

Laurent Picard  
Juin 2019

SMART FITNESS

# Présentation du projet « Smart fitness »

« Manage your fitness center »



More connected, more advantageous

## Table des matières

Introduction.....	iii
1 – Genèse du projet .....	1
1.1 Analyse de la concurrence .....	1
1.1.1 Le marché .....	1
1.1.2 Le modèle connecté .....	1
1.2 Les motivations du projet.....	2
1.3 Les utilisateurs de l'application .....	2
1.3.1 L'axe clientèle .....	2
1.3.2 L'axe gérance.....	2
1.4 Contextualisation de l'application.....	3
2. Modélisation .....	4
2.1 Analyse des besoins utilisateurs.....	4
2.1.1 Diagramme package .....	4
2.2 Réservation en ligne d'une séance (fonctionnalité « réservation ») .....	5
2.2.1 Diagramme de cas d'utilisation .....	5
2.2.2 User Story « Service d'inscription ».....	6
2.2.3 User Story « Service d'inscription ».....	6
2.2.4 User Story « Constituer une séance en réservant un ou plusieurs équipements » .....	6
2.2.3 Diagramme de séquence.....	8
2.3 Package manager .....	9
2.3.1 La gestion du parc des équipements.....	9
2.3.2 La gestion des offres.....	9
2.3.3 Le pilotage opérationnel de l'activité.....	9
2.4 Package Admin .....	9
2.5 Prototypes d'interfaces (« Wireframes »).....	10
2.5.1 Wireframe « Liste des équipements disponibles » (User case « Réserver une séance ») ...	10
2.5.2 Wireframe « Tableaux évolution du taux de réservation & rendement par équipement » (fonctionnalité pilotage opérationnel de l'activité) .....	11
2.6 Gestion des utilisateurs et des accès .....	12
2.7 Diagramme de classes (MOO, Modèle Orientée Objet).....	13
2.7.1 Diagrammes de classes – commentaires.....	14
2.8 Le Modèle Logique de Données (MLD) .....	15
.....	15
2.9 Le Modèle Physique de Données (MPD) .....	16
3 – Développement .....	17

3.1 Architecture de l'application .....	17
3.1.1 Le serveur de présentation .....	17
3.1.2 Le serveur d'application .....	19
3.1.2 Le serveur de base de données .....	20
3.1.3 Le fichier application.properties .....	20
3.2 Mise en place de l'environnement de développement .....	21
3.2.1 Les outils de développement .....	21
3.2.2 Le jeu des annotations .....	22
4. Module d'authentification .....	23
4.1 Mise en base des données utilisateurs .....	23
4.2 Mise en place de la politique de sécurité .....	24
4.2.1 La gestion de l'authentification .....	24
4.2.2 La gestion des tokens .....	24
4.2.3 La gestion des autorisations .....	26
4.2.3 La gestion des accès aux pages du site .....	28
5. La gestion de réservation d'équipements .....	29
5.1 Obtention de la liste d'équipements .....	29
5.2 Constitution et validation d'une séance .....	31
5.2.1 Mise en œuvre côté « back-end » (serveur d'application) .....	31
5.2.2 Mise en œuvre côté « front-end » (serveur de présentation) .....	31
5.2.3 Gestion de sélections concurrentes d'équipements .....	33
.....	33
La section suivante traite de la gestion du parc .....	33

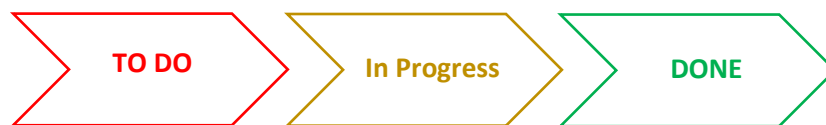
## Introduction

Ce document a pour but de présenter le projet « Smart fitness » en partant du cahier des charges jusqu'à l'implémentation du site. Le thème du projet est la gestion d'un centre de fitness qui permettra d'une part aux clients de se constituer en ligne des séances de fitness avec une grille de tarification « low-cost », et d'autre part à aider le staff dans ce qui relève de l'organisation du centre.

Pour ce qui est de la conduite de notre projet, nous travaillerons en mode agile, en cherchant à implémenter une ébauche de solution qui soit opérationnelle dès le départ et que nous enrichirons au fur et à mesure. Autrement dit de faire de l'adage « Arrêtons de commencer, commençons par finir » un principe de base. Nous appliquerons ce mode opératoire pour les différentes thématiques structurant notre projet dont voici le fil d'Ariane :



Nous matérialiserons l'état de l'avancée de chaque module par le code couleur suivante :



\*

\* \*

La section qui suit présente la genèse du projet.



# 1 – Genèse du projet

## 1.1 Analyse de la concurrence

### 1.1.1 Le marché

Une étude sur le marché du fitness publiée en 2018 par [Europe Active](#) nous révèle que le chiffre d'affaire pour l'exercice 2017 a été de 26,6 milliard d'euros en Europe. Le nombre total de membres de clubs de santé et de fitness a ainsi atteint la barre des 60 millions de personnes (80 millions selon des projections pour 2025), ce qui en fait la première activité sportive européenne. Une tendance de fond se dessine également avec l'observation de la baisse du revenu moyen par membre alors que le nombre d'adhésion continue de croître. Si l'on se focalise sur le marché français, on dénombre 4200 clubs soit une hausse de 5% en un an. En outre, avec la démocratisation du fitness et le développement de la concurrence entre clubs, le coût mensuel moyen dépensé par adhérent est passé de 41€ à 40€ par mois entre 2016 et 2017.

Aujourd'hui, avec l'aménagement de réseaux de salles de sport, les activités ne se limitent plus à la musculation et au cardio-training mais proposent tout un éventail d'activités avec le support de coaches diplômés. En réponse au low-cost et à la standardisation, le concept de *Boutiques Gyms* propose lui aussi la pratique d'une seule activité de façon très « immersive » à des prix plutôt élevés. Dans le cadre de notre projet, nous reprendrons pour notre modèle le principe du « pay as you go » (*je ne paye que ce que je consomme sans m'engager*) qui a fait le succès de ces *Boutiques Gyms*.

### 1.1.2 Le modèle connecté

Sur le créneau du numérique, le marché des Apps pour le sport connaît également un fort engouement. Elles permettent une gestion et un suivi personnalisé des pratiques sportives et visent d'une manière générale à prendre en main sa santé : Aujourd'hui plus de 165 000 applications la santé sont disponibles sur l'App Store. L'utilisation des objets connectés, en particulier celles des montres, facilitent la gestion des activités et le suivi personnalisé au quotidien. Il existe ainsi des applications, comme l'application *Course à pied* permettant de suivre ses trajets et ses temps directement sur son smartphone. Ces données sont sauvegardées de façon à pouvoir analyser ensuite les courses.

Les équipements des salles de fitness sont orientés dans une approche connectée. Ils sont dorénavant équipés d'écrans tactiles permettant à un utilisateur d'entrer son login, de traquer et moduler à sa convenance l'intensité de ses efforts. Dans le cadre de notre projet, le site proposera un catalogue de montres connectées pour permettre aux utilisateurs d'entrer leurs données de suivis. En aparté des modèles de montres connectées, le catalogue proposera également en guise de service un panel de boissons énergisantes et de produits d'alimentation.



## 1.2 Les motivations du projet

- Sur le plan fonctionnel : le sujet du projet doit pouvoir s'appuyer sur un cas d'étude dont la mise en œuvre réside dans sa capacité à répondre à un besoin réel. Un autre critère relève de la diversité des problématiques organisationnelles, comme la gestion des commandes ou la planification de la réservation d'équipements.
- Sur le plan technique : le projet doit permettre de couvrir les différentes couches techniques d'une application web tant sur le plan du backend que celui du frontend, le tout adossé à une base de données relationnelle
- Le choix guidant le thème de l'application se mesure également en termes de plus-values qu'elle est en mesure d'apporter à ses différents utilisateurs : d'une part, le suivi des commandes et de la planification des séances pour les clients, et d'autre part une synthèse du parc des équipements en termes de coût et de revenus pour les gestionnaires d'un centre de fitness.

C'est pourquoi le choix d'un centre de fitness semble bien se prêter pour aborder ces différentes thématiques.

## 1.3 Les utilisateurs de l'application

Cette partie va nous permettre de définir qui utilisera l'application, de quelle manière et quelles seront les fonctionnalités implémentées. Pour ce faire, nous allons procéder à une analyse des besoins.

Les fonctionnalités générales du site peuvent être réparties autour de deux axes : celui de la clientèle et celui de la gérance.

### 1.3.1 L'axe clientèle

Les clients auront la possibilité de créer un compte utilisateur afin d'avoir accès aux différents services proposés par le site :

- Constituer une ou plusieurs séances en sélectionnant pour chacune d'entre elles un ou plusieurs équipements.
- Visualiser sous forme de feuille de route le contenu de chaque séance réservée.
- Souscrire à un abonnement afin de bénéficier des séances à moitié prix.
- Visualiser le catalogue de la boutique en ligne et acheter des produits.
- Accéder à l'historique des commandes.

### 1.3.2 L'axe gérance

Le staff de « Your smart fitness » disposeront des fonctionnalités suivantes :

- La gestion des infrastructures du site (ajout et paramétrage des équipements).
- La gestion des offres (création de formules d'abonnements et mise en ligne d'un catalogue)
- La balance des revenus et dépenses pour chaque équipement.
- La synthèse annuelle glissante de l'évolution du taux de réservation.
- La gestion des comptes utilisateurs du staff.
- La gestion de diffusion d'information



## 1.4 Contextualisation de l'application

Le périmètre contextuel de l'application nous permet d'en fixer ses modalités d'usage :

- L'ensemble des équipements disponibles à la réservation et leurs tarifs de prestation sont saisis par le staff. Cette grille tarifaire peut être évolutive au fil du temps et est propre à chaque équipement. On appelle prestation, la réservation d'un équipement par un client pour une durée de 10'. Chaque séance est une séquence de réservation d'équipements de d'une durée de 10'
- Chaque équipement est affecté à une catégorie et est localisée dans une salle. On appelle catégorie, une famille d'équipement.
- Une séance est constituée d'au moins une réservation d'un équipement (donc 10') et au plus d'un ensemble de réservations limitées à une même journée. Un client peut se créer plusieurs séances dans une journée.
- Le staff saisit l'ensemble des offres (types d'abonnements et articles du catalogue) qui seront proposées aux clients disposant d'un compte « Smart Fitness ».
- Chaque client devra donc créer un compte utilisateur pour pouvoir accéder aux différents services proposés par le site.
  - L'abonnement permet aux clients de bénéficier de la réservation des équipements à moitié prix.

La section suivante aborde la modélisation de l'application



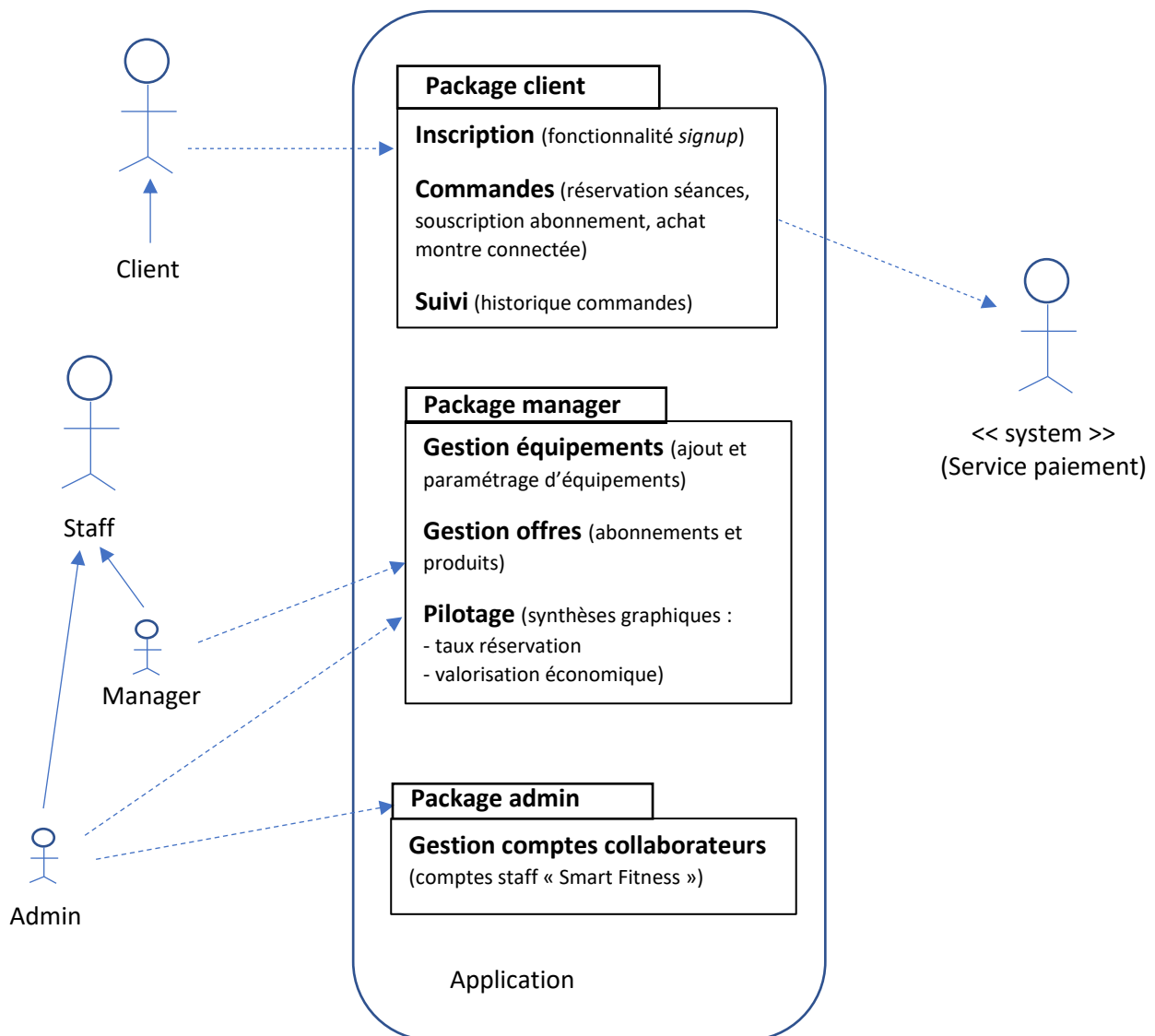
## 2.Modélisation

### 2.1 Analyse des besoins utilisateurs

Cette section a pour but de dresser les différents scénarios d'utilisation de l'application.

