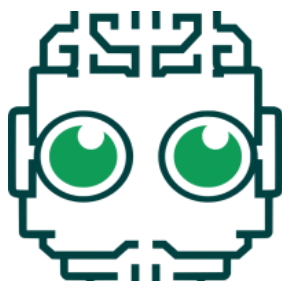

RAPPORT DE STAGE

Création d'une application qui affiche le contenu d'un menu personnalisable



OPINAKA

Réalisé par
Etienne TILLER

Sous la direction de

Maître de stage :
Moh ATTIK

Tuteur de stage :
Marc JOANNIDES

Pour l'obtention du DUT
Année universitaire: 2020 - 2021

Remerciements

Suite à la réalisation de ce projet, je tiens à exprimer ma profonde gratitude à mon tuteur de stage M.Joannides, pour ses conseils et son suivi.

Je tiens également à remercier mon maître de stage, M.Attik pour son suivi quotidien, ses conseils, sa confiance et son accueil dans l'entreprise.

Enfin, merci à toutes les personnes qui m'ont conseillé pour la réalisation du design de l'application.

Sommaire

Remerciements	2
Sommaire	3
Table des figures	5
Glossaire	7
Introduction	8
Présentation de l'entreprise	9
1. Analyse de l'existant	10
1.1. Recherche bibliographique	10
1.1.1. React	10
1.1.1.1. Les composants	10
1.1.1.2. Les hooks	13
1.1.1.3. Les props	16
1.2. Recherche de clones	16
1.2.1. Présentation de Deer	17
1.2.1.1. Organisation des composants React	17
1.2.1.2. Le stockage des données	19
1.2.1.3. Utilisation de Deer	20
2. Cahier des charges	21
2.1. Analyse des besoins fonctionnel	21
2.2. Analyse des besoins non fonctionnels	22
3. Rapport technique	22
3.1. Conception	23
3.1.1. Choix technologiques	23
3.1.2. Conception des bases de données	23
3.1.3. Conception des algorithmes	29
3.2. Réalisation	30
3.2.1. Réalisation de la page de connexion	31
3.2.2. Réalisation de la synchronisation des bases de données	36
3.2.3. Lecture et modification des données dans la base de données locale	38
3.2.4. Génération du menu	40
3.2.5. Réalisation du style personnalisable	41
4. Résultat	44

4.1. Manuel d'utilisation	44
4.2. Validation	50
5. Rapport d'activité	51
5.1. Méthode de développement et outils	51
5.2. Bilan critique par rapport au cahier des charges	52
Conclusion	52
Visa du maître de stage	53
Bibliographie	54
Annexe	55
4ème de couverture	56
Résumé en français	56
Résumé en anglais	56

Table des figures

Figure 1 : Variable en JSX	10
Figure 2 : Composant React "ProfilePage" au format d'une fonction	11
Figure 3 : Composant React "ProfilePage" au format d'une classe	11
Figure 4 : Composants React réutilisables	12
Figure 5 : Schéma des composants en React	13
Figure 6 : Exemple useState	14
Figure 7 : Exemple useEffect	15
Figure 8 : Article pour/contre Clone TMDb	16
Figure 9 : Article pour/contre de l'application Deer	17
Figure 10 : Schéma de l'organisation des composants de l'application Deer	18
Figure 11 : Return du composant noteList	19
Figure 12 : Table Note	20
Figure 13 : Diagramme d'utilisation de Deer	20
Figure 14 : Diagramme d'utilisation de l'application	21
Figure 15 : Diagramme de la base de données embarquée (SQLite)	24
Figure 16 : Service d'authentification Firebase	25
Figure 17 : Base de données en temps réel Firebase	25
Figure 18 : Base de données Firebase (authLocal)	26
Figure 19 : Base de données Firebase (menu)	26
Figure 20 : Dossier stockant les sites web en local	27
Figure 21 : Base de données Firebase (style)	28
Figure 22 : Diagramme de séquence de la connexion à l'application	29

Figure 23 : Diagramme de séquence de l'utilisation de l'application.....	30
Figure 24 : Return du composant LoginMenu.....	31
Figure 25 : Classe disabled CSS.....	32
Figure 26 : Return du composant LoginFirebase.....	33
Figure 27 : Méthode handleCheckLogin.....	33
Figure 28 : Méthode connexionFirebase.....	34
Figure 29 : Méthode displayConnexion.....	35
Figure 30 : Méthode syncBD.....	36
Figure 31 : Méthode syncMenu.....	37
Figure 32 : Méthode fetchItems.....	38
Figure 33 : Méthodes cacher et chacherOnline.....	39
Figure 34 : Return du composant ListItem.....	40
Figure 35 : Méthode menuRender.....	41
Figure 36 : Méthode loadStyleUser.....	42
Figure 37 : useEffect permettant d'afficher le style personnalisé.....	43
Figure 38 : Méthode updateStylePage.....	44
Figure 39 : Page de choix de connexion de l'application.....	45
Figure 40 : Formulaire de connexion Firebase de l'application.....	46

Rapport de projet pour l'entreprise Opinaka : Menu personnalisable

Tillier Etienne

Figure 41 : Page d'accueil de l'application en étant connecté.....	47
Figure 42 : Affichage d'un site dans l'application (tmdb).....	48
Figure 43 : Authentification firebase (ajout utilisateur).....	49
Figure 44 : Base de données (RealtimeDB) de Firebase.....	50

Glossaire

BtoB : Business to business, ce qui signifie que l'entreprise génère de l'argent en rendant des services à d'autres entreprises

BIC : Business & Innovation Centre

Front-End : Partie d'un programme informatique responsable uniquement de l'interface utilisateur.

Framework : Ensemble de composants structurels permettant de construire des logiciels ou des sites internet.

Introduction

De nos jours, pour les entreprises, de plus en plus d'outils deviennent indispensables pour rester compétitifs. Des innovations qui ne cessent d'apparaître et auxquelles il faut s'adapter.

Le personnel de ces organisations doit donc savoir utiliser ces outils qui sont souvent tous différents et indépendants les uns des autres. Ils ne sont donc pas forcément accessibles par le biais des mêmes plateformes et cela peut être contraignant pour les entreprises.

L'entreprise Opinaka est une startup du numérique qui travaille au contact des entreprises. Cette entreprise m'a confié le projet qui est de développer une application qui permet de regrouper dans un menu, les outils disponibles par navigateur qui seraient disponibles directement dans l'application.

Dans un premier temps, je présenterai l'entreprise Opinaka. Ensuite, après avoir fait une analyse de l'existant ainsi qu'une présentation du cahier des charges, je détaillerai les choix de conception et le déroulement de la réalisation présent dans le rapport technique. Finalement, je mettrai en avant les résultats du projet mais aussi un rapport d'activité qui décrit les méthodes de travail que j'ai utilisé durant le projet.

Présentation de l'entreprise

Afin de présenter au mieux l'entreprise, j'ai fait mes recherches sur internet ainsi qu'auprès de mon maître de stage car très peu d'informations sont présentes sur internet. De même, durant le stage, peu d'informations concernant l'entreprise m'ont été transmises. Malgré tout, j'ai pu ci-dessous rédiger une présentation de l'entreprise avec son essentiel.

Opinaka est une startup qui a été créée il y a plus de 5 ans à Montpellier. C'est donc une entreprise montpelliéraine et elle fonctionne en BtoB, ce qui signifie que son business consiste à proposer ses services à d'autres entreprises. Opinaka propose des services dans l'analyse avancée de données (détections de doublons, nettoyage de données, transformations de données, prédiction, etc.). Monsieur Attik est le seul responsable de la startup où les membres de cette dernière contribuent à l'entreprise en fonction de leur disponibilité et des missions en cours. Les personnes travaillant pour l'entreprise sont répartis par services, il existe trois types de service dans la startup : les services professionnels qui permettent d'assurer la modélisation et l'analyse des données, les services marketing et commercial qui ont pour objectif de rechercher de nouveaux clients et qui sont également en charge de la veille marketing, et enfin les services de recherche et développement qui ont pour but de développer des solutions techniques (développement web/mobile/algorithme). Le travail se fait à distance et est régulièrement mis en commun lors de réunions avec le reste du groupe travaillant sur le même projet. Bien évidemment, le nombre de personnes par groupe dépend de la tâche à réaliser. Dans le cadre du stage, nous étions plusieurs équipes de stagiaires travaillant chacune sur un projet différent.

En plus de proposer des services pour des entreprises, Opinaka travaille également en collaboration avec des chercheurs, elle réalise des publications internationales sur des problèmes complexes. Dernièrement, suite à la crise sanitaire, la startup s'intéresse davantage au domaine médical. En effet, elle travaille désormais en collaboration avec le BIC de Montpellier dans le but de monter un projet de création d'assistants intelligents dans le domaine de la santé.

1. Analyse de l'existant

Afin de savoir comment et avec quels moyens j'allais pouvoir réaliser mon projet, j'ai au préalable mené une analyse de l'existant sur les bibliothèques Front-End en JavaScript ainsi que sur des clones d'interfaces utilisateurs codés en JavaScript. L'objectif de ces recherches est de se rendre compte des solutions open source déjà développées pour ne pas réinventer la roue.

1.1. Recherche bibliographique

A travers cette recherche, j'ai découvert de nombreuses bibliothèques ou framework utilisés par de grandes entreprises comme Uber pour leur application. Uber a développé sa propre bibliothèque "BaseWeb" qui est basée sur le framework React afin de faciliter le travail de ses équipes travaillant sur le développement de leurs applications et sites web.

1.1.1. React

1.1.1.1. Les composants

Comme pour Uber, le framework React est aujourd'hui beaucoup utilisé car très pratique. React fonctionne avec des composants, ces composants sont comparables à des éléments HTML. En effet, on développe en React avec du JavaScript et plus précisément du JSX, il s'agit d'un langage permettant d'implémenter des balises HTML dans du code JavaScript.

```
const element = <h1>Bonjour, monde !</h1>;
```

Figure 1 : Variable en JSX

Les composants React sont semblables à une fonction classique avec un return, c'est dans ce return qu'on renvoie les balises en JSX qui seront retranscrites dans la page en HTML.

```
function ProfilePage(props) {  
  const showMessage = () => {  
    alert('Vous suivez désormais ' + props.user);  
  };  
  
  const handleClick = () => {  
    setTimeout(showMessage, 3000);  
  };  
  
  return (  
    <button onClick={handleClick}>Suivre</button>  
  );  
}
```

Figure 2 : Composant React "ProfilePage" au format d'une fonction

Dans la figure 2, le composant renvoie un bouton qui sera affiché comme s'il avait été directement implémenté dans une page HTML classique. Il faut aussi savoir que c'est assez récent que les composants React soient codés au format d'une fonction, auparavant, ils étaient codés sous forme de classe.

```
class ProfilePage extends React.Component {  
  showMessage = () => {  
    alert('Vous suivez désormais ' + this.props.user);  
  };  
  
  handleClick = () => {  
    setTimeout(this.showMessage, 3000);  
  };  
  
  render() {  
    return <button onClick={this.handleClick}>Suivre</button>;  
  }  
}
```

Figure 3 : Composant React "ProfilePage" au format d'une classe

Comme on peut le voir dans la figure 3, il s'agit du même composant que dans la figure 2 codé différemment. Dans cet exemple de composant, cela renvoie seulement un bouton, or il est possible que cela renvoie davantage de balise. On

pourrait imaginer un formulaire d'inscription par exemple avec plusieurs inputs. En plus de pouvoir regrouper plusieurs balises HTML dans un même composant, il est aussi possible de faire appel à ses composants dans d'autres composants. C'est là que React devient très puissant, puisque tout ce que l'on crée peut être utilisé à n'importe quel moment. Les composants sont importés et utilisables comme des balises HTML classiques.

```
class GreatGrandParent extends Component {
  render(){
    return <GrandParent message={"hello context"}/>
  }
}
class GrandParent extends Component {
  render(){
    return <Parent message={this.props.message}/>
  }
}
class Parent extends Component {
  render(){
    return <You message={this.props.message}/>
  }
}
class You extends Component {
  render(){
    return <p>Great grandpa says {this.props.message}</p>
  }
}
```

Figure 4 : Composants React réutilisables

Dans la figure 4, les composants "You", "Parent" et "GrandParent" sont appelés dans d'autres composants. Cet exemple est très simple mais on peut imaginer un composant header ainsi qu'un composant footer. Ces deux composants pourraient être appelés au début et à la fin de chaque page juste en insérant la balise dans le

return.

