Intégration de Systèmes Web et Multimédia

Mercenier Louis, Martinot Romain & Streignard Rémi

SESSION JANVIER 2022

Ce document fait office de rapport pour le cours d' *Intégration de Systèmes Web et Multimédia*, lors duquel nous, Mercenier Louis, Martinot Romain et Streignard Rémi, avons du réaliser un projet du nom de ViewCast afin de mettre en pratique la théorie vue au cours du même nom.

Table des matières

Introduction

Ce document vient en réponse à la demande formulée lors du cours d'*Intégration de Systèmes Web et Multimédia* par Madame Ramonfosse et Monsieur Marchal. L'objectif de ce projet est la réalisation d'un site web permettant l'affichage de contenu personnalisé et dynamique sur une multitude d'écrans, et ce, grâce aux technologies de notre choix.

La suite du document est divisée comme telle :

- Une partie *Cahier des charges* qui vise à synthétiser l'ensemble des informations apportées par le client afin de réaliser le projet.
- Une partie *Architecture du projet* qui reprend l'architecture et détaille les différents composants implémentés, et plus globalement les communications entre ces derniers.
- Une partie *Analyse technique* qui comprend, pour chacun des composants de la solution, les choix technologiques et techniques ainsi que les schémas d'analyse qui s'y rapportent.
- Une partie *Conclusion* qui reprend les retours formulés à l'égard du projet ainsi que les potentielles améliorations qu'il pourrait subir dans la suite de son développement.

Cahier des charges

Après la présentation du projet par le client et notre analyse de celle-ci, le cahier des charges suivant s'est dessiné :

- Créer une application web permettant d'administrer les écrans de son entreprise.
- Permettre un déploiement facile dans et pour les entreprises.
- Considérer le changement de langage au sein de l'application (FR / NL / EN).
- Rendre l'outil utilisable sur n'importe quelle télévision dite "smart", sans installation spécifique.
- Permettre une gestion horaire complète du contenu à afficher sur chaque view.
- Regrouper les télévisions en groupes qui peuvent être administrés par plusieurs utilisateurs simultanément.
- Empêcher les diffusions pirates grâce à un aspect sécurité non délaissé.
- Visualiser l'ensemble des Casts en cours au travers du client web.
- Minimiser les efforts de l'utilisateur au travers de l'interface.
- Décentraliser la solution pour la rendre propre à chacune des entreprises qui l'installe, et non pas centralisée pour l'ensemble des clients ViewCast.
- Maximiser la place disponible pour l'affichage des informations relatives à ViewCast et sa gestion.

Certaines demandes initiales du client étaient impossibles à réaliser techniquement et ont donc dû être mises de côté. C'est le cas, par exemple, de la gestion de la veille des télévisions depuis le panel d'administration ou la possibilité d'utiliser n'importe quel type d'écran - une télévision smart étant nécessaire pour pouvoir accéder à internet.

Architecture du projet

Le projet ViewCast se présente de la manière suivante :

- Un client web développé en Flutter.
- Un serveur écrit en Node JS, nommé l'*Orchestrator*, dont le rôle est la synchronisation entre l'ensemble des écrans lors des Casts.
- Une API REST qui permet au client web ViewCast tout comme à l'*Orchestrator* d'accéder aux informations stockées en base de données. Cette dernière est développée en Python et utilise notamment le package FastAPI.
- Une base de données relationnelle, dans le cas présent MySQL.

