



DOSSIER PROFESSIONNEL (DP)

Nom de naissance ➤ DELLACASE
Nom d'usage ➤ DELLACASE
Prénom ➤ Thomas
Adresse ➤ 23 rue Wulfram Puget 13008 Marseille

Titre professionnel visé

Concepteur développeur d'applications

MODALITÉ D'ACCÈS :

- ☒ Parcours de formation
☐ Validation des Acquis de l'Expérience (VAE)

Présentation du dossier

Le dossier professionnel (DP) constitue un élément du système de validation du titre professionnel.
Ce titre est délivré par le Ministère chargé de l'emploi.

Le DP appartient au candidat. Il le conserve, l'actualise durant son parcours et le présente
obligatoirement à chaque session d'examen.

Pour rédiger le DP, le candidat peut être aidé par un formateur ou par un accompagnateur VAE.
Il est consulté par le jury au moment de la session d'examen.

Pour prendre sa décision, le jury dispose :

1. des résultats de la mise en situation professionnelle complétés, éventuellement, du questionnaire professionnel ou de l'entretien professionnel ou de l'entretien technique ou du questionnement à partir de productions.
2. du **Dossier Professionnel (DP)** dans lequel le candidat a consigné les preuves de sa pratique professionnelle.
3. des résultats des évaluations passées en cours de formation lorsque le candidat évalué est issu d'un parcours de formation
4. de l'entretien final (dans le cadre de la session titre).

*[Arrêté du 22 décembre 2015, relatif aux conditions de délivrance des titres professionnels
du ministère chargé de l'Emploi]*

Ce dossier comporte :

- pour chaque activité-type du titre visé, un à trois exemples de pratique professionnelle ;
- un tableau à renseigner si le candidat souhaite porter à la connaissance du jury la détention d'un titre, d'un diplôme, d'un certificat de qualification professionnelle (CQP) ou des attestations de formation ;
- une déclaration sur l'honneur à compléter et à signer ;
- des documents illustrant la pratique professionnelle du candidat (facultatif)
- des annexes, si nécessaire.

DOSSIER PROFESSIONNEL ^(DP)

Pour compléter ce dossier, le candidat dispose d'un site web en accès libre sur le site.

 <http://travail-emploi.gouv.fr/titres-professionnels>

Sommaire

Exemples de pratique professionnelle

Concevoir et développer des composants d'interface utilisateur en intégrant les recommandations de sécurité	p.	5
- Maquetter une application	p. p.	
- Développer une interface utilisateur de type desktop	p. p.	
- Développer des composants d'accès aux données	p p.	
- Développer la partie front-end d'une interface utilisateur web	p p.	
- Développer la partie back-end d'une interface utilisateur web	p p.	
Concevoir et développer la persistance des données en intégrant les recommandations de sécurité	p.	
- Concevoir une base de données	p. p.	
- Mettre en place une base de données	p. p.	
- Développer des composants dans le langage d'une base de données	p p.	
Concevoir et développer une application multicouche répartie en intégrant les recommandations de sécurité	p.	
- Concevoir et développer une application multicouche répartie en intégrant les recommandations de sécurité	p. p.	
- Concevoir une application	p. p.	
- Développer des composants métier	p p.	
- Construire une application organisée en couches	p p.	
- Développer une application mobile	p p.	
- Préparer et exécuter les plans de tests d'une application	p p.	
- Préparer et exécuter le déploiement d'une application	p p.	

DOSSIER PROFESSIONNEL ^(DP)

Titres, diplômes, CQP, attestations de formation *(facultatif)*

Déclaration sur l'honneur

Documents illustrant la pratique professionnelle *(facultatif)*

Annexes *(Si le RC le prévoit)*

p.

p.

p.

p.

EXEMPLES DE PRATIQUE PROFESSIONNELLE

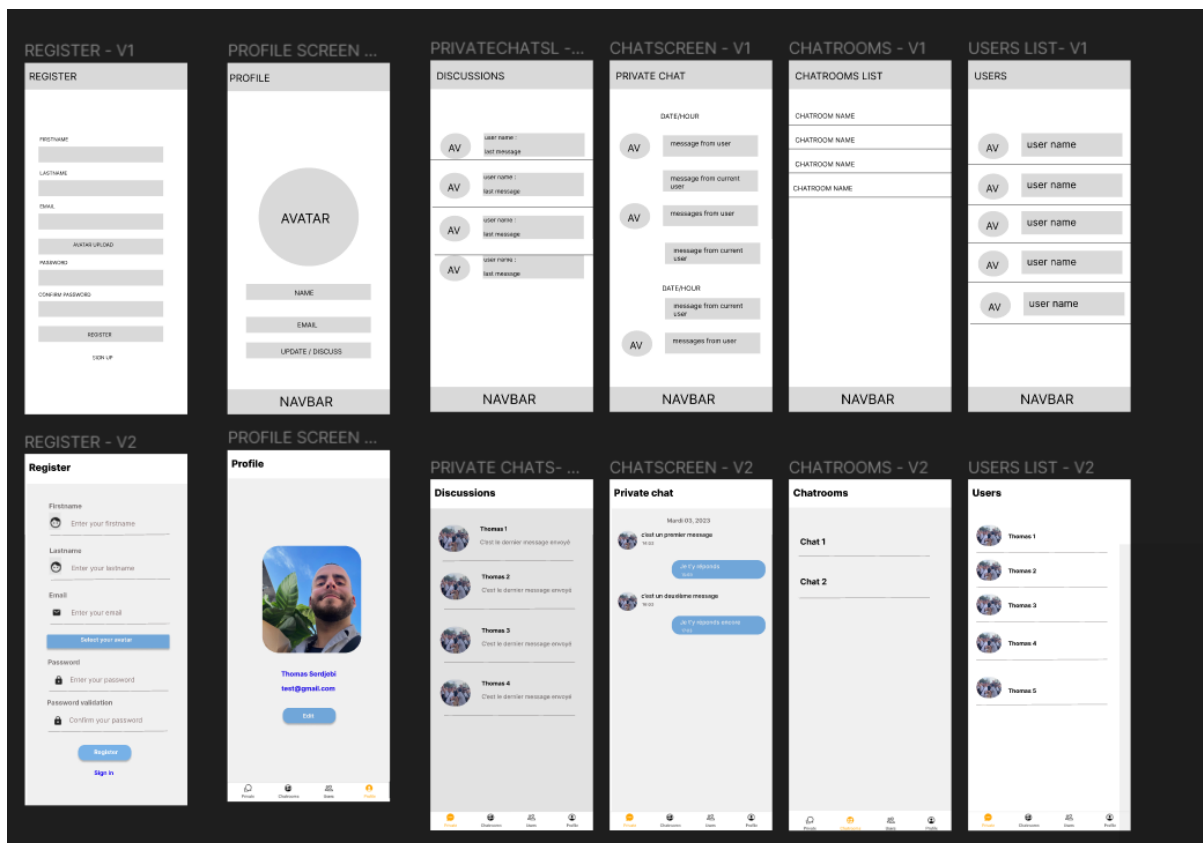
Activité-type 1

Concevoir et développer des composants d'interface utilisateur en intégrant les recommandations de sécurité.

Exemple n°1 - Maquetter une application

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

Le maquettage de l'application a été effectué avec l'outil **Figma**. Nous avons d'abord établi un **wireframe basse fidélité** avec seulement les blocs clés des différents écrans. Par la suite, nous avons établi un wireframe détaillé de chaque écran : style des formulaires, style de la topbar, couleurs et taille des différentes typographies, barre de navigation en bas de l'écran, forme et affichage des avatars ainsi que des messages.



2. Précisez les moyens utilisés :

Figma

3. Avec qui avez-vous travaillé ?

Maxime Hadj , Thomas Serdjebi et moi meme.

DOSSIER PROFESSIONNEL ^(DP)

4. Contexte

Nom de l'entreprise, organisme ou association ▶ *La plateforme*

Chantier, atelier, service ▶ *Projet ChatApp dans le cadre de la formation*

Période d'exercice ▶ Du : *01/11/22* au : *01/04/2023*

5. Informations complémentaires (facultatif)

Exemple n° 2 ▶ **Développer une interface utilisateur de type desktop**

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

2. Précisez les moyens utilisés :

3. Avec qui avez-vous travaillé ?

4. Contexte

Nom de l'entreprise, organisme ou association ▶ *Cliquez ici pour taper du texte.*

Chantier, atelier, service ▶ *Cliquez ici pour taper du texte.*

Période d'exercice ▶ Du : *Cliquez ici* au : *Cliquez ici*

5. Informations complémentaires (facultatif)

Exemple n° 3 - Développer des composants d'accès aux données

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

Le fichier **database.js** permet l'**accès à la base de données**. En effet, il utilise la fonction **createPool** du module **mysql2** afin de définir la **constante pool** qui permettra l'accès à la base de données. Les informations sont stockées dans **un fichier .env** qui n'est pas publié sur github lors du push des branches. Enfin la dernière ligne du fichier permet d'**exporter la constante** afin de l'utiliser dans les **fichiers Services** qui contiennent les **requêtes SQL**.

```
1  const { createPool } = require('mysql2');
2
3  const pool = createPool ({
4    port: process.env.DB_PORT,
5    host: process.env.DB_HOST,
6    user: process.env.DB_USER,
7    password: process.env.DB_PASS,
8    database: process.env.MYSQL_DB
9  });
10
11 });
12
13 module.exports = pool ;
```

Les fichiers **.services** ont tous une **structure identique**. En effet, on en trouve un pour les **messages**, un pour les **utilisateurs** et un pour les **messages**. Chacun de ces fichiers va fournir différentes **fonctionnalités**.

En premier lieu, on appelle la **constante pool** contenue dans le fichier **database.js** qui permettra l'accès à la base de données. Ensuite le **module.exports** va permettre d'exporter toutes ces **fonctionnalités** qui seront appelées dans les **fichiers .controller**.

Ensuite, à l'intérieur des **accolades du modules.exports**, on définit donc **chaque fonctionnalité** de la façon suivante :

- **nom de la fonction et ses paramètres** (dans l'exemple ci-dessous data et callBack)
- création de **variables supplémentaires** nécessaires à l'exécution de la requête
- la **méthode pool.query** contenant la **requête SQL** liée à la **fonctionnalité** désirée, qui prend plusieurs paramètres :
 - la **requête SQL**
 - les paramètres qui vont permettre de **remplacer les valeurs dans la requête SQL** et sont extraits initialement du **paramètre data** ou des **autres variables supplémentaires** créées comme la date de création
 - la **fonction de rappel dit callBack** est utilisée pour retourner le résultat ou **gérer les erreurs**. On constate que s'il n'y a pas d'erreurs, seul le résultat est retourné, s'il y a une erreur, l'erreur est retournée.