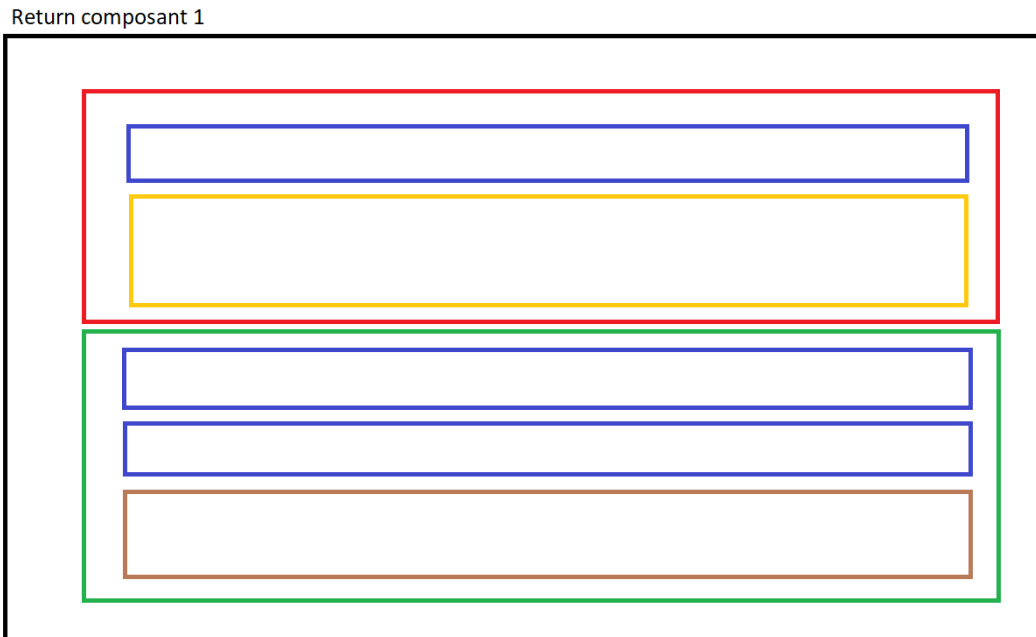


Figure 5 : Schéma des composants en React

Chaque rectangle de couleur représente un composant React dans la figure 5, les rectangles de même couleur représentent le même composant. On remarque donc qu'un composant peut être appelé autant de fois que nécessaire dans un composant.

1.1.1.2. Les hooks

Les composants React possèdent tous un cycle de vie, il est différent selon le format du composant (classe ou méthode). Dans mes recherches et pour mon projet, j'ai fait la connaissance des "hooks" qui fonctionnent avec le format méthode des composants. Les hooks permettent de dynamiser les composants, effectivement, ce sont des variables qui peuvent interagir directement avec le composant et lui permettre de rafraîchir son return afin que les données affichées soient actualisées par exemple. Parmi ces hooks, on peut y trouver le "useState" ou encore le "useEffect". Le useState est un couple composé d'une variable permettant de stocker une valeur ainsi que d'une fonction permettant de modifier la valeur de la variable.

```
import React, { useState } from 'react';

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Figure 6 : Exemple useState

Les useState appelés dans le return du composant permettent un rafraîchissement du composant dès lors que la valeur du useState est modifiée. Comme présenté dans la figure 6, le useState "count" est appelé dans une balise "<p>", et sa méthode de modification est appelée dans la fonction "onClick" du bouton. Cela signifie que lorsque le bouton est pressé, alors la variable du useState change et donc le composant se met à jour. Le useEffect est différent du useState, il faut plus le voir comme une méthode qui va s'exécuter au chargement du composant et s'exécuter de nouveau si une des variables en paramètre de ce dernier est mise à jour.

```
function MonComposantSFC() {  
  const [formations, setFormations] = useState([]);  
  
  useEffect(() => {  
    async function fetchData() {  
      const result = await axios(  
        "https://formation-cepegra.be/wp-json/wp/v2/session"  
      );  
      setFormations(result.data);  
    }  
    fetchData();  
  }, []);  
  
  return (  
    <ul>  
      {formations.map(el => (  
        <li>{el.title.rendered}</li>  
      ))}  
    </ul>  
  );  
}
```

Figure 7 : Exemple useEffect

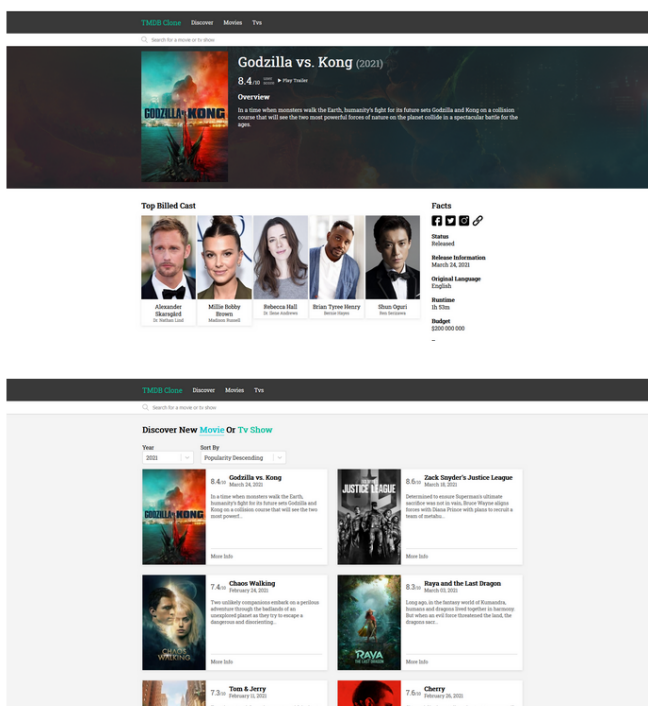
Dans l'exemple présenté dans la figure 7, on y retrouve un `useEffect` qui fait office de méthode de récupération de données. Dès que le composant est chargé, le `useEffect` va chercher les données et dans l'exemple, il les stocke dans une `useState` qui lui est appelé dans le `return` pour l'affichage. Comme je l'ai dit précédemment, il est possible que le `useEffect` se relance. Dans l'exemple, il y a des crochets encadrés en rouge, c'est ici qu'on peut mettre en paramètre des variables. S'il y en a et qu'elles viennent à changer alors même que le composant est déjà chargé, alors le `useEffect` se relance. On pourrait imaginer pour cet exemple, une base de donnée qui se met à jour régulièrement et afin d'afficher les données à jour de manière synchronisée, alors on pourrait mettre une variable qui serait mise à jour dès que la base de donnée le serait, ainsi en la mettant en paramètre du `useEffect`, ce dernier se relancerait au moment de la mise à jour.

1.1.1.3. Les props

Les props sont les données qui sont transférées de composants en composants, étant donné qu'en JavaScript les variables peuvent contenir tout et n'importe quoi, alors les props peuvent aussi être de tout type (Hook,variable,méthode...). Ainsi, avec ce transfert de donnée, il est possible de modifier un hook dans un composant afin qu'un autre composant soit mis à jour suite à la modification de ce hook.

1.2. Recherche de clones

Sur internet et plus précisément sur Github, on retrouve énormément de projets et parmi ces derniers, certains correspondent à des clones d'applications ou sites web existants. J'ai mené une recherche de clones dans un premier temps afin de me rendre compte de ce qu'il est possible de réaliser à partir d'un code en JavaScript avec le framework React. Ensuite, j'avais pour but de me servir d'un clone comme base afin de réaliser mon projet. J'ai trouvé plus d'un centaine de clones mais la grande majorité ne sont pas fonctionnels ou sont incomplets.



- Utilise l'API TMDB
- Page de recherche de films et séries télé
- Tri par catégorie pour la recherche
- Page détaillée pour chaque film
- Vidéo trailer pour chaque film disponible
- Liste des acteurs de chaque film
- Recommandations de films
- Les films sont notés
- Réseaux sociaux de chaque film

- Il n'y a pas de page pour les acteurs
- On ne peut pas noter les films
- Il n'y a pas de système de connexion

Figure 8 : Article pour/contre Clone TMDB

Parmi les clones fonctionnels comme pour le clone TMDB dans la figure 8, certains possèdent une interface utilisateur qui se rapproche énormément de l'application clonée et offre de nombreuses fonctionnalités.

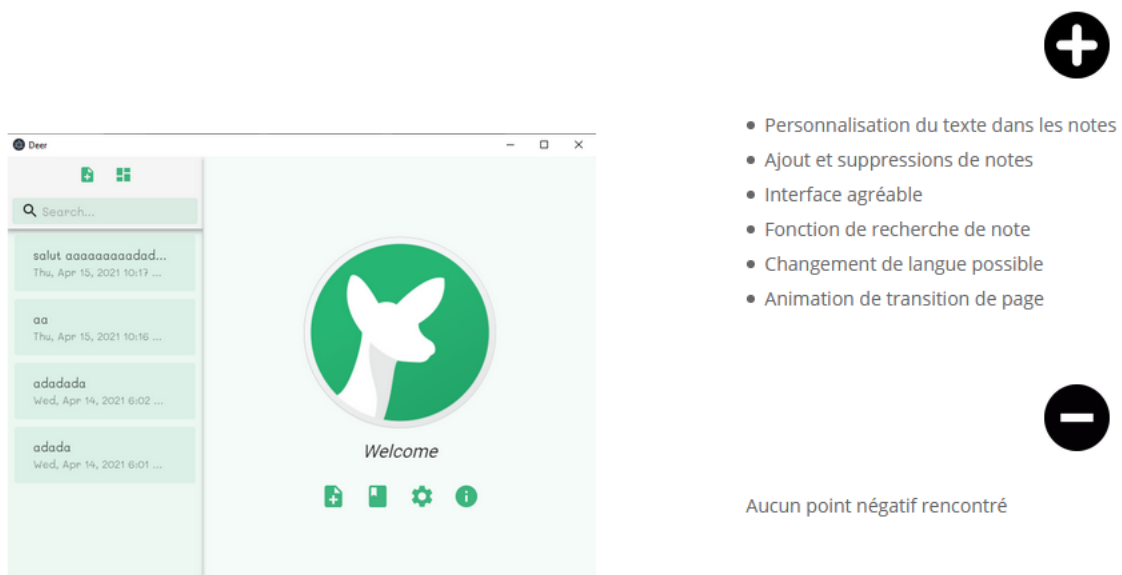


Figure 9 : Article pour/contre de l'application Deer

Il est parfois possible de tomber non pas sur un clone mais sur une application open source comme pour l'application Deer présentée dans la figure 9. Cela n'a pas été un problème dans ma démarche puisque je pouvais, si je le souhaitais, utiliser le code pour mon projet final. C'est d'ailleurs ce que j'ai fait avec l'application Deer.

1.2.1. Présentation de Deer

Deer est une application développée en JavaScript avec React sous Electron. Electron permet de créer un exécutable d'une application web. Deer permet la création et la sauvegarde de notes personnalisables. Ce sont des étudiants dans une université qui ont développé Deer et ils ont déposé le code sur GitHub. Je n'ai pas analysé Deer dans son intégralité car tout ne m'était pas utile dans l'application, je me suis focalisé sur les fonctionnalités qui me semblaient pertinentes.

1.2.1.1. Organisation des composants React

Comme pour toute application développée sous React, Deer est composée de composants React.

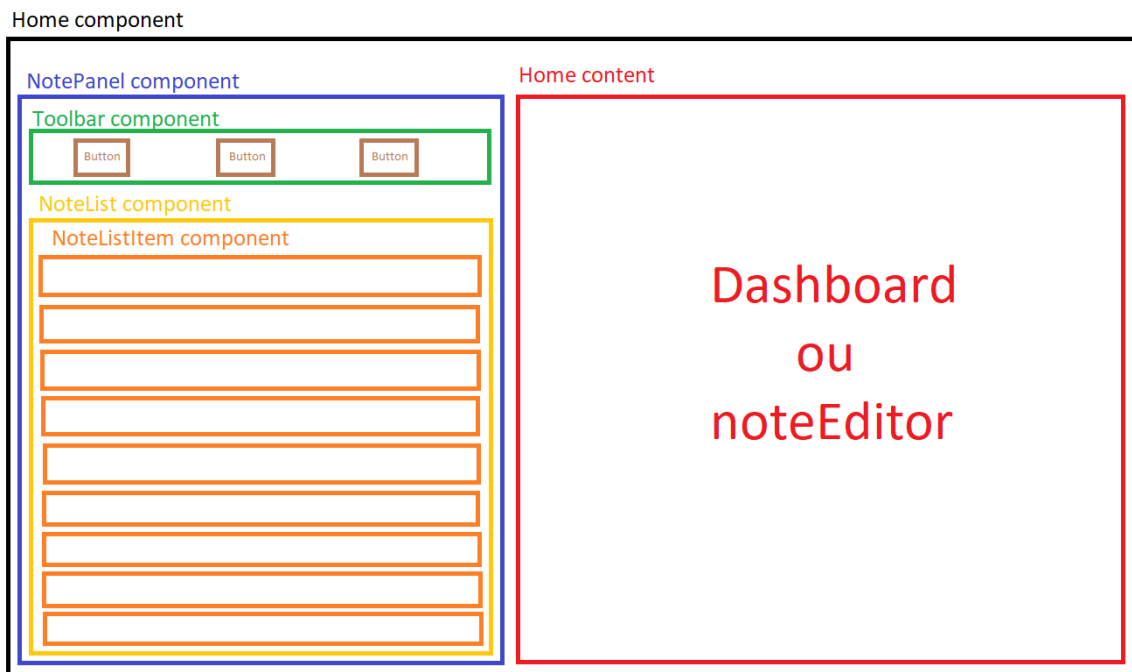


Figure 10 : Schéma de l'organisation des composants de l'application Deer

Tous les composants React de l'application sont organisés comme présenté dans la figure 10. Le composant "Home" en noir, est le composant de base de Deer, c'est à dire que c'est lui qui est appelé en premier par la page web. C'est donc à travers lui que toute l'application fonctionne. Il contient le composant notePanel, comme son nom l'indique, c'est le menu situé sur la gauche où sont stockées les notes enregistrées. De ce fait, ce composant est composé d'un composant noteList mais aussi d'un composant Toolbar qui permet à l'aide de ses boutons d'ajouter une nouvelle note ou de revenir au dashboard. Le composant noteList quant à lui, permet l'affichage des notes à travers le composant noteListItem.

