

---

TB

---

QUENTIN GIGON

HEIG-VD

---

# Sections and Chapitres

Quentin Gigon

## Table des matières

<b>1</b>	<b>Préface</b>	<b>5</b>
<b>2</b>	<b>Introduction au projet</b>	<b>6</b>
<b>3</b>	<b>Présentation des technologies utilisées</b>	<b>7</b>
3.1	Framework Play! . . . . .	7
3.2	EventSource . . . . .	7
3.3	PostgreSQL . . . . .	7
3.4	JPA . . . . .	7
3.5	HTML5 . . . . .	7
<b>4</b>	<b>Analyse et Architecture</b>	<b>8</b>
4.1	Site . . . . .	8
4.2	Team et utilisateurs . . . . .	8
4.3	Organisation des flux . . . . .	8
4.3.1	Flux . . . . .	8
4.3.2	Schedules . . . . .	9
4.3.3	Diffuser . . . . .	10
4.4	Ecrans . . . . .	10
4.4.1	Affichage . . . . .	10
4.4.2	Events . . . . .	10
4.4.3	Authentification . . . . .	11
4.4.4	Groupe d'écrans . . . . .	11
<b>5</b>	<b>Schéma de base de donnée</b>	<b>13</b>
5.1	Equipes et utilisateurs . . . . .	13
5.2	Flux . . . . .	13
5.3	Schedule . . . . .	13
5.4	Diffuser . . . . .	15
5.5	Ecrans . . . . .	15
<b>6</b>	<b>Réalisation</b>	<b>16</b>
6.1	Organisation des flux . . . . .	16
6.1.1	Bloc-horaire . . . . .	16
6.1.2	RunningScheduleThread . . . . .	17
6.1.3	FluxManager . . . . .	19
6.1.4	AutomatedScheduleStarter . . . . .	20
6.1.5	FluxChecker . . . . .	21
6.2	Contrôleurs . . . . .	22
6.2.1	EventSourceController . . . . .	22
6.2.2	ScreenController . . . . .	24
6.2.3	ScheduleController . . . . .	25
6.2.4	DiffuserController . . . . .	27
6.3	DAOs . . . . .	29
6.3.1	Requêtes SQL . . . . .	29
6.3.2	Services . . . . .	30
6.3.3	Implémentation Java des entités BD . . . . .	31
6.4	Restriction d'accès . . . . .	32
6.5	Triggers . . . . .	33
6.6	Vues . . . . .	34

6.6.1	Affichage des flux . . . . .	34
6.6.2	Echange de données . . . . .	35
<b>7</b>	<b>Interface</b>	<b>38</b>
7.1	Home . . . . .	38
7.2	Teams . . . . .	38
7.3	Users . . . . .	39
7.4	Fluxes . . . . .	39
7.5	Screens . . . . .	40
7.6	Schedules . . . . .	40
7.7	Diffusers . . . . .	41
<b>8</b>	<b>Tests</b>	<b>42</b>
8.1	Tests unitaire . . . . .	42
8.2	Tests fonctionnels . . . . .	42
<b>9</b>	<b>Commentaires et conclusion</b>	<b>43</b>
	<b>Appendices</b>	<b>44</b>
<b>A</b>	<b>Cahier des charges</b>	<b>44</b>
A.1	Contraintes et besoins . . . . .	44
A.2	Fonctionnalités . . . . .	44
A.3	Echéancier . . . . .	45
<b>B</b>	<b>Journal de travail</b>	<b>46</b>
<b>C</b>	<b>Cas d'utilisation</b>	<b>47</b>
C.1	Administrateur : . . . . .	47
C.2	TeamAdmin : . . . . .	49
C.3	TeamMember : . . . . .	50
<b>D</b>	<b>Mockups</b>	<b>52</b>
D.1	Utilisateurs . . . . .	52
D.2	Ecrans . . . . .	53
D.3	Teams . . . . .	54
D.4	Flux . . . . .	55
D.5	Schedules . . . . .	56
D.6	Diffusers . . . . .	57

## Table des figures

1	Protocole - écran connu par le serveur . . . . .	11
2	Protocole - écran inconnu par le serveur . . . . .	12
3	Schéma de la base de donnée . . . . .	14
4	Organisation des flux . . . . .	16
5	Algorithme de scheduling des flux . . . . .	19
6	Schéma des services d'accès aux données . . . . .	29
7	Homepage . . . . .	38
8	Teampage . . . . .	38
9	Ajout d'utilisateur . . . . .	39
10	Création de Flux . . . . .	39
11	Schedules page . . . . .	40
12	Schedules page . . . . .	40
13	Schedules page . . . . .	41
14	Interface d'ajout de nouveaux utilisateurs . . . . .	52
15	Interface de connexion des utilisateurs . . . . .	52
16	Page principale des écrans . . . . .	53
17	Interface d'ajout de nouvel écran . . . . .	53
18	Page principale des teams . . . . .	54
19	Interface de création de team . . . . .	54
20	Page principale des flux . . . . .	55
21	Interface de création de flux . . . . .	55
22	Page principale des Schedules . . . . .	56
23	Interface de création de Schedule . . . . .	56
24	Page principale des diffuseurs . . . . .	57
25	Interface de création de diffuser . . . . .	57

## Listings

1	Création du bloc-horaire . . . . .	16
2	Eventsource Java . . . . .	17
3	FluxManager . . . . .	19
4	FluxManager . . . . .	21
5	Eventsource Java - EventSourceController.java . . . . .	22
6	Eventsource Scala - EventSourceController.scala . . . . .	23
7	Akka Actor Scala - EventSourceController.scala . . . . .	23
8	Authentification des écrans - ScreenController.java . . . . .	24
9	Désactivation des écrans - ScreenController.java . . . . .	25
10	Activation d'un Schedule - ScheduleController.java . . . . .	26
11	Désactivation d'un Schedule - ScheduleController.java . . . . .	26
12	Activation d'un Diffuser - DiffuserController.java . . . . .	27
13	Détection d'un Diffuser actif lors de l'envoi de Flux - RunningScheduleThread.java . . . .	28
14	Exemples de requêtes SQL avec JPA . . . . .	29
15	Exemple de création de service . . . . .	30
16	Exemple de modèle Java - Schedule.java . . . . .	31
17	Admin Authentification Action . . . . .	32
18	Utilisation d'une Action custom . . . . .	32
19	Triggers SQL - Suppression . . . . .	33
20	Eventsource HTML . . . . .	34
21	Eventsource JS . . . . .	34
22	UserData.java . . . . .	35
23	Exemple échange serveur-client . . . . .	35
24	Exemple échange client-serveur . . . . .	36
25	Exemple des messages d'erreur . . . . .	36
26	Exemple de test unitaire - FluxUnitTest.java . . . . .	42

# 1 Préface

Dans ce document seront utilisés des termes anglais avec une majuscule. Ces termes représentent des objets ou entités de l'application associés à des éléments du monde réel. Leur traduction française étant souvent moins représentative, le choix a été fait de les utiliser tels quels. Ces termes sont les suivants :

- Team
- Schedule
- Diffuser

Certains termes informatiques n'ayant pas de réelle traduction mot-à-mot peuvent également avoir été utilisés.

## 2 Introduction au projet

La HEIG-VD dispose de nombreux écrans pour la diffusion d'informations, réparti sur ses différents sites. A l'heure actuelle, un système existe déjà pour permettre une certaine organisation du contenu diffusé mais il s'agit plutôt d'une solution temporaire. Il ne fournit en effet que des fonctionnalités de diffusions pures, ne propose pas de système d'utilisateurs ou d'équipes et ne propose pas de modèle générique pour l'ajout de flux au sein de l'application.

Le but de ce travail de Bachelor est donc de centraliser la gestion des écrans et de l'affichage de flux à travers une application Web, et de fournir un programme répondant le plus possible aux exigences fournies. Si ces dernières sont explicitées dans le cahier des charges (voir Annexe A), en voici une courte liste non-exhaustive :

- Gérer les droits des utilisateurs
- Permettre des opérations de maintenance à distance
- Limiter l'affichage de flux selon diverses conditions
- Gérer l'ajout d'écrans au système
- Modélisation d'un flux de données

Un élément très important dans la direction qu'a prise le projet est la différence matérielle entre les écrans. Il s'agit en effet soit de SmartTV, soit de PC faisant tourner Ubuntu Server. Cela a eu un gros impact dans la construction de l'architecture du programme, car il fallait une solution de rendu sur les écrans compatibles avec les deux types de matériel. La technique utilisée actuellement, soit un affichage des flux dans l'onglet d'un navigateur, a été très vite définie comme la solution à implémenter.

## 3 Présentation des technologies utilisées

### 3.1 Framework Play!

Play! est un framework web open-source qui suit le modèle MVC et qui permet d'écrire rapidement des application web en Java (ou en Scala). A la différence d'autre frameworks Java, Play! est *stateless*, ce qui veut dire qu'il n'y a pas de session créée à chaque connexion. Il fourni aussi à ses utilisateurs des frameworks de tests unitaires et fonctionnels.

### 3.2 EventSource

Les EventSource, ou Server-Sent Events (SSE), sont une technologie permettant à un navigateur internet de recevoir des mises à jour automatiques d'un serveur par une connexion HTTP persistante. L'API Javascript (*Server-Sent Events EventSource API*) fut instaurée la première fois dans Opera en 2006 et a été normalisée dans le cadre de HTML5.

Les Events envoyés sont au format *text/event-stream* et sont reçus par le navigateur sous la forme d'Event de type *message*. La connexion reste ouverte tant qu'elle n'a pas été fermée par le navigateur, et contrairement aux WebSockets, les SSE sont uni-directionnels et ne permettent donc pas aux clients de communiquer avec le serveur.

### 3.3 PostgreSQL

PostgreSQL est un système de gestion de base de données relationnelle et open-source. Les différences principales entre PostgreSQL et ses concurrents sont la prise en charge de plus de types de donnée que les types traditionnels (entiers, caractères, ...), ainsi qu'une communauté plus active et un développement plus rapide que MySql par exemple.

### 3.4 JPA

La Java Persistence API (JPA) est une interface de programmation permettant aux utilisateurs de la plateforme Java (SE et EE) d'organiser facilement et clairement leurs données relationnelles. Elle utilise des annotations pour définir des "objet-métiers" qui serviront d'interface entre la base de données et l'application.

JPA définit aussi le Java Persistence Query Language (JPQL), qui est utilisé pour créer les requêtes SQL dans le cadre de JPA. Les requêtes effectuées dans ce langage ressemblent beaucoup à du SQL classique, sauf que le JPQL fonctionne avec des entités (créées avec des annotations) plutôt que des tables de la base de données.

### 3.5 HTML5

HTML5 est la dernière version majeure du HTML (octobre 2014). Elle vient avec plein de nouveaux éléments, comme la balise *video*, qui permet d'insérer un contenu vidéo en streaming dans un fichier HTML, ou encore *footer*, qui lui permet de facilement afficher du texte en bas de page.



## 4 Analyse et Architecture

Dans les chapitres suivants seront détaillés l'architecture du programme (les modèles et leur représentation réelle) ainsi que les différentes idées et versions qui ont été envisagées. Leurs équivalents en base de données seront explicités dans le chapitre suivant.

### 4.1 Site

Un Site représente un emplacement physique de l'HEIG-VD, donc les sites de Cheseaux, St-Roch et Y-Park. Ils servent principalement à localiser les écrans et restreindre l'affichage de flux selon le lieu. Par exemple pour l'affichage des horaires, où l'horaire de Cheseaux importe peu les gens de St-Roch et vice-versa.

Une idée et demande de mon mentor était d'avoir un seul flux horaire qui, selon le site où il est affiché, prend un paramètre de requête différent (p.ex. *www.horaire.ch?site=che*). Cela permet d'offrir plus de possibilités lors la création de flux pour l'utilisateur. Malheureusement, à cause d'un oubli de ma part, cette fonctionnalité n'est pas présente et il faut donc avoir un flux d'horaire pour chaque site.

### 4.2 Team et utilisateurs

Pendant la phase d'analyse, il a été spécifié qu'un système d'équipe était nécessaire, afin de restreindre les fonctions du programme selon l'appartenance de l'utilisateur courant à telle ou telle équipe. Cela faisait également sens vu que le programme sera utilisé par les différents départements de la HEIG-VD.

Les **Teams** ont donc une place centrale dans l'architecture du programme car les actions proposées à l'utilisateur utilisent uniquement les données accessibles par son équipe. Elle est composée de membres et de chefs d'équipe, qui sont les deux représentés par des **TeamMember**, et qui ont différents niveaux d'accès.

Dans la première version proposée pour la représentations des utilisateurs, les équipes étaient composées de trois types : un administrateur, des chefs d'équipe et des membres. Lors de la rédaction des cas d'utilisation et après un entretien avec mon mentor, il a été décidé que d'avoir autant de status différents au sein d'une même équipe rendait la division des actions possibles moins logique et l'utilisation finale plus compliquée sans grandes raisons. Les administrateurs d'équipe ont ainsi disparu pour être remplacés par les chefs d'équipe.

En plus des équipes, il fallait également un rôle d'administrateur central, à qui sont réservées quelques fonctionnalités (p.ex. l'ajout de nouveaux écrans au système).

### 4.3 Organisation des flux

Un flux dans ce programme représente un contenu à afficher sur un écran. Pendant la phase d'analyse, il a été spécifié que l'application devait pouvoir gérer deux sortes de flux : un flux externe à l'application comme le flux de la RTS ou un flux interne, comme les horaires des cours ou les réservations de salle.

J'avais également proposé de fournir un template RSS permettant un ajout facile de nouveau flux RSS, mais cette fonctionnalité a été abandonnée par manque de temps.

En ce qui concerne leur diffusion, il fallait proposer un moyen d'organiser un horaire de flux ainsi que la possibilité d'envoyer immédiatement un flux sur un écran. Deux types d'objets ont ainsi été créés : des Schedules, qui représentent l'horaire des flux pour une journée et des Diffusers, qui eux permettent d'envoyer un flux aux écrans sans passer par un Schedule. Ils sont explicités dans les sections suivantes.

#### 4.3.1 Flux

Un flux est caractérisé par un nom, un type, une durée d'affichage et un contenu. Ce contenu est défini par le type du flux ; un flux 'URL' contiendra une url, tandis un flux de type 'Image' contiendra l'adresse de l'image sur le serveur.

Ils sont regroupés en quatre types :

- URL, ou type standard
- Vidéo
- Image
- Texte

Le traitement de ces flux par le système et la manière par-laquelle ils sont affichés sur les écrans changent selon leur type. Par exemple, un flux 'Image' sera rendu dans une balise <img>, tandis qu'un flux 'URL' le sera dans une *iframe*.

De plus, il y a trois sortes de flux : des flux généraux, des flux localisés et des flux de fallback. Ils sont comme des sous-populations de flux : ils possèdent une référence vers un flux existant. Comme son nom l'indique, un flux général peut être diffusé partout. Par opposition, un flux localisé est lui uniquement affichable sur le Site correspondant. Et enfin un flux peut être un flux de fallback. Ces flux sont spécifiés à la création d'un Schedule et utilisés par celui-ci pour remplacer un flux programmé mais pour lequel le serveur ne détecte aucune données à afficher. Dans le cas où aucun flux de fallback n'est sélectionné, un flux d'erreur est envoyé à la place.

La durée d'affichage d'un flux est déterminée par deux facteurs : son nombre de phases et la durée d'une phase. Cette décomposition a été implémentée car elle règle un problème mentionné par mon mentor, à savoir qu'il y a des flux (les réservations de salles par exemple) qui peuvent avoir trop de données à afficher en une seule fois (30 éléments pour une table de 20, etc).

Une autre demande de mon mentor était d'avoir un système permettant de vérifier si un flux avait l'autorisation de s'afficher selon diverses conditions, la plus simple étant une condition temporelle avec des bornes. Ma solution proposée est d'avoir un micro-serveur ou service à qui l'on peut faire des requêtes et qui, selon les paramètres de la requête, nous renvoie des fichiers Json contenant diverses informations. Pour reprendre l'exemple d'une condition temporelle, on peut imaginer que l'application doive vérifier si un flux a le droit de s'afficher. Elle fera donc une requête à ce service qui ressemblera à ceci :

```
https://flux_check.com?fluxId=4&type=time
```

On recevrait alors un Json contenant une borne de début et une de fin et le programme pourrait vérifier si la date courante est dans les bornes ou non. On peut aussi imaginer d'autres type de vérification/limitation d'où le paramètre de requête *type*.

### 4.3.2 Schedules

Un Schedule représente un horaire de flux. Un Schedule n'est pas assigné à des écrans à sa création mais au moment de son activation pour permettre une réutilisation plus facile des Schedules créés. Chaque chef d'équipe peut en créer qui seront utilisables par tous les membres de son équipe.

La première version de cet horaire modélisait un cycle, où les flux choisis par l'utilisateur bouclaient à l'infini. Ce système a été utilisé avec succès pour de la recherche et des tests sur la faisabilité du programme mais fut vite obsolète quand il s'agissait d'avoir plus de contrôle sur l'horaire, par exemple définir une heure de début pour un flux donné.

Il a donc fallu inventer un système permettant à la fois de créer un horaire à respecter pour les Schedules et de modifier cet horaire à la volée pour les Diffusers, tout en garantissant la cohérence du programme.