DOSSIER PROFESSIONNEL (DP)

```
1 const pool = require ("../../config/database");
2
3 module.exports = {
4
5   //Create a user
6   create: (data, callback) => {
7     const now = new Date(Date.now());
8     const created = now.toISOString().slice(0, 19).replace('T', ' ');
9     const defaultRole = 1;
10
11     pool.query(
12       'insert into users (firstname, lastname, email, password, role_id, created_at, updated_at, avatar) values (?, ?, ?, ?, ?, ?, ?, ?)',
13       [
14         data.firstname,
15         data.lastname,
16         data.email,
17         data.password,
18         defaultRole,
19         created,
20         data.updated_at,
21         data.imageUrl
22       ],
23       (error, results, fields) => {
24         if (error) {
25           console.log(error)
26           return callback(error)
27         }
28         return callback(null, results)
29       }
30     );
31   },
32 }
```

2. Précisez les moyens utilisés :

Pour développer les composants d'accès aux données nous avons utilisé les bibliothèques suivantes :
mysql and mysql2
expo
socket.io and socket.io-client

3. Avec qui avez-vous travaillé ?

Maxime Hadj , Thomas Serdjebi et moi même.

4. Contexte

Nom de l'entreprise, organisme ou association ▶ *La plateforme*

Chantier, atelier, service ▶ *Projet ChatApp dans le cadre de la formation*

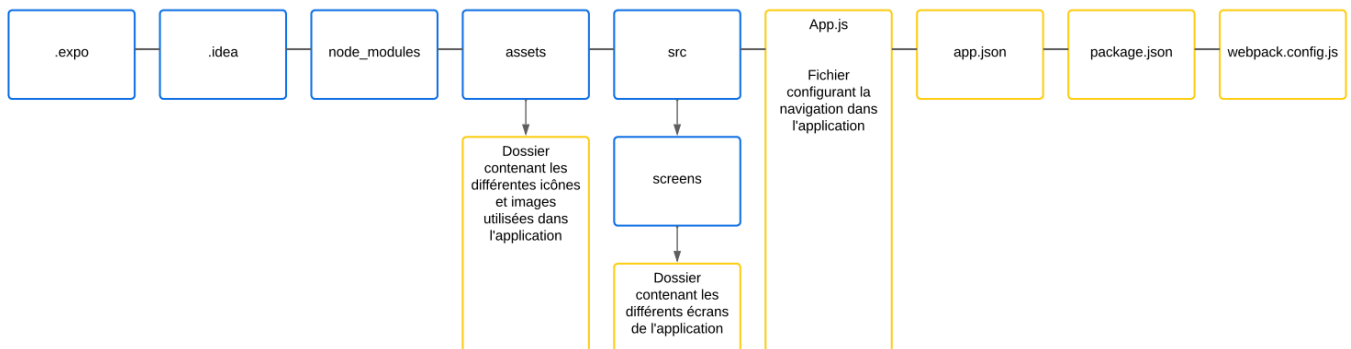
Période d'exercice ▶ Du : *01/11/22* au : *01/04/2023*

5. Informations complémentaires (facultatif)

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

Vous pourrez trouver un **schéma de l'arborescence de l'application** mobile ce dessous. Hormi les fichiers générés par l'installation de l'environnement et des différentes librairies, on peut distinguer les dossiers et fichiers principaux :

- un **fichier App.js** qui vient configurer la **navigation** dans l'application
- un **sous dossier src/screens** qui contient l'ensemble des **écrans de l'application**
- un **sous dossier assets** qui contient les images et icônes présentes dans l'application



Nous avons développé cette application dans un **environnement Node.js** et avons également installé **React Native**, le **framework Javascript** utilisé pour développer des applications multiplateformes ainsi qu'**Expo**, qui est un ensemble d'outils qui facilitent le développement d'applications React Native.

La navigation est gérée dans le **fichier App.js**. Elle nécessite tout d'abord d'**importer les différentes fonctions** qui viennent générer les écrans de l'application.

```
1 import React from 'react'
2 import 'react-native-gesture-handler';
3 import ChatroomScreen from './src/screens/ChatroomScreen';
4 import ChatroomsListScreen from './src/screens/ChatroomsListScreen';
5 import PrivateChatsListScreen from './src/screens/PrivateChatsListScreen';
6 import LoginScreen from './src/screens/LoginScreen';
7 import ProfileScreen from './src/screens/ProfileScreen';
8 import RegisterScreen from './src/screens/RegisterScreen';
9 import UsersListScreen from './src/screens/UsersListScreen';
10 import PrivateChatScreen from './src/screens/PrivateChatScreen';
11 import UpdateProfileScreen from './src/screens/UpdateProfileScreen';
12 import { Icons } from '@expo/vector-icons';
13
```

Ensuite nous importons les **composants de navigation** fournis par les **bibliothèques de React Native** :

```
//Navigations components
import {NavigationContainer} from '@react-navigation/native'
import {createStackNavigator} from '@react-navigation/stack'
import {createBottomTabNavigator} from '@react-navigation/bottom-tabs'
```

- **NavigationContainer**, qui va être le **contenant racine de la navigation** et va la gérer dans sa **globalité** en fournissant le **contexte de navigation** à tous les composants.
- **createStackNavigator**, qui va permettre de **générer des “piles” de navigation**, en empilant successivement plusieurs écrans les uns avec les autres, en suivant la trame logique du parcours utilisateur.
- **createBottomTabNavigator**, qui va permettre de **générer une barre de navigation** située en bas de l'écran, et qui dans notre cas, nous donnera **accès alors pour chaque bouton** à un **navigateur empilé (stack navigator) différent**.

Enfin nous utilisons **la fonction createStackNavigator** que nousinstancions à des constantes pour créer nos différents stack navigators ainsi que createBottomTabNavigator pour créer la barre de navigation. Nous allons maintenant décortiquer la structure de la navigation en elle-même.

```
22  const AuthStack = createStackNavigator();
23  const PrivateStack = createStackNavigator();
24  const ChatroomStack = createStackNavigator();
25  const ProfileStack = createStackNavigator();
26  const UsersStack = createStackNavigator();
27
28  const AppStack = createBottomTabNavigator();
29
```

Comme expliqué précédemment, les stacks navigators permettent de créer un empilement dans lequel on pourra naviguer d'un écran à l'autre. Voici quelques exemples :

```
//Stack for the sign in/ signup
function AuthStackScreen() {
  return (
    <AuthStack.Navigator>
      <AuthStack.Screen name="Login" component={LoginScreen} />
      <AuthStack.Screen name="Register" component={RegisterScreen} />
    </AuthStack.Navigator>
  );
}
```

Dans ce premier exemple, on **génère un stack navigator** pour l'**authentification**. La **constante AuthStack** a été créée plus haut. On indique donc que ce stack navigator va **contenir les deux écrans LoginScreen et RegisterScreen** dont les noms à l'affichage seront respectivement **Login** et **Register**. Ainsi dans ce navigateur on peut **passer de l'écran de connexion à l'écran d'inscription** et revenir en arrière pour basculer sur l'écran de connexion.

```
//Stack for the PrivateChats Screen => Private chats list, private chat room, user profile
function PrivateStackScreen() {
  return(
    <PrivateStack.Navigator>
      <PrivateStack.Screen name="Discussions" component={PrivateChatsListScreen} options={{headerLeft: null}} />
      <PrivateStack.Screen name="Private Chat" component={PrivateChatScreen} options={{}} />
      <PrivateStack.Screen name="Profile" component={ProfileScreen} options={{}} />
    </PrivateStack.Navigator>
  )
}
```

La barre de navigation est générée par la **fonction AppStackScreen**. On y définit d'abord la **constante screenOptions** qui va permettre de gérer les options d'affichage de la barre en fonction du nom de la page où on se situe (icône, couleur en fonction du focus, taille, couleur de fond....) .

```
function AppStackScreen() {
  const screenOptions = ({ route }) => ({
    tabBarIcon: ({ color, size, focused }) => {
      let iconName;

      switch (route.name) {
        case "Private":
          iconName = focused ? "chatbubble-ellipses" : "chatbubbles-outline";
          break;
        case "Chatrooms":
          iconName = focused ? "people-circle" : "people-circle-outline";
          break;
        case "Users":
          iconName = focused ? "people" : "people-outline";
          break;
        case "Profile":
          iconName = focused ? "person-circle" : "person-circle-outline";
          break;
        default:
          iconName = "";
      }

      return <Ionicons name={iconName} size={size} color={color} />;
    },
    tabBarActiveTintColor: "orange",
    tabBarInactiveTintColor: "black",
    tabBarLabelStyle: {
      fontSize: 12,
    },
    tabBarStyle: {
      backgroundColor: "white",
    },
  });
}
```

Enfin, la **fonction AppStackScreen** retourne donc les **composants de la barre de navigation**. On y injecte les différents **stack navigators** générés précédemment. Lorsque l'utilisateur **clique** sur une **icône**, il aura donc **accès aux écrans** en fonction du stack navigator associé.

```
return (
  <AppStack.Navigator screenOptions={screenOptions}>
    <AppStack.Screen name="Private" component={PrivateStackScreen} options={{ headerShown: false }} />
    <AppStack.Screen name="Chatrooms" component={ChatroomStackScreen} options={{ headerShown: false }} />
    <AppStack.Screen name="Users" component={UsersStackScreen} options={{ headerShown: false }} />
    <AppStack.Screen name="Profile" component={ProfileStackScreen} options={{ headerShown: false }} />
  </AppStack.Navigator>
);
```

