

# Équida

Doc Technique

MARTIN Justine  
BOTTON Léa

# Table des matières

<b>1</b>	<b>Contexte et présentation</b>	<b>1</b>
A	Présentation du contexte . . . . .	1
B	Choix techno . . . . .	1
C	Organisation du projet . . . . .	2
D	Interactions entre les différentes parties du projet . . . . .	4
<b>2</b>	<b>Core</b>	<b>7</b>
A	Organisation des packages . . . . .	7
B	Exemple d'Entity . . . . .	7
C	Exemple de Repository . . . . .	9
D	Exemple de Service . . . . .	9
E	Exemple d'exception (NotFoundException) . . . . .	11
F	Authentification . . . . .	12
G	Utils . . . . .	12
<b>3</b>	<b>Application web</b>	<b>13</b>
A	Organisation des packages . . . . .	13
B	Configuration de l'application . . . . .	15
C	Fichiers ressources . . . . .	17
D	Gestion de l'authentification . . . . .	20
E	Exemple Route . . . . .	20
F	Exemple Form . . . . .	21
G	Classe InputOutputAttribute . . . . .	23
H	Les Contrôleurs . . . . .	24
<b>4</b>	<b>Application mobile</b>	<b>26</b>
A	API REST . . . . .	26
B	Ionic . . . . .	30
<b>5</b>	<b>Ressources</b>	<b>42</b>

# 1 Contexte et présentation

## A Présentation du contexte

---

Créée en 2006, Equida est une société spécialisée dans la vente aux enchères de chevaux de course. Avec un effectif de vingt-sept personnes, la société a réalisé, en 2012, un chiffre d'affaire de 87 millions d'euros. Ses clients sont des vendeurs de chevaux, principalement des haras, des entraîneurs et de grands propriétaires de chevaux, situés en France et à l'étranger. Pour être plus proche de sa clientèle étrangère, elle s'appuie sur une quinzaine de correspondants répartis dans de nombreux pays comme l'Irlande, la Turquie, ou encore le Japon.

Pour gérer son activité, la société utilise un site web qui permet notamment la consultation des ventes, une application Planning qui permet de gérer les clients, une application de gestion des ventes aux enchères ainsi qu'une application de gestion des informations des chevaux.

Equida souhaite combiner ses différentes applications en une seule, ce qui lui permettra donc de gérer les chevaux et leurs informations, leurs mises en vente, leurs enchères ainsi que les clients et leur compte.

Cette application combine des fonctionnalités pour les clients (enregistrement d'un cheval, proposition d'un cheval à une vente, ...) et des fonctionnalités pour l'administrateur (gestion des clients et de leur compte, validation des propositions, ajout de ventes, ...), elle devra contenir deux niveaux d'authentification.

Pour plus d'informations, vous pouvez consulter la totalité du [cahier des charges](#).

On a fait le choix de réaliser un seul projet contenant deux applications : une application web (voir [Application web](#)) et une application mobile (voir [Application mobile](#)) qui utilisera une api. L'application web est plus complète car destinée à être utilisée surtout par l'administrateur ; l'application mobile, elle, se concentre surtout sur des fonctionnalités propres à l'utilisateur avec tout de même quelques possibilités de gestion pour l'administrateur.

## B Choix techno

---

### i MySQL

MySQL, en comparaison avec Oracle, est open source et gratuit. Ce qui le rend avantageux. De plus, le cahier des charges fourni par le client retenait la solution de MySQL pour la base de données.

### ii Spring Boot

On fait le choix de Spring Boot pour le projet car c'est, très probablement, le framework Java pour le développement web le plus utilisé.

Il permet de créer facilement un contrôleur, en effet, il suffit de créer une classe et de l'annoter @Controller. Les méthodes auront l'annotation @GetMapping, @PostMapping ou pour toute autre méthode HTTP, l'annotation suivant le modèle @XMapping, qui indique l'URL de la page, ainsi que la méthode HTTP,

qui lui correspond.

De plus, il embarque l'équivalent d'un serveur TomCat lors de la compilation du projet (en faisant `Tasks > boot > bootRun` sur netbeans ou `gradle bootRun` en ligne de commande) qui permet le démarrage du projet par l'intermédiaire du serveur web embarqué.

Vous pouvez retrouver ce framework [ici](#) ainsi que des tutoriels pour l'utiliser [ici](#) et [là](#).

### iii Ionic

Ionic est un framework open source qui permet de créer des applications multiplateformes (mobile et navigateur) performantes tout en utilisant des technologies web connues (HTML, CSS, JavaScript). Il possède en plus une intégration d'Angular qui permet d'appréhender une page web comme un assemblage de composants webs indépendants mais qui peuvent communiquer entre eux.

Ionic intègre également des composants visuels natifs, ainsi, les utilisateurs d'Android ou d'iOs conserveront leurs habitudes sur l'application.

Vous pouvez retrouver ce framework [ici](#) avec sa documentation [là](#).

### iv Gradle

Gradle est un "build automation system" (moteur de production). Il est un équivalent plus récent et plus complet à Maven. Il possède de meilleures performances, un bon support pour de nombreux IDE et permet d'utiliser de nombreux dépôts, dont ceux de Maven.

## C Organisation du projet

---

### i Git et branches

On utilise donc Git comme logiciel de gestion de versions, ce qui nous permet de travailler en parallèle sur les mêmes fichiers et d'effectuer chacune les modifications qui nous concernent sans gêner le travail de l'autre.

#### Branches

Afin d'utiliser au mieux Git, nous avons fait le choix de créer deux branches "principales". Il s'agit donc de *master* et de *develop*.

La branche *master* correspond à la version en production de nos applications. Ainsi, on ne travaillera jamais sur cette branche. Elle ne nous servira donc qu'à récupérer l'application dans un état stable afin d'y mettre les différentes applications en production.

La branche *develop*, quant à elle, est donc la branche à partir de laquelle nous travaillons. C'est à partir de cette dernière que nous créerons les différentes branches pour le développement de nos fonctionnalités. Ne sont poussées sur celle-ci que les nouvelles fonctionnalités opérationnelles des applications. C'est donc la version en cours de développement.

Les branches créées à partir de *develop* sont donc les branches correspondant aux fonctionnalités développées, elles commencent toutes par *features/XXX* (correspondant à la modification). Par exemple, pour la gestion des clients, on crera une branche *features/gestionClients*.

## Nomenclature

Pour une meilleure homogénéisation de la gestion des versions, on choisit d'établir et d'utiliser une nomenclature pour les messages de commit. Cette nomenclature est consultable dans le fichier *CONVENTIONS.md*.

On y retrouve notamment les commits pour l'ajout de fonctionnalités sous le nom de *feat*, pour les corrections de bug *fix*, pour la documentation *doc*, ... Ce qui donne des messages comme celui-ci : *feat : Ajout de la modification d'un client*.

## ii Les différents dossiers

### Doc

On a fait le choix de rédiger la documentation du projet avec Latex car c'est un système de création de documents opensource. De plus, les fichiers Latex sont facilement gérés par Git contrairement aux fichiers Word. On peut ainsi manipuler aisément les différentes versions des documents. Il permet également une compilation directe au format pdf.

### SQL

Les fichiers SQL (situés dans le dossier */sql*) sont donc ceux qui vont créer la base de données. Il suffit d'importer les scripts dans l'ordre afin de la restituer. On nomme les fichiers précédés d'un numéro (ordre d'importation) suivi de l'intitulé de la modification. Les tables de la BDD sont en majuscules séparées par des underscore si besoin tandis que les champs respectent la norme CamelCase.

### Sources

Dans le dossier source, on retrouve donc deux sous-dossiers : un sous-dossier ionic et un Spring. Le sous-dossier ionic, comme son nom le suggère, correspond au code propre à l'application mobile (développée avec Ionic). On retrouvera donc à l'intérieur, des fichiers json, html, service.ts. On reviendra plus en détail sur cette partie par la suite. Le sous-dossier Spring, correspondant à SpringBoot, contient, quant à lui, les dossiers core, gradle, rest et webApp. Ces derniers comprennent notamment et respectivement les fichiers communs aux deux applications (bdd, service...), les fichiers de configuration, le code de l'API ainsi que le code de l'application web (controller, routes, formulaires, ...).

## iii Trello

Le planning, la répartition des tâches et le fonctionnement du projet sont visibles sur le [Trello](#). Pensez à consulter les cartes archivées pour voir le travail effectué. En effet, les modifications effectuées sont archivées afin de ne pas les conserver dans les activités à faire ou en cours.

## D Interactions entre les différentes parties du projet

### i Les différentes parties

Le projet Équida est composé de 2 applications. Une application web, qui est également l'application principale et une application mobile qui est à l'usage principal des clients. Les 2 applications s'appuient sur la même base de données. L'application web y est directement connectée. L'application mobile, elle, passe par une API. En effet, si celle-ci se connectait directement à la base de données, comme c'est le cas pour l'application web, une personne mal intentionnée serait en mesure de décompiler l'application mobile et ainsi obtenir les identifiants de la base de données. L'utilisation de cette API empêche donc, notamment, ce problème de sécurité.

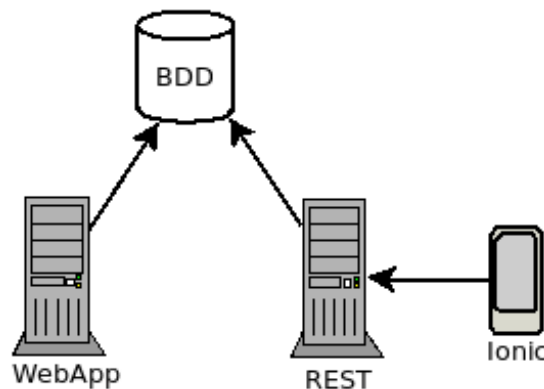


FIGURE 1.1 – La connexion à la BDD selon le projet

L'API ainsi que l'application web utilisent le framework Spring Boot. Ces 2 applications constituent donc 2 projets différents, *webApp* pour la partie web et *rest* pour l'API. Ces dernières nécessitant un code identique pour les Services, les Entity et les Repository, nous avons donc fait le choix de créer un projet commun *core*. On y retrouve donc tout le code commun aux 2 autres projets (cités précédemment mais également les exceptions ou certains outils).

