Développement Logiciel

P2019

La borne to be alive

Documentation

François Carlué

Antoine Rousselot-Vigier

Simon Sauvestre

Sébastien Gahat

TABLE DES MATIÈRES

DESCRIPTION

Notre mission

CampusFire est la suite du projet Totem débuté en Octobre 2017 par des élèves d'OMIS. Son objectif, tel qu'imaginé par Christian Jalain et François Brucker, est d'offrir à l'Ecole Centrale de Marseille une borne interactive sur laquelle les usagers pourront interagir à l'aide de leur téléphone. Cette borne permettra de rassembler et de partager, plus que de donner des informations déjà accessibles en ligne.

De plus, l'innovation technologique souhaitée est l'utilisation du téléphone comme une télécommande. L'idée est d'effectuer des actions sur la borne sans avoir à regarder son téléphone : ainsi, c'est la borne qui est au centre de l'attention.

Nous sommes donc intervenus au tout début de la phase de développement informatique, et avons implémenté les technos nécessaires à la connexion et à l'interaction entre un smartphone Android et un serveur.

Contexte et Organisation

Le projet a été lancé un an avant notre arrivée en 3A, dans le cadre du Projet OMIS, par Gabriel Carlotti, Joséphine Solier et Victor Yvergniaux.

Leur rapport est disponible ici :
https://github.com/campusfire/resources/blob/master/Team2017/Rapport-Final-29_03.pdf

Plus généralement, le projet est entièrement disponible sur Github :
https://github.com/campusfire

Celui-ci appartient durant chaque passation à François Brucker, qui le transmettra ainsi à la nouvelle équipe. Les collaborateurs du projet Github sont organisés par équipe. Ainsi, pour tout problème concernant un code écrit par la "Team2018" par exemple, un commentaire précédé d'un "@Team2018" nous permettra d'être notifiés et d'en répondre aux prochaines équipes.

Tout le code informatique pourra être retrouvé dans les "repos" associés au serveur et à l'application Android, et nous le libérons de tout droits d'auteur pour les prochains élèves de Digital·e ou tout autre élève de Centrale Marseille désigné par François Brucker.

VUE D'ENSEMBLE DE L'ARCHITECTURE

Schéma global

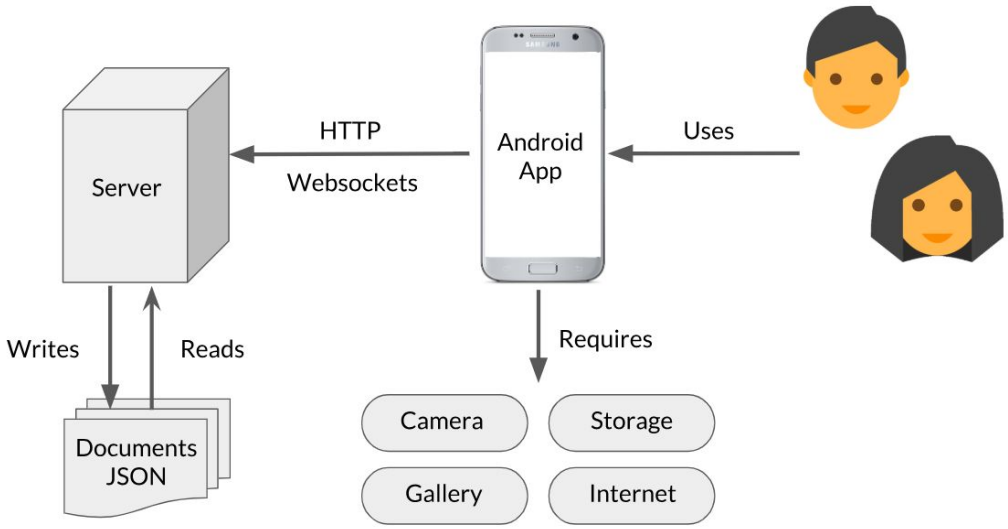


Figure 1 : Schéma global du projet

Architecture

Node.js

Nous avons choisi le framework Node.js pour coder notre plateforme. Node.js permet de développer à la fois la partie frontend et la partie backend du serveur, en javascript. C'est donc un framework très adaptable auquel on peut greffer tout un tas d'autres bibliothèques très pratiques accessibles via le manager de packages npm. Parmi ces bibliothèques, nous avons fait le choix d'utiliser Express.js qui est le framework le plus "standard" pour le développement de serveurs en Node et qui permet de créer très facilement et rapidement des applications web.