Finalement, la **fonction principale App** de l'écran vient **centraliser la navigation** dans le **NavigationContainer**. Dans le navigateur principal, on accède tout d'abord au **stack navigateur Auth pour la connexion/inscription**.

```
const Stack = createStackNavigator();

export default function App() {

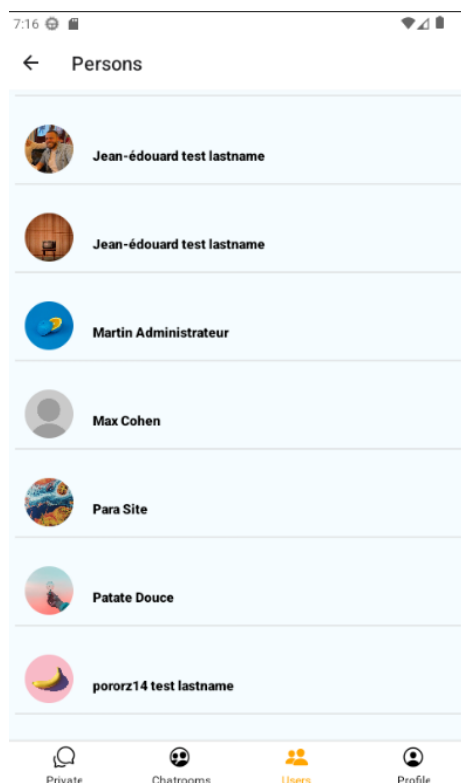
  return (
    <NavigationContainer>
      <Stack.Navigator initialRouteName="Auth">
        <Stack.Screen name="Auth" component={AuthStackScreen} options={{headerShown: false}} />
        <Stack.Screen name="App" component={AppStackScreen} options={{headerShown: false, headerLeft:null}} />
      </Stack.Navigator>
    </NavigationContainer>
  );
}
```

Une fois l'**utilisateur connecté**, il sera redirigé vers le **stack App**, avec donc la **barre de navigation**. La redirection après connexion redirige l'utilisateur par défaut vers la liste de ses discussions privées.

```
props.navigation.navigate('App',{
  screen: 'Private',
  headerLeft:null
})
```

Les "**screens**" (écrans) sont des **composants** de l'application. Dans notre projet, ils sont générés par une **fonction fléchée** qui prend en **paramètres les "props"**, des données transmises de composants parents à composants enfants, et **retourne un élément React**.

Prenons l'exemple de la **structure du fichier UsersListScreen.js**, retournant l'ensemble des utilisateurs de l'application dans l'écran comme sur la capture d'écran ci à gauche.



La constante **UsersListScreen** vient retourner un **ensemble de composants imbriqués** les uns dans les autres :
un **composant View** auquel on applique le **style** défini auparavant et qui contient tous les autres

le **composant FlatList** qui va **générer la liste des utilisateurs** et qui prend plusieurs **paramètres** comme le **paramètre data**, auquel on affecte le **tableau users**, le **renderItem** qui va définir pour **chaque users** des **paramètres d'affichage** de la liste, et le **keyExtractor** qui retourne pour chaque élément de la liste une valeur unique qui représente la clé de l'élément.

```
return (  
  <View style={styles.container}>  
    <FlatList  
      data={users}  
      showsVerticalScrollIndicator={false}  
      showsHorizontalScrollIndicator={false}  
      renderItem={({ item }) => (  
        <TouchableOpacity  
          style={styles.userContainer}  
          onPress={() => {  
            props.navigation.navigate('Profile', {  
              id_user: item.id_user,  
            });  
          }}>  
          <Image  
            style={styles.avatar}  
            source={{uri: item.avatar}}  
          />  
          <Text  
            style={styles.userName}  
          >  
            {item.firstname} {item.lastname}  
          </Text>  
        </TouchableOpacity>  
      )}  
      keyExtractor={item => item.id_user.toString()}  
    />  
  </View>  
);
```

Le style est généré dans la **constante styles** qui vient utiliser la **méthode create** du **module StyleSheet** de **React Native**. On y définit le **style des différents éléments** qui vont être retournés : le conteneur principal, le conteneur d'un utilisateur, le style du nom ainsi que celui de l'avatar.

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#F5FCFF',
  },
  userContainer: {
    flex: 1,
    flexDirection: 'row',
    margin: 10,
    padding: 10,
    borderBottomWidth: 1,
    borderColor: '#d3d3d3',
  },
  userName: {
    fontSize: 14,
    marginLeft: 20,
    padding: 5,
    fontWeight: 'bold'
  },
  avatar: {
    width: 50,
    height: 50,
    borderRadius: 25,
    margin: 5
  },
});
```

GiftedChat est une **bibliothèque React Native** facilitant l'**implémentation de fonctionnalités de chat dans les applications**. Elle fournit des composants préconstruits et personnalisables pour la création d'interfaces utilisateur de chat. Le composant fourni GiftedChat est au cœur de cette bibliothèque.

```
return (
  <GiftedChat
    messages={messages}
    showAvatarForEveryMessage={true}
    onSend={messages => onSend(messages)}
    user={{
      _id: userId,
      name: userName,
      avatar: userAvatar,
    }}
    renderAvatar={props => {
      const id = props.currentMessage.user._id;
      return (
        <TouchableOpacity onPress={() => avatarPressed(id)}>
          <Image
            style={styles.avatar}
            source={{ uri: props.currentMessage.user.avatar }}
          />
        </TouchableOpacity>
      )
    }}
  />
);
```

Extrait fichier

PrivateChatScreen.js de l'application - rendu du composant PrivateChatScreen

Le **composant GiftedChat** prend en paramètres :

- les **messages** initialisées dans la variable grâce au **useState** et au **setMessages**

```
<GiftedChat
  messages={messages}
```

- le **onSend** qui va appeler la fonction d'**insertion des messages** en base de données lorsque l'utilisateur va envoyer un message

```
onSend={messages => onSend(messages)}
```

```
//Call sending messages in db and then messages from db to reload the discussion
const onSend = useCallback((messages = []) => {
  const { _id, createdAt, text, user, } = messages[0]
  sendMessagesInDb(messages[0].text)
}, []);
```

- les **informations** associées à l'utilisateur qui envoie le message dans le paramètre user

```
user={ {
  _id: userId,
  name: userName,
  avatar: userAvatar,
}}
```

- la **méthode renderAvatar** permet de rendre l'**avatar cliquable**, de lui **appliquer du style**. Le fait de rendre l'avatar cliquable permet à l'utilisateur d'aller sur le profil de la personne avec qui il est en discussion en appelant la fonction **avatarPressed** qui prend en **paramètre l'id** de l'**utilisateur visé** via les **props de GiftedChat**.

```
renderAvatar={props => {
  const id = props.currentMessage.user._id;
  return (
    <TouchableOpacity onPress={() => avatarPressed(id)}>
      <Image
        style={styles.avatar}
        source={{ uri: props.currentMessage.user.avatar }}
      />
    </TouchableOpacity>
  )
}}
```

```
const avatarPressed = (id) => {
  props.navigation.navigate('Profile', {
    id_user: id
  });
}
```


DOSSIER PROFESSIONNEL ^(DP)

2. Précisez les moyens utilisés :

expo
@react-navigation/native et @react-navigation/stack, @react-navigation/bottom-tabs,
react-native-gesture-handler, react-native-reanimated, react-native-safe-area-context, et
react-native-screens,
react-native-gifted-chat

3. Avec qui avez-vous travaillé ?

Maxime Hadj , Thomas Serdjebi et moi meme.

4. Contexte

Nom de l'entreprise, organisme ou association ▶ *La plateforme*

Chantier, atelier, service ▶ *Projet ChatApp dans le cadre de la formation*

Période d'exercice ▶ Du : *01/11/22* au : *01/04/2023*

5. Informations complémentaires (facultatif)

Exemple n° 5 - Développer la partie back-end d'une interface utilisateur web

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

Import des modules et composants.

On **importe** d'abord les **modules** et **composants** depuis les **bibliothèques React Native** et d'autres afin de construire cet écran. Le **useState** permet de **définir l'état des variables users** et **loading**. La **FlatList** permettra de générer la **liste des utilisateurs**, l'**image** de charger l'**avatar** des utilisateurs, l'**asyncStorage** permettra de récupérer le **token de l'utilisateur** stocké à sa connexion afin d'exécuter l'**appel à notre API**, et le **jwtDecode** permettra de **décoder le token** afin de récupérer les informations de l'utilisateur.

```
import React, { useState, useEffect } from 'react';
import { View, Text, ScrollView, StyleSheet, StatusBar, TouchableOpacity, } from 'react-native';
import { Image, FlatList } from 'react-native';
import AsyncStorage from '@react-native-async-storage/async-storage';
import jwtDecode from 'jwt-decode';
```

Définition du composant UsersListScreen

Ensuite, nous définissons le **composant UsersListScreen** qui va contenir tous les **éléments** et **fonctions** qui vont **générer** cet **écran**. Les composants screens sont définis comme des fonctions prenant en paramètres les props. On initialise immédiatement la variable **users** qui contient la liste des informations de tous les utilisateurs et la variable **loading** qui permettra d'afficher que la page se charge en cas de lenteur, avec le hook **useState**.

```
const UsersListScreen = (props) =>{
  const [users, setUsers] = useState([]);
  const [loading, setLoading] = useState(true);
```

La **fonction** **getAllUsers** est une **fonction asynchrone** qui vient **appeler l'API** afin de **récupérer** la **liste des utilisateurs** et l'**affecter** à la variable **processedUsers**. Elle est **contenue** dans le **composant UsersListScreen**. Voici sa structure :

Dans un premier temps, nous définissons les **différentes constantes** qui vont nous être utiles. La fonction **getAllUsers** est **asynchrone** puisqu'il sera nécessaire de **récupérer le token** de l'utilisateur grâce à l'**asyncStorage**.

```
const unknownAvatar = 'https://icon-library.com/images/unknown-person-icon/unknown-person-icon-4.jpg'
const userToken = await AsyncStorage.getItem('user_token');
const decodedToken = jwtDecode(userToken)
const idUser = decodedToken.result.id_user
```

La **constante unknownAvatar** permettra d'**afficher une image** représentant un **individu inconnu** si l'utilisateur n'a pas souhaité s'inscrire en enregistrant un avatar. La **const userToken** se voit affecter le **token de l'utilisateur** qui a été stocké grâce à l'**AsyncStorage** sous le nom d'item 'user_token'. La **constante decodedToken** vient **décoder le token** et donc contient le tableau **d'informations de l'utilisateur** actuellement **connecté**.

Appel de l'API

Dans un second temps nous utilisons la **fonction fetch** qui prend en **paramètre** la **route de l'API** que nous désirons utiliser, la méthode '**GET**' car nous souhaitons **récupérer** des **informations** ainsi que les **headers** et notamment l'**Authorization** qui est le **token** enregistré dans la variable **userToken**. La réponse sera ensuite chargée au **format json** et retraitée par la suite.