```
return (
  <div className={classes.root}>
    <Scrollbars>
      <List component='nav' className={classes.list}>
        <FlipMove typeName={null}>
          {this.props.notes.map(note => (
            <NoteListItem
              key={note.id}
              id={note.id}
              text={note.title}
              modified={note.modified}
              selected={this.props.selectedNoteID === note.id}
              isInNoteBook={Boolean(this.props.noteBookNotes[note.id])}
              noteBookIsActive={this.props.activeNoteBookID !== 'none'}
              onClick={this.handleOnNoteSelect}
              onDelete={this.handleOnNoteDelete}
              onImportant={this.handleOnCustom}
              onNoteBook={this.handleOnNoteBookClick}
            />
          ))}
        </FlipMove>
      </List>
    </Scrollbars>
  </div>
)
```

Figure 11 : Return du composant noteList

Ce composant permet l'affichage d'une note et est appelé autant de fois qu'il y a de notes stockées comme montré dans la figure 11. Dans le composant Home, il y a le "Home content" qui n'est pas le seul contenu du Home comme nous venons de le voir mais c'est de cette manière qu'il est nommé dans le code. Il n'est pas réellement un composant mais permet d'afficher un composant, ici le dashboard ou le noteEditor. En effet, c'est en fait une méthode qui affiche l'un ou l'autre selon une condition. Si une note est sélectionnée, alors cela affiche le composant noteEditor afin que l'utilisateur puisse modifier sa note. Si aucune note n'est sélectionnée ou si l'utilisateur appuie sur le bouton menu situé dans la toolbar, alors le dashboard s'affiche. Dans cette application, d'autres composants sont présents comme settings ou noteBook mais n'ont aucune utilité pour mon projet.

1.2.1.2. Le stockage des données

Deer possède une base de données PouchDB afin de stocker les notes enregistrées. Pour la réalisation de mon projet, je devais changer de base de données, donc je ne me suis pas grandement attardé sur PouchDB.

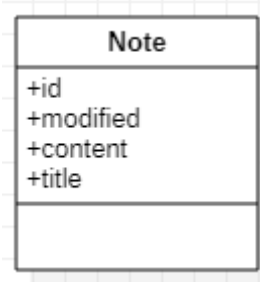


Figure 12 : Table Note

Néanmoins, dans Deer, comme présenté dans la figure 12, chaque note contient un id qui lui est propre, l'attribut "modified" correspond à la date de dernière modification et est affiché sur le composant noteListItem, l'attribut "content" contient le contenu de la note qui est affiché dans le composant noteEditor et enfin l'attribut "title" contient le titre de la note qui est affiché sur le composant noteListItem. La base de données est chargée au démarrage de l'application via un fichier qui lui est dédié "db.js". Ainsi, un "fetchAll" est réalisé juste après le chargement de la base afin d'obtenir les notes stockées dans la base. Une fois obtenues, elles sont affichées à l'aide du composant noteListItem où l'on remarque dans la figure 11 que des props (propriétés) sont insérés. Parmi ces props, on y trouve l'id, le titre et le "modified", ces derniers permettent le bon comportement du composant. En effet, pour l'affichage, le composant noteListItem a besoins titre et de la date de la dernière modification et pour l'évènement lorsque l'on clique dessus, il a besoins de l'id pour transmettre aux autres composants que c'est cet id qui est sélectionné afin d'afficher le bon contenu dans le composant noteEditor.

1.2.1.3. Utilisation de Deer

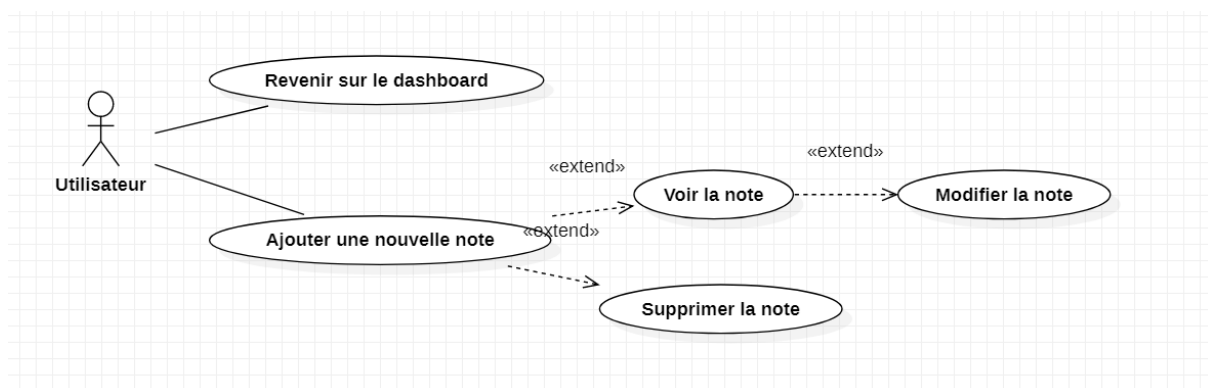


Figure 13 : Diagramme d'utilisation de Deer

La figure 13 permet de mieux comprendre l'utilisation de l'application. L'utilisateur peut dans tous les cas créer une nouvelle note ou accéder au tableau de bord. Si une note est présente dans la base de données car déjà créée au préalable, alors

l'utilisateur peut la voir ou la supprimer à l'aide d'un bouton présent sur le composant noteListItem. Une fois la note affichée dans le composant noteEditor, alors l'utilisateur peut modifier la note.

2. Cahier des charges

J'ai réalisé ce cahier des charges après coups car la réalisation du projet s'est réalisée de manière agile. J'avais à chaque fois quelques jours pour réaliser une ou plusieurs tâches. Après avoir fait part de mon travail, le maître de stage me donnait de nouvelles tâches. Je n'avais donc aucune idée de ce à quoi le projet allait ressembler à la fin du stage. De plus, régulièrement, je suis revenu en arrière sur le projet en changeant de base de donnée par exemple. Donc, pour ce cahier des charges, je ne vais mettre en évidence que les besoins fonctionnels et non fonctionnels de la version finale.

2.1. Analyse des besoins fonctionnel

Le résultat attendu est une application qui peut contenir des liens de site web ou des applications web stockées en locale, le tout regroupé dans un menu dynamique. Le projet doit répondre à certains critères, premièrement, elle doit disposer d'un système de connexion. L'utilisateur doit avoir le choix de pouvoir se connecter en ligne ou en local.

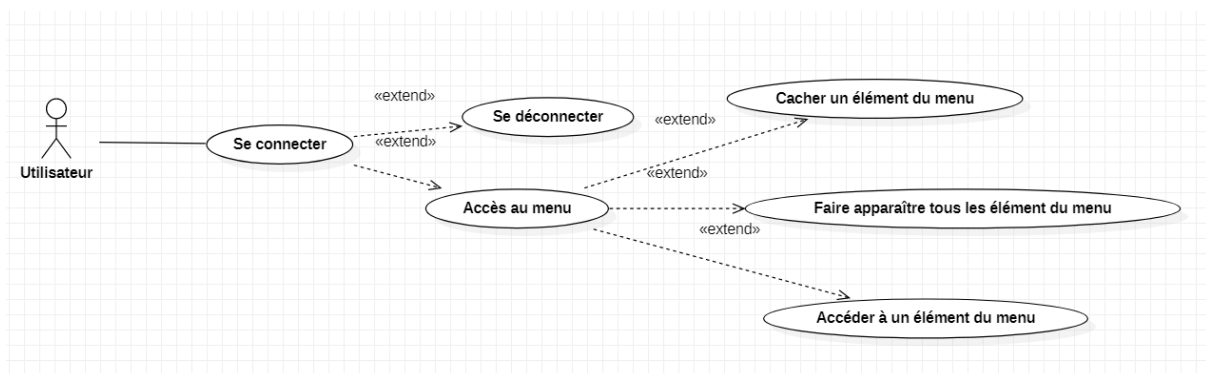


Figure 14 : Diagramme d'utilisation de l'application

Une fois connecté, comme indiqué dans la figure 14, l'utilisateur doit pouvoir accéder à un menu qui lui est propre et personnalisable, il peut cacher des items du menu ou tous les faire apparaître à l'aide de boutons. Également, le style de l'application et plus précisément de la page de connexion est personnalisable directement dans la base de données en ligne. Si l'utilisateur clique sur un item du menu, cela doit afficher le contenu de la page web en ligne ou stockée localement dans une webview présente dans l'application. Aussi, il doit y avoir un bouton qui permet à l'utilisateur de se déconnecter de l'application. Enfin, afin de rendre accessible

l'application localement comme avec internet, l'application doit synchroniser les deux bases de données quand l'utilisateur se connecte avec internet.

Il est aussi important de notifier que l'utilisateur ne peut s'inscrire seul, en effet l'application ne doit pas disposer d'un formulaire d'inscription. C'est à partir de la base de données en ligne (Firebase) qu'il doit être seulement possible d'ajouter un compte ou de modifier les données de ce dernier. Ainsi, il est possible qu'un utilisateur ait un compte mais qu'il ne possède pas de donnée. Dans ce cas, il faut que ce dernier se voit attribuer des données par défaut (style et menu). Pour ce qui est des comptes locales, ils sont aussi stockées dans les données en lignes en lien avec le compte, donc une fois que l'utilisateur s'est connecté au moins une première fois grâce à la méthode en ligne, alors la synchronisation doit avoir ajouté le compte locale dans la base de donnée locale pour permettre à l'utilisateur pour ses prochaines connexions de pouvoir se connecter avec son compte local doit être indépendant du compte en ligne mais doit permettre d'accéder aux mêmes données. De la même manière, l'application doit lire ses données uniquement sur la base de données locale afin qu'il n'y ait aucun problème lorsque l'utilisateur n'ait pas internet.

2.2. Analyse des besoins non fonctionnels

Le projet doit être codé en JavaScript et avec comme base une application existante sous Electron. Il doit comporter deux bases de données, une locale en SQLite et une en ligne avec Firebase. Les données stockées dans les bases de données doivent être au format JSON pour le bon fonctionnement avec la base de données RealtimeDB de Firebase. La connexion en ligne doit être possible par le système d'authentification Firebase mais aussi par KeyCloak. La base de données doit être compréhensible et facilement modifiable car c'est à partir de cette dernière qu'il est possible de personnaliser le menu comme le style ou l'authentification locale pour chaque utilisateur. Enfin, le design de l'application et en particulier des pages de connexion doit être travaillé afin de donner envie à l'utilisateur de poursuivre sur l'application.

3. Rapport technique

Mon projet réalisé pendant ce stage se doit de respecter le cahier des charges définis au préalable. Ce rapport technique va vous présenter la conception du projet ainsi qu'une description du code qui permet de répondre aux critères.

3.1. Conception

L'objectif de cette partie est de mettre en avant les technologies qui ont été choisies et évidemment de présenter les choix de conception pris lors de la réalisation de l'application.

3.1.1. Choix technologiques

Les technologies utilisées dans ce projet ont toutes été prédéfinies dans le cahier des charges. Donc, je n'ai pas eu vraiment le choix du langage pour coder l'application. En effet, comme précisé dans les besoins non fonctionnels du cahier des charges, afin de réaliser ce projet, j'ai dû me baser sur une application existante qui est codée en JavaScript. Ainsi après des recherches et une petite étude, j'ai décidé que je me baserai sur l'application Deer présentée dans l'analyse de l'existant. Deer est une application qui est codée en JavaScript avec le framework React, elle utilise le CSS pour son style. Donc naturellement, mon application est codée pareillement en JavaScript avec le framework React et également du css pour le style. De la même manière, l'application dispose d'Electron, il permet le développement d'un exécutable à partir de langage web. De ce fait, le React se transforme en quelque sorte en HTML pour qu'ensuite Electron puisse utiliser le HTML/CSS et JavaScript présent pour en faire une application utilisable sur Windows. Electron fonctionne avec nodeJS qui est évidemment présent sur l'application, nodeJS permet une grande flexibilité pour l'installation de nouvelles bibliothèques.

3.1.2. Conception des bases de données

Pour que l'application puisse fonctionner quand l'utilisateur ne possède pas internet, naturellement il a fallu mettre en place une base de données embarquée. Cependant, il était aussi nécessaire que la base de données soit modifiable en ligne donc une base de données en ligne a dû être mise en place. Tout comme pour les langages de programmation, les bases de données en ligne et locale ont été prédéfinies dans le cahier des charges. Une des bases de données embarquée les plus utilisées est SQLite, elle utilise le langage SQL pour exécuter les requêtes sur ses tables et possède une documentation bien documentée pour son implémentation avec nodeJS.