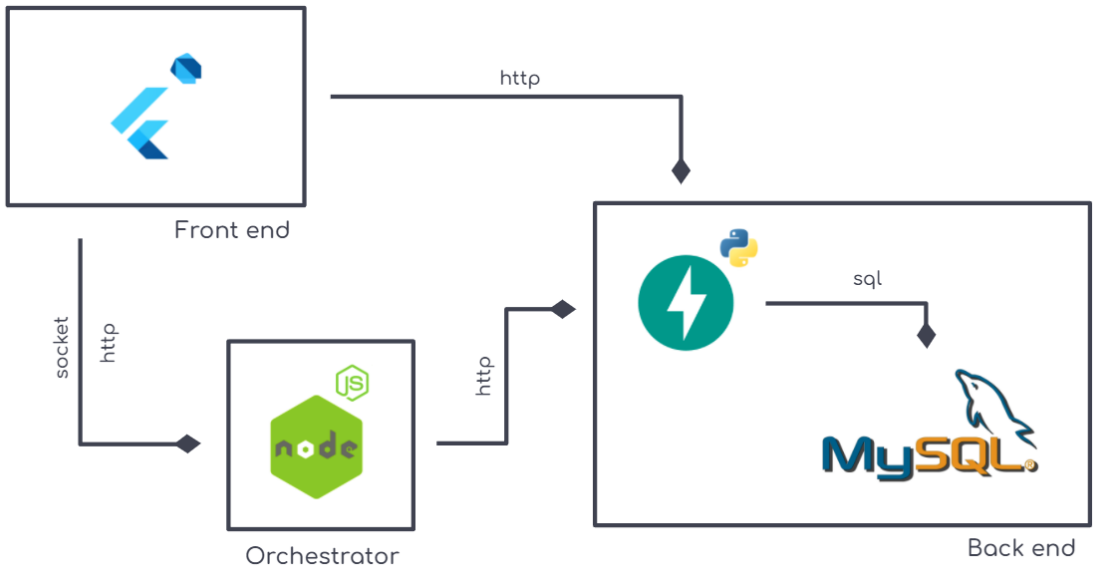

Figure 1: Architecture de ViewCast

Le rôle exact et les détails de chacun des composants sont fournis dans la partie *Analyse technique* propre à chacun de ces derniers.

Quant aux communications, elles sont de trois types distincts :

- HTTP pour les communications incluant l'API et entre le client web et l'*Orchestrator*.
- TCP/IP, au travers de sockets, entre le client web et l'*Orchestrator*
- SQL, entre l'API et la base de données.

Analyse technique

Cette section présente, pour chacun des composants du projet, leur rôle respectif au sein du projet, des schémas d'analyse, la justification de leur présence et des technologies qui s'y rapportent ainsi que leur liaison avec les autres composants.

Accès aux données

L'accès aux données à travers l'ensemble du projet est sous la responsabilité de l'API. Cette dernière est écrite en Python et basée sur la libraire FastAPI.

Grâce à elle, le client Flutter tout comme l'Orchestrator peuvent accéder aux ressources de la base de données de manière contrôlée et limitée, où les requêtes sont formulées envers l'API au travers du protocole HTTP et majoritairement au format JSON.

Quant aux méthodes mises à disposition par l'API, elles peuvent être retrouvées sur la documentation qui s'y rapporte grâce à OpenAPI (anciennement Swagger).

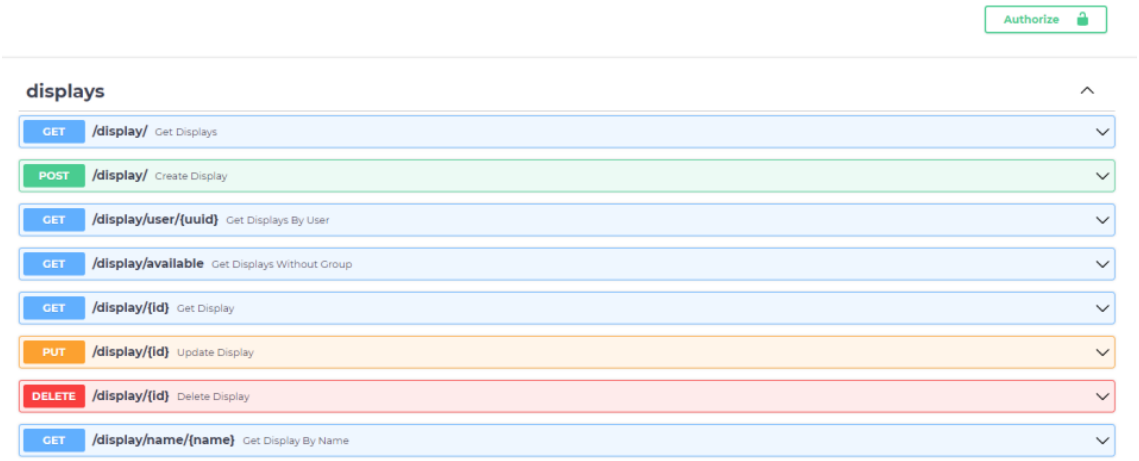

Figure 2: Exemple des méthodes disponibles

De plus, afin de simplifier les communications et les appels entre l'API et la base de données, nous avons sélectionné Tortoise en tant qu'ORM, et cela, car il présentait des performances élevées et une documentation moins hasardeuse que d'autres ORMs plus communs tels que Flask.