J'ai choisi de représenter une journée complète d'affichage par des blocs de 1 minute chacun. Cette plage d'affichage est bornée par une heure de départ (inclusive) et une heure de fin (exclusive) qui sont fixes, mais modifiables si besoin. A titre d'exemple, en prenant les valeurs que j'ai défini, soit de 8h à 22h, on obtient une plage horaire de 15h, ce qui est équivalent à 900 blocs.

Si on regarde maintenant la procédure de création puis d'activation d'un Schedule : Quand un utilisateur crée un Schedule, il peut lui spécifier plusieurs choses :

- Un nom (unique)
- Des flux avec heure de début (ScheduledFlux)
- Des flux sans heure de début
- Des flux de fallback
- Garder l'ordre des flux ou non

Pour chaque flux avec une heure de début associée, une "entrée de calendrier", ou ScheduledFlux, est créée et ajoutée dans la base de données. Cette "entrée" contient des références vers le Flux et le Schedule concerné, ainsi que le numéro de son bloc de départ.

Chaque flux sans heure de début est simplement ajouté au Schedule, pareil pour les flux de fallback. L'option de garder l'ordre des flux permet de garantir que les flux sans heure fixe seront affichés dans l'ordre dans lequel l'utilisateur les a rentrés.

A l'activation d'un Schedule, on peut choisir parmi les écrans auxquels on a accès ceux qui seront concernés. Un objet `RunningSchedule` est créé, une entité temporaire qui existe uniquement tant que le Schedule reste actif. Cette entité est ensuite utilisée pour créer un "bloc-horaire", qui associe avec chaque bloc le flux correspondant (en regardant parmi les `ScheduledFlux`) ou une absence de flux. Un `Runnable` (`RunningScheduleThread`) est ensuite lancé avec cet horaire. La manière dont il choisit le prochain flux est détaillée dans le chapitre **Réalisation**.

Ce système permet de répondre à une autre demande du projet, la reprise de l'exécution des Schedules actifs lors du redémarrage du serveur ou après une maintenance. En donnant un moyen de faire correspondre une heure donnée avec un numéro de bloc, on peut très facilement reprendre l'exécution là où elle s'était arrêtée.

### 4.3.3 Diffuser

Les Diffusers sont utilisés pour diffuser directement du contenu sur les écrans, sans devoir passer par un Schedule. Ils ont été rajoutés car mon application manquait d'un moyen d'envoyer rapidement un flux, p.ex. pour un message d'alerte. Ils doivent également permettre de choisir une période de validité pendant laquelle le Diffuser sera actif.

A sa création, l'utilisateur doit spécifier les attributs suivants :

- Un nom (unique)
- Un flux
- Une heure de début

Le comportement d'un Diffuser est le suivant : il remplace l'exécution d'un potentiel Schedule actif pour les écrans auxquels il est associé tant qu'il est actif.

Comme pour les Schedules, un Diffuser actif est un `RunningDiffuser`, lui aussi une entité temporaire qui reste active selon la validité spécifiée à sa création ou jusqu'à sa désactivation/destruction. Une fois que le `RunningDiffuser` est arrivé en fin de vie, les écrans auxquels il diffusait un flux peuvent reprendre leur Schedule.

## 4.4 Ecrans

Lors de mes discussions avec mon mentor, il a été assez vite recommandé et conseillé que j'utilise des `Eventsources` (voir doc) pour envoyer les ordres d'affichages aux écrans. Les sections suivantes résument et décrivent leur utilisation dans ce programme.

### 4.4.1 Affichage

Pour l'affichage des flux sur les écrans, les technologies autorisées étaient le HTML5, CSS3 et Javascript pur (sans frameworks). La demande initiale était de pouvoir afficher le contenu associé à une URL grâce aux iframes. Il fallait donc un moyen pour les écrans de recevoir une URL sous forme texte. En partant de ce constat, j'ai remarqué que si un Event contenait une URL en texte, il pouvait contenir d'autres données. J'ai donc proposé d'implémenter des flux de type image et texte. Les écrans sont capable de déterminer le type du flux reçu et de l'afficher de la manière adéquate. Les différents moyens utilisés l'affichage en question sont décrits dans la section **Réalisation**.

### 4.4.2 Events

Un des problèmes à résoudre était la question d'envoyer les bons Events aux bons écrans. Il y avait deux manières principales de faire, soit envoyer tous les Events à tous les écrans avec la liste des écrans concernés, soit générer dynamiquement des endpoints pour chaque Schedule. Pour des raisons de simplicité et parce que le nombre d'écrans restera raisonnable, la première possibilité a été choisie.

Les Events générés par les Schedules sont donc envoyés à tous les écrans s'étant connectés auprès du serveur. Chaque Event contient les adresses MAC des écrans concernés et c'est à l'écran de vérifier s'il est concerné par l'Event qu'il vient de recevoir.

Un Event est construit de la manière suivante : il contient son type, ses "données" (url ou texte) et les écrans concernés. Ci-dessous un exemple pour chaque type :

- URL :  
`url?https://heig-vd.ch|mac_address1,mac_address2`
- Image :

image?/assets/image1|mac\_address2, mac\_address3

- Video :  
video?https://www.youtube.com/embed/dQw4w9WgXcQ|mac\_address1, mac\_address3
- Texte :  
text?Hello World|mac\_address2

#### 4.4.3 Authentification

Un écran ne peut recevoir d'Event qu'une fois authentifié, pour des raisons évidentes de sécurité. Il fallait donc trouver un moyen d'identifier de manière unique chaque écran. La première piste envisagée fut d'utiliser les hostnames des écrans car cette information était facilement récupérable en Java, et spécialement avec Play. Mais en raison de l'architecture réseau de la HEIG et de la volonté d'intégrer le protocole WakeOnLan au programme, il a été décidé d'utiliser les adresses MAC à la place. WakeOnLan est un standard Ethernet qui permet d'allumer un ordinateur à distance en utilisant son adresse MAC.

La difficulté inhérente à ceci était de récupérer ces adresses depuis Java. Pour remédier à ce problème et en même temps fournir une couche de sécurité au niveau des écrans, il a été choisi que lors de l'ajout d'un écran au système et de son authentification, son adresse MAC devrait être précisée.

Le protocole de connexion des écrans au serveur est donc le suivant :

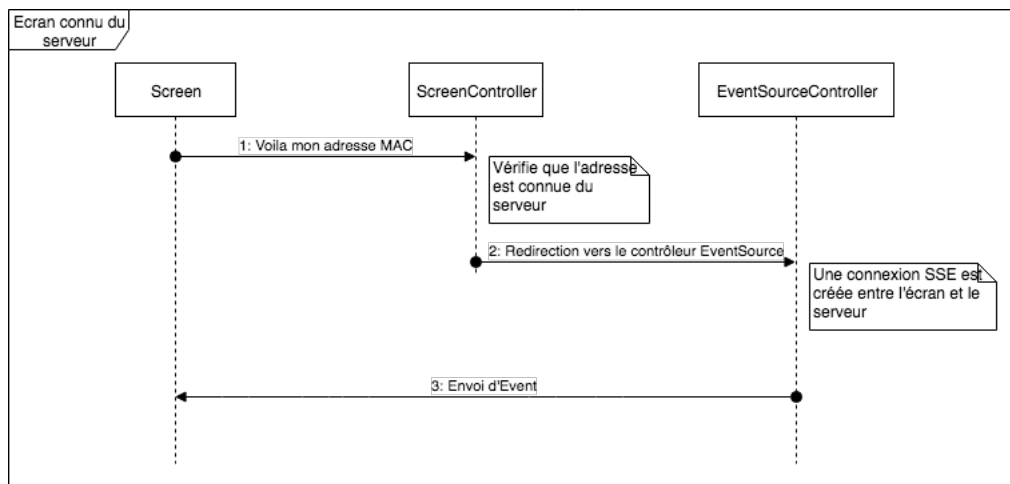


FIGURE 1 – Protocole - écran connu par le serveur

L'accès des écrans au serveur se fait donc en deux étapes. Il faut d'abord que l'écran se connecte à la route d'authentification des écrans, tout en spécifiant comme paramètre de requête son adresse MAC. Exemple :

`http://server/screens/auth?mac=1234`

Là, si l'adresse fournie est connue par le serveur, l'écran est redirigé vers le contrôleur chargé d'envoyer les Events aux écrans. Ce faisant, le ScreenController ajoute dans les cookies l'adresse MAC de l'écran ainsi que sa résolution (utile pour la gestion de l'affichage) et passe l'écran comme actif.

Ce protocole prend aussi en charge le cas où l'écran n'est pas connu du serveur. A ce moment là, il nous renvoie un code permettant l'ajout de l'écran dans le système depuis le site web.

Ce code est généré aléatoirement et unique pour chaque nouvel écran. Si l'écran essaie à nouveau de s'authentifier sans avoir été ajouté au système, le même code lui sera renvoyé. En cas de perte de connexion pendant l'échange, ce sera à l'écran de réitérer sa tentative.

#### 4.4.4 Groupe d'écrans

Une idée datant du début de ma réflexion sur le problème était de proposer à l'utilisateur de regrouper des écrans en groupes "logiques", par exemple le groupe des écrans du hall. Cela devait per-

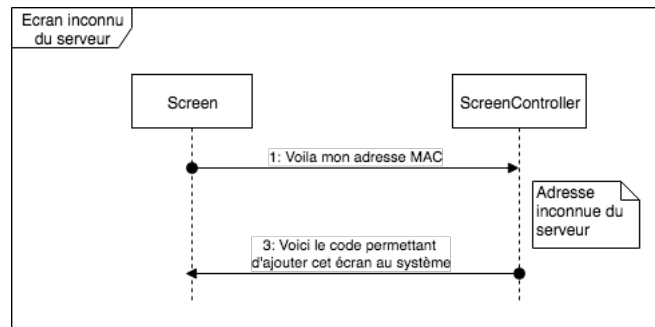


FIGURE 2 – Protocole - écran inconnu par le serveur

mettre d'assigner un Schedule à un groupe d'écrans et ainsi gagner du temps et rendre le programme plus simple.

Malheureusement, c'est un détail qui m'a échappé lors de la réalisation de l'application. Quand j'ai remarqué mon oubli, il était trop tard pour l'intégrer car il aurait fallu changer plusieurs modèles ainsi que la base de données et le temps manquait.

## 5 Schéma de base de donnée

Une des directives principales du projet était la représentation en tout temps de l'état actuel du programme en base de données. Le schéma a donc été pensé pour répondre à cette demande et les limitations voulues pour les différentes entités ont été au maximum intégrées dans la construction de la base. Dans les sections suivantes seront expliqués les choix et divers changements opérés pendant la phase d'analyse ainsi que des explications pour les éléments importants. Le schéma de la page suivante offre une représentation des tables et relations de la base de données et servira de référence dans ce chapitre.

### 5.1 Equipes et utilisateurs

Regrouper les gens en équipes permet de limiter le contenu auquel ils ont accès. La table **Team** est donc en relation avec toutes les entités manipulables, c'est-à-dire les écrans, **Flux**, **Schedules** et **Diffusers** pour filtrer les données envoyées à l'utilisateur. Comme mentionné dans le chapitre **Analyse et Architecture**, les équipes sont composées de membres simples et de chefs d'équipe (leur nombre n'est pas limité). Cette distinction permet de limiter les actions qui sont proposées par l'application selon le rôle de l'utilisateur (p.ex. restreindre la création de nouveaux **Schedules** aux chefs d'équipe mais laisser leur utilisation possible pour un membre).

En ce qui concerne les utilisateurs, pour chacun d'entre eux existe une entrée de la table **User** qui contient leurs données personnelles (mot de passe, email). Cette table est "héritée" par **TeamMember**, qui contient une clé étrangère vers la table **Team**, et par **Admin**, qui représente un administrateur système.

### 5.2 Flux

Dans les premières directives du projet figurait le besoin d'avoir des flux localisés et des flux généraux. Plus tard, toujours pendant la phase d'analyse, il a été jugé nécessaire de rajouter un modèle représentant les flux à envoyer lorsque le système détecte un problème avec le flux courant (envoi d'un flux localisé au mauvais site p.ex.). Comme pour les utilisateurs, il a été décidé de modéliser une relation "d'héritage" pour répondre à tout cela. Il y a donc une table **Flux** qui contient chaque flux créé (ses paramètres, type, etc). Les éléments de cette table ne peuvent pas être utilisés tels quels, il faut obligatoirement passer par les tables qui en héritent :

- **LocatedFlux**, qui représente un flux localisé et donc fait un lien entre **Site** et **Flux**.
- **GeneralFlux**, qui représente un flux général.
- **FallbackFlux**, pour les flux de fallback

### 5.3 Schedule

Pendant la réalisation du projet, j'ai dû trouver une solution pour stocker en base de données les **Schedules** créés et surtout l'horaire choisi par l'utilisateur pour ses différents flux. Comme expliqué dans le chapitre **Analyse et Architecture** (section **Schedule**), j'ai mis au point une représentation d'une plage de diffusion en blocs d'une minute. Je ne voulais pas stocker l'entièreté de l'horaire mais plutôt enregistrer les flux avec une heure de début définie. J'ai donc trouvé cette solution : créer une table mettant en relation un **Schedule** et un **Flux** tout en spécifiant le numéro de son "bloc de départ" (qui correspond à son heure de début). Cette table s'appelle **ScheduledFlux** et ne représente pas vraiment un flux mais plutôt une entrée de calendrier.

Un **Schedule** doit également référencer ses flux sans heure de départ attribuée et ses flux de fallback afin d'être complet. Pour ce faire, des tables intermédiaires existent entre **Schedule** et **Flux** ainsi que **Schedule** et **FallbackFlux**.

Un **Schedule** doit pouvoir être activé et cette information doit être visible en base de données. Comme précisé dans le chapitre précédent, cette activation est représentée par la création d'une entité temporaire **RunningSchedule**, qui existe tant que le **Schedule** reste activé. Un **Schedule** ne possédant aucune référence vers des écrans, c'est le **RunningSchedule** qui contient ces informations. Outre cela, il ne contient qu'une référence vers son **Schedule** associé.

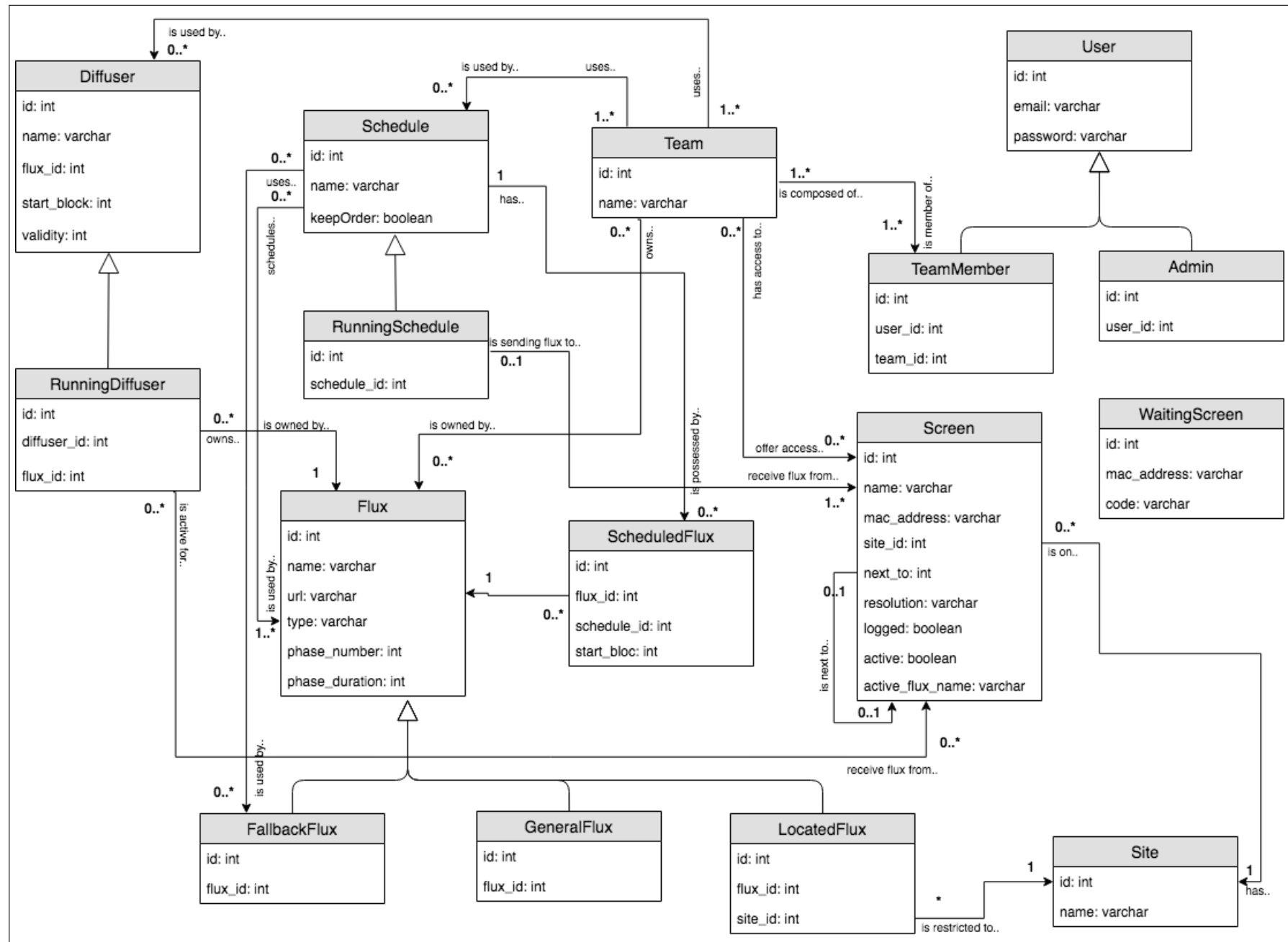


FIGURE 3 – Schéma de la base de donnée

## 5.4 Diffuser

En termes de base de données, les Diffusers sont très similaires aux Schedules. Comme on doit également pouvoir activer un Diffuser, la même technique est utilisée, avec un RunningDiffuser créé à l'activation qui contient une référence vers les écrans concernés ainsi que le flux diffusé. Ils ont par contre quelques attributs en plus qui déterminent leur comportement :

- **Startblock**, qui contient l'indice du bloc de départ du Diffuser
- **Validity**, un entier représentant la plage de validité du Diffuser.