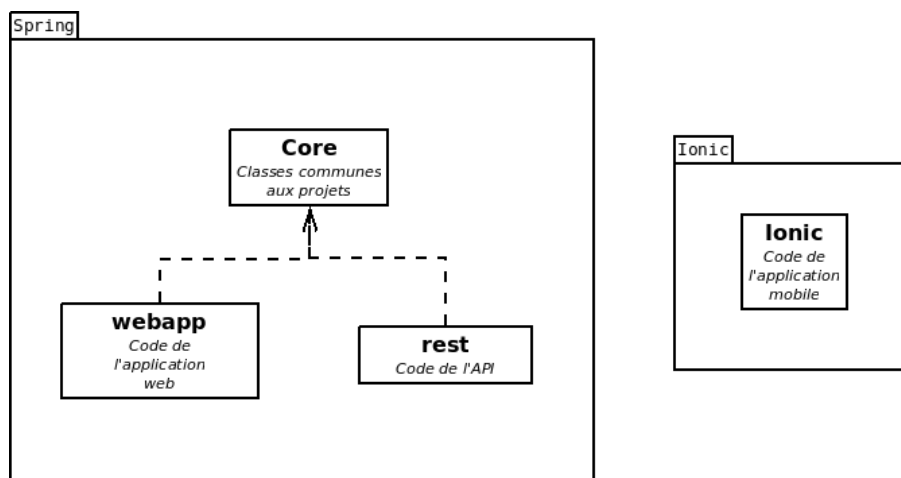


FIGURE 1.2 – Les dépendances entre les projets

## ii Configuration Gradle

Pour gérer correctement les différents projets basés sur Spring, leurs dépendances ainsi que leurs configurations, nous avons donc utilisé Gradle. Dans le dossier *src/Spring* on retrouve le *build.gradle* qui se charge de configurer la totalité du projet. On peut observer la configuration suivante pour tous.

```
allprojects {
    apply plugin : 'java'
    apply plugin : 'io.spring.dependency-management'
    apply plugin : 'org.springframework.boot'

    ext {
        springBootVersion = '2.1.3.RELEASE'
    }

    repositories {
        mavenCentral()
        jcenter()
        maven {
            url 'https://plugins.gradle.org/m2/'
        }
    }

    dependencies {
        implementation 'org.springframework.boot:spring-boot-starter'
        implementation 'org.springframework.boot:spring-boot-starter-web'
        implementation 'org.springframework.boot:spring-boot-starter-actuator'
        implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
        implementation 'org.springframework.boot:spring-boot-starter-security'

        testImplementation 'org.springframework.boot:spring-boot-starter-test'
    }
}
```

**FIGURE 1.3** – Configuration Gradle commune à tous les projets

On précise donc la version de Spring à utiliser, en plus des dépendances communes à tous les projets (spring-boot-starter-web, spring-boot-starter-data-jpa, ...).

Par la suite, on définit les dépendances uniques à chaque projet.

```
project(':core') {
    jar {
        enabled = true
    }

    dependencies {
        implementation 'com.h2database:h2'
        implementation 'mysql:mysql-connector-java'
    }
}

project(':rest') {
    dependencies {
        implementation project(':core')
    }
}

project(':webApp') {
    dependencies {
        implementation project(':core')

        implementation 'org.springframework.boot:spring-boot-starter-freemarker'
    }
}
```

**FIGURE 1.4** – Configuration Gradle propre à chaque projet

De même, concernant le projet core, on active uniquement la compilation en jar (comme une librairie) et non pas en jar bootable (comme c'est le cas lorsque l'on utilise Spring Boot).

D'autres scripts "build.gradle" se trouvent dans chaque dossier du projet, cependant, ceux-ci ne configurent que le nom du projet à l'issue du build, la version du JDK utilisée ainsi que le package de base du projet.



## 2 Core

### A Organisation des packages

---

L'organisation des packages est la suivante :

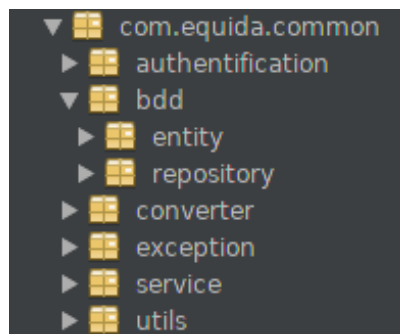


FIGURE 2.1 – Packages de Core

**authentication** : Contient toutes les classes relatives à l'authentification

**bdd** : Contient toutes les classes relatives à la base de données

**entity** : Contient toutes les classes Métiers

**repository** : Contient toutes les classes qui héritent de `CrudRepository`

**converter** : Contient toutes les classes qui héritent de `AttributeConverter`

**exception** : Contient toutes les Exceptions

**service** : Contient toutes les classes Services

**utils** : Contient quelques classes utiles (`Sha256PasswordEncoder`, `DateUtils`, ...)

### B Exemple d'Entity

---

Une entité est la correspondance de la BDD dans une classe Java.

Ainsi, on annote la classe de la mention `@Entity` et on précise avec `@Table` le nom de la table.

Ensuite, on crée les variables annotées avec `@Column` correspondant aux différents champs de la table.

Pour l'id on précisera, en plus du fait qu'il s'agit d'un id avec `@Id`, qu'il est auto-généré avec `@GeneratedValue`.

Pour les relations `ManyToOne`, `OneToMany`, ect. On utilisera `@OneToMany` (ou autre en fonction de la relation)

Pour finir, on implémente le(s) constructeur(s) ainsi que le(s) getter et setter.

```
@Entity
@Table(name = Role.TABLE)
public class Role {

    public static final String TABLE = "ROLE";

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "ID")
    private Long id;

    @Column(name = "LIBELLE")
    private String libelle;

    @Column(name = "DELETED")
    private Boolean deleted;

    @OneToMany(mappedBy = "role", cascade = CascadeType.ALL)
    private List<Compte> compte;

    public Role(Long i) {
        this.id = i;
    }

    public Role() {
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }
}
```

**FIGURE 2.2** – Exemple extrait du code de l'entity Role

## C Exemple de Repository

---

Le repository se charge des requêtes à la BDD pour l'entité correspondante. Ainsi, on annote la classe de la mention @Repository.

Ensuite, on implémente les requêtes sql souhaitées dans les @Query. Ces requêtes seront appelées dans le service de l'entité.

```
@Repository
public interface RoleRepository extends CrudRepository<Role, Long> {

    @Query(value = "SELECT r FROM Role r WHERE r.deleted=0")
    public List<Role> findAll();

    @Query(value = "SELECT r FROM Role r WHERE r.deleted=0")
    public List<Role> findAll(PageRequest pageRequest);

    @Query(value = "SELECT r FROM Role r WHERE r.id = ?1 AND r.deleted=0")
    public Optional<Role> findById(Long idRole);

}
```

**FIGURE 2.3** – Exemple extrait du code du repository Role

## D Exemple de Service

---

Le service sert d'interface entre le repository et le controller de l'entité qui permettra, en plus, une meilleure gestion des contrôles et des erreurs. Ainsi, on annote la classe de la mention @Service.

Ensuite on déclare le repository concerné, et on implémente les méthodes du service. Elles retourneront les valeurs des méthodes du repository.

```

@Service
public class RoleService {

    @Autowired
    private RoleRepository roleRepository;

    public List<Role> getAll() {
        return roleRepository.findAll();
    }

    public List<Role> getAll(PageRequest pageRequest) {
        return roleRepository.findAll(pageRequest);
    }

    public Role getById(Long idRole) throws NotFoundException {
        Optional<Role> role = roleRepository.findById(idRole);

        if(!role.isPresent()) {
            throw new NotFoundException("L'id du role spécifié n'existe pas.");
        }

        return role.get();
    }
}

```

**FIGURE 2.4** – Exemple extrait du code du service Role

## E Exemple d'exception (NotFoundException)

Le module Core permet de fournir certaines exceptions. Actuellement elles sont au nombre de 4.

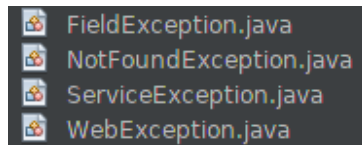


FIGURE 2.5 – Les différentes exceptions

**FieldException** : Permet de signaler une erreur sur un champs d'un formulaire alors que l'information saisie est valide d'un point de vue HTML. Par exemple, si un SIRE saisi n'existe pas dans la base de données ou, autre exemple, un login saisi existe déjà en base de données, cette erreur est déclenchée.

**NotFoundException** : Permet, notamment, de signaler que l'enregistrement demandé n'existe pas dans la base de données ou bien que l'utilisateur n'est pas autorisé à le voir, comme c'est le cas avec un cheval qui appartient à un autre client, par exemple.

**ServiceException** : Permet de signaler une erreur dans le comportement du Service. Elles sont surtout utilisées pour signaler qu'un champs requis vaut null.

**WebException** : Permet d'encapsuler une exception pour ne pas gêner l'affichage utilisateur. Lors d'un ServiceException, par exemple, l'envoi d'un WebException permet d'afficher une page d'erreur personnalisée.

Le code des exceptions est plutôt simple et court. Voici, par exemple, le code pour NotFoundException.

```
@ResponseStatus(value=HttpStatus.NOT_FOUND)
public class NotFoundException extends Exception {

    public NotFoundException() {
        super("La page demandée n'existe pas.");
    }

    public NotFoundException(String msg) {
        super(msg);
    }
}
```

FIGURE 2.6 – Code de NotFoundException

On définit un simple message d'erreur et on change le code de réponse HTTP, grâce à l'annotation `@ResponseStatus`, afin qu'il renvoie le code 404 et affiche la page adéquate.

## F Authentification

---

Ce package permet de gérer les différentes classes communes à l'authentification de l'utilisateur pour les modules Rest et WebApp. On y retrouve 2 classes :

**AuthenticationService** : Cette classe implémente l'interface *UserDetailsService*. Elle permet, pour un login donné, d'aller récupérer l'utilisateur correspondant dans la base de données et retourne un objet de type *AuthenticatedUser*.

**AuthenticatedUser** : Représente l'utilisateur actuellement connecté. Cette classe implémente l'interface *UserDetails*. On y retient notamment la classe *Compte* qui est la classe métier de la table COMPTE dans la base de données. Grâce à cet attribut, on pourra facilement obtenir les informations sur l'utilisateur connecté, par le biais de la méthode *Utilisateur getUtilisateur()*.

