

Documentation Technique

Application M2L

Enoncé des besoins

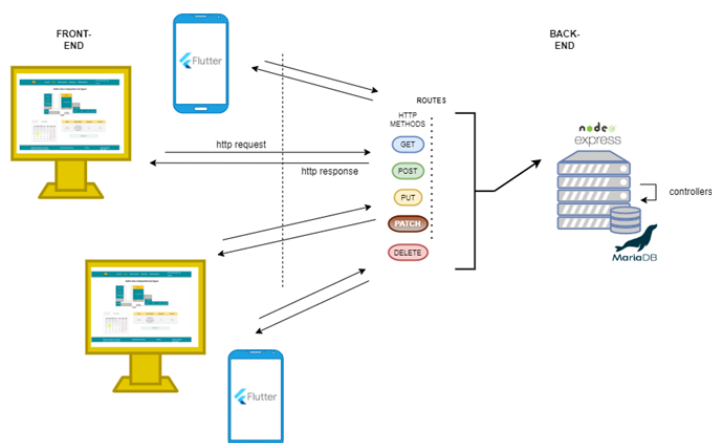
La Maison des Ligues de Lorraine (M2L), établissement du Conseil Régional de Lorraine, est responsable de la gestion du service des sports et en particulier des ligues sportives ainsi que d'autres structures hébergées. La M2L doit fournir les infrastructures matérielles, logistiques et des services à l'ensemble des ligues sportives installées.

La M2L souhaite mettre en place un service en ligne de gestion de réservations de salles de réunion. Elle souhaite également mettre en place un service mobile permettant la commande de services de restauration au cours d'un événement, la confirmation de la présence d'un participant à un événement, ainsi qu'une fonctionnalité permettant à un participant de se déclarer COVID positif.

Solution proposée

La solution proposée repose sur une architecture client-serveur. Les données sont stockées sur une base de données et accessibles via des requêtes envoyées à un serveur http. Les requêtes http permettant d'accéder aux données et de les manipuler, peuvent être envoyées au serveur depuis un navigateur internet dans le cas de la réalisation 1, ou une application mobile dans le cas de la réalisation 2.

Cette approche permet de centraliser les ressources afin d'éviter les problèmes de redondance et de contradiction. De même, le nombre réduit de points d'accès aux données permet une meilleure sécurité. Cependant, cette centralisation est également un inconvénient. En effet, en cas de dysfonctionnement du serveur, les données et la logique interne ne sont plus accessibles.



L'équipe

Notre équipe est constituée de trois développeurs full-stack, Guillaume Saulnier, Benjamin Sevault et Matthieu Siegel. Afin de répondre au mieux au besoin exprimé par la maison des ligues de Lorraine, nous nous sommes réparti les tâches tout au long du projet et sur toute la chaîne de production.

Technologies utilisées

Pour la réalisation de notre application web nous avons créé une base de données, une API (Application Program Interface) permettant d'accéder à la base de données et une interface homme-machine (IHM) pour que les utilisateurs puissent réserver les salles.

Notre base de données a été créée sous MariaDB, un fork de MySQL, qui est un système de gestion de base de données open source.



Notre API, quant à elle, est développée avec Node.js. Pour permettre le déploiement d'un serveur nous avons utilisé la librairie Express. Ce qui nous a permis de définir des URL pour accéder à notre base de données.



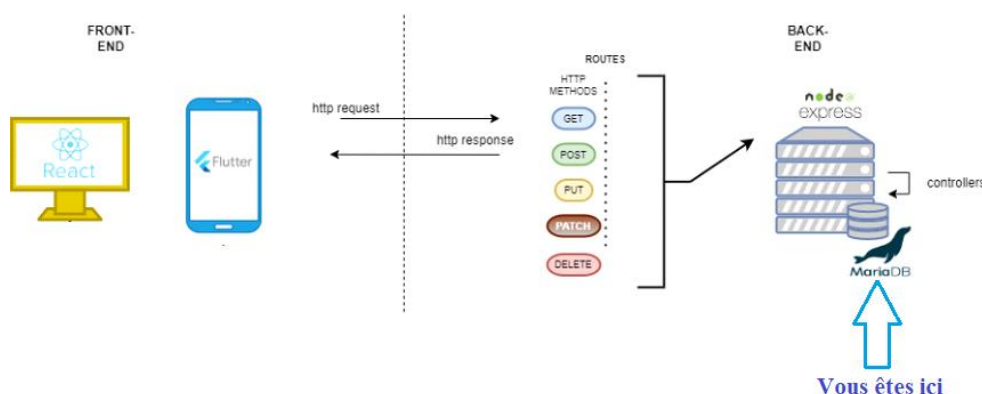
Pour la partie front-end web (IHM) nous avons pris le parti de développer une application React.js. Les avantages de celle-ci sont, la création de composants réutilisables, une meilleure maintenabilité du code et un chargement complet du site web.



Enfin, concernant l'application mobile, nous avons décidé d'utiliser le cadre de travail Flutter qui est basé sur le langage Dart. Le découpage en Widget et le langage orienté objet (POO), nous permet un développement plus facile et instinctif.



La Base de données



Conception

Dictionnaire de données

Pour concevoir le modèle de la base de données du projet, nous avons commencé par lister les tables et les champs associés dans un dictionnaire de données :

Tables USERS contenant les données des utilisateurs inscrits

id	INT	identifiant numérique d'un utilisateur
nom	VARCHAR	nom d'un utilisateur
prenom	VARCHAR	prénom d'un utilisateur
ddn	VARCHAR	date de naissance d'un utilisateur
email	VARCHAR	email d'un utilisateur
PASSWORD	TEXT	mot de passe hashé en MD5 d'un utilisateur
tel	VARCHAR	téléphone d'un utilisateur
adresse	VARCHAR	adresse d'un utilisateur
is_active	BOOL	Statut actif ou inactif d'un utilisateur
is_admin	BOOL	statut administrateur d'un utilisateur

Table SALLES contenant les données des salles

id	INT	identifiant numérique d'une salle
nom	VARCHAR	nom d'une salle
description	VARCHAR	descriptif d'une salle
capacite	INT	capacité maximum d'une salle, en nombre de personne
prix	FLOAT	prix d'une réservation d'une salle
is_active	BOOL	statut actif ou inactif d'une salle

Table PRODUITS contenant les données liées aux produits

id	INT	identifiant numérique d'un produit
nom_produit	VARCHAR	nom d'un produit
description	VARCHAR	description du produit
qte_dispo	INT	stock disponible
prix	FLOAT	prix d'un produit
is_active	BOOL	statut actif ou inactif d'un produit

Table TICKETS contenant les tickets de réclamation des utilisateurs

id	INT	identifiant numérique d'un ticket
date_ticket	DATE	date de création d'un ticket
date_problème	DATE	date de survenue du problème décrit dans le ticket
description	VARCHAR	description du problème rencontré par l'utilisateur
is_resolved	BOOL	statut résolu ou non-résolu d'un ticket

Table RESERVATIONS contenant les données liées aux réservations

id	INT	identifiant numérique d'une réservation
date_resa	DATE	date d'une réservation
is_paid	BOOL	statut du paiement de la salle
is_covide	BOOL	statut covid positif d'une réservation
check_participants	BOOL	statut terminé de la vérification de la présence des participants

Table PAIEMENTS contenant les données liées aux lignes de paiements

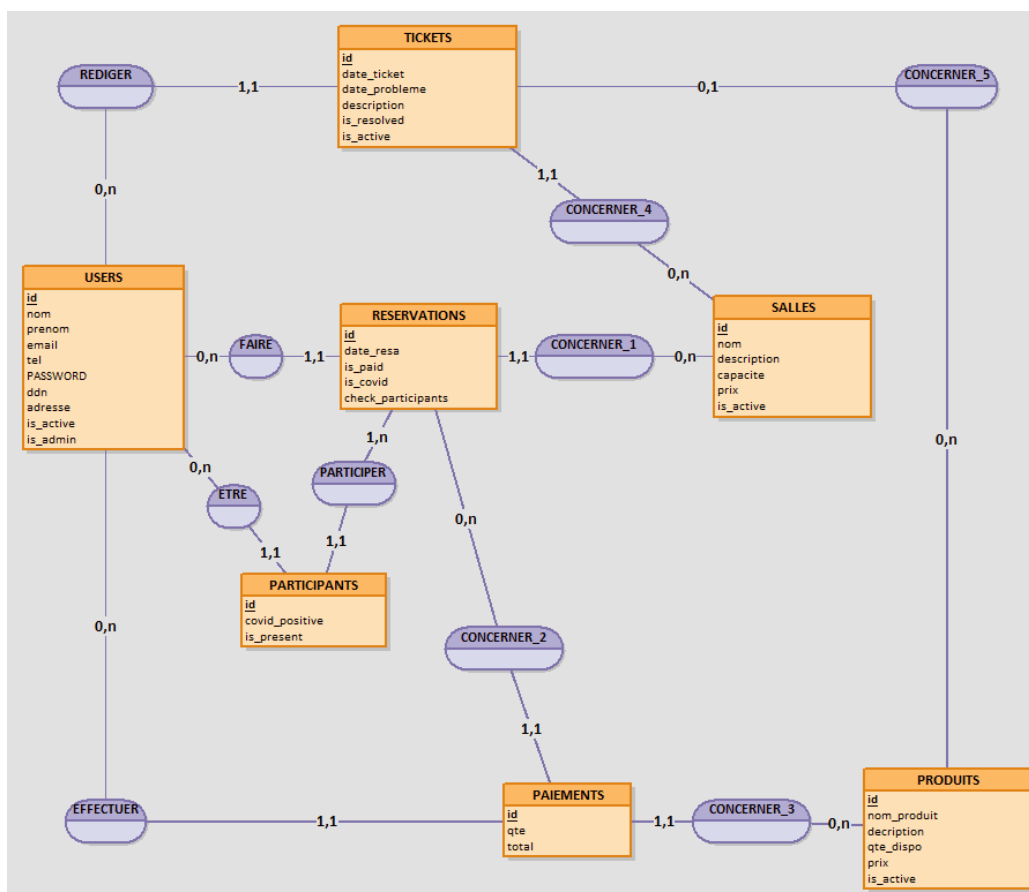
id	INT	identifiant numérique d'une ligne de paiement
qte	INT	quantité de produits achetés
total	FLOAT	montant total du paiement

