

# 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
i	MySQL . . . . .	1
ii	Spring Boot . . . . .	1
iii	Ionic . . . . .	2
iv	Gradle . . . . .	2
C	Organisation du projet . . . . .	2
i	Git et branches . . . . .	2
ii	Les différents dossiers . . . . .	3
iii	Trello . . . . .	3
D	Interactions entre les différentes parties du projet . . . . .	4
i	Les différentes parties . . . . .	4
ii	Configuration Gradle . . . . .	5
<b>2</b>	<b>Core</b>	<b>7</b>
A	Organisation des packages . . . . .	7
B	Exemple d'Entity . . . . .	7
C	Exemple de Repository . . . . .	8
D	Exemple de Service . . . . .	9
E	Exemple d'exception (NotFoundException) . . . . .	10
F	Authentification . . . . .	11
G	Utils . . . . .	11
i	DateUtils . . . . .	11
ii	Sha256PasswordEncoder . . . . .	11

<b>3</b>	<b>Application web</b>	<b>12</b>
A	Organisation des packages . . . . .	12
i	Packages . . . . .	12
ii	Resources . . . . .	12
B	Parler configuration de l'application . . . . .	13
i	application.properties . . . . .	13
ii	Configuration par le code . . . . .	13
C	Fichiers resources . . . . .	13
i	FreeMarker . . . . .	13
D	Parler authentication . . . . .	15
i	Gestion template et controller . . . . .	15
ii	Interceptor . . . . .	15
E	Exemple Route . . . . .	15
F	Exemple Form . . . . .	16
G	Classe InputOutputAttribute . . . . .	17
H	Exemple Controller . . . . .	17
<b>4</b>	<b>Application mobile</b>	<b>19</b>
A	API REST . . . . .	19
i	Organisation des packages . . . . .	19
ii	Parler configuration de l'application . . . . .	19
iii	Parler authentication . . . . .	19
iv	Exemple Route . . . . .	19
v	Exemple Dto . . . . .	20
vi	Exemple Controller . . . . .	20
B	Ionic . . . . .	20
i	Organisation des packages . . . . .	20
ii	Les pages . . . . .	21
iii	Rest Api . . . . .	28

iv	Authentification à l'Api . . . . .	28
v	Problèmes connus . . . . .	28
<b>5</b>	<b>Ressources</b>	<b>29</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'affaires 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 qui lui permettra donc de gérer les chevaux et leurs informations, leurs mises en vente, leurs enchères ainsi qu'une gestion des clients et de leur compte. Cette application combinant 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 : <https://github.com/justine-martin-study/Equida/blob/master/doc/Cahier%20des%20charges.pdf>

On a fait le choix de réaliser un seul projet contenant deux applications : une application web (cf : lien vers appli web) et une application mobile (cf : lien vers appli 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 SpringBoot 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. Chacune des méthodes aura l'annotation @GetMapping, @PostMapping ou toute autre méthode HTTP ayant le modèle suivant @RequestMapping, qui indique l'URL de la page, ainsi que sa 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 : <https://spring.io/projects/spring-boot> ainsi que des tutoriels pour l'utiliser ici [https://www.tutorialspoint.com/spring\\_boot/index.htm](https://www.tutorialspoint.com/spring_boot/index.htm) et là <https://www.javatpoint.com/spring-boot-tutorial>

### 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 : <https://ionicframework.com/docs/installation/cli> avec sa documentation : <https://ionicframework.com/docs/components>.

### iv Gradle

Gradle est un "build automation system" (moteur de production). Il est un équivalent plus récent et complet à Maven. Il possède de meilleures performances, possède un bon support dans 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 créons 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 créera 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 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 gérer 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 séparés par des underscore si besoin tandis que les champs de plusieurs mots sont séparés par des majuscules (comme en 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 : cf lien vers le bon truc. Le sous-dossier Spring, correspondant à SpringBoot, contient, quant à lui, les dossiers core, gradle, rest et webApp. Contenant 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, ...). De même, vous trouverez plus d'informations : cf lien vers le bon truc.

## iii Trello

Le planning, la répartition des tâches et le fonctionnement du projet sont visibles sur le Trello : <https://trello.com/b/jrKixhpu/equida-spring> 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.

