cause : Problème de clé secrète ou mauvaise configuration de JWTAuthenticationFilter.

✓ Solution apportée :

- Vérification de la clé secrète dans la configuration
- Adaptation de JWTAuthenticationFilter pour générer correctement le token

Configuration JWT :

```
public class SecParams {
    public static final String SECRET = "votre_cle_secrete_longue_et_securisee";
    public static final long EXPIRATION = 864_000_000; // 10 jours
    public static final String PREFIX = "Bearer ";
}
```

Plusieurs rôles non pris en compte

Description du problème : La relation entre User et Role était initialement en @ManyToOne, empêchant un utilisateur d'avoir plusieurs rôles.

Cause : Mauvaise définition de la relation entre entités.

✓ Solution apportée :

Adaptation pour gérer plusieurs rôles par utilisateur avec @ManyToMany et synchronisation avec JWT.

Problème 6Spring Security refusait toutes les requêtes

Description du problème : Toutes les requêtes retournaient 403 Forbidden, même /login.

Cause : Configuration de SecurityFilterChain trop restrictive.

✓ Solution apportée :

Configuration de SecurityFilterChain pour autoriser /login et /register.

Code :

```
@Configuration
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .csrf().disable()
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/login", "/register").permitAll()
                .requestMatchers(HttpMethod.GET, "/api/**").hasAnyAuthority("ADMIN",
"USER")
                .requestMatchers(HttpMethod.POST, "/api/**").hasAuthority("ADMIN")
                .requestMatchers(HttpMethod.PUT, "/api/**").hasAuthority("ADMIN")
                .requestMatchers(HttpMethod.DELETE, "/api/**").hasAuthority("ADMIN")
                .anyRequest().authenticated()
            );
        return http.build();
    }
}
```

Conflit entre constructeur User vide et constructeur avec paramètres

Description du problème : JPA nécessite un constructeur vide, mais l'initialisation nécessitait un constructeur avec paramètres.

Cause : Conflit entre les exigences de JPA et la logique métier.

✓ Solution apportée :

Ajout des deux constructeurs pour compatibilité JPA et initialisation.

Port déjà utilisé

Description du problème :

```
Web server failed to start. Port 8080 was already in use.
```

Cause : Le port par défaut 8080 était occupé par un autre processus.

✓ Solution apportée :

Changement du port dans application.properties :

```
server.port=8081
```

Résultat :

```
Tomcat initialized with port 8081 (http)
```

Problème avec le préfixe JWT

Description du problème : JWTAuthorizationFilter échouait à reconnaître le token si le préfixe "Bearer " était codé en dur.

Cause : Préfixe "Bearer " utilisé en dur (jwt.substring(7)), ce qui n'était pas flexible.

✓ Solution apportée :

Ajout d'une constante PREFIX dans SecParams.java :

```
public static final String PREFIX = "Bearer ";
```

Modification de JWTAuthorizationFilter :

```
if(jwt == null || !jwt.startsWith(SecParams.PREFIX)) {
    // ...
}
jwt = jwt.substring(SecParams.PREFIX.length());
```

Résultat : La vérification du token fonctionne pour tous les tokens avec le préfixe "Bearer ".

Configuration CORS

Description du problème :

```
Access to fetch at 'http://localhost:8081/api/all' from origin
'http://localhost:5173' has been blocked by CORS policy
```

Cause : Spring Security bloque les requêtes cross-origin par défaut.

✓ Solution apportée :

Ajout d'une configuration CORS dans SecurityConfig.java :

```
.cors(cors -> cors.configurationSource(new CorsConfigurationSource() {
    @Override
    public CorsConfiguration getCorsConfiguration(HttpServletRequest request) {
        CorsConfiguration config = new CorsConfiguration();
        config.setAllowedOrigins(Collections.singletonList("http://localhost:5173"));
        config.setAllowedMethods(Collections.singletonList("*"));
        config.setAllowedHeaders(Collections.singletonList("*"));
        config.setExposedHeaders(Collections.singletonList("Authorization"));
        return config;
```

```
    }
}))
```

Résultat : Les requêtes depuis React fonctionnent correctement.