Table PARTICIPANTS contenant les données des participants aux réunions

id	INT	identifiant numérique d'un participant à une réunion
covid_positive	BOOL	statut covid positif ou négatif d'un participant
is_present	BOOL	statut présent ou absent d'un participant à une réunion

Modélisation

Nous avons ensuite conçu un modèle entités - associations qui définit les relations entre les tables :



Scripts d'initialisation de la base de données

Partant de ce modèle nous avons écrit des scripts SQL pour créer le schéma la base de données, les procédures stockées liées aux fonctionnalités du projet et des données de test.

Extrait du script schema.sql, qui crée les tables conformément à la modélisation :

```
CREATE TABLE SALLES(  
  id INT NOT NULL AUTO_INCREMENT,  
  nom VARCHAR(255) NOT NULL,  
  description VARCHAR(255),  
  capacite INT NOT NULL,  
  prix FLOAT NOT NULL,  
  is_active BOOLEAN NOT NULL DEFAULT 1,  
  PRIMARY KEY(id)  
);
```

Extrait du script de procédures stockées p_salles.sql, qui crée les procédures nécessaires à la manipulation des données :

```
CREATE OR REPLACE PROCEDURE getOneSalle (IN p_id INT)  
BEGIN  
  SELECT id, nom, description, capacite, prix FROM SALLES  
  WHERE id = p_id;  
▶ Run SQL  
END //
```

Extrait du script de création de données de test, qui crée les enregistrements nécessaires aux phases de test :

```
INSERT INTO USERS (nom, prenom, email, tel, password, ddn, adresse)  
VALUES  
( 'test1_nom', 'test1_prenom', 'test1@email.com', '0123456789', MD5('test1'), '1981-01-01', 'test1_adresse'),  
( 'test2_nom', 'test2_prenom', 'test2@email.com', '0234567891', MD5('test2'), '1982-01-01', 'test2_adresse'),  
( 'test3_nom', 'test3_prenom', 'test3@email.com', '0345678912', MD5('test3'), '1983-01-01', 'test3_adresse'),
```

Pour des raisons de sécurité, nous avons créé un utilisateur MariaDB aux privilèges limités à la seule exécution des procédures stockées. C'est cet utilisateur qui se connecte à la base de données depuis le serveur http. Cela permet de limiter les risques d'attaque par injection SQL et d'accès non autorisé à la base de données.

Il n'est pas nécessaire de créer un tel utilisateur en faisant tourner le projet en local, les risques cités précédemment n'existant pas dans ce cas de figure.

Installation

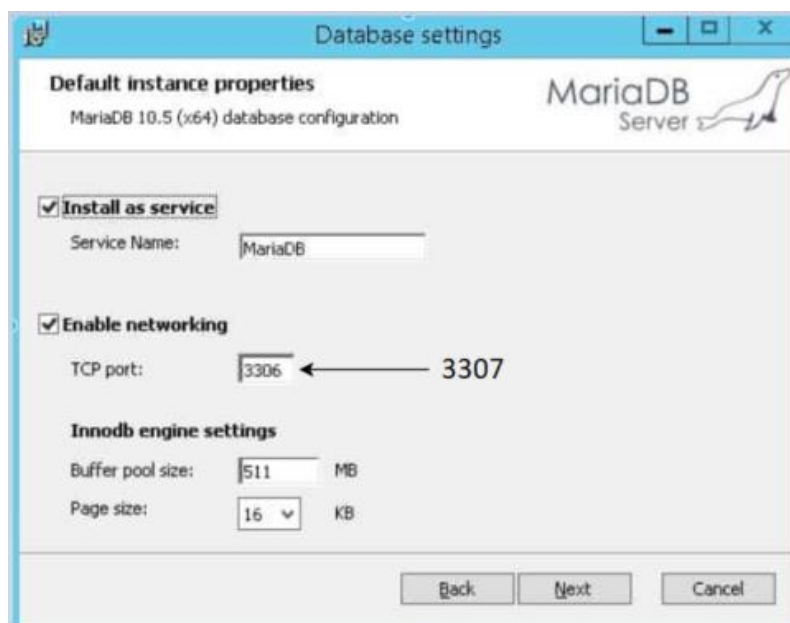
Windows

Pour installer MariaDB sur un système Windows, allez à la page de téléchargement du site officiel mariadb.com/downloads/community/community-server/ et sélectionnez la dernière version disponible et l'OS « MS Windows (64-bit) ».

Puis lancez l'installation et suivez les étapes décrites dans ce [tutoriel](#)

Note : ne pas tenir compte de la version de MariaDB dans le tutoriel.

Attention : lors de la troisième étape d'installation, modifiez le champ « TCP port » à **3307**.



Linux (Ubuntu)

Pour installer MariaDB sur un système Windows, allez à la page de téléchargement du site officiel mariadb.com/downloads/community/community-server/ et sélectionnez la dernière version disponible et l'OS « Ubuntu 20.04 Focal (amd64) ».

Puis lancez l'installation et suivez les étapes décrites dans ce [tutoriel](#)

Note : ne pas tenir compte de la version de MariaDB dans le tutoriel.

Création des utilisateurs

Créer un utilisateur **root** (avec des droits de création) sous MariaDB pour exécuter la commande d'initialisation de la base de données.

Créer un fichier **permissions.sql** dans le dossier `"/PPE_M2L_backend/database/"`

Dans celui-ci on va créer un utilisateur "webapp" qui n'aura que les droits d'exécution de procédures stockées sur notre base de données.

Script SQL à inclure dans le fichier **permissions.sql** :

```
CREATE OR REPLACE USER 'webapp'@'localhost' IDENTIFIED BY 'password';
```

```
GRANT EXECUTE ON M2L_DB.* TO 'webapp'@'localhost';
```

Le mot de passe password est à titre indicatif, vous pouvez mettre celui que vous désirez.

Création des variables d'environnements

Créer un fichier prod.env dans le dossier ./PPE_M2L_backend/config/ Ce fichier va contenir les variables de connexion à la base de données. Exemple de de fichier .env

#General

PORT=3001 -> Port d'écoute du serveur Express

#Database

HOST=localhost -> Adresse de la base de données

USER=webapp -> Utilisateur de la base de données

PASSWORD=password -> Mot de passe de l'utilisateur de la base de données

DATABASE=m2l_db -> Nom de la base de données dans MariaDB

DB_PORT=3306 -> Port d'écoute de la base de données

Initialisation de la base de données

Pour créer la base de données, insérer les procédures SQL et des données tests, et créer l'utilisateur webapp

Sous Windows

Exécutez le script : **npm run resetDBLight**

Sous Linux

Exécutez la série de scripts :

```
sudo mariadb -u root -p <./database/schema.sql
```

```
sudo mariadb -u root -p <./database/light_dummy_datas.sql
```

```
sudo mariadb -u root -p <./database/procedures/p_produits.sql
```

```
sudo mariadb -u root -p <./database/procedures/p_salles.sql
```

```
sudo mariadb -u root -p <./database/procedures/p_users.sql
```

```
sudo mariadb -u root -p <./database/permissions.sql
```

Test des procédures stockées

Les procédures stockées ont été testées avec l'outil [DBeaver](#). Cet outil permet de visualiser et de manipuler la base de données en temps réel, et d'exécuter des scripts et des procédures pour en vérifier le bon comportement.

Exemple de test

Procédure de création de salle :

```
CREATE OR REPLACE PROCEDURE createSalle (IN p_nom VARCHAR(255), IN p_desc VARCHAR(255), IN p_capa INT, IN p_prix FLOAT, IN p_active BOOLEAN)
BEGIN
    INSERT INTO SALLES (nom, description, capacite, prix, is_active) VALUES (p_nom, p_desc, p_capa, p_prix, p_active);
END //
```

Pour tester cette procédure on l'exécute avec DBeaver, et l'on contrôle si le résultat observé est conforme à l'attendu.