#### 2.1.1 Diagramme package

Le diagramme de package va nous permettre de décomposer le système en modules et d'indiquer quels sont les acteurs et à quel niveau ils interagissent avec l'application. Pour notre projet, l'application sera divisée en trois packages. Au sein de chaque package, nous pouvons répertorier les fonctionnalités suivantes :



Nous nous concentrerons en priorité sur le package **client** dans la mesure où il met en interaction le client et le système et qu'il constitue le cœur de l'application, sans négliger toutefois les deux autres. Nous étudierons donc dans un premier temps le scénario traitant de la réservation en ligne d'une séance à l'aide de deux diagrammes de modélisation : le diagramme de cas d'utilisation et le diagramme de séquence. Puis dans un second temps, nous présenterons les fonctionnalités des deux autres packages : la gestion du parc des équipements, celles des offres, le pilotage opérationnel du



point de vue comptable (package **manager**), et la gestion des comptes utilisateurs du staff (package **admin**).

Le *diagramme de cas d'utilisation* permet de représenter les fonctionnalités proposées aux utilisateurs. Il est orienté utilisateur et modélise à QUOI sert le système en décrivant un ensemble de services initiés par l'utilisateur et rendus par le système. Dans notre cas, nous compléterons le diagramme de cas d'utilisation par une User Story pour décrire en détails l'enchaînement des différentes séquences.

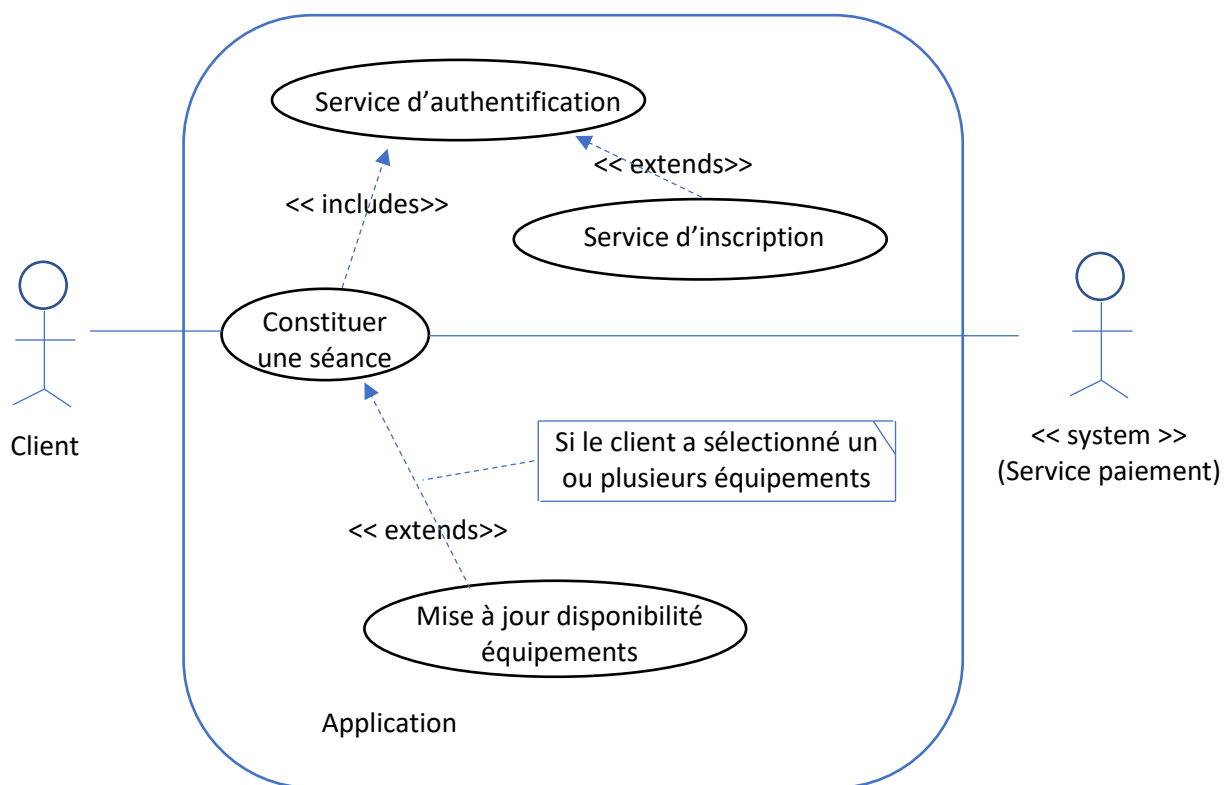
En complément du diagramme de cas d'utilisation, le *diagramme de séquence* permet lui de montrer les interactions entre les acteurs et le système selon un ordre chronologique. Il décrit COMMENT les éléments du système interagissent entre eux et avec les acteurs :

- Les objets au cœur d'un système interagissent en s'échangeant des messages
- Les acteurs interagissent avec le système au moyen d'IHM (Interfaces Homme-Machine).

Dans notre cas, le diagramme de séquence correspondra à la retranscription visuelle du User Story.

## 2.2 Réservation en ligne d'une séance (fonctionnalité « réservation »)

### 2.2.1 Diagramme de cas d'utilisation



Les sections suivantes se proposent de détailler chaque item du diagramme de cas d'utilisation au travers de « user stories ».

### 2.2.2 User Story « Service d'inscription »

Cette section a pour but de détailler l'item

Service d'inscription

La création d'un compte utilisateur sur le site de « Smart Fitness » est le préalable nécessaire pour accéder à l'ensemble des fonctionnalités proposées aux clients du sites

*En tant qu'utilisateur, je souhaite pouvoir créer un nouveau compte en cas d'inexistence de celui-ci. Un bouton me permettra d'ouvrir une nouvelle page d'inscription sur laquelle je renseigne les informations suivantes :*

- *Un identifiant unique. Je dois être immédiatement averti si l'identifiant est déjà pris.*
- *Mes nom et prénom.*
- *Mon email et une confirmation d'email afin d'être certain de la saisie.*
- *Un password comportant au moins sept caractères, une majuscule et un caractère spécial. La confirmation de mon password.*
- *Ma date de naissance via un calendrier.*
- *Mon numéro de téléphone*
- *Mes adresses du domicile et de livraison avec la possibilité d'affecter l'adresse du domicile à l'adresse de livraison.*

*Je souhaite que les erreurs affichées soient explicites :*

- *En cas d'erreur sur l'email, afficher un message explicite disant que l'erreur porte sur l'email*
- *En cas d'erreur sur le mot de passe, afficher un message explicite disant que l'erreur porte sur le mot de passe*
- *En cas d'indisponibilité du service, afficher un message d'indisponibilité du service d'authentification et offrir un numéro de support téléphonique*

*A la suite de mon inscription, je recevrai un email me permettant de confirmer la création de mon compte pour me connecter au site.*

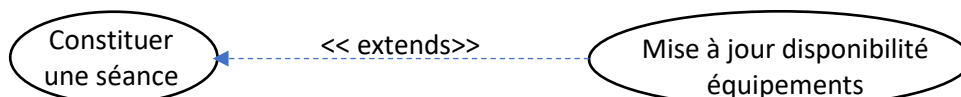
### 2.2.3 User Story « Service d'inscription »

Service d'authentification

- Le client accède à son espace personnel par le service d'authentification.

*En tant qu'utilisateur, je voudrais pouvoir m'authentifier à travers une IHM à mon compte pour pouvoir accéder aux opérations de réservation de séances.*

### 2.2.4 User Story « Constituer une séance en réservant un ou plusieurs équipements »



Cette section présente le processus mettant en situation un utilisateur qui sélectionne et ajoute un ou plusieurs équipements à sa séance, ce qui implique de mettre à jour l'affichage de la disponibilité des équipements pour chaque tranche horaire.

- Le client sélectionne le jour.

*Je souhaite disposer à la fois des fonctionnalités d'un calendrier, du défilement incrémentiel des jours et de la saisie du type champ texte pour choisir à ma convenance le jour de ma séance.*

- Le client accède à la page de la liste des équipements positionnée sur la prochaine tranche horaire à venir. Le client peut se positionner sur une autre tranche de 10' jusqu'à 22h ou changer de jour. Les équipements disponibles sont regroupés par type d'équipement : les elliptiques, les tapis roulants, les vélos, l'espace musculation ...

*A tout instant la liste des équipements doit être à jour. Pour chaque équipement, je dois pouvoir accéder à une fiche descriptive comportant une photo de l'équipement en question.*

- Le client sélectionne un équipement afin de l'ajouter à sa séance (ici on peut assimiler la séance à la notion de panier).

*Si je choisis un équipement qui était disponible au moment du chargement de la page, mais qu'entre-temps un autre utilisateur l'a réservé, je dois être informé par un message que je ne serai pas en mesure de l'intégrer dans ma séance.*

- Le client ne peut réserver qu'un équipement à la fois. Ce mode de fonctionnement est logique puisque par principe un client ne peut pratiquer qu'une activité à la fois.

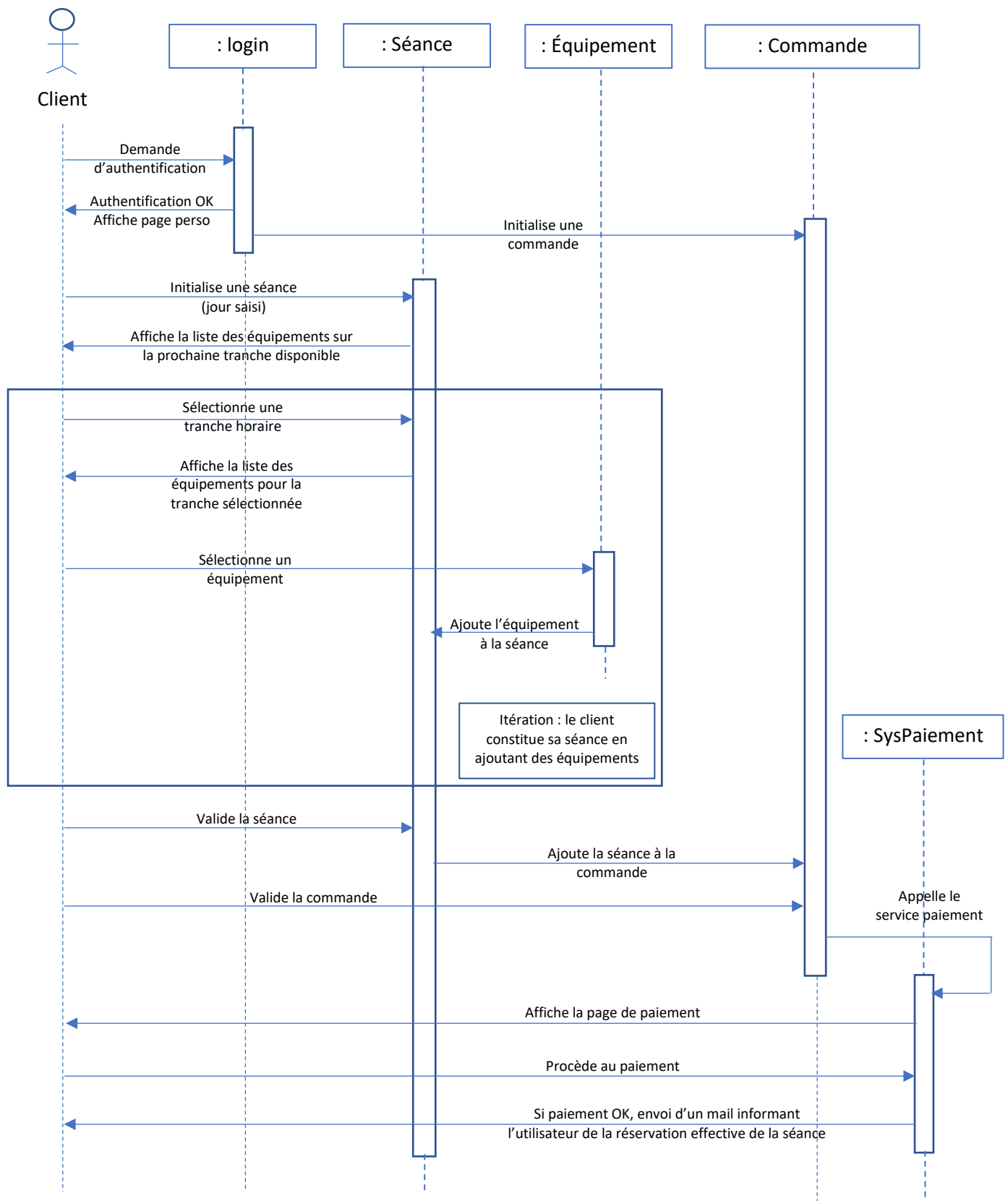
Si une séquence d'interruption volontaire (je m'abstiens de toute activité pendant 10' en ne sélectionnant aucun d'équipement entre deux tranches horaires) ou non (il n'y avait plus d'équipement disponible sur telle tranche horaire) intervient dans la programmation de ma séance, je ne serai pas facturé.

- Le contenu de la séance (heures et équipements sélectionnés) doit en permanence être accessible visuellement.
- Quand un équipement est sélectionné pour une tranche horaire, les autres équipements ne peuvent plus être sélectionnés.
- A tout instant, le client peut enlever de la programmation de sa séance un équipement préalablement sélectionné. Les équipements disponibles pour cette tranche horaire seront alors de nouveau visibles et sélectionnables
- Le client procède ou non à l'ajout d'autres équipements pour d'autres tranches horaires.
- Si le client veut programmer une séance pour un jour différent, il doit d'abord valider la séance du jour ou l'annuler.
- Le client peut programmer plusieurs séances par jour. Pour chaque nouvelle séance, il sera informé si des équipements ont déjà été réservés lors du balayage des tranches horaires.
- Une fois établi le programme de sa séance, le client la valide. La séance est alors ajoutée au panier d'achat.
- Enfin, le client passe à l'étape de paiement en ligne. Si celui-ci est validé, le client recevra un mail confirmant la réservation effective de la séance.
- Le client peut accéder ensuite aux feuilles de routes détaillant le contenu des séances programmées et validées.