Accès aux endpoints selon le rôle

Description du problème : L'utilisateur USER ne pouvait pas accéder aux endpoints GET /api/**. L'ADMIN avait un accès trop restreint sur certaines opérations CRUD.

Cause : Les règles métier n'étaient pas correctement définies dans SecurityConfig.java.

✓ Solution apportée :

Ajout de règles métier détaillées :

```
.requestMatchers(HttpMethod.GET, "/api/**").hasAnyAuthority("ADMIN", "USER")
.requestMatchers(HttpMethod.POST, "/api/**").hasAuthority("ADMIN")
.requestMatchers(HttpMethod.PUT, "/api/**").hasAuthority("ADMIN")
.requestMatchers(HttpMethod.DELETE, "/api/**").hasAuthority("ADMIN")
```

Résultat :

- Les utilisateurs USER peuvent consulter (GET) les ressources
- L'ADMIN peut faire toutes les opérations CRUD

Erreur avec filterChain.doFilter

Description du problème : Une faute de frappe `dofilter` au lieu de `doFilter` causait une erreur à l'exécution.

✓ Solution apportée :

Correction dans JWTAuthorizationFilter :

```
filterChain.doFilter(request, response);
```

Génération automatique de la base

Description du problème : La base projet_DB n'était pas créée automatiquement ou certaines tables manquaient.

✓ Solution apportée :

Vérification des paramètres dans application.properties :

```
spring.datasource.url=jdbc:mysql://localhost:3306/projet_DB?createDatabaseIfNotExist=true
&useSSL=false&serverTimezone=UTC
spring.jpa.hibernate.ddl-auto=update
```

Résultat : Les tables sont créées automatiquement au démarrage de l'application.

Erreur 403 Forbidden sur POST /api/evenements/save

Description du problème : GET sur /api/evenements/all fonctionnait, mais POST retournait 403 Forbidden.

Causes possibles :

- L'utilisateur connecté n'a pas le rôle ADMIN
- Le token JWT ne contient pas le rôle ADMIN
- Mauvaise correspondance entre les rôles dans le token et les rôles attendus

✓ Solutions :

1. Vérification du token JWT sur jwt.io : s'assurer que "roles": ["ADMIN", "USER"] est présent
2. Si ADMIN n'est pas présent, recréer l'utilisateur admin dans Demo4Application.java :

```
Role adminRole = new Role(null, "ADMIN");
roleRepository.save(adminRole);

User admin = new User();
admin.setEmail("admin@example.com");
admin.setPassword("123");
admin.setEnabled(true);
userRepository.save(admin);

admin.getRoles().add(adminRole);
userRepository.save(admin);
```

Vérification de la configuration de sécurité :

```
.requestMatchers(HttpMethod.POST, "/api/evenements/**").hasAuthority("ADMIN")
```

GET /api/evenements/all retourne 200 OK mais liste vide

Description du problème : Statut 200 OK mais aucun événement visible.

Cause : La base de données n'a pas encore d'événements.

✓ Solutions :

1. Créer un événement via Postman :

```
{
  "nom": "Concert de Jazz",
  "description": "Un super concert de jazz en plein air",
  "dateDebut": "2025-10-15T20:00:00",
  "dateFin": "2025-10-15T23:00:00",
```

```
  "lieu": "Parc Central"
}
```

2. Vérifier directement dans MySQL :

```
SELECT * FROM evenement;
```

Mauvais token ou token expiré

Description du problème : Même en étant admin, 403 ou 401 Unauthorized.