La principale limite de Node.js est que tout se fait de manière asynchrone, ce qui peut être un peu pénible pour le débogage et nécessite de prendre les précautions nécessaires en terme de gestion des exceptions par exemple. Autrement, c'est un framework très simple à prendre en main et pour lequel il existe un grand nombre de tutoriels en ligne.

API Externe

Afin de générer des QR codes pour gérer la connexion des utilisateurs, nous faisons appel à une API externe dont la documentation se trouve ici. Cette API permet tout simplement de générer des QR codes avec plusieurs paramètres variables comme les couleurs, la taille…

Base de données

Par manque de temps, et dans un souci de faire une première version fonctionnelle de la plateforme le plus tôt possible, nous n'avons pas implémenté de base de données. Actuellement, toutes les images reçues par le serveur sont stockées directement en statique (dans un dossier donc), et nous ne récupérons pas non plus d'informations sur les utilisateurs.
A l'avenir, il faudrait donc penser à en mettre une en place afin de sécuriser davantage la plateforme, et d'éviter de passer par des fichiers .txt ou .json pour garder des données.

Tests

Nous avons choisi le framework Jest pour les tests unitaires sur une partie de notre application. Jest a l'avantage de nécessiter assez peu de configuration et très peu de code préliminaire, ce qui en fait le framework de test le plus utilisé pour React Native, et un des plus utilisés pour les autres frameworks avec lesquels il est compatible : Typescript, Vue, Angular… Jest permet de faire des tests par snapshots : lancer un test pour la première fois sauvegarde le résultat sous forme de json, et relancer le test après modification du code permet de comparer le nouveau résultat au json sauvegardé précédemment. Il est bien sûr possible de vouloir effectivement changer le rendu d'une fonction et de réinitialiser un ou plusieurs tests. Jest permet donc de faire des tests facilement et de s'assurer que nos modifications successives ne changent pas le comportement final de nos fonctions et de nos objets.

En revanche, cela ne permet pas de faire du test-driven development, comme les fonctions doivent être déjà écrites pour que Jest en sauvegarde le comportement.