## D Interactions entre les différentes parties du projet

### i Les différentes parties

Le projet Équida est composé de 2 applications. Une l'application web, qui est également l'application principale, une application mobile qui est à usage principal des utilisateurs. 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 connecterait 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 afin d'obtenir les identifiants de la base de données. L'utilisation de cette API empêche donc notamment ce problème de sécurité.

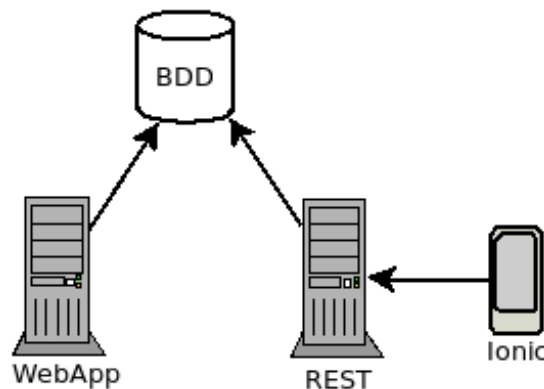


FIGURE 1.1 – La connexion à la BDD selon le projet

L'API ainsi que l'application web utilisent sur le Framework Spring Boot. Ces 2 applications font donc parti de 2 projets différents, "webApp" pour la partie web et "rest" pour l'api. Celles si demandant un code identique pour les Services, les Entités ainsi que les Repository, le choix a donc été fait de faire un projet commun dénommé "core" dans lequel on peut retrouver tout le code qui sera commun aux 2 autres parties, non seulement concernant les éléments cités plus haut mais également concernant les exceptions ou certains outils.