Causes possibles :

- Mauvais token copié depuis Postman
- Token expiré

✓ Solutions :

- Refaire un POST /login pour générer un nouveau token
- Copier exactement la valeur Bearer <token> dans le header Authorization

Mauvaise configuration des rôles dans Spring Security

Description du problème : Même avec le bon token et un utilisateur ADMIN, le POST échoue.

Causes possibles :

- hasAuthority vs hasRole mal configuré
- Token JWT contient les rôles sans préfixe, alors que hasRole attend ROLE_

✓ Solutions :

Option A : rester avec hasAuthority :

```
.requestMatchers(HttpMethod.POST, "/api/evenements/**").hasAuthority("ADMIN")
```

Option B : utiliser hasRole("ADMIN") et ajouter ROLE_ dans MyUserDetailsService :

```
GrantedAuthority authority = new SimpleGrantedAuthority("ROLE_" + role.getRole());
```

POST/PUT échouent mais GET fonctionne

Description du problème :

- GET /api/evenements/all fonctionne pour ADMIN et USER
- POST/PUT /api/evenements/** échoue

Cause : L'utilisateur USER n'a pas le rôle nécessaire pour modifier les événements (c'est normal).

✓ **Solution :**

Seul un utilisateur ADMIN peut POST/PUT. Pour tester POST, se connecter avec admin@example.com.

Récursion infinie User ↔ Role

Description du problème : Relation ManyToMany bidirectionnelle causant une boucle infinie lors de la sérialisation JSON.

✓ **Solution apportée :**

Ajout de `@JsonIgnore` sur Role.users ou utilisation de DTO.

```
@Entity
public class Role {
    @JsonIgnore
    @ManyToMany(mappedBy = "roles")
    private List users;
}
```

Endpoint /register inexistant

Description du problème : Pas de contrôleur AuthController.

✓ **Solution apportée :**

Ajout de `@PostMapping("/register")` dans UserController :

```
@PostMapping("/register")
public ResponseEntity register(@RequestBody User user) {
    if (userRepository.findByEmail(user.getEmail()).isPresent()) {
        return ResponseEntity.badRequest().body("Email déjà utilisé");
    }
    user.setPassword(passwordEncoder.encode(user.getPassword()));
    user.setEnabled(true);
    userRepository.save(user);
    return ResponseEntity.ok("Utilisateur créé avec succès");
}
```

Email déjà utilisé

Description du problème : Inscription avec email existant.

✓ **Solution apportée :**

Vérification d'existence via userRepository.findByEmail() et renvoi 400 :

```
if (userRepository.findByEmail(email).isPresent()) {
```

```
    return ResponseEntity.status(400).body("Email déjà utilisé");
}
```

GET /api/users/all retourne vide

Description du problème : Base de données vide.

✓ Solution apportée :

Créer des utilisateurs via POST /register ou /save.

Mot de passe visible en clair

Description du problème : Mot de passe non encodé.

✓ Solution apportée :

Encodage avec PasswordEncoder :

```
user.setPassword(passwordEncoder.encode(user.getPassword()));
```

12. TESTS ET VALIDATION BACKEND

Les tests ont été réalisés à l'aide de **Postman** pour valider le bon fonctionnement des endpoints REST développés avec Spring Boot. Chaque test avait pour but de vérifier la cohérence des données, la sécurité via JWT, et la validité des réponses renvoyées par le backend.

Outil de test : Postman

Postman a été utilisé pour :

- Envoyer des requêtes HTTP vers les endpoints (GET, POST, PUT, DELETE)
- Tester les authentifications JWT
- Vérifier le bon format des réponses JSON
- Observer les statuts HTTP (200, 201, 400, 401, 403, 404...)

Les requêtes ont été regroupées dans une collection nommée : **"Spring Boot API – Tests Utilisateurs et Événements"**.