L'authentification sera alors gérée de manière différente selon le module. Dans le cas du module WebApp, il s'agira d'une connexion par le biais d'une page web tandis que pour l'api Rest on utilisera [Basic Authentication](#).

## G Utils

---

### i DateUtils

Cette classe permet de gérer certaines fonctionnalités au niveau des dates. On peut par exemple citer la méthode *boolean isBetween(Date, Date, Date)* qui permet de savoir si la dernière Date est comprise entre les 2 premières. Cette classe permet également d'afficher une date fournie en paramètre au format "jour/mois/année" grâce à la méthode *String format(Date)*.

### ii Sha256PasswordEncoder

Cette classe permet de gérer le hachage des mots de passe. Elle implémente l'interface *PasswordEncoder*, qui fournit une méthode publique *String encode(CharSequence)*. Cette méthode retourne en String le hachage, en sha256 dans notre cas, de la chaîne de caractères fournie en paramètre. Cette classe est transmise à Spring Security afin de vérifier les informations saisies par l'utilisateur lors d'une connexion. Elle est également utilisée manuellement lors de l'enregistrement d'un nouveau client.

## 3 Application web

### A Organisation des packages

---

#### i Packages

L'organisation des packages se présente comme suit :

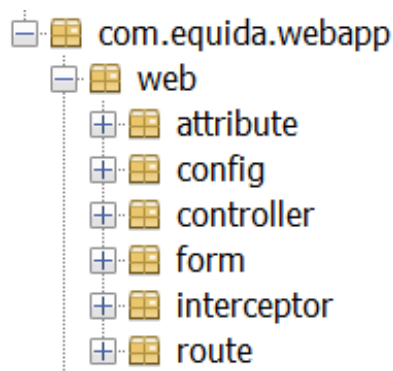


FIGURE 3.1 – Packages de WebApp

**attribute** : Contient la classe `InputOutputAttribute` qui gère toutes les constantes dont on pourrait avoir besoin

**config** : Contient les classes relatives à la configuration et la sécurité de l'application

**controller** : Contient tous les contrôleurs qui héritent de la classe `AbstractWebController`

**form** : Contient tous les formulaires qui héritent de la classe `IForm`

**interceptor** : Contient la classe `UserInterceptor` qui hérite de `HandlerInterceptorAdapter` (voir [Interceptor](#) et/ou [là](#))

**route** : Contient toutes les routes qui héritent de `IRoute`

## ii Resources

L'organisation des ressources se présente comme suit :

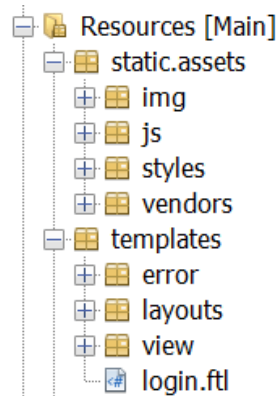


FIGURE 3.2 – Ressources de WebApp

**static.assets** : Contient toutes les ressources statiques qui ne nécessitent aucune compilation

**img** : Contient toutes les images de l'application

**js** : Contient tous les fichiers JavaScript notamment celui pour la gestion des classements à une course d'un cheval et celui pour la page d'accueil avec son carrousel et son menu

**styles** : Contient le fichier css de base

**vendors** : Contient toutes les dépendances externes du projet soit : Materialize (bibliothèque qui gère le design des vues de l'application), jQuery, Google (police de caractères)

**templates** : Contient tous les fichiers Freemarker

**error** : Contient les fichiers ftl pour les erreurs 403, 404, 500

**layouts** : Contient les fichiers ftl communs à toutes les pages de l'application

**view** : Contient toutes les vues de l'application (lister, consulter, form)

**login.ftl** : Fichier utilisé par SpringSecurity pour la page d'authentification



## B Configuration de l'application

---

### i application.properties

L'application web possède un fichier *application.properties* qui permet de configurer certaines parties de l'application.

```
server.port = 1515
server.servlet.session.cookie.secure=false

# JPA specific configs
spring.jpa.hibernate.ddl-auto=none
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLInnoDBDialect
spring.jpa.properties.hibernate.enable_lazy_load_no_trans=true
spring.jpa.properties.hibernate.id.new_generator_mappings=false
spring.jpa.hibernate.naming.physical-strategy=org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl
spring.datasource.url=jdbc:mysql://localhost/equida?serverTimezone=Europe/Paris
spring.datasource.username=USR_EQUIDA
spring.datasource.password=mpEQUIDA
spring.datasource.pool-size=30
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

**FIGURE 3.3** – Configuration par application.properties

Ce fichier configure les informations relatives à l'application dans sa globalité, comme le port à utiliser, la configuration pour connexion avec la base de données (nom utilisateur, mot de passe, ip, ...) ou encore la configuration du moteur de template, FreeMarker en l'occurrence.

## ii Configuration par le code

La configuration de Spring Security est directement faite dans le code source grâce à l'utilisation de l'annotation `@Configuration`.

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private AuthenticationService authenticationService;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests().anyRequest().permitAll()
            .and().formLogin()
                .loginPage("/login")
                .permitAll()
            .and().logout()
                .permitAll()
            .and().csrf().disable();
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.authenticationProvider(authenticationProvider());
    }

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

    @Bean
    public PasswordEncoder getPasswordEncoder() {
        return new Sha256PasswordEncoder();
    }
}
```

FIGURE 3.4 – Configuration de Spring Security

La méthode `void configure(HttpSecurity http)` permet de configurer l'accès aux différentes pages et de changer la configuration des requêtes HTTP. Ainsi, on autorise la connexion sur toutes les pages web, d'autant plus sur login et logout. De cette manière, le jour où l'on décidera de changer cette configuration, le code pour autoriser l'accès à `/logout` et `/login` sera déjà présent et empêchera un potentiel oubli.

Les méthodes suivantes sont relatives à l'authentification. La méthode `void configure(AuthenticationManagerBuilder)` permet de changer la méthode d'authentification utilisée par Spring Security. Elle fait appel à `DaoAuthenticationProvider authenticationProvider()` qui retourne le DAO à utiliser. On doit donc fournir le `UserDetailsService` du module core, c'est à dire `AuthenticationService` (cf : [Authentification](#)), et le `PasswordEncoder` approprié. De ce fait on utilise donc une nouvelle instance de `Sha256PasswordEncoder` (cf : [Sha256PasswordEncoder](#)).

## C Fichiers ressources

### i FreeMarker

FreeMarker est un moteur de template basé sur Java qui est à l'origine de la génération de pages web dynamiques dans une architecture logicielle.

Comme mentionné auparavant, tous les fichiers de FreeMarker sont contenus dans le dossier */resources/templates*. Il est donc possible d'utiliser les directives propres à ce moteur de template.

#### Page de base

La page *base.ftl* correspond à la page type de l'application. On y retrouve ainsi ce qui sera inclus sur toutes les pages de l'application.

Ainsi, *base.ftl* contient le header de la page avec l'inclusion de la feuille de style ; sa partie body contient, elle, le contenu du fichier *nav.ftl* correspondant au menu, ainsi que le footer et les fichiers de scripts nécessaires.

Les macros permettent le chargement du contenu aux endroits prévus à cet effet. Par exemple, la directive `<@content/>`, présente dans le body de *base.ftl*, chargera le code dans la macro *content* du fichier *x.ftl* à cet endroit.

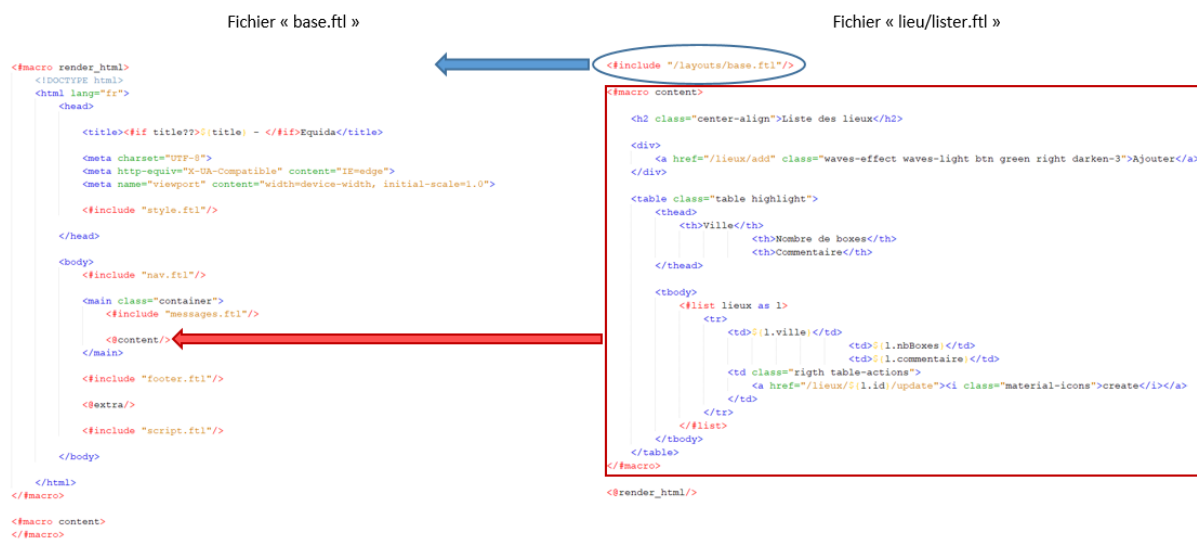
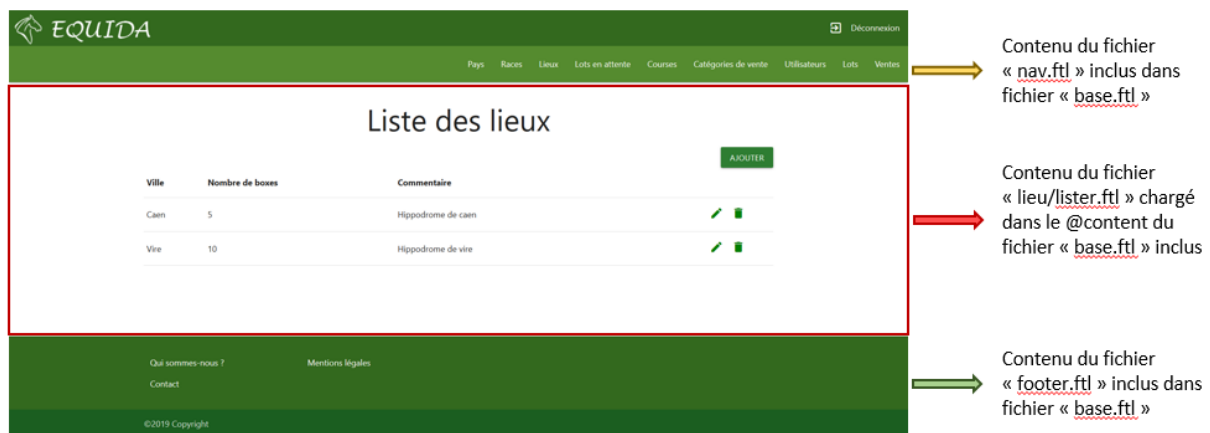


FIGURE 3.5 – Exemple du fonctionnement de la page *base.ftl* avec la page *lieux/liste.ftl*



**FIGURE 3.6** – Rendu lors du chargement de la vue qui liste les lieux (correspondant à la page lieux/lister.ftl précédente)

## Page d'erreur

Les pages d'erreur sont chargées automatiquement par Spring et contiennent des messages explicites. Nous avons gérés les erreurs 403 (permissions non autorisées), 404 (page inexistante) et 500 (exception lors de l'exécution du code). Elles reprennent, elles aussi, le design de base de l'application (base.ftl).