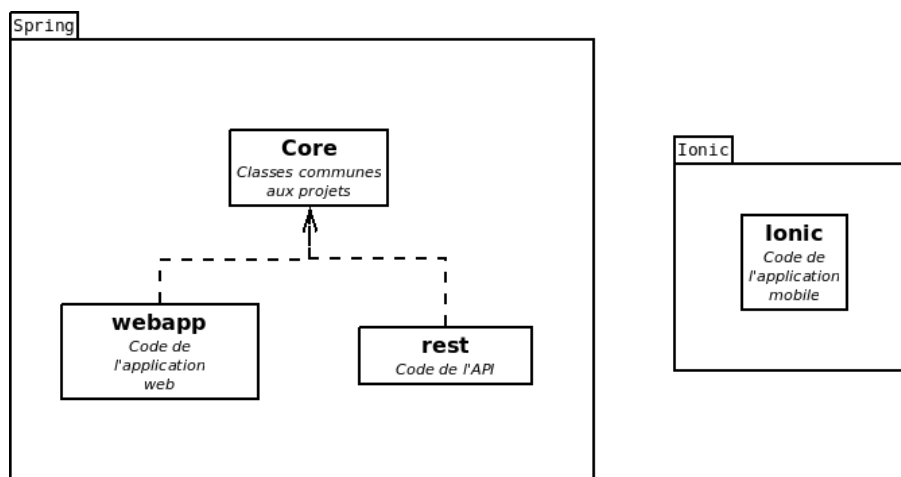


FIGURE 1.2 – Les dépendances entre les projets



## ii Configuration Gradle

Pour gérer correctement les différents projets basés sur Spring, leur dépendances ainsi que leur configuration nous avons donc utilisé Gradle comme mentionné plus haut. Dans le dossier "src/Spring" on retrouve le "build.gradle" qui se charge de configurer tout le projet. On peut observer la configuration suivante pour tout les projets.

```
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 de tous les projets

On définit donc la version de Spring à utiliser, en plus des dépendances commune à chaque projet (spring-boot-starter-web, spring-boot-starter-data-jpa, ...). On va par la suite définir 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 individuelle des projets

De même, concernant le projet core, on active uniquement la compilation en jar (comme une lib) 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 se déroule comme suit :

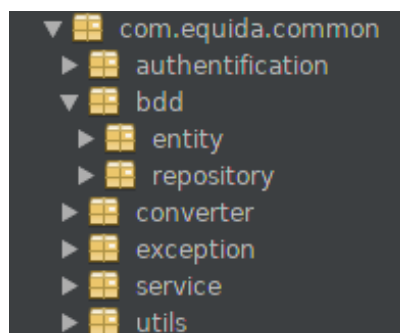


FIGURE 2.1 – Packages de Core

**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 qu'il s'agit d'un id et qu'il est auto-généré avec `@Id` et `@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 les 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 ensuite 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.

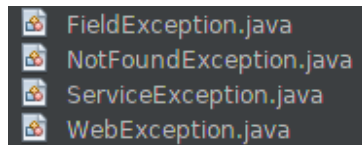


FIGURE 2.5 – Les différentes exceptions

**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. On peut par exemple citer un sire qui n'existe pas dans la base de données, un login qui existe déjà, ...

**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 utilisés 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 plutot 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 y définit un simple message d'erreur et on change le code de réponse HTTP, grace à l'annotation ResponseStatus, pour qu'il envoie 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 ce qui est du module 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 géré 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 fournit en parametre au format "jour/mois/année" grâce à la méthode "String format(Date)".