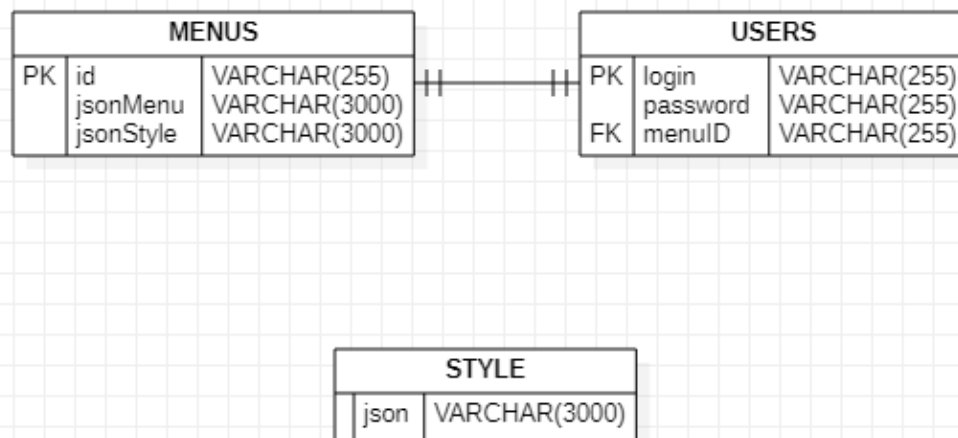
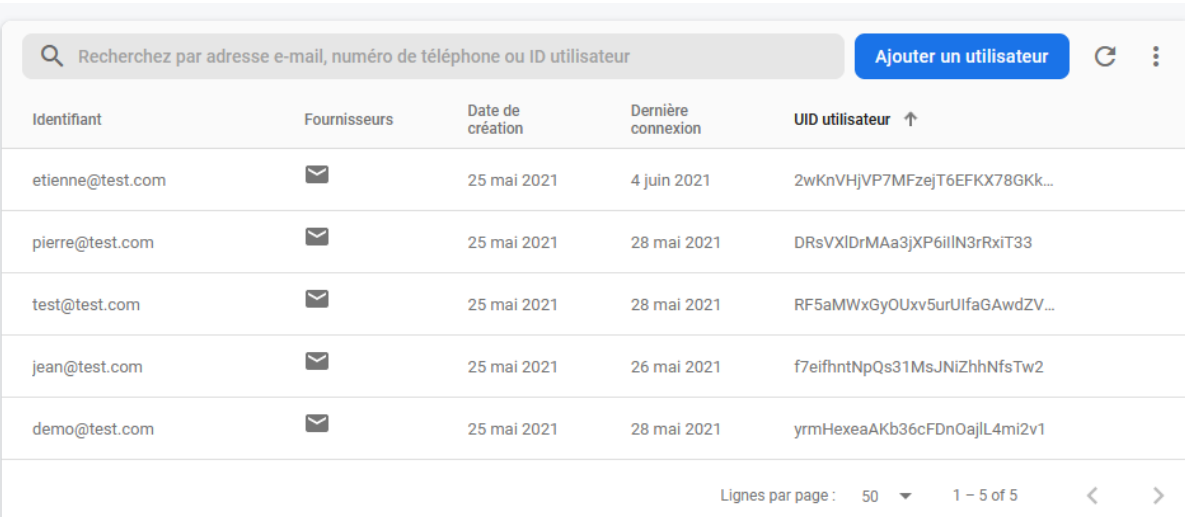


Figure 15 : Diagramme de la base de données embarquée (SQLite)

Dans le diagramme présenté dans la figure 15, la table **MENUS** sert à stocker les informations concernant l'application pour chaque utilisateur enregistré (style + menu). C'est dans "jsonMenu" qu'est stocké le menu de l'utilisateur au format JSON, pareillement, c'est dans "jsonStyle" qu'est stocké le style de l'application pour l'utilisateur au format JSON. La table **MENUS** contient également un `id` qui est une clé primaire, cet `id` est généré par le service d'authentification de la base de données en ligne que j'aborderai ultérieurement. Ainsi, pour chaque `id`, c'est-à-dire pour chaque utilisateur, il y a un style et un menu. Ensuite, dans la table **USERS** on y trouve un `login` en clé primaire ainsi qu'un `password` qui sont en fait des identifiants pour se connecter localement. Cette table contient aussi un "menuID" qui est une clé étrangère faisant référence à la clé primaire "id" dans la table **MENUS**. Cette clé étrangère permet lors de la connexion en locale, de mettre en relation les deux tables et donc d'obtenir les données (style + menu) pour l'utilisateur. Pour finir avec cette base de données locale, il y a une table **STYLE** qui permet simplement de stocker sous le format JSON le style actuel affiché dans l'application. Ainsi, si un utilisateur se connecte, son style sera chargé dans cette table. Pour ce qui est de la base de données en ligne, il m'a été demandé d'utiliser Firebase. Il s'agit d'un service proposé par Google qui permet de stocker des données en temps réel et propose aussi un service d'authentification.

Rapport de projet pour l'entreprise Opinaka : Menu personnalisable

Tillier Etienne



Identifiant	Fournisseurs	Date de création	Dernière connexion	UID utilisateur ↑
etienne@test.com	✉	25 mai 2021	4 juin 2021	2wKnVHjVP7MFzejT6EFKX78GKk...
pierre@test.com	✉	25 mai 2021	28 mai 2021	DRsVXIDrMAa3jXP6iilN3rRxiT33
test@test.com	✉	25 mai 2021	28 mai 2021	RF5aMWxGyOUxv5urUlfAGAwDZV...
jean@test.com	✉	25 mai 2021	26 mai 2021	f7eifhntNpQs31MsJNiZhhNfsTw2
demo@test.com	✉	25 mai 2021	28 mai 2021	yrmHexeaAKb36cFDnOajlL4mi2v1

Lignes par page : 50 1 - 5 of 5

Figure 16 : Service d'authentification Firebase

Ce service d'authentification présenté dans la figure 16 permet d'accéder à un panneau de gestion d'utilisateurs. On remarque qu'un utilisateur est défini par un identifiant ainsi qu'un UID. Cet UID est très important et nous allons le voir par la suite. Évidemment, l'utilisateur possède un mot de passe mais ce dernier n'est pas affiché dans le panneau de gestion. Ainsi ce système d'authentification permet à ceux qui ont accès à la console Firebase, de pouvoir créer ou supprimer des utilisateurs. Car je le rappelle, il est impossible de s'inscrire ou de se désinscrire directement depuis l'application.

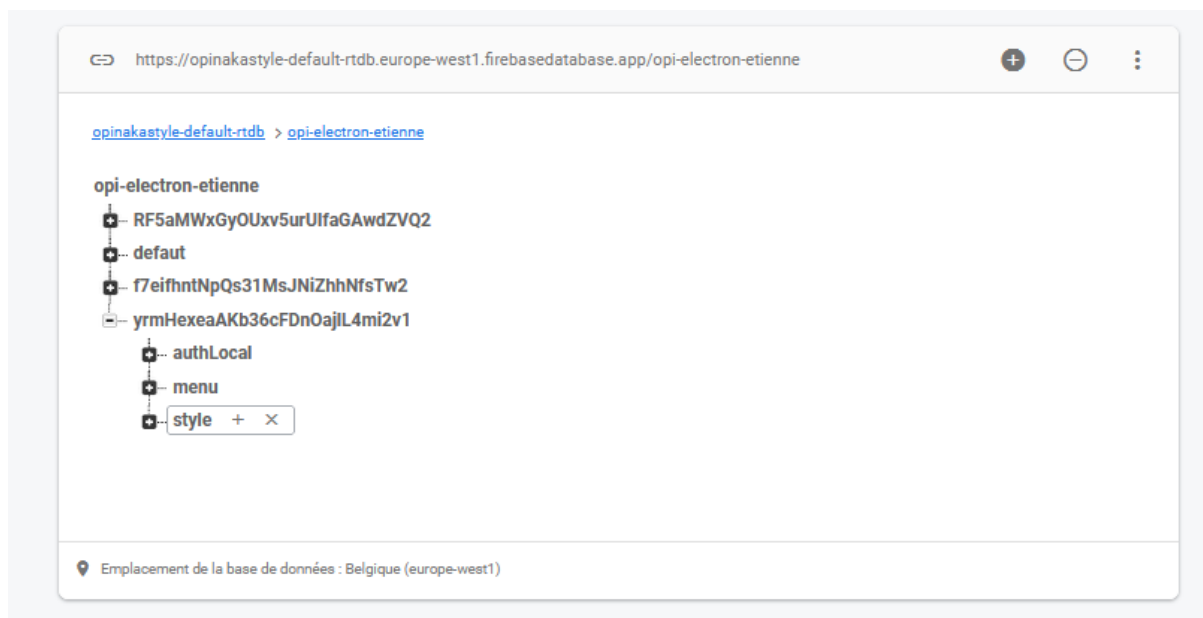


Figure 17 : Base de données en temps réel Firebase

Rapport de projet pour l'entreprise Opinaka : Menu personnalisable

Tillier Etienne

La base de données Firebase est au format JSON. La base de données en ligne pour mon application est présentée dans la figure 17, on remarque que le nom de l'objet, c'est à dire le nom de la base de données, s'appelle "opi-electron-etienne". C'est pour rendre l'application plus flexible et lui permettre de fonctionner avec différentes base de données car en changeant le nom dans le code, alors on pourrait accéder à une autre base (autre objet). Dans cet objet, sont présents les UID des utilisateurs. Ils ne sont pas tous présents, car dans la figure 16, il y a plus de comptes utilisateurs que d'IUD ici. Cela signifie que l'utilisateur peut se connecter avec le service d'authentification Firebase donc en ligne mais accèdera à un menu ainsi qu'un style par défaut que l'on peut voir dans la figure 17. Dans chaque UID ou dans défaut, il y a plusieurs autres objets.



Figure 18 : Base de données Firebase (authLocal)

L'objet "authLocal" présenté dans la figure 18, fait référence à l'authentification locale que l'on fournit à l'utilisateur qui se connecte en ligne. Cet objet est bien évidemment composé d'un login et d'un password et c'est ce qui sera en fait copié dans la table USERS de la base de donnée locale après une synchronisation que nous verrons plus tard. L'UID "défaut" contient également une authentification locale mais elle n'est pas utilisable pour se connecter.



Figure 19 : Base de données Firebase (menu)

L'objet "menu" présenté dans la figure 19, contient les données du menu de l'utilisateur. Cet objet est un tableau qui contient des items (ici un seul). Les items

Rapport de projet pour l'entreprise Opinaka : Menu personnalisable

Tillier Etienne

sont composés d'un "id" pour le bon fonctionnement avec l'application, d'un statut pour savoir s'il doit s'afficher ou pas et pour permettre à l'utilisateur de cacher l'item ou de le faire réapparaître, d'un "title" qui est le titre de l'item et enfin d'un "url" qui correspond soit à un vrai url pour accéder à un site web classique ou alors qui correspond à "#" afin d'accéder à un site web stocké en local avec pour dossier ayant comme nom le "title".

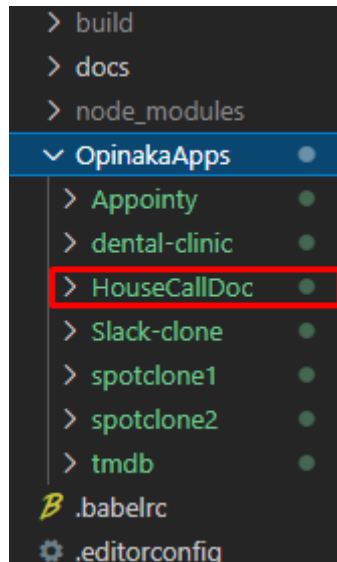


Figure 20 : Dossier stockant les sites web en local.

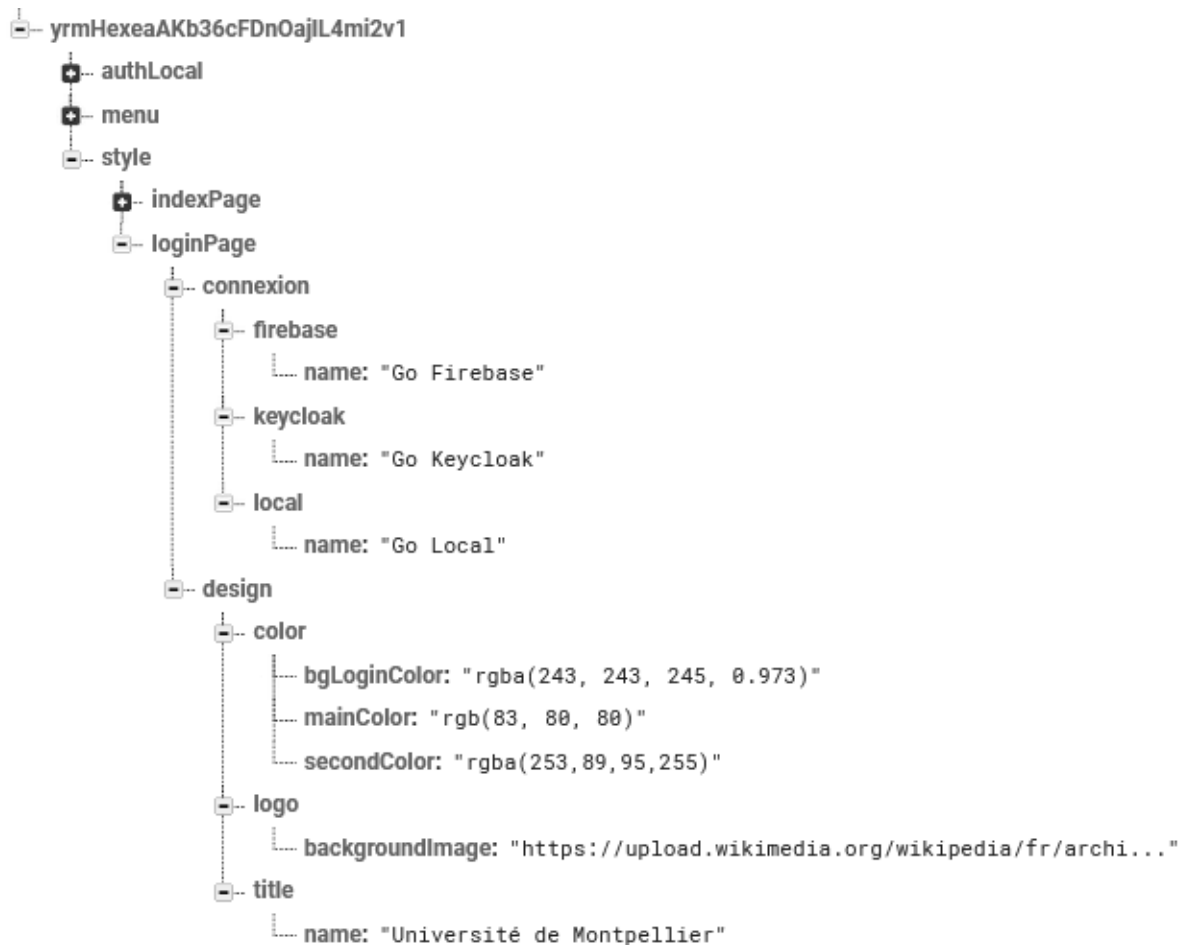


Figure 21 : Base de données Firebase (style)

L'objet "style" contient le style personnalisé de l'utilisateur qui sera chargé dans l'application à sa connexion. Cet objet est séparé en deux sous-objets, il s'agit d'une architecture qui m'a été imposée afin de généraliser plusieurs bases de données entre différents stagiaires. Or, je ne me suis servi que de l'objet "loginPage", donc je n'aborderai pas la partie "indexPath" que je n'ai pas utilisé. Dans l'objet "loginPage", on retrouve "connexion" qui contient les différents moyens de connexion possible pour l'application. Cela fait finalement référence aux boutons de connexion présents dans la page de connexion. Le "name" de chacun de ces boutons est personnalisable. Dans la partie "design", il y a un objet "color" qui contient un thème de trois couleurs qui sont modifiables. La partie "logo" permet de changer le logo de l'application. Et pour finir, la partie "title" permet de changer le titre de la page. Afin qu'un utilisateur puisse obtenir les mêmes données avec internet ou sans internet, j'ai dû concevoir une synchronisation des deux bases. Cette dernière a lieu lorsqu'un utilisateur se connecte avec internet localement ou en ligne. Ainsi, les données relatives à son compte utilisateur stockées dans la base de données Firebase sont insérées dans la base de données embarquée SQLite.

3.1.3. Conception des algorithmes

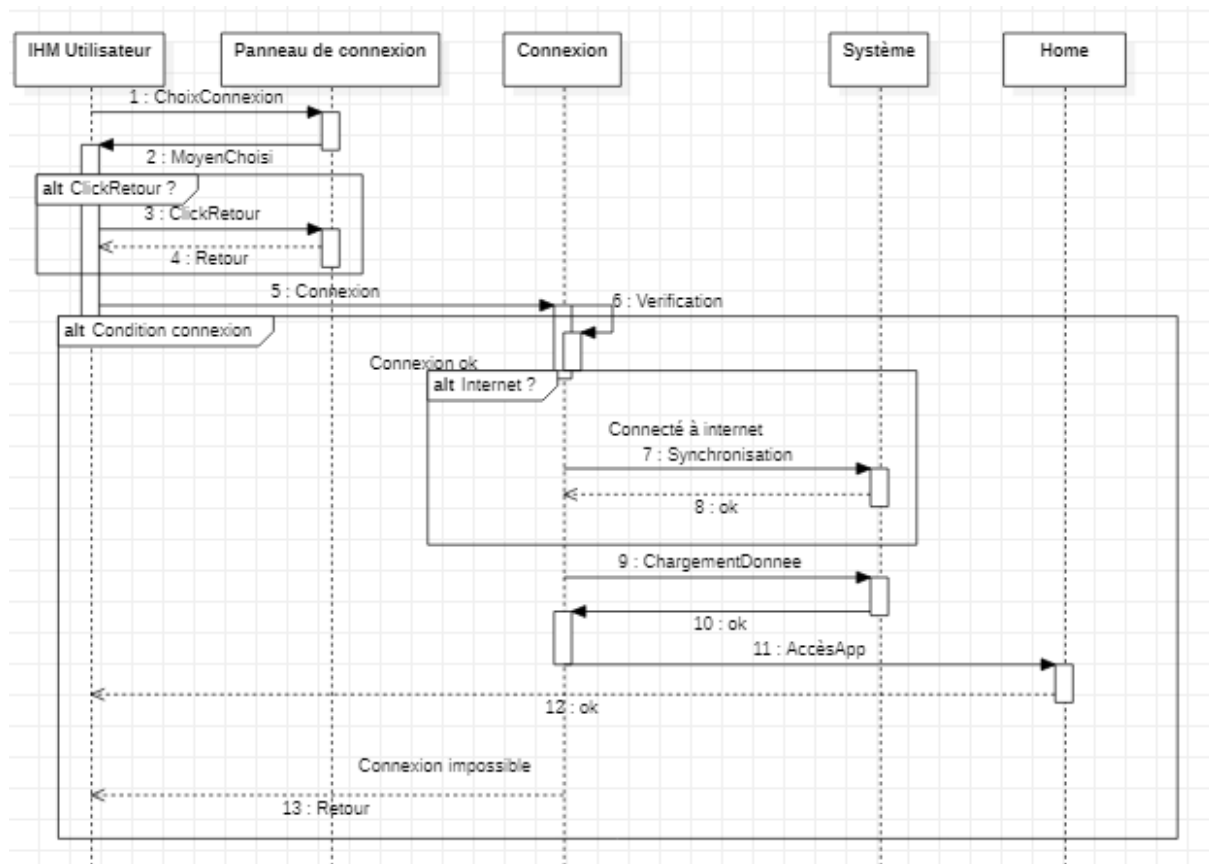


Figure 22 : Diagramme de séquence de la connexion à l'application

Le diagramme de séquence présenté dans la figure 22 met en avant le déroulement de la connexion à l'application d'un utilisateur. Premièrement, au démarrage de l'application, l'utilisateur se trouve sur un panneau de connexion proposant différents moyens de connexion. Pour en choisir un, il appuie sur un bouton qui active ChoixConnexion(1), ce qui renvoie l'utilisateur sur la page de connexion avec le moyen choisi. L'utilisateur peut à ce moment-là, s'il le souhaite, appuyer sur un bouton retour qui active ClickRetour(3) afin de revenir à la page précédente. S'il veut se connecter, alors il rentre des identifiants et appuie sur le bouton de connexion ce qui active Connexion(5) qui va entraîner une vérification des identifiants. Si les identifiants sont valides, alors le système doit vérifier si l'utilisateur est connecté à internet, si c'est le cas alors le système procède à la synchronisation des données utilisateurs par Synchronisation(7). Avant d'accéder au menu, il faut que les données soient chargées, c'est pourquoi le système fait un chargementDonne(9) pour qu'enfin l'utilisateur puisse accéder au composant home via AccèsApp(11). Dans le cas où les identifiants n'étaient pas valables, alors l'utilisateur serait revenu au panneau de connexion choisi.

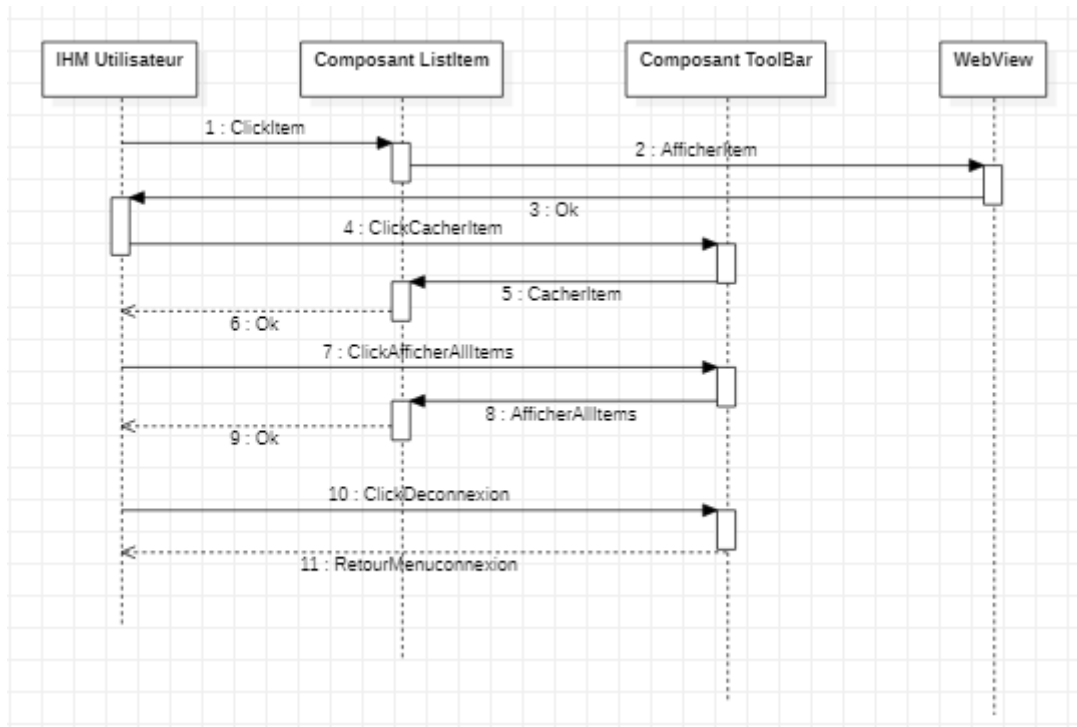


Figure 23 : Diagramme de séquence de l'utilisation de l'application

A travers le diagramme de séquence dans la figure 23, est présenté le déroulement des différentes méthodes durant l'utilisation de l'application. Une fois que l'utilisateur accède à l'application, c'est à dire au composant "Home", il a accès à un menu qui est le composant "ListItem", une barre d'outils qui est le composant "ToolBar" et enfin une webview qui permet d'afficher des sites webs. L'utilisateur peut alors cliquer sur un item de la ListItem ce qui active ClickItem(1) qui affiche le contenu de l'item dans la WebView par la méthode AfficherItem(2). Une fois un item sélectionné, il est possible de le cacher en appuyant sur un bouton qui active ClickCacherItem(4) qui permet d'activer CacherItem(5) qui cache l'item dans la ListItem. Avec les boutons présents dans la ToolBar, il est possible en cliquant sur l'un d'entre eux d'activer ClickAfficherAllItem(7) qui permet de faire réapparaître tous les items présents dans la ListItem par la méthode AfficherAllItems(8). Enfin, il est possible de se déconnecter de l'application par la méthode ClickDeconnexion(10) qui ramène l'utilisateur à l'interface pour se connecter.

3.2. Réalisation

C'est à travers cette partie que vous trouverez une description détaillée des principales fonctionnalités de l'application.

3.2.1. Réalisation de la page de connexion

Lorsque l'utilisateur démarre l'application, il se retrouve sur une page présentée dans l'annexe ? qui lui propose trois moyens de connexion qui sont Firebase, KeyCloak et Hors-Ligne. Le bouton KeyCloak est désactivé car ce moyen de connexion n'est pas compatible avec Électron et donc ne fonctionne pas correctement. L'utilisateur peut alors choisir en cliquant sur un bouton parmi les trois. Pour la réalisation de la partie connexion, premièrement, ce qu'il faut savoir, c'est que dans l'application de base (Deer), il n'y avait pas de page de connexion. Donc tout a été conçu et réalisé par moi sur cette partie.