### 2.2.3 Diagramme de séquence

Le diagramme de séquence suivant décrit le scénario nominal de la programmation d'une séance par un utilisateur jusqu'à son paiement en ligne :



## 2.3 Package manager

Ce package est destiné aux managers de « Smart Fitness ». Il se compose de trois grandes fonctionnalités : la gestion du parc des équipements, celle des offres relatives aux abonnements et aux montres connectées et la fonctionnalité de pilotage opérationnelle.

### 2.3.1 La gestion du parc des équipements

La gestion du parc des équipements inclue la localisation d'un équipement dans une salle, son appartenance à une famille d'équipements et des éléments les caractérisant. Ces éléments concernent :

- Le prix d'achat de l'équipement
- Le tarif d'utilisation pour une prestation de 10'
- Une description destinée à le présenter aux utilisateurs
- Une photo représentant l'équipement.
- La possibilité d'ajouter des tickets d'intervention de maintenance

### 2.3.2 La gestion des offres

Les offres concernent :

- Les abonnements : les managers pourront créer et paramétrer des formules d'abonnements ; les paramètres étant le nom, le tarif et la durée de l'abonnement.
- Un catalogue de produits : Les managers pourront ajouter et proposer des produits (par exemple des modèles de montres connectées, des boissons énergisantes, des produits d'alimentation). Les informations saisies concerneront les nom, prix, description et photo de l'article.

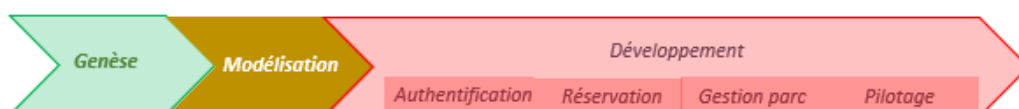
### 2.3.3 Le pilotage opérationnel de l'activité

Les fonctionnalités de pilotage se matérialiseront par l'édition de tableaux graphiques représentant :

- Une synthèse mensuelle glissante pour comparer le taux de réservation au cours des deux dernières années.
- La balance des revenus et dépenses pour chaque équipement.

## 2.4 Package Admin



Le package Admin est destiné à la gestion des comptes utilisateurs du staff de « Smart Fitness ». Seul l'administrateur du site sera habilité à créer et gérer les comptes des collaborateurs de « Smart Fitness ».



## 2.5 Prototypes d'interfaces (« Wireframes »)

Dans cette section nous proposons de présenter quelques prototypes d'interfaces dans le but de nous aider à définir une identité visuelle et ergonomique du site.


### 2.5.1 Wireframe « Liste des équipements disponibles » (User case « Réserver une séance »)

Laurent


Week 01, 2017

07 : 23


Espace musculation



Muscle Device 1  
☐ Réserver




Muscle Device 2  
☐ Réserver




Muscle Device 3  
☐ Réserver


Espace cardio-training



Running Trainer 1  
☐ Réserver



Running Trainer 2  
☐ Réserver



Running Trainer 3  
☐ Réserver

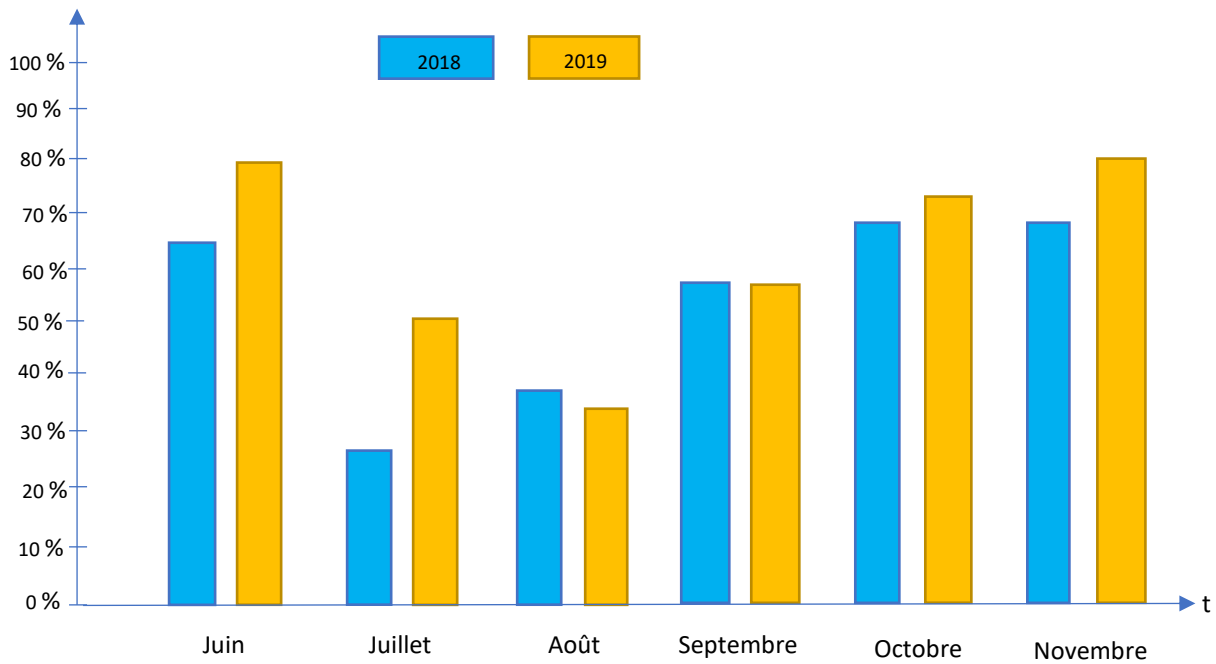
Submit

### 2.5.2 Wireframe « Tableaux évolution du taux de réservation & rendement par équipement » (fonctionnalité pilotage opérationnel de l'activité)

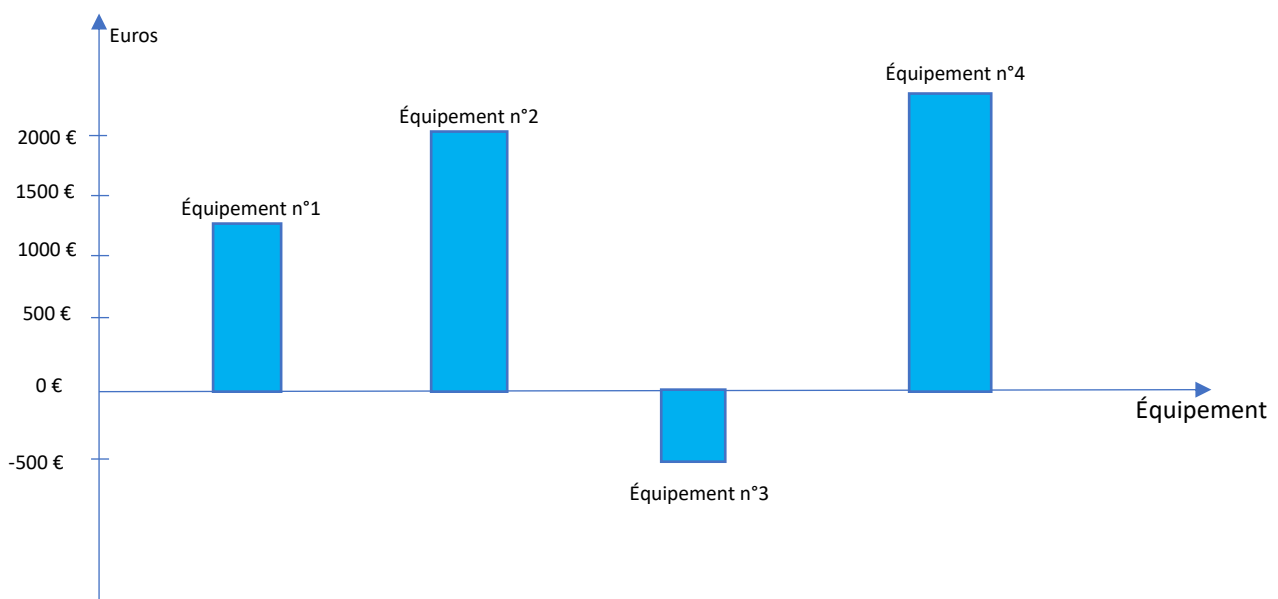


Manager

#### Evolution du taux de réservation 2018 - 2019



#### Balance revenus / dépenses par équipement



## 2.6 Gestion des utilisateurs et des accès

Il sera mis en œuvre une gestion d'attribution de rôle en fonction du statut de l'utilisateur. Le tableau ci-dessous dresse les correspondances entre le statut d'un utilisateur, son rôle et les droits d'accès qu'il procure :

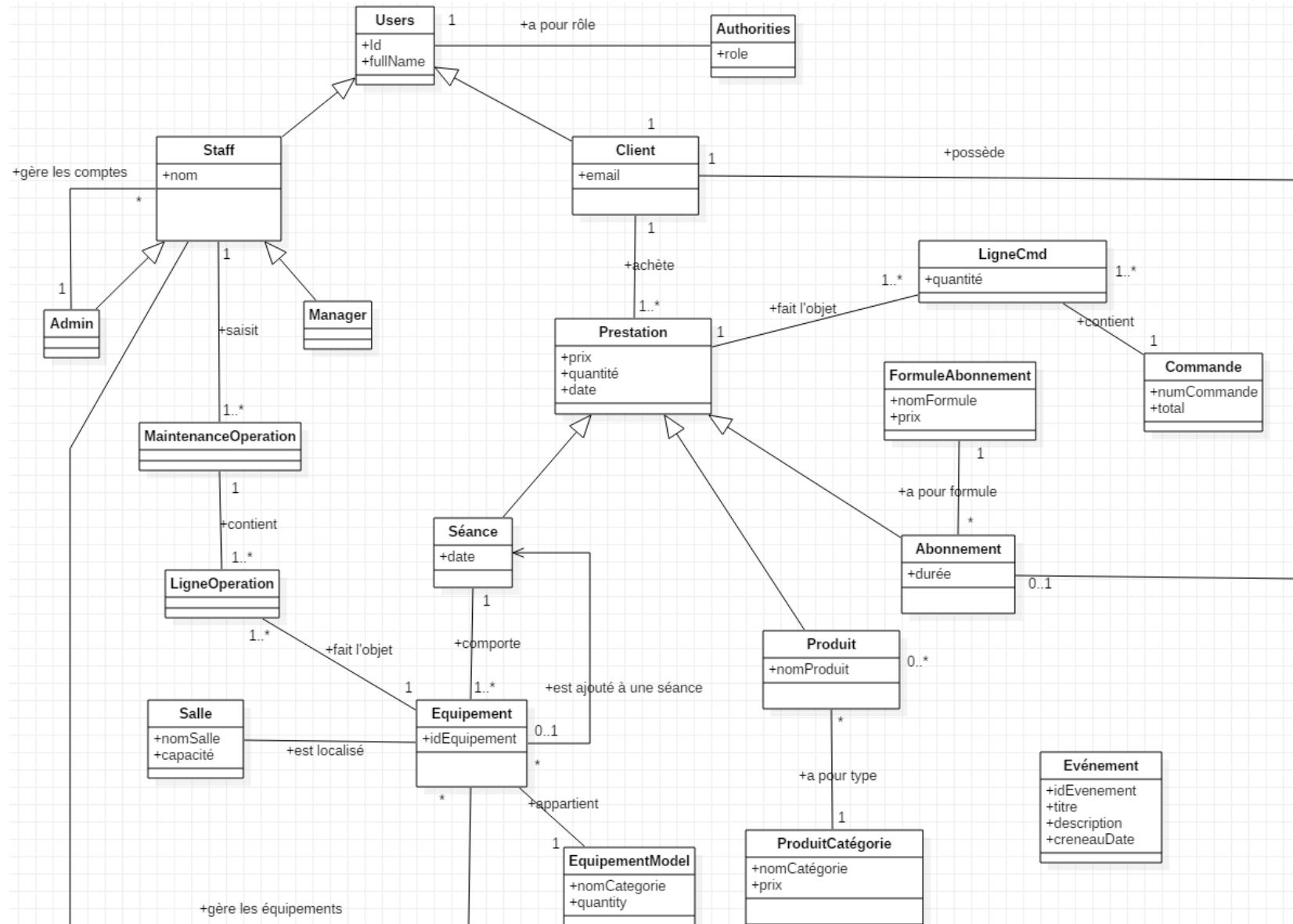
Utilisateur	Rôle	Droits d'accès
<i>Web internaute</i>		
Client non connecté	ANONYMOUS	<ul style="list-style-type: none"><li>- Page d'accueil (home page)</li><li>- Page d'information et de contact</li><li>- Page d'inscription (signUp)</li><li>- Page de login (signIn)</li></ul>
Client connecté	CUSTOMER	<ul style="list-style-type: none"><li>- ANONYMOUS +</li><li>- Pages du module de réservation de séance</li><li>- Pages du module offre (abonnements et catalogue des produits)</li><li>- Pages de suivi des commandes et feuilles de route des séances.</li></ul>
Gestionnaire	MANAGER	<ul style="list-style-type: none"><li>- Pages de gestion du parc équipements</li><li>- Pages de gestion des offres</li><li>- Pages du pilotage opérationnel</li></ul>
Administrateur	ADMIN	<ul style="list-style-type: none"><li>- MANAGER +</li><li>- Pages de gestion des comptes</li></ul>

Ce tableau se traduira dans notre diagramme de classes (voir section suivante) par une table 'Authorities' associé à une table 'Users'.





## 2.7 Diagramme de classes (MOO, Modèle Orientée Objet)



### 2.7.1 Diagrammes de classes – commentaires

Le diagramme comporte deux héritages :

- L'un se réfère à la classe '*Users*' et donne lieu à deux branches : celle relative au '*Staff*' rassemblant les profils administrateurs et managers du site et celle relative au '*Client*' qui fait référence à l'ensemble des mobinautes et internautes.

- L'autre concerne la classe '*Prestation*' qui traduit un acte d'achat d'un client web se décline en trois sous-classes :

1. '*Séance*', qui traduit le fait qu'un client veut se constituer une séance comportant une ou plusieurs activités chacune se déroulant sur un équipement référencé dans la classe '*Equipement*' (le lien récursif entre les deux classes permet d'indiquer que le client peut au fur et à mesure ajouter un autre équipement dans la programmation de sa séance) ;

2. '*Produit*' qui traduit le fait qu'un client a fait le choix d'acheter un article du catalogue.