ARCHITECTURE DÉTAILLÉE

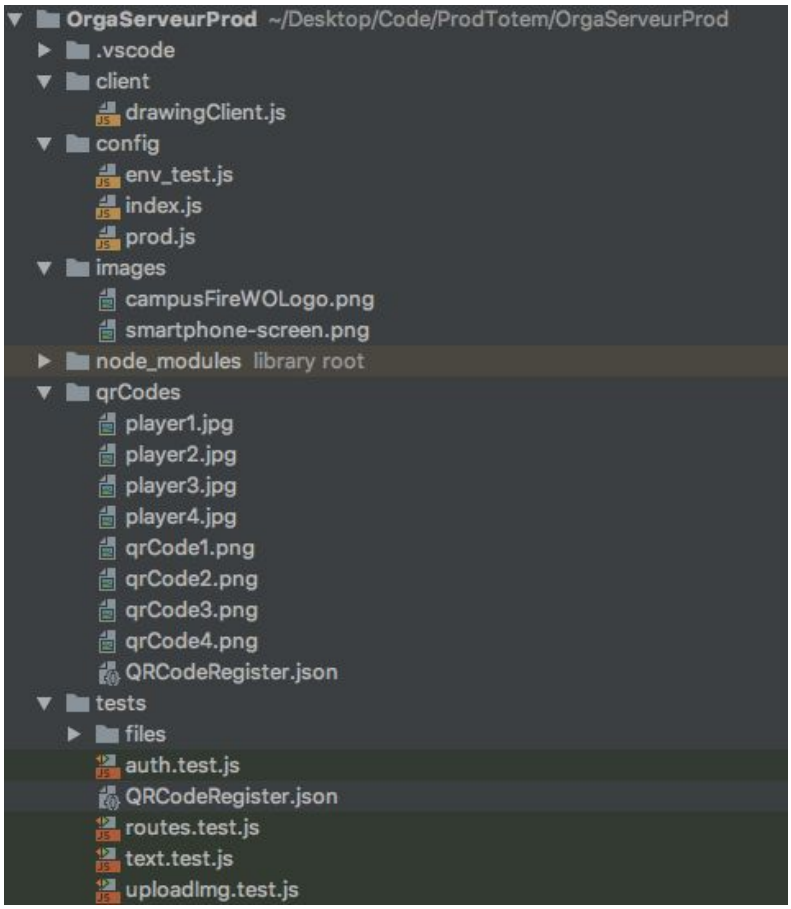
Utilisation

Fonctionnalités

- Authentification par QR code : l'utilisateur peut scanner un QR code affiché sur la page web et s'authentifier sur l'application par ce biais.
- Upload d'images : l'utilisateur peut sélectionner une image de sa bibliothèque locale puis l'envoyer sur le serveur qui l'affichera sur sa page principale
- Upload de texte : l'utilisateur peut écrire un texte sur son téléphone puis l'envoyer sur le serveur. Ce texte sera ajouté à la liste affichée sur la page principale.
- Download d'image : l'utilisateur peut télécharger une image
- Dessin en temps réel : l'utilisateur peut dessiner sur son application Android et voir son dessin en temps réel affiché sur la page web.