Les performances élevées sont nécessaires, car les ressources qui sont traitées par les clients de ViewCast sont principalement des contenus visuels, ce qui représente une charge (sur le réseau) relativement élevée.

Synchronisation des communications

Le rôle de l'Orchestrator consiste à synchroniser les diffusions entre les différentes télévisions. La technologie utilisée pour le réaliser est Node.js, celle-ci donnant la possibilité d'implémenter un fonctionnement asynchrone qui permet d'augmenter grandement les performances et le rendant donc efficace même lors d'une utilisation intensive.

Lorsqu'une télévision souhaite se connecter à un Cast, les opérations suivantes sont réalisées :

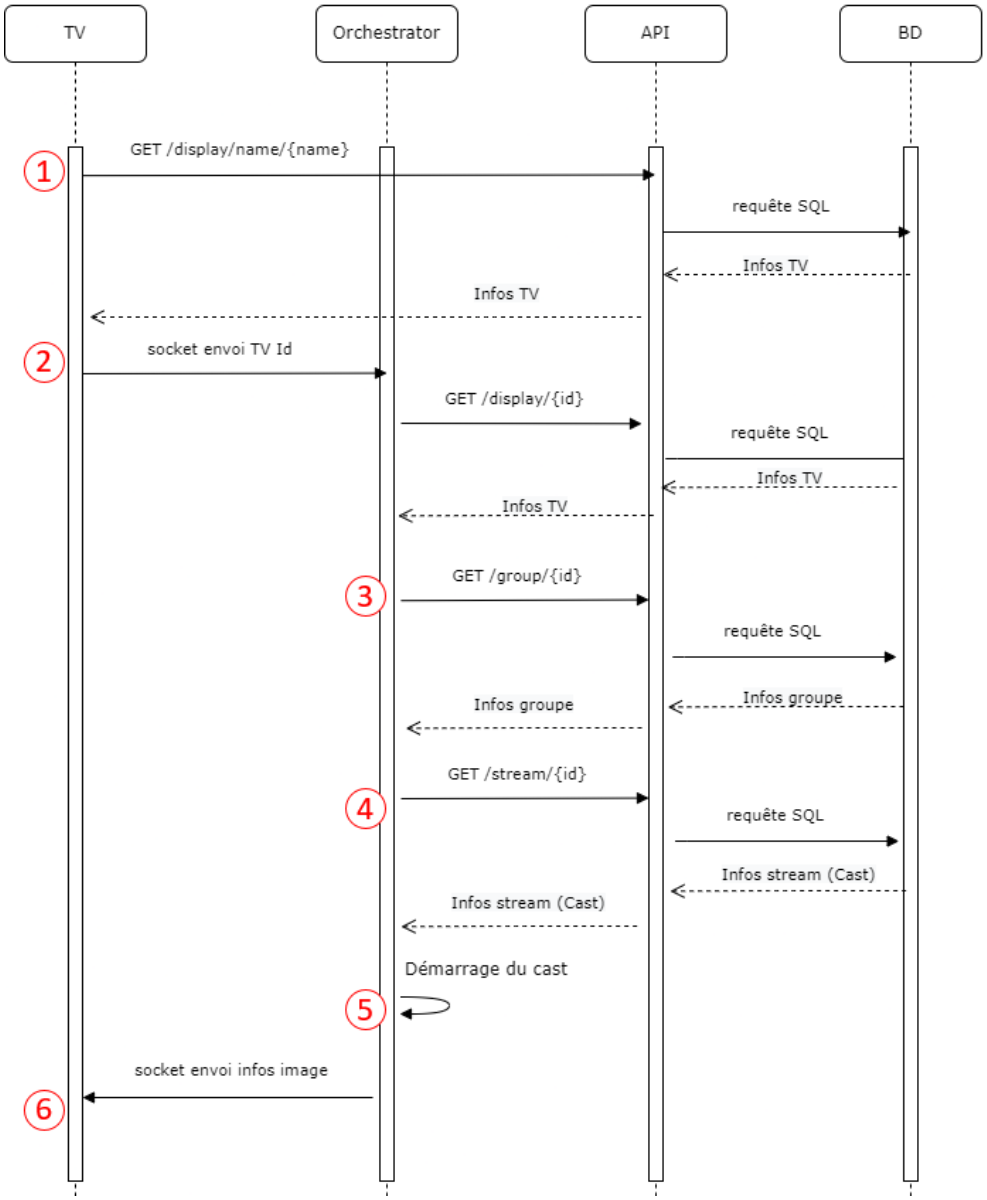

Figure 3: Connexion d'une TV à un Cast

1. La télévision envoie une requête à l'API pour récupérer les informations la concernant.
2. La télévision transmet son identifiant à l'Orchestrator via une communication en socket. L'Orchestrator va à son tour envoyer une requête vers l'API afin d'obtenir des informations supplémentaires concernant la télévision.
3. Si la télévision est associée à un groupe et que ce dernier n'est pas déjà chargé en mémoire, l'Orchestrator envoie une requête à l'API pour en récupérer les informations.
4. Si le groupe est associé à un Cast et que ce dernier n'est pas déjà chargé en mémoire, l'Orchestrator envoie une requête à l'API pour en récupérer les informations.
5. Si le Cast n'est pas en cours d'exécution, il est démarré. La télévision reçoit désormais les événements de ce Cast.
