

Équida

Doc Technique

MARTIN Justine
BOTTON Léa

Table des matières

1	Contexte et présentation	1
A	Présentation du contexte	1
B	Choix techno	1
i	MySQL	1
ii	Spring Boot	1
iii	Ionic	1
iv	Gradle	1
C	Organisation du projet	1
i	Git et branches	1
ii	Les différents dossiers	1
iii	Trello	1
D	Interactions entre les différentes parties du projet	2
i	Les différentes parties	2
ii	Configuration Gradle	3
2	Core	5
A	Organisation des packages	5
B	Exemple d'Entity	5
C	Exemple de Repository	5
D	Exemple de Service	5
E	Exemple d'exception (NotFoundException)	5
F	Authentification	6
G	Utils	7
i	DateUtils	7
ii	Sha256PasswordEncoder	7

3	Application web	8
A	Organisation des packages	8
i	Packages	8
ii	Resources	8
B	Configuration de l'application	8
i	application.properties	8
ii	Configuration par le code	8
C	Fichiers resources	9
i	Freemarker	9
ii	Assets (images, js, ...)	11
D	Gestion de l'authentification	11
i	Gestion template et controller	11
ii	Interceptor	11
E	Exemple Route	12
F	Exemple Form	12
G	Classe InputOutputAttribute	12
H	Les Controleurs	13
i	AbstractWebController	13
ii	Exemple Controller	14
4	Application mobile	15
A	API REST	15
i	Organisation des packages	15
ii	Parler configuration de l'application	15
iii	Système d'authentification sur l'API	16
iv	Exemple Route	17
v	Les Dto	17
vi	Exemple Controller	17
B	Ionic	17

i	Organisation des packages	17
ii	Les pages	18
iii	Rest Api	18
iv	Problèmes connus	22

5	Ressources	23
----------	-------------------	-----------

1 Contexte et présentation

A Présentation du contexte

B Choix techno

i MySQL

ii Spring Boot

iii Ionic

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

Branches

Nomenclature

ii Les différents dossiers

Doc

SQL

Sources

iii Trello

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.

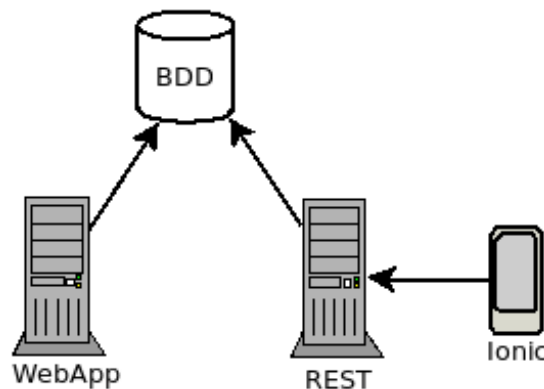


FIGURE 1.1 – La connexion à la BDD selon le projet

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.

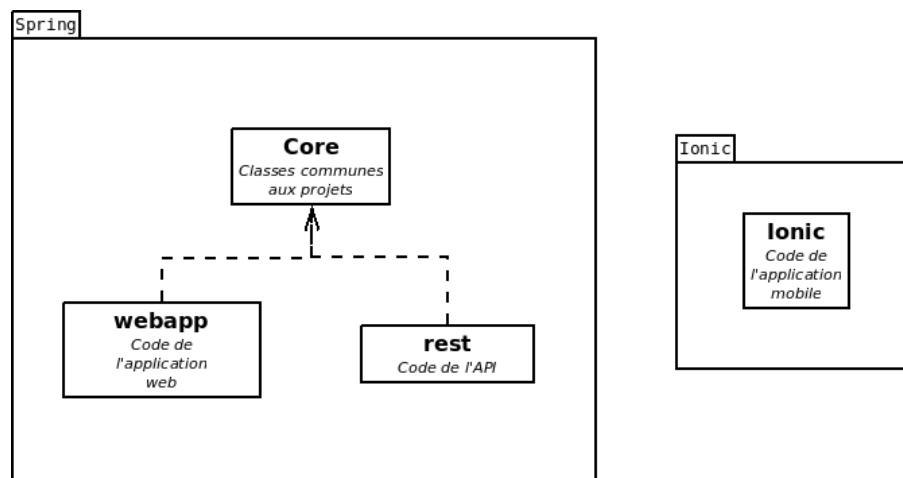


FIGURE 1.2 – Les dépendances entre les projets

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 :

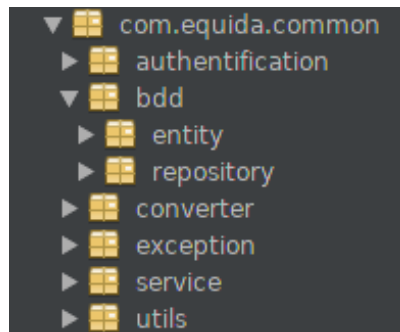


FIGURE 2.1 – Packages de Core

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

C Exemple de Repository

D Exemple de Service

E Exemple d'exception (`NotFoundException`)