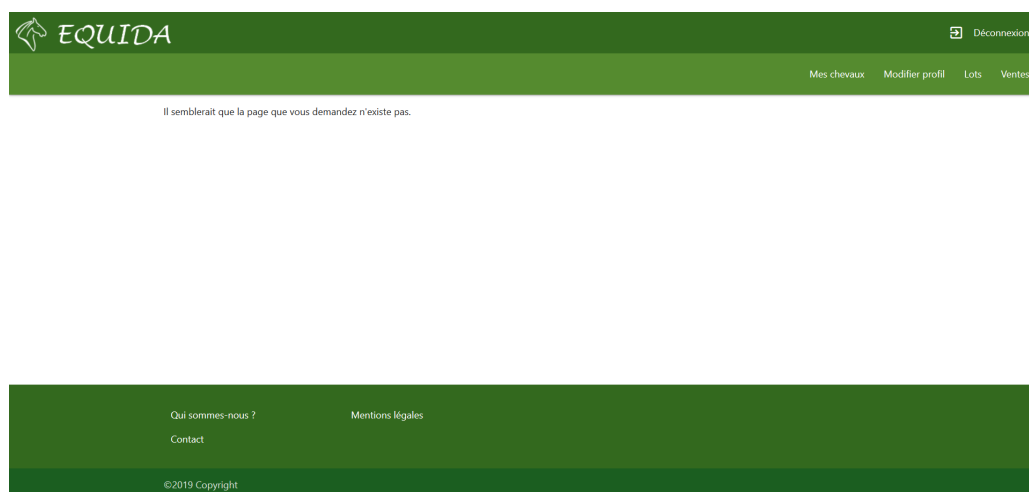
```
<#include "/layouts/base.ftl"/>

<#global title="404">

<#macro content>
    <p>Il semblerait que la page que vous demandez n'existe pas.</p>
</#macro>

<@render_html/>
```

**FIGURE 3.7** – Code de l'erreur 404



**FIGURE 3.8** – Rendu lors d'une tentative de chargement d'une page inexistante

## Fichiers à inclure

Le dossier */view/include* contient les vues communes aux différentes pages. On peut les inclure en utilisant les directives `<#include />`. On retrouve par exemple le fichier *lotLister* qui permet d’afficher les cartes pour les différents lots.

```
<div class="row row-eq-height">
  <div class="col s12">
    <#list lots as lot>
      <div class="col s12 m4 l3">
        <div class="card">
          <div class="card-image">
            
            <span class="card-title">${lot.cheval.nom}</span>
          </div>

          <div class="card-content">
            <p>Race : ${lot.cheval.raceCheval.libelle}</p>
            <p>Sexe : ${lot.cheval.sexe}</p>
            <p>Prix de départ : ${lot.prixDepart}€</p>
          </div>

          <div class="card-action">
            <a href="/chevaux/${lot.cheval.id}">Voir plus</a>
          </div>
        </div>
      </div>
    </#list>
  </div>
</div>
```

FIGURE 3.9 – Extrait de code du fichier lotLister

Le code reste celui de n’importe quelle autre vue et ne comporte aucune spécificité.

## Exemple de page web

Ainsi, avec tous les éléments cités précédemment, on peut créer des vues telles que celle ci :

```
<#include "/layouts/base.ftl"/>
<#assign title=vente.nom>
<#macro content>
  <h2 class="center-align">${vente.nom}</h2>
  <#if user?? && user.hasRole("ADMIN")>
    <div>
      <a href="/ventes/${vente.id}/delete" class="waves-effect waves-light btn red right darken-3">Supprimer</a>
      <#if now < vente.dateVente >
        <a href="/ventes/${vente.id}/update" class="waves-effect waves-light btn green right darken-3">Modifier</a>
      </#if>
    </div>
  </#if>
  <div class="row">
    <p>Catégorie : ${vente.categVente.libelle}</p>
    <p>Lieu : ${vente.lieu.ville}</p>
    <p>Date vente : ${vente.dateVente?string["dd/MM/yyyy"]}</p>
    <#if isInscriptionOuverte>
      <p>Les inscriptions sont ouvertes jusqu'au ${vente.dateFin?string["dd/MM/yyyy"]}</p>
    </#if>
    <#if !isInscriptionOuverte>
      <p>Les inscriptions sont uniquement ouvertes du ${vente.dateDebut?string["dd/MM/yyyy"]} au ${vente.dateFin?string["dd/MM/yyyy"]}</p>
    </#if>
  </div>
</macro>
```

FIGURE 3.10 – Code de la vue pour la consultation d’une vente

On retrouve donc, l’inclure de */layouts/base.ftl* pour reprendre le template de base, un changement de valeur concernant le titre, l’utilisation de la macro content, ...

## D Gestion de l'authentification

### i Gestion template et controller

Un contrôleur existe afin de gérer l'affichage d'un template FreeMarker concernant la page de connexion à l'application. Ce contrôleur se charge uniquement de l'affichage de l'information. En effet, le traitement des identifiants est fait par Spring Security (comme mentionné dans [Configuration par le code](#)).

### ii Interceptor

Afin de faciliter la gestion de l'utilisateur actuellement connecté dans la vue ou les contrôleurs, une classe *UserInterceptor* permet de fournir automatiquement l'instance de *AuthenticatedUser* à la vue ou au contrôleur.

```
@Component
public class UserInterceptor extends HandlerInterceptorAdapter {

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) {
        request.setAttribute(InputOutputAttribute.USER, getAuthenticatedUser());
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest hsr, HttpServletResponse hsr1, Object o, ModelAndView mav) throws Exception {
        AuthenticatedUser user = getAuthenticatedUser();

        if(user != null && mav != null)
            mav.addObject(InputOutputAttribute.USER, user);
    }

    private AuthenticatedUser getAuthenticatedUser() {
        Authentication authentication = SecurityContextHolder.getContext().getAuthentication();

        if(authentication instanceof UsernamePasswordAuthenticationToken)
            return (authentication.getPrincipal() instanceof AuthenticatedUser) ? (AuthenticatedUser) authentication.getPrincipal() : null;

        return null;
    }
}
```

FIGURE 3.11 – Code de UserInterceptor

On peut alors avoir accès à la variable *user* dans la vue comme n'importe quelle variable fournie par le contrôleur ou dans le contrôleur en utilisant le paramètre suivant dans une méthode d'un contrôleur `@RequestAttribute(name = InputOutputAttribute.USER, required = false) AuthenticatedUser user`.

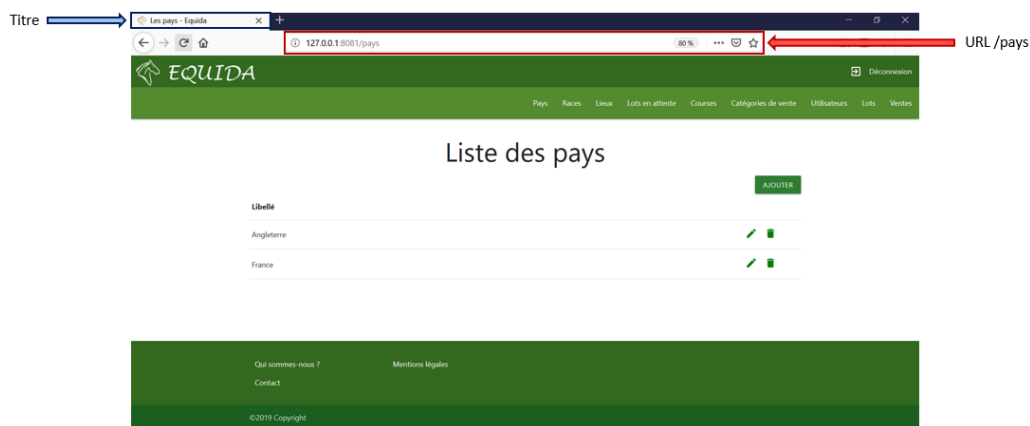
## E Exemple Route

L'interface *IRoute* décrit les méthodes qui doivent être implémentées par les classes filles. Ainsi chaque fichier route contiendra une méthode *getUri()*, une méthode *getView()* et *getTitle()* qui retourneront respectivement l'URL, la vue et le titre à utiliser dans la page concernée (pour la route correspondante).

Par exemple, pour *PaysRoute*, qui est donc la route principale selon notre nomenclature, l'URL correspond à */pays*, c'est à cette url là, qu'on chargera la vue *pays/lister* avec le titre "Les pays".

Interface	Classe fille
<pre>public interface IRoute {      public String getUri();      public String getTitle();      public String getView();  }</pre>	<pre>public class PaysRoute implements IRoute{      public static final String RAW_URI = "/pays";      @Override     public String getUri() {         return RAW_URI;     }      @Override     public String getView() {         return "view/pays/lister";     }      @Override     public String getTitle() {         return "Les pays";     }  }</pre>

**FIGURE 3.12** – Code de l’interface et utilisation par la classe fille PaysRoute



**FIGURE 3.13** – Rendu obtenu avec la vue pays/lister

## F Exemple Form

La classe mère *IForm* est une classe abstraite qui utilise la généréité ce qui nous permettra d’adapter les méthodes en fonction de l’entité x pour laquelle le formulaire est fait. L’héritage nous permet donc d’utiliser les variables et méthodes déclarées dans le formulaire neutre xForm.