```
fetch('http://192.168.0.14:3000/api/users', {
  method: 'GET',
  headers: { Authorization: 'Bearer ' + userToken },
})
.then(response => response.json())
.then(response => {
```

Traitement de la réponse

Le **traitement de la réponse** traite la réponse pour lui donner un format spécifique :

- si l'utilisateur courant est présent dans la liste retournée par l'API, on le supprime de cette liste
- si l'utilisateur n'a pas d'avatar, alors il faut remplacer la donnée avatar par la constante unknownAvatar qui permettra d'afficher une icône d'une personne inconnue
- si l'utilisateur a un avatar, il faudra modifier le chemin d'accès afin qu'il soit fonctionnel pour que ce dernier soit affiché

On définit donc la **constante processedUsers** en utilisant la **méthode .map** qui va permettre de **parcourir** le **tableau de réponse** retourné par l'API pour **chaque utilisateur**. C'est ici que l'on vient **comparer l'id de chaque user avec l'id du user actuellement connecté** afin de le supprimer de la liste pour qu'il ne voit pas son profil s'afficher en **retournant null** si les **deux ids sont identiques**. S'il n'y a **pas d'avatar**, on retourne l'ensemble des informations de l'utilisateur excepté l'**avatar** qui va prendre la **valeur de unknownAvatar**.

```
const processedUsers = response.data.map(user => {
  if (user.id_user === idUser){
    return null;
  }
  if (!user.avatar) {
    return {
      ...user,
      avatar: unknownAvatar
    }
  } else {
    return {
      ...user,
      avatar: user.avatar.replace("localhost", "192.168.0.14"),
    }
  }
}).filter(Boolean);
```

En revanche, si les informations contiennent le **chemin**

d'accès vers l'avatar, on le transforme en remplaçant le localhost par l'adresse **IPV4 utilisée par l'API**. Enfin on utilise la **méthode .filter** pour **supprimer les éléments nuls** de la constante processedUsers.

Définition de l'état des variables users et loading

Après retraitement de la réponse, on **définit l'état des variables users** et **loading** en utilisant le **useState**. La **variable users** contient désormais le **tableau retraité de la réponse contenu** dans la **constante processedUsers** et **loading est set à false** afin de pouvoir effacer l'affichage du Loading.... et afficher la liste des utilisateurs.

```
setUsers(processedUsers);  
setLoading(false);
```

En cas d'erreur, on va afficher le message d'erreur dans la console :

```
.catch(function(error) {  
  console.log('There has been a problem with your fetch operation: ' + error.message);  
})  
}
```

Le chat : fonctionnement et websockets

Le chat est une des principales fonctionnalités de notre projet. En effet, on distingue les **chats dans les salons de discussion** et les **chats privés**, qui sont tous deux **instantanés**, sans avoir à rafraîchir la page. Dans notre cas nous allons expliquer **comment s'articule la fonctionnalité du chat entre l'application client et le serveur**.

La mise en place des websockets

Les sockets sont un **mécanisme de communication** en **temps réel** entre un **client** et un **serveur**, permettant de transférer les données de manière **asynchrone** et **instantanée**. Ils doivent donc être mis en place à la fois chez le client et dans le serveur.

- Dans notre **API** les sockets sont mis en place de la manière suivante :

Après avoir créé l'**instance d'express dans la constante app** et créé le **serveur http** avec la méthode **createServer** contenu dans la **constante server**, on importe le **module Socket.IO** de Javascript qui facilite la **gestion des sockets en temps réel**. On initialise enfin une **instance de socketIO** dans la **constante io** qui prend en paramètre le **serveur** créé précédemment.

- Dans notre **application**, dans les écrans qui nécessitent l'utilisation de sockets, leur mise en place de la manière suivante :

```
import io from 'socket.io-client';
const socket = io.connect('http://192.168.0.14:3000')
```

Extrait fichier

PrivateChatScreen.js de l'application ChatApp - mise en place des websockets

On **importe la bibliothèque io de SocketIO** et on utilise la **méthode connect** dans la constante socket qui prend en paramètre l'**adresse du serveur et son port**.

Le **PrivateChatScreen** contient la **fonction sendMessagesInDb** qui permet d'envoyer des messages. Cette fonction vient appeler **successivement deux routes** :

- la **première route de l'API** appelée (**api/message**) vient **enregistrer le message** en base de données avec le texte, la date de création, et l'id du user qui envoie le message dans la **table Messages** dont on peut voir la **requête de l'API** associée ci-après :

```
//Sending messages in database
const sendMessagesInDb = async(text) => {
  fetch('http://192.168.0.14:3000/api/message', {
    method: 'POST',
    headers: { 'Authorization': 'Bearer ' + token,
              'Content-Type': 'application/json' },
    body: JSON.stringify({
      text: text,
    })
  })
  .then(data => data.json())
  .then(data => {
    if(data.error) {
      Alert.alert(data.error)
    }
  })
}
```

*Extrait fichier PrivateChatScreen.js de l'application ChatApp - fonction sendMessagesInDb (partie 1)
Appel de la première route localhost:3000/api/message/*

```
create: (data, callback)=>{
  const now = new Date(Date.now());
  const created = now.toISOString().slice(0,19).replace('T', ' ');
  pool.query(
    'insert into messages (text, date, id_user, channel_id) values (?, ?, ?, ?)',
    [
      data.text,
      created,
      data.id,
      data.channel_id
    ],
    (error, result, fields)=>{
      if(error){
        return callback(error);
      }
      return callback(null, result)
    }
  );
};
```

Extrait

fichier messages.services.js - fonction create permettant l'insertion d'un message dans la table messages de la base de données - appelée par la route localhost:3000/api/message/

- la **seconde route appelée**, et seulement en cas de **succès de la première requête**, va enregistrer dans la **table de liaisons users_messages** les informations telles que l'id du message qui vient d'être créé, l'id du user qui envoie le message et l'id du user à qui il est destiné avec la **requête de l'API** associée ci-après :

```
85 } else if (data.succes == 1) {
86   fetch('http://192.168.0.14:3000/api/message/private', {
87     method: 'POST',
88     headers: { 'Authorization': 'Bearer ' + token,
89               'Content-Type': 'application/json'
90             },
91     body: JSON.stringify({
92       id_user_from: userId,
93       id_user_to: towardUserId,
94       id_message: data.data.insertId,
95     })
96   })
```

Extrait

fichier PrivateChatScreen.js de l'application ChatApp - fonction sendMessagesInDb (partie 2)
Appel de la seconde route localhost:3000/api/message/private en cas de succès de la 1ère requête

```
createPrivateMessage: (body, callBack)->{
  pool.query(
    'insert into users_messages (id_user_from, id_user_to, id_message) values (?, ?, ?)',
    [
      body.id_user_from,
      body.id_user_to,
      body.id_message
    ],
    (error, result, fields)->{
      if(error){
        return callBack(error);
      }
      return callBack(null, result)
    }
  );
},
```

Extrait fichier messages.services.js - fonction createPrivateMessage permettant l'insertion des données d'un message dans la table de liaison users_messages appelée par la route localhost:3000/api/message/private

Si cette dernière **requête renvoie un succès**, c'est à ce moment là qu'intervient l'**envoi de socket** :

```
})
})
.then(data => data.json())
.then(data => {
  if(data.error) {
    Alert.alert(data.error)
  } else if (data.succes == 1) {
    socket.emit('private message', {
      text: text,
      name: userName,
      id: `${socket.id}${Math.random()}`,
      socketID: socket.id
    })
  }
})
})
```

Extrait fichier PrivateChatScreen.js de l'application ChatApp - fonction sendMessagesInDb (partie 3)
Suite au succès de la second requête, émission de l'évènement socket 'private message'

En effet, grâce à la **méthode socket.emit** le socket émet l'**événement 'private message'** qui est envoyé au serveur avec plusieurs informations telles que le contenu du texte, le nom de l'utilisateur et l'id du socket. Comment le serveur réceptionne-t-il alors l'émission du socket private message ?

La fonction de réception des messages et la réception de sockets

Dans le **fichier app.js de notre API** on crée le **gestionnaire d'évènement** qui va **écouter les connexion au serveur Socket.IO**

```
io.on('connection', (socket) => {
  console.log(`Connecté au client ${socket.id}`);

  socket.on('message', (data) => {
    console.log('message')
    io.emit('messageResponse', data);
  });
  socket.on('private message', (data) => {
```

En effet, grâce à la **méthode socket.on** on peut définir quels **événements** doivent être **écoutés par le serveur**. La ligne `socket.on('private message')` va donc écouter cet événement. A la suite de l'écoute de l'évènement, le serveur va donc lui-même

émettre au client l'événement **privateMessageResponse** qui contient les data qui lui ont été envoyées au préalable.

Extrait fichier App.js de l'API - gestionnaire d'évènement socket.IO

```
apiUrl = apiMessage + towardUserId

fetch(apiUrl, {
  method: 'GET',
  headers: { Authorization: 'Bearer ' + token },
})
.then(response => response.json())
.then(response => {
  let array = response.data
  setMessages(
    array.map(data => ({
      _id: data.id_message,
      createdAt: data.date,
      text: data.text,
      user: {
        _id: data.id_user_from,
        name: data.firstname + ' ' + data.lastname,
        avatar: data.avatar_from ? data.avatar_from.replace("localhost", "192.168.0.14") : unknownAvatar
      },
    })),
  )
}).catch(function(error) {
  console.log('There has been a problem with your fetch operation: ' + error.message);
});
```

Dans notre **client**, la fonction **getMessagesFromDb** va faire appel à la route de l'API **recupérant les messages privés entre deux utilisateurs**. Après avoir récupéré les informations désirées, elle va les **re-traiter** afin de pouvoir **structurer les données selon notre souhait**.

Extrait fichier PrivateChatScreen.js de l'application - fonction **getMessageFromDb**

```
// Call fetching previous messages with useEffect
useLayoutEffect(() => {
  socket.on('privateMessageResponse', (data) =>
    getMessagesFromDb(),
    setMessages([...messages, data]),
  );
}, [socket, messages]);
```

Extrait fichier

PrivateChatScreen.js de l'application - useLayoutEffect

Dans le **useLayoutEffect** ci-dessus, un hook similaire au **useEffect** mais exécuté de manière **synchrone** après les **mutations du DOM**, on utilise la **méthode socket.on** qui écoute l'événement **privateMessageResponse** envoyé par le **serveur**. A ce moment-là, on lance la **fonction getMessagesFromDb** et on utilise le **hook useState** défini plus haut sur la **constante messages** afin d'affecter à la constante messages les informations récupérées en base de données.