```
return (  
  <div className="loginMenu" style={{textAlign: 'center'}}>  
    {logChoice == "" ? ( styleData != "" ?  
      <div className="loginPage">  
        <div className="loginChoice">  
          <img className="imglogin" src={{(onLine ? styleData.loginPage.design.logo.backgroundImage : require('../../assets/images/Opinaka.png'))}} />  
          <h2>{styleData.loginPage.design.title.name}</h2>  
          <div className="divButton">  
            <div disabled={{(onLine ? false : true)}}  
              className={classNames("buttonConnexion", (onLine ? "" : "disabled"))}  
              variant="contained"  
              color="primary"  
              onClick={{(onLine ? () => buttonClicked("firebase") : () => console.log("no connexion"))}}  
              {styleData.loginPage.connexion.firebase.name}  
            </div>  
            <div className={classNames("buttonConnexion", "disabled")} variant="contained" color="secondary" /*onClick={{() => buttonClicked("keyCloak")}}*/>  
              {styleData.loginPage.connexion.keycloak.name}  
            </div>  
            <div className="buttonConnexion" variant="contained" color="inherit" onClick={{() => buttonClicked("offline")}}>  
              {styleData.loginPage.connexion.local.name}  
            </div>  
          </div>  
        </div>  
      </div>  
    : "" ) : (logChoice == "firebase" ?  
      <LoginFireBase  
        isNewUser={isNewUser}  
        styleData={styleData}  
        idMenu={idMenu}  
        setIdMenu={setIdMenu}  
        setIsNewUser={setIsNewUser}  
        internet={onLine}  
        setlogChoice={setlogChoice}/>  
      : (logChoice == "keyCloak" ?  
        <LoginkeyCloak  
          isNewUser={isNewUser}  
          styleData={styleData} idMenu={idMenu}  
          setIdMenu={setIdMenu}  
          setIsNewUser={setIsNewUser}  
          internet={onLine}  
          setlogChoice={setlogChoice}/>  
        : <LoginOffline  
          isNewUser={isNewUser}  
          styleData={styleData}  
          idMenu={idMenu}  
          setIdMenu={setIdMenu}  
          setIsNewUser={setIsNewUser}  
          internet={onLine}  
          setlogChoice={setlogChoice}/> ) )  
    </div>  
  );  
);
```

Figure 24 : Return du composant LoginMenu

Dans la figure 24 est présenté le return du composant LoginMenu. On peut remarquer qu'il est séparé en deux parties, en effet, beaucoup de conditions sont présentes dans ce return. Dans ce composant, il y a un useState qui se nomme "logChoice" qui contient le choix de l'utilisateur pour se connecter. Par défaut ce useState est initialisé comme nul. La première condition présente en haut du rectangle rouge vérifie que logChoice ne contient aucune valeur, si c'est le cas, la

condition vérifie que le style de l'utilisateur est bien chargé à l'aide d'un autre `useState` `styleData` que nous verrons plus tard. Si ces deux conditions sont vérifiées, cela signifie que l'utilisateur n'a pas encore choisi mais que le style est chargé auquel cas il faut afficher la panneau de connexion permettant à l'utilisateur de choisir. Ce panneau est dans l'encadré rouge où l'on peut remarquer trois rectangles jaunes qui correspondent aux trois boutons permettant de choisir. Ces boutons sont représentés par des `div`, ils possèdent tous les trois la même classe `"bouttonConnexion"` pour le style et évidemment une méthode `"onClick"` permettant de mettre à jour le `useState` `logChoice` avec la valeur du bouton en question. Également, certains bouton comme le bouton `"Firebase"` peuvent contenir plus d'une classe, en effet si l'utilisateur possède internet ou non, un `useState` `onLine` est mis à jour et dans ce cas, si l'utilisateur n'a pas internet, alors le bouton est désactivé en lui ajoutant un attribut et une classe pour le style.