Par exemple, prenons l’entité *Lieu*. On crée le formulaire "neutre" *LieuxForm* qui héritera de *IForm* et qui permettra de définir les éléments communs aux formulaire d’ajout et de modification.

On fera donc hériter de ce formulaire "neutre" *LieuxAddForm* et *LieuxUpdateForm* et on passera, dans un premier cas, le la variable *isCreation* à true et dans le second, à false.



FIGURE 3.14 – Exemple implémentation interface IForm avec LieuForm

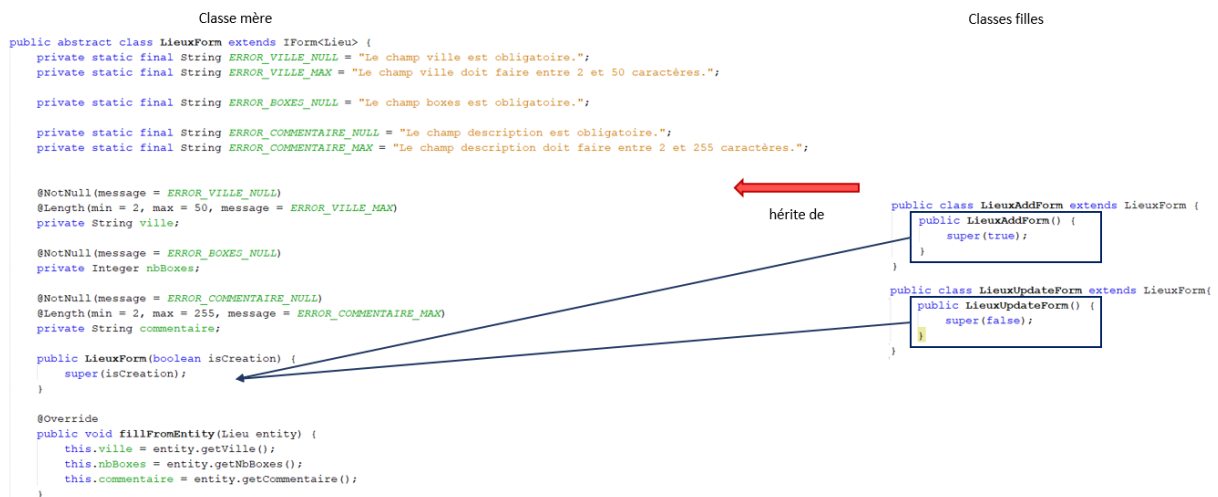


FIGURE 3.15 – Exemple avec LieuUpdateForm et LieuAddForm



## G Classe InputOutputAttribute

Cette classe permet la définition de constantes qui seront utilisées au travers de l'application.

```
public class InputOutputAttribute {  
  
    public static final String CATEG_VENTE = "categVente";  
  
    public static final String ERROR_LIST = "errors";  
    public static final String EXCEPTION = "exception";  
  
    public static final String FORM = "form";  
  
    public static final String IS_INSCRIPTION_OUVERTE = "isInscriptionOuverte";  
  
    public static final String LISTE_CATEG_VENTES = "categVentes";  
    public static final String LISTE_CHEVAUX = "chevaux";  
    public static final String LISTE_CLIENTS = "clients";  
    public static final String LISTE_COURSES = "courses";  
    public static final String LISTE_ENCHERES = "encheres";  
    public static final String LISTE_LIEUX = "lieux";  
    public static final String LISTE_LOTS = "lots";  
    public static final String LISTE_PARTICIPER = "participations";  
    public static final String LISTE_PAYS = "pays";  
    public static final String LISTE_RACES = "races";  
    public static final String LISTE_ROLES = "roles";  
    public static final String LISTE_UTILISATEURS = "utilisateurs";  
    public static final String LISTE_VENTES = "ventes";  
  
    public static final String MESSAGES_LIST = "messages";  
  
    public static final String TITLE = "title";  
  
    public static final String URL = "url";  
    public static final String USER = "user";  
  
    public static final String CHEVAL = "cheval";  
    public static final String LOT = "lot";  
    public static final String UTILISATEUR = "utilisateur";  
    public static final String VENTE = "vente";  
  
}
```

FIGURE 3.16 – Code de InputOutputAttribute

Ces constantes permettent de garder une unité dans la définition des noms et d'éviter d'éventuelles erreurs d'écriture. Ainsi, lorsque l'on doit fournir une clé de type String on pourra utiliser une des constantes de la classe. Elles sont donc très utilisées pour transmettre les informations du contrôleur vers la vue.

```
modelAndView.addObject(InputOutputAttribute.TITLE, route.getTitle());  
modelAndView.addObject(InputOutputAttribute.VENTE, vente);  
modelAndView.addObject(InputOutputAttribute.IS_INSCRIPTION_OUVERTE, DateUtils.isBefore(  
modelAndView.addObject(InputOutputAttribute.LISTE_LOTS, lots);
```

FIGURE 3.17 – Exemple d'utilisation de InputOutputAttribute

## H Les Contrôleurs

### i AbstractWebController

Cette classe est la classe mère de tous les contrôleurs. On y définit certaines méthodes, par exemple, la possibilité d'ajouter des messages d'erreurs, qui seront transférées à la vue.

```
protected void addError(String messageError, ModelAndView modelAndView) {
    Map<String, String> errorMap = (Map<String, String>) modelAndView.getModel().get(InputOutputAttribute.ERROR_LIST);
    if(errorMap == null)
        errorMap = new HashMap<>();

    errorMap.put(""+errorMap.size(), messageError);
    modelAndView.addObject(InputOutputAttribute.ERROR_LIST, errorMap);
}

protected void addMessage(String message, ModelAndView modelAndView) {
    ArrayList<String> messages = (ArrayList<String>) modelAndView.getModel().get(InputOutputAttribute.MESSAGES_LIST);
    if(messages == null)
        messages = new ArrayList<>();

    messages.add(message);
    modelAndView.addObject(InputOutputAttribute.MESSAGES_LIST, messages);
}
```

FIGURE 3.18 – Méthodes pour la gestion des messages

On y implémente aussi la gestion des exceptions, dans lesquelles, on passe les variables nécessaires au bon affichage de la vue.

```
@ExceptionHandler(Exception.class)
public ModelAndView handleException(HttpServletRequest request, Exception exception) {
    if(exception instanceof AccessDeniedException) {
        return handle403(request, exception);
    } else if(exception instanceof NotFoundException) {
        return handleNotFoundException(request, exception);
    } else if(exception instanceof WebException) {
        return handleWebException(request, exception);
    } else {
        return handleDefault(request, exception);
    }
}

public ModelAndView handle403(HttpServletRequest request, Exception exception) {
    ModelAndView modelAndView = new ModelAndView();
    modelAndView.addObject(InputOutputAttribute.EXCEPTION, exception);
    modelAndView.addObject(InputOutputAttribute.URL, request.getRequestURL());
    modelAndView.setViewName("error/403");

    return modelAndView;
}

public ModelAndView handleNotFoundException(HttpServletRequest request, Exception exception) {
    ModelAndView modelAndView = new ModelAndView();
    modelAndView.addObject(InputOutputAttribute.EXCEPTION, exception);
    modelAndView.addObject(InputOutputAttribute.URL, request.getRequestURL());
    modelAndView.setViewName("error/404");

    return modelAndView;
}
```

FIGURE 3.19 – Méthodes pour la gestion des exceptions

Elle comprend, en plus, 2 méthodes qui permettent de simplifier le code de gestion des formulaires, l'une afin de les créer et de les compléter si besoin, l'autre afin de faciliter la gestion des erreurs sur ceux-ci.

```
public <T> void registerForm(ModelAndView modelAndView, Model model, Class<? extends IForm<T>> formClass, T entity) {
    if (!model.containsKey(InputOutputAttribute.FORM)) {
        IForm form = null;
        try {
            form = formClass.newInstance();
        } catch (InstantiationException | IllegalAccessException ex) {
            ex.printStackTrace();
            return;
        }

        if (entity != null) {
            form.fillFromEntity(entity);
        }

        modelAndView.addObject(InputOutputAttribute.FORM, form);
    } else {
        modelAndView.addObject(InputOutputAttribute.FORM, model.asMap().get(InputOutputAttribute.FORM));
    }
}

public boolean checkForError(BindingResult bindingResult, RedirectAttributes attributes, IForm form) {
    if (bindingResult.hasErrors()) {
        bindErrors(bindingResult, attributes);
        attributes.addFlashAttribute(InputOutputAttribute.FORM, form);

        return true;
    }

    return false;
}
```

FIGURE 3.20 – Méthodes pour la gestion des formulaires

## ii Exemple Controller

Les différents contrôleurs créés pour les entités *x* héritent tous de la classe *AbstractWebController*. Les contrôleurs contiennent différentes méthodes associées aux méthodes GET, POST, PATCH et DELETE ainsi qu'à une route. Enfin, elles utilisent les services pour interagir avec la base de données. On y définit aussi les différentes autorisations avec l'annotation *@PreAuthorize* afin de savoir qui peut accéder à la page, et donc, exécuter la méthode.