### ii Sha256PasswordEncoder

Cette classe permet de gérer le hachage des mots de passes. Elle implémente l'interface "PasswordEncoder". Celle ci fournit une méthode publique "String encode(CharSequence)" retournant en String le hachage, en sha256 dans notre cas, de la chaine de caractère fournit en paramètre. Cette classe est fournit à Spring Security afin de vérifier les informations saisies par l'utilisateur lors d'une connexion. Elle est également utilisé 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 :

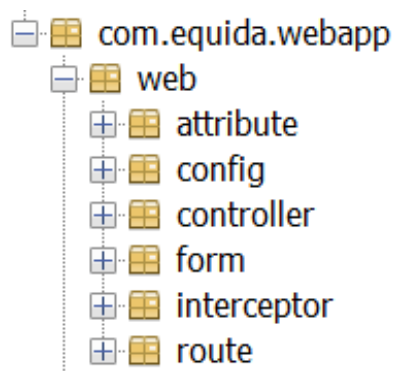


FIGURE 3.1 – Packages de WebApp

**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 toutes 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 [https://www.tutorialspoint.com/spring\\_boot/spring\\_boot\\_interceptor.htm](https://www.tutorialspoint.com/spring_boot/spring_boot_interceptor.htm))

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

#### ii Resources

L'organisation des ressources se présente comme suit :



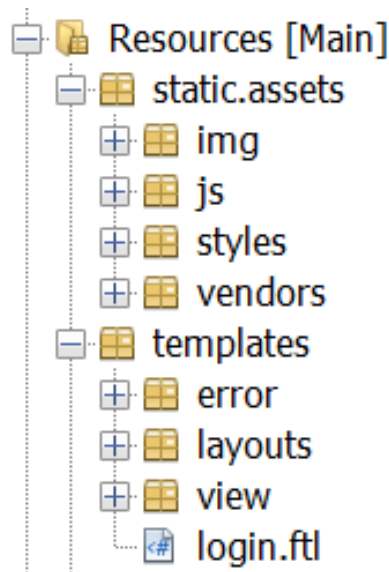


FIGURE 3.2 – Ressources de WebApp

**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 Parler configuration de l'application

---

### i application.properties

### ii Configuration par le code

## 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.

FreeMarker lit donc les fichiers Model, les combine avec les objets Java pour finalement générer un document de sortie type HTML (dans un format de fichier FTL FreeMarker Template Language).

## 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 macro @content, présente dans le body de base.ftl, chargera le code du fichier x.ftl à cet endroit.

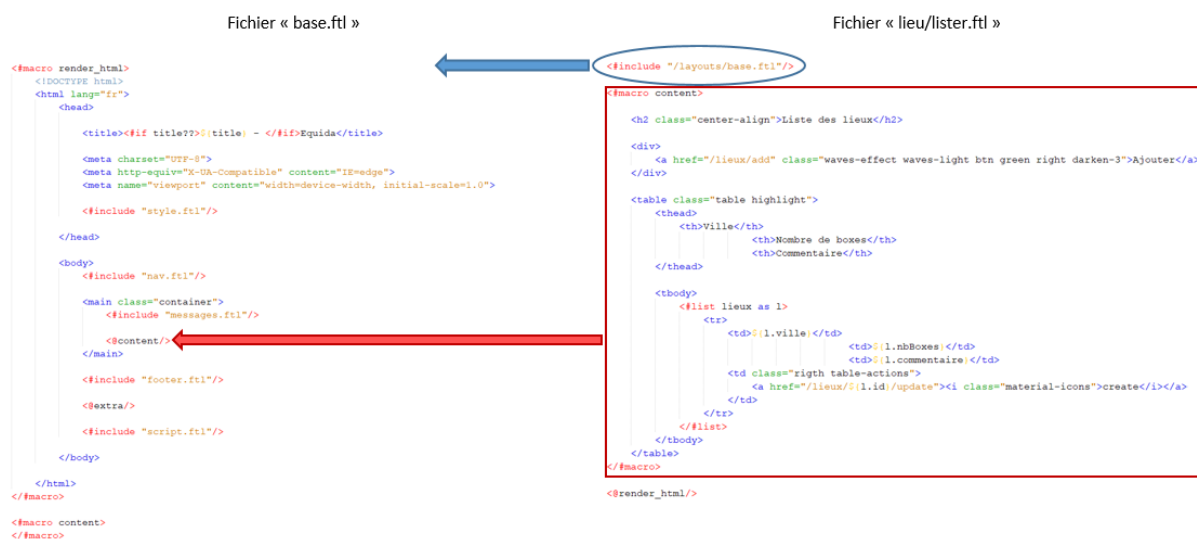


FIGURE 3.3 – Exemple du fonctionnement de la page base.ftl avec la page lieux/liste.ftl

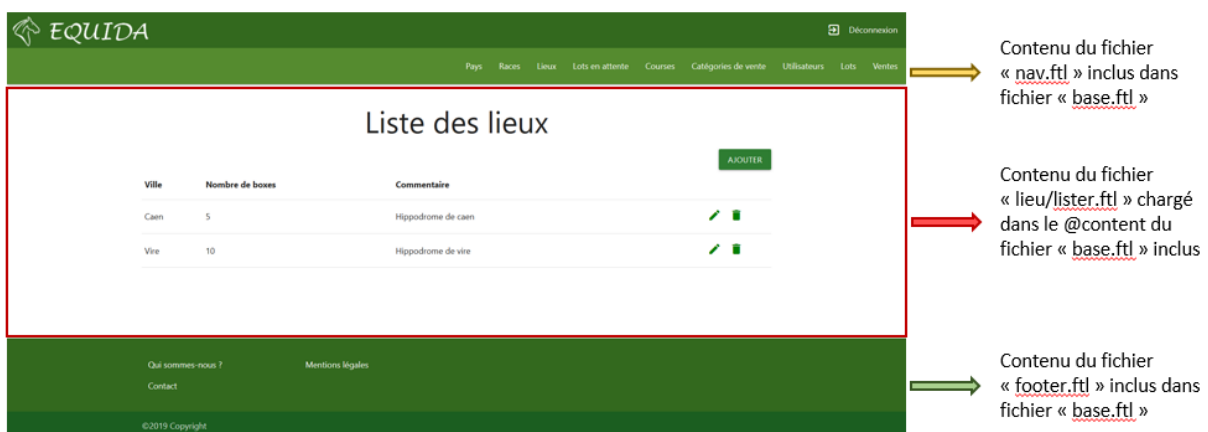