```
const [messages, setMessages] = useState([]);
```

Extrait fichier PrivateChatScreen.js de l'application - useState sur la variable messages

Pour **résumer le fonctionnement des websockets** dans le chat, il faut les **mettre en place** à la fois dans l'**API côté serveur** ainsi que dans l'**application côté client**. **Côté serveur**, il faut penser à **paramétrer les évènements qui vont être écoutés**. Ensuite **côté client**, lorsque la fonction d'**insertion des messages** envoyés en base de données renvoie un **succès**, on envoie au serveur un **événement** qui sera **écouté**, et va **émettre** une

DOSSIER PROFESSIONNEL (DP)

réponse au **client** stipulant qu'il a bien reçu l'information. A ce moment-là, on exécute la **fonction de récupération des messages** entre les deux utilisateurs en base de données pour les afficher dans le GiftedChat.

2. Précisez les moyens utilisés :

@react-navigation/stack	mime
bcrypt	multer
buffer	mysql
cors : Cors (Cross-Origin Resource Sharing)	sharp
dotenv	socket.io
Jim	Nodemon
jsonwebtoken	

3. Avec qui avez-vous travaillé ?

Maxime Hadj , Thomas Serdjebi et moi meme.

4. Contexte

Nom de l'entreprise, organisme ou association ▶ *La plateforme*

Chantier, atelier, service ▶ *Projet ChatApp dans le cadre de la formation*

Période d'exercice ▶ Du : *01/11/22* au : *01/04/2023*

5. Informations complémentaires (facultatif)

Concevoir et développer la persistance des données en intégrant les recommandations de sécurité

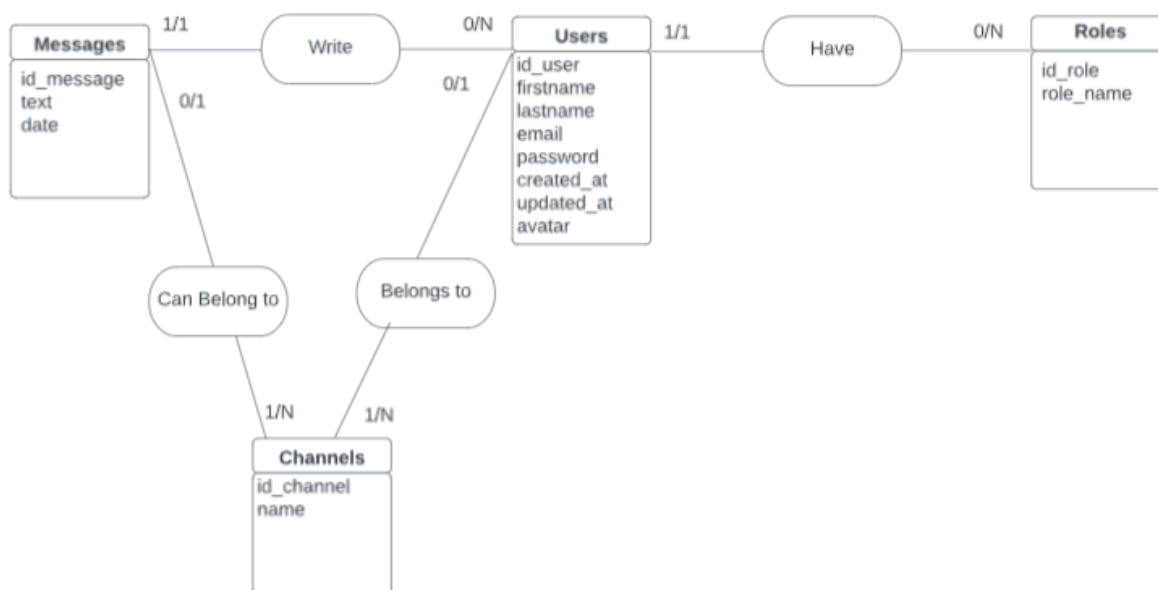
Activité-type 2

Exemple n° 1 - Concevoir une base de données

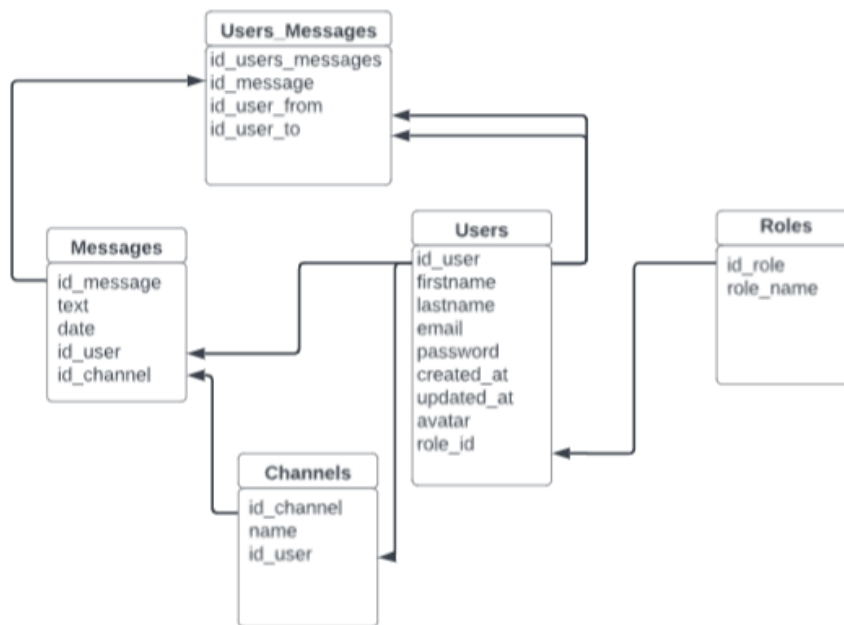
1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

Modèle Conceptuel de données

MCD : Le modèle conceptuel des données a pour but d'écrire de façon formelle les données.

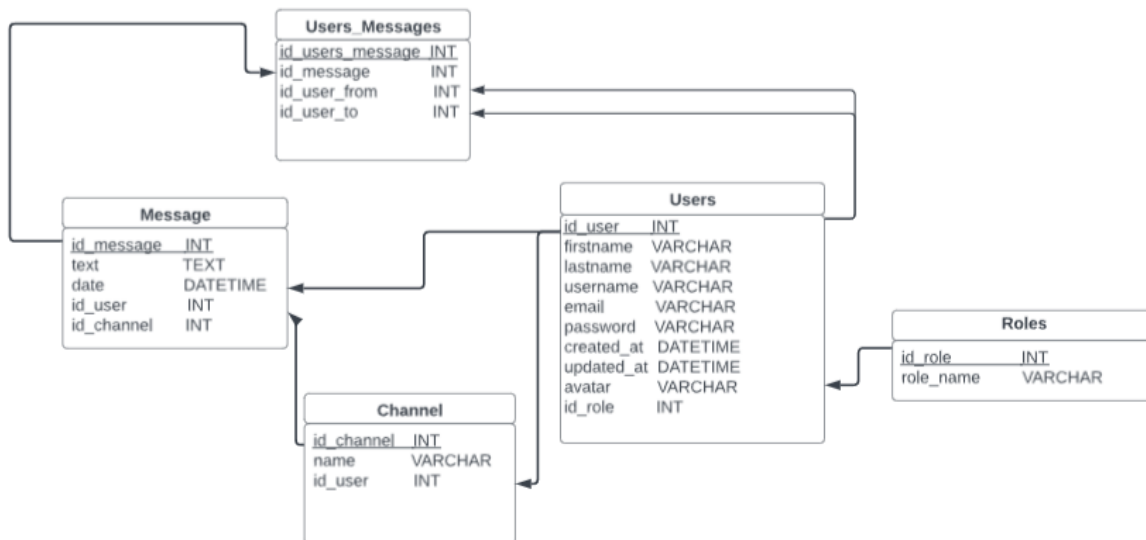


Modèle Logique de données



- Pour passer du MCD au MLD il faut effectuer les étapes suivantes:
- Les entités deviennent des tables.
- On supprime les verbes à l'infinitif.
- On supprime les cardinalités.
- Les relations entre les tables disparaissent pour laisser des places à des flèches qui vont de la primary key à la foreign key concernée.
- les relations n a n sont remplacées par des tables de liaisons.

Modèle Physique de données



Pour passer au MPD on ajoute le typage de toutes les variables

2. Précisez les moyens utilisés :

Mysql, Php MyAdmin

3. Avec qui avez-vous travaillé ?

Maxime Hadj , Thomas Serdjebi et moi même.

4. Contexte

Nom de l'entreprise, organisme ou association ➤ La plateforme

Chantier, atelier, service ➤ Projet ChatApp dans le cadre de la formation

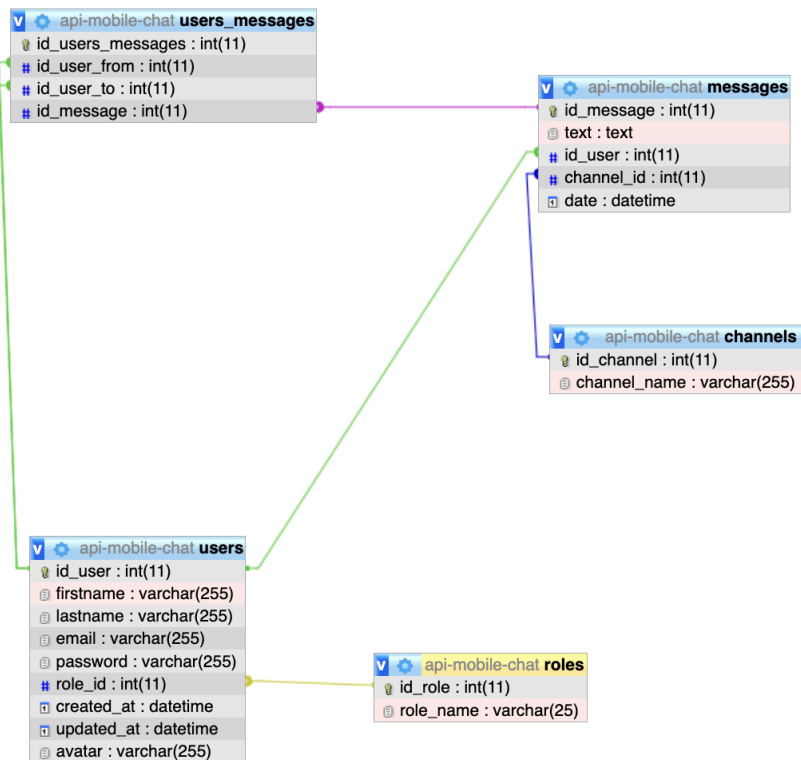
Période d'exercice ➤ Du : 01/11/22 au : 01/04/2023

5. Informations complémentaires (facultatif)

Exemple n° 2 - Mettre en place une base de données

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

Schéma de notre base de données avec les relations utilisées pour le bon traitement des données.



DOSSIER PROFESSIONNEL (DP)

Server: localhost:8889 » Database: api-mobile-chat

Structure SQL Search Query Export Import Operations Privileges Routines Events Triggers Designer

Filters
Containing the word:

Table	Action	Rows	Type	Collation	Size	Overhead
<input type="checkbox"/> channels	Browse Structure Search Insert Empty Drop	2	InnoDB	utf8_general_ci	16.0 KiB	-
<input type="checkbox"/> messages	Browse Structure Search Insert Empty Drop	188	InnoDB	utf8_general_ci	48.0 KiB	-
<input type="checkbox"/> roles	Browse Structure Search Insert Empty Drop	3	InnoDB	utf8_general_ci	16.0 KiB	-
<input type="checkbox"/> users	Browse Structure Search Insert Empty Drop	22	InnoDB	utf8_general_ci	32.0 KiB	-
<input type="checkbox"/> users_messages	Browse Structure Search Insert Empty Drop	66	InnoDB	utf8_general_ci	64.0 KiB	-
5 tables	Sum	281	InnoDB	utf8_general_ci	176.0 KiB	0 B

☐ Check all With selected:

Print Data dictionary

Create table

Name: Number of columns:

2. Précisez les moyens utilisés :

Mysql, Php MyAdmin

3. Avec qui avez-vous travaillé ?

Maxime Hadj , Thomas Serdjebi et moi meme.

4. Contexte

Nom de l'entreprise, organisme ou association ▶ *La plateforme*

Chantier, atelier, service ▶ *Projet ChatApp dans le cadre de la formation*

Période d'exercice ▶ Du : *01/11/22* au : *01/04/2023*

5. Informations complémentaires (facultatif)

Exemple n° 1 ▸ Développer des composants dans le langage d'une base de données

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

La base de données se devait d'être modélisée en suivant le **MCD** et le **MLD**. Elle devait à minima contenir les **tables** suivantes : **Users, Messages, Channels, Roles**. Sa gestion est en **MYSQL** est elle est hébergée sur **PHPMYADMIN**, un outil de gestion pour les systèmes de gestion de base de données MySQL et MariaDB réalisée principalement en PHP.

Composants d'accès à la base de données