```
.disabled {  
  background: grey !important;  
  color: black !important;  
}
```

Figure 25 : Classe disabled CSS

Le texte présent dans la `div` qui est finalement le texte qui est affiché dans le bouton dépend comme présenté dans la conception, de ce qui est entré dans le style chargé dans la table `Style`. Dans le cas où l'utilisateur clique sur un bouton, le `useState` `loginChoice` est mis à jour est donc la première condition n'est plus vérifié et n'affiche donc plus cette page. Elle va à la place afficher un composant `React` en fonction du mode de connexion choisi représenté dans les encadrés verts. Je passe à ces composants toutes les propriétés que l'on appelle `"props"` pour que le composants puisse fonctionner correctement. On peut voir que je passe par exemple la fonction qui permet de modifier `LogChoise` qui se nomme `setLogChoice`, `internet` qui est un `boolean` qui permet de savoir si l'utilisateur est connecté à internet ou non, `idMenu` et sa méthode `setIdMenu` qui permettent de savoir et de mettre à jour l'`UID` pour que les bonnes données soient affichées, ou encore `styleData` qui permet de transférer le style au format `JSON` entre les composants.

```
return (  
  <div className="loginFirebase" style={{textAlign: 'center'}}>  
    {isSignedIn ? (  
      <Home styleData={props.styleData} idMenu={props.idMenu} internet={props.internet} setIsSignedIn={setIsSignedIn}/>  
    ) : (  
      <div className="loginPage">  
        <div className="loginChoice">  
          <img className="imgLogin" src={((props.internet ? props.styleData.loginPage.design.logo.backgroundImage : require('../../assets/images/Opinaka.png')))} />  
          <h2>{props.styleData.loginPage.design.title.name} Connexion Firebase</h2>  
          <div className="divButton">  
            <input placeholder="Login" id="login" name="login" type="email" onChange={(e) => {handleLoginChange(e)}} />  
            <input placeholder="Mot de passe" id="password" name="password" type="password" onChange={(e) => {handlePasswordChange(e)}} />  
            <div id="error" style={{color:'red'}}>{error}</div>  
            <div className="buttonConnexion" variant="contained" color="primary" onClick={handleCheckLogin}>  
              Se connecter  
            </div>  
            <div className="buttonConnexion" variant="contained" color="secondary" onClick={() => returnButton()}>  
              Retour  
            </div>  
          </div>  
        </div>  
      </div>  
    )  
  )  
);
```

Figure 26 : Return du composant LoginFirebase

Pareillement que dans la figure 24, le return du composant LoginFirebase présenté dans la figure 26 est divisé en deux parties. En effet, ce composant possède lui aussi un useState qui se nomme isSignedIn qui comme son nom l'indique, vérifie si un utilisateur est connecté. Si un utilisateur est connecté, alors il renvoie vers le composant Home qui est le point de départ afin d'utiliser l'application. Si ce n'est pas le cas, alors, il renvoie vers un formulaire de connexion classique qui utilisera dans ce composant des méthodes différentes de vérification puisqu'il s'agit du LoginFirebase. Je tiens à préciser que les composants de connexion sont assez semblables, seulement leurs méthodes de vérification varient. Après avoir rentré ses identifiants, l'utilisateur peut alors appuyer sur le bouton "se connecter" qui active la méthode "connexionFirebase".