6. L'Orchestrator envoie directement (en socket) à la télévision les informations de l'image actuellement en cours de diffusion pour qu'elle l'affiche. À chaque changement d'image, un nouveau message est envoyé.

Lorsqu'une télévision se déconnecte d'un Cast et qu'aucune autre télévision ne suit cette diffusion, le Cast est automatiquement stoppé et est retiré de la mémoire de l'Orchestrator jusqu'à ce qu'une nouvelle télévision souhaite s'y connecter.

Client Web

La technologie utilisée pour le développement du client web est Flutter. Ce choix est motivé par sa popularité grandissante et par l'envie d'essayer ce dernier avec un nouvel axe en vue.

Logique de programmation

Ce point se concentre sur la logique utilisée lors de la programmation de l'application web.

Tout d'abord, d'un point de vue des communications, il est bien important de replacer les différents acteurs qui doivent communiquer de manière directe avec l'application web.

Le premier est l'API, qui se charge de récupérer, modifier, ajouter ou supprimer les données en base de données, suivant les requêtes HTTP du client. Ces requêtes sont donc toujours originaires du client et suivent le schéma suivant :

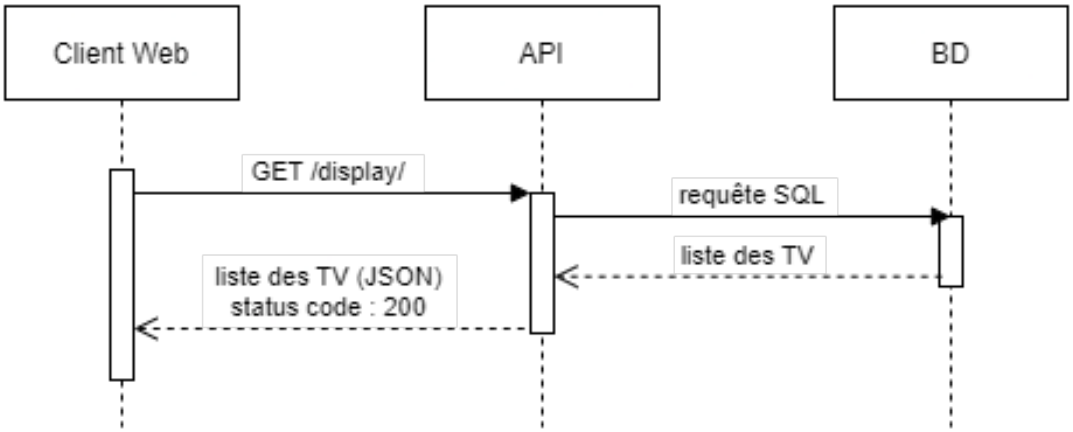

Figure 4: Exemple de communication avec l'API

La logique est sensiblement identique dépendant de la méthode HTTP envoyée (POST, DELETE, PUT), le seul changement notable étant la réponse et le status code de la réponse (201 pour un ajout par exemple).