Etat de la table SALLES avant l'exécution de la procédure :

id	nom	description	capacite	prix	is_active
1	N.A.	N.A.	0	0	0
2	Majorelle	Service de sonorisation et vid	30	2 000	1
3	Restauration et	Service de sonorisation et vid	50	3 000	1
4	Grüber	Vidéo projecteur disponible	20	1 000	1
5	Lamour	Service de sonorisation et vid	25	1 500	1
6	Amphithéâtre	Service de sonorisation et vid	100	3 500	1
7	Longwy	Vidéo projecteur disponible	15	800	1
8	Daum	Vidéo projecteur disponible	20	1 000	1
9	Gallé	Vidéo projecteur disponible	20	1 000	1
10	Corbin	Vidéo projecteur disponible	20	1 000	1
11	Baccarat	Vidéo projecteur disponible	20	1 000	1

Exécution de la procédure avec DBeaver :

```
{ CALL m2l_db.createSalle(:p_nom,:p_desc,:p_capa,:p_prix,:p_active) }
```

Bind parameter(s)

#	Name	Value
1	p_nom	"test"
2	p_desc	"description test"
3	p_capa	42
4	p_prix	4400
5	p_active	1

```
{ CALL m2l_db.createSalle("test","description test",42,4400,1) }
```

☐ Hide parameters set in script

Use Tab to switch. String values must be quoted. You can use expressions in values

Ignorer OK Annuler

Etat de la base de données après exécution de la procédure :

id	nom	description	capacite	prix	is_active
1	N.A.	N.A.	0	0	0
2	Majorelle	Service de sonorisation et vid	30	2 000	1
3	Restauration et	Service de sonorisation et vid	50	3 000	1
4	Grüber	Vidéo projecteur disponible	20	1 000	1
5	Lamour	Service de sonorisation et vid	25	1 500	1
6	Amphithéâtre	Service de sonorisation et vid	100	3 500	1
7	Longwy	Vidéo projecteur disponible	15	800	1
8	Daum	Vidéo projecteur disponible	20	1 000	1
9	Gallé	Vidéo projecteur disponible	20	1 000	1
10	Corbin	Vidéo projecteur disponible	20	1 000	1
11	Baccarat	Vidéo projecteur disponible	20	1 000	1
12	test	description test	42	4 400	1

On constate que la procédure a permis de créer un enregistrement supplémentaire dans la base de données conformément aux paramètres qui lui ont été passés.

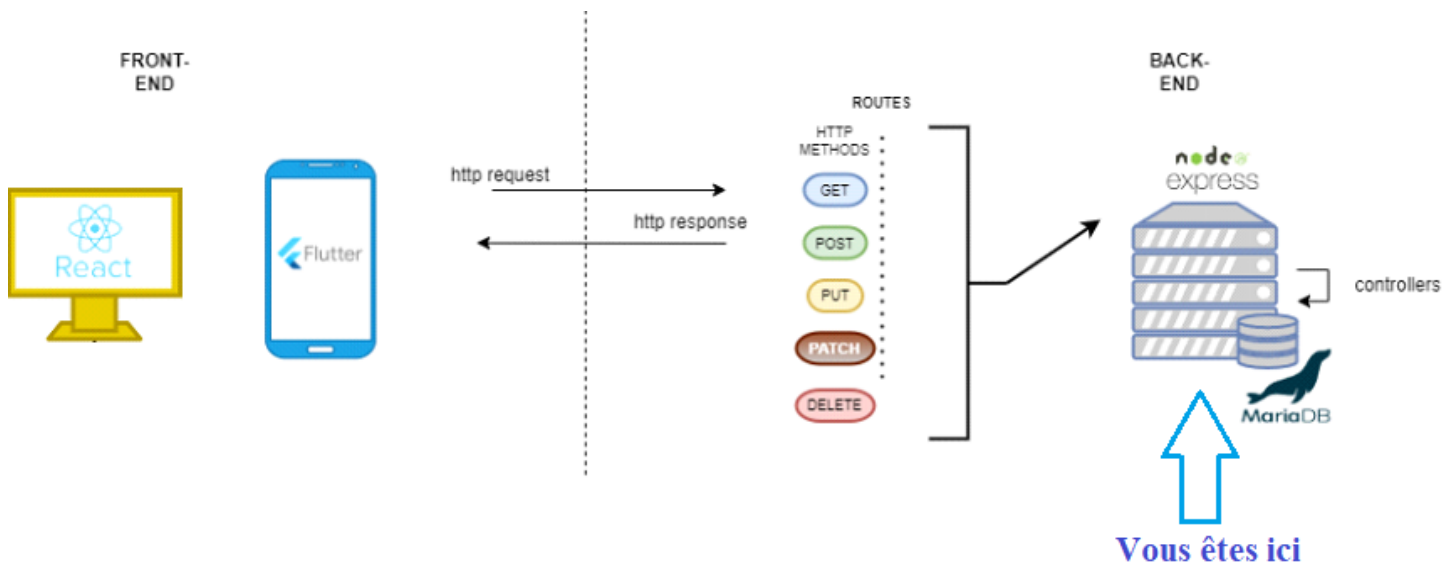
La procédure s'est exécutée sans erreur, le résultat observé dans la base de données est conforme à l'attendu, le test est valide.

Toutes les procédures du projet ont été testées de cette façon, et toutes ont donné des résultats de test satisfaisants.

Ainsi nous avons pu commencer le développement des autres modules du projet, en nous appuyant sur ces procédures pour accéder aux données et les manipuler de façon rigoureuse.

L'Application Program Interface (API)

Afin de pouvoir communiquer avec notre base de données depuis notre [webApp React](#) (réalisation 1) et [Application Mobile Flutter](#) (réalisation 2), il a été nécessaire de créer une API prenant à charge cette partie du besoin.



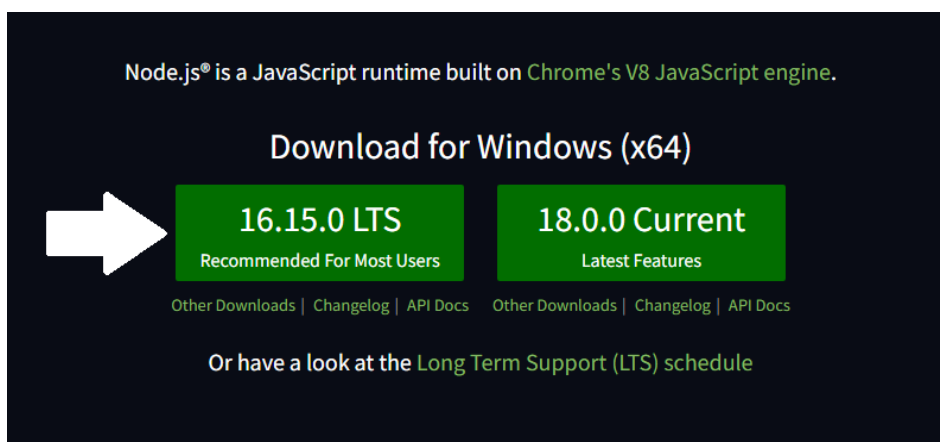
Dans cette section sera abordé les points suivants:

- Dépendances à installer
- Mise en fonctionnement
- Librairies utilisées
- Architecture
- Tests

Dépendances à installer

Afin de pouvoir fonctionner sur votre machine, notre API nécessite l'installation de nodejs. Vous pouvez télécharger la dernière version LTS (Long Term Support) en date à cette adresse:

<https://nodejs.org/> (Le numéro 16.15.0 peut changer)



Une fois nodejs installé, vérifiez le bon fonctionnement du programme avec la commande:

node -v

Celle-ci devrait fournir en résultat le numéro de la version précédemment installée.

A noter que l'installation de la Base de données est aussi nécessaire au bon fonctionnement de l'API, celle-ci a été couverte dans la partie [précédente](#).

Mise en fonctionnement

Il vous est possible de télécharger l'API directement à cette [adresse](#) ou bien de le cloner via ligne de commande (Mais il faudra alors avoir [installé Git](#) sur votre machine).

```
git clone https://github.com/BSevault/PPE_M2L_backend.git
```

Une fois cloné, naviguez à la racine du dossier téléchargé, puis tapez la commande suivante:

```
npm install
```

Cette commande installera automatiquement toutes les librairies utilisées dans l'API.

Une fois fait, vous pouvez démarrer l'application via la commande suivante

```
npm run prod (windows)
```

```
npm run prodlinux (linux)
```

Si l'installation fonctionne, le message suivant devrait apparaître sur votre console:

```
> backend@1.0.0 prod
> cross-env NODE_ENV=prod node app.js

Server listening on port 3001
```

Librairies utilisées

- [Express](#): Framework principal.
- [Cors](#): Restriction des requêtes à certaines origines.
- [Express-session](#): Création & gestion des sessions utilisateur.
- [Express-session-mariadb-store](#): Crée une table session (nécessaire pour répondre au [warning](#) du middleware express-session)
- [Mariadb](#): Connecteur entre l'API et la base de données
- [Express-promise-router](#): Gestion des routes (endpoint http) de l'API, permet l'utilisation des requêtes asynchrones.
- [Cross-Env](#): Gestion des commandes sur multiples OS
- [Dotenv](#): Gestion des variables d'environnements

Architecture

L'API comporte 3 dossiers notables:

- **config**
- **routes**
- **controllers**

Le contenu de ces 3 dossiers est utilisé dans le fichier app.js, à la racine du projet.

A noter que le dossier **utils** contient une unique fonction (call). Cette fonction a à charge la connexion entre l'api et la base de données.

Config contient les données nécessaires à la connexion à la base de données Mariadb (données utilisées par la fonction call, du fichier **utils**) ainsi que le port d'écoute de l'API (ici, 3001).

Routes contient la définition des pointeurs http auxquels répondra l'API. Ces routes sont séparées en 4 sous parties:

- **routes_produits**: définit les routes liées aux produits.
- **routes_salles**: définit les routes liées aux salles.
- **users/routes**: définit les routes liées aux users (séparées en plusieurs fichier dans le dossier **users** pour une meilleure lisibilité).
- **flutter**: définit les routes liées à l'utilisation de flutter.