Dans le cadre du projet, l'attribut Validity est implémenté en base de données et dans les modèles mais il n'est pas utilisé. Comme expliqué dans le chapitre **Réalisation**, le fonctionnement des Diffusers a changé en cours de route et l'implémentation d'une durée de validité n'a pas été faite dans la nouvelle architecture. L'implémentation pensée se mariait bien avec le système de blocs, *validity* devait définir un nombre de bloc pendant lequel le Diffuser était actif.

## 5.5 Ecrans

Pour les écrans, la base de données est assez simple. Chaque entrée de la table représente un écran physique et donc contient des informations sur son lieu, son matériel et son état.

Une des demandes explicitées pendant les discussions avec mon mentor était d'avoir la possibilité de spécifier un voisin pour nos écrans (certains écrans de la HEIG-VD étant côte-à-côte). Ceci afin de ne pas afficher deux fois le même flux mais deux différents. Si cette information est modélisée en base de données, elle n'est pas utilisée dans le cadre de l'application.

Il existe également la table WaitingScreen, qui représente un écran en attente d'enregistrement. Comme précisé dans le chapitre précédent, les nouveaux écrans inconnus du système qui essaient de s'authentifier reçoivent un code à fournir lors de leur ajout au système (en passant par l'interface). Entre le moment où le code est fourni et celui où l'écran a été ajouté, une entrée de cette table fait le lien entre l'adresse MAC du nouvel écran et le code fourni par le serveur.



## 6 Réalisation

### 6.1 Organisation des flux

Cette section se concentre sur la manière dont la diffusion de flux est gérée par le système, de l'analyse des Schedules à l'envoi d'Events aux écrans. Ce traitement se fait principalement à l'aide de trois objets, dont voici un schéma :

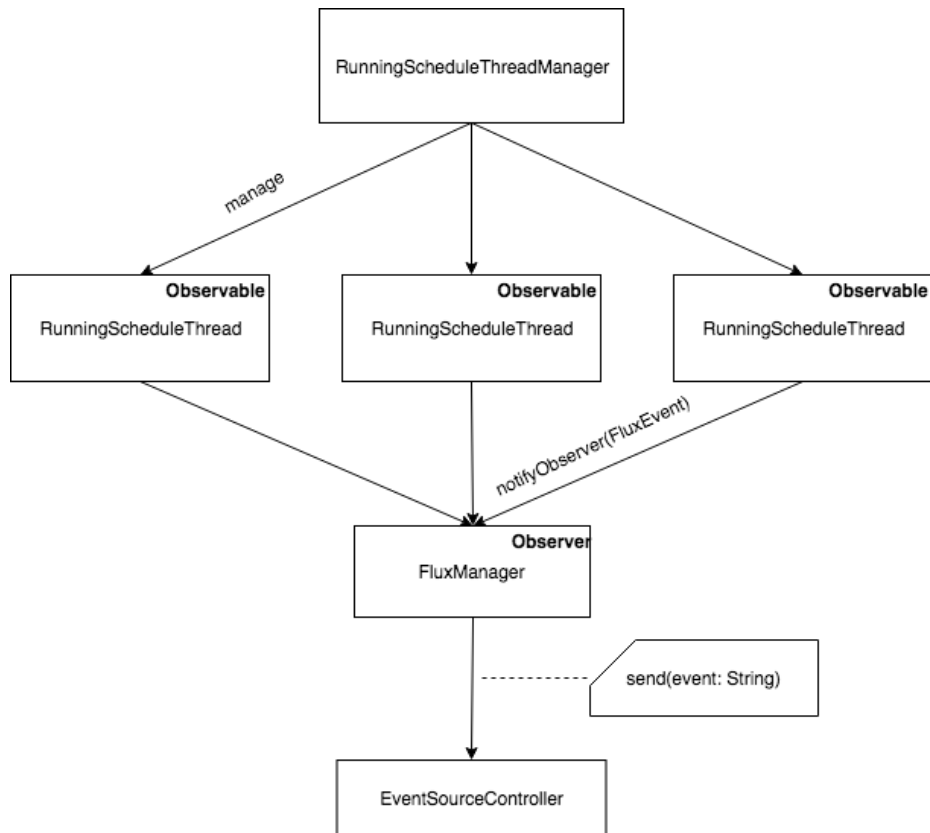


FIGURE 4 – Organisation des flux

En élément central, nous avons les `RunningSchedulesThreads`. Comme leur nom l'indique, c'est eux qui représentent l'exécution d'un `Schedule` actif. Ils sont gérés par un manager, qui permet de les activer, désactiver et de les récupérer. Ils génèrent des événements selon les flux de leur `Schedule` et les transmettent au `FluxManager` via le patron de conception **Observer**.

Le `FluxManager` est un Singleton dont les seules tâches sont de récupérer les Events générés par les différents `RunningSchedulesThreads`, construire les messages à envoyer (donc le type, l'url, les écrans concernés, etc) et les transmettre aux écrans. Il utilise pour cela la méthode `send(event : String)` fournie par l'`EventSourceController`.

Comme précisé dans le chapitre Analyse et Architecture, un système de "bloc-minute" est utilisé pour organiser la diffusion de flux en créant un "bloc-horaire" qui est utilisé par les `RunningSchedulesThreads` pour déterminer les flux à envoyer selon l'heure courante.

#### 6.1.1 Bloc-horaire

La première implémentation que j'ai faite de ces horaires utilisait une simple liste de flux qui, si elle me permettait de vérifier le bon fonctionnement du programme n'offraient que peu de souplesse pour les différentes fonctionnalités à implémenter. Il a donc été choisi de représenter une plage d'affichage comme un ensemble de bloc et j'ai choisi comme valeur temporelle une minute afin de simplifier le processus et d'offrir une plus grande granularité à l'utilisateur.

Concrètement, un "bloc-horaire" est une `Map<Integer, Integer>` qui associe à chaque index de bloc l'id du flux programmé à cette heure-ci ou -1 afin de représenter une absence de flux. Il est construit à chaque fois qu'un `Schedule` est activé à partir de ses données puis fourni au `RunningSchedulesThread` associé. On peut voir dans la figure suivante la manière dont cet horaire est généré :

```

1 public HashMap<Integer, Integer> getTimeTable(Schedule schedule) {
2     List<ScheduledFlux> scheduledFluxes = scheduleRepository.
        getAllScheduledFluxesByScheduleId(schedule.getId());
3     Flux lastFlux = new Flux();
4     long lastFluxDuration = 0;
5     boolean noFluxSent;
6
7     HashMap<Integer, Integer> timetable = new HashMap<>();
8     for (int i = 0; i < blockNumber; i++) {
9         noFluxSent = true;
10
11         if (lastFluxDuration != 0) {
12             lastFluxDuration--;
13             timetable.put(i, lastFlux.getId());
14         }
15         else {
16             for (ScheduledFlux sf : scheduledFluxes) {
17                 // si un flux doit commencer a ce bloc
18                 if (sf.getStartBlock().equals(i)) {
19                     Flux flux = fluxRepository.getById(sf.getFluxId());
20                     lastFlux = flux;
21                     lastFluxDuration = flux.getTotalDuration() - 1;
22                     timetable.put(i, flux.getId());
23                     noFluxSent = false;
24                     scheduledFluxes.remove(sf);
25                     break;
26                 }
27             }
28             if (noFluxSent) {
29                 // aucun flux ne commence a ce bloc
30                 timetable.put(i, -1);
31             }
32         }
33     }
34     return timetable;
35 }

```

Listing 1 – Création du bloc-horaire

On commence par récupérer les flux avec heure de début du Schedule concerné, puis on remplit l'horaire bloc par bloc en vérifiant à chaque fois si un flux est prévu pour le bloc courant.

Ce système offre principalement deux avantages :

- Il est très facile de modifier le "bloc-horaire" d'un Schedule actif pour en modifier le comportement (changer de flux, en rajouter ou même en supprimer) sans devoir arrêter le programme.
- La reprise automatique de l'exécution des Schedules actifs est elle aussi très simple : le système inspecte la base de données au démarrage et si elle contient des RunningSchedules, il génère et lance les threads associés.

### 6.1.2 RunningScheduleThread

C'est vraiment dans cette classe que la logique de "scheduling" est implémentée. Voici une version simplifiée de sa méthode *run()* mais qui garde la même logique fondamentale :

```

1 public void run() {
2     while (running) {
3
4         DateTime dt = new DateTime();
5         int hours = dt.getHourOfDay();
6         int minutes = dt.getMinuteOfHour();
7
8         // si on est dans la plage horaire active
9         if (hours >= beginningHour && hours < endHour) {
10             int blockIndex = getBlockNumberOfTime(hours, minutes);
11
12             do {

```

```

13     Flux currentFlux = fluxRepository.getById(timetable.get(blockIndex++));
14
15     // si un flux est prévu pour ce bloc
16     if (currentFlux != null) {
17         sendFluxEventAsGeneralOrLocated(currentFlux);
18     }
19     // sinon on choisit un flux sans heure de debut
20     else if (!unscheduledFluxIds.isEmpty()) {
21         sendUnscheduledFlux(blockIndex);
22     }
23     // sinon on choisit un flux de fallback
24     else if (!fluxRepository.getAllFallbackIdsOfSchedule(runningSchedule.getScheduleId
25     ()).isEmpty()) {
26         sendFallbackFlux(blockIndex);
27     }
28     // sinon on envoie le flux d'erreur
29     else {
30         sendFluxEvent(fluxRepository.getByName(WAIT_FLUX), screens);
31     }
32     // sleep pendant 1 minutes
33     try {
34         if (Thread.currentThread().isInterrupted()) {
35             throw new InterruptedException("Thread interrupted");
36         }
37         Thread.sleep((long) blockDuration * 60000);
38     } catch (InterruptedException e) {
39         e.printStackTrace();
40     }
41     } while (blockIndex < blockNumber && running);
42 }
43 }
44 }

```

Listing 2 – Eventsource Java

Lorsqu'il débute, le thread récupère l'heure actuelle et vérifie qu'il est dans les bornes de la plage horaire. Si oui, il récupère l'indice du bloc courant puis commence son exécution. Dans le code ci-dessus ne figure pas toutes les opérations effectuées normalement (par soucis de concision), mais il donne une assez bonne idée de son fonctionnement.

On peut résumer l'algorithme de choix de flux ainsi :

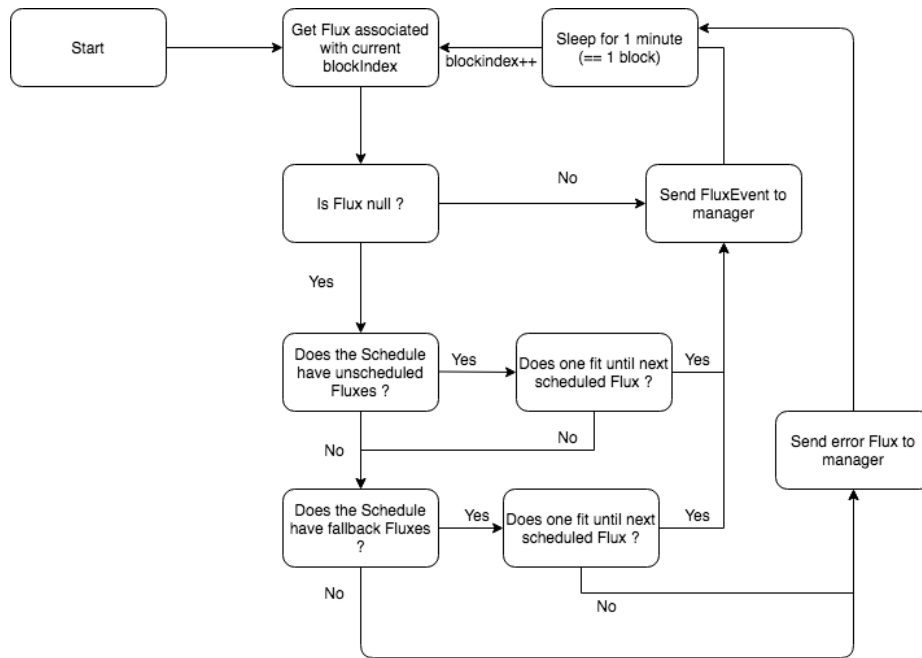


FIGURE 5 – Algorithme de scheduling des flux

Un Schedule voudra toujours afficher en priorité un flux ayant une heure de début. Le Running-ScheduleThread va donc vérifier si le bloc courant a un flux associé. Si oui, cela veut dire qu'un ScheduleFlux existe pour cette heure-ci et le flux associé est envoyé aux écrans. Si non, le thread essaie de "caser" un des flux sans heure de début dans l'espace restant dans le bloc-horaire jusqu'au prochain ScheduledFlux. S'il n'y arrive pas, il envoie soit un des flux de fallback (si le Schedule en possède), soit un flux prédéterminé qui informe l'écran qu'il recevra bientôt des informations.

On peut observer à la ligne 18 du Listing 2 un appel à la fonction *sendFluxEventAsGeneralOrLocated()*. Elle sert de point d'envoi des Events au FluxManager et s'assure que les différents flux envoyés soient correctement dispatchés aux écrans selon leur contenu. Par exemple, lors de l'envoi d'un flux localisé, tout les écrans n'étant pas sur ce site mais faisant partie du Schedule vont recevoir soit un des flux de fallback s'il en existe, soit un flux d'erreur.

### 6.1.3 FluxManager

Le FluxManager est l'entité responsable de regrouper tout les Events générés par les Schedule pour les transmettre au contrôleur chargé de les envoyer aux écrans. C'est un Runnable Singleton qui est créé par injection de dépendance au démarrage de l'application et dont les contrôleurs ou autres objets peuvent obtenir une référence, par injection de dépendance à nouveau. Pour représenter l'événement d'un flux, j'utilise un objet FluxEvent, qui est composé d'un flux et de la liste des adresses MAC des écrans concernés par cet événement.

C'est également le FluxManager qui construit le contenu de l'Event qui sera envoyé aux écrans (voir chapitre Analyse et Architecture, section Events). Présenté ci-dessous, sa fonction *run()* simplifiée :

```

1 public void run() {
2     while (running) {
3         // il y a des event a envoyer
4         if (!fluxEvents.isEmpty()) {
5             FluxEvent currentFlux = fluxEvents.remove(0);
6
7             boolean run = true;
8
9             do {
10                 eventController.send(
11                     currentFlux.getFlux().getType().toLowerCase() +
12                     "?" +
13                     currentFlux.getFlux().getUrl() +

```

```

14         "|" +
15         String.join(",", currentFlux.getMacs())
16     );
17
18     if (!fluxEvents.isEmpty()) {
19         currentFlux = fluxEvents.remove(0);
20         if (fluxEvents.isEmpty()) {
21             run = false;
22         }
23     }
24     else {
25         run = false;
26     }
27     } while (run);
28 }
29 // attend un peu avant de recommencer
30 else {
31     try {
32         Thread.sleep(2000);
33     } catch (InterruptedException e) {
34         e.printStackTrace();
35     }
36 }
37 }
38 }

```

Listing 3 – FluxManager

La logique de cette classe est elle aussi plutôt simple ; le FluxManager itère sur sa liste de FluxEvents et pour chacun d'entre eux, il donne l'ordre au contrôleur d'envoyer un Event aux écrans. Dans le cas où il n'a pas de FluxEvent à traiter, il *sleep* pendant quelques instants avant de recommencer la procédure.

#### 6.1.4 AutomatedScheduleStarter

Le code de cette classe est appelé au démarrage du programme afin de régénérer des RunningScheduleThreads pour chaque RunningSchedule dans la base de données. Grâce au système de blocs, il devient très facile de reprendre l'exécution après un arrêt ou une maintenance du serveur car il suffit de récupérer l'heure courante, de la faire correspondre à un numéro de bloc et d'utiliser ce numéro comme point de départ pour le RunningScheduleThread nouvellement créé. La manière de faire étant la même que pour l'activation d'un Schedule, il n'y a pas d'exemple fourni.