3. '*Abonnement*' qui traduit le fait qu'un client souhaite souscrire à une formule d'abonnement.

Notes : Le principe de l'héritage mis en œuvre ici permettra d'étoffer le catalogue des prestations proposées en étendant la classe '*Prestation*' à d'autres catégories d'articles non encore implémentées à ce jour.

Le diagramme comporte deux relations « Many to Many ». Afin d'implémenter ce type de relation, il est nécessaire d'ajouter des tables intermédiaires pour obtenir des relations de types

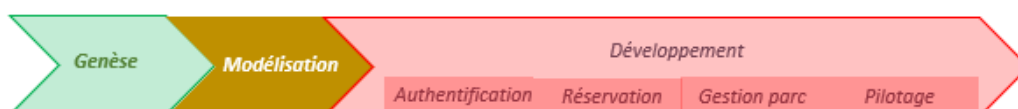
Table A 1 -> n Table Intermédiaire n ->1 Table B :

- La relation *Prestation* – *Commande* met en œuvre la table intermédiaire '*LigneCmd*'.
- La relation *MaintenanceOpération* – *Equipement* met en œuvre la table intermédiaire '*LigneOpération*'.

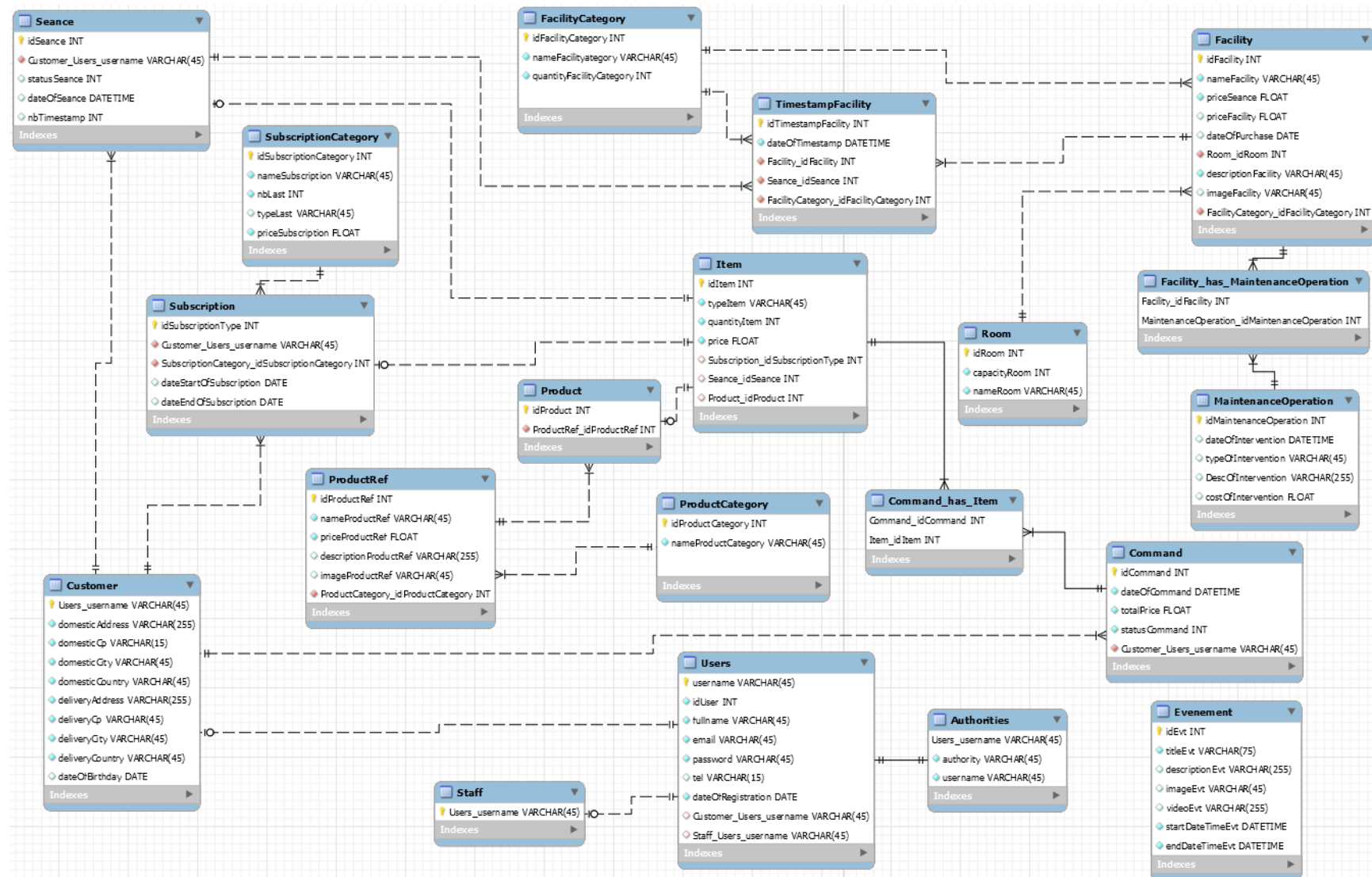
Le diagramme comporte trois relations « One to Many ». Dans le cas présent elles ont pour but de matérialiser l'appartenance d'un ensemble d'objets de même nature à une catégorie. Il en est ainsi pour les relations :

- *Equipement* – *ModèleEquipement*, (plusieurs équipements de même nature appartiennent à un modèle d'équipement).
- *MontreConnectée* – *ModèleMontreConnectée*, (plusieurs montres de même nature appartiennent à un modèle de montre connectée).
- *Abonnement* – *FormuleAbonnement*, (plusieurs abonnements de même type sont catégorisés par une formule d'abonnement).

Le Modèle Logique de Données (MLD) présenté ci-après est la traduction sous forme de tables de bases de données notre diagramme de classes



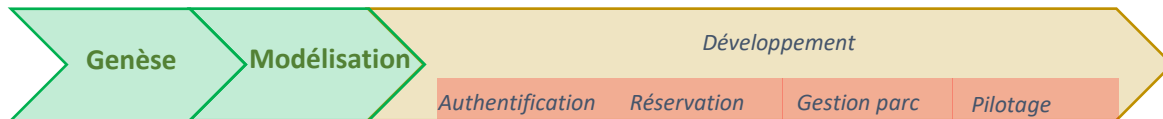
## 2.8 Le Modèle Logique de Données (MLD)



## 2.9 Le Modèle Physique de Données (MPD)

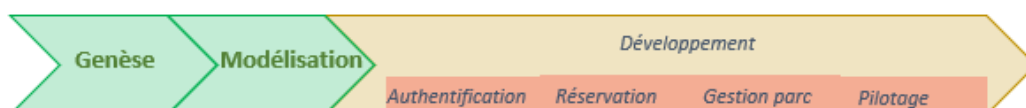
Le modèle physique de données est la transcription sous forme de script SQL du modèle logique de données. L'ensemble de ces scripts se trouvent en annexe 1.

Les sections qui suivent seront consacrées à l'architecture et à l'implémentation des différents services de l'application.



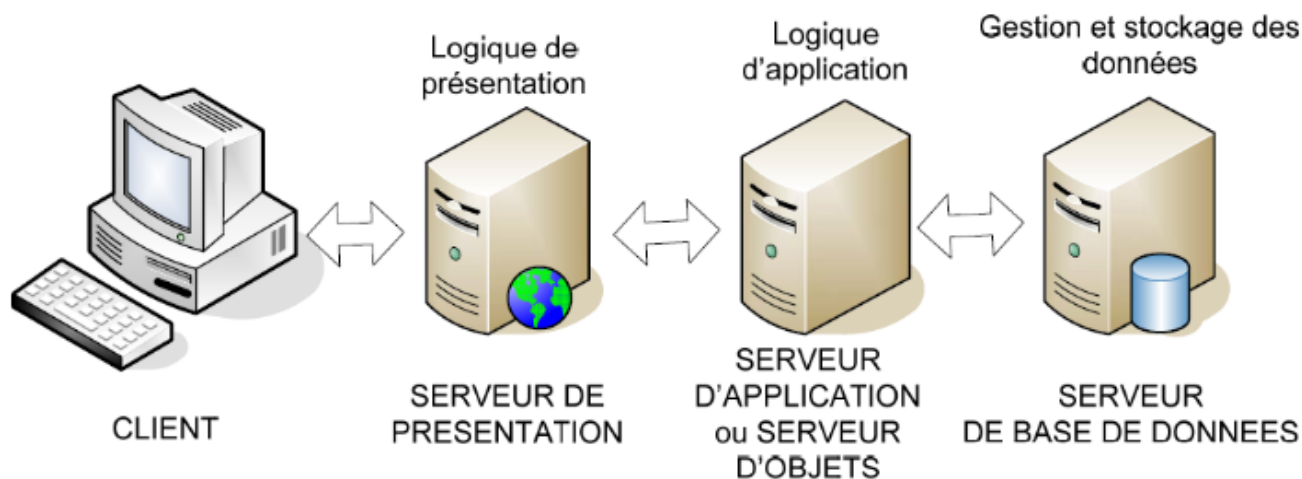
\*

\* \*



## 3 – Développement

### 3.1 Architecture de l'application



L'architecture logicielle de l'application « Smart Fitness » repose sur une architecture multi-tiers déclinée en 4-Tier dans lequel chacune des trois couches applicatives (logique de présentation, logique d'application, gestion et stockage des données) tourne sur un serveur distinct (respectivement un serveur de présentation, un serveur d'application et un serveur de base de données). Ce type d'architecture facilite une répartition de la charge entre tous les niveaux et contribue à la réutilisation des développements.

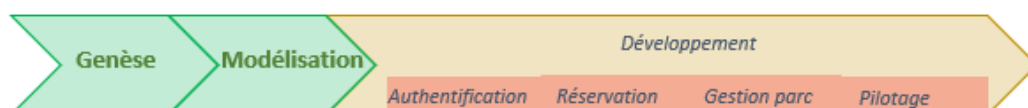
Les sections qui suivent sont consacrées à la présentation de l'implémentation de chacun des trois serveurs mis en œuvre dans le cadre du projet.

#### 3.1.1 Le serveur de présentation

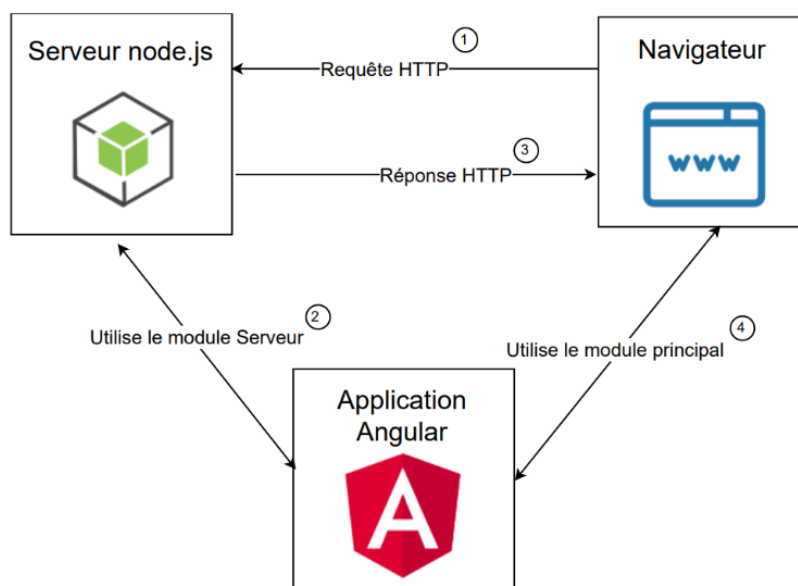
Le serveur de présentation repose sur le framework Angular qui permet de réaliser des applications de type « Single Page Application » et Node.js qui est utilisé comme plateforme de serveur Web (<http://localhost:4200>)

Une SPA est une Single Page Application. Il s'agit d'avoir une seule page où les données et les vues sont rechargées par JavaScript au lieu de faire des appels au serveur pour recharger les pages. On a donc les mécanismes suivants :

- Chargement des données de l'application via une API REST de manière asynchrone (c'est ce qui est exécuté côté serveur d'application)
- Modification du DOM (Domain Object Model) lorsque l'on souhaite modifier la vue (cela se passe côté client)

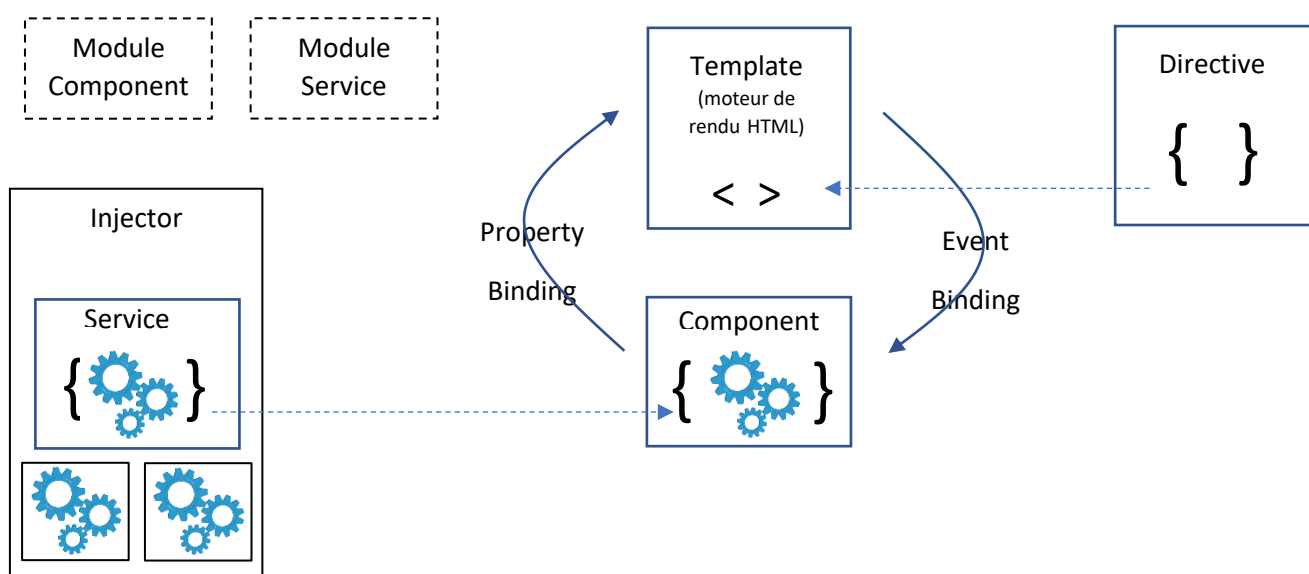


Ci-dessous un schéma explicatif des cheminement et traitement des requêtes HTTP (Source : <https://blogs.infinitiesquare.com/posts/web/rendu-cote-serveur-d-angular-part-1-3>)