Arborescence

Server



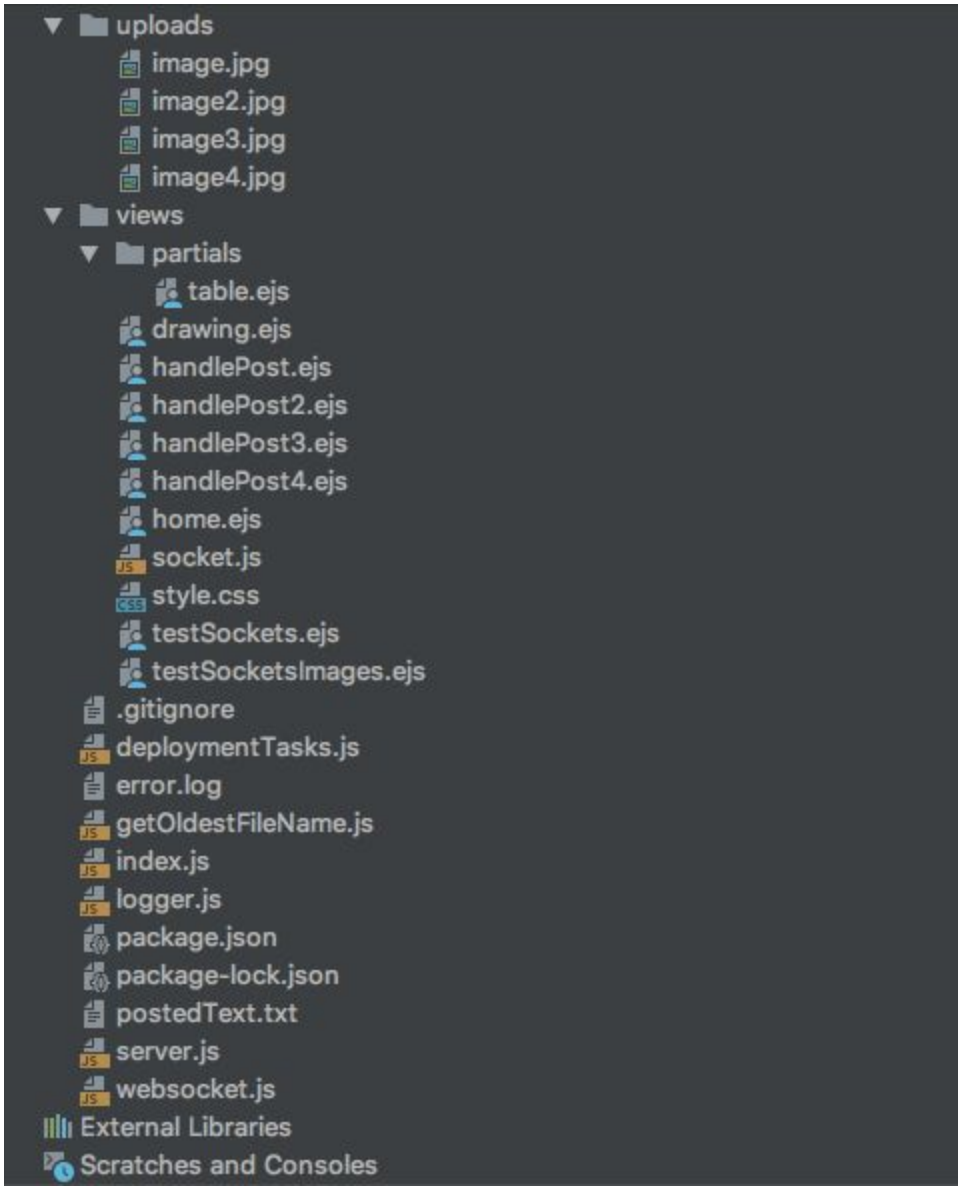


Figure 2 : Arborescence du projet côté Serveur (NodeJS)

Le fichier **server.js** sert au lancement du serveur. En l'exécutant tous les fichiers nécessaires au fonctionnement du serveur seront appelés et le serveur passe en écoute des requêtes.

Le fichier **index.js** est le fichier principal qui contient tous les appels de routage c'est à dire qu'il gère toutes les requêtes HTTP POST et HTTP GET du serveur. Souvent, les requêtes faites au serveur résultent en l'exécution d'une ou plusieurs actions, comme c'est le cas pour l'upload de fichiers, l'upload de texte, l'authentification ou encore l'affichage de la page principale "display".

Le fichier **deploymentTasks.js** est un autre fichier appelé au lancement du serveur car il se charge notamment de générer les QR codes à afficher sur la page web principale pour permettre l'authentification.

Le fichier **getOldestFileName.js** contient une fonction qui renvoie le fichier le plus ancien parmi un ensemble de fichiers. Cette fonction est utilisée dans index.js lors d'un upload de fichier (méthode POST sur /file-upload) afin de remplacer le fichier le plus ancien par le nouveau fichier uploadé.

Le fichier **websocket.js** est appelé par server.js au lancement du serveur pour établir une connexion websocket. Cette connexion permet de réceptionner les coordonnées du dessin fait sur l'application Android et de les ajouter en temps réel dans une liste. Cette liste est ensuite dessinée sur la page web drawing.

Le dossier **/views** contient les pages web à renvoyer lors des requêtes faites au serveur. Ces pages web sont au format .ejs ce qui permet un traitement facile avec express. Le sous-dossier /partials contient les parties du code .ejs qui a été factorisé, le fichier table.js est ainsi utilisé dans les autres fichiers .ejs.

Le dossier **/uploads** est celui qui stocke les différentes images uploadées sur le serveur.

Le dossier **/tests** rassemble les tests unitaires que nous avons réalisé sur le serveur avec jest et supertest. Son sous-dossier /files contient les fichiers exemples nécessaires à ces tests.