Controllers contient la partie exécutive des requêtes reçues via les pointeurs cités ci-dessus. Les contrôleurs sont appelés par les routes afin d'effectuer des actions sur la base de données. A l'instar de l'architecture ci-dessus, ces contrôleurs sont séparés comme suit:

- **controller_produits**: requêtes liées aux produits
- **controller_salles**: requêtes liées aux salles
- **users/controller**: requêtes liées aux users (séparés en plusieurs fichier dans le dossier **users** pour une meilleure lisibilité)
- **flutter**: requêtes liées à l'utilisation de flutter.

Tests

Durant le développement de l'application, divers tests unitaires & d'intégrations ont été effectués.

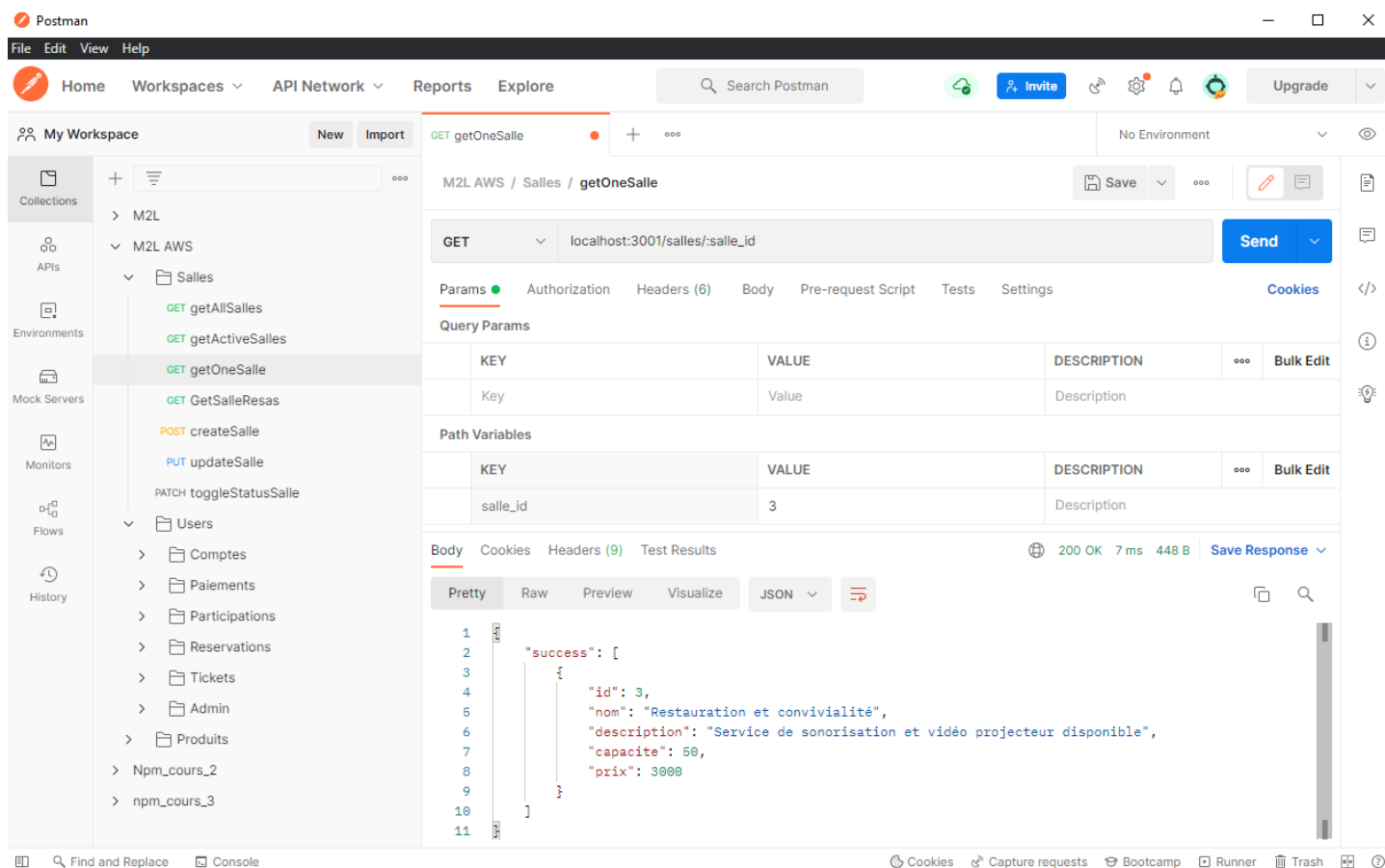
Ces tests consistent à utiliser un des endpoints déployés par l'API express au moyen d'un des moyens cités ci-dessous et de vérifier que le comportement attendu est correctement observé.

4 vecteurs de tests principaux sont à noter:

- Tests via [Postman](#)
- Tests via l'[IHM React](#)
- Tests via l'[IHM Flutter](#)
- Tests via le Web Browser directement

A ces quatre moyens d'entrée s'ajoute la vérification éventuelle des données transitées sur l'IHM utilisée, généralement au moyen de `console.log()` & `print()` selon le langage utilisé, ainsi que sur la base de données ; soit par ligne de commande, ou à l'aide de [DBever](#).

Exemple d'un test de la route `/salles/:salle_id` via postman:



The screenshot shows the Postman interface with a GET request to `localhost:3001/salles/:salle_id`. The path variable `salle_id` is set to `3`. The response is a JSON object with the following structure:

```

{
  "success": [
    {
      "id": 3,
      "nom": "Restauration et convivialité",
      "description": "Service de sonorisation et vidéo projecteur disponible",
      "capacite": 50,
      "prix": 3000
    }
  ]
}

```

Cet endpoint est initialisé dans `app.js` via la ligne suivante avec un préfixe commun à toutes les routes impliquant les salles de `"salles"`:

```

app
  .use('/produits', routes_produits)
  .use('/salles', routes_salles)
  .use('/users', routes_users_comptes)

```

Le `/:salle_id` est ensuite ajouté dans le fichier `routes/route_salles.js` et dans le cas de ce test, la requête est de type GET, appelant le contrôleur `getOneSalle`.

```
router
  .route('/:salle_id')
  .get(getOneSalle)
  .put(updateSalle)
  .patch(toggleStatusSalle)
```

Le contrôleur `getOneSalle` est défini dans le fichier `controllers/controller_salles.js`, il contient le code suivant:

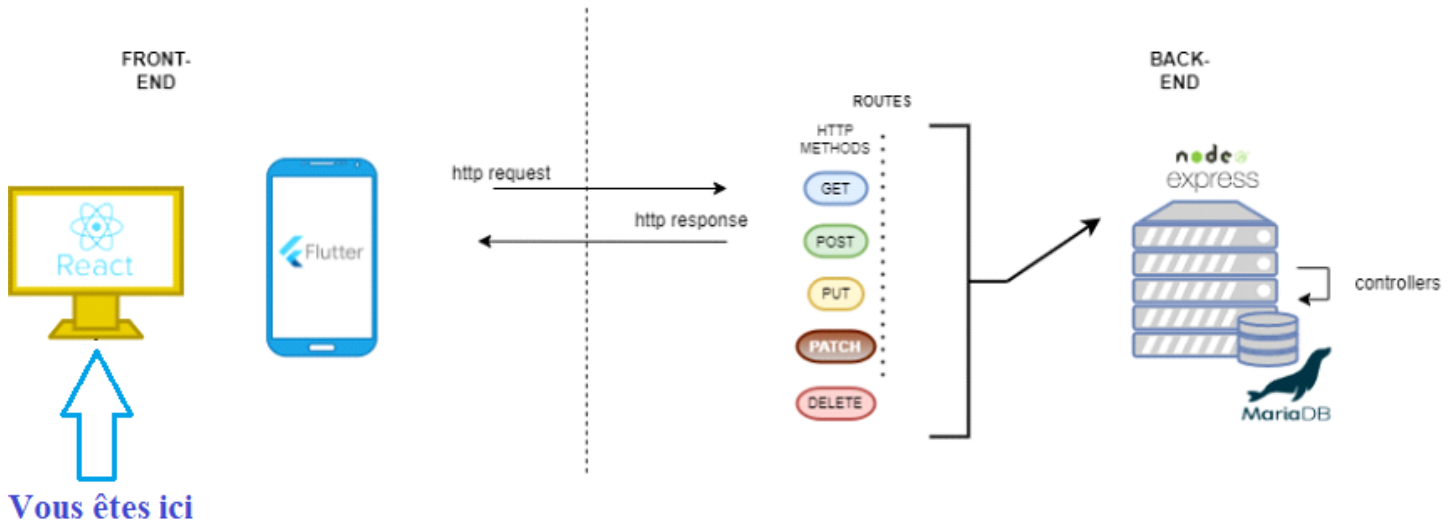
```
getOneSalle: async (req, res) => {
  const { salle_id } = req.params;
  await call(res, async (connexion) => {
    const result = await connexion.query("CALL getOneSalle(?)", [salle_id]);
    return res.status(200).json({ success: result[0] });
  });
},
```

Cette fonction fait un appel à la base de données via la procédure stockée `getOneSalle(salle_id)` et renvoie les données liées à l'id fournie.

Dans le cas d'un mauvais fonctionnement lors d'un test similaire, les messages d'erreurs sont utilisés pour trouver l'origine du bug, et s'ensuit sa résolution avant intégration de la route au code commun (via gitHub). Une fois la route intégrée au reste de l'API, des tests similaires sont menés afin de vérifier qu'aucun bug n'a été introduit dans la version commune (test d'intégration).