Le second est l'Orchestrator. Il offre deux possibilités pour communiquer avec lui :

- La première, les requêtes HTTP, est utilisée pour notifier l'Orchestrator qu'un changement impactant vient d'avoir lieu (par exemple : la suppression d'un écran d'un groupe d'écrans).
- La seconde, à savoir la communication par socket HTTP, est utilisée à deux endroits (panneau des lives et en mode écran) et permet aux clients de recevoir les updates à effectuer en temps réel (quel écran doit être affiché actuellement sur telle ou telle télévision).

Ainsi, dans l'exemple de la suppression d'un écran de son groupe :

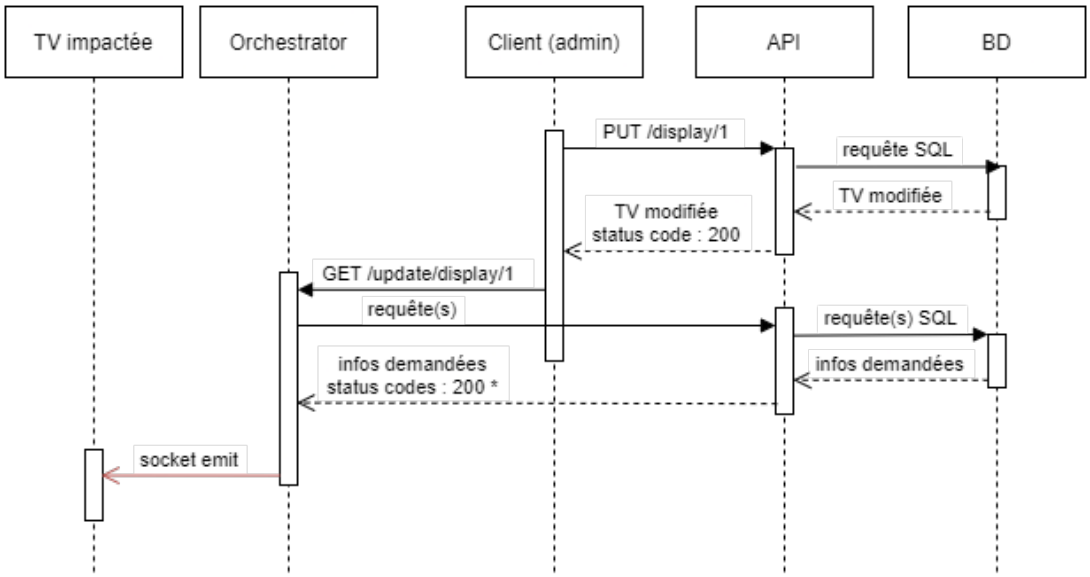

Figure 5: Exemple de communication avec l'Orchestrator

Le second point à aborder est la prise en main des Providers et leur utilisation tout au long du projet. Les Providers sont une fonctionnalité ajoutée par le package homonyme et qui permettent une meilleure gestion des données au travers de l'application.

En Flutter classique, pour permettre le rechargement dynamique des widgets, il faut recourir à l'utilisation de widgets dits stateful, qui sont relativement difficiles à gérer, surtout lorsque la hiérarchie de widgets devient lourde. En effet, il faut alors passer énormément de données du parent vers ses différents enfants, voir même recourir à des callbacks (fonctions codées dans le parent et passées en argument aux enfants).

Avec l'utilisation des Providers, tous les widgets sont stateless et la gestion de leurs états est prise en charge par les différents Providers. Cela permet de cibler les rechargements des widgets et de minimiser les données devant transiter d'un parent vers ses enfants. Cela est particulièrement pratique dans le cas de requêtes HTTP, qui peuvent ainsi être effectuées sur demande.

Interface Utilisateur

Ce point met l'emphase sur la réalisation de l'interface utilisateur.

La conception de l'interface a débuté par la réalisation de maquettes où ont été rassemblées les idées évoquées lors des réunions entre les membres du groupe. À travers ces maquettes, l'objectif principal était d'émettre des idées sur les fonctionnalités qui seraient présentes au sein de l'application, mais aussi d'aborder, lors d'une première itération, l'expérience utilisateur.