FIGURE 3.4 – Rendu lors du chargement de la vue qui liste les lieux (correspondant à la page lieux/liste.ftl précédente)

## 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 3.5 – Code de l'erreur 404

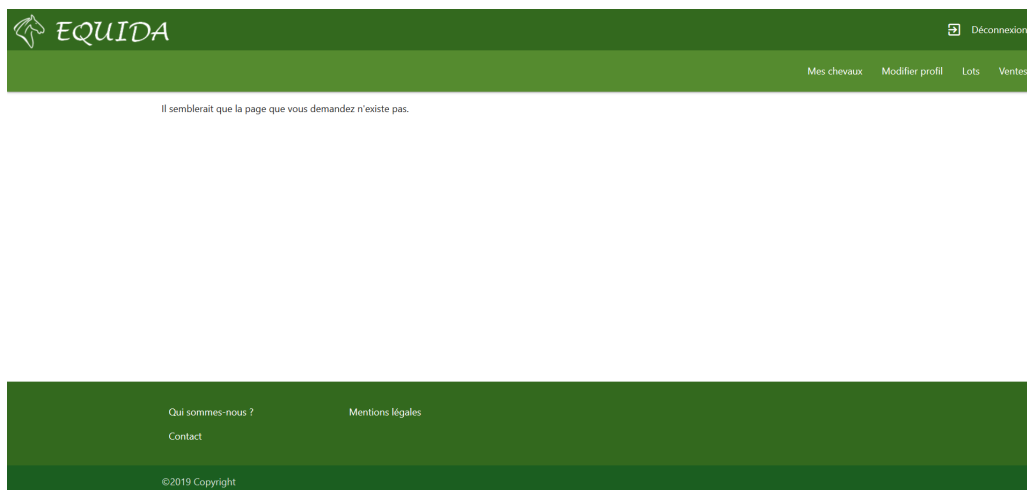


FIGURE 3.6 – Rendu lors d'une tentative de chargement d'une page inexistante

## Autre

## D Parler authentication

### i Gestion template et controller

### ii Interceptor

## E Exemple Route

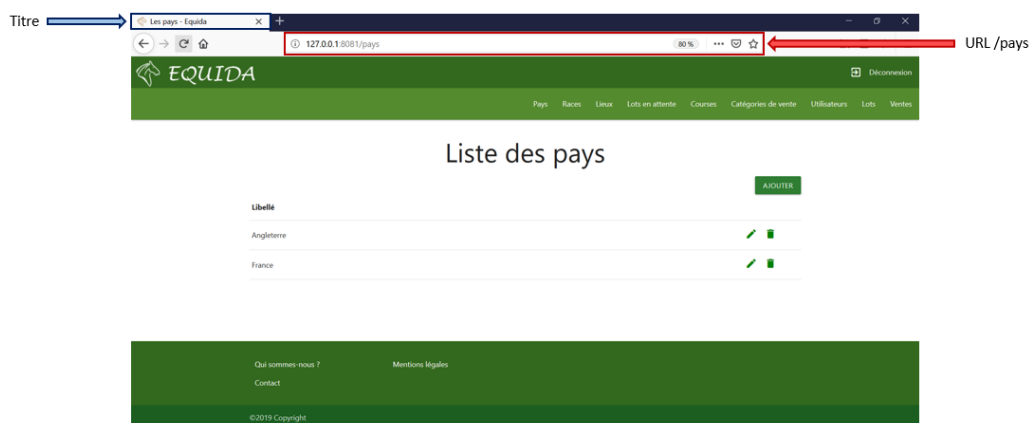
L'interface IRoute décrit les méthodes qui doivent être implémentées par les classes filles. Ainsi chaque fichier route contiendra une méthode `getUri()`, une méthode `getView()` et `getTitle()` qui

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/liste 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/liste";     }      @Override     public String getTitle() {         return "Les pays";     }  } </pre>

**FIGURE 3.7** – Code de l'interface et utilisation par la classe fille PaysRoute



**FIGURE 3.8** – Rendu obtenu avec la vue pays/liste

## F Exemple Form

La classe mère IForm est une classe abstraite qui utilise la généréicité 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 3.9 – Exemple implémentation interface IForm avec LieuxForm

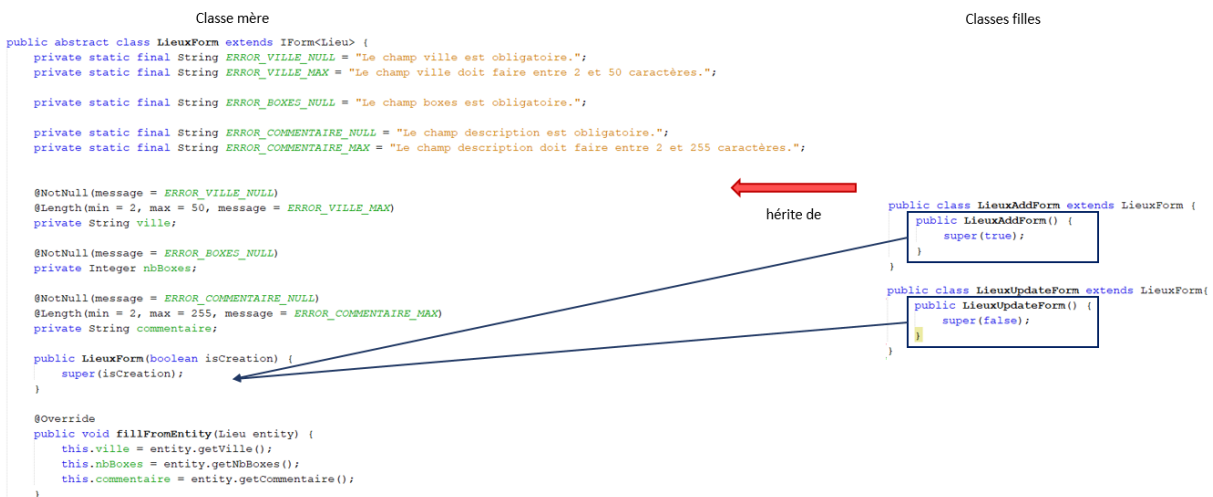


FIGURE 3.10 – Exemple avec LieuxUpdateForm et LieuxAddForm

## G Classe InputOutputAttribute

## H Exemple Controller

Les différents contrôleurs créés pour les entités x héritent tous de la classe AbstractWebController. Les contrôleurs contiennent les différentes fonctions correspondant aux méthodes get, post, patch et delete qui utilisent les services et les routes associées. C'est aussi ici qu'on gère les différentes autorisations afin de savoir qui peut accéder, utiliser telle fonction.

Par exemple, dans EncheresController, la fonction addGet n'est possible que pour un utilisateur ayant le role 'ADMIN', elle est reliée à l'URL EncheresAddRoute.RAW\_URI (URL stockée dans la variable RAW\_URI de la classe EncheresAddRoute). Elle permet de récupérer le formulaire d'ajout d'une enchère.