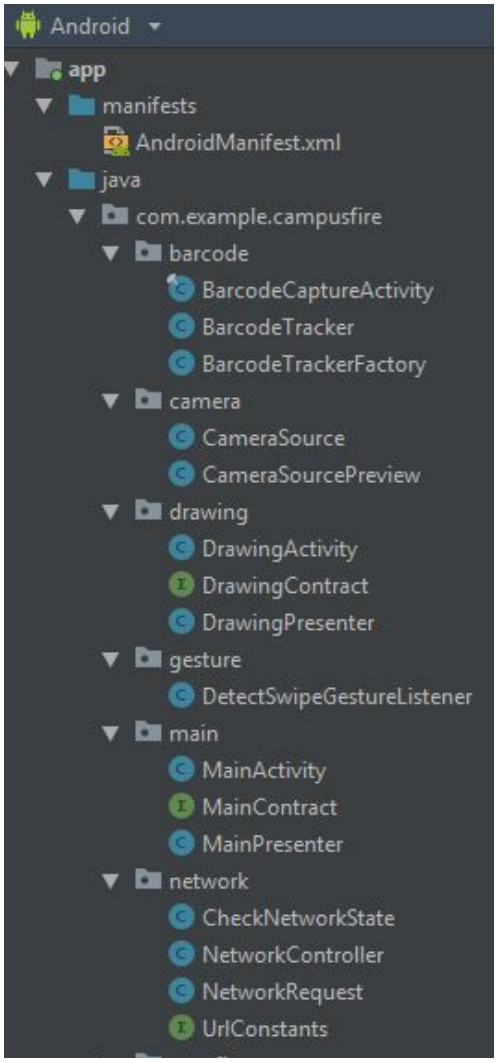
Le dossier **/qrCodes** contient les QR codes générés au lancement du serveur, ainsi que les images destinées à remplacer le QR codes sur la page web une fois qu'ils ont été scannés.

Le dossier **/images** contient les images utilisées dans nos visuels, notamment le logo.

Le dossier **/client** contient le code utilisé pour afficher le dessin à l'aide des websockets.

Le dossier **/config** contient les différents environnements déployés : test ou prod.

Client



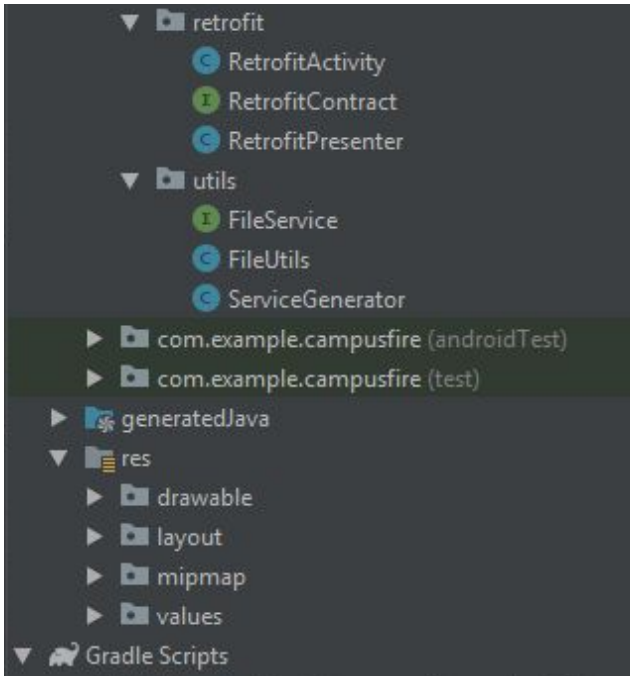
Le fichier **AndroidManifest.xml** est le fichier de configuration de l'application Android. Il contient les autorisations requises par l'application et les propriétés de chaque activité.

La structure des activités suit l'architecture Model-View-Presenter (MVP). Cet article et également celui-ci présentent le pattern de façon très satisfaisante dans le but de reprendre le projet.

Par exemple, le dossier **/java/main** contient la vue (**MainActivity.java**), le modèle (**MainContract.java**) et le présenteur (**MainPresenter.java**). Les interactions avec l'UI sont gérées par la vue, et tous les calculs sont dans le présenteur pour décharger la vue. Le modèle agit comme une interface.

Les dossiers **/java/drawing** et **/java/retrofit** contiennent les activités associées à l'envoi/réception d'images et de texte, et au dessin sur le canvas (voir explication partie server).

Les dossiers **/java/barcode** et **/java/network** contiennent quant à eux du code "utile" pour gérer la vérification du QR code et la connection au réseau, et ne sont pas organisés selon le pattern MVC.



Enfin, le dossier **/res** contient toutes les ressources liées au style de façon assez classique (voir cours Android). C'est dans ce dossier que vous pourrez modifier le layout d'une activité ou le logo de l'application, par exemple.