```
1  const { createPool } = require('mysql2');
2
3  const pool = createPool ({
4    port: process.env.DB_PORT,
5    host: process.env.DB_HOST,
6    user: process.env.DB_USER,
7    password: process.env.DB_PASS,
8    database: process.env.MYSQL_DB
9  });
10
11 });
12
13 module.exports = pool ;
```

Le fichier **database.js** permet l'**accès à la base de données**. En effet, il utilise la fonction **createPool** du module **mysql2** afin de définir la **constante pool** qui permettra l'accès à la base de données. Les informations sont stockées dans **un fichier .env** qui n'est pas publié sur github lors du push des branches. Enfin la dernière ligne du fichier permet d'**exporter la constante** afin de l'utiliser dans les **fichiers Services** qui contiennent les **requêtes SQL**.

Les fichiers “Services”

Les fichiers **.services** ont tous une **structure identique**. En effet, on en trouve un pour les **messages**, un pour les **utilisateurs** et un pour les **messages**. Chacun de ces fichiers va fournir différentes **fonctionnalités**.

En premier lieu, on appelle la **constante pool** contenue dans le fichier **database.js** qui permettra l'accès à la base de données. Ensuite le **module.exports** va permettre d'exporter toutes ces **fonctionnalités** qui seront appelées dans les **fichiers .controller**.

Ensuite, à l'intérieur des **accolades du modules.exports**, on définit donc **chaque fonctionnalité** de la façon suivante :

- **nom de la fonction et ses paramètres** (dans l'exemple ci-dessous data et callBack)
- création de **variables supplémentaires** nécessaires à l'exécution de la requête
- la **méthode pool.query** contenant la **requête SQL** liée à la **fonctionnalité** désirée, qui prend plusieurs paramètres :
 - la **requête SQL**
 - les paramètres qui vont permettre de **remplacer les valeurs dans la requête SQL** et sont extraits initialement du **paramètre data** ou des **autres variables supplémentaires** créées comme la date de création
 - la **fonction de rappel dit callBack** est utilisée pour retourner le résultat ou **gérer les erreurs**. On constate que s'il n'y a pas d'erreurs, seul le résultat est retourné, s'il y a une erreur, l'erreur est retournée.