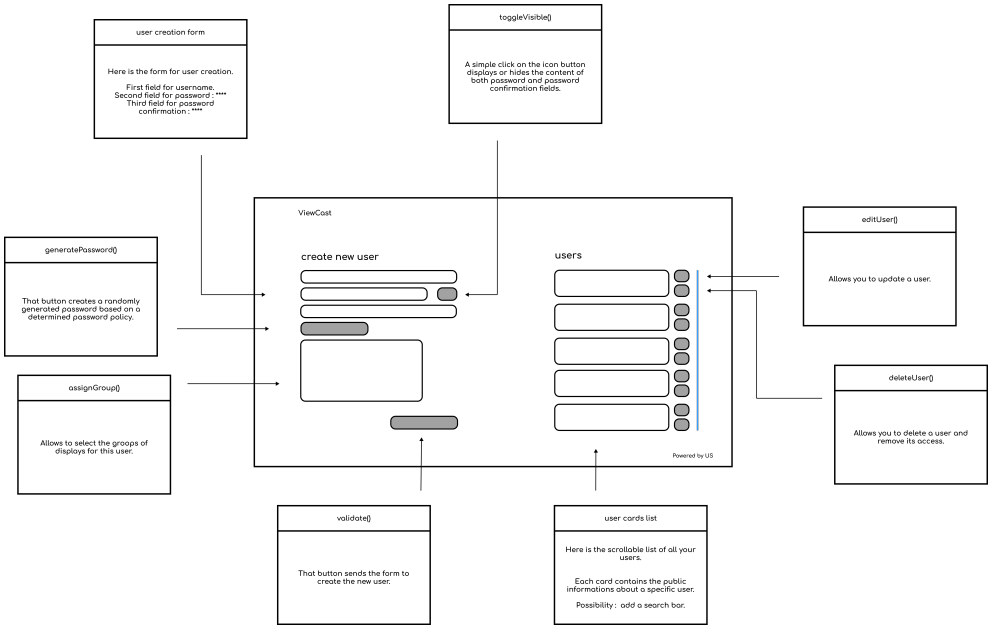

Figure 6: Maquette : 1ère itération

Les maquettes (exemple ci-dessus) ont ensuite été présentées au client afin d'obtenir un avis constructif sur le travail réalisé et, par la même occasion, vérifier avec ledit client si l'ensemble des fonctionnalités pensées étaient, pour lui, pertinentes.

Une fois les remarques du client prises en considération, une seconde itération de l'interface utilisateur a été réalisée afin d'obtenir un résultat plus proche de ce que le client désire.

Voici, ci-dessous, les trois idées et demandes bien spécifiques qui ont dirigé la conception des nouvelles maquettes :

1. Le menu et, par conséquent, la navigation au sein de l'application doivent être les plus simples possible pour l'utilisateur.
2. Les données doivent être au centre de l'attention et disposer du maximum de place possible afin de faciliter leur compréhension par l'utilisateur.
3. L'ensemble des formulaires de création et modification doit être présent sous la forme de pop-up, modales afin de ne pas encombrer l'écran lorsque cela n'est pas nécessaire.

Le résultat de cette seconde itération fut le suivant :

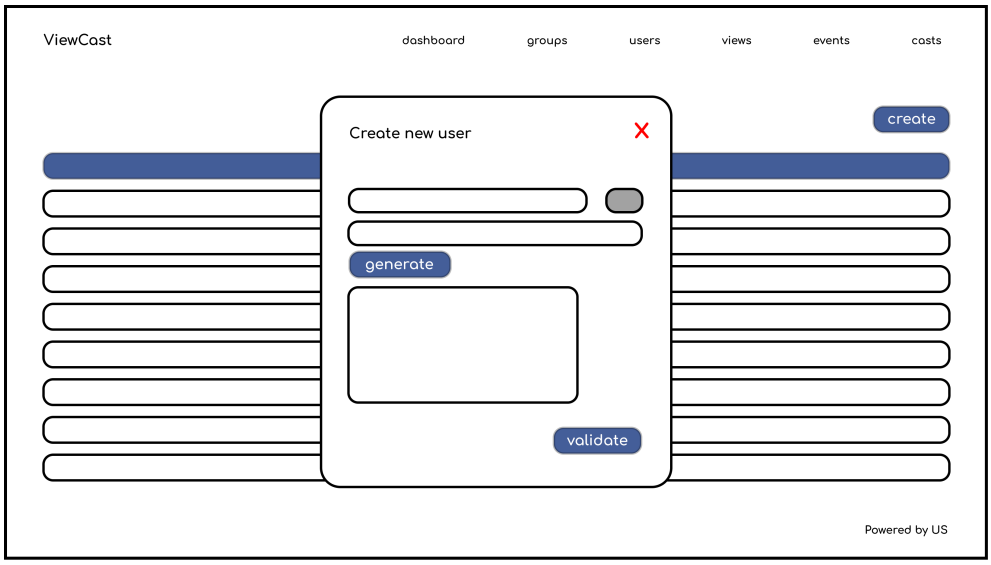

Figure 7: Maquette : 2ème itération

La nouvelle interface est caractérisée par plus de simplicité, une meilleure segmentation des ressources, une présentation plus sobre et bien plus de place pour les données.