API & routage

Afin de gérer les fonctionnalités listées ci-dessus, nous avons fait le choix de créer notre propre API afin de gérer les communication entre le smartphone et la tablette à l'aide de simples requêtes HTTP. Nous avons donc implémenté un certain nombre de routes qu'il est important de connaître avant de reprendre le projet, afin comprendre la structure générale et savoir où il faut regarder en cas de bug. Toutes ces routes sont définies à l'aide du framework Express.js dans le fichier index.js.

Note: Pour tester ces routes: **http://node.oignon.ovh1.ec-m.fr/nomDeLaRoute**
Note 2: Une bonne manière de tester si les routes fonctionnent est d'utiliser le logiciel gratuit Postman

- **/display - GET**

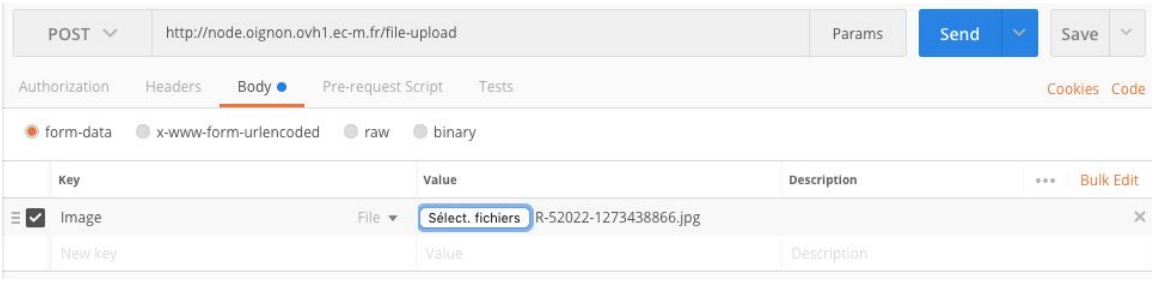
Il s'agit de la route sur laquelle est affichée la page principale de la plateforme. Pour la tester, il suffit d'ouvrir un navigateur et de rentrer l'adresse correspondante: http://node.oignon.ovh1.ec-m.fr/display

- **/drawing - GET**

Il s'agit de la route sur laquelle est affichée la page de dessin. Pour la tester, il suffit de rentrer l'adresse suivante dans son navigateur:
http://node.oignon.ovh1.ec-m.fr/display

- **/file-upload - POST**

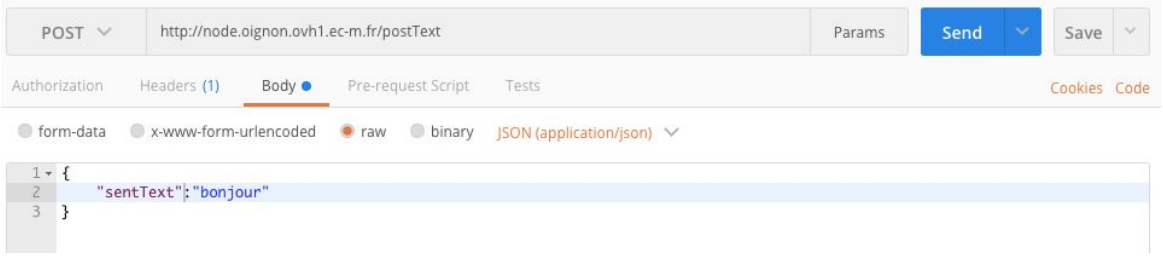
Permet de gérer la réception des images. Pour tester cette route avec Postman, il suffit d'y envoyer un fichier en format .png ou .jpg:



/!\ Pour que l'envoi fonctionne, il faut bien nommer l'image "Image" dans le champ "Key"

- **/postText - POST**

Permet de gérer la réception du texte. Pour tester cette route, il faut y envoyer du json comme ceci:



- **/authenticationPlayer - POST**

Permet de gérer la connection d'un utilisateur. Cette route reçoit du JSON sous cette forme:

Remarque: Le champ "barcodeSent" peut recevoir "CodeJoueur[i]" avec i allant de 1 à 4.
Dans la pratique, la route reçoit "CodeJoueur[i]" lorsque le QR code [i] est scanné sur l'application. Cela change alors ce QR code en l'avatar de l'utilisateur.