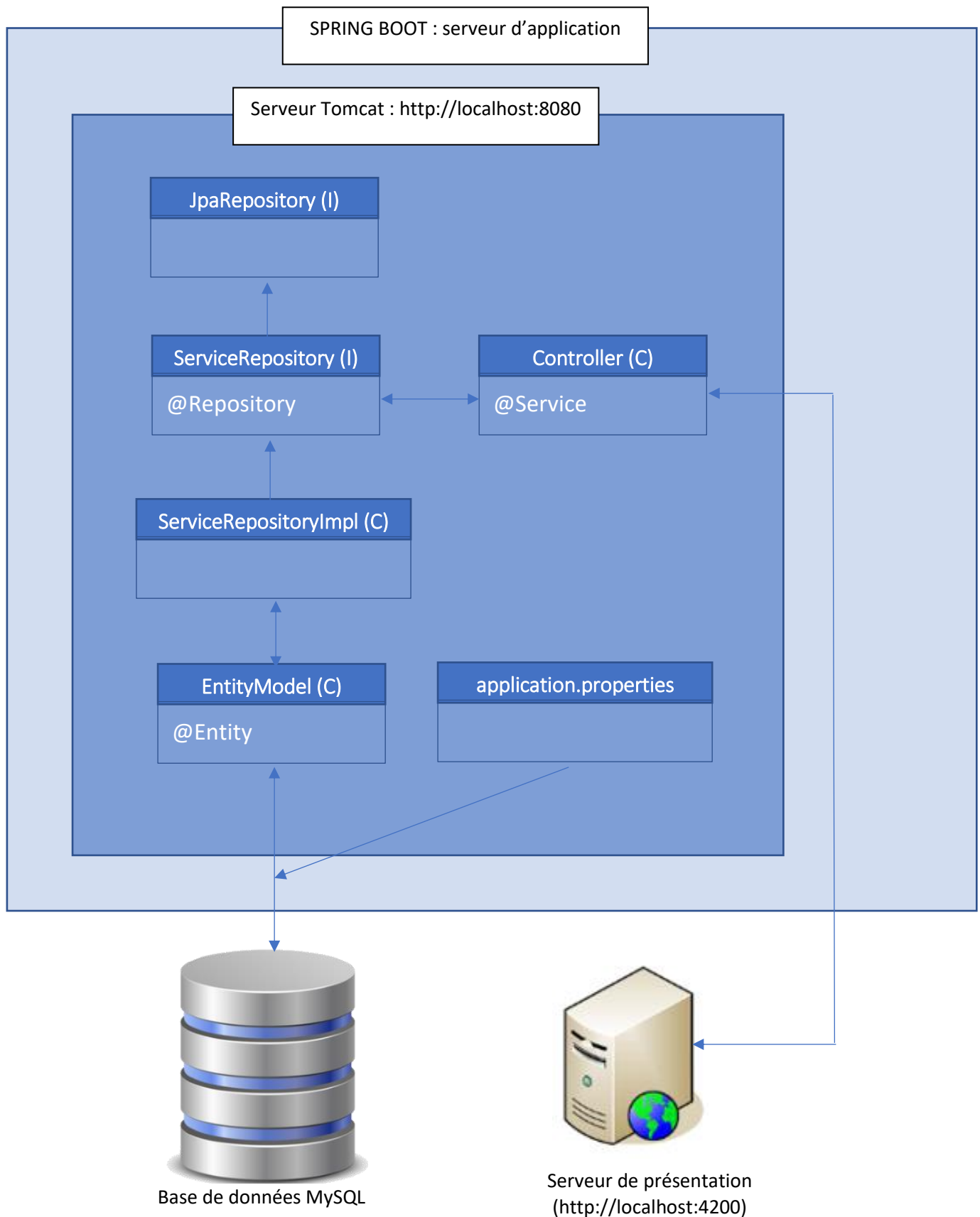
Explication : Le navigateur envoie une requête HTTP au serveur (1). Ensuite, le serveur Node.JS exécute l'application Angular en utilisant le module créé à cet effet (2). Lors de cette étape, le serveur utilise l'application pour générer l'HTML correspondant à la requête fournie par le navigateur. Une fois généré, le serveur retourne cet HTML au navigateur qui va pouvoir l'afficher tel quel (3). Enfin, le navigateur va charger l'application en utilisant son module principal (4).

En ce qui concerne une application Angular, les principaux blocs de construction sont les modules, les composants, les modèles, la liaison de données, les directives, les services et l'injection de dépendance. (Source : <https://fr.wikipedia.org/wiki/Angular>) [libre de copier]



### 3.1.2 Le serveur d'application

Le serveur d'application est implémenté sous Spring Boot qui est un framework permettant la mise en œuvre d'un projet Spring et tournant sur un serveur Apache tomcat (<http://localhost:8080>)



Explication du schéma : Parmi les modules mis à la disposition d'une application tournant sous SpringBoot, Spring Data JPA (Java Persistence API) fournit une implémentation de la couche d'accès à une base de données via l'interface **JpaRepository**. Pour notre projet, nous implémenterons cette interface sous la forme de services ( **ServiceRepository** - **ServiceRepositoryImpl** ), permettant la communication de la couche modèle( **EntiyModel** ) avec la couche controller ( **Controller** ). Quant au fichier **application.properties**, il contient entre autres la configuration des paramètres de connexion à la base de données (voir section 3.1.3)

Le principe mis en œuvre par une application implémentant la couche JPA repose sur le mapping relationnel objet (ORM), c'est-à-dire sur la correspondance entre les entités du modèle objet et les tables de la base de données relationnelle. Pour notre projet, nous utiliserons le système des annotations (voir section 3.2.2) pour la réalisation effective du mapping. En outre, en plus des méthodes d'accès de type CRUD (Create – Read – Update – Delete) proposées par l'interface JpaRepository, nous écrirons dans nos services qui étendront cette interface (« **extends JpaRepository** ») des requêtes personnalisées de type JPQL (Java Persistence Query Langage) et natives SQL.

### 3.1.2 Le serveur de base de données

Nous utiliserons MySQL comme base de données pour gérer et stocker les données. Nous l'utiliserons également pour mettre en œuvre des procédures stockées et des triggers.

Add-on MySQL : L'outil graphique '*Workbench*' de MySQL nous a permis de concevoir le modèle logique de données (section 2.8) puis de générer le fichier **sql** correspondant au modèle physique de données (section 2.9 et annexe 1).

### 3.1.3 Le fichier application.properties

La section aborde deux points concernant la configuration de la connexion à la base de données depuis le serveur d'application au niveau du fichier application.properties.

- Après avoir créé un compte utilisateur standard MySQL dédié à la connexion à la base de données, le fichier application.properties doit y faire référence. Or le paramétrage suivant

```
spring.datasource.url=jdbc:mysql://localhost:3306/db_fitness?useSSL=false
spring.datasource.username=fitness
spring.datasource.password=Colis062019!
```

génère l'erreur suivante :

Access denied for user 'fitness'@'localhost' (using password: NO)

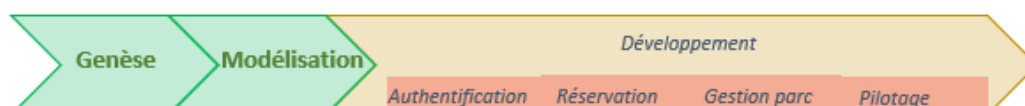
La soumission de ce problème sur internet nous a permis d'obtenir les éléments de réponse suivants sur <https://stackoverflow.com/questions/52174740/java-sql-sqlexception-access-denied-for-user-localhost-using-password-no>

Just do this in the properties file in the properties file: Change this

```
spring.datasource.url=jdbc:mysql://localhost:3306/taskdb?useSSL=false
spring.datasource.username=springuser
spring.datasource.password=1Qazxsw@
```

To this:

```
spring.datasource.url=jdbc:mysql://localhost:3306/taskdb?user=springuser&password=1Qazxsw@
```





L'adaptation de la solution proposée à notre problème a donc abouti à la configuration suivante de `application.properties` :

```
spring.datasource.url=jdbc:mysql://localhost:3306/db_fitness?useSSL=false&user=fitness&password=Colis062019!
```

```
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

- Au lancement du serveur d'application (<http://localhost:8080>), j'exécute le script `data.sql` pour injecter des données en base ainsi qu'un trigger associé à la table `timestamp_facility`. Or si l'injection de requêtes SQL de type INSERT ne pose pas de problème, il n'en est pas de même avec la création de procédures stockées et triggers. Une des solutions préconisées (source [stackoverflow](https://stackoverflow.com/questions/44110000/spring-boot-mysql-database-initialization-error-with-stored-procedures) => Spring Boot MySQL Database Initialization Error with Stored Procedures) consiste à rajouter dans le fichier `applications.properties` la ligne suivante :

```
spring.datasource.separator=^;
```

Ce qui se traduit par la nécessité de terminer chaque instruction SQL du fichier `data.sql` par `^;`

Extrait de mon fichier `data.sql` :

```
insert into product_ref (product_category_id_product_category, name_product_ref, price_product_ref, image_product_ref, description_product_ref) values (2, "Ovotamine", 5, "Ovotamine_ovotamine.jpg", "")^;
```

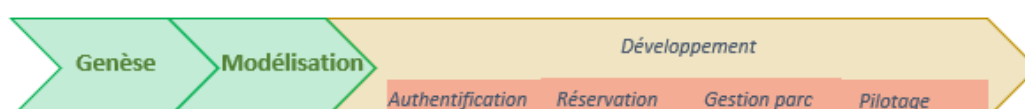
```
DROP FUNCTION IF EXISTS func_count_timestamp^;
CREATE FUNCTION func_count_timestamp(dateTimeOfSeance DATETIME, idFacility INT)
RETURNS int
BEGIN
    DECLARE nb INT;
    SELECT COUNT(*) INTO nb FROM db_fitness.timestamp_facility WHERE
date_of_timestamp=dateTimeOfSeance AND facility_id_facility=idFacility;
    RETURN nb;
END^;
```

```
CREATE TRIGGER triggerTimestamp BEFORE INSERT ON timestamp_facility
FOR EACH ROW
BEGIN
    IF `func_count_timestamp`(NEW.date_of_timestamp,
NEW.facility_id_facility) > 0 THEN
        signal sqlstate '45000';
    END IF;
END^;
```

## 3.2 Mise en place de l'environnement de développement

### 3.2.1 Les outils de développement

Nous avons utilisé la plate-forme de Eclipse pour faire tourner le serveur d'application SPRING BOOT qui héberge le serveur *tomcat* (<http://localhost:8080>), et l'IDE VS-Code conjointement avec le CLI (Command Line Interface) Angular pour développer et faire tourner le serveur de présentation `node.js` accessible par l'adresse <http://localhost:4200>. La communication entre les deux serveurs étant basée sur des l'échange de messages REST (REpresentational



State Transfer) pour effectuer des interrogations de type GET, POST, PUT, DELETE (lecture, ajout, mis à jour et suppression de données). Pour le serveur de base de données mysql, la version utilisée est la suivante : (informations obtenues par la commande `mysqld -version`, le 'd' signifiant que l'on se réfère au serveur et non au client)

mysqld Ver 5.7.26-0ubuntu0.18.04.1 for Linux on x86\_64 ((Ubuntu))

### 3.2.2 Le jeu des annotations

Le modèle objet a été généré à partir du script **sql** du modèle physique de données (section 2.9) via l'outil JPA Project qui s'appuie sur EclipseLink 5.2 pour réaliser le processus de mapping ORM. La technique de mapping utilisée pour établir la correspondance entre les entités objets et les tables relationnelles repose sur les annotations. Cette approche consiste à faire précéder chaque champ des entités par une annotation du genre `@Column`. Nous pouvons classer les annotations en deux grandes catégories : celles relatives à la définition intrinsèque d'un champ et celles relatives au type de relation qu'entretiennent deux champs de deux entités différentes.

En ce qui concerne les informations propres à un champ, nous avons principalement utilisés les annotations suivantes :

`@Id`

`@GeneratedValue(strategy = GenerationType.IDENTITY)`

Ces deux annotations permettent de définir la clé primaire de la table et de positionner la propriété `AUTO_INCREMENT` à true, ce qui permet une gestion automatique par la base du séquençage incrémentiel des identifiant.

`@Column(unique=true, name= "nameColumn")`

Cette annotation permet d'apporter des informations complémentaires concernant un champ, comme de préciser que les valeurs d'une colonne sont tous uniques ou d'attribuer un nom de colonne différent de celui de son corollaire en base.

`@Temporal`

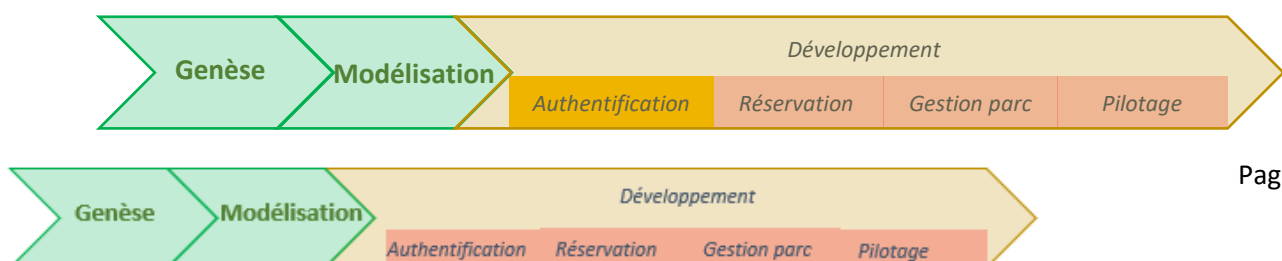
Cette annotation permet la gestion des données temporelles. Nous les avons utilisées pour les la gestion des dates et des heures des séances constituées par les clients.

`@Inheritance`

Cette annotation permet de mettre en œuvre l'héritage entre entités. Nous avons utilisé cette annotation pour l'entité *Item* pour ajouter un niveau d'abstraction au niveau des articles contenus dans une commande. Il sera ainsi facile d'étendre l'entité *Item* à d'autres catégories d'articles que celles déjà en œuvre( les séances, les abonnements et les produits du catalogue). Une deuxième utilisation de l'annotation `@Inheritance` concerne l'entité *User* qui sert d'entité de base aux entités *Customer* et *Staff* (représentant respectivement les clients du site et les collaborateurs du centre). Dans ce dernier cas, le principe de l'héritage permet de factoriser les données communes aux entités *Customer* et *Staff* comme le nom, l'identifiant et le mot de passe.

Pour les annotations relationnelles, nous les présenterons au cas par cas dans les phases de développement.

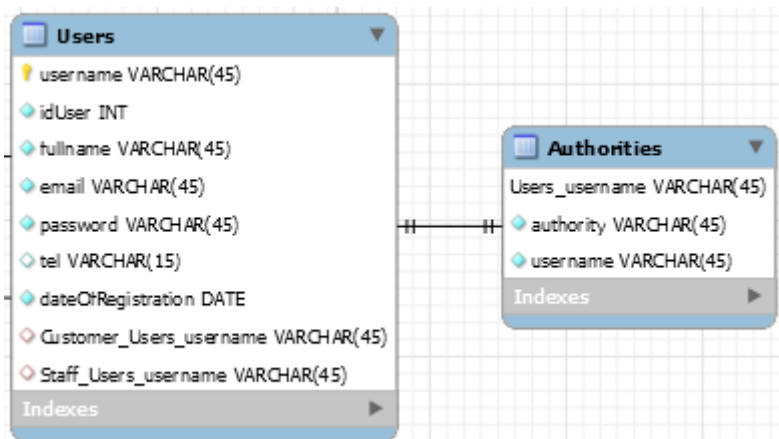
Dans les sections suivantes, nous allons aborder le module d'authentification.



## 4. Module d'authentification

### 4.1 Mise en base des données utilisateurs

Le module d'authentification de notre application repose sur deux tables et donc sur deux entités au sens objet conformément au mapping ORM : `users` et `authorities`.



La première table, '`users`', contient des informations générales relatives à un utilisateur (qu'il soit un client ou un collaborateur du staff Smart Fitness) comme son nom, son identifiant (`username`), son email, son mot de passe ; l'identifiant étant la clé primaire de la table. La seconde table, '`authorities`', contient deux champs : `username` qui est joint par relation au champ `username` de la table '`users`' et un

second champ `authority` correspondant au rôle de l'utilisateur. Pour rappel (section 2.6 Gestion des utilisateurs et des accès), selon son statut, un utilisateur se voit attribuer l'un des quatre rôles suivants : ANONYMOUS, CUSTOMER, MANAGER ou ADMIN.

Comme il en vient d'être précisé, les deux tables sont reliées par le champ '`username`'. La nature de la relation est ici @One-to-One. Comme ce champ représente la clé primaire côté '`users`', ce même champ représentera la clé étrangère côté '`authorities`'. En termes d'annotations, cela se traduit dans l'entité '`users`' par les déclarations suivantes :