L'Interface Homme Machine Web (IHM React)

La M2L souhaite mettre en place un service en ligne de gestion de réservations de salles de réunion. Pour répondre à ce besoin, une solution Client Léger a été conçue sous la forme d'une WebApp React.



La solution est en mesure de répondre aux besoins évalués dans le useCase ci-dessous:



Dans cette section sera abordé les points suivants:

- Dépendances à installer
- Mise en fonctionnement
- Librairies utilisées
- Architecture
- Tests

Dépendances à installer

Afin de pouvoir fonctionner sur votre machine, notre webApp React nécessite l'installation de nodejs.

Se référer à l'installation évoquée [ici](#)

De plus, l'application nécessite une adresse de backend sur laquelle faire ses requêtes. Cette adresse est définie dans le fichier src/components/contexts/AuthContext.js

```
9  const AuthContextProvider = ({ children }) => ({
10    const [user, setUser] = useState('');
11    const endpoint = 'http://localhost:3001';
12    // const endpoint = 'https://groupe-a.lemonravioli.ovh';
```

← Adresse locale

← Adresse en ligne

Note: A l'heure de la rédaction de ce document, l'adresse "groupe-a.lemonravioli.ovh" est un backend de secours pour l'épreuve du BTS, il est improbable que celui-ci reste disponible.

Mise en fonctionnement

Il vous est possible de télécharger la webApp directement à cette [adresse](#) ou bien de le cloner via ligne de commande (Mais il faudra alors avoir [installé Git](#) sur votre machine).

```
git clone https://github.com/BSevault/PPE_M2L_frontend.git
```

Une fois cloné, naviguez à la racine du dossier téléchargé, puis tapez la commande suivante:

```
npm install
```

Cette commande installera automatiquement toutes les librairies utilisées dans l'application React

Une fois fait, vous pouvez démarrer l'application via la commande suivante:

```
npm start
```

Cette commande mettra à disposition une version local de l'application Web à l'adresse suivante:

```
http://localhost:3000
```

A noter que, bien que fonctionnelle, cette commande est prévue d'être utilisée uniquement dans un contexte de développement. Se référer à la documentation de [React](#) pour la procédure de déploiement en production.

Librairies utilisées

- [React](#): Framework principal.
- [React-calendar](#): Calendrier manipulable sous la forme d'un composant react
- [Pdfmake](#): Créer un document PDF selon les données fournies

- [Axios](#): Analogue à fetch, facilite les requêtes http.

Architecture

La webApp profite du squelette auto-généré par le framework React pour fonctionner. La plupart des fichiers liés au développement de l'application spécifique à la Maison des Ligues de Lorraine peut être trouver dans le dossier `src/`

Dans ce dossier se trouve 4 sous dossiers principaux:

- **assets**
- **components**
- **pages**
- **hooks**

Assets contient sans distinction tout documents imagés et non directement liés à l'exploitation du framework React.

Components contient, comme son nom l'indique, les composants REACT développés pour répondre aux besoins de la M2L. 2 sous dossiers présents dans components/ ont une importance notable:

- **components/Routes**: Contient les éléments nécessaires à l'utilisations des routes dans la WebApp
- **components/contexts**: Contient et génère des données utilisées par la WebApp, notamment les sessions ainsi que l'adressage des requêtes (endpoint)

Pages contient les composants REACT utilisés pour l'affichage des pages de la WebApp M2L.

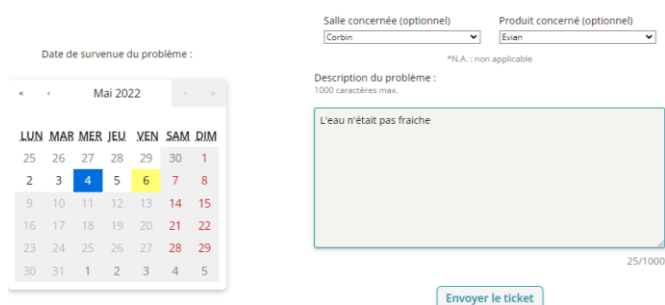
Hooks contient des composants utilitaires réduisant la syntaxe nécessaire à l'utilisation de certaines fonctions souvent appelées.

Tests

Les différentes fonctionnalités de l'application web ont fait l'objet de tests unitaires au cours de leur développement, puis, une fois validées, de tests d'intégration à l'ensemble du projet.

Test de la fonctionnalité d'ajout d'un ticket de réclamation :

- l'utilisateur souhaite ajouter un ticket de réclamation pour un évènement ayant eu lieu le 04 mai 2022, et qui concerne la salle Corbin et le produit Evian, avec pour description « L'eau n'était pas fraîche ».



The screenshot shows a web form for submitting a complaint. On the left, there is a calendar for May 2022 with the 4th highlighted. The main form has two dropdown menus: 'Salle concernée (optionnel)' with 'Corbin' selected, and 'Produit concerné (optionnel)' with 'Evian' selected. Below these is a text area for 'Description du problème : 1000 caractères max.' containing the text 'L'eau n'était pas fraîche'. At the bottom right of the text area is a character count '25/1000'. A blue button labeled 'Envoyer le ticket' is at the bottom center.

Vos réclamations en cours

N° ticket	Date du ticket	Date du problème	Salle concernée	Produit concerné	Description
-----------	----------------	------------------	-----------------	------------------	-------------

Vous n'avez pas de réclamations en cours

On constate que le ticket a bien été ajouté aux tickets en cours de l'utilisateur, avec les informations qu'il a renseignées.

Date de survenue du problème :

Salle concernée (optionnel) :
 Produit concerné (optionnel) :

*N/A : non applicable

Description du problème :
1000 caractères max.

L'eau n'était pas fraîche

25/1000

Envoyer le ticket

Vos réclamations en cours

N° ticket	Date du ticket	Date du problème	Salle concernée	Produit concerné	Description
10	06/05/2022	04/05/2022	Corbin	Evian	L'eau n'était pas fraîche

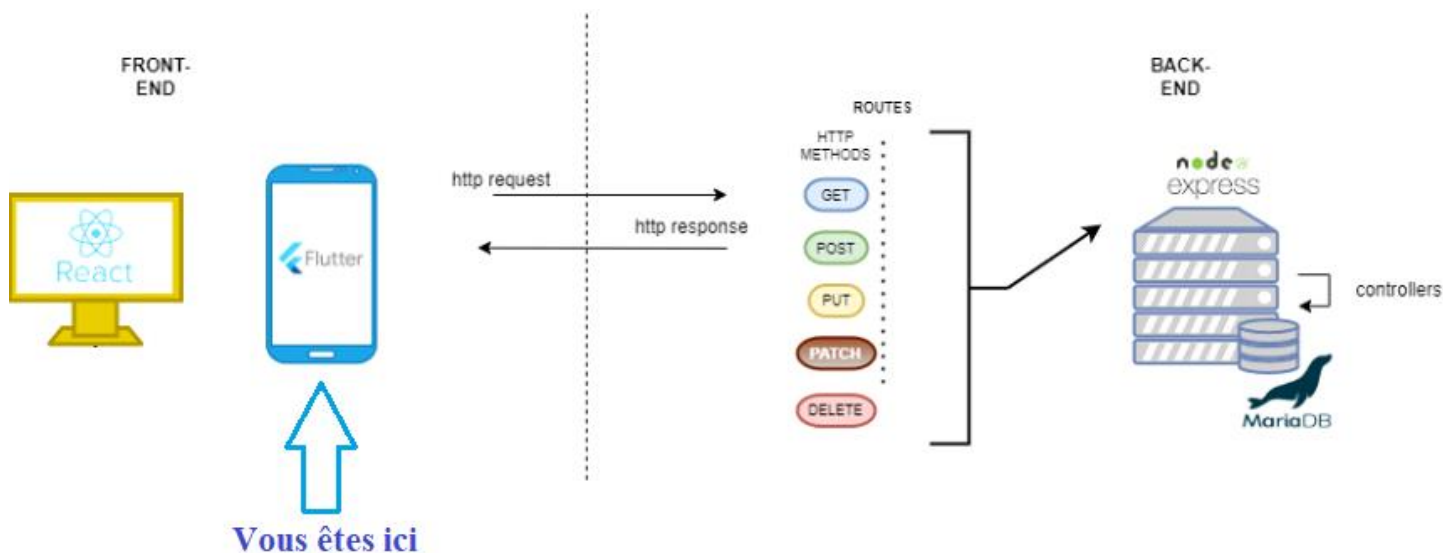
On constate que ces informations ont également été mises à jour correctement dans la base de données :