```
const handleCheckLogin = () => {  
  console.log('login = ' + login + " password = " + password);  
  connexionFirebase(login,password,displayConnexion)  
}
```

Figure 27 : Méthode handleCheckLogin

```
const connexionFirebase = (login,password,callback) => {  
  firebase.auth().signInWithEmailAndPassword(login, password)  
    .then((userCredential) => {  
      if (userCredential){  
        console.log("userhere");  
        //Signed in  
        checkerConnexion(firebase.auth().currentUser.uid).then((value) => {  
          if (value == true){  
            console.log("userid " + firebase.auth().currentUser.uid);  
            callback(true,firebase.auth().currentUser.uid);  
          }  
          else {  
            callback(true,"default");  
          }  
        })  
      }  
    })  
    .catch((error) => {  
      console.log("nouser");  
      callback(false);  
    });  
}
```

Figure 28 : Méthode connexionFirebase

Cette méthode présentée dans la figure 28 contient trois paramètres qui sont les identifiants de connexion ainsi qu'un callback. La bibliothèque de firebase installée avec nodeJS permet de tester la connexion à l'aide de méthodes fournies. Si un utilisateur correspond, alors dans tous les cas le callback sera renvoyé avec la valeur "true" mais en deuxième paramètre, il faut l'UID et si l'utilisateur n'est pas présent dans la base de donnée Firebase alors il se verra attribuer l'UID "default". C'est ce que je vérifie à l'aide de la méthode "checkerConnexion". Bien évidemment, si la connexion avec Firebase échoue, alors le callback prendra la valeur "false". Le callback qui est utilisé en paramètre de cette méthode est la méthode "displayConnexion".

```
const displayConnexion = function(bool,id) {  
  if (bool) {  
    if (props.internet){  
      console.log("internettt");  
      syncBD(id).then(() => {  
        console.log("propss" + JSON.stringify(props))  
        props.setidMenu(id);  
        loadStyleUser(id).then(() => {  
          props.setisNewUser(props.isNewUser + 1);  
          setError("");  
          setIsSignedIn(true);  
        })  
      })  
    }  
  }  
  else {  
    console.log(" no internettt");  
    console.log("propss" + JSON.stringify(props))  
    props.setidMenu(id);  
    props.setisNewUser(props.isNewUser + 1);  
    setError("");  
    setIsSignedIn(true);  
  }  
}  
else {  
  setIsSignedIn(false);  
  setPassword("");  
  setLogin("");  
  document.getElementById("login").value = "";  
  document.getElementById("password").value = "";  
  setError("Mauvaise combinaison de mot de passe et de login");  
}  
}
```

Figure 29 : Méthode displayConnexion

La méthode displayConnexion présentée dans la figure 29, permet en fonction de ses paramètres, d'accéder à l'application ou non. En effet, si le "bool" renvoyé est "true", alors, dans un premier temps on va vérifier si l'utilisateur possède internet pour savoir s'il faut réaliser une synchronisation des bases de données. Dans tous les cas, si la connexion a réussi et donc que la valeur est "true", alors le useState idMenu va prendre comme valeur l'id renvoyé par la méthode connexionFirebase, le useState isNewUser va aussi être modifié, il s'agit d'un useState pour le style, donc je ne vais pas en parler ici. Enfin et c'est le plus important, le useState isSignedIn va prendre la valeur "true" et donc indiquer au composant qu'un utilisateur est connecté. Ainsi, dans le return, il ne va plus afficher le formulaire de connexion mais le composant home. Bien évidemment, si la valeur de "bool" est "false", dans ce cas, l'utilisateur reste sur le formulaire car le useState isSignedIn ne bouge pas et un message d'erreur s'affiche.

3.2.2. Réalisation de la synchronisation des bases de données

Comme on a pu le voir précédemment, lorsqu'un utilisateur accède à l'application et qu'il est connecté à internet, il y a une synchronisation qui, comme on a pu le voir, est définie dans une méthode "syncBD".

```
const syncBD = (id) => {  
  return new Promise((resolve, reject) => {  
    syncMenu(id).then((value) => {  
      if (value){  
        syncUsers(id).then(() => {  
          resolve(true);  
        });  
      }  
    })  
  })  
}
```

Figure 30 : Méthode syncBD

Dans la figure 30, on peut voir que la méthode syncDB fonctionne avec une promesse comme beaucoup de mes méthodes dans ce projet. J'ai procédé de cette manière afin de pouvoir garder le contrôle sur l'ordre d'exécution des méthodes dans le projet. Dans cette méthode qui prend en paramètre l'UID de l'utilisateur, par exemple, elle va commencer par synchroniser le menu de l'utilisateur et une fois terminé, elle va synchroniser son authentification locale.

```
const syncMenu = (id) => {
  return new Promise((result, reject) => {
    var query = firebase.database().ref(dataTable + "/" + id);
    console.log("entre syncMenu");
    query.once("value", (snapshot) => {
      if (snapshot.val()){
        console.log("ok syncMenu");
        let sqlDel = "DELETE FROM MENUS WHERE id = ?";
        db.run(sqlDel,[id],(err) => {
          if (err){
            console.log(err);
            reject(false);
          }
          else {
            console.log("okok");
            let sqlInsert = 'INSERT INTO MENUS (id,json,jsonStyle) VALUES(?,?,?);';
            db.run(sqlInsert,[id,JSON.stringify(snapshot.val().menu),JSON.stringify(snapshot.val().style)]);
            result(true);
          }
        });
      }
      else {
        reject(false)
      }
    })
  })
}
```

Figure 31 : Méthode syncMenu

La méthode syncMenu qui est présentée dans la figure 31 va tout simplement copier le menu et le style en fonction de l'id entré en paramètre dans la table MENUS de la base de données locale SQLite. Pour cela, elle instancie une variable "query" qui sera l'adresse en quelque sorte dans la base de données Firebase de l'objet de l'UID faisant référence à l'id en paramètre. Ainsi, grâce aux méthodes de la bibliothèque Firebase, il est possible de récupérer les données à cette adresse qui seront stockées dans la variable "snapshot". La méthode vérifie ensuite si la variable snapshot n'est pas nulle, si c'est le cas, cela signifie que les données ont bien été récupérées et donc qu'elles vont pouvoir être copiées dans la base de données locale. Avant de les copier, pour éviter des doublons, la méthode supprime le tuple faisant référence à l'id en paramètre à l'aide d'un simple DELETE en SQL. Les méthodes SQLite fonctionnent aussi avec des promesses, cela permet justement d'attendre que le tuple soit supprimé avant d'insérer le nouveau tuple avec les données mise à jour. Pour insérer le tuple mis à jour, la méthode procède comme pour la suppression à l'aide d'un INSERT en SQL. On peut remarquer que les données insérées sont rendues en String à l'aide de "JSON.stringify" car malgré que la base de données Firebase soit un objet en JSON, lorsque l'on récupère des données dans celle-ci, elle les renvoie en objet JavaScript. Cependant, dans la base de données locale, les données sont des VARCHAR qui sont en fait des données au format JSON, donc il est nécessaire de les remettre dans leur format d'origine. La méthode syncUser qui permet de rentrer les données dans la table USERS fonctionne de la même manière que la méthode syncMenu.

3.2.3. Lecture et modification des données dans la base de données locale

Sauf pour la synchronisation, l'application fonctionne uniquement avec la base de données locale SQLite. Ainsi, il y a des méthodes qui permettent de lire et modifier les données de cette dernière. Par exemple, afin de générer le menu de l'application en fonction de l'utilisateur, il est important au préalable de posséder les données requises.