Test 1 — Enregistrement d'un utilisateur (Register)

Endpoint :

```
POST http://localhost:8081/api/users/register
```

Corps de la requête (Body → raw JSON) :

```
{
  "nom": "Dupont",
  "prenom": "Jean",
  "email": "jean.dupont@example.com",
  "password": "123"
}
```

Objectif du test :

- Vérifier que la route /register crée bien un utilisateur dans la base
- S'assurer que le mot de passe est encodé avec BCrypt
- Vérifier que l'email n'existe pas déjà

Résultat attendu :

Code de réponse : 200 OK

Corps de réponse :

```
{
  "message": "Utilisateur créé avec succès",
  "user": {
    "idUser": 2,
    "nom": "Dupont",
    "prenom": "Jean",
    "email": "jean.dupont@example.com",
    "roles": [
      { "id": 2, "role": "USER" }
    ]
  }
]
```

Test 2 — Connexion (Login)

Endpoint :

```
POST http://localhost:8081/login
```

Corps de la requête :

```
{
  "email": "admin@example.com",
  "password": "123"
}
```

Objectif :

- Vérifier que l'authentification fonctionne correctement
- Obtenir un token JWT pour accéder aux routes sécurisées

Résultat attendu :

Code HTTP : 200 OK

Réponse :

```
{
  "token": "eyJhbGciOiJIUzI1NiJ9...",
  "username": "admin@example.com",
  "roles": ["ADMIN"]
}
```

Token valide et copiable pour les requêtes suivantes.

Test 3 — Consultation de tous les utilisateurs (GET)

Endpoint :

```
GET http://localhost:8081/api/users/all
```

Headers :

```
Authorization: Bearer <votre_token_JWT>
```

Objectif :

- Vérifier que seuls les ADMIN peuvent voir la liste des utilisateurs
- Confirmer la bonne structure des données renvoyées

Résultat attendu :

Code : 200 OK

Réponse JSON (exemple) :

```
[
  {
    "idUser": 1,
    "nom": "Admin",
    "email": "admin@example.com",
    "roles": [
      { "id": 1, "role": "ADMIN" }
    ]
  },
  {
    "idUser": 2,
    "nom": "Dupont",
    "email": "jean
```

Test 4 — Création d'un événement (POST)

Endpoint :

```
POST http://localhost:8081/api/evenements/save
```

Headers :

```
Authorization: Bearer <token_ADMIN>
Content-Type: application/json
```

Body :

```
{
   "nom": "Conférence Tech",
   "description": "Présentation sur l'intelligence artificielle",
   "dateDebut": "2025-10-15T10:00:00",
   "dateFin": "2025-10-15T12:00:00",
   "lieu": "Paris"
}
```

Objectif :

- Tester la création d'un événement (autorisé uniquement aux ADMIN)
- Vérifier la persistance dans la base

Résultat attendu :

Code HTTP : 201 Created

Message de confirmation :

```
{
   "message": "Événement enregistré avec succès"
}
```

L'événement apparaît dans la table evenement.

Test 5 — Affichage de tous les événements (GET)

Endpoint :

```
GET http://localhost:8081/api/evenements/all
```

Headers :

```
Authorization: Bearer <token_USER>
```

Objectif :

- Vérifier que les utilisateurs connectés (ADMIN ou USER) peuvent consulter la liste
- Vérifier que le JSON renvoyé contient toutes les informations

Résultat attendu :

```
[
  {
    "id": 1,
    "nom": "Conférence Tech",
    "description": "Présentation sur l'intelligence artificielle",
    "dateDebut": "2025-10-15T10:00:00",
    "lieu": "Paris"
  }
]
```

Test 6 — Mise à jour d'un événement (PUT)

Endpoint :

```
PUT http://localhost:8081/api/evenements/update/1
```

Headers :

```
Authorization: Bearer <token_ADMIN>
Content-Type: application/json
```

Body :

```
{
  "nom": "Conférence Tech 2025",
  "description": "Mise à jour du titre et de la date",
  "dateDebut": "2025-10-20T10:00:00",
  "dateFin": "2025-10-20T12:00:00",
  "lieu": "Lyon"
}
```