```

@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 3.11** – Exemple d'un contrôleur

## 4 Application mobile

### A API REST

#### i Organisation des packages

L'organisation des packages se déroule comme suit :

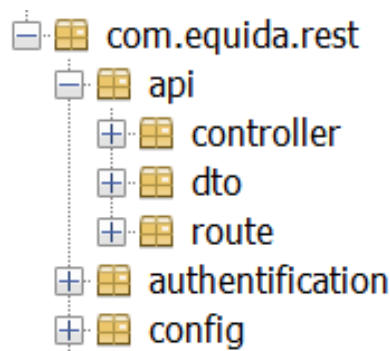


FIGURE 4.1 – Packages de l’api Rest

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

**controller** : Contient la classe qui gère toutes les variables

**dto** : Contient les classes relatives à la configuration et la sécurité de l’application

**route** : Contient toutes les contrôleurs qui héritent de la classe AbstractWebController

**authentification** : Contient la classe de configuration de gestion de l’authentification sur l’api rest.  
Implémentation de BasicAuthenticationEntryPoint sur l’application

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

#### ii Parler configuration de l’application

**application.properties**

**Configuration par le code**

#### iii Parler authentification

#### 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 selon notre nomenclature, l'URL renvoyée est /api/lots.

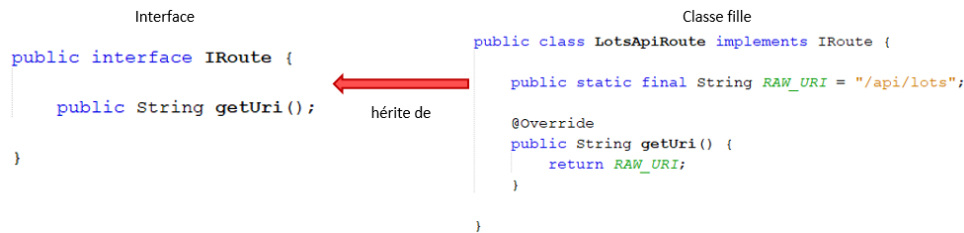


FIGURE 4.2 – Interface IRoute et exemple avec LotsApiRoute

## v Exemple Dto

## vi Exemple Controller

Les différents contrôleurs sont des classes précédées de l'annotation @RestController. Ils contiennent les différentes fonctions correspondant aux méthodes get, post, patch et delete qui utilisent les services et les routes associées. C'est aussi ici qu'on gère les différentes autorisations afin de savoir qui peut accéder, utiliser telle fonction avec l'annotation @PreAuthorize("hasRole('ROLE\_X')").