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

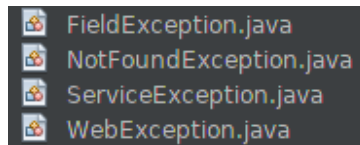


FIGURE 2.2 – Les différentes exceptions

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 2.3 – 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. Grace 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

ii Resources

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 3.1 – Configuration par `application.properties`

Dans 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 3.2 – 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" fournis par le 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 resources

i Freemarker

Comme mentionné auparavant, tout les fichiers de freemarker sont contenus dans le dossier "templates" dans les ressources. Il est donc possible d'utiliser les directives propres à ce moteur de template.

Page de base

Page d'erreur

Fichiers à inclure

Le dossier "view/include/" contient des 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 3.3 – Configuration de Spring Security

Le code reste celui de n'importe quelle autre vue et ne comporte aucune spécificité telle que des directives "<#macro>".

Exemple de page web

Ainsi, avec toutes 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>
        <#else>
        <p>Les inscriptions sont uniquement ouvertes du ${vente.dateDebut?string["dd/MM/yyyy"]} au ${vente.dateFin?string["dd/MM/yyyy"]}</p>
        </#if>
    </div>
</div>

```

FIGURE 3.4 – Code de la vue pour la consultation d'une vente

On retrouve tout les éléments précédemment cités. L'include de "/layouts/base.ftl" pour reprendre le template de base, un changement de valeur concernant le titre, l'utilisation de la macro "content", ...

ii Assets (images, js, ...)

D Gestion de l'authentification

i Gestion template et controller

Un controleur existe afin de gérer l'affichage d'un template freemarker concernant la page de connexion à l'application. Ce controleur 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 controleurs une classe "UserInterceptor" permet de fournir automatiquement l'instance de "AuthenticatedUser" à la vue ou au controleur.

```

@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 3.5 – Code de UserInterceptor

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

E Exemple Route

F Exemple Form

G Classe InputOutputAttribute

Cette classe permet la définition de constantes qui seront utilisés 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 3.6 – Code de InputOutputAttribute

Elles 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. Ces constantes sont donc énormément utilisés pour transmettre les informations du controleur vers la vue.


```

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

```

FIGURE 3.7 – Exemple d'utilisation de InputOutputAttribute

H Les Controleurs

i AbstractWebController

Cette classe est la classe mère de tout les controleurs. On y définit certaines méthodes comme la possibilité d'ajouter des messages d'erreurs, ou à titres informatifs, qui seront transférés 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 3.8 – Méthodes pour la gestion des messages

On y définit aussi la gestion des exception dans lesquels on passe les variables necessaires 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 3.9 – Méthodes pour la gestion des exceptions

Elle définit aussi 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 3.10 – Méthodes pour la gestion des form

ii Exemple Controller

4 Application mobile

A API REST

i Organisation des packages

ii Parler 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 4.1 – Configuration par application.properties

Dans 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 4.2 – Configuration de Spring Security

Ici, on exige l'authentification sur toutes les requêtes faites à l'API. Celle-ci 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 4.3 – 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 ce dont chaque URL utilisait.

```
@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 4.4 – Configuration de l'authentification

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

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.SC_UNAUTHORIZED);
    }

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

FIGURE 4.5 – 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

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 4.6 – 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 classe métier en un Dto et "T convertToEntity()" permettant de convertir notre Dto en une instance de nos classe métiers.

Exemple de Dto

vi Exemple Controller

B Ionic

i Organisation des packages

ii Les pages

Lister

Consulter

Ajouter

Modifier

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 4.7 – Code de l'interface IDto

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 de d'appeler "/api/login" et d'obtenir les informations relatives à l'utilisateur si les informations fournis 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 4.8 – Code de l'interface IDto

Ces informations sont ensuite réutilisées pour permettre l'authentification à l'API. Pour cela la méthode "getHttpOptions" permet d'obtenir l'entête à fournir à 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 4.9 – Code de l'interface IDto

On effectue ici l'appel à l'url fournit en paramètre tout en récupérant les entêtes nécessaire pour l'authentification puis, selon que l'appel ai réussi ou non, on retourne le résultat ou on appel 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 change pas tellement si ce n'est que pour POST et PATCH on fournis en plus un objet "data" qui correspond aux informations à fournir à l'API mais aussi que le nom des méthodes appellées 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 4.10 – Code de l'interface IDto

La méthode "handleError" affiche les informations relatives à 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 4.11 – Code de l'interface IDto

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

Actuellement, en cas d'erreur sur les formulaires, une boîte de dialogue avec un message expliquant qu' "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 boîte de dialogue systématiquement dans le cas où l'API retourne un code d'erreur.

5 Ressources

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

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