### 6.1.5 FluxChecker

Le code présenté dans cette section n'a pas été testé car le service ou micro-serveur n'existe pas. Il s'agit uniquement d'une ébauche visant à mieux expliquer mon idée pour cette fonctionnalité.

```
1 public class FluxChecker implements WSBodysReadables, WSBodysWritables {
2     private final WSClient ws;
3
4     @Inject
5     public FluxChecker(WSClient ws) {
6         this.ws = ws;
7     }
8     // Cette fonction fait une requete vers l'URL du micro-serveur
9     // en fournissant l'id du flux vise et le type de validation requise
10    // Le Json reçu est transforme en ValidityInterval pour tester les valeurs recues
11    // return true si la date courante est dans l'interval
12    public boolean checkIfFluxHasSomethingToDisplayByDateTime(Flux flux) {
13        final JsonNode[] jsonNode = new JsonNode[1];
14
15        ws.url("data check URL")
16            .addQueryParam("fluxId", String.valueOf(flux.getId()))
17            .addQueryParam("type", "time")
18            .get()
19            .thenApply(r -> jsonNode[0] = r.getBody(json()));
20
21        if (jsonNode[0] != null) {
22            ValidityInterval interval = Json.fromJson(jsonNode[0], ValidityInterval.class);
23            DateTime dt = new DateTime();
24
25            // true if date of today is in the interval returned
26            return interval.beginning.compareTo(dt) <= 0 &&
27                interval.end.compareTo(dt) >= 0;
28        }
29        return false;
30    }
31 }
```

Listing 4 – FluxManager

Cette fonction sera utilisée par les `RunningSchedulesThreads` mais elle ne doit être appelée non pas au moment d'afficher un flux. Comme la requête pourrait potentiellement prendre du temps ou échouer, ils doivent effectuer cette vérification en amont, par exemple pour le prochain flux prévu.

## 6.2 Contrôleurs

La plupart des contrôleurs de l'application servent uniquement à exécuter des opérations CRUD, mais certains offrent des fonctionnalités plus poussées qui sont décrites dans les sections suivantes.

### 6.2.1 EventSourceController

Lors des discussions préalables avec mon mentor, j'avais été prévenu que dans la version 2.7 de Play Java, les Eventources ne fonctionnaient pas car les informations de session étaient perdues lors d'un mapping. Travaillant avec la version 2.7.1 (sensée avoir résolu ce problème), je suis donc parti sur une implémentation en Java. J'ai d'abord pensé avoir réussi, car j'observais un comportement normal, mais en analysant de manière plus approfondie les échanges clients-serveurs lors d'une séance avec mon mentor, nous nous sommes aperçu que la connexion Eventsource était recrée toutes les 3 secondes, quand le client essayait de se reconnecter au serveur. Il a donc été nécessaire de passer à une version en Scala.

Ci-dessous une version simplifiée de l'ancien code Java :

```
1 @Singleton
2 public class EventSourceController extends Controller implements Observer {
3     private static Source<String, ?> source;
4
5     @Override
6     public synchronized void update(Observable o, Object arg) {
7         if (source == null) {
8             source = Source.tick(Duration.ZERO, Duration.ofSeconds(5), "tick");
9         }
10        else {
11            List<String> list = new ArrayList<>();
12            list.add((String) arg);
13            Source<String, ?> s = Source.from(list);
14            source.merge(s);
15        }
16    }
17
18    public Result events() {
19        final Source<EventSource.Event, ?> eventSource;
20
21        return ok().chunked(source
22            .map(EventSource.Event::event)
23            .via(EventSource.flow()))
24            .as(Http.MimeTypes.EVENT_STREAM);
25    }
26 }
```

Listing 5 – Eventsource Java - EventSourceController.java

L'idée de ce contrôleur était de mettre à jour une source à chaque fois qu'un Event était envoyé grâce au patron de conception *Observer*.

Après un peu de recherche sur les différents moyen d'implémenter une utilisation des Eventsource en Scala, j'ai choisi une solution basée sur les Akka Actor. Elle semblait la plus simple et la plus adaptée à mon problème, car elle permet de représenter chaque écran par un acteur. Le contrôleur équivalent en Scala :

```

1 @Singleton
2 class EventSourceController @Inject() (system: ActorSystem,
3                                     cc: ControllerComponents)
4                                     (implicit executionContext: ExecutionContext)
5 extends AbstractController(cc) {
6
7     private[this] val manager = system.actorOf(EventActorManager.props)
8     implicit def pair[E]: EventNameExtractor[E] = EventNameExtractor[E](_ => Some("test1"))
9
10    def send(event: String) = {
11        print("Send event to screens : " + event)
12        manager ! SendMessage(event)
13        Ok
14    }
15
16    def index = Action {
17        Ok(eventsource.render())
18    }
19
20    def events = Action {
21        val source =
22            Source
23                .actorRef[String](32, OverflowStrategy.dropHead)
24                .watchTermination() { case (actorRef, terminate) =>
25                    manager ! Register(actorRef)
26                    terminate.onComplete(_ => manager ! UnRegister(actorRef))
27                    actorRef
28                }
29
30        val eventSource = Source.fromGraph(source.map(EventSource.Event.event))
31        Ok.chunked(eventSource via EventSource.flow).as(ContentTypes.EVENT_STREAM)
32    }
33 }

```

Listing 6 – Eventsource Scala - EventSourceController.scala

On peut voir que ce contrôleur est au final très simple. Il offre deux actions principales : la possibilité pour un écran de s'enregistrer auprès du manager grâce à la méthode *events()* et un moyen pour le système d'envoyer un Event aux écrans avec la méthode *send(event : String)*. Il utilise pour ce faire un ActorSystem, construit à l'aide d'une sous-classe d'Actor, créée dans le cadre du projet. En voici l'implémentation :

```

1 class EventActor extends Actor {
2     private[this] val actors = mutable.Set.empty[ActorRef]
3
4     def receive = {
5         case Register(actorRef) => actors += actorRef
6         case UnRegister(actorRef) => actors -= actorRef
7         case SendMessage(message) => actors.foreach(_ ! message)
8     }
9 }
10
11 object EventActorManager {
12     def props: Props = Props[EventActor]
13
14     case class SendMessage(message: String)
15     case class Register(actorRef: ActorRef)
16     case class UnRegister(actorRef: ActorRef)
17 }

```

Listing 7 – Akka Actor Scala - EventSourceController.scala



## 6.2.2 ScreenController

### 6.2.2.1 Authentification

C'est par ce contrôleur que passe les écrans souhaitant s'authentifier auprès du système et ainsi recevoir des flux. La logique du comportement étant décrite dans le chapitre **Analyse et Architecture**, seul l'implémentation sera évoquée ici.

Voici la version simplifiée de cette fonction :

```
1 public Result authentication(Http.Request request) {
2     String macAdr = request.queryString().get("mac")[0];
3     Screen screen = getScreenByMacAddress(macAdr);
4
5     // Ecran inconnu du systeme
6     if (screen == null) {
7
8         // L'ecran a deja essayer de s'identifier
9         if (getWaitingScreenByMacAddress(macAdr) != null) {
10             return ok(screen_code.render(getWaitingScreenByMacAddress(macAdr).getCode()));
11         }
12
13         String code = screenRegisterCodeGenerator();
14         add(new WaitingScreen(code, macAdr));
15
16         // Envoi du code
17         return ok(screen_code.render(code));
18     }
19     // Ecran connu du systeme
20     else {
21         // Pas de Schedule actif pour cet ecran
22         if (getRunningScheduleIdByScreenId(screen.getId()) == null) {
23             return redirect(routes.ErrorPageController.noScheduleView());
24         }
25
26         // Ecran actif
27         if (!screen.isLogged()) {
28             screen.setLogged(true);
29         }
30         return ok(eventsource.render()).withCookies(
31             Http.Cookie.builder("mac", macAdr)
32                 .withHttpOnly(false)
33                 .build(),
34             Http.Cookie.builder("resolution", screen.getResolution())
35                 .withHttpOnly(false)
36                 .build());
37     }
38 }
```

Listing 8 – Authentification des écrans - ScreenController.java

On s'aperçoit que son fonctionnement est assez simple : si l'écran n'a pas encore été ajouté au système, une entité WaitingScreen est créée (ou existe déjà) avec la même adresse MAC et le code est envoyé à l'écran. Sinon, l'écran est simplement redirigé vers l'EventSourceController après lui avoir rajouté des cookies. Dans le cas où le serveur ne détecte aucun Schedule actif pour cet écran, il est redirigé vers une page d'erreur.

### 6.2.2.2 Désactivation

Il pourrait arriver de vouloir arrêter l'affichage sur un écran en particulier sans pour autant stopper le Schedule associé et continuer l'affichage sur les autres écrans. J'ai donc du mettre au point une manière d'enlever un écran de la liste des écrans concernés par un Schedule. En cherchant des moyen de faire ceci, j'ai réalisé que j'avais des problèmes dans mon code que je n'avais pas remarqué jusqu'à cet instant. Pour des raisons que je n'ai pas tout a fait comprises, lorsque je mettais à jour des attributs de mes threads pour en modifier le comportement, les changements n'étaient pas immédiats ou alors temporaires. Il a donc fallu trouver une autre solution. J'ai décidé qu'au lieu de changer les

threads courants, j'allais les arrêter puis en créer des nouveaux avec les nouvelles valeurs et démarrer ceux-ci. A nouveau, mon système de blocs m'a bien rendu service car la reprise par le nouveau thread de l'exécution du Schedule se fait très simplement. La fonction permettant ceci est disponible dans une version simplifiée ci-dessous :

```
1 public Result deactivate(Http.Request request, String mac) {
2     Screen screen = servicePicker.getScreenService().getScreenByMacAddress(mac);
3     Integer rsId = scheduleService.getRunningScheduleOfScreenById(screen.getId());
4     RunningSchedule rs = scheduleService.getRunningScheduleById(rsId);
5
6     // ecran actif
7     if (rs != null) {
8         // enleve l'ecran courant du RunningSchedule concerne
9         List<Integer> screenIds =
10             scheduleService.getAllScreenIdsOfRunningScheduleById(rs.getId());
11         screenIds.remove(screen.getId());
12         rs.setScreens(screenIds);
13
14         RunningScheduleThread rst =
15             threadManager.getServiceByScheduleId(rs.getScheduleId());
16
17         List<Screen> screenList =new ArrayList<>();
18         for (Integer screenId: screenIds) {
19             screenList.add(screenService.getScreenById(screenId));
20         }
21         // stop l'ancien thread
22         rst.abort();
23         threadManager.removeRunningSchedule(rs.getScheduleId());
24
25         Schedule schedule = scheduleService.getScheduleById(rs.getScheduleId());
26
27         // on cree un nouveau thread a partir de l'ancien
28         RunningScheduleThread task = new RunningScheduleThread(
29             rs,
30             screenList,
31             new ArrayList<>(schedule.getFluxes()),
32             rst.getTimetable(),
33             fluxRepository,
34             fluxChecker,
35             schedule.isKeepOrder());
36
37         task.addObserver(fluxManager);
38         threadManager.addRunningScheduleThread(rs.getScheduleId(), task);
39     }
40     // mise a jour du RunningSchedule associe
41     scheduleService.update(rs);
42
43     return index(request);
44 }
45 }
```

Listing 9 – Désactivation des écrans - ScreenController.java

## 6.2.3 ScheduleController

Ce contrôleur, en plus d'offrir des opérations CRUD sur les Schedules, permet de les activer et désactiver. Ce sont ces deux fonctionnalités qui seront décrites dans les sections suivantes. Le code présenté est à nouveau simplifié; les diverses vérifications faites sur les données ou sur l'état du programme ne sont par exemple pas présentes.

### 6.2.3.1 Activation

L'activation d'un Schedule se fait en deux étapes principales. On crée d'abord un objet RunningSchedule à partir du Schedule à activer. On itère aussi à travers les écrans concernés par ce Schedule

afin mettre à jour quelques valeurs. Dans la deuxième partie (ligne 25), ce RunningSchedule sert ensuite à créer un RunningScheduleThread qui est directement lancé par le manager.

```
1 public Result activate(Http.Request request) {
2     final Form<ScheduleData> boundForm = form.bindFromRequest(request);
3     ScheduleData data = boundForm.get();
4     Schedule schedule = scheduleService.getScheduleByName(data.getName());
5
6     // On cree le RunningSchedule
7     RunningSchedule rs = new RunningSchedule(schedule);
8     rs = scheduleService.create(rs);
9
10    // creation de la liste des ecrans concernes par le Schedule
11    List<Screen> screens = new ArrayList<>();
12    for (String screenMac : data.getScreens()) {
13        Screen screen = screenService.getScreenByMacAddress(screenMac);
14        rs.addToScreens(screen.getId());
15        screen.setRunningscheduleId(rs.getId());
16        screen.setActive(true);
17
18        screens.add(screen);
19
20        screenService.update(screen);
21    }
22    scheduleService.update(rs);
23
24    // creation du thread et ajout du FluxManager comme observateur
25    RunningScheduleThread service2 = new RunningScheduleThread(
26        rs,
27        screens,
28        new ArrayList<>(schedule.getFluxes()),
29        timeTableUtils.getTimeTable(schedule),
30        fluxRepository,
31        fluxChecker,
32        schedule.isKeepOrder());
33
34    service2.addObserver(fluxManager);
35
36    // activation du thread
37    threadManager.addRunningSchedule(schedule.getId(), service2);
38
39    return index(request);
40 }
```

Listing 10 – Activation d'un Schedule - ScheduleController.java

### 6.2.3.2 Désactivation

La désactivation est quant à elle bien plus simple, car elle se résume à stopper le thread correspondant ainsi qu'à supprimer de la base de données le RunningSchedule visé.

```
1 public Result deactivate(String name, Http.Request request) {
2     ScheduleService scheduleService = servicePicker.getScheduleService();
3     Schedule schedule = scheduleService.getScheduleByName(name);
4
5     RunningSchedule rs = scheduleService.getRunningScheduleByScheduleId(schedule.getId());
6
7     // supprime de la BD
8     scheduleService.delete(rs);
9
10    // stop le thread correspondant
11    threadManager.removeRunningSchedule(schedule.getId());
12
13    return index(request);
14 }
```

## 6.2.4 DiffuserController

Ce contrôleur offre des opération CRUD sur les Diffusers et permet de les activer et désactiver. L'activation d'un Diffuser a été une des fonctionnalités les plus dures à implémenter. Je voulais en effet les connecter entièrement avec mes Schedules et les faire utiliser le même système d'horaire. Une fois celui-ci défini et testé avec les Schedules, j'ai commencé à réfléchir aux Diffusers. Je me suis aperçu qu'il fallait les différencier en deux types : standard et urgent. Pour le type standard, le flux diffusé est rajouté dans l'horaire du Schedule associé à l'écran si possible et pour l'urgent, le flux est diffusé immédiatement puis l'exécution reprend son cours habituel.

C'est en tout cas ce que j'ai fait avant de réaliser que mon système ne fonctionnait pas. En effet, pendant la rédaction de ce rapport, j'ai remarqué qu'un Schedule peut être activé sur plusieurs écrans et donc modifier tout un Schedule pour un écran générera des effets de bords en modifiant l'affichage sur des écrans non-choisis. Mes deux types de Diffusers utilisant la même logique sous-jacente (à savoir modifier l'horaire d'un Schedule actif), ils sont donc tout les deux inutiles. Je suis passé à côté de ce détail pendant tout le semestre car pendant mes tests je travaillais avec des Schedule contenant peu d'écrans et ce cas de figure n'était jamais arrivé.

Ma logique initiale était fautive ; je n'aurais pas dû essayer de mixer les Schedules et Diffusers, trop de problèmes sont survenus à cause de cela. Afin de palier un peu à ce problème, j'ai effectué des modifications de dernière minute afin d'avoir quand même un Diffuser fonctionnel, en tout cas une première implémentation.

L'activation d'un Diffuser est désormais représentée par l'ajout dans la base de donnée d'un Running-Diffuser. A chaque fois qu'un RunningScheduleThread envoie des Events aux écrans, il vérifie si un RunningDiffuser existe pour ces écrans. Si oui, il leur envoie le flux diffusé et si non, il continue l'exécution normale de son Schedule. Grâce à cela, une séparation est faite entre Diffuser et Schedule, le premier étant prioritaire sur l'autre.