```
1  const pool = require ("../../config/database");
2
3  module.exports = {
4
5    //Create a user
6    create: (data, callBack) => {
7      const now = new Date(Date.now());
8      const created = now.toISOString().slice(0, 19).replace('T', ' ');
9      const defaultRole = 1;
10
11      pool.query(
12        'insert into users (firstname, lastname, email, password, role_id, created_at, updated_at, avatar) values (?, ?, ?, ?, ?, ?, ?, ?)',
13        [
14          data.firstname,
15          data.lastname,
16          data.email,
17          data.password,
18          defaultRole,
19          created,
20          data.updated_at,
21          data.imageUrl
22        ],
23        (error, results, fields) => {
24          if (error) {
25            console.log(error)
26            return callBack(error)
27          }
28          return callBack(null, results)
29        }
30      );
31    },
32  }
```

2. Précisez les moyens utilisés :

DOSSIER PROFESSIONNEL ^(DP)

3. Avec qui avez-vous travaillé ?

4. Contexte

Nom de l'entreprise, organisme ou association ▶ Cliquez ici pour taper du texte.

Chantier, atelier, service ▶ Cliquez ici pour taper du texte.

Période d'exercice ▶ Du : Cliquez ici au : Cliquez ici

5. Informations complémentaires (facultatif)

Concevoir et développer une application multicouche répartie en intégrant les recommandations de sécurité

Activité-type 3

Exemple n° 1 - Collaborer à la gestion d'un projet informatique et à l'organisation de l'environnement de développement

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

Pour la répartition des tâches, nous avons décidé de **travailler simultanément sur certaines tâches et séparément pour d'autres**.

Au sujet de la **base de données**, sa modélisation et sa création, nous avons **effectué les tâches ensemble**. Idem en ce qui concerne la **documentation**, les **installations** d'environnement et le **maquettage**, que ce soit pour l'API, l'application mobile, ou l'interface administrateur.

En ce qui concerne l'**API**, pour la **création des fichiers "Services", "Controllers" ou "Routes"**, nous nous sommes **répartis le travail en fonction des tables de base de données** qui allaient être impactées :

- Un collaborateur a développé les fichiers services, controllers, routes relatifs aux **utilisateurs**
- Un collaborateur a développé les fichiers services, controllers, routes relatifs aux **messages**
- Un collaborateur a développé les fichiers services, controllers, routes relatifs aux **salons de discussions**
- Nous nous sommes **répartis le développements des différents middlewares**
- Chacun a effectué ses **tests après développement** puis **communiquait dessus** avec les autres collaborateurs.

A propos de l'**application mobile** nous avons travaillé ensemble du maquettage en passant par l'installation de l'environnement jusqu'au développement du premier système temporaire de navigation.

Ensuite, nous nous sommes **répartis les écrans à développer, toujours dans une logique de trame que suis l'utilisateur à partir de la navigation** :

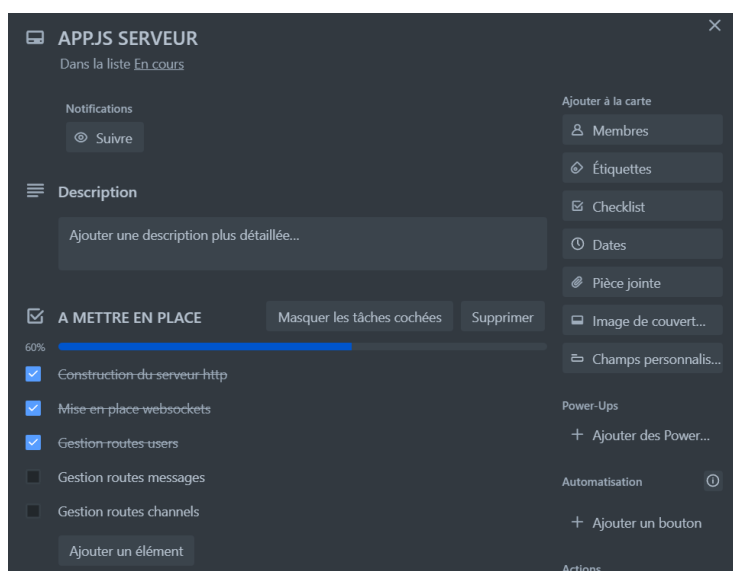
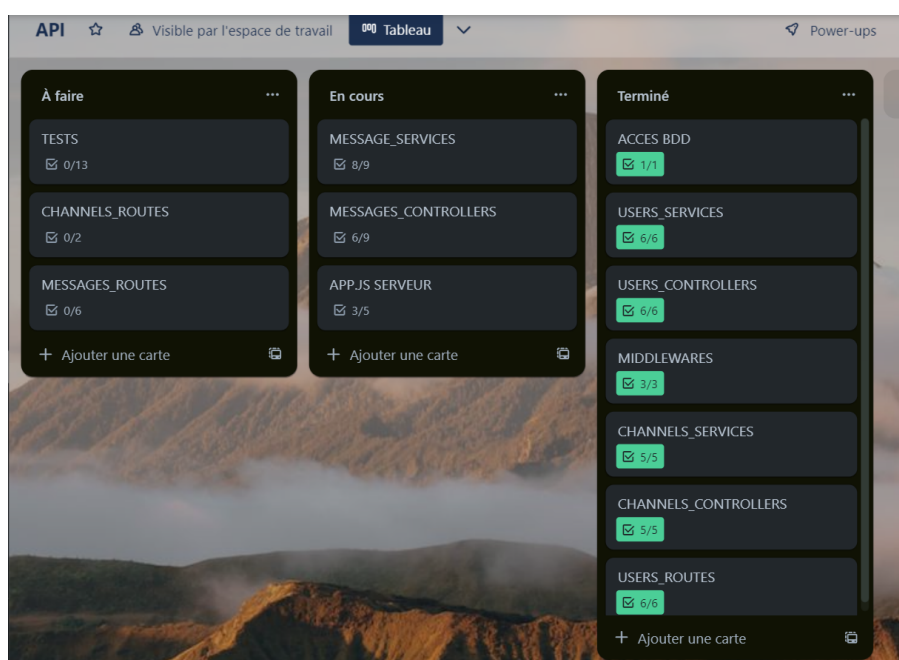
- Un collaborateur a développé les écrans d'inscription/connexion/profil/modification du profil
- Un collaborateur a développé les des écrans de la liste des utilisateurs, la liste des conversations privées, et l'écran de discussion privée
- Un s'est occupé des écrans de la liste des espaces de discussion, de l'écran d'espace de discussion ainsi que de la navigation finale

Nous faisons **un point ensemble** tous les deux jours et réalisons des **tests globaux** ensemble afin de voir les **corrections à apporter**.

Outil de suivi

Afin d'effectuer un suivi de nos tâches, nous avons utilisé l'**outil en ligne et collaboratif Trello**, permettant de créer un **tableau des tâches à effectuer**. En effet, nous avons créé des "**Cards**" dans lesquelles nous avons établi des **checklists des sous-tâches à effectuer**. Nous nous les avons assigné à l'oral mais l'outil permettait de les assigner. Cet outil permet aussi de changer le statut de chaque Card du statut "**A faire**" vers "**En cours**" ou "**Terminé**".

Voici quelques captures d'écrans afin de vous montrer le Trello mis en place pour l'API au cours du projet :

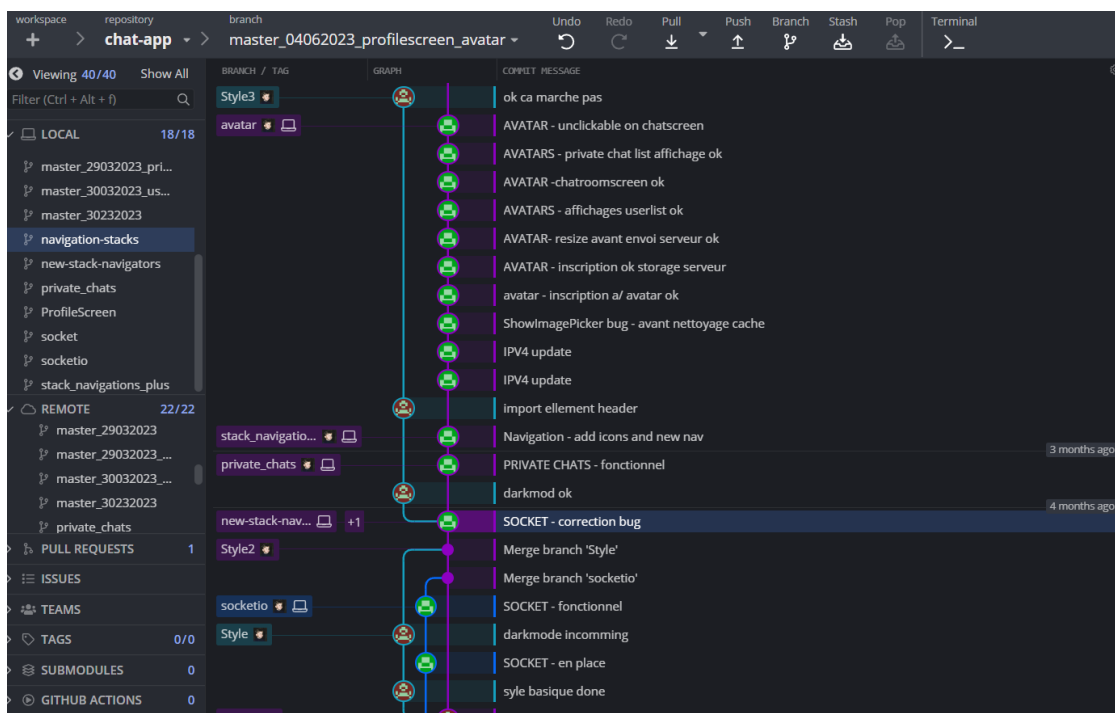


Versionning

Pour la gestion du versionning, nous avons utilisé **Git**. En effet, nos **repository** sont stockés sur **Github**. Nous avons créé une **branche à chaque développement** d'une **fonctionnalité** / d'un **fichier** / d'un **écran** liés par la trame logique que suit l'utilisateur.

Nous effectuons **plusieurs "commits"** avec des sur les **branches spécifiques** puis, après avoir achevé un développement nous effectuons des tests, apportons les corrections et enfin nous **fusionnons la branche spécifique à la branche principale**.

Quand la majorité des développements étaient aboutis et afin d'apporter les dernières modifications suite à des tests globaux, nous avons changé de méthode et créé des branches avec le nom des écrans ainsi que la date de modification. Quand tous les tests étaient terminés, nous fusionnions ces branches spécifiques datées à la branche principale.



2. Précisez les moyens utilisés :

Trello , Vs code , git, gitHub

3. Avec qui avez-vous travaillé ?

DOSSIER PROFESSIONNEL ^(DP)

Maxime Hadj , Thomas Serdjebi et moi meme.

4. Contexte

Nom de l'entreprise, organisme ou association ▶ *La plateforme*

Chantier, atelier, service ▶ *Projet ChatApp dans le cadre de la formation*

Période d'exercice ▶ Du : *01/11/22* au : *01/04/2023*

5. Informations complémentaires (facultatif)

Exemple n° 1 ➤ Développer des composants métier

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

Contrôleur : Le contrôleur gère les **demandes entrantes** et définit la logique de création d'un message privé. **Il vérifie la validité des paramètres** du corps de la demande (tels que id_user_from, id_user_to et id_message). Si l'un de ces paramètres est manquant, il renvoie une réponse d'erreur. Sinon, elle appelle la fonction **createPrivateMessage** en lui transmettant le corps de la requête et une fonction de rappel. Si une erreur survient lors de la création du message privé, elle **renvoie une réponse d'erreur**. Dans le cas contraire, il renvoie une réponse de succès avec les données du message privé créé.

```
createPrivateMessage: (req, res) =>{
  const body = req.body;
  if(!body.id_user_from){
    return res.status(500).json({
      succes: 0,
      error: "Your message is not sent from an existing user."
    })
  }

  if(!body.id_user_to){
    return res.status(500).json({
      succes: 0,
      error: "Your message is not sent to an existing user."
    })
  }


  if(!body.id_message){
    return res.status(500).json({
      succes: 0,
      error: "No message to sent."
    })
  }

  createPrivateMessage(body, (err, results)=>{
```

Service : Le composant service définit la fonction **createPrivateMessage** qui est responsable de l'insertion d'un nouveau message privé dans la base de données. Elle utilise des requêtes SQL pour insérer les valeurs **id_user_from**, **id_user_to** et **id_message** fournies dans la table **users_messages**. Si une erreur survient au cours de l'opération de base de données, la fonction de rappel est appelée avec l'erreur. Dans le cas contraire, elle appelle la fonction de rappel avec le résultat.

```
createPrivateMessage: (body, callBack)=>{
  pool.query(
    'insert into users_messages (id_user_from, id_user_to, id_message) values (?, ?, ?)',
    [
      body.id_user_from,
      body.id_user_to,
      body.id_message
    ],
    (error, result, fields)=>{
      if(error){
        return callBack(error);
      }
      return callBack(null, result)
    }
  );
},
```

Routeur : Le composant routeur définit **les routes liées à la gestion des messages**. Il importe les **fonctions de contrôleur nécessaires** et met en place les itinéraires à **l'aide du routeur express**. Les itinéraires pertinents pour les messages privés sont les suivants :

```
message >  message.router.js > ...
const {
  createMessage,
  deleteMessage,
  getMessagesFromChannel,
  getPrivateMessages,
  getPrivateDiscussions,
  getMessages,
  getMessage,
  createPrivateMessage, editMessage,
} = require("../message.controller");

const router = require("express").Router();
const { checkToken } = require("../auth/token_validation");
const { checkAdmin } = require("../auth/admin_validation");

//ROUTES ACCES MESSAGE
//Route create message
// router.post("/", checkToken, checkAdmin, createMessage);
//Route display messages
router.get("/", checkToken, getMessages);

//Route to send a message (in a channel)
router.post("/", checkToken, createMessage);
//Route to send a private message channel or private)
router.post("/private", checkToken, createPrivateMessage);
//Route delete message
router.delete("/:id", checkToken, deleteMessage);

//Route display edit message page
router.get("/:id", checkToken, getMessage);

//Route edit message
router.put("/:id", checkToken, editMessage);

router.delete("/:id", checkToken, checkAdmin, deleteMessage);
//Route get messages by id_channel
router.get("/channel/:id", checkToken, getMessagesFromChannel);
//Route get private messages
router.get('/private/:id', checkToken, getPrivateMessages);
//Route get private discussions
router.get('/discussions/:id', checkToken, getPrivateDiscussions);