```
@OneToOne (cascade=CascadeType.REMOVE)
@JoinColumn (name="username")
protected Authority authority;
```

L'option (`cascade=CascadeType.REMOVE`) de l'annotation @OneToOne a pour effet de supprimer à la fois dans les tables '`users`' et '`authorities`' le tuple pour un `username` donné. L'annotation @JoinColumn permet de déclarer le champ passé en paramètre de la propriété `name` (`name="username"`) comme champ de jointure.

Quant à l'entité reliée '`authorities`', elle comporte les déclarations suivantes :

```
@OneToOne (mappedBy="authority")
private User user;
```

La propriété (`mappedBy="authority"`) de l'annotation @OneToOne permet de mettre en place le concept de champ inversé, c'est-à-dire que la relation ne se trouve pas directement dans l'entité principale ('`users`') mais au sein de la deuxième entité ('`authorities`').

## 4.2 Mise en place de la politique de sécurité

### 4.2.1 La gestion de l'authentification

L'authentification repose sur la brique SpringSecurity de SpringBoot qui utilise l'interface *AuthenticationManager* et sa méthode *authenticate* pour établir l'identité de l'utilisateur cherchant à se connecter à l'application.

Ci-dessous, un extrait de la classe UserController implémentant l'interface *AuthenticationManager* :

```
Authentication authentication = authenticationManager.authenticate(  
    new  
    UsernamePasswordAuthenticationToken(user.getUsername(), user.getPassword()));  
    return ResponseEntity.ok(jwtTokenProvider.generateToken(authentication,  
user, this.userService));  
}
```

Ici, la classe *UsernamePasswordAuthenticationToken* sert à transmettre les « credentials » à l'*authenticationManager* c'est-à-dire l'identifiant et le mot de passe, permettant ainsi à Spring Security de procéder à l'authentification proprement dite.

En ce qui concerne l'encodage du mot de passe, Spring Security s'appuie sur l'algorithme de hachage BCrypt (algorithme de chiffrement symétrique plus évolué que MD5, SHA1 ou SHA256) pour générer des mots de passe cryptés sur une soixantaine de caractères. Ci-après un exemple extrait de la table 'users' :  
{bcrypt}\$2a\$10\$woFD.JoUP44f4iyS0YLywO5TLT4xabSvFZF9T4NEwhcGLmjGkKsOe

Au final, si l'authentification échoue, une exception est levée et nous renvoyons via notre classe *JwtAuthenticationEntryPoint* le code d'erreur 401, sinon le programme retourne à l'application appelante (l'application tournant localhost://4200, autrement dit le serveur de présentation node.js) un token dont la signature algorithmique correspond à une clé hachée de type HS512 dans notre cas.

### 4.2.2 La gestion des tokens

Comme notre application fonctionne en mode « stateless », chaque requête en provenance du serveur <http://localhost:4200> et envoyée vers le serveur <http://localhost:8080> doit inclure un bloc d'informations (situé dans le header de la requête) permettant d'identifier l'utilisateur initiateur de cette requête. Afin de sécuriser ces échanges, l'authentification est basée sur l'échange de tokens JWT (Json Web Token).

Principe d'un token JWT (source : <https://www.grafikart.fr/tutoriels/json-web-token-presentation-958>)

Un token JWT se compose de trois chaînes de caractères séparées par un « . » :

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
eyJzdWIiOiIxMjMONTY3ODkwiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0Ij0iOnRydWV9.  
TjVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```



- La première partie, l'**en-tête**, est un JSON encodé en base64 qui décrit le type de token et l'algorithme utilisé pour la signature.
- La seconde partie, le **payload**, est aussi un JSON encodé en base64 qui contient les informations à transmettre. Ce JSON peut contenir des clefs prédéfinies (appelées **claims**) qui permettent de donner des informations supplémentaires sur le token (comme la date d'expiration, la cible, le sujet...)
- La dernière partie, la **signature**, est la partie la plus importante du token JWT car elle permet de s'assurer de l'authenticité du token. Cette signature est générée à partir des 2 premières clefs avec un algorithme particulier (HS512 avec une clef secrète dans le cas de notre application).

Dans le cas de notre projet, nous implémentons la génération de tokens dans la classe JwtTokenProvider :

```

**
* Génération du token
* @param authentication
* @param user
* @param userService
* @return
*/
public AuthToken generateToken(Authentication authentication, User user,
UserService userService) {

    user = userService.findByUsername(user.getUsername()); // pour
    récupérer l'id

    SecurityContextHolder.getContext().setAuthentication(authentication);

    //User user = (User)authentication.getPrincipal();
    Date now = new Date(System.currentTimeMillis());
    Date expireDate = new Date(now.getTime() + TOKEN_EXPIRATION_TIME);
    Map<String, Object>claims = new HashMap<>();
    claims.put("id", (Long.toString(user.getIdUser())));
    claims.put("username", user.getUsername());
    claims.put("role", authentication.getAuthorities());
    String jwt = TOKEN_PREFIX + Jwts.builder()
        .setSubject(user.getUsername())
        .setClaims(claims)
        .setIssuedAt(now)
        .setExpiration(expireDate)
        .signWith(SignatureAlgorithm.HS512, SECRET_KEY)
        .compact();
    return new AuthToken(jwt);
}

```

Dans le payload nous intégrons dans la HashMap **claims** les clés suivantes : "id", "username", "role". Nous pouvons ainsi récupérer le rôle de l'utilisateur c'est-à-dire son authority définissant ses autorisations. La gestion des autorisations est abordée à la section suivante.



### 4.2.3 La gestion des autorisations

En fonction de son rôle, un utilisateur peut accéder ou non aux services d'une application (les ressources URI pour une application web). Dans le cas présent, la gestion des autorisations s'apparente à une gestion des routes circonscrites aux rôles. Dans Spring Security, cette gestion se fait dans la méthode `configure()` de la classe `WebSecurityConfig` dont voici un extrait de notre implémentation :

```
@Override
protected void configure(HttpSecurity http) throws Exception {

    http.cors().and().csrf().disable()

    .exceptionHandling().authenticationEntryPoint(unauthorizedHandler).and()

        .sessionManagement()

        .sessionCreationPolicy(SessionCreationPolicy.STATELESS)

        .and()

        .headers().frameOptions().sameOrigin() //Pour activer la base de
donnees MySQL

        .and()

        .authorizeRequests()

        .antMatchers("/").permitAll()

        .antMatchers("/postman/**").permitAll()

        .antMatchers("/userctrl/newcustomer").permitAll()

        .antMatchers("/emailctrl/signupconfirm/**").permitAll()

        .antMatchers("/userctrl/authority/**").hasAnyRole("ADMIN",
"MANAGER", "CUSTOMER")

        .antMatchers(SIGN_UP_URLS).permitAll()

        .anyRequest().authenticated();

    http.addFilterBefore(jwtAuthenticationFilter(),
UsernamePasswordAuthenticationFilter.class);

}

}
```

L'obtention du rôle d'un utilisateur se fait lors de l'appel de

```
http.addFilterBefore(jwtAuthenticationFilter(),
UsernamePasswordAuthenticationFilter.class);
```



Puis dans la classe `jwtAuthenticationFilter`, `SecurityContextHolder` nous permet de récupérer et mettre à jour les informations relatives à une requête utilisateur. Nous nous en servons pour affecter le rôle effectif de l'utilisateur via la classe `SimpleGrantedAuthority` :

```
SecurityContextHolder.getContext().setAuthentication(authenticationToken);

typeRole = typeRole.toString().split("=")[1];
typeRole = typeRole.toString().substring(0,
typeRole.toString().length() - 1);

SimpleGrantedAuthority authority = new
SimpleGrantedAuthority(typeRole.toString());

List<SimpleGrantedAuthority> updatedAuthorities = new
ArrayList<SimpleGrantedAuthority>();

updatedAuthorities.add(authority);

SecurityContextHolder.getContext().setAuthentication(
    new UsernamePasswordAuthenticationToken(
SecurityContextHolder.getContext().getAuthentication().getPrincipal(),
SecurityContextHolder.getContext().getAuthentication().getCredentials(),
        updatedAuthorities)
);
```

Au final, en fonction de son rôle, un utilisateur aura le droit ou non d'accéder à une ressource.



### 4.2.3 La gestion des accès aux pages du site

La gestion de l'authentification et des autorisations s'effectue via le module Spring Security sur le serveur d'application en <http://localhost:8080> . Côté serveur de présentation en <http://localhost:4200>, nous implémentons un autre niveau de couche de sécurité basée sur les guards. Un guard est un service qui s'exécute lorsqu'un utilisateur essaye de naviguer vers une page du site. Il retourne une valeur booléenne indiquant si l'utilisateur est autorisé ou non à visualiser la page sur laquelle est appliquée le guard.

Pour mon projet, j'ai mis en œuvre quatre guard correspondant au rôle de l'utilisateur :

- CUSTOMER
- MANAGER
- ADMIN

+ un guard spécifique pour éviter à tout utilisateur ayant payé une commande de revenir en arrière lorsque son paiement a été validé.

Ci-dessous, un extrait de l'implémentation du guard relatif à un utilisateur de profil admin

```
export class AuthGuardAdminService implements CanActivate {  
  
  constructor(private loginService: LoginService,  
    private router: Router) {}  
  
  canActivate(  
    route: ActivatedRouteSnapshot,  
    state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {  
    if(this.loginService.isAuth && this.loginService.authority == 'ROLE_ADMIN') {  
      return true;  
    } else {  
      this.router.navigate(['/login']);  
    }  
  }  
}
```

Dans la section j'aborde la fonctionnalité permettant aux utilisateurs de se constituer une séance à partir de la sélection d'équipements par tranche de 10'.





## 5. La gestion de réservation d'équipements

C'est le module permettant aux utilisateurs de se constituer des séances.

### 5.1 Obtention de la liste d'équipements

Pour chaque tranche horaire de 10' pris entre 6h et 22h, l'utilisateur se verra proposer une liste d'équipements disponibles. Cette liste se présente sous la forme d'un composant graphique de type accordéon extensible (techniquement parlant un *mat-accordion* constitué de *mat-panel* pour chaque type d'équipement) :

Elliptique (dispo : 3 )	▼
Tapis roulant (dispo : 2 )	▼
Vélo (dispo : 1 )	▼

Sa source de données est le résultat de deux requêtes SQL

- Une pour recenser les équipements disponibles de chaque catégorie d'équipements pour une tranche horaire donnée :

```
@Query(value= "SELECT facility.* FROM facility INNER JOIN facility_category ON "  
    + " facility_category_id_facility_category = id_facility_category "  
    + " WHERE name_facility_category like ?1 AND id_facility NOT IN "  
    + " (SELECT facility_id_facility FROM timestamp_facility INNER JOIN "  
    facility_category ON "  
    + " facility_category_id_facility_category = id_facility_category "  
    + " WHERE date_of_timestamp like ?2 ) ", nativeQuery = true)  
List<Facility> findByFacilityAvailable(String facilityName, String  
timestampToString);
```

- L'autre pour obtenir le nombre d'équipements disponibles par catégorie d'équipements (ce qui correspond à l'indication *dispo* chiffrée entre parenthèses):