<

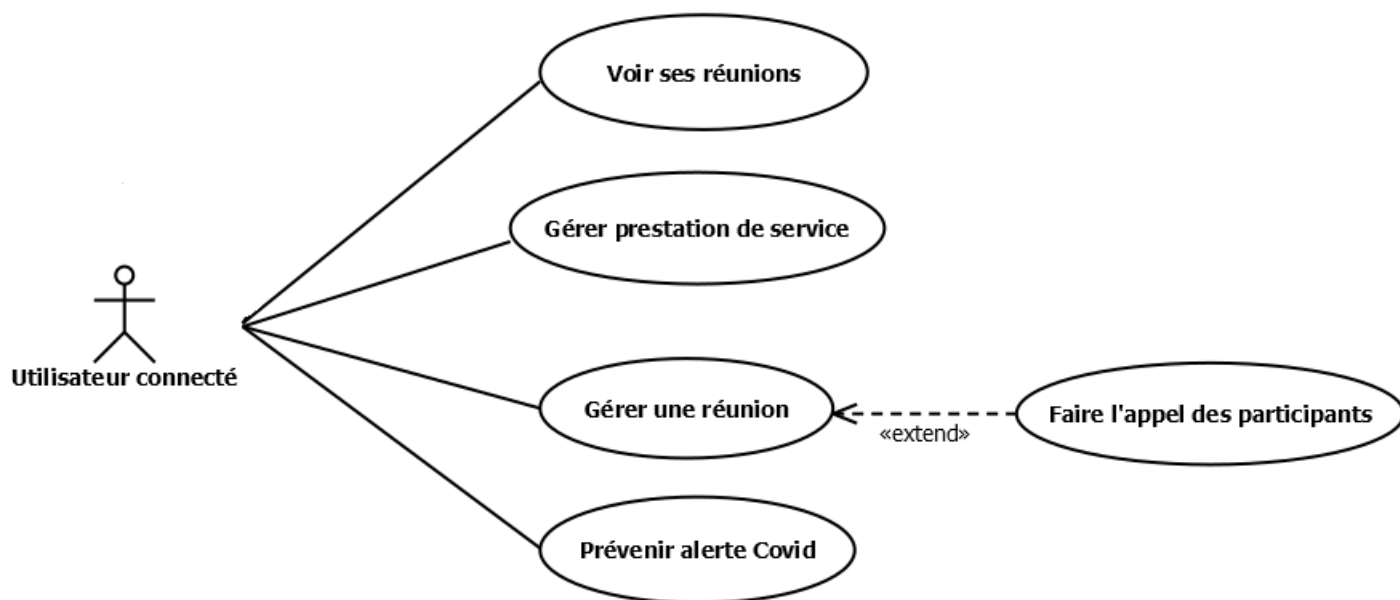
Ce test démontre que la fonctionnalité, ainsi que le contrôleur de l'API et la procédure stockée de la base de données qui lui sont liés, ont bien le comportement attendu.

L'interface Homme Machine Mobile (IHM Flutter)

Afin de répondre au besoin exprimé par la M2L, nous avons mis en place une application mobile de commande de services, disponible au cours d'une réunion, ainsi qu'un système permettant de valider la présence des participants ; ces derniers pourront également se signaler COVID positif, en cas de besoin.

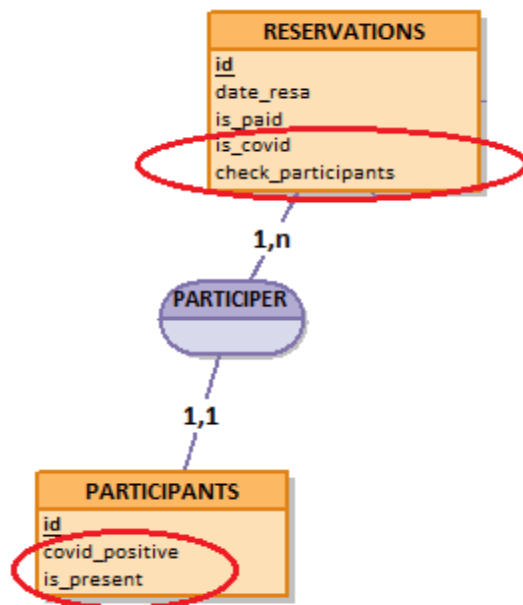


La solution proposée est en mesure de répondre aux besoins évalués dans le diagramme des cas d'utilisations :



Rappel de l'API

Notre application mobile utilise la même API que notre [webApp React](#) (réalisation 1). Les fonctionnalités présentes sur le mobile, nécessitent donc l'adaptation de notre base de données ainsi que de l'API.



Ajout des colonnes "is_covid" et "check_participants" qui sont des booléens (vrai/faux) dans la table RÉSERVATIONS.

Si "**is_covid**" est à VRAI, il y a un participant qui est covid positif lors de cette réunion.

Si "**check_participants**" est à VRAI, permet de s'assurer que l'appel a bien été fait lors de la réunion.

Ajout des colonnes "covid_positive" et "is_present" qui sont des booléens (vrai/faux) dans la table PARTICIPANTS.

Si "**covid_positive**" est à VRAI, le participant est covid positif.

Si "**is_present**" est à VRAI, le participant est présent à la réunion.

Pour accéder à notre base de données nous avons dû mettre en place de nouvelles procédures stockées, de nouvelles routes et de nouveaux contrôleurs dans notre API.

Nous avons créé une nouvelle route destinée uniquement à notre application mobile :

http://localhost:3001/flutter

En découle toutes les routes suivantes :

```
router
  .route('/login')
  .post(getAccountIdByEmail);
```

→ URL pour le login de l'utilisateur (POST)

```
router
  .route('/:user_id/reservations')
  .get(getFutureReservation)
  .put(updateCheckResa);
```

→ URL pour récupérer la liste des réservations d'un utilisateur (GET)

→ URL pour mettre à jour le booléen "check_participants" (PUT)

```
router
  .route('/:user_id/reservations/history')
  .get(getBeforeReservation);
```

→ URL pour récupérer les réservations passées (GET)

```
router
  .route('/:user_id/participations/history')
  .get(getUserParticipationBefore)
```

→ URL pour récupérer l'historique des réservations de l'utilisateur connecté (GET)

→ URL pour récupérer la liste des produits disponible (GET)

```
router
  .route('/produits')
  .get(getAllProducts);
```

→ URL pour récupérer les participants à une réservations (GET)

→ URL pour mettre à jour le booléen "is_present" si un participant est présent à la réunion.

```
router
  .route('/:id_resa/participants')
  .get(getResaParticipants)
  .put(updatePresentParticipations);
```

Des tests unitaires et d'intégrations ont été effectués lors de la création de ces routes sur notre API. Nous

avons vérifié que les données transitaient entre la base de données et l'API, et nous nous sommes assurés que l'application mobile les recevait bien. L'exemple de test appliqué à nos applications web et mobile a été montré plus tôt dans ce document.

Pour rappel, les étapes d'installation de notre API sont disponibles [ici](#) ou dans le [Readme.md](#) du dépôt distant Github.

Dépendances à installer

Afin de pouvoir fonctionner sur votre machine, notre application Flutter nécessite l'installation de :

- Git, télécharger et installer [Git](#)
- VS Code, télécharger et installer [VSCode](#)
- Suivez [les étapes d'installation](#) pour mettre en place votre environnement Flutter.

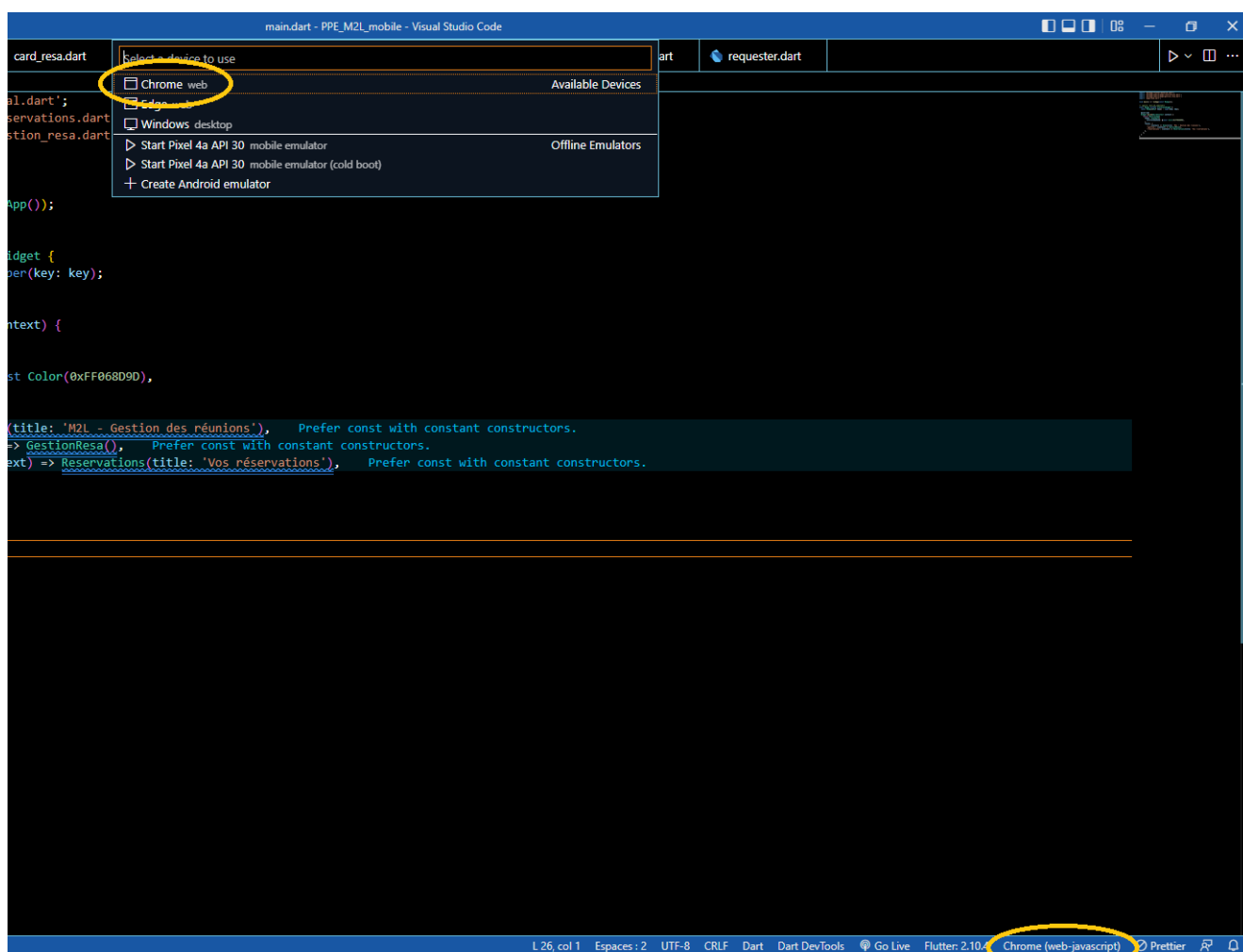
Mise en fonctionnement

Il vous est possible de télécharger l'API directement à cette [adresse](#) ou bien de le cloner via ligne de commande.

```
git clone https://github.com/BSevault/PPE_M2L_mobile.git
```

Une fois les étapes d'installation effectuées, ouvrez le dossier ./PPE_M2L_mobile/ dans VS Code.