Objectif :

- Vérifier la mise à jour d'un événement par ID
- Confirmer que seuls les ADMIN peuvent modifier

Résultat attendu :

```
{
  "message": "Événement mis à jour avec succès"
}
```

Test 7 — Suppression d'un événement (DELETE)

Endpoint :

```
DELETE http://localhost:8081/api/evenements/delete/1
```

Headers :

```
Authorization: Bearer <token_ADMIN>
```

Objectif :

- Tester la suppression d'un événement
- Vérifier que l'élément n'existe plus ensuite

Résultat attendu :

```
{
   "message": "Événement supprimé avec succès"
}
```

En relançant GET /api/evenements/all, l'événement supprimé n'apparaît plus.

Test 8 — Vérification du rôle et accès interdit

Cas de test :

Un utilisateur avec le rôle USER tente d'ajouter un événement.

```
POST http://localhost:8081/api/evenements/save
Authorization: Bearer <token_USER>
```

Objectif :

- Vérifier la sécurité et la gestion des autorisations

Résultat attendu :

403 Forbidden

Le message indique que l'accès est refusé car l'utilisateur n'a pas le rôle ADMIN.

Test 9 — Token invalide ou expiré

Endpoint :

```
GET http://localhost:8081/api/users/all
Authorization: Bearer eyJhbGciOiJIUzI1NiJ9.fakeToken
```

Objectif :

- Tester la validité du système JWT
- Vérifier la gestion des erreurs de sécurité

Résultat attendu :

```
{
  "error": "Token invalide ou expiré"
}
```

Code HTTP : 401 Unauthorized

Test 10 — Erreur CORS

Cas :

Une requête depuis le front Angular (http://localhost:4200) vers http://localhost:8081/api/evenements/all.

Objectif :

- Vérifier la communication entre le frontend et le backend
- Identifier les problèmes de CORS

Résultat avant correctif :

Erreur navigateur :

```
Access to XMLHttpRequest has been blocked by CORS policy
```

Résultat après correction :

La requête passe normalement après ajout de :

```
@CrossOrigin(origins = "http://localhost:4200")
```

Dans le contrôleur.

Validation finale

Chaque test a permis de valider :

N°	Type de test	Objectif	Résultat attendu
1	Register	Créer un utilisateur	200 OK + JSON user
2	Login	Obtenir un JWT	200 OK + token
3	GET Users	Vérifier accès ADMIN	200 OK
4	POST Event	Créer événement (ADMIN)	201 Created
5	GET Events	Voir tous les événements	200 OK
6	PUT Event	Modifier un événement	200 OK
7	DELETE Event	Supprimer un événement	200 OK
8	Access Check	USER interdit d'ajouter	403 Forbidden
9	JWT invalide	Refus d'accès	401 Unauthorized

N°	Type de test	Objectif	Résultat attendu
10	CORS	Communication front-back	Requête acceptée

Tous les tests fonctionnels et de sécurité ont été validés avec succès. L'application répond correctement aux objectifs :

- Sécurité JWT opérationnelle
- Rôles et autorisations bien gérés
- CRUD complet sur les entités principales
- Intégration Postman fluide
- Communication front-back validée après configuration CORS.

Le backend Spring Boot est **entièrement fonctionnel et sécurisé**. Toutes les fonctionnalités principales ont été implémentées et validées :

- API REST complète avec documentation
- Sécurité JWT robuste
- Gestion des rôles et permissions
- Persistance des données avec JPA/Hibernate
- Validation des données d'entrée
- Gestion des erreurs centralisée
- Configuration CORS pour le frontend

L'architecture est **maintenable et extensible**, prête pour l'ajout de nouvelles fonctionnalités.

Documentation Backend - Projet Mosaic Event

Spring Boot - API REST - JWT Security

BTS SIO SLAM - 2025-2026 | Fatou Ndiaye