```
@Query(value = "SELECT (facility_category.quantity_facility_category) - "  
    + " (SELECT COUNT(*) FROM timestamp_facility INNER JOIN "  
    facility_category ON "  
    + " timestamp_facility.facility_category_id_facility_category = "  
    facility_category.id_facility_category "  
    + " WHERE facility_category.name_facility_category like ?1 AND "  
    timestamp_facility.date_of_timestamp like ?2 ) "  
    + " FROM facility_category WHERE "  
    facility_category.name_facility_category like ?1", nativeQuery = true)  
int findByFacilityCategoryCount(String nameFacilityCategory, String timestamp);
```



Pour disposer de ces informations au sein d'une même entité, j'ai créé dans le package modèle une classe adaptateur `FacilityAvailableAdaptater` qui n'a pas lieu d'être persistée en base, elle ne possède donc pas d'annotations JPA . Sa raison d'être consiste à collecter et mettre sous forme de composants d'accès aux données les résultats des deux requêtes SQL précédentes via un service implémentée de la manière suivante :

```
int availableFacilities = 0;

String nameFacilityCategory = "";

ArrayList<FacilityAvailableAdaptater> facilitiesAvailableAdaptater =
new ArrayList<FacilityAvailableAdaptater>() ;

List<Facility> facilities = null;

List<FacilityCategory> facilityCategories =
this.facilityCategoryRepo.findAll();

for (int i=0; i<facilityCategories.size(); i++) {
    nameFacilityCategory =
facilityCategories.get(i).getNameFacilityCategory();

    availableFacilities =
this.timestampFacilityRepo.findByFacilityCategoryCount(nameFacilityCategory,
timestampToString);

    facilities =
this.facilityRepo.findByFacilityAvailable(nameFacilityCategory,
timestampToString);

    facilitiesAvailableAdaptater.add(new
FacilityAvailableAdaptater(nameFacilityCategory, availableFacilities,
facilities));
}

return facilitiesAvailableAdaptater;
```

Le service retourne bien une instance de `facilitiesAvailableAdaptater`.



## 5.2 Constitution et validation d'une séance

### 5.2.1 Mise en œuvre côté « back-end » (serveur d'application)

Le fait qu'une séance peut être constituée d'un ou plusieurs équipements selon ce qu'aura sélectionné l'utilisateur implique une relation de type `@OneToMany` dans *Seance* et `ManyToOne` dans *TimestampFacility*. Ce qui se traduit dans *Seance* par une variable de type *List* contenant un ensemble de *TimestampFacility* d'une part,

```
//bi-directional many-to-one association to TimestampFacility

@OneToMany(mappedBy="seance", cascade=CascadeType.REMOVE)

@JsonManagedReference

private List<TimestampFacility> timestampFacilities;
```

La déclaration de la relation `@OneToMany(mappedBy="seance")` se fait par champ inverse, donc avec la propriété `mappedBy`. Quant à la propriété `cascade=CascadeType.REMOVE`, sa présence implique la suppression de la liste des *TimestampFacility* lorsque la *Seance* à laquelle ils sont associés se trouve supprimée.

et dans *TimestampFacility* par une variable scalaire de type *Seance* d'autre part.

```
//bi-directional many-to-one association to Seance

@ManyToOne

@JoinColumn(name="Seance_idSeance")

@JsonBackReference

private Seance seance;
```

Du fait que ce côté de la relation n'est relié qu'à une seule valeur (de type *Seance*), un champ *seance\_id\_seance* a été créé dans la table *TimestampFacility* de la base MySQL, ce champ pouvant être considéré comme une clé étrangère.

### 5.2.2 Mise en œuvre côté « front-end » (serveur de présentation).

L'ensemble des opérations de sélection / dé-sélection d'équipements à différentes tranches horaires doit pouvoir être effectué par l'utilisateur au sein d'un même espace visuel. La solution que j'ai retenue consiste à associer un composant parent à deux composants enfant : ainsi par le jeu des interactions, les différents composants s'échangent des messages, et donc de pouvoir de procéder à la réactualisation des données qu'ils affichent. Cette relation parent-enfant entre composants se traduit par la gestion de *router-outlet*:

- dans le module *app-routing* :

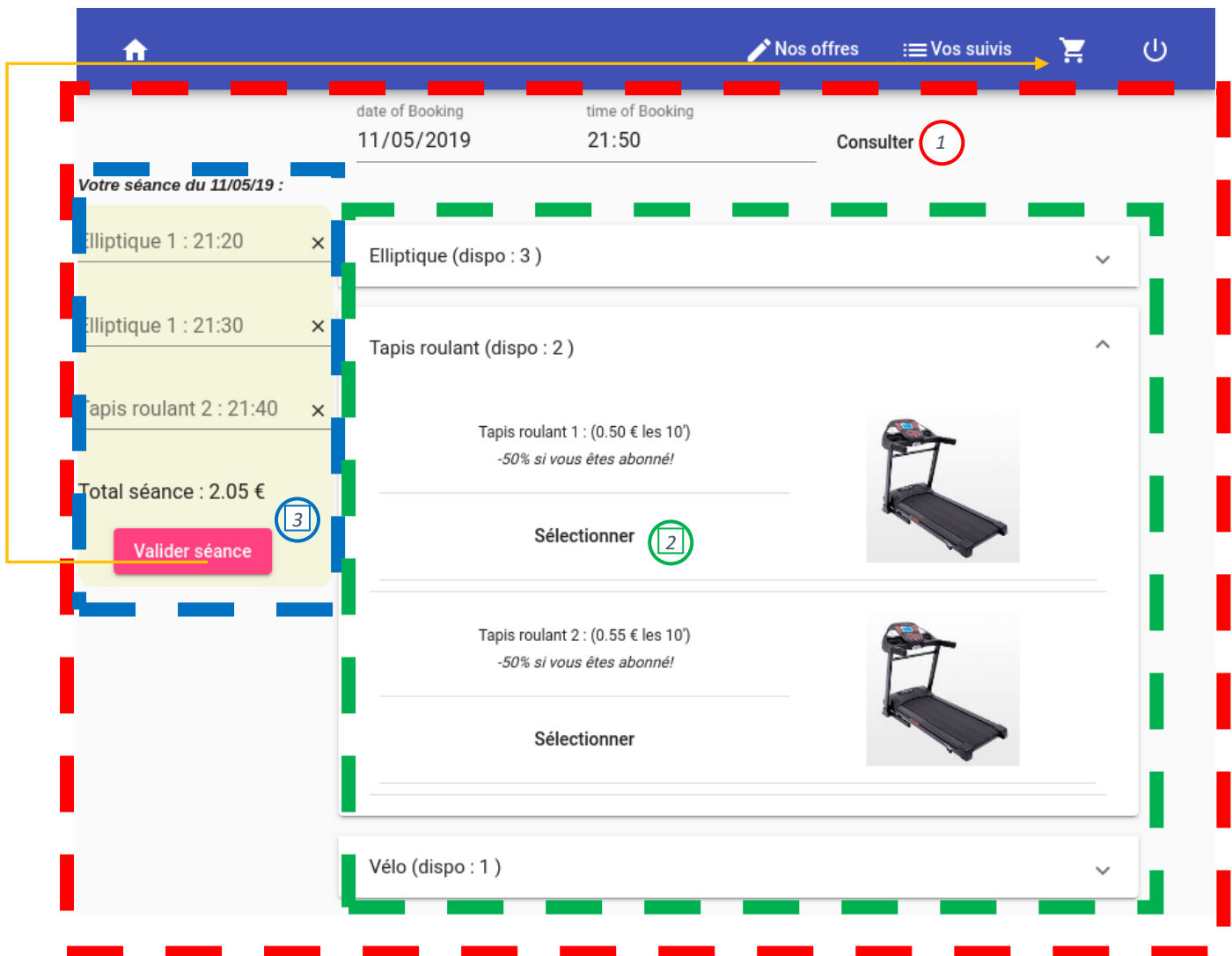
```
{ path: 'seance-booking', canActivate: [AuthGuardCustomerService], component:
SeanceBookingComponent, children: [
```

```
{ path: 'facility-category-booking', canActivate: [AuthGuardCustomerService], component:
FacilityCategoryBookingComponent, outlet: 'facility-category-router-outlet' },
{ path: 'facility-booking', canActivate: [AuthGuardCustomerService], component:
FacilityBookingComponent, outlet: 'facility-router-outlet' } ] }
```

Signification du bloc d'instructions : **'seance-booking'** est le composant parent et ses deux enfants sont identifiés par **'facility-category-booking'** et **'facility-booking'**.



- Au niveau de l'interface visuelle, ces trois composants sont disposés de la manière suivante :



Le composant *Seance* est représenté en pointillé rouge et joue le rôle de container pour ses deux composants enfants *FacilityCategoryBooking* en pointillé vert et *FacilityBooking* en pointillé bleu. Le cheminement nominal des actions utilisateurs est le suivant :

- A chaque click sur (1) la liste des équipements se trouvant dans la zone verte est réactualisée.
- Lorsque l'utilisateur clique en (2), l'équipement ainsi que la tranche horaire viennent compléter la liste de la zone bleue
- Puis enfin lorsque l'utilisateur valide la séance en (3), celle-ci se rajoute au panier comme le matérialise la flèche ocre.



### 5.2.3 Gestion de sélections concurrentes d'équipements

Afin que deux utilisateurs différents ne sélectionnent pas le même équipement à la même tranche horaire, j'ai mis en place un trigger associé la table *TimestampFacility* pour gérer cette situation. Son implémentation ainsi que la fonction stockée à laquelle elle fait référence se présentent de la manière suivante :

```
CREATE TRIGGER triggerTimestamp BEFORE INSERT ON timestamp_facility
FOR EACH ROW
BEGIN
    IF `func_count_timestamp`(NEW.date_of_timestamp,
NEW.facility_id_facility) > 0 THEN
        signal sqlstate '45000';
    END IF;
END^;

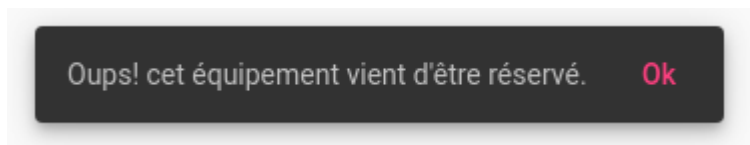
DROP FUNCTION IF EXISTS func_count_timestamp^;

CREATE FUNCTION func_count_timestamp(dateTimeOfSeance DATETIME, idFacility INT)
RETURNS int
BEGIN
    DECLARE nb INT;

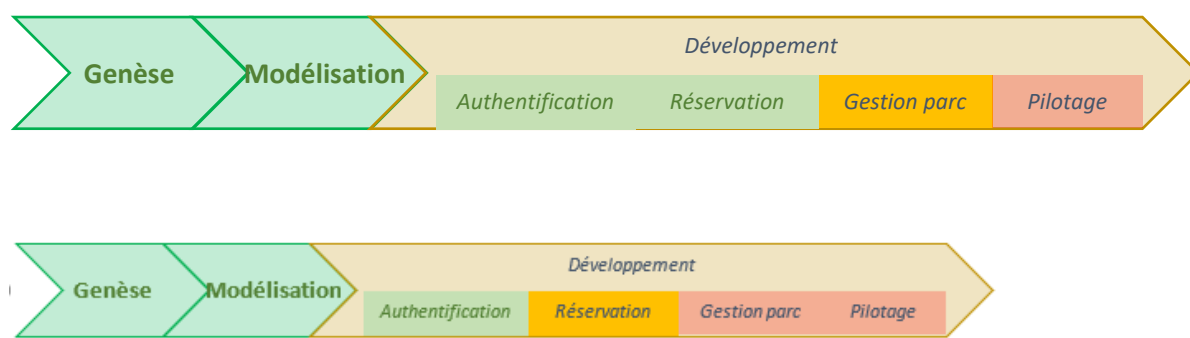
    SELECT COUNT(*) INTO nb FROM db_fitness.timestamp_facility WHERE
date_of_timestamp=dateTimeOfSeance AND facility_id_facility=idFacility;