Celle-ci permet à l'utilisateur de naviguer au travers de volets propres à chacune des ressources considérées par ViewCast (ex.: utilisateurs, views, groupes, etc.). Au sein, de chacun de ces volets, l'utilisateur a ensuite la possibilité de créer, éditer et supprimer lesdites ressources.

Ci-dessous, un exemple d'un écran finalisé de ViewCast :

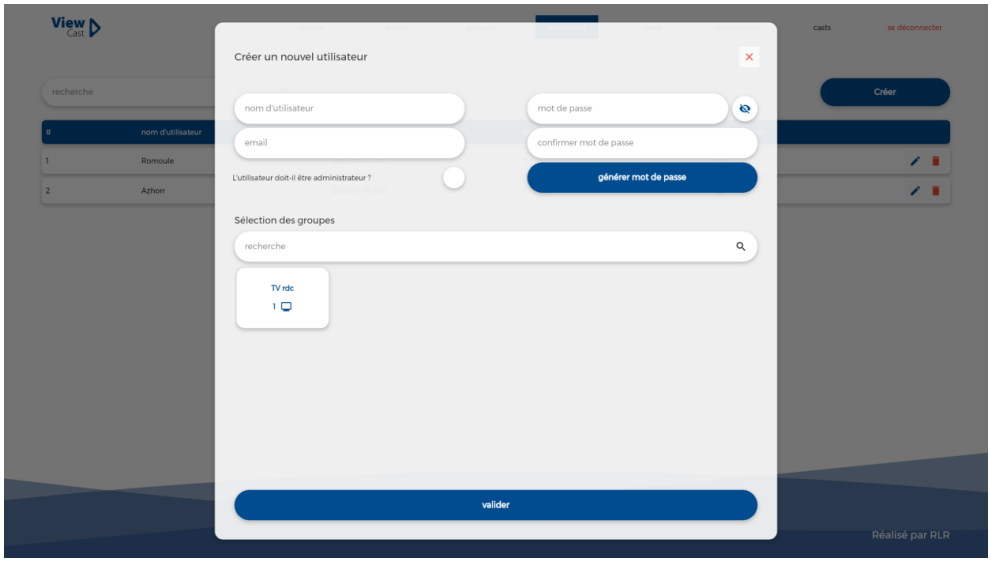

Figure 8: Capture d'écran de l'application

Résultat obtenu

Arrivée au terme de son développement, l'application dispose de la presque totalité des fonctionnalités qui avaient été envisagées et discutées avec le client. En effet, par souci de temps, nous n'avons pas pu cocher l'entièreté du cahier des charges. Cependant la liste des fonctionnalités présentes au sein de l'application n'est pas des moindres pour autant.

Ainsi, ViewCast vous permet de :

- Accédez soit à un volet d'administration, soit un volet prévu pour le casting.
- Administrez vos écrans et les groupes qui les contiennent.
- Administrez vos utilisateurs en leur octroyant des privilèges et des permissions sur des groupes d'écrans bien spécifiques.
- Créez des views, ensemble de contenu multimédia qui seront affichées sur vos écrans, en uploadant directement le contenu depuis votre machine.
- Créez votre programmation horaire grâce à des événements récurrents (basés sur un ou plusieurs jours de la semaine) ou à une date spécifique.
- Liez vos événements à un groupe d'écrans afin de partager le contenu désiré.
- L'ensemble des pages contiennent des barres de recherches afin de faciliter votre parcours de l'application.
- Créez, modifiez, copiez ou supprimez l'ensemble de vos ressources.
- Adaptez la langue de l'application à votre convenance.
- Surveillez en temps réel les multiples Casts en cours grâce à l'écran des Lives.