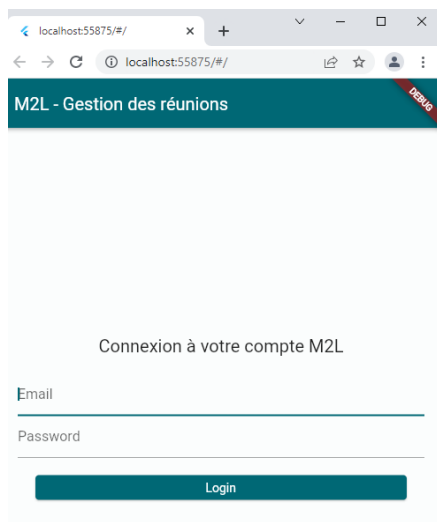
Sélectionnez l'émulateur Chrome comme sur la capture d'écran ci-dessous.



Pour lancer l'application ouvrez le fichier ./PPE_M2L_mobile/lib/main.dart dans VS Code et exécuter sans débogage (CTRL + F5).

L'IDE va alors construire l'application et lancer Chrome.

Voici ce que vous devriez avoir :



Pour que l'application mobile puisse accéder à notre backend pour faire ses requêtes, il faut définir cette adresse dans le fichier `./lib/utils/requester.dart`

```
abstract class Requester {  
  // ip aws  
  // static const String _base_url = 'http://15.237.109.149:3001';  
  
  // ip localhost vm android  
  // static const String _base_url = 'http://10.0.2.2:3001';  
  
  // ip VM Gefor PC1  
  // static const String _base_url = 'http://192.168.0.49:3001';  
  
  // ip chrome  
  static const String base_url = "http://localhost:3001"; Prefer
```

Choisir comme ci-dessus l'adresse "<http://localhost:3001>". Le port sera en fonction du port défini dans votre backend, dans le fichier `prod.env` comme expliqué dans le [Readme.md](#) du dépôt distant Github.

C'est quoi Flutter ?

Le cadre de développement Flutter est basé sur le langage Dart qui est un langage de programmation orienté objet (POO). L'un des gros avantages de Flutter c'est que ce langage permet le développement multi-plateforme (aussi bien IOS qu'Android) sans avoir besoin de coder à nouveau toute l'application. De plus, Flutter fournit tout un [catalogue](#) de Widgets.

Mais qu'est-ce qu'un widget ?

Un Widget c'est tout ce qui compose votre application que ce soit un texte, une image, un bouton, l'ensemble d'une page ou bien les opérations qui permettent de donner un résultat.

Ce bouton est un Widget

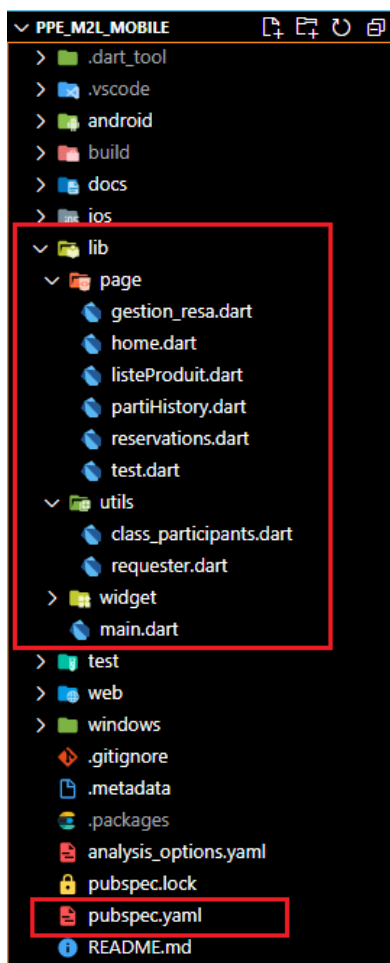


Comme React.js, Flutter permet de gérer l'état des Widgets et va les reconstruire si leur état change. Par exemple, si je veux afficher la météo maintenant et que nous sommes le matin, la page se charge et j'obtiens le résultat du matin. Mais si maintenant je veux savoir la météo de cet après-midi, je clique sur après-midi et j'obtiens le temps de cet après-midi. Il y a donc un changement d'état de la part de mon widget.

Librairies utilisées

- [material.dart](#) : Permet d'utiliser les Widgets à destination des téléphones Android.
- [http.dart](#) : Permet de faire des requêtes HTTP vers notre API pour transmettre et récupérer des données.
- [dart:convert](#) : est utilisé pour encoder ou décoder des données en format JSON.
- [date_symbol_data_local.dart](#) : Pour la conversion de date en format local.
- [intl.dart](#) : Pour la manipulation de date.

Architecture



→ Les fichiers de développement se situent dans le dossier lib/

A l'intérieur de celui-ci il y a 3 dossiers :

- **page** : correspond aux pages de notre application mobile.
- **utils** : contient les classes utilitaires qui ne sont pas des widgets, mais nécessaires pour répondre aux attentes de la M2L.
- **widget** : contient les widgets que nous avons créés. Permet la réutilisation de ces derniers dans d'autres pages de l'application et offre également une meilleure lisibilité du code.

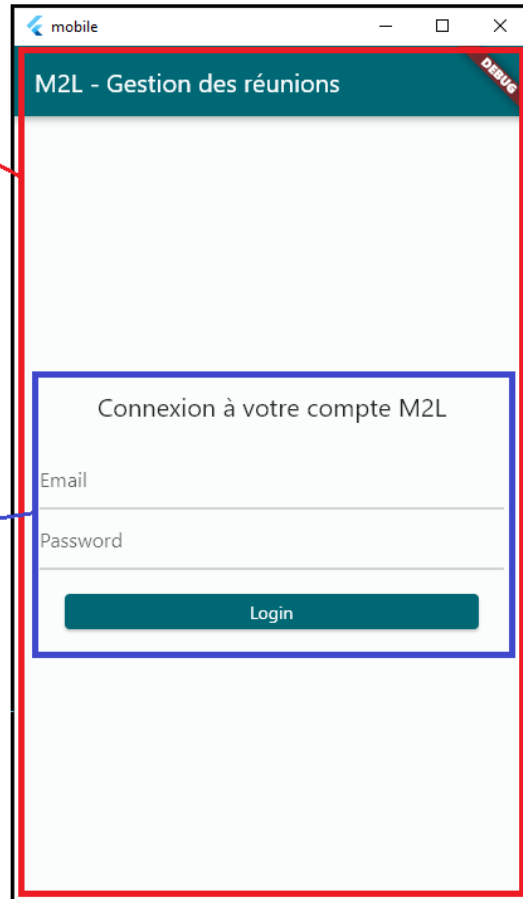
Enfin à la racine du dossier lib/ se trouve le fichier **main.dart** qui est le point d'entrée de notre application mobile. Il peut s'apparenter à l'index.html d'un site web. C'est à partir de lui que va être construite l'application.

→ Le fichier pubspec.yaml répertorie toutes les dépendances de notre application mobile.

Exemple de notre page d'accueil sur l'application mobile M2L :

Page d'accueil de l'application
Fichier "main.dart" fait appelle
au widget ./lib/page/home.dart