```
const fetchItems = (id) => {  
  console.log("id" + id);  
  return new Promise((result, reject) => {  
    var queries = [];  
    let sql = "SELECT json FROM MENUS WHERE id = ?"  
    db.get(sql,[id] ,(err,row) => {  
      console.log("ouiiiiii");  
      if (row) {  
        console.log("rowww = " + row.json);  
        let data = JSON.parse(row.json);  
        console.log("ici ?");  
        for(let i = 0; i < data.length; i++){  
          console.log("row : " + data[i]);  
          queries.push(data[i]);  
        }  
        console.log("queries = " + queries)  
        result(queries); // resolve the promise  
      }  
      else {  
        console.log("error = " + err);  
        console.log("row = " + row);  
      }  
    })  
  });  
}
```

Figure 32 : Méthode fetchItems

Dans la figure 32 est présentée la méthode fetchItems qui permet de récupérer dans la table MENUS en locale les items du menu en fonction de l'id entré en paramètre. Cette méthode renvoie une promesse contenant un tableau contenant tous les items du menu. Pour cela, elle instancie une variable "sql" qui est la base de la commande SQL où seulement l'id peut varier. Ensuite, elle exécute cette commande à l'aide de "db.get" où elle rentre l'id en paramètre de la fonction à l'intérieur pour qu'il soit pris en compte dans le WHERE. Une fois que le SELECT est fait, elle vérifie si des valeurs sont parvenue en testant si row est nul ou pas. Si row n'est pas nul, alors des valeurs ont été sélectionnées, le problème est qu'il s'agit d'une chaîne de caractère qui est en fait du JSON. Elle instancie donc une variable data qui convertit le JSON en objet JavaScript pour ensuite faire une boucle sur le tableau d'item afin de les ajouter un à un au tableau "query" qui sera renvoyé avec la promesse.

L'application permet aussi de cacher ou de faire réapparaître des items du menu. La visibilité des items dépend de la valeur de leur statut présent dans la base de données. Donc pour faire varier leur visibilité, il faut faire varier le statut et donc modifier les données dans la base de données.

```
const cacherOnline = (item,id) => {
  const newData = {
    status: "0"
  }
  var query = firebase.database().ref(dataTable + "/" + id);
  console.log("queryyy = " + query);
  query.once("value", (snapshot) => {
    console.log("snap " + JSON.stringify(snapshot));
    //snapshot.val().ref("menu").orderByChild("id").equalTo(note._id).once("child_added", (value) => {
    //value.ref.update(newData);
    // })
    let newQuery = snapshot.ref.child("menu").orderByChild("_id").equalTo(item._id);
    newQuery.once("child_added", (snap) => {
      snap.ref.update(newData);
    })
  });
}

const cacher = (item,id) => {
  return new Promise((result, reject) => {
    let itemCache = item;
    let stringCondition = "";
    let setSql = [];
    itemCache.status = "0";
    fetchItems(id).then((value) => {
      stringCondition = JSON.stringify(value);
      console.log("bd & " + stringCondition);
      for (let item of value){
        if (item._id == itemCache._id){
          setSql.push(itemCache);
        }
        else {
          setSql.push(item);
        }
      }
      let sql = `UPDATE MENUS ` +
        `SET json = ? ` +
        `WHERE json = ?`;
      console.log("base update : " + JSON.stringify(setSql))
      db.run(sql,[JSON.stringify(setSql),stringCondition]);
      console.log("BDD updated");
      isOnline(cacherOnline(item,id),() => {console.log("no internet")});
      result(true);
    })
  })
}
```

Figure 33 : Méthodes cacher et chacherOnline

La méthode cacher qui est présentée dans la figure 33 est appelée suite à l'appuie sur un bouton afin de cacher un item dans la liste d'item du menu. Cette méthode prend en paramètre l'item ainsi que son id. Etant donnée que dans la base de donnée, l'item est stockée dans la chaîne de caractères jsonMenu dans la table MENUS, il est impossible de la modifier directement à partir d'une commande SQL.

Ainsi il est nécessaire de modifier directement la chaîne de caractère, et pour cela, la méthode va se servir de la méthode `fetchItems` afin de récupérer tous les items et de les ajouter un à un dans un tableau sauf pour celui qui doit être caché et qui lui se verra modifié avant d'être ajouté au tableau. Ainsi, il ne reste plus qu'à transformer le tableau en une chaîne de caractère au format JSON et d'update le tuple qui correspond à la liste d'item du `fetchItem` en mettant à jour avec la nouvelle chaîne de caractère. Ensuite, cette méthode vérifie si l'utilisateur est connecté à internet via `"isOnline"`, si c'est le cas elle va mettre à jour la base de donnée Firebase afin que lors des prochaines synchronisations, le choix de l'utilisateur puisse être sauvegardé. Pour cela elle utilise la bibliothèque de Firebase comme pour la synchronisation, cependant, pour la mise à jour, elle utilise la méthode `update` où l'on met en paramètre les attribut et leur valeur qui doit être mis à jour, ici c'est `"newData"`. Pour ce qui est de faire réapparaître les items, ce sont quasiment les mêmes fonctions avec une qui se lance et appelle la version en ligne s'il y a une connexion à internet.

3.2.4. Génération du menu

Le menu est généré en fonction du `useState` `idMenu` qui est transféré de composant en composant.

```
render () {
  console.log(this.props.notes);
  const { classes } = this.props
  return (
    <div className={classes.root}>
      <Scrollbars>
        <List component='nav' className={classes.list}>
          <FlipMove typeName={null}>
            {this.menuRender()}
          </FlipMove>
        </List>
      </Scrollbars>
    </div>
  )
}
```

Figure 34 : Return du composant `ListItem`

```
this.menuRender = () => {  
  console.log("props : " + JSON.stringify(this.state.listItems))  
  return this.state.listItems.map(item => (  
    (item.status == 1 ? // savoir s'il faut afficher ou pas  
    <Item  
      key={item._id}  
      id={item._id}  
      text={item.title}  
      selected={this.props.selectedNoteID === item._id}  
      isInNoteBook={Boolean(this.props.noteBookNotes[item._id])}  
      noteBookIsActive={this.props.activeNoteBookID !== 'none'}  
      onClick={this.handleOnNoteSelect}  
      onDelete={this.handleOnNoteDelete}  
      onImportant={this.handleOnCustom}  
      onNoteBook={this.handleOnNoteBookClick}  
    />  
    : null)  
  ))  
}
```

Figure 35 : Méthode menuRender

C'est à partir du composant ListItem qu'est généré le menu de l'application. Le render de ce composant est présenté dans la figure 34, où finalement on s'aperçoit qu'il fait appel à une méthode pour afficher le menu que l'on retrouve dans la figure 35. Cette méthode menuRender parcourt un useState listItems qui est au préalable rempli à l'aide d'un fetchAllItems. Ainsi, pour chaque items présent dans ce useState, cette méthode return un nouveau composant Item auquel elle passe en paramètre l'id, le titre ainsi que de nombreuses méthodes qui permettent le bon fonctionnement. Ce qui est important et qui permet la flexibilité de ce menu, c'est qu'avant de générer un nouveau composant, elle vérifie le statut de l'item. S'il est égale à 0, alors elle ne génère rien, sinon elle le génère. Etant donné qu'il s'agit d'un useState et que je travaille avec React, si le useState vient à varier suite à une mise à jour, alors la fonction va être de nouveau exécutée et donc va actualiser le menu.

3.2.5. Réalisation du style personnalisable

Dans cette application, le style est en partie personnalisable depuis la base de données en ligne. On peut donc retrouver un style différent pour chaque utilisateur. Comme on l'a vu précédemment, dans la base de données locale, le style est synchronisé dans la table MENUS au format JSON. Il existe aussi une table STYLE où est inséré le style qui doit être pris en compte par l'application. Pour insérer le style dans cette table, il y a une méthode.

```
const loadStyleUser = (id) => {
  return new Promise((result, reject) => {
    let sql = "SELECT jsonStyle FROM MENUS WHERE id = ?"
    db.get(sql,[id],(err,row) => {
      if (row){
        console.log("id courant = " + id);
        let data = row.jsonStyle;
        let deleteRowsSQL = "DELETE FROM STYLE";
        db.run(deleteRowsSQL,(err) => {
          if (err) {
            console.log(err);
            reject(false);
          }
          else {
            let sqlInsert = "INSERT INTO STYLE (json) VALUES (?)"
            db.run(sqlInsert,[data]);
            console.log("ok sync style users");
            result(true);
          }
        });
      }
      else {
        reject(false);
      }
    })
  })
}
```

Figure 36 : Méthode loadStyleUser

Cette méthode se nomme loadStyleUser et est présentée dans la figure 36. Elle prend en paramètre un id pour charger un style spécifique qui dépend de l'utilisateur qui est connecté. Semblablement aux autres méthodes vues précédemment, elle effectue un SELECT sur la table MENUS avec un WHERE en fonction de l'id entré en paramètre, une fois la valeur récupérée, elle est ensuite copiée dans la table STYLE après une suppression. Pour afficher ce style modulable et surtout le charger après chaque connexion d'un nouvel utilisateur, j'ai utilisé un useEffect dans le composant qui est le premier qui doit être stylisé. Ce composant est le loginMenu car c'est sur celui-ci qu'on arrive en démarrant l'application.

```
useEffect(() => {
  let nbstyleline = 0;
  isStyleFull().then((value) => {
    console.log('nbrlignevalue = ' + value);
    console.log('idmenueffect = ' + idMenu);
    console.log('Nbruser = ' + isNewUser);
    nbstyleline = value;
    if (nbstyleline >= 1){
      if (idMenu !== ""){
        console.log("loooogeeeeeeed");
        loadStyleUser(idMenu).then(() => {
          selectStyle().then((data) => {
            setStyleData(data);
            updateStylePage(data);
          })
        })
      }
    }
    else {
      selectStyle().then((data) => {
        setStyleData(data);
        updateStylePage(data);
      })
    }
  })
  else {
    let data = require("../utils/stylebase.json")
    setStyleData(data);
    updateStylePage(data);
  }
},[isNewUser])
```

Figure 37 : useEffect permettant d'afficher le style personnalisé

Comme on peut le voir dans la figure 37, ce useEffect va s'exécuter dans tous les cas au démarrage du composant mais va de nouveau s'exécuter si le useState `isNewUser` vient à changer. Ce useState, on l'a vu précédemment, lorsqu'on se connecte il est incrémenté de un, donc varie, ainsi ce useEffect est de nouveau exécuté à chaque connexion. Au départ de ce dernier, il y a `nbstyleline` de déclarée. Cette variable permet de savoir s'il y a du style de chargé dans la table `STYLE`. En fait, lorsque l'application est démarrée pour la première fois, aucune synchronisation a été faite et donc la base de données locale est vide. Cependant, les composants de connexion ont besoin d'un style pour s'afficher correctement, c'est pour cela que j'utilise cette variable afin de vérifier ou non si le style a bien été chargé dans la table. Cette variable va prendre la valeur qui va lui être donnée par la méthode `isStyleFull` qui renvoie un `SELECT(COUNT)` de la table `STYLE`. Donc si `nbstyleline` est strictement supérieur à 1, cela signifie que du style est présent dans la table et qu'on va pouvoir s'en servir. Dans le cas contraire, mis à disposition un style de base dans un fichier afin que l'application puisse s'en servir dans ce cas précis. Dans le cas où un style est chargé, le useEffect vérifie si le useState `idMenu` est non nul, si

c'est le cas, alors il va recharger le style dans la table STYLE à l'aide de la méthode loadStyleUser, c'est ce qui se passe à la connexion car l'idMenu va prendre une valeur. Après avoir chargé le style dans la table, il est sélectionné dans cette dernière afin d'être stocké dans le useState styleData et appliqué dans l'application à l'aide de la méthode updateStylePage. Si le useState idMenu est nul, cela signifie qu'il n'y a pas eu de connexion mais que simplement l'utilisateur est sur le menu