Figure 9: ViewCast : page d'accueil

Déploiement

Pour permettre un déploiement facile et rapide, Docker a été utilisé. Chacune des 4 parties du projet possède son conteneur propre et elles sont déployées en même temps à l'aide d'un docker-compose. La partie web est déployée sur un serveur Apache.

Pour la marche à suivre, vous pouvez vous référer à la section Déploiement du README disponible sur le GitLab ou dans les Annexes de ce document.

Conclusion

En conclusion, le travail réalisé répond aux demandes réalisables du client. Évidemment, de nombreuses améliorations fonctionnelles sont possibles, dont notamment :

- Affichage d'un horaire pour chaque groupe d'écrans, permettant de savoir facilement le programme prévu. Cette idée n'a pu se faire manque de temps et d'une librairie toute faite.
- Possibilité d'ajouter d'autres types de fichiers que des images (par exemple : vidéos).
- Possibilité d'ajouter un contenu dynamique tel qu'une horloge.
- Possibilité de créer les écrans directement depuis la partie administration (placer divers widgets, les redimensionner,...).

De plus, d'un point de vue optimisation, le code front end (Flutter) peut être nettoyé. Encore une fois, c'est le manque de temps qui a joué, le groupe ayant préféré se concentrer sur l'ajout de toutes les fonctionnalités proposées dans la solution finale.

Enfin, il est intéressant d'être critique du résultat. Nous avons pu constater lors du développement que Flutter n'est actuellement pas totalement mature pour du développement web. Bien que toutes les fonctionnalités souhaitées ont pu être implémentées, des complications sont survenues lors de la conception de certains éléments. Par exemple, le fonctionnement de la navigation est visiblement plus adapté pour une application mobile que pour une utilisation web.

Annexes

Déploiement

> Envirronement

Rendez vous sur le site de ubuntu et sélectionnez l'option 2 afin de faire une installation manuelle d'***Ubuntu server 20.04.3 LTS***.

Suivez la procédure d'installation standard

Partie 1 : récupération du projet git

> Clonez le projet viewcast et déplacez-vous dans le répertoire viewcast avec les commandes suivantes :

```
sudo git clone https://gitlab.com/DTM-Henallux/MASI/etudiants/streignard-remi/web/viewcast.git
```

A ce stage, introduisez votre nom d'utilisateur gitlab ainsi que le mot de passe correspondant à l'utilisateur ... sans vous tromper (:

Partie 2 : compilation du projet

> Installez flutter

```
sudo snap install flutter --classic
```

> Vérifiez si flutter est bien installé

```
sudo flutter
```

> Déplacez-vous dans le dossier du projet

```
cd viewcast
```

> Modifiez l'apiIP (ligne 1) du fichier suivant par l'IP de la machine sur laquelle vous installer les services.

```
sudo nano Applications/viewcast/lib/services/api_config.dart
```

> Déplacez-vous dans le dossier de l'application web

```
cd Applications/viewcast
```

> Compilez viewcast en version web

```
sudo flutter build web
```

Partie 3 : déployer le projet

> Installez docker

```
sudo snap install docker
```

> Déplacez-vous dans le dossier du projet

```
cd ../..
```

> Créez l'entièreté de la stack viewcast avec la commande suivante :

```
sudo docker-compose -p viewcast up -d
```

Une fois la stack créée, attendez 20 secondes que l'ensemble des machines se démarrent correctement

---

Partie 4 : initialiser le client web

> Finalement, Créez votre utilisateur root (administrateur) avec la commande suivante :

!! N'oubliez pas de changer les variables USERNAME EMAIL & PASSWORD par vos propres informations !!

- **Le champ USERNAME doit être d'une longueure minimale de 5 caractères**
- **Le champ EMAIL doit être dans format correspondant à un email : example@example.ex**
- **Le champ PASSWORD doit contenir au minimum 12 caractères, une majuscule, une minuscule, un chiffre et un caractère spécial**

```
curl -X 'POST' \
'http://localhost:8887/user/' \
-H 'accept: application/json' \
-H 'Content-Type: application/json' \
```

```
-d '{
"username": "USERNAME",
"email": "EMAIL",
"password": "PASSWORD",
"admin": true,
"groups_ids": [
]
}'
```

Dans le cas d'une réussite, vous recevrez une réponse de l'API contenant votre UUID !

Rendez vous ensuite sur l'IP de votre serveur au port 8888.

Vous pourrez vous connectez à la partie administration avec vos identifiants.