Upload d'images

Depuis l'application mobile, l'utilisateur peut envoyer des images (une par une). Chaque image envoyée apparaît alors sur la page principale. Nous avons pour l'instant limité le nombre d'images total sur l'interface à 4. Ainsi, dès qu'un utilisateur veut en uploader une 5ème, le serveur cherche l'image la plus ancienne et la remplace par la nouvelle.

Le protocole utilisé pour cette fonctionnalité est le protocole HTTP. Pour être plus précis, on envoie des requêtes post multipart. Concrètement, et sans rentrer dans les détails, ce type de requête permet d'envoyer autre chose que du JSON (donc du texte). Pratique donc lorsqu'on veut envoyer des fichiers.
Cette requête pointe sur la route /file-upload dans le fichier index.js

Côté serveur, l'image est reçue en binaire et stockée dans le dossier statique **/uploads**. Elle est ensuite "renommée" (on peut voir ça comme une conversion) afin d'être accessible et lisible en html.

Upload de texte

Depuis l'application mobile, l'utilisateur peut écrire du texte dans la case prévue à cet effet et l'envoyer au serveur node.js. Ce texte sera ensuite affiché sur la page principal en étant ajouté à la liste.

Le protocole utilisé pour cette fonctionnalité est le protocole HTTP. L'application android crée un ainsi une requête POST contenant en paramètre le texte à uploader. Cette requête pointe sur l'endpoint /postText du serveur et ce dernier récupère le texte puis l'ajoute au fichier postedText.txt.

Pour l'affichage du texte, le fichier /views/partials/table.ejs contient un tableau basé sur le framework Bulma. Ce tableau est rempli avec le contenu de postedText.txt via un script aussi présent dans table.ejs.

QR Codes

Depuis l'application mobile, l'utilisateur peut se connecter en scannant un des 4 QR codes présents sur le côté droit de la borne. Chaque Qr code contient une information de type CodeJoueur[i] qui est envoyée au serveur au moment du scan via protocole HTTP POST. Une fois que le serveur reçoit cette information, le QR code est remplacé par l'avatar de l'utilisateur qui l'a scanné.

En amont, au moment du déploiement du serveur, les 4 QR Codes sont générés à l'aide d'une api externe et stockés (au même titre que les avatars des utilisateurs) dans le dossier statique **/qrCodes.**

Dessin en temps réel

Depuis l'application Android, l'utilisateur peut dessiner et voir son dessin se réaliser en temps réel sur le serveur.

Accès : sur la page principale de l'application Android, l'icône vert représentant des pinceaux permet d'accéder à la page de dessin. Pour voir le dessin, il suffit d'accéder à l'endpoint /drawing (page http://node.oignon.ovh1.ec-m.fr/drawing).

La techno utilisée ici est la **websocket** : elle permet de créer un tunnel entre deux appareil afin qu'ils communiquent en temps réel. Une connexion est ainsi établie entre l'appareil Android et le serveur node.js via l'application CampusFire.

Sur l'application, à chaque fois qu'un mouvement est détecté sur le Canvas de dessin, les coordonnées du dessin en cours sont envoyés au serveur via un événement "draw_line" de socket.emit.

Côté serveur, à chaque fois qu'un "draw_line" avec des coordonnées est reçu, le fichier websocket.js se charge de les ajouter à une variable liste et de les envoyer au client drawingClient.js via un autre

événement “draw_line”. En accédant à la page /drawing le script /client/drawingClient.js est exécuté et crée le dessin à partir des données envoyée par le serveur.

Enfin, lorsqu’on fait un “retour” sur l’application Android, un événement “reset_line” est envoyé au serveur qui efface l’historique des coordonnées et transmet aussi au client drawingClient.js pour qu’il efface le dessin courant.

Tests serveur

Le dossier tests contient les différents tests unitaires à raison d’un fichier de test pour chaque fichier de code testé. Les librairies utilisées pour ces tests sont jest et supertest et la convention de nommage de ce dossier permet à Jest de trouver les tests à effectuer . Aussi, un environnement de test est utilisé pour lancer le serveur.