### 6.2.4.1 Activation/désactivation

Dans l'extrait de code suivant figurent des versions simplifiées des nouvelles fonctions d'activation et désactivation des Diffusers. Comme précisé précédemment, elles consistent uniquement à ajouter ou retirer un objet RunningDiffuser de la base de données.

```

1 // activation
2 public Result activate(Http.Request request) {
3     final Form<DiffuserData> boundForm = form.bindFromRequest(request);
4     DiffuserData data = boundForm.get();
5     Diffuser diffuser = diffuserService.getDiffuserByName(data.getName());
6
7     // ids des ecrans concernés par le diffuser
8     Set<Integer> screenIds = new HashSet<>();
9     for (String mac: data.getScreens()) {
10         Screen screen = screenService.getScreenByMacAddress(mac);
11         if (screen == null) {
12             return activateViewWithErrorMessage(data.getName(), request, "Screen MAC address
13             does not exists");
14         }
15         screenIds.add(screen.getId());
16     }
17
18     // creation du diffuser
19     Flux diffusedFlux = fluxService.getFluxById(diffuser.getFlux());
20     RunningDiffuser rd = new RunningDiffuser(diffuser);
21     rd.setDiffuserId(diffuser.getId());
22     rd.setScreens(new ArrayList<>(screenIds));
23     rd.setFluxId(diffusedFlux.getId());
24     diffuserService.create(rd);
25
26     return index(request);
27 }
```

```

27
28 // desactivation
29 public Result deactivate(Http.Request request, String name) {
30     Diffuser diffuser = diffuserService.getDiffuserByName(name);
31
32     RunningDiffuser rd = diffuserService.getRunningDiffuserByDiffuserId(diffuser.getId());
33     diffuserService.delete(rd);
34     return index(request);
35 }

```

Listing 12 – Activation d'un Diffuser - DiffuserController.java

#### 6.2.4.2 Détection d'un Diffuser actif

Au lieu de modifier les Schedules avec les Diffusers, l'application peut maintenant détecter qu'un RunningDiffuser est actif pour un écran auquel il s'apprêtait à envoyer un flux et ainsi modifier son comportement pour envoyer le flux diffusé et non le flux prévu.

```

1 private void sendFluxEvent(Flux flux, List<Screen> screenList) {
2     // verifie si un Diffuser est actif pour un des ecran de la liste
3     // si oui, envoie le flux diffuse et enleve l'ecran de la liste
4     for (Screen screen: screenList) {
5         Integer rdId = diffuserService.getRunningDiffuserIdByScreenId(screen.getId());
6
7         if (rdId != null) {
8             RunningDiffuser rd = diffuserService.getRunningDiffuserById(rdId);
9             List<Screen> screens = new ArrayList<>();
10            screens.add(screen);
11            screenList.remove(screen);
12            FluxEvent diffusedEvent = new FluxEvent(fluxService.getFluxById(rd.getFluxId()),
13            screens);
14            setChanged();
15            notifyObservers(diffusedEvent);
16        }
17
18        // envoie l'event prévu aux ecrans restants
19        FluxEvent event = new FluxEvent(flux, screenList);
20        lastFluxEvent = event;
21        setChanged();
22        notifyObservers(event);
23    }

```

Listing 13 – Détection d'un Diffuser actif lors de l'envoi de Flux - RunningScheduleThread.java

Cette fonction est le point de sortie final de la classe RunningScheduleThread, où le FluxEvent est envoyé au FluxManager.

## 6.3 DAOs

Dans cette section sera décrite l'implémentation des services d'accès à la base de données, ou **DAO** (pour **DataAccessObject**). Voici un schéma simple en décrivant l'architecture :

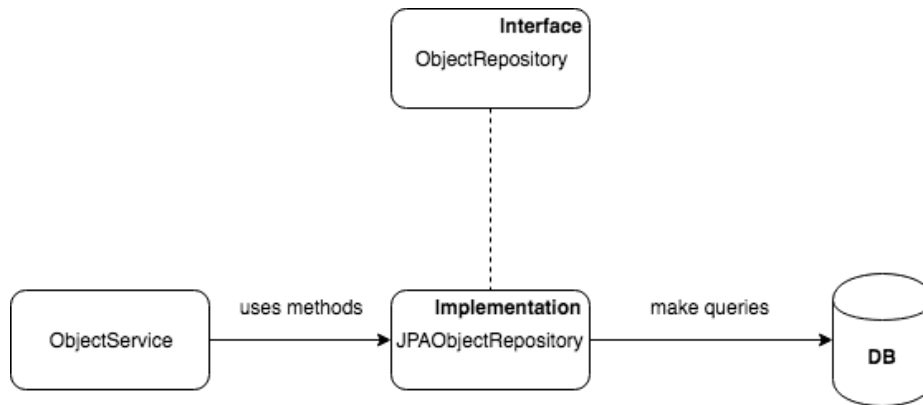


FIGURE 6 – Schéma des services d'accès aux données

Dans cette figure, l'objet *ObjectService* représente le service utilisé pour effectuer des opérations de base de données. Utiliser cette structure pour les DAO est recommandée par Play car elle regroupe la logique des fonctions d'accès aux données au même endroit et elle facilite grandement la mise en place de tests à base de mocks.

### 6.3.1 Requêtes SQL

Les requêtes à la base de données sont effectuées dans le cadre d'une transaction, ce qui permet de roll-back en cas d'erreur et ainsi de ne pas modifier la base de données avec de fausses valeurs. J'utilise l'injection de dépendance afin d'obtenir un objet JPAApi, qui offre des moyens simples et efficaces d'écrire des requêtes SQL. On peut voir ci-dessous un exemple des méthodes standards d'insertion, mise-à-jour, récupération et de suppression :

```
1 @Singleton
2 public class JPAUserRepository implements UserRepository {
3
4     private final JPAApi jpaApi;
5
6     @Inject
7     public JPAUserRepository(JPAApi jpaApi) {
8         this.jpaApi = jpaApi;
9     }
10
11     @Override
12     public User create(User user) {
13         jpaApi.withTransaction(entityManager -> {
14             entityManager.persist(user);
15         });
16         return user;
17     }
18
19     @Override
20     public User get(String email, String password) {
21         return jpaApi.withTransaction(entityManager -> {
22             String mail = "'" + email + "'";
23             String pw = "'" + password + "'";
24             Query query = entityManager.createNativeQuery(
25                 "SELECT * FROM users WHERE email = " + mail
26                 + " and password = " + pw, User.class);
27             try {
28                 return (User) query.getSingleResult();
29             } catch (NoResultException e) {
30                 return null;
31             }
32         });
33     }
34 }
```

```

31     }
32     });
33 }
34
35 @Override
36 public void delete(User user) {
37     jpaApi.withTransaction(entityManager -> {
38         entityManager.remove(entityManager.contains(user) ? user : entityManager.merge(
39             user));
40     });
41 }
42
43 @Override
44 public User update(User user) {
45     jpaApi.withTransaction(entityManager -> {
46         entityManager.merge(user);
47     });
48     return user;
49 }

```

Listing 14 – Exemples de requêtes SQL avec JPA

On peut observer que la ressource JPA est obtenue par injection de dépendance et offre des raccourcis très pratiques avec son `EntityManager`, par exemple pour insérer des données avec la méthode `persist()`. Il est bien sûr aussi possible d'écrire des requêtes natives comme pour la fonction `get(String email, String password)`.

### 6.3.2 Services

Les services regroupent les fonctions d'accès aux données par type de données. Par exemple, un `FluxService` offre des méthodes d'ajout, de suppression mais aussi une fonction qui retourne la liste de tous les flux d'une `Team` précise. Ils contiennent aussi des fonctions de "casting", par exemple pour récupérer non pas une liste de `Flux` mais plutôt une liste de `FluxData`, utile pour l'affichage de vues. Dans le cadre de cette application, ils sont générés à la demande par un objet `ServicePicker`, qui utilise l'injection de dépendance pour créer les différents service selon le besoin. Ci-dessous un extrait de code montrant la manière dont les services sont fournis par la classe `ServicePicker.java`

```

1 public class ServicePicker {
2     @Inject
3     TeamRepository teamRepository;
4
5     // + tout les autres repositories ...
6
7     public TeamService getTeamService() {
8         return new TeamService(teamRepository);
9     }
10
11     // + les autres ServiceGetters
12 }
13
14 public class TeamService {
15
16     private final TeamRepository teamRepository;
17
18     public TeamService(TeamRepository teamRepository) {
19         this.teamRepository = teamRepository;
20     }
21
22     public Team getTeamByName(String name) {
23         return teamRepository.getByName(name);
24     }
25
26     // + autres fonctions
27 }

```

### 6.3.3 Implémentation Java des entités BD

Pour lier une classe Java et une table de base de données SQL, on utilise des annotations pour expliciter les relations entre attributs et colonnes. Dans l'extrait de code suivant, on peut voir une version simplifiée du modèle Schedule :

```

1 @Entity
2 @Table(name="schedule", schema="public")
3 public class Schedule {
4     @Id
5     @GeneratedValue(strategy = GenerationType.IDENTITY)
6     @Column(name="schedule_id")
7     private Integer id;
8
9     @Column(name="name")
10    private String name;
11    @ElementCollection(fetch = FetchType.EAGER)
12    private Set<Integer> fluxes;
13
14    public Schedule() {
15
16    }
17 }

```

Listing 16 – Exemple de modèle Java - Schedule.java

Plusieurs choses sont à noter :

- Des annotations sont utilisées pour permettre à Play de faire correctement les liens avec les tables et colonnes de la base de données. Par exemple les `@Table` et `@Column`, où le nom est indiqué.
- Comme les IDs des objets en base de données sont du type SERIAL, la stratégie de la génération de l'identité doit être `GenerationType.IDENTITY`.
- L'attribut *fluxes* représente la liste des flux d'un Schedule et doit donc être annoté avec un `@ElementCollection`. Cet attribut ne correspondant pas immédiatement avec une colonne de la table Schedule, il faut lui préciser sa stratégie de récupération de données comme *EAGER*. En effet, le comportement standard est du type *LAZY* et comme les données n'existent pas dans la table Schedule mais dans la table schedule\_fluxes, une évaluation paresseuse rendra des données nulles ou une erreur.
- Il faut fournir un constructeur sans paramètres.



## 6.4 Restriction d'accès

Certaines vues ou fonctionnalités peuvent être limitées à une ou plusieurs catégories d'utilisateurs. Ces restrictions ont été implémentées en utilisant des Actions. Une Action est basiquement une fonction qui analyse les paramètres d'une requête et produit un résultat qui est renvoyé au client. Comme il s'agit d'une fonction, il est très facile de composer plusieurs actions ensemble. On peut dès lors construire une action qui va vérifier les paramètres de la requête pour récupérer un cookie d'authentification et selon la valeur de ce cookie, passer à l'Action suivante ou être redirigé ailleurs. Comme un exemple est bien plus parlant, voici l'Action vérifiant si l'utilisateur courant est un admin :

```
1 public class AdminAuthentificationAction extends play.mvc.Action.Simple {
2     private HttpExecutionContext executionContext;
3     private final Form<UserData> form;
4
5     @Inject
6     public AdminAuthentificationAction(HttpExecutionContext executionContext,
7                                       FormFactory formFactory) {
8         this.executionContext = executionContext;
9         this.form = formFactory.form(UserData.class);
10    }
11
12    public CompletionStage<Result> call(Http.Request req) {
13        if (req.cookie("email") != null) {
14            if (userService.getAdminByUserEmail(req.cookie("email").value()) != null) {
15                return delegate.call(req);
16            }
17        }
18        return CompletableFuture.supplyAsync(() -> "", executionContext.current())
19            .thenApply(result -> badRequest(user_login.render(form,
20                "You must be an admin to access this page")));
21    }
22 }
```

Listing 17 – Admin Authentification Action

On peut voir que la méthode principale de cette classe, *call()*, vérifie simplement si le cookie "email" existe, et dans l'affirmative, s'il correspond à l'email d'un admin. Si oui, il continue son exécution et passe la main (délègue) à l'Action suivante. Si non, il renvoie à la page de login avec un message d'erreur. Cette Action est ensuite utilisée de la manière suivante :

```
1 // TeamController.java
2 @With(AdminAuthentificationAction.class)
3 public Result index() {
4     return ok(team_page.render(servicePicker.getTeamService().getAllTeams(),
5         null));
6 }
```

Listing 18 – Utilisation d'une Action custom

L'annotation **@With** permet de spécifier que l'on souhaite passer par une autre Action avant d'exécuter le contenu de la méthode *index()*.

## 6.5 Triggers

Comme la majorité des tables intermédiaires (par exemple `team_fluxes`) ont des références non nulles vers des lignes d'autres tables, la suppression de l'une de ces lignes entraînerait des erreurs de transaction et l'annulation de l'opération en cours. J'ai donc du mettre en place des triggers SQL qui s'occupent d'effacer les lignes des tables intermédiaires lorsque nécessaire. Les entités concernées sont toutes celles possédant une clé étrangère non nulle. Ces triggers étant sensiblement tous les mêmes, un seul exemple est fourni :

```
1  -- Fonction
2  DROP FUNCTION IF EXISTS delete_fluxes_of_schedule();
3  CREATE FUNCTION delete_fluxes_of_schedule()
4  RETURNS TRIGGER
5  AS $$
6  BEGIN
7      DELETE FROM schedule_fluxes
8      WHERE schedule_schedule_id = OLD.schedule_id;
9
10     DELETE FROM schedule_fallbacks
11     WHERE schedule_schedule_id = OLD.schedule_id;
12
13     DELETE FROM schedule_scheduledfluxes
14     WHERE schedule_schedule_id = OLD.schedule_id;
15
16     DELETE FROM scheduled_flux
17     WHERE schedule_id = OLD.schedule_id;
18 RETURN OLD;
19 END;
20 $$
21 LANGUAGE plpgsql;
22
23 -- Trigger
24 CREATE TRIGGER on_schedule_delete BEFORE DELETE
25 ON schedule
26 FOR EACH ROW EXECUTE PROCEDURE delete_fluxes_of_schedule();
```

Listing 19 – Triggers SQL - Suppression

Le trigger est appelé avant l'opération de DELETE pour qu'il s'exécute avant que l'erreur n'apparaisse.

J'en utilise également un autre qui se déclenche cette fois après une insertion dans la table Team-Member pour ajouter le nouveau membre dans la liste des membres de la Team correspondante.

J'ai conscience que le projet étant plutôt axé sur la modélisation et la base de données, il aurait bénéfique d'avoir des triggers servant à la validation/vérification des données insérées afin d'être entièrement sûr de la cohérence de la base.

## 6.6 Vues

Ce chapitre décrit la manière dont les différentes vues de l'application ont été construites. Elles font usage des fonctionnalités proposées par Play, par exemple des helpers qui facilitent la construction de divers objets Bootstrap ou l'injection de code Scala dans les fichiers HTML.

### 6.6.1 Affichage des flux

L'affichage des Events reçus par les écrans se fait à l'aide d'une page HTML simple, qui définit les balises nécessaires à l'affichage des différents types de flux. A part un peu de CSS, rien d'autre ne s'y passe. En voici le body :

```
1 <body>
2   <div class="frame">
3     <iframe id="frame0" class="frame" frameborder="0"></iframe>
4   </div>
5
6   <div>
7     <img id="image0">
8   </div>
9
10  <div id="footer0" class="footer"></div>
11</body>
```

Listing 20 – Eventsource HTML

Il est couplé d'un script Javascript qui crée la connexion SSE avec le serveur et qui traite les Events qu'il reçoit par la suite. Selon le type de flux, il appelle des fonctions d'affichage différentes, qui agissent avec JQuery sur les balises présentées précédemment pour en modifier le contenu ou les cacher. Ci-dessous est présenté la fonction principale de ce script :

```
1 $(document).ready(function () {
2   var evtSource = new EventSource("http://localhost:9000/eventsource", {withCredentials:
3     true});
4   var macAddress = getCookie("mac");
5   var resolution = getCookie("resolution");
6
7   evtSource.addEventListener('message', function(e) {
8     var type = e.data.substr(0, e.data.indexOf("?"));
9     var data = e.data.substr(e.data.indexOf("?") + 1, e.data.indexOf("|") - type.length
10     - 1);
11     var macs = e.data.substr(e.data.indexOf("|") + 1).split(",");
12
13     if (macs.includes(macAddress)) {
14       if (type === "url") {
15         displayFlux(data);
16       }
17       else if (type === "image") {
18         if (resolution === "1080") {
19           displayImage(data, 1900, 1080);
20         }
21         else if (resolution === "720") {
22           displayImage(data, 1280, 720);
23         }
24       }
25       else if (type === "text") {
26         displayFooterText(data);
27       }
28       else if (type === "video") {
29         displayVideo(data);
30       }
31     }
32  });
33});
```

Listing 21 – Eventsource JS

On peut observer dans la déclaration de la fonction de Listening de notre Eventsource la logique de déconstruction des messages reçus, ainsi que le traitement adéquat selon le type du flux.

### 6.6.2 Echange de données

Pour échanger des informations entre le serveur et le client, j'ai choisi d'utiliser les fonctionnalités offertes par Play, donc avoir des entités représentant mes données (par exemple un Flux devient un FluxData) et de les utiliser avec les templates Scala, Play permettant de facilement transférer des objets Java des contrôleurs aux vues. Ces templates sont des blocs de texte contenant du code Scala qui est par la suite compilé en HTML. Il devient alors facile de combiner cela à Bootstrap pour l'affichage ou l'envoi de données depuis le client.

Exemple d'entité :

```
1 public class UserData {
2
3     private String email;
4
5     private String password;
6
7     public UserData() {
8     }
9     // getters and setters...
10 }
```

Listing 22 – UserData.java

#### 6.6.2.1 Serveur -> client

```
1 // Serveur
2 public Result index() {
3     return ok(team_page.render(dataUtils.getAllTeams(), null));
4 }
5
6 // Client: team_page.scala.html
7 @(teams: util.List[TeamData], error: String)
8
9 @if(teams != null) {
10     @for(team <- teams) {
11         <tr>
12             <td>@team.getName()</td>
13             <td>@if(team.getMembers() != null) {
14                 team.getMembers().length
15             }</td>
16         </tr>
17     }
18 }
```

Listing 23 – Exemple échange serveur-client

Ici, *datautils.getAllTeams()* renvoie une liste de **TeamData** qui sont récupérés et utilisés ensuite par le fichier HTML. On peut observer ici l'intégration d'une boucle for Scala avec une Table Bootstrap.