    RETURN nb;
END^;
```

Si jamais un utilisateur tente de sélectionner un équipement qui vient de l'être par un autre utilisateur, le trigger envoie le message d'erreur `signal sqlstate '45000'` ; ce qui provoquera l'apparition du message suivant sur l'écran du premier utilisateur :



La section suivante traite de la gestion du parc

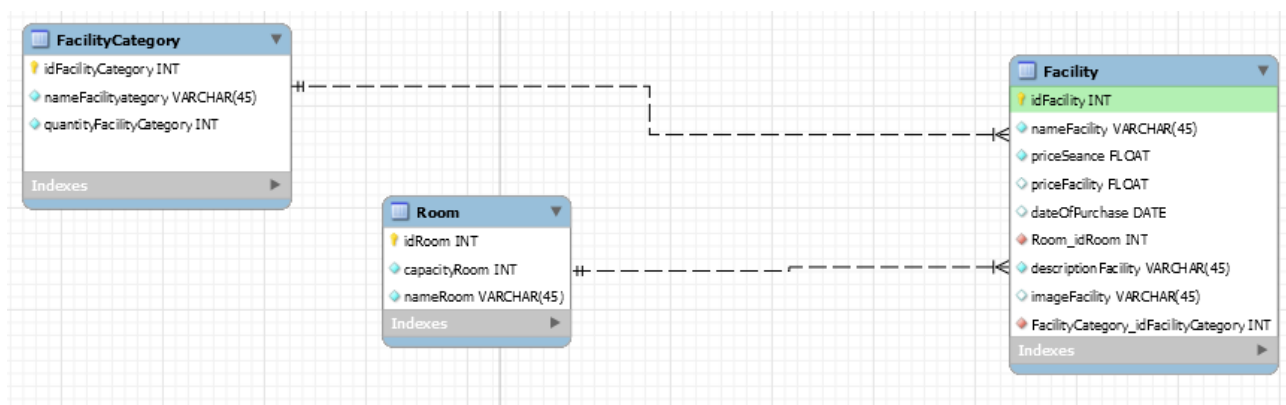


## 6. La gestion du parc

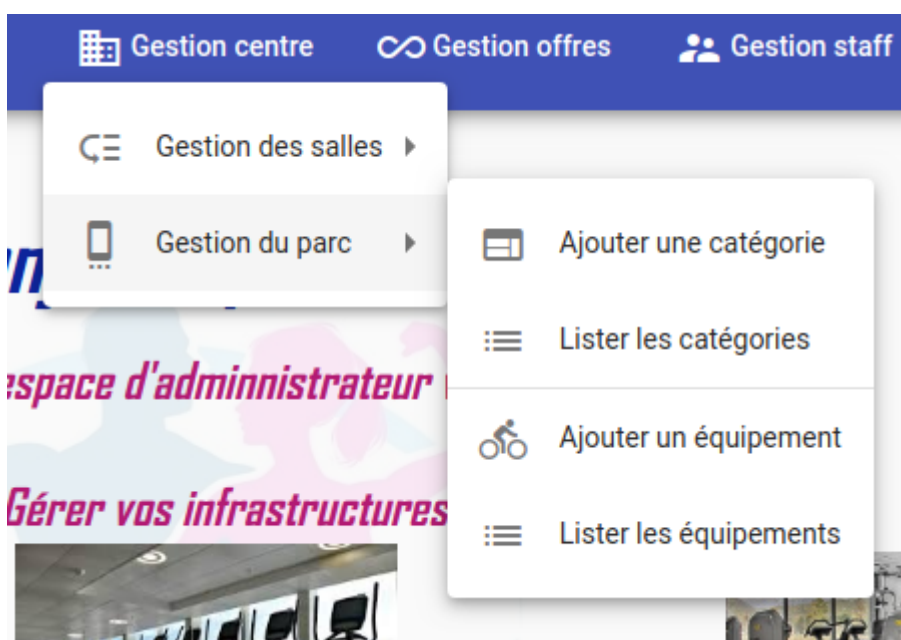
Ce module permet au staff de tenir à jour la base de données des équipements proposés en réservation pour un client Smart Fitness.

### 6.1 L'organisation logique des données relatives aux équipements

Chaque référence d'équipement est enregistrée dans la table *Facility*. En outre, en plus des données intrinsèques les caractérisant, chaque équipement se singularise par son appartenance à une salle et à une catégorie d'équipement. C'est pourquoi dans notre modèle logique de données, les tables dont *FacilityCategory* et *Room* se trouvent liées à *Facility* comme l'illustre le schéma suivant :



En conséquence, pour saisir et mettre à jour une fiche relative à un équipement, un manager devra entre autres renseigner la salle et la catégorie d'équipement, voire si besoin en créer : aussi la rubrique Gestion parc donne-t-elle la possibilité aux managers d'entrer et modifier des informations relatives aux salles et aux catégories d'équipements (voir à ce sujet la rubrique suivante pour plus de détails sur l'implémentation de ces fonctionnalités)



## 6.2 La gestion du contrôle de cohérence de la saisie des données.

Lorsqu'un manager accède à la page pour créer ou modifier une salle, un contrôle de cohérence sur le nom de ce dernier est mis en œuvre. Ce contrôle est basé sur la notion de *Validators* et me permet de m'assurer de l'unicité du nom de la salle lors de sa saisie. Son implémentation se répartit dans plusieurs fichiers :

- En premier lieu dans le type script du component au niveau de la création du formulaire :

```
createForm(){  
  
  this.roomForm = this.formBuilder.group({  
    nameRoom: ['', [  
      Validators.required,  
      Validators.minLength(1),  
      RoomValidator.nameRoomValidator(this.managerService.listNameRooms)  
    ]],  
  });  
}
```

- En second lieu, mon custom Validator est implémenté par *RoomValidator* et comporte les deux méthodes suivantes :

```
import { AbstractControl } from '@angular/forms';  
  
export class RoomValidator {  
  static nameRoomValidator(rooms: string[]) {  
    return (control: AbstractControl): { [key: string]: any } | null => {  
      let isValid = false;  
      if (control.value) {  
        const checkNameRoom: string = control.value;  
        isValid = !(rooms.find(nameRoom => nameRoom.toLowerCase() ===  
          checkNameRoom.toLowerCase()));  
        if (isValid) {return null;}  
      } else {  
        return { nameRoom: true };  
      }  
    }  
  }  
  
  static nameRoomDetailValidator(rooms: string[], nameRoomInit: string) {  
    return (control: AbstractControl): { [key: string]: any } | null => {  
      let isValid = false;  
      if (control.value) {  
        const checkNameRoom: string = control.value;  
        isValid = !(rooms.find(nameRoom => (nameRoom.toLowerCase() === checkNameRoom.toLowerCase())  
          && (checkNameRoom.toLowerCase() !== nameRoomInit.toLowerCase())));  
        if (isValid) {return null;}  
      } else {  
        return { nameRoom: true };  
      }  
    }  
  }  
}
```

Explication : Les deux méthodes indiquent l'existence ou non du nom de la salle passé en paramètre, retournant soit **true** dans la première éventualité et **null** dans la seconde.

Remarque : il existe deux méthodes, l'une dédiée à la création (**nameRoomValidator**) et l'autre à la modification (**nameRoomDetailValidator**). La différence entre les deux s'explique par le fait que dans le cas d'une modification d'une salle, un manager doit toujours avoir la possibilité de la renommer avec son nom initial.

- En troisième lieu, l'appel depuis un service ( *managerService* ) d'une méthode du front-end ( `updateRoom(idRoom:number, nameRoom: string, capacityRoom: number)` ) mettant à jour la base de données via un appel REST de type *put* (`this.httpClient.put<Room>('http://localhost:8080/managerctrl/updateroom')`), doit avoir pour corollaire l'actualisation du *Behavioursubject* associé à la liste des noms des salles afin de garder le caractère opérationnel du *Validator* chargé du contrôle de la cohérence des noms de salles :

```
let index = this.listRooms.findIndex(room => room.idRoom === idRoom);

this.listRooms[index].nameRoom = nameRoom;
this.listRooms$.next(this.listRooms);
this.listNameRooms = [];
for(let i = 0; i < this.listRooms.length; i++){
  this.listNameRooms.push(this.listRooms[i].nameRoom);
}
this.listNameRooms$.next(this.listNameRooms);
this.router.navigate(['room-listing']);
```

### 6.3 L'upload d'images

L'application donne la possibilité aux managers d'uploader des images pour compléter la fiche d'un équipement. Cette image sert d'encart visuel dans la liste des équipements disponibles lorsqu'un client Smart se trouve dans le module de réservation. L'implémentation de cette fonctionnalité m'a posé quelques difficultés. J'en explique les raisons dans les sections suivantes :

#### 6.3.1 Implémentation de la fonctionnalité d'upload côté « front-end »

Afin de pouvoir mettre en œuvre cette fonctionnalité, il m'a fallu effectuer plusieurs recherches sur le Net. Pour le « front-end » je me suis inspiré du site <http://blog.shipstone.org/post/angular-material-spring-upload/> pour implémenter la fonctionnalité d'upload dans les composants y faisant appel :

```
export class AppComponent implements OnInit {
...
  @ViewChild('fileInput') fileInput: ElementRef;
...
  selectFile(): void {
    this.fileInput.nativeElement.click();
  }
...
}
```



Tous les échanges de messages faisant appel aux méthodes de l'API REST entre le serveur d'application (8080) et le serveur de présentation (4200) pour un utilisateur connecté (client et staff), nécessitent l'échange de tokens dans le header de chaque requête afin de préserver le caractère sécurisé de ces échanges. Ce header a donc le format suivant :

```
{
  headers: {
    "Content-Type": "application/json",
    "Authorization": this.token.getToken()
  }
}
```

Or pour réaliser un upload d'image depuis un poste client vers le serveur, le format du header est de type binaire correspondant à l'image transféré. Il ne devient donc plus possible d'échanger de tokens ni par conséquent de garder la trace de l'identité de l'utilisateur initiateur des échanges. Afin que l'utilisateur dispose toujours d'une session active sécurisée sans interruption de service, je me sers des *window.localStorage* pour stocker, pendant le temps de transfert de l'image, son *username* et son *password* afin d'opérer une reconnexion automatique puis un *routing* vers la page listant les salles si l'utilisateur venait par exemple de créer ou modifier une salle.

### 6.3.2 Implémentation de la fonctionnalité d'upload côté « back-end »

En guise de transition avec la partie « front-end », et pour mettre en évidence la faille de sécurité dû à l'absence d'échange de tokens, il n'y a pas de restrictions au niveau des rôles pour le *end-point* faisant référence à l'upload. Ce qui se traduit par la ligne suivante dans le package *security* du serveur d'application :

```
.antMatchers("/managerctrl/upload").permitAll()
```

D'autres investigations sur l'upload d'images (<https://www.javaguides.net/2018/11/spring-boot-2-file-upload-and-download-rest-api-tutorial.html>) ont abouti au développement suivant :

- Dans un premier temps, la mise en œuvre d'un service par injection de dépendance dans le controller où se situe le *end-point* relatif à l'upload:

```
@Autowired
```

```
private FileStorageService fileStorageService;
```

- Dans un second temps, toujours dans le même controller l'appel à la fonction opérant à proprement parlé l'upload du fichier :

```
this.fileStorageService.storeFile(multipartFile);
```

Ce qui nécessite d'intégrer les éléments suivants, objet du troisième temps

Dans un troisième temps, le service *fileStorageService* va aller lire la valeur du paramètre *file.upload-dir* qui se trouve dans le fichier *application.properties*. Dans le cas présent, ce paramètre a été initialisé avec la valeur suivante :

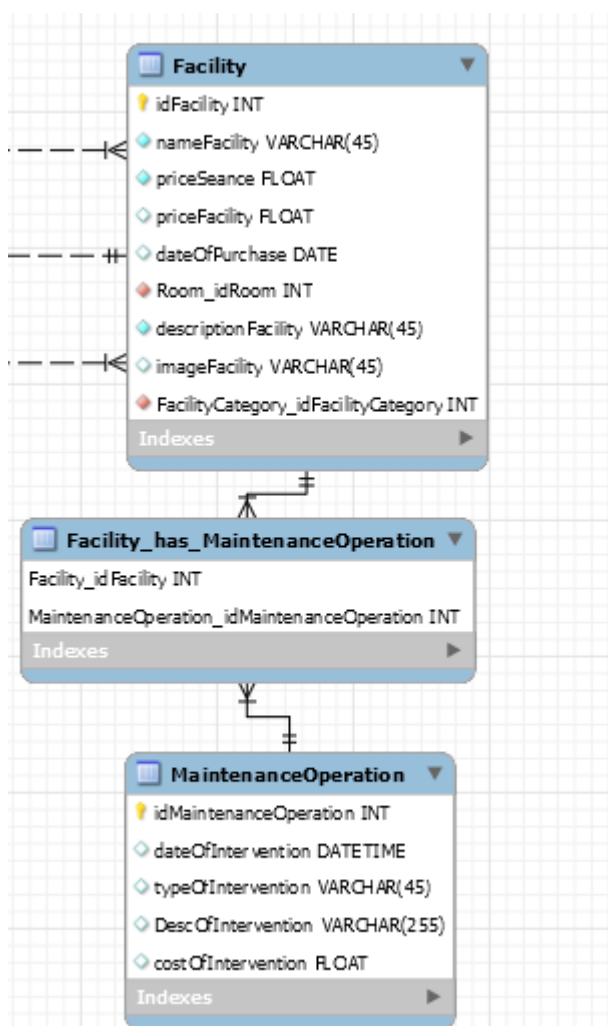
*/home/laurent/smartFitness/dev/front/src/assets/images/facilities*, ce qui permet d'indiquer des chemins relatifs dans le « front-end » qui se base sur le répertoire de ressources *assets*.



## 6.4 La gestion des opérations de maintenance des équipements.

L'application propose aux managers un outils pour saisir des opérations de maintenance effectuées sur les équipements du parc. L'intérêt de ce module se situe à deux niveaux :

- Au niveau du « back-end », une relation @ManyToMany a été instaurée entre les entités 'MaintenanceOperation' et 'Facility'. Une relation ManyToMany est une relation à valeur multiple des deux côtés de la relation. Il y a donc une liste d'entités 'facilities' (`private List<Facility> facilities;`) au sein de MaintenanceOperation, et réciproquement une liste d'entités 'maintenanceOperations' (`private List<MaintenanceOperation> maintenanceOperations;`) au sein de Facility. Cette relation se traduit dans la base de données par une table d'intersection (Facility\_has\_MaintenanceOperation) avec deux clés étrangères (Facility\_idFacility et MaintenanceOperation\_idMaintenanceOperation portant respectivement sur les entités Facility et MaintenanceOperation ).

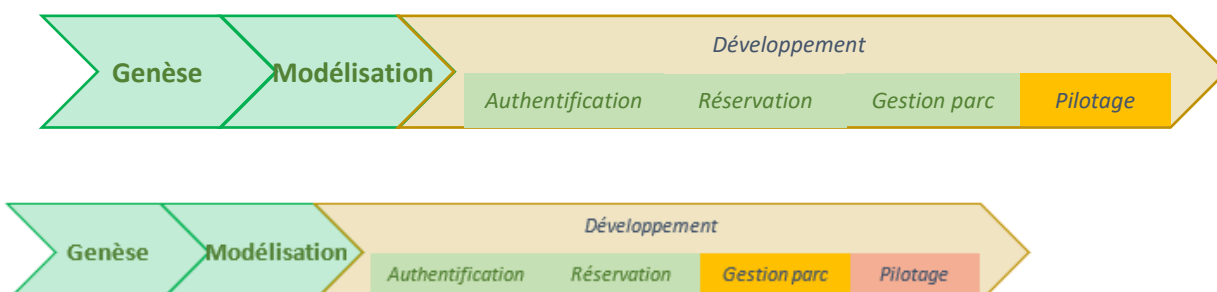


- Au niveau du « front-end », la page de maintenance de l'équipement en question propose, outre le récapitulatif de l'historique de ses opérations de maintenance, sa valorisation économique en termes de gains et de dépenses générés par l'équipement. Les gains correspondant à la somme des réservations enregistrées par l'équipement, les dépenses à la somme de son prix d'achat et de ses opérations de maintenance (Pour une question de simplicité d'implémentation, le calcul du gain est basé sur le tarif de prestation en base, il ne prend donc pas en compte si le client est abonné, qui correspond demi-tarif de la prestation ni d'un changement de prix de la prestation au cours du temps). Cette balance commerciale entre les gains et les dépenses est traduit de manière suivante sous forme graphique :

$(\text{Somme gains} - \text{Somme dépenses}) / (\text{Somme gains} + \text{Somme dépenses})$

Une barre rouge si  $\text{Somme gains} < \text{Somme dépenses}$ , une barre verte dans le cas contraire, cette barre étant proportionnelle à l'expression mathématique ci-dessus.

La section suivante aborde le module de pilotage



## 7. Le module de pilotage