- **auth.test.js**

Nous disposons ainsi de tests pour l’authentification qui vérifie le refus lorsqu’un mauvais QR code est envoyé au serveur, et qui vérifie également la bonne authentification lorsque le QR code envoyé est valide.

- **routes.test.js**

Le test de routage permet de vérifier le bon renvoi de la page principale, ainsi que le bon renvoi de la page de dessin.

- **text.test.js**

Le fichier text.test.js concerne l’écriture et l’accès au fichier postedText.txt qui stocke les textes uploadés.

- **uploadImg.test.js**

Enfin le fichier uploadImg.test.js vérifie que l’upload d’une image au mauvais format est bien refusé.

Tests client

Grâce au pattern MVP (présenté plus tôt dans cette doc), le projet peut bénéficier de tests unitaires et UI clairement différenciés. Tout se passe dans le dossier src.

Le dossier **main** contient notre code de façon assez classique.
Le dossier **test** contient les tests unitaires.
Le dossier **androidTest** contient les tests UI.

- **tests unitaires**

Suivant une nouvelle fois le pattern MVP, les tests sont regroupés par “couche de code”, associés aux mêmes couches trouvées dans le dossier *main*.

Ces tests unitaires consistent uniquement à tester la liaison View-Presenter. Pour cela, on va *mocker* une vue, et vérifier que chaque cas d’appel à la vue par le présenteur est bien fonctionnel : vérifier que la fonction de la vue est appelée, et qu’elle renvoie le bon output.

Ces tests utilisent les frameworks **junit** et **mockito**.

- **tests d’intégration**

Les tests UI sont quant à eux plus compliqués à effectuer car ils doivent prendre en compte toutes les possibles interactions de l’utilisateur sur son téléphone directement. C’est donc des tests qui vont tester un à un tous les *OnClick*, *OnResult*, … récupérés par la vue avant leur traitement par le présenteur.

Le test est théoriquement simple : simuler un click et vérifier que telle fonction du présenteur est bien appelée et renvoie bien le bon output.

En pratique, pour faire cela, on peut utiliser les framework **junit** et surtout **Espresso** (très costaud pour ça).

Note Importante : il reste beaucoup de tests à écrire, et il est de notre point de vue très important de les coder en priorité. Pour cela, tout plein de doc existe, que ce soit du côté local ou UI. Encore une fois, il est nécessaire de bien maîtriser la structure type “archi hexa” associée au pattern MVP.

https://developer.android.com/studio/test
https://developer.android.com/training/testing/unit-testing/local-unit-tests
https://developer.android.com/training/testing/ui-testing/espresso-testing
https://android.jlelse.eu/the-real-beginner-guide-to-android-unit-testing-3859d2f25186 (Medium en général est très fourni en super tutos)

Configuration

L'application peut être installée sur téléphone Android uniquement. Pour cela, il faut cloner le repo Github sur son ordinateur et run le code sur son smartphone à l'aide d'Android Studio, ou générer un fichier .apk et l'ouvrir sur son smartphone.

Pour gérer la partie "serveur", toutes les consignes sont disponibles sur le README du repo associé.

CONCLUSION

C'est la fin de cette documentation (ouf !), et également le début d'une nouvelle année pour le projet Totem si vous êtes en train de lire ces mots.

Nous avons été très heureux de contribuer à ce projet que nous avons trouvé extrêmement formateur pour nous qui voulions continuer dans le développement web et mobile. CampusFire est pour nous le meilleur projet sur lequel on peut travailler durant le S9 à Centrale, et nous en profitons pour remercier Christian JALAIN et François BRUCKER pour leur volonté sans faille et leur engagement en ce qui concerne l'apparition de cette borne dans l'école.

Lors de notre soutenance finale, nous avons émis pas mal de pistes de développement pour améliorer la solution CampusFire. Vous pouvez d'ailleurs retrouver les slides dans le repo "resources" du projet.

Si vous avez une quelconque interrogation vis-à-vis du projet (par exemple), nous serons très heureux de vous répondre et de vous aider à continuer CampusFire.