### 6.6.2.2 Client -> serveur

```
1 // Client
2 @helper.form(routes.UserController.register()) {
3     @helper.CSRF.formField
4     <div class="form-group">
5         <input name="email" type="email" class="form-control">
6     </div>
7     <div class="form-group">
8         <input name="password" type="password" class="form-control">
9     </div>
10    <button type="submit" class="btn btn-primary">Register</button>
11 }
12
13
14 // Serveur: UserController.java
15 public Result register(Http.Request request) {
16     final Form<UserData> boundForm = form.bindFromRequest(request);
17     UserData data = boundForm.get();
18     User newUser = new User(data.getEmail(), data.getPassword());
19
20     // email is not unique
21     if (userRepository.getByEmail(newUser.getEmail()) != null) {
22         return registerViewWithErrorMessage("Email is already used");
23     }
24     else
25         ...
26 }
```

Listing 24 – Exemple échange client-serveur

Dans cet extrait de code, on a un exemple des fonctionnalités offertes par Play sous la forme de helpers servant à faciliter la création de formulaires Bootstrap. L'action effectuée par le bouton du formulaire est directement liée à la méthode de `UserController.java`. Il faut choisir comme valeur pour l'attribut `name` des balises `<input>` les noms des attributs correspondants dans le modèle associé (`UserData` en l'occurrence).

Le serveur est par la suite capable de reconstruire un objet du même type en récupérant le formulaire depuis la requête.

### 6.6.2.3 Erreurs

Parmi les exigences du projet figurait la nécessité d'empêcher l'utilisateur de faire des fausses manipulations sur les données et surtout de l'informer en cas d'erreur de l'application avec des messages utiles. Il a donc fallu trouver un système permettant d'afficher facilement un message dans les vues de l'application en cas d'erreur.

Ci-dessous, un extrait de trois fichiers explicitant le traitement des erreurs dans l'application :

```
1 // UserController.java
2 public Result index() {
3     return ok(user_page.render(servicePicker.getUserService().getAllUsers(),
4         null));
5 }
6 public Result indexWithErrorMessage(String error) {
7     return badRequest(user_page.render(servicePicker.getUserService().getAllUsers(),
8         error));
9 }
10
11 // user_page.scala.html
12 @(users: util.List[UserData],
13     error: String)
14 @main("Users Main Page", error) {
15     <div>
16         ...
17     </div>
18 }
```

```

17     </div>
18 }
19
20 // main.scala.html
21 ...
22 @if(error != null) {
23     <div class="alert">
24         <span class="closebtn" onclick="this.parentElement.style.display='none';">&times;</
25         span>
26         <strong>@error</strong>
27     </div>
28 }
29 ...

```

Listing 25 – Exemple des messages d'erreur

Les contrôleurs ont à chaque fois deux méthodes pour retourner une vue, une sans message d'erreur et une avec (lignes 2 à 9). Même si certaines de ces méthodes sont identiques, il peut être utile de séparer la logique d'affichage en cas d'erreur pour potentiellement effectuer des opérations supplémentaires ou envoyer des données différentes.

Ce message d'erreur est récupéré par la vue correspondante, dans notre cas : *user\_page.scala.html*, qui l'envoie ensuite au template main. Ce template est appelé avant chaque vue et définit entre autre des headers, styles, etc. Il contient également le troisième extrait de code du Listing précédent. Ce code affiche une alerte avec comme message l'erreur envoyée par le serveur. Cette implémentation permet de simplifier le traitement des erreurs en regroupant toute la logique frontend au même endroit et en offrant des méthodes simples d'utilisation pour le serveur.

## 7 Interface

Dans cette section sera présentée et expliquée la version finale de l'interface utilisateur. La navigation dans les différentes pages de l'application est organisée avec une Navbar Bootstrap. Chacune des sections suivantes décrit une de ces pages.

### 7.1 Home

Cette page sert de page d'accueil dans le programme et affiche les écrans, Schedules et Diffusers actifs dans des Tables Bootstrap. Elle permet également d'arrêter la diffusion en cours sur un écran en le désactivant à l'aide d'un bouton.

Home

Teams

Users

Fluxes

Screens

Schedules

Diffusers

Active screens

Name	Site	Current flux	
1234	1	flux2	<div>Deactivate Screen</div>

Active Schedules

Name	Activated
schedule1	true

Active Diffusers

Name	Flux	Hour	Activated
diffuser1		14:06	true

FIGURE 7 – Homepage

Cette technique d'affichage des données avec des Tables est celle utilisée pour toutes les pages principale de l'application, soit celles accessibles par la Navbar.

### 7.2 Teams

Cette page est uniquement accessible par un admin car depuis elle, on a accès à toutes les teams existantes, avec la possibilité de les mettre à jour ou des les supprimer. On peut également créer une nouvelle team et cette page est un peu particulière car toutes les données y sont accessibles. Un admin peut en effet vouloir directement spécifier des Flux ou utilisateurs ou autre entité lors de la création, donc les données envoyées ne sont pas limitées par la Team de l'utilisateur courant (c'est logique, seul l'admin y a accès).

<div>Home</div> <div>Teams</div> <div>Users</div> <div>Fluxes</div> <div>Screens</div> <div>Schedules</div> <div>Diffusers</div>						
Name		Members				
team1		<div>Delete Team</div>				
<div>Create new Team</div>						

FIGURE 8 – Teampage

## 7.3 Users

Cette page est accessible par les admins et les chefs d'équipe. Elle référence les utilisateurs inscrits dans le système et permet leur mise à jour ou suppression. Elle offre aussi un lien vers la page d'ajout d'utilisateur, qui est présentée ci-dessous :

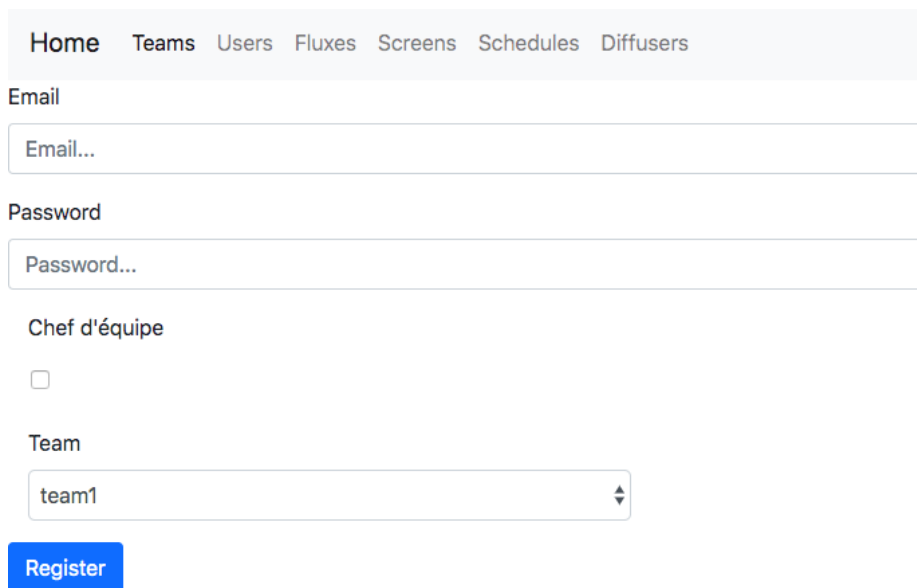


FIGURE 9 – Ajout d'utilisateur

Il n'y a pas besoin d'être authentifié pour accéder à cette page car elle est utilisée pour se créer un compte. On peut préciser la team dont il fait partie et s'il s'agit d'un chef d'équipe.

## 7.4 Fluxes

De la même manière que les autres, cette page présente les flux accessibles par l'utilisateur courant et permet de les modifier, supprimer ou d'en créer des nouveaux. C'est l'interface de création qui sera présentée dans cette section :

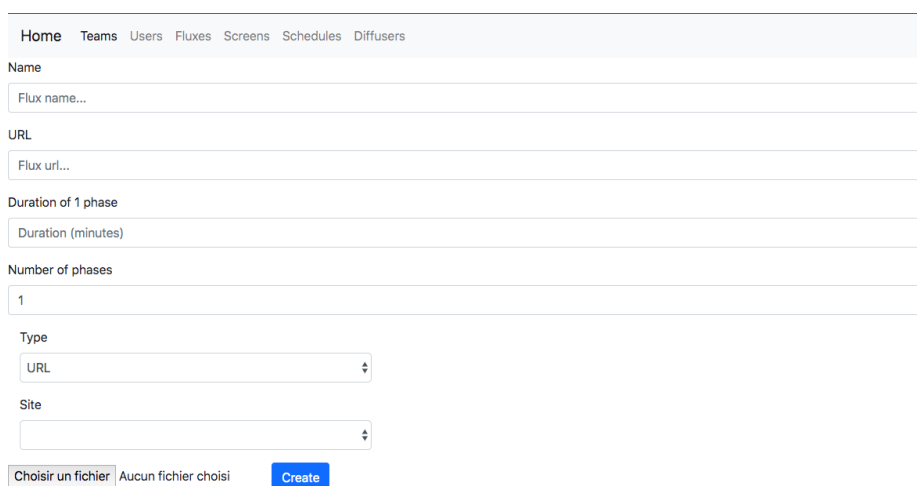


FIGURE 10 – Création de Flux

On y voit les différents moyens de création de flux. J'aurais voulu changer le formulaire affiché selon le type de flux créé (URL, VIDEO, etc), car pour l'instant, tous les champs sont accessibles tout le temps. Voici la procédure minimale à suivre par type :

- **URL** : entrer l'URL de la page à afficher et choisir une durée.
- **IMAGE** : choisir une durée et uploader une image.
- **VIDEO** : entrer l'URL de la vidéo Youtube à afficher et choisir une durée.



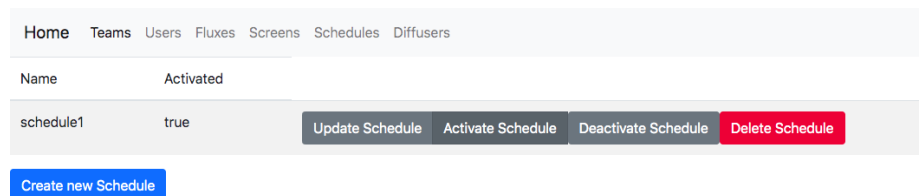
— **TEXT** : entrer le texte à afficher dans le champs URL et choisir une durée  
Pour chacun de ces choix, il est également possible de choisir un Site et ainsi créer un flux localisé.

## 7.5 Screens

Cette page présente les différents écrans accessibles par la Team de l'utilisateur courant et offre les mêmes fonctionnalités que les autres pages : mise à jour, suppression, ajout d'écran au système. La seule particularité de cette section est d'exiger un code d'authentification pour l'ajout d'écran.

## 7.6 Schedules

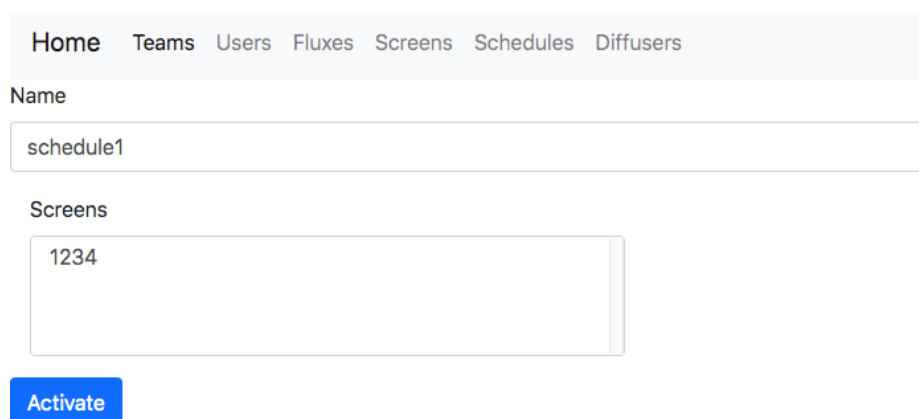
Cette page est un peu spéciale car en plus des habituelles opérations CRUD proposées, elle offre aussi la possibilité d'activer et de désactiver les Schedules, comme on peut le voir sur la figure suivante :



Home Teams Users Fluxes Screens Schedules Diffusers					
Name	Activated				
schedule1	true	Update Schedule	Activate Schedule	Deactivate Schedule	Delete Schedule
Create new Schedule					

FIGURE 11 – Schedules page

Ce bouton nous redirige vers une autre vue, qui elle permet de choisir les écrans qui seront concernés par ce Schedule. Les écrans affichés dans la liste sont les écrans accessibles par la Team de l'utilisateur courant.



Home Teams Users Fluxes Screens Schedules Diffusers

Name

schedule1

Screens

1234

Activate

FIGURE 12 – Schedules page

La page la plus compliquée que j'ai eu à réaliser fût celle permettant la création de Schedules. Il fallait en effet trouver un moyen de choisir les flux composant le Schedule et pour chacun de ces flux avoir la possibilité de définir une heure de début. Ayant assez peu d'expérience en *frontend* de manière générale, j'ai beaucoup cherché d'idées en ligne et ai fini par trouver un exemple que j'ai pu adapter à mon problème. En associant une Table Bootstrap avec du Javascript, je suis arrivé au résultat présenté dans la figure suivante :

FIGURE 13 – Schedules page

On peut observer deux parties à cette page, une pour l'ajout de flux (avec heure de début ou non) ainsi qu'une pour le choix des flux de fallback du Schedule. En cliquant sur le bouton *Add new*, une nouvelle ligne apparaît dans le tableau. Dans la colonne *Name*, on peut choisir parmi les flux disponibles pour la Team courante et dans la colonne *Time* on peut rentrer une heure sous ce format : hh :mm. Si aucune heure n'est rentrée, cela signifie que le flux est sans heure de début. Dans le deuxième tableau, on peut choisir quels seront (s'il y en a) les flux de fallback du Schedule. L'option *Keep order* permet de choisir si l'ordre des flux sans heure de début doit être aléatoire ou toujours le même (donc équivalent à l'ordre dans lequel ils ont été ajoutés au Schedule).

## 7.7 Diffusers

Cette page est très similaire à celle des Schedules, car elle permet d'effectuer les mêmes actions (un Diffuser pouvant également être activé/désactivé). La seule différence se trouve dans la page de création qui est bien plus simple que celle des Schedules car très standard.

## 8 Tests

Le framework Play fourni plusieurs moyens d'implémentation de tests, pour plusieurs sortes de tests différents : unitaires, fonctionnels, tests de base de données, etc. Leur implémentation est décrite dans cette section.

### 8.1 Tests unitaire

Les tests unitaires ont été réalisé comme conseillé par la documentation de Play, c'est-à-dire en utilisant des Mocks pour isoler les tests des dépendances externes (dans mon cas les JPARepositories contenant la logique d'accès à la base de données). Dans le chapitre précédent (section DAO), il a été mentionné que les services existaient en partie pour faciliter la mise en place des tests par la suite. Ces services n'étaient pas encore créés lorsque j'ai commencé à rédiger les tests et j'ai donc du refactor une bonne partie du programme pour ne pas utiliser directement mes JPARepositories dans le code. Ces services étant validés par mes tests, je m'assure ainsi d'avoir le comportement espéré.

Le principe est le suivant :

Pour chaque fonction de test, je mock le comportement attendu d'une ou plusieurs fonctions du dépôt voulu, puis crée le service correspondant en le passant à son constructeur. Lorsque la fonction mockée de ce dépôt sera utilisé par le service, elle aura le comportement choisi. Les valeurs retournées sont ensuite vérifiées avec les *Assert* de JUnit. De cette façon, on peut vérifier le bon fonctionnement de ces fonctions dans toutes les conditions (cas standard, erreur, mauvais paramètre d'entrée, etc).

Dans la figure suivante est présenté un exemple de ces tests :

```
1 @Mock
2 private FluxRepository mockFluxRepository;
3 private FLux fluxToReturn;
4 ...
5
6 @Before
7 public void setUp() {
8     MockitoAnnotations.initMocks(this);
9     ...
10 }
11
12 @Test
13 public void testGetFluxByName() {
14     when(mockFluxRepository.getByNome("fluxName")).thenReturn(fluxToReturn);
15     FluxService service = new FluxService(mockFluxRepository);
16
17     assertEquals(fluxName, service.getFluxByName("fluxName").getName());
18 }
```

Listing 26 – Exemple de test unitaire - FluxUnitTest.java

On peut y voir la même structure que toutes les classes de tests possèdent, à savoir un dépôt mocké, une fonction *setUp()* initialisant les différents paramètres de la classe et les fonctions de test.

### 8.2 Tests fonctionnels

Il était prévu de fournir des tests fonctionnels pour au moins valider les scénarios d'utilisation basiques tels que la création puis activation d'un Schedule. Mais lorsque j'ai dû les implémenter, je n'ai pas réussi à les faire fonctionner. Mon erreur a été de les commencer trop tard. J'avais déjà des tests unitaires et après avoir regardé la documentation et quelques exemples, je pensais réussir à les faire fonctionner rapidement mais cela n'a pas été le cas.

Il n'y a donc aucun tests en plus des tests unitaires.

## **9 Commentaires et conclusion**

# A Cahier des charges

Ce cahier des charges à du être mis à jour suite aux remarques et demandes de mon mentor.