Par exemple, dans *EncheresController*, la méthode *addGet* n'est possible que pour un utilisateur ayant le rôle 'ADMIN', elle est reliée à l'URL *EncheresAddRoute.RAW\_URI* (URL stockée dans la variable constante *RAW\_URI* de la classe *EncheresAddRoute*). Elle permet de récupérer les différentes variables nécessaires à l'affichage de la page, comme le formulaire d'ajout d'une enchère par exemple.

```
@PreAuthorize("hasRole('ROLE_ADMIN')")
@GetMapping(EncheresAddRoute.RAW_URI)
public ModelAndView addGet(Model model, @PathVariable(EncheresAddRoute.PARAM_ID_LOT) Long idLot) {
    IRoute route = new EncheresAddRoute(idLot);

    ModelAndView modelAndView = new ModelAndView(route.getView());
    modelAndView.addObject(InputOutputAttribute.TITLE, route.getTitle());
    modelAndView.addObject(InputOutputAttribute.LISTE_CLIENTS, clientService.getAll());
    modelAndView.addObject(InputOutputAttribute.LISTE_ENCHERES, enchereService.getAllByIdLot(idLot));
    registerForm(modelAndView, model, EncheresAddForm.class, null);

    return modelAndView;
}
```

FIGURE 3.21 – Exemple d'un contrôleur

## 4 Application mobile

### A API REST

#### i Organisation des packages

L'organisation des packages se déroule comme suit :

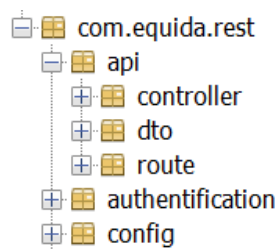


FIGURE 4.1 – Packages de l’api Rest

**api** : Contient les fichiers de l’api

**controller** : Contient tous les controleurs

**dto** : Contient tous les Dto

**route** : Contient toutes les routes

**authentification** : Contient la classe qui implémente BasicAuthenticationEntryPoint pour configurer Basic Authentication sur l’application

**config** : Contient les classes de configuration et de sécurisation de l’application

#### ii Configuration de l’application

##### application.properties

L’API Rest possède un fichier *application.properties* qui permet de configurer certaines parties de l’application.

```
server.port = 1515
server.servlet.session.cookie.secure=false

# JPA specific configs
spring.jpa.hibernate.ddl-auto=none
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLInnoDBDialect
spring.jpa.properties.hibernate.enable_lazy_load_no_trans=true
spring.jpa.properties.hibernate.id.new_generator_mappings=false
spring.jpa.hibernate.naming.physical-strategy=org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl
spring.datasource.url=jdbc:mysql://localhost/equida?serverTimezone=Europe/Paris
spring.datasource.username=USR_EQUIDA
spring.datasource.password=mpEQUIDA
spring.datasource.pool-size=30
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

FIGURE 4.2 – Configuration par application.properties

Ce fichier configure les informations relatives à l'application dans sa globalité, comme le port à utiliser, la configuration pour connexion avec la base de données (nom utilisateur, mot de passe, ip, ...).

### Configuration par le code

La configuration de Spring Security est directement faite dans le code source grâce à l'utilisation de l'annotation `@Configuration`.

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests().anyRequest().fullyAuthenticated()
        .and().httpBasic().authenticationEntryPoint(authEntryPoint)
        .and().cors()
        .and().csrf().disable();
}
```

FIGURE 4.3 – Configuration de Spring Security

Ici, on exige l'authentification sur toutes les requêtes faites à l'API. L'authentification doit être faite en utilisant *Basic Authentication* (voir [Système d'authentification sur l'API](#)). On y active les CORS et on désactive le csrf.

```
//https://stackoverflow.com/questions/36968963/how-to-configure-cors-in-a-spring-boot-spring-security-application
@Bean
CorsConfigurationSource corsConfigurationSource() {
    UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
    CorsConfiguration corsConfiguration = new CorsConfiguration();
    corsConfiguration.applyPermitDefaultValues();
    corsConfiguration.addAllowedMethod(HttpMethod.GET);
    corsConfiguration.addAllowedMethod(HttpMethod.POST);
    corsConfiguration.addAllowedMethod(HttpMethod.PATCH);
    corsConfiguration.addAllowedMethod(HttpMethod.DELETE);
    source.registerCorsConfiguration("/*", corsConfiguration);
    return source;
}
```

FIGURE 4.4 – Configuration des requêtes CORS

Afin de permettre l'utilisation des requêtes DELETE et PATCH sur l'API, il a fallu changer la configuration des CORS. Le choix a été fait d'autoriser ces requêtes pour toutes les URL afin de simplifier la chose. Pour un code plus propre, il aurait fallu autoriser uniquement celles dont chaque URL avaient besoin.

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.authenticationProvider(authenticationProvider());
}

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

@Bean
public PasswordEncoder getPasswordEncoder() {
    return new Sha256PasswordEncoder();
}
```

FIGURE 4.5 – Configuration de l'authentification

Enfin, comme pour WebApp, on définit le fonctionnement de l'authentification. Ainsi, vous pouvez retrouver les explications dans [Authentification](#).

### iii Système d'authentification sur l'API

L'API utilisant *Basic Authentication*, il est nécessaire de fournir à Spring Security une implémentation de ce modèle. C'est le rôle d'*AuthenticationEntryPointImpl* dont voici le code :

```
@Component
public class AuthenticationEntryPointImpl extends BasicAuthenticationEntryPoint {

    @Override
    public void commence(HttpServletRequest request, HttpServletResponse response, AuthenticationException authEx)
        throws IOException, ServletException {
        response.addHeader("WWW-Authenticate", "Basic realm=" + getRealmName());
        response.setStatus(HttpStatus.UNAUTHORIZED);
    }

    @Override
    public void afterPropertiesSet() throws Exception {
        setRealmName("Equida Api");
        super.afterPropertiesSet();
    }
}
```

FIGURE 4.6 – Code d'AuthenticationEntryPointImpl

Le code reste extrêmement simple ici car Spring Security nous offre une couche d'abstraction pour l'implémentation du modèle. Ainsi, le seul élément que l'on définit est le realm.

### iv Exemple Route

L'interface *IRoute* décrit la méthode qui doit être implémentée par les classes filles. Ainsi chaque fichier route contiendra une méthode *getUri()* qui retournera l'URL à utiliser dans la page concernée (pour la route correspondante).

Par exemple, pour *LotsApiRoute*, qui est donc la route principale de l'api pour la gestion des lots selon notre nomenclature, l'URL renvoyée est */api/lots*.

Interface		Classe fille
<pre>public interface IRoute {     public String getUri(); }</pre>	<div>← hérite de</div>	<pre>public class LotsApiRoute implements IRoute {     public static final String RAW_URI = "/api/lots";      @Override     public String getUri() {         return RAW_URI;     } }</pre>

FIGURE 4.7 – Interface IRoute et exemple avec LotsApiRoute

## v Les Dto

### L'interface IDto

Afin d'alléger les requêtes SQL à la base de données et la sortie en JSON des classes métiers, le choix à été fait d'utiliser les DTO. Aucun lien vers un autre objet n'est fait, seul son ID est laissé et il faudra ensuite utiliser l'URL adéquate afin de récupérer les informations si celles-ci nous intéressent.

Pour uniformiser nos DTO, une interface IDto a été faite.

```
public interface IDto<T, U> {  
    public static <T, U> U convertToDto(T entity) {  
        return null;  
    }  
  
    public T convertToEntity();  
}
```

FIGURE 4.8 – Code de l'interface IDto

Celle ci se base donc sur une généricité double, T étant notre classe métier provenant du package *com.equida.common.bdd.entity* et U étant la classe DTO correspondante. Cette interface définit 2 méthodes, *<T, U> U convertToDto(T entity)* permettant de convertir une instance de nos classes Métier en un DTO et *T convertToEntity()* qui permet de convertir notre DTO en une instance de nos classes Métier.

### Exemple de Dto

```
public class PaysDto implements IDto<Pays, PaysDto>{  
  
    private Long id;  
    private String libelle;  
    private Boolean deleted;  
  
    public static PaysDto convertToDto(Pays entity) {  
        PaysDto paysDto = new PaysDto();  
  
        paysDto.setId(entity.getId());  
        paysDto.setLibelle(entity.getLibelle());  
        paysDto.setDeleted(entity.getDeleted());  
  
        return paysDto;  
    }  
  
    @Override  
    public Pays convertToEntity() {  
        Pays pays = new Pays();  
  
        pays.setId(this.id);  
        pays.setLibelle(this.libelle);  
        pays.setDeleted(this.deleted);  
  
        return pays;  
    }  
  
    public Long getId() {  
        return id;  
    }  
}
```

FIGURE 4.9 – Exemple code de PaysDto

## vi Exemple Controller

Les différents contrôleurs sont des classes précédées de l'annotation `@RestController`. Les contrôleurs contiennent différentes méthodes associées aux méthodes GET, POST, PATCH et DELETE ainsi qu'à une route. Enfin elles utilisent les services pour interagir avec la base de données. On y définit aussi les différentes autorisations avec l'annotation `@PreAuthorize` afin de savoir qui peut accéder à la page, et donc, exécuter la méthode.

Par exemple, pour *LotDetailsRestController*, on implémente la méthode *getLot*, liée à la route *LotDetailsApiRoute.RAW\_URI*, qui prend en paramètre l'id du lot concerné. Elle va retourner (grâce à la méthode *getById* de *LotService*) le lot correspondant sous la forme d'un *LotDto*.

```
@RestController
public class LotDetailsRestController {

    @Autowired
    private LotService lotService;

    @GetMapping(LotDetailsApiRoute.RAW_URI)
    public LotDto getLot(@PathVariable(value = LotDetailsApiRoute.PARAM_ID_LOT) Long idLot) throws NotFoundException {
        Lot lot = lotService.getById(idLot);