Widget "form_login.dart"
Dossier ./lib/widget



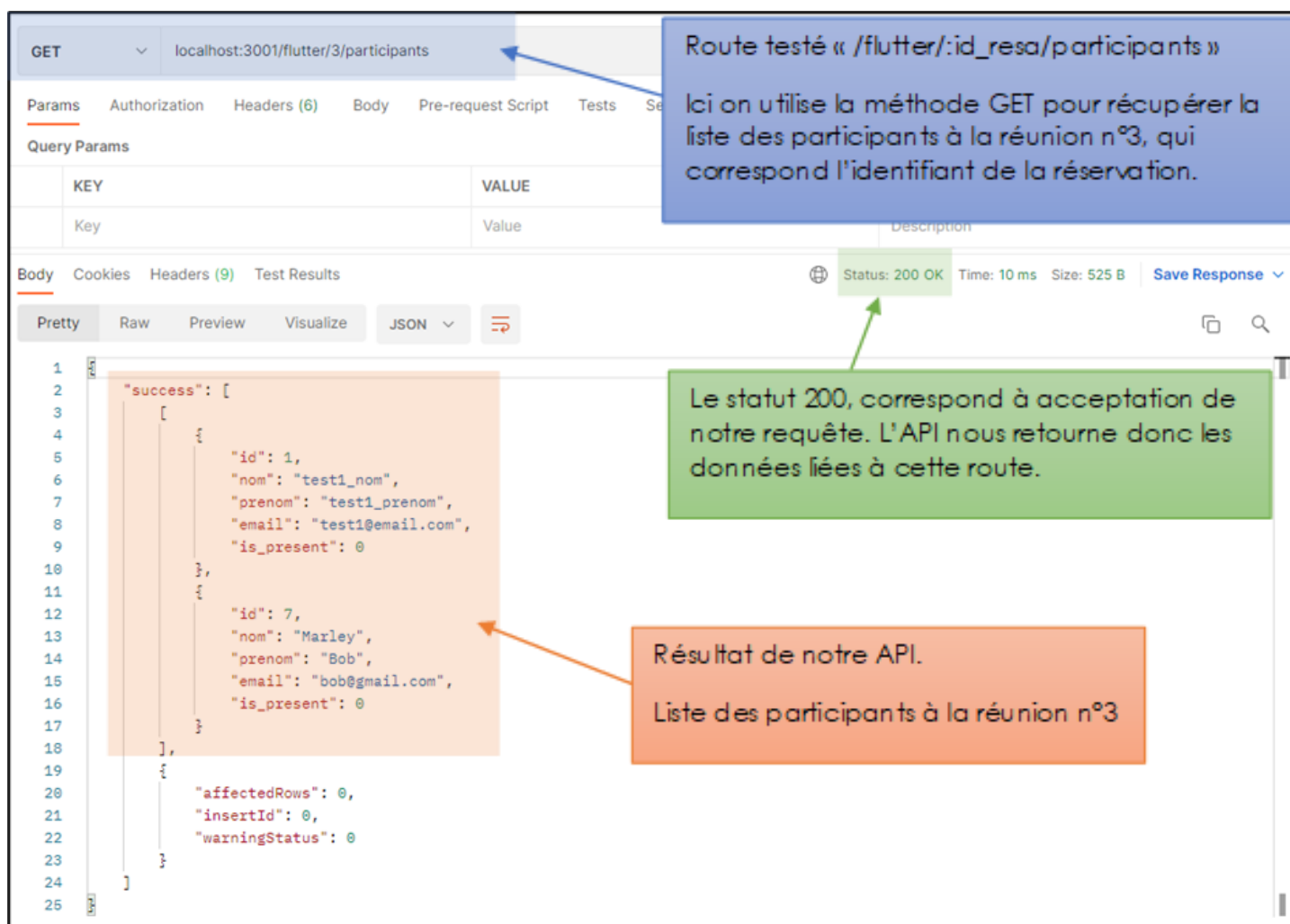
Tests

Nous avons effectué des tests unitaires et d'intégration lors du développement de l'application mobile. L'objectif de ces tests est de s'assurer de l'accessibilité des données, que les données récupérées sont bien celles voulues et qu'elles sont complètes, vérification du bon transit entre notre frontend et notre API.

Ces tests se sont déroulés dans l'ordre suivant :

- Tests via [Postman](#)
- Tests via [DBeaiver](#)
- Tests via l'[IHM Flutter](#)

Exemple d'un test avec la route `/flutter/:id_resa/participants` via Postman :



Route testé « `/flutter/:id_resa/participants` »

Ici on utilise la méthode GET pour récupérer la liste des participants à la réunion n°3, qui correspond l'identifiant de la réservation.

Status: 200 OK Time: 10 ms Size: 525 B Save Response

Le statut 200, correspond à acceptation de notre requête. L'API nous retourne donc les données liées à cette route.

Résultat de notre API.
Liste des participants à la réunion n°3

```

1  {
2    "success": [
3      {
4        "id": 1,
5        "nom": "test1_nom",
6        "prenom": "test1_prenom",
7        "email": "test1@email.com",
8        "is_present": 0
9      },
10     {
11       "id": 7,
12       "nom": "Marley",
13       "prenom": "Bob",
14       "email": "bob@gmail.com",
15       "is_present": 0
16     }
17   ],
18   "affectedRows": 0,
19   "insertId": 0,
20   "warningStatus": 0
21 }
22
23
24
25

```

Les routes liées à notre application mobile sont toutes préfixées avec « `/flutter` »

```

router
  .route('/:id_resa/participants')
  .get(getResaParticipants)
  .put(updatePresentParticipations);

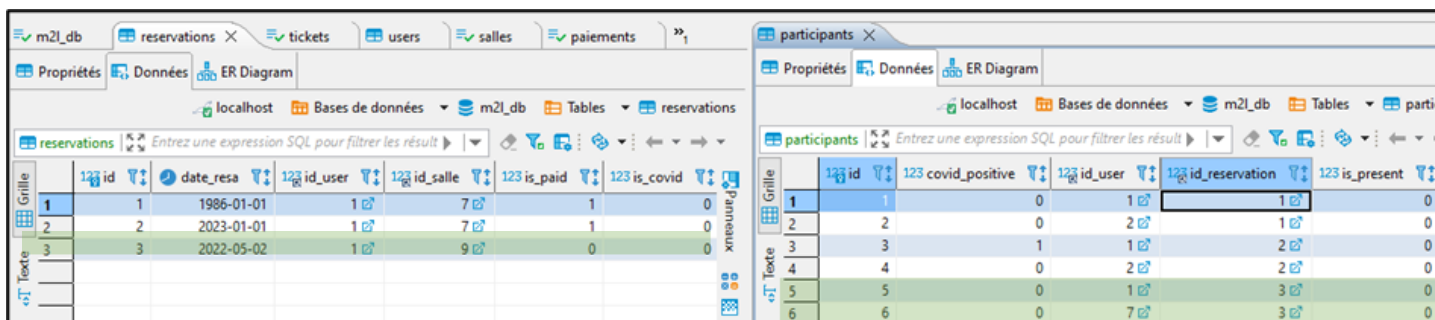
```

La route testée ci-dessus utilise la méthode GET et appelle donc le contrôleur « `getResaParticipants` ».

Ce contrôleur permet de faire un appel à la base de données via la procédure stockée "getResaParticipants(id_resa)" qui liste tous les participants invités à une réservation (id_resa).

```
getResaParticipants: async (req, res) => {
  const { id_resa } = req.params;
  await call(res, async (connexion) => {
    const result = await connexion.query("CALL getResaParticipants(?)", [id_resa] )
    return res.status(200).json({ success: result });
  });
},
```

Nous vérifions les données recueillies via Postman sur l'interface DBeaver.

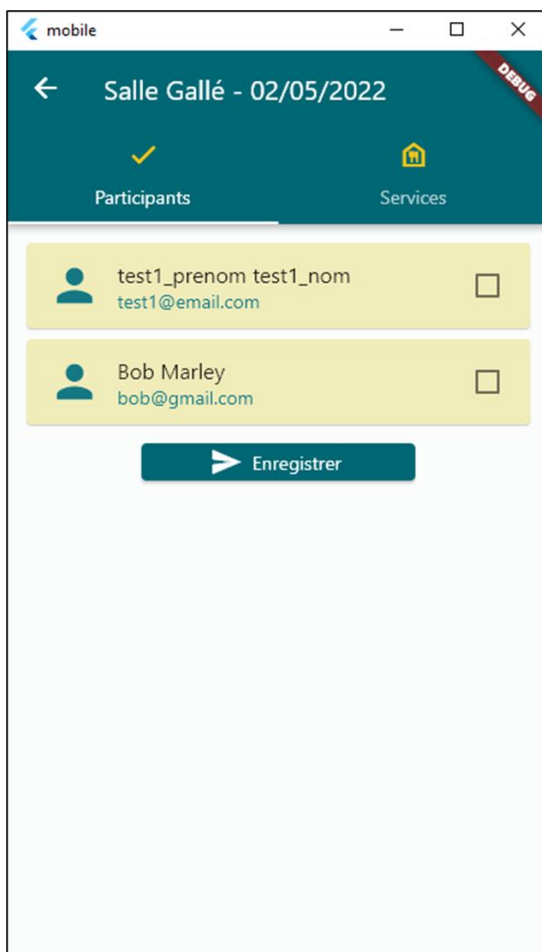


The screenshot shows two database tables in DBeaver. The left table is 'reservations' and the right table is 'participants'.

id	date_resa	id_user	id_salle	is_paid	is_covid
1	1986-01-01	1	7	1	0
2	2023-01-01	1	7	1	0
3	2022-05-02	1	9	0	0

id	covid_positive	id_user	id_reservation	is_present
1	0	1	1	0
2	0	2	1	0
3	1	1	2	0
4	0	2	2	0
5	0	1	3	0
6	0	7	3	0

Nous nous apercevons que les données sont correctes, il y a bien 2 participants à la réunion n°3 et ils correspondent bien aux utilisateurs ayant les identifiants respectifs n°1 et 7.



Notre API est bien commune aux 2 réalisations, puisque nous avons réservé notre salle et ajouté un participant via la WebApp React.js et nous avons récupéré la liste des participants pour s'assurer de leur participation à la réunion via l'application Flutter.

Nous pouvons nous apercevoir que les données sont bien récupérées dans l'application mobile, puisque nous voyons nos 2 participants à la réunion n°3 du 02 mai. Ces données correspondent bien aux données testées via Postman et via DBeaver.

Enfin, lors de la création des pages et des widgets sous Flutter, nous avons tout d'abord testé la bonne conduite de ces derniers. Une fois les tests unitaires réussis, nous intégrons les fonctionnalités développées et vérifions qu'aucun bug n'est dû à l'intégration de la nouvelle page ou du widget (test d'intégration).