## A.1 Contraintes et besoins

Les besoins principaux de cette application sont les suivants :

- Gestion de l’affichage de flux d’information sur des écrans dans la HEIG-VD (smartTV ou ordinateur) par une interface web.
- Protocole concis de communication entre les écrans et le serveur limitant les échanges.
- Affichage de flux "controlés" (générés par l’application, par exemple un flux RSS) et "non-controlés" (flux de la RTS, horaires des cours, etc).
- Un Schedule qui s’occupe de changer les flux affichés selon un horaire prédéfini.
- Une modélisation générique des flux et une factorisation de ceux venant de sources externes afin de pouvoir les envoyer de la même manière aux écrans
- La possibilité de diffuser plusieurs types de médias (images, vidéos, etc)
- La possibilité de passer outre le Schedule et d’afficher un flux voulu (annonce importante, etc) avec reprise de l’exécution prévue par la suite.

Les contraintes principales quand à elles sont les suivantes :

- L’affichage des flux doit être fait dans un navigateur supportant le Javascript, HTML5 et CSS3.
- Il doit y avoir une base de données qui enregistre les utilisateurs ainsi que les écrans.
- Il y a plusieurs types d’utilisateurs qui, selon leur emplacement (campus) et/ou leur niveau d’autorisation, peuvent modifier l’affichage des écrans.
- Le système doit être tolérant face aux panne, avec une reprise automatique.
- Le système doit disposer d’une interface simple et être utilisable par des gens du domaine et par des personnes non-initiées.

## A.2 Fonctionnalités

Les fonctionnalités nécessaires et principales du programme sont divisées en plusieurs catégories :

### — Frontend

1. Interface de login et register sur le site (register dans le cadre du TB)
2. Ecrans
  - (a) Visualisation des écrans actifs
  - (b) Visualisation des informations d’un écran spécifique
  - (c) Modification de la diffusion actuelle sur un écran/groupe d’écrans
3. Flux
  - (a) Opérations CRUD sur les flux (interface de création)
  - (b) Visualisation des flux utilisables par le système et infos sur leur contenu
4. Schedules
  - (a) Opérations CRUD sur les Schedules (interface de création)
  - (b) Visualisation des Schedules utilisables par le système et infos sur leur contenu et horaire
5. Diffusers
  - (a) Opérations CRUD sur les Diffusers (interface de création)
  - (b) Visualisation des Diffusers utilisables par le système et infos sur leur contenu et horaire
6. Affichage des flux
  - (a) Solution pour l’affichage de flux dans des *iframes*
  - (b) Solution pour l’affichage d’autres types de flux (vidéo, image, texte)

### — Backend - Play

1. Ecrans
  - (a) Opérations CRUD sur les écrans
  - (b) Solution d’authentification des écrans auprès du serveur
  - (c) Solution de pilotage des écrans (arrêt de l’affichage et autres)

2. Flux
  - (a) Opérations CRUD sur les flux
  - (b) Diffusion de flux aux écrans selon un Schedule
  - (c) Diffusion de flux hors-Schedule (annonces, alertes, etc)
  - (d) Formatage et mise en page des flux externes (RTS ou autre)
3. Schedules
  - (a) Opérations CRUD sur les Schedules
  - (b) Assignment d'un Schedule à un écran/groupe d'écrans
  - (c) Activation/désactivation d'un Schedule
4. Utilisateurs
  - (a) Register
  - (b) Login
  - (c) Niveaux d'autorisation

— **Base de données**

1. Utilisateurs avec différents niveaux d'autorisations
2. Ecrans, avec leurs caractéristiques et emplacement
3. Flux utilisés par le système
4. Diffusers
5. Schedules de flux

Les fonctionnalités suivantes sont considérées comme secondaires et seront réalisées si le temps le permet :

— **Frontend**

1. Modification en live du contenu d'un flux

— **Backend**

1. Monitoring de l'état des écrans
2. Plusieurs Schedules par écran

### A.3 Echancier

Le travail sera divisé en 4 parties :

— **Analyse et Modélisation** (2 semaines)

Analyse des contraintes et besoins du travail et modélisation d'un système parvenant à y répondre. Recherches sur les différentes solutions possibles pour répondre aux exigences du projet. C'est pendant cette phase que les contours principaux du programme seront définis.

— **Architecture** (1 semaine)

Création d'une architecture de code permettant la réalisation du programme et mise au point des différents algorithmes nécessaires. Par exemple l'algorithme de scheduling des flux ou le protocole de connexion des écrans au serveur.

— **Développement et Tests** (7 semaines)

Codage de l'application, en commençant par le serveur et en finissant avec le frontend. Création de la base de données en parallèle. Tests unitaires et fonctionnels

— **Rapport et Documentation** (3 semaines)

Total : env. 13 semaines à partir du 25.02

## B Journal de travail

Mon journal de travail est divisé en semaines.

- **25.02 - 01.03** : première rédaction du cahier des charges puis prise en compte des remarques et commentaires pour un deuxième jet.
- **04.03 - 08.03** : premier jet du protocole de communication écrans-serveur, du schéma de base de données et de l'architecture du projet (Flux, Schedule, ...).
- **11.03 - 15.03** : recherches sur Play, les EventSources, HTML, Bootstrap pour bien comprendre leur fonctionnement et chercher des moyens de résoudre les différents problèmes.
- **18.03 - 22.03** : début du codage de l'application et mise en place des éléments essentiels au programme, à savoir implémentation de JPA et première implémentation (en Java) du contrôleur des EventSources.
- **25.03 - 29.03** : travail sur l'authentification des écrans auprès du serveur et premier jet du système d'authentification des utilisateurs en utilisant JWT.
- **01.04 - 05.04** : rédaction du rapport intermédiaire et définition plus claire de mes objectifs pour ce programme.
- **08.04 - 12.04** : pas de travail effectué lors de cette semaine à cause du Baleinev Festival.
- **15.04 - 19.04** : création des modèles, dépôts et contrôleurs nécessaire au programme et première utilisation du patron de conception Observer pour la gestion des flux. Refactor de la manière d'envoyer les Events aux écrans (EventSourceController.java). Arrêt de l'utilisation de JWT pour identifier les utilisateurs pour la remplacer par des simples cookies.
- **22.04 - 26.04** : premier jet pour les threads associés aux Schedules actifs. Création des multiples vues de l'application et prise en charge des messages d'erreur. Implémentation de l'activation des Schedules.
- **29.04 - 03.05** : raffinement des vues Bootstrap et ajout de nouvelles vues. Rajout des fonctionnalités CRUD pour les contrôleurs. Implémentation de la désactivation des Schedules. Passage du contrôleur des EventSources de Java à Scala pour cause de bug dans la version Java de Play. Premier ajout des vues et du contrôleur des Diffusers. Début de la logique concernant l'activation des Diffusers.
- **06.05 - 10.05** : rédaction presque finale du script de base de données. Solution pour la persistance de liste d'objets en base de données avec JPA. Mise au point du système de bloc-horaire et refactor général du code pour l'utiliser. Amélioration de la gestion d'erreurs. Amélioration du ThreadManager. Implémentation de la restriction d'accès selon l'utilisateur courant. Améliorations diverses.
- **13.05 - 17.05** : Implémentation l'interface de création des Schedules. Nettoyage et refactor du code. Création des triggers SQL et correction et amélioration du script de base de données. Interface des Team nettoyée et débuggée pour servir d'exemple pour la suite. Implémentation d'un système de reprise automatique de l'exécution des Schedules après un redémarrage du serveur.
- **20.05 - 24.05** : fin de la rédaction des triggers. Implémentation de méthodes d'affichage des flux selon leur type (vidéo, image, texte). Création des services et gros refactor du code pour les utiliser. Premier jet des tests unitaires. Ajout d'un moyen d'uploader des images sur le serveur pour la création de flux. Ajout des pages d'erreur et de maintenance. Reprise de la rédaction du rapport.
- **27.05 - 03.06** : Rédaction du rapport et quelques changements opérés dans le code suite à la découverte de bugs ou de problèmes.

## C Cas d'utilisation

Avertissement : cette section a servi à délimiter les contours du programme pendant la phase d'analyse. Elle n'a pas été mise à jour depuis le rendu du rapport intermédiaire.

Les cas d'utilisation suivants sont regroupés par catégorie d'utilisateurs. Il y en a trois : les administrateurs, les chefs d'équipe (TeamAdmin) et les simples membres d'une équipe (TeamMember). Les admins ne sont associés à aucun écrans tandis que les deux autres sont restreints à certains écrans. Toute action possible pour une catégorie l'est également pour celles en dessus.

### C.1 Administrateur :

L'admins peut effectuer toutes les actions et est le seul à pouvoir ajouter ou supprimer des écrans ou des utilisateurs au système. L'écran est allumé et connecté dans tous les cas suivants.

— **Scénario 1 : Ajout d'un écran**

**Déroulement :** L'admins rentre l'URL d'authentification des écrans dans le navigateur en spécifiant l'adresse MAC de l'écran comme paramètre de requête. Le serveur ne reconnaît pas l'adresse MAC envoyée et renvoie donc un code servant à enregistrer l'écran dans le système. L'admins passe donc par le site pour ajouter l'écran en spécifiant entre autres son adresse MAC, son emplacement et le code fourni précédemment.

**Résultat :** L'écran sera maintenant reconnu par le serveur et correctement redirigé à la prochaine tentative.

**Erreurs potentielles :** Si la connexion est perdue entre l'écran et le backend à n'importe quel moment du scénario, les mêmes opérations seront effectuées à la re-connexion de l'écran (envoi du code).

— **Scénario 2 : Mise à jour des infos d'un écran**

**Pré-requis :** l'écran est déjà connu par le système.

**Déroulement :** L'admins se connecte au site et utilise l'interface fournie pour mettre à jour les infos souhaitées (nécessite potentiellement que l'écran ne soit pas actif).

**Résultat :** L'admins est informé du succès (ou de l'échec) de l'opération.

— **Scénario 3 : Suppression d'un écran**

**Pré-requis :** l'écran est déjà connu par le système.

**Déroulement :** L'admins se connecte au site et supprime l'écran du système en utilisant l'interface.

**Résultat :** L'admins est informé du succès (ou de l'échec) de l'opération et l'adresse MAC de l'écran est supprimée du système.

— **Scénario 4 : Ajout d'un utilisateur**

**Déroulement :** L'admins se connecte au site et ajoute l'utilisateur en utilisant l'interface fournie. Lors de l'ajout, il spécifie les écrans auxquels l'utilisateur pourra assigner des Schedules.

**Résultat :** L'admins est informé du succès (ou de l'échec) de l'opération et l'utilisateur est ajouté à la base de donnée.

— **Scénario 5 : Modération : désactivation de Schedule**

**Pré-requis :** Le Schedule est activé.

**Déroulement :** L'admins se connecte au site et va sur la page des Schedules. Dans la liste des actifs, il sélectionne celui qu'il veut désactiver et confirme son choix.

**Résultat :** L'admins est informé du succès (ou de l'échec) de l'opération et le Schedule est désactivé.

— **Scénario 6 : Modération : désactivation de Diffuser**

**Pré-requis :** Le Diffuser est activé.

**Déroulement :** L'admins se connecte au site et va sur la page des Diffusers. Dans la liste des actifs, il sélectionne celui qu'il veut désactiver et confirme son choix.

**Résultat :** L'admins est informé du succès (ou de l'échec) de l'opération, le Diffuser est désactivé et son flux est retiré des Schedules correspondants.



— **Scénario 7 : Création d'une Team**

**Déroulement :** L'admins se connecte au site et va sur la page des Teams. Il utilise l'interface fournie pour créer une nouvelle Team. Il doit spécifier à la création le nom de la Team et les écrans accessibles par ses membres.

**Résultat :** L'admins est informé du succès (ou de l'échec) de l'opération et la Team est créée et ajoutée en BD.

**Erreurs potentielles :**

- Si le nom choisi pour la Team existe déjà, une erreur sera lancée et l'admins devra en choisir un autre.

— **Scénario 8 : Modification d'une Team**

**Pré-requis :** La Team existe.

**Déroulement :** L'admins se connecte au site et va sur la page des Teams. Il utilise la même interface que pour la création pour mettre à jour les infos souhaitées (nom, membres, admins).

**Résultat :** L'admins est informé du succès (ou de l'échec) de l'opération et la Team est modifiée.

**Erreurs potentielles :**

- Si le nom choisi pour la Team existe déjà, une erreur sera lancée et l'admins devra en choisir un autre.

— **Scénario 9 : Suppression d'une Team**

**Pré-requis :** La Team existe.

**Déroulement :** L'admins se connecte au site et va sur la page des Teams. Il sélectionne dans la liste celle qu'il souhaite supprimer et utilise l'interface fournie pour le faire.

**Résultat :** L'admins est informé du succès (ou de l'échec) de l'opération et la Team est supprimée. Les entités associés avec cette équipe sont également supprimées (Schedules et Diffuser).

## C.2 TeamAdmin :

Un TeamAdmin ne peut ajouter d'écrans mais a la permission d'activer Schedules et Diffusers.

— **Scénario 1 :** Activation d'un Schedule

**Pré-requis :** Le Schedule existe et les écrans choisis ne sont pas déjà assignés à un autre Schedule.

**Déroulement :** Le TeamAdmin se connecte au site et va sur la page des Schedules. Il choisit dans la liste affichée celui qu'il veut activer et utilise l'interface pour assigner des écrans ou groupes d'écrans à ce Schedule. Il peut ensuite activer son Schedule.

**Résultat :** Le TeamAdmin est informé du succès (ou de l'échec) de l'opération.

**Erreurs potentielles :**

- Si le Schedule contient un flux restreint à un site et que l'on l'assigne à un écran sur un autre site, le système nous empêchera de le faire. Par contre, assigner un groupe d'écran avec un sous-ensemble de ce groupe d'un site différent sera possible (un flux de backup sera diffusé sur cet écran à la place).
- Si, parmi les écrans choisis, un ou plusieurs sont déjà assignés à un Schedule, le TeamAdmin en est prévenu et doit changer sa sélection.

— **Scénario 2 :** Activation d'un Diffuser

**Pré-requis :** Le Diffuser existe et les écrans choisis sont assignés à un Schedule.

**Déroulement :** Le TeamAdmin se connecte au site et va sur la page des Diffusers. Il choisit dans la liste affichée celui qu'il veut activer et utilise l'interface pour assigner des écrans ou groupes d'écrans à ce Diffuser. Il peut ensuite l'activer.

**Résultat :** Le TeamAdmin est informé du succès (ou de l'échec) de l'opération.

**Erreurs potentielles :**

- L'heure de début prévue pour le flux du Diffuser est identique (ou à peine après) à l'heure de début d'un flux du Schedule. Il y a plusieurs manières de traiter ce cas : checker la durée du nouveau flux et reprendre l'exécution de l'ancien une fois fini, repousser un des deux flux (pas top je pense).

— **Scénario 3 :** Création de groupe d'écrans

**Déroulement :** Le TeamAdmin se connecte au site et va sur la page des écrans. Il choisit les écrans (au moins 2) dans la liste pour son groupe et confirme son choix. (Les écrans peuvent appartenir à plusieurs groupes)

**Résultat :** Le TeamAdmin est informé du succès (ou de l'échec) de l'opération et le groupe est créé en BD.

**Erreurs potentielles :**

- Moins de 2 écrans sont choisis, le TeamAdmin est informé et doit choisir plus d'écrans.

— **Scénario 4 :** Modification de groupe d'écrans

**Pré-requis :** Le groupe existe.

**Déroulement :** Le TeamAdmin se connecte au site et va sur la page des écrans/groupes. Il choisit dans la liste des groupes celui ou ceux qu'il désire modifier et utilise pour ce faire la même interface que pour la création de groupe.

**Résultat :** Le TeamAdmin est informé du succès (ou de l'échec) de l'opération et le groupe est modifié en BD.

**Erreurs potentielles :**

- Si le groupe est actuellement assigné à un RunningSchedule, la modification est empêchée et le TeamAdmin en est informé.
- Si le groupe modifié contient moins de deux écrans, la modification est empêchée et le TeamAdmin en est informé.

— **Scénario 5 :** Suppression de groupe d'écrans

**Pré-requis :** Le groupe existe.

**Déroulement :** Le TeamAdmin se connecte au site et va sur la page des écrans/groupes. Il choisit dans la liste des groupes celui ou ceux qu'il désire supprimer et utilise l'interface fournie pour le faire.

**Résultat :** Le TeamAdmin est informé du succès (ou de l'échec) de l'opération et le groupe est supprimé en BD.

**Erreurs potentielles :**

- Si le groupe est actuellement assigné à un RunningSchedule, la suppression est empêchée et le TeamAdmin en est informé.

**C.3 TeamMember :**

Un TeamMember est assigné à une Team, qui elle à accès à des écrans, Schedules et Diffusers. Il peut créer et modifier des Schedules et Diffuser non-actifs mais ne peut pas les activer. Comme tous les autres types d'utilisateurs, il peut créer des flux.

- **Scénario 1 :** Création d'un flux

**Déroulement :** Le TeamMember se connecte au site et va sur la page des flux. Il entre les paramètres de son flux (à définir)

**Résultat :** Le TeamMember est informé du succès (ou de l'échec) de l'opération et le flux est ajouté à la liste des flux disponibles.

- **Scénario 2 :** Modification d'un flux

**Pré-requis :** Le flux existe.

**Déroulement :** Le TeamMember se connecte au site et va sur la page des flux. Il choisit le flux à modifier dans la liste et utilise la même interface que pour la création pour la mise à jour.

**Résultat :** Le TeamMember est informé du succès (ou de l'échec) de l'opération.

- **Scénario 3 :** Suppression d'un flux

**Pré-requis :** Le flux existe.

**Déroulement :** Le TeamMember se connecte au site et va sur la page des flux. Il choisit le flux à supprimer et confirme son choix.