        return LotDto.convertToDto(lot);
    }
}
```

FIGURE 4.10 – Exemple d'un contrôleur avec *LotDetailsRestController*

## B Ionic

### i Organisation des packages

L'organisation des packages se déroule comme suit :

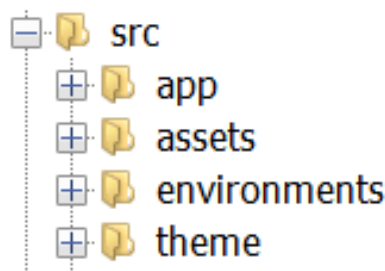


FIGURE 4.11 – Packages de ionic

**app** : Contient les fichiers de l'application

**assets** : Contient les images de l'application

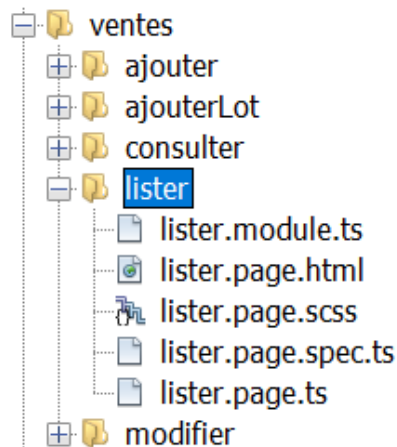
**environments** : Contient les différents "mode de fonctionnement" de l'application. On aura par exemple un environnement de développement (utilisé par défaut) ainsi qu'un autre pour faire fonctionner l'application en mode production. On pourra ainsi affiner le niveau de debug et de log pour voir, ou non, plus d'erreur et intégrer des outils pour simplifier le débogage de l'application.

**theme** : Contient le fichier `variables.scss` qui définit les couleurs utilisées par ionic



## ii Les pages

Les différentes pages générées (via la commande *ionic g*) suivent la même nomenclature : elles sont placées dans un dossier du nom de l'entité concernée suivit d'un verbe (souvent : lister, ajouter, consulter, modifier).



**FIGURE 4.12** – Résultat de la commande *ionic g* avec un nom de page correspondant à lister dans le dossier ventes

Dans chaque sous-dossier (un sous-dossier pour lister, un pour ajouter, ...) créée par la commande ionic, on retrouvera les mêmes types de fichiers.

**x.module.ts** : Exporte notre module pour être utilisable par ionic

**x.page.html** : Contient le code HTML de la page ainsi que les composants ionic

**x.page.scss** : Permet de modifier le style de la page

**x.page.spec.ts** : Permet d'effectuer des tests unitaires

**x.page.ts** : Contient toutes les méthodes à exécuter ainsi que l'initialisation de la page

On ne travaillera que sur les fichiers x.page.ts, x.page.html.

### Lister

La page *lister.page.html*, contient donc un header avec le titre de la page, ainsi qu'un content avec la liste et une boucle permettant l'affichage des ventes (*\*ngFor="let v of ventes"*) et la gestion d'un clic sur une vente (*routerLink="/ventes/v.id"*) qui renvoie donc sur la page de la vente en question.

On retrouve aussi un test sur le rôle de l'utilisateur (*\*ngIf="role == 'ADMIN'"*), si l'utilisateur a un rôle 'ADMIN' alors il verra un bouton flottant d'ajout. Ce bouton le renverra vers la page d'ajout d'une vente. Pour finir, on affiche plus de ventes sur la page lors d'un scroll de l'utilisateur si celui-ci atteint le bas de la page.

```

<ion-header>
  <ion-toolbar>
    <ion-title>Liste des ventes</ion-title>
  </ion-toolbar>
</ion-header>

<ion-content padding>
  <ion-list>
    <ion-item *ngFor="let v of ventes" routerLink="/ventes/{{v.id}}">
      <ion-label>
        <p>{{v.nom}}</p>
        <p>{{v.dateDebut}} - {{v.dateFin}}</p>
        <p>{{v.dateVente}}</p>
        <p>{{v.lieu.ville}} - {{v.lieu.commentaire}}</p>
        <p>{{v.categVente.libelle}}</p>
      </ion-label>
    </ion-item>
  </ion-list>

  <div *ngIf="role == 'ADMIN'">
    <ion-fab vertical="bottom" horizontal="end" slot="fixed">
      <ion-fab-button color="danger" routerLink="/ajouter/ventes" >
        <ion-icon name="add" ></ion-icon>
      </ion-fab-button>
    </ion-fab>
  </div>

  <ion-infinite-scroll threshold="100px" (ionInfinite)="loadData($event)">
    <ion-infinite-scroll-content
      loadingSpinner="bubbles"
      loadingText="Chargement...">
    </ion-infinite-scroll-content>
  </ion-infinite-scroll>
</ion-content>

```

FIGURE 4.13 – Code de /ventes/lister/lister.page.html

La page *lister.page.ts*, contient le constructeur, l'initialisation des variables, l'initialisation de la page avec la méthode *ngOnInit* qui appelle notamment la méthode *getVentes*. Cette méthode appelle la méthode *loadVentes* qui chargera donc les ventes, pour peu qu'il en reste à récupérer (et qui récupèrera le lieu et la catégorie de vente, par leur id respectif, afin de les afficher).

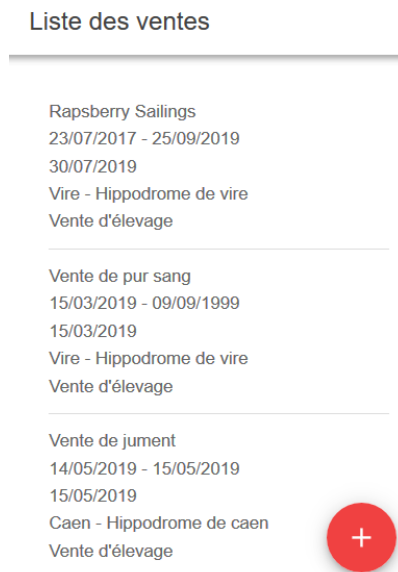
```

async loadVentes() {
  await this.api.getVentes(this.currentOffset)
    .then(async res => {
      console.log(res);

      if(res.length == 0) {
        this.shouldDisableInfiniteScroll = true;
      } else {
        for(let i = 0 ; i < res.length ; i++) {
          await this.api.getLieuById(res[i].idLieu).then(async l => {
            res[i].lieu = l;
          }, err => {
            console.log(err);
          });
          await this.api.getCategVenteById(res[i].idCategVente)
            .then(async cv => {
              res[i].categVente = cv;
            }, err => {
              console.log(err);
            });
          this.ventes.push(res[i]);
        }
      }
    }, err => {
      console.log(err);
    });
}

```

**FIGURE 4.14** – Code méthode loadVentes de /ventes/lister/lister.page.ts



**FIGURE 4.15** – Résultat de la page lister les ventes

## Consulter

La page *consulter.page.html*, contient donc un header avec le titre de la page; ainsi qu'un content avec un composant ionic card (purement visuel) dans lequel on retrouve des items correspondant aux différents champs d'une vente.

Puisque c'est une page de consultation, les champs affichent les valeurs contenues dans la base de données.

On trouve, là-aussi des boutons. Un pour proposer un cheval, seulement disponible pour un utilisateur classique ainsi qu'un pour supprimer, présent seulement si l'utilisateur connecté est un administrateur. Sur le même principe, on a aussi la liste des lots en vente (que l'on affiche pas ci-dessous dans le code mais qui est présent dans le fichier).

```
<ion-card-content>
  <ion-item>
    Dates des inscriptions : {{vente.dateDebut}} - {{vente.dateFin}}
  </ion-item>

  <ion-item>
    Date de la vente : {{vente.dateVente}}
  </ion-item>

  <ion-item>
    Lieu : {{vente.lieu.ville}} - {{vente.lieu.commentaire}}
  </ion-item>

  <ion-item>
    Catégorie de vente : {{vente.categVente.libelle}}
  </ion-item>

  <ion-button *ngIf="role == 'USER'" routerLink="/ajouterLot/ventes/{{vente.id}}" >
    <ion-icon slot="start" name="add"></ion-icon>
    Proposer un cheval
  </ion-button>

  <ion-button *ngIf="role == 'ADMIN'" routerLink="/modifier/ventes/{{vente.id}}">
    <ion-icon slot="start" name="create"></ion-icon>
    Modifier
  </ion-button>

  <ion-button *ngIf="role == 'ADMIN'" color="danger" (click)="deleteVente()">
    <ion-icon slot="start" name="trash"></ion-icon>
    Supprimer
  </ion-button>
</ion-card-content>
```

FIGURE 4.16 – Extrait du code de /ventes/consulter/consulter.page.html

La page *consulter.page.ts*, contient le constructeur, l'initialisation des variables, l'initialisation de la page avec la méthode *ngOnInit* qui appelle notamment les méthodes *getVenteById* et *getLotsByIdVente*. La méthode *getVenteById* se charge donc de récupérer la vente par l'id passé en paramètre de la route. Afin d'afficher le lieu et la catégorie de vente, elle utilise aussi les méthodes *getLieuById* et *getCategVenteById* de l'api qui renvoient le bon nom, libellé en fonction de l'id fourni par la vente affichée. On trouve aussi la méthode *deleteVente* appelée lors du clic sur le bouton supprimer. La vente dont l'id est passé en paramètre sera supprimée et l'utilisateur sera redirigé vers la page des ventes.

```

async getVenteById() {
  await this.api.getVenteById(this.route.snapshot.paramMap.get('id'))
    .then(async res => {
      await this.api.getLieuById(res.idLieu).then(async l => {
        res.lieu = l;
      }, err => {
        console.log(err);
      });
      await this.api.getCategVenteById(res.idCategVente).then(async cv => {
        res.categVente = cv;
      }, err => {
        console.log(err);
      });
      this.vente = res;
    }, err => {
      console.log(err);
    });
}

async deleteVente() {
  const loading = await this.loadingController.create({
    message: 'Envoi des informations'
  });
  await loading.present();
  await this.api.deleteVente(this.route.snapshot.paramMap.get('id'))
    .then(res => {
      console.log(res);
      loading.dismiss();
      this.navCtrl.navigateForward('/ventes');
    }, err => {
      console.log(err);
      loading.dismiss();
    });
}

```

FIGURE 4.17 – Extrait du code de /ventes/consulter/consulter.page.ts

Consultation vente


Rapsberry Sailings


Dates des inscriptions :  
23/07/2017 - 25/09/2019

Date de la vente :  
30/07/2019

Lieu : Vire - Hippodrome de  
vire

Catégorie de vente : Vente  
d'élevage

 MODIFIER

 SUPPRIMER

Les lots en vente :

Kanelle

FIGURE 4.18 – Résultat de la page de consultation d'une vente

## Ajouter

La page *ajouter.page.html*, contient donc un header avec le titre de la page ; ainsi qu'un content avec les différents items et input.

Dans les input et les select, on retrouve `[(ngModel)="nomChamp"]` qui fera donc le lien entre les données saisies dans ce champ et les variables associées.

On a aussi un bouton qui lorsqu'il est cliqué, appelle la méthode *addVente*.