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 (objet simplifié).

```
@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);
    }
}
```

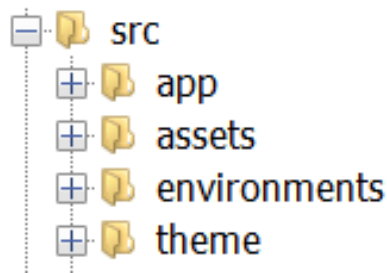
FIGURE 4.3 – Exemple d'un contrôleur avec LotDetailsRestController

# B Ionic

## i Organisation des packages

L'organisation des packages se déroule comme suit :





**FIGURE 4.4** – Packages de ionic

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

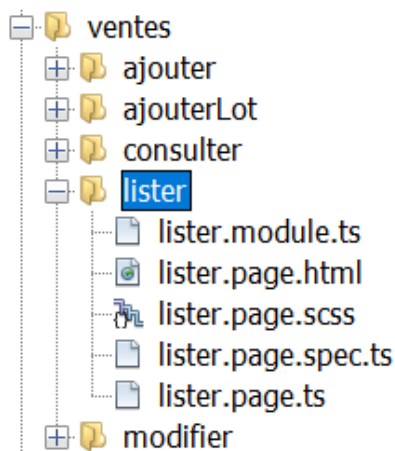
**assets** : Contient les images de l'application

**environments** : Contient

**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 nommées par un verbe (souvent : lister, ajouter, consulter, modifier). Ces pages sont donc placées dans un dossier du nom de l'entité concernée.



**FIGURE 4.5** – 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, ...) créé par la commande ionic, on retrouvera les mêmes types de fichiers.

**x.module.ts** : Contient

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

**x.page.scss** : Contient

**x.page.spec.ts** : Contient

**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 role de l'utilisateur (\*ngIf="role == 'ADMIN'"), si l'utilisateur a un role 'ADMIN' alors il verra un bouton flottant d'ajout. Ce bouton le renverra vers la page d'ajout d'une vente. Pour finir, on gère le nombre de ventes affichées sur la page et le chargement d'autres lors d'un scroll de l'utilisateur.

```
<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 4.6 – Code de /ventes/lister/lister.page.html

La page `lister.page.ts`, contient l'initialisation des variables, le constructeur, l'initialisation de la classe avec la méthode `ngOnInit` qui appelle notamment la méthode `getVentes`. Cette méthode appelle la méthode `loadVentes` qui chargera donc les ventes tant qu'il y en a (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 4.7 – Code méthode loadVentes de /ventes/lister/lister.page.ts

Liste des ventes

Rapsberry Sailings
23/07/2017 - 25/09/2019
30/07/2019
Vire - Hippodrome de vire
Vente d'élevage
Vente de pur sang
15/03/2019 - 09/09/1999
15/03/2019
Vire - Hippodrome de vire
Vente d'élevage
Vente de jument
14/05/2019 - 15/05/2019
15/05/2019
Caen - Hippodrome de caen
Vente d'élevage

FIGURE 4.8 – 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 que des boutons pour modifier et supprimer, tous deux présents 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 4.9 – Extrait du code de /ventes/consulter/consulter.page.html

La page `consulter.page.ts`, contient l'initialisation des variables, le constructeur, l'initialisation de la classe avec la méthode `ngOnInit` qui appelle notamment les méthodes `getVenteById` et `getLotsById-Vente`.

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 `getCateg-VenteById` 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 les pages 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 4.10 – Extrait du code de /ventes/consulter/consulter.page.ts

Consultation vente


Raspberry 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 4.11 – 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 4.12 – Code de /ventes/ajouter/ajouter.page.html

La page ajouter.page.ts, contient l'initialisation des variables, le constructeur, l'initialisation de la classe 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 4.13 – 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 4.14** – 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 l’initialisation des variables, le constructeur, l’initialisation de la classe 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 4.15** – 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 4.16** – Résultat de la page de modification d'une vente

### iii Rest Api

### iv Authentification à l'Api

### v 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.



## 5 Ressources

Github du projet : <https://github.com/justine-martin-study/Equida>

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