module.exports = router;
```

POST /private : Elle appelle la fonction **createPrivateMessage** lorsqu'une requête POST est envoyée à la route /private. **Elle exige une authentification par jeton (checkToken)** et crée un message privé sur la base du corps de la requête.

GET /private/:id : Il récupère les messages privés d'un utilisateur spécifique sur la base de l'identifiant fourni. **Il nécessite une authentification par jeton (checkToken).**

GET /discussions/:id : Récupère les discussions privées d'un utilisateur spécifique sur la base de l'identifiant fourni. **Il nécessite une authentification par jeton (checkToken).**

DOSSIER PROFESSIONNEL ^(DP)

2. Précisez les moyens utilisés :

React, SQL, Node.js

3. Avec qui avez-vous travaillé ?

Maxime Hadj , Thomas Serdjebi et moi meme.

4. Contexte

Nom de l'entreprise, organisme ou association ▶ *La plateforme*

Chantier, atelier, service ▶ *Projet ChatApp dans le cadre de la formation*

Période d'exercice ▶ Du : *01/11/22* au : *01/04/2023*

5. Informations complémentaires (facultatif)

Exemple n° 1 ▶ Construire une application organisée en couches

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

Une application en couches est une architecture logicielle qui **organise les différents composants** d'une application en couches distinctes, chacune ayant ses propres responsabilités et fonctionnalités. Cette approche contribue à la modularité, à l'évolutivité et à la maintenabilité de l'application.

Couche de présentation (application mobile) :

Cette couche représente l'interface utilisateur de l'application mobile construite à l'aide de **React Native et Expo**. Elle est responsable du **rendu des composants** de l'interface utilisateur, de la gestion des interactions avec l'utilisateur et de la présentation des données aux utilisateurs finaux. La couche de présentation communique avec les autres couches par le biais d'API et reçoit des données de la couche API pour les afficher.

Couche de logique d'entreprise (API) :

La couche API, mise en œuvre à l'aide de **Node.js**, sert d'intermédiaire entre la couche de présentation et la couche de données. **Elle gère la logique de l'application**, qui comprend le traitement des demandes des utilisateurs, l'exécution de diverses opérations et l'interaction avec la couche de données. **La couche API fournit des end point** que l'application mobile peut appeler pour demander et envoyer des données.

Couche de données (base de données) :

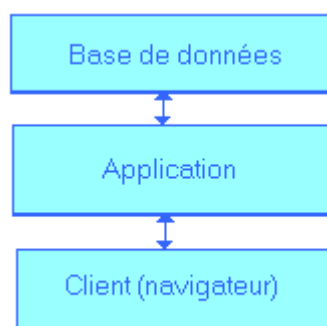
La couche de données est constituée d'une base de données où sont stockées les données de l'application. Il peut s'agir d'une **base de données relationnelle** (par exemple, MySQL, PostgreSQL) ou d'une base de données NoSQL (par exemple, MongoDB). La couche de données est responsable du stockage, de l'extraction et de la modification des données de l'application en fonction des demandes reçues de la couche API.

Couche administrative (back office) :

La couche back office est un composant ou une **interface distincte** conçue à des fins administratives. Elle permet aux utilisateurs autorisés, tels que les administrateurs ou les gestionnaires de contenu, d'**effectuer des tâches administratives liées à l'application**. Cette couche a **sa propre interface** utilisateur et sa propre fonctionnalité, offrant des fonctions telles que la gestion du contenu, la gestion des utilisateurs.

En divisant l'application en ces couches, vous obtenez une séparation des préoccupations, où chaque couche se concentre sur ses tâches spécifiques. Cette séparation **améliore la flexibilité**, la maintenabilité et la testabilité de l'application. Elle permet également un développement parallèle et la **possibilité de remplacer ou de mettre à jour des couches individuelles sans affecter les autres**.

La communication entre ces couches se fait généralement par **l'intermédiaire d'API bien définies**, ce qui garantit une séparation claire et l'encapsulation des fonctionnalités. La couche de présentation interagit avec la couche API pour récupérer et envoyer des données, tandis que la couche API gère la logique commerciale et communique avec la couche de données pour accéder aux données et les modifier.



DOSSIER PROFESSIONNEL ^(DP)

2. Précisez les moyens utilisés :

Vs code

3. Avec qui avez-vous travaillé ?

Maxime Hadj , Thomas Serdjebi et moi meme.

4. Contexte

Nom de l'entreprise, organisme ou association ▶ *La plateforme*

Chantier, atelier, service ▶ *Projet ChatApp dans le cadre de la formation*

Période d'exercice ▶ Du : *01/11/22* au : *01/04/2023*

5. Informations complémentaires (facultatif)

Exemple n° 1 ▶ Développer une application mobile

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

Le développement d'une application mobile implique plusieurs composants et technologies pour créer une expérience fonctionnelle. Dans notre application de chat, nous avons utilisé divers outils et cadres pour atteindre nos objectifs.

React Native : React Native est un framework JavaScript populaire pour la création d'applications mobiles multiplateformes. Il permet aux développeurs d'écrire du code une seule fois et de le déployer sur **les plateformes iOS et Android**, ce qui leur permet d'économiser du temps et des efforts. Avec React Native, nous pouvons créer **une interface utilisateur de type natif et accéder à des fonctionnalités spécifiques à chaque appareil**.

Node.js : Node.js est un moteur d'exécution JavaScript côté serveur qui nous **permet de construire le backend de notre application**. Il offre une architecture évolutive et axée sur les événements, **ce qui le rend adapté à la gestion de la communication en temps réel**, telle que la fonctionnalité de chat. Node.js nous permet de gérer efficacement le stockage des données, l'authentification et d'autres opérations côté serveur.

SQL : SQL (Structured Query Language) est un langage standard pour la gestion et la manipulation des bases de données relationnelles. Nous avons utilisé SQL pour stocker et récupérer les messages de chat, les informations sur les utilisateurs et d'autres données pertinentes. SQL fournit **un moyen structuré et efficace de gérer les données**, garantissant l'intégrité et la cohérence des données dans notre application.

Gifted Chat : Gifted Chat est une bibliothèque de **composants d'interface utilisateur préconstruite**, spécialement conçue pour créer des interfaces de chat. Elle offre un ensemble de composants de chat personnalisables et riches en fonctionnalités qui facilitent la mise en œuvre de la fonctionnalité de messagerie. Avec Gifted Chat, nous pouvons afficher des conversations, envoyer et recevoir des messages, **gérer les interactions des utilisateurs** et incorporer divers formats de médias dans notre application de chat.

Expo : Expo est une plateforme qui simplifie le développement, **les tests et le déploiement des applications React Native**. Elle fournit une suite d'outils, de bibliothèques et de services qui rationalisent le processus de développement. Avec Expo, nous pouvons facilement prévisualiser et tester notre application sur des appareils physiques, accéder aux API natives sans avoir besoin de code natif, et simplifier le processus de déploiement en générant des builds autonomes.

En combinant ces technologies, nous avons pu créer une application de chat robuste et efficace. React Native nous a permis de construire une interface utilisateur multiplateforme, Node.js a facilité la communication en temps réel, SQL a assuré la persistance des données, Gifted Chat a fourni une interface de messagerie intuitive, et Expo a simplifié le flux de travail de développement et de déploiement. Ensemble, ces technologies ont constitué la base de notre processus de développement d'applications mobiles réussies.

2. Précisez les moyens utilisés :

3. Avec qui avez-vous travaillé ?

Maxime Hadj , Thomas Serdjebi et moi meme.

4. Contexte

Nom de l'entreprise, organisme ou association ▶

La plateforme

DOSSIER PROFESSIONNEL ^(DP)

Chantier, atelier, service ▶

Projet ChatApp dans le cadre de la formation

Période d'exercice ▶ Du : 01/11/22 au : 01/04/2023

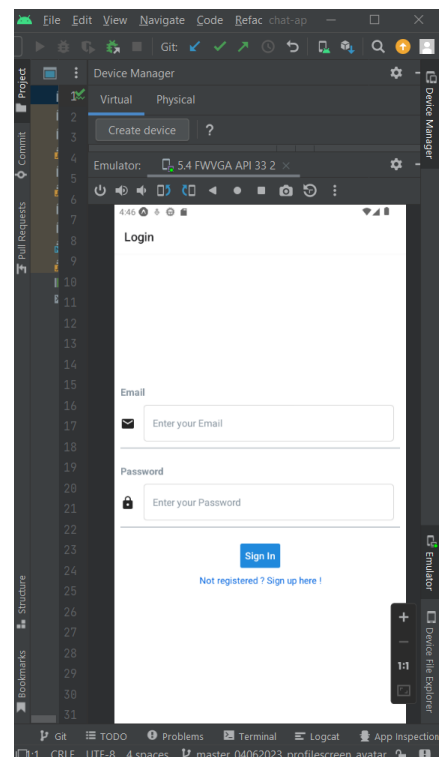
5. Informations complémentaires (facultatif)

Exemple n° 1 - Préparer et exécuter le déploiement d'une application

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

Nous avons développé notre application à l'aide d'**Expo**. Le **déploiement d'une application** est un **processus** qui permet de **rendre l'application mobile disponible pour les utilisateurs**, et Expo est justement un **ensemble d'outils** qui facilite le développement d'applications en utilisant notamment React Native.

Nous avons donc installé **Node.js** et **Expo CLI** dans notre projet développé avec Visual Studio Code, puis créé le projet grâce à la **commande expo init** dans le terminal qui vient **structurer le projet de base** avec les fichiers et dossiers préconfigurés. Au cours du développement, nous avons utilisé la **commande 'expo start'** qui permet de **démarrer le serveur de développement Expo** et permet de **tester l'application** via notre **émulateur Android Studio**(permettant de générer une image de différents appareils de différentes tailles iPhone, Android, tablettes...) ou **directement sur notre téléphone** en scannant le **QR code** généré dans le terminal et ouvrant notre application dans l'application mobile Expo Go.



Expo offre aussi la possibilité de **publier l'application** et de la rendre disponible via le service d'hébergement d'Expo, de la distribuer par la génération de fichiers binaires ou de les soumettre aux stores respectifs.

DOSSIER PROFESSIONNEL ^(DP)

2. Précisez les moyens utilisés :

3. Avec qui avez-vous travaillé ?

Maxime Hadj , Thomas Serdjebi et moi meme.

4. Contexte

Nom de l'entreprise, organisme ou association ▶ *La plateforme*

Chantier, atelier, service ▶ *Projet ChatApp dans le cadre de la formation*

Période d'exercice ▶ Du : *01/11/22* au : *01/04/2023*

5. Informations complémentaires *(facultatif)*

DOSSIER PROFESSIONNEL ^(DP)

Exemple n° 1 ▶ *Préparer et exécuter le déploiement d'une application*

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

2. Précisez les moyens utilisés :

3. Avec qui avez-vous travaillé ?

4. Contexte

Nom de l'entreprise, organisme ou association ▶ Cliquez ici pour taper du texte.

Chantier, atelier, service ▶ Cliquez ici pour taper du texte.

Période d'exercice ▶ Du : Cliquez ici au : Cliquez ici

5. Informations complémentaires (facultatif)

Titres, diplômes, CQP, attestations de formation

DOSSIER PROFESSIONNEL ^(DP)

(facultatif)

Intitulé	Autorité ou organisme	Date
Cliquez ici.	Cliquez ici pour taper du texte.	Cliquez ici pour sélectionner une date.

Déclaration sur l'honneur

Je soussigné(e) [prénom et nom] [Cliquez ici pour taper du texte.](#) ,
déclare sur l'honneur que les renseignements fournis dans ce dossier sont exacts et que je suis
l'auteur(e) des réalisations jointes.

Fait à [Cliquez ici pour taper du texte.](#) le [Cliquez ici pour choisir une date](#)
pour faire valoir ce que de droit.

Signature :

Documents illustrant la pratique professionnelle

(facultatif)

Intitulé
Cliquez ici pour taper du texte.

DOSSIER PROFESSIONNEL ^(DP)

ANNEXES

(Si le RC le prévoit)