```
<ion-content>
  <ion-item>
    <ion-label>Nom : </ion-label>
    <ion-input [(ngModel)]="nom"></ion-input>
  </ion-item>
  <ion-item>
    <ion-label>Date début : </ion-label>
    <ion-datetime [(ngModel)]="dateDebut" display-format="D MM YYYY"></ion-datetime>
  </ion-item>
  <ion-item>
    <ion-label>Date fin : </ion-label>
    <ion-datetime [(ngModel)]="dateFin" display-format="D MM YYYY"></ion-datetime>
  </ion-item>
  <ion-item>
    <ion-label>Date vente : </ion-label>
    <ion-datetime [(ngModel)]="dateVente" display-format="D MM YYYY"></ion-datetime>
  </ion-item>
  <ion-item>
    <ion-label>Catégorie de la vente :</ion-label>
    <ion-select [(ngModel)]="idCategVente" placeholder="Select One">
      <ion-select-option *ngFor="let cv of categVente" value="{{cv.id}}">{{cv.libelle}}</ion-select-option>
    </ion-select>
  </ion-item>
  <ion-item>
    <ion-label>Lieu : </ion-label>
    <ion-select [(ngModel)]="idLieu" placeholder="Select One">
      <ion-select-option *ngFor="let l of lieu" value="{{l.id}}">{{l.ville}}</ion-select-option>
    </ion-select>
  </ion-item>
  <ion-button (click)="addVente()">Enregistrer</ion-button>
</ion-content>
```

FIGURE 4.19 – Code de /ventes/ajouter/ajouter.page.html

La page *ajouter.page.ts*, contient le constructeur, l'initialisation des variables, l'initialisation de la page avec la méthode *ngOnInit* qui appelle notamment les méthodes *getCategVente* et *getLieux* de l'api. La méthode *addVente* se chargera donc de créer la vente grâce aux paramètres passés dans la méthode de l'api.

```
async addVente() {
  const loading = await this.loadingController.create({
    message: 'Envoie des informations'
  });
  await loading.present();

  await this.api.addVente(this.nom, this.dateDebut, this.dateFin, this.dateVente, this.idLieu, this.idCategVente)
    .then(res => {
      loading.dismiss();
      this.navCtrl.pop();
    },
    err => {
      console.log(err);
      loading.dismiss();
    });
}
```

FIGURE 4.20 – Extrait du code de /ventes/ajouter/ajouter.page.ts

Ajouter une vente

---

Nom :

---

Date début :

---

Date fin :

---

Date vente :

---

Catégorie de la vente : Select One ▼

---

Lieu : Select One ▼

---

ENREGISTRER

FIGURE 4.21 – Résultat de la page d’ajout d’une vente

## Modifier

La page *modifier.page.html*, est sensiblement la même que *ajouter.page.ts*. A la différence que le bouton, lorsqu’il est cliqué, appelle la méthode *updateVente*.

La page *modifier.page.ts*, contient le constructeur, l’initialisation des variables, l’initialisation de la page avec la méthode *ngOnInit* qui récupère les valeurs de la vente à modifier. La méthode *updateVente* se chargera donc de modifier la vente grâce aux paramètres passés dans la méthode de l’api.

```
async addVente() {  
    const loading = await this.loadingController.create({  
        message: 'Envoie des informations'  
    });  
    await loading.present();  
  
    await this.api.addVente(this.nom, this.dateDebut, this.dateFin, this.dateVente, this.idLieu, this.idCategVente)  
        .then(res => {  
            loading.dismiss();  
            this.navCtrl.pop();  
        },  
        err => {  
            console.log(err);  
            loading.dismiss();  
        });  
}
```

FIGURE 4.22 – Extrait du code de /ventes/modifier/modifier.page.ts

Modifier la vente

Nom : Rapsberry Sailings	
Date début :	23 07 2017
Date fin :	25 09 2019
Date vente :	30 07 2019
Catégorie de la ven...	Vente d'él... ▼
Lieu :	Vire ▼

ENREGISTRER

FIGURE 4.23 – Résultat de la page de modification d'une vente

### iii Rest Api

Pour faciliter l'exécution des différentes requêtes à l'API la classe *rest-api.service* a été conçue.

#### Authentification à l'Api

```
public saveCredentials(username : string, passwd : string, role: string, userId: string) {
    localStorage.setItem("username", username);
    localStorage.setItem("passwd", passwd);
    localStorage.setItem("role", role);
    localStorage.setItem("user-id", userId);
}

public removeCredentials() {
    localStorage.removeItem("username");
    localStorage.removeItem("passwd");
    localStorage.removeItem("role");
    localStorage.removeItem("user-id");
}

checkLogin(username : string, passwd : string): Promise<any> {
    const url = this.apiUrl+'/login';
    return this.http.get(url, this.getHttpOptions()).pipe(map(this.extractData)).toPromise();
}
```

FIGURE 4.24 – Code de la classe RestApiService

Ici, *saveCredentials* permet d'enregistrer les informations relatives à l'authentification du client dans le local storage. Il existe *removeCredentials* qui elle, à l'inverse de *saveCredentials*, supprime les informations relatives à l'utilisateur connecté. On a ainsi 2 méthodes qui permettent de gérer facilement la connexion et la déconnexion d'un utilisateur.

La méthode *checkLogin* permet d'appeler */api/login* et d'obtenir les informations relatives à l'utilisateur si les informations fournies sont correctes.



```

private getHttpOptions() {
    let username = localStorage.getItem("username");
    let passwd = localStorage.getItem("passwd");

    if(username == undefined || passwd == undefined) {
        this.navCtrl.navigateForward('/login');
        return;
    }

    let base64Auth = btoa(username+":"+passwd);
    const httpOptions = {
        headers: new HttpHeaders({'Content-Type': 'application/json',
            'Authorization' : 'Basic '+base64Auth
        }),
    };

    return httpOptions;
}

```

FIGURE 4.25 – Code de la classe RestApiService

Ces informations sont ensuite réutilisées pour permettre l'authentification à l'API. Pour cela, la méthode *getHttpOptions* permet d'obtenir l'entête à fournir pour chaque appel à l'API. Dans le cas où le mot de passe ou le login est null, on redirige l'utilisateur vers la page de connexion.

### Execution des différentes méthodes

Les appels à l'API utilisant les méthode HTTP GET, POST, PATCH et DELETE, une méthode existe, dans la classe *rest-api.service*, pour chacune de ces méthodes HTTP. Voici un exemple avec la méthode HTTP GET.

```

public execGetMethod(url: string): Promise<any> {
    return this.http.get(url, this.getHttpOptions()).pipe(
        map(this.extractData),
        catchError(async err => {
            if(err.status == 401) {
                this.removeCredentials();
                this.navCtrl.navigateForward('/login');
                return;
            }
            return this.handleError(err);
        })).toPromise();
}

public execPostMethod(url: string, data: any): Promise<any> {
    return this.http.post(url, data, this.getHttpOptions()).pipe(
        map(this.extractData),
        catchError(async err => {
            if(err.status == 401) {
                this.removeCredentials();
                this.navCtrl.navigateForward('/login');
                return;
            }
            return this.handleError(err);
        })).toPromise();
}

```

FIGURE 4.26 – Code de la méthode HTTP GET

On effectue, ici, l'appel à l'URL fourni en paramètre tout en récupérant les entêtes nécessaires pour l'authentification. Selon si l'appel a réussi ou non, on retourne le résultat ou on appelle *handleError* afin de générer un message d'erreur ainsi qu'une exception. Il est important de vérifier si le code d'erreur est 401 car si c'est le cas, l'utilisateur est, ou désactivé, ou alors a son mot de passe de changé. Si tel est le cas, on le déconnecte et on le redirige vers la page de connexion.

Les autres méthodes ne changent pas tellement si ce n'est que pour POST et PATCH on fournit en plus un objet "data" qui correspond aux informations à fournir à l'API mais aussi le nom des méthodes appelées sur l'objet HTTP CHANGE.

```
private async handleError(error: HttpResponse) {
  if (error.error instanceof ErrorEvent) {
    // A client-side or network error occurred. Handle it accordingly.
    console.error('An error occurred:', error.error.message);
  } else {
    // The backend returned an unsuccessful response code.
    // The response body may contain clues as to what went wrong,
    console.error(
      'Backend returned code '+error.status+', ' +
      'body was: '+error.error
    );

    const alert = await this.alertController.create({
      header: 'Oops...',
      subHeader: '',
      message: 'Une erreur s\'est produite. Merci de réessayer plus tard.',
      buttons: ['Valider']
    });

    await alert.present();
  }

  throw new Error(error.error.message);
}
```

**FIGURE 4.27** – Code de la méthode *handleError*

La méthode *handleError* affiche les informations à propos de l'erreur dans la console du navigateur et se charge également d'afficher une boîte de dialogue expliquant qu'une erreur est survenue. Dans le cadre du développement, cela permet également de rappeler que plus d'informations sont disponibles dans la console.

## Exemples

Ainsi, l'appel aux différents endpoints de l'API tient en quelques lignes et se retrouve extrêmement simplifié comme on peut le voir sur les méthodes suivantes :

```
addPays(libelle: string): Promise<any> {
    let data = {
        libelle : libelle
    };

    const url = this.apiUrl+'/pays';
    return this.execPostMethod(url, data);
}

deleteCheval(id: string): Promise<any> {
    const url = this.apiUrl+'/chevaux/'+id;
    return this.execDeleteMethod(url);
}

getCategVente(offset : number): Promise<any> {
    const url = this.apiUrl+'/categoriesVente?offset='+offset;
    return this.execGetMethod(url);
}
```

FIGURE 4.28 – Code des appels

Le fait de retourner une instance de *Promise* permet d'utiliser *await* sur la méthode et donc d'attendre la réponse de l'API avant de continuer l'exécution du code.

## iv Problèmes connus

Cette application mobile comporte des erreurs sur lesquelles nous n'avons pas pu, su intervenir.

En effet, après un ajout ou une modification (quelque soit l'entité concernée), lorsque l'utilisateur est redirigé vers la page principale (celle qui liste), les informations ne sont pas mises à jour car la page n'est pas actualisée.

Afin de palier à cette erreur, il est nécessaire d'actualiser "manuellement" la page.

Actuellement, en cas d'erreur sur les formulaires, une boite de dialogue avec le message "oups, une erreur est survenue" s'affiche. A terme, il faudrait afficher un message sur l'écran expliquant le champs sur lequel l'erreur est survenue ainsi que la raison de celle ci. Cela est actuellement impossible à faire car il faut restructurer la façon donc *rest-api.service* fonctionne afin d'y permettre, d'une part, une meilleure gestion des erreurs, d'autre part le fait de ne pas afficher la boite de dialogue systématiquement dans le cas où l'API retourne un code d'erreur.

## 5 Ressources

Github du projet : <https://github.com/jmartin-pro/Equida>

Trello : <https://trello.com/b/jrKixhpu/equida-spring>