**Résultat :** Le TeamMember est informé du succès (ou de l'échec) de l'opération et le flux est retiré de la liste des flux disponibles.

- **Scénario 4 :** Création d'un Schedule

**Pré-requis :** Des flux ont préalablement été créés.

**Déroulement :** Le TeamMember se connecte au site et va sur la page de création de Schedules. Il choisit les heures de début des flux en associant le flux voulu. Il peut encore spécifier le nom du Schedule ou un commentaire sur son utilité. Il confirme son choix.

**Résultat :** Le TeamMember est informé du succès (ou de l'échec) de l'opération et le Schedule est ajouté à la liste des Schedules disponibles.

**Erreurs potentielles :** Les heures de début de flux ne sont pas cohérentes (confirmation alors impossible).

- **Scénario 5 :** Modification d'un Schedule

**Pré-requis :** Le Schedule existe.

**Déroulement :** Le TeamMember se connecte au site et va sur la page des Schedules. Il choisit le Schedule à modifier dans la liste et utilise la même interface que pour la création pour la mise à jour.

**Résultat :** Le TeamMember est informé du succès (ou de l'échec) de l'opération.

**Erreurs potentielles :** Les heures de début des nouveaux flux ne sont pas cohérentes (confirmation alors impossible).

- **Scénario 6 :** Création d'un Diffuser

**Pré-requis :** Des flux ont préalablement été créés.

**Déroulement :** Le TeamMember se connecte au site et va sur la page de création de Diffuser. Il choisit les heures de début du flux voulu et précise sa durée de validité (en jours?, semaines?). Il peut encore spécifier le nom du Diffuser ou un commentaire sur son utilité. Il confirme son choix.

**Résultat :** Le TeamMember est informé du succès (ou de l'échec) de l'opération et le Diffuser est ajouté à la liste des Diffusers disponibles.

**Erreurs potentielles :** Les heures de début de flux ne sont pas cohérentes (confirmation alors impossible).

— **Scénario 7 : Modification d'un Diffuser**

**Pré-requis :** Le Diffuser existe.

**Déroulement :** Le TeamMember se connecte au site et va sur la page des Diffuser. Il choisit le Diffuser à modifier dans la liste et utilise la même interface que pour la création pour la mise à jour.

**Résultat :** Le TeamMember est informé du succès (ou de l'échec) de l'opération.

**Erreurs potentielles :** Les heures de début des nouveaux flux ne sont pas cohérentes (confirmation alors impossible).

## D Mockups

Les mockups suivant ne sont pas représentatifs de l'aspect final de l'application mais plutôt des fonctionnalités offertes. Ils ont été réalisés dans le cadre du rapport intermédiaire pour donner une idée de la direction prise pour l'interface.

### D.1 Utilisateurs

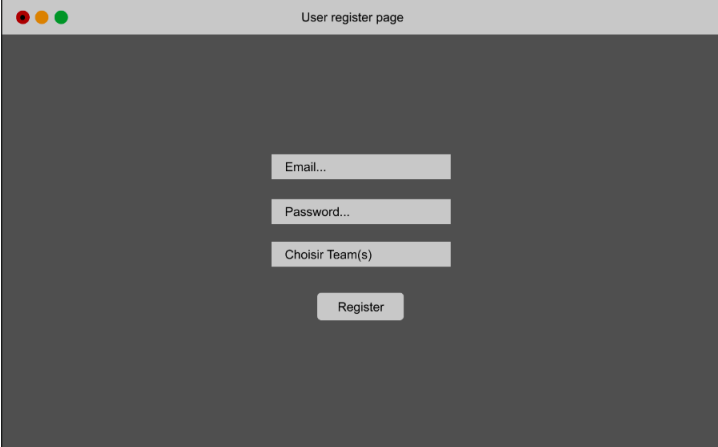
A mockup of a web browser window titled "User register page". The window has a dark gray background. In the center, there are three light gray input fields stacked vertically, labeled "Email...", "Password...", and "Choisir Team(s)". Below these fields is a light gray button labeled "Register". The browser window has a standard macOS-style title bar with red, yellow, and green window control buttons on the left.

FIGURE 14 – Interface d'ajout de nouveaux utilisateurs

Cette page en Figure 14 servira de page d'ajout d'utilisateur et sera accessible par tous (dans le cadre de mon TB en tout cas).

A mockup of a web browser window titled "User login page". The window has a dark gray background. In the center, there are two light gray input fields stacked vertically, labeled "Email..." and "Password...". Below these fields is a light gray button labeled "Login". The browser window has a standard macOS-style title bar with red, yellow, and green window control buttons on the left.

FIGURE 15 – Interface de connexion des utilisateurs

La page de login est simple et standard.

## D.2 Ecrans

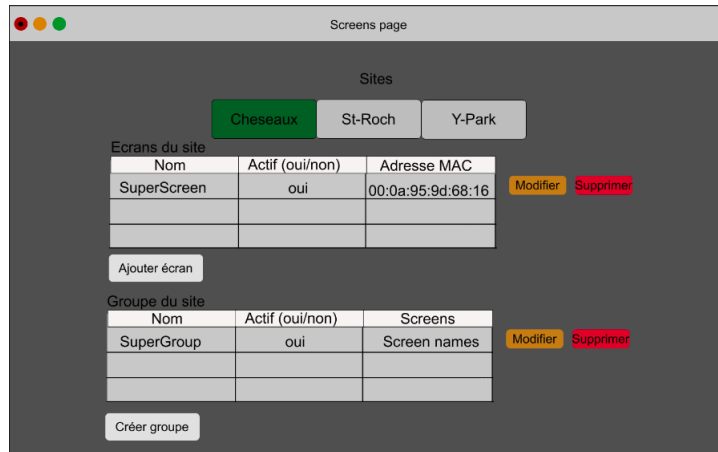


FIGURE 16 – Page principale des écrans

Cette page affichera les écrans accessibles par l'utilisateur actuel, regroupés par site. On aura également accès aux groupes d'écrans. C'est depuis cette page qu'on pourra rajouter des écrans.



FIGURE 17 – Interface d'ajout de nouvel écran

L'administrateur peut ajouter un nouvel écran au système par le biais de cette interface. Il doit préciser quelques paramètres et surtout donner le code fourni précédemment par le serveur.

## D.3 Teams

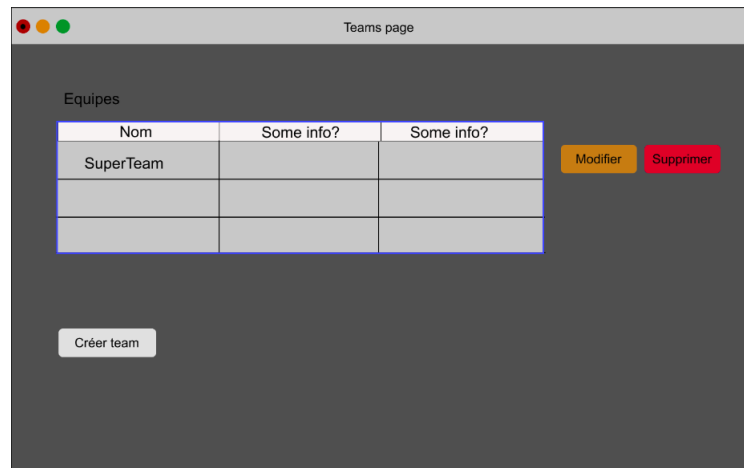


FIGURE 18 – Page principale des teams

Cette page sera accessible uniquement par les administrateurs car elle répertoriera les différentes Team de l'application avec la possibilité d'effectuer des actions dessus.

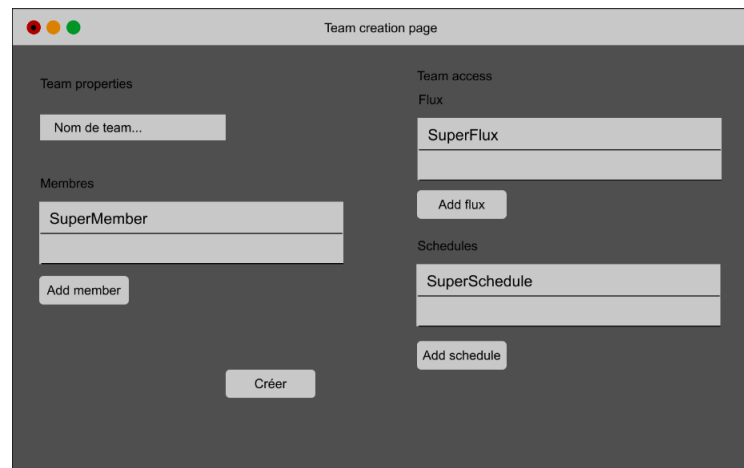


FIGURE 19 – Interface de création de team

On pourra créer des Team en spécifiant directement les objets auxquels elle a accès (Flux, Schedules, Membres, etc).

## D.4 Flux

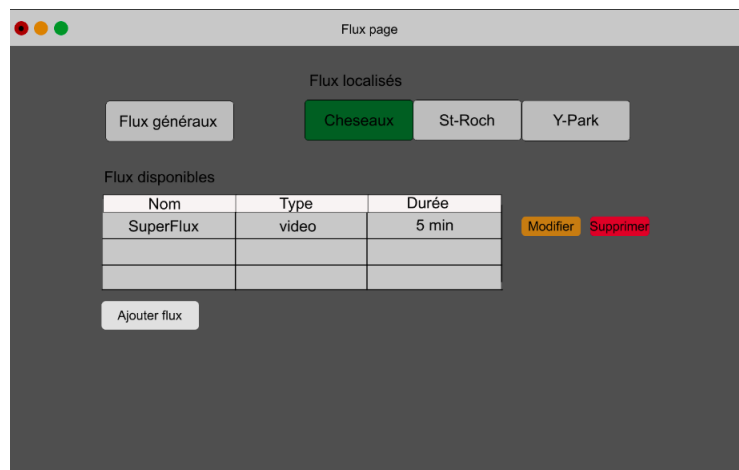


FIGURE 20 – Page principale des flux

Cette page affichera les flux disponibles pour la Team courante et permettra également des opérations sur ces flux. Idéalement, ils seraient regroupés par site.

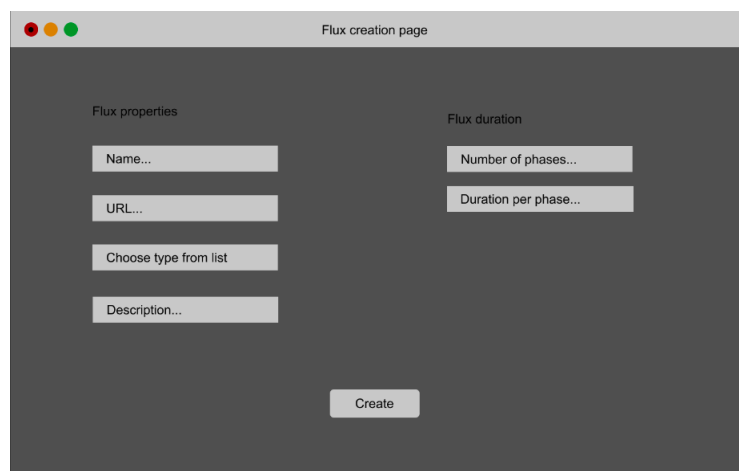


FIGURE 21 – Interface de création de flux

C'est depuis cette page que l'on pourra créer de nouveaux flux.



## D.5 Schedules

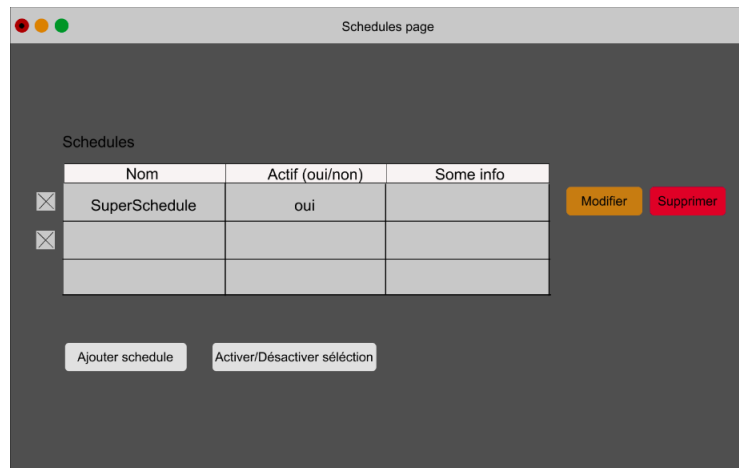


FIGURE 22 – Page principale des Schedules

Cette page affichera les Schedules de la Team courante et permettra des les activer/désactiver en plus des actions standards.

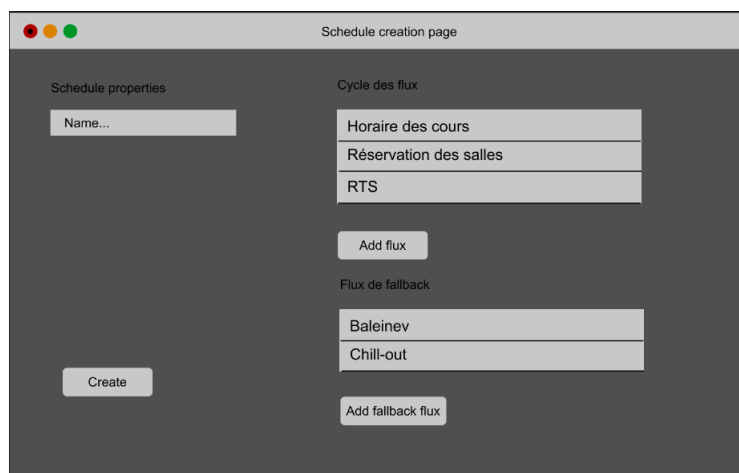


FIGURE 23 – Interface de création de Schedule

On choisit les flux qui vont constituer le cycle du Schedule. On peut également définir des flux de fallback qui prendront le relais si le flux actif n'a aucune information à afficher.

## D.6 Diffusers

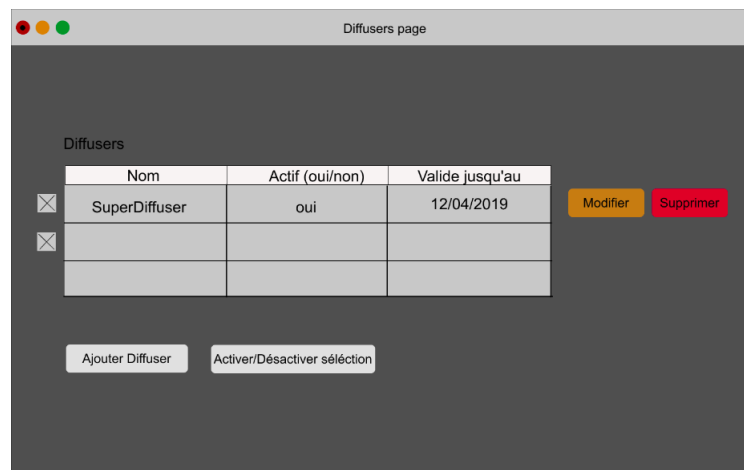


FIGURE 24 – Page principale des diffusers

Comme pour les Schedules, cette page permet d'activer/désactiver des Diffusers en plus des actions standards.

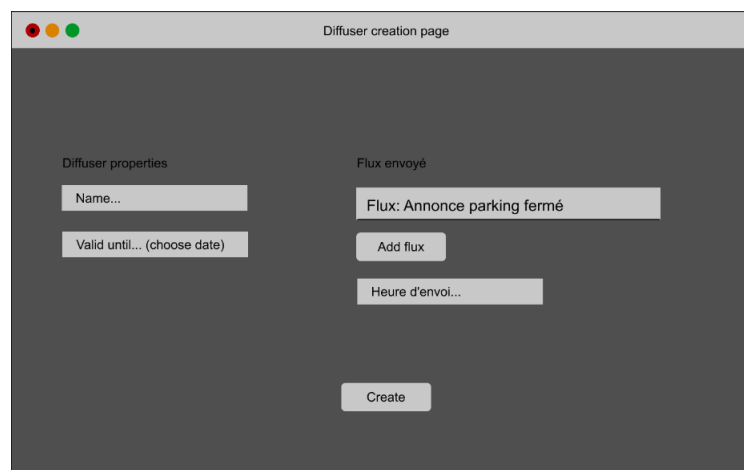


FIGURE 25 – Interface de création de diffuser

On crée un Diffuser en choisissant une heure de début, un flux et une durée de validité.

## Références

- [1] Play Framework documentation :  
<https://www.playframework.com/documentation/2.7.x/Home>
- [2] Exemple de solution SSE avec Scala Play :  
<https://github.com/septeni-original/play-scala-sse-example>
- [3] Stack Overflow, pour des réponses à mes questions :  
<https://stackoverflow.com/>
- [4] Exemple de table Bootstrap dynamique :  
<https://www.tutorialrepublic.com/snippets/preview.php?topic=bootstrap&file=table-with-add-and-d>
- [5] Documentation sur les Eventsource :  
<https://developer.mozilla.org/en-US/docs/Web/API/EventSource>  
[https://developer.mozilla.org/en-US/docs/Web/API/Server-sent\\_events/Using\\_server-sent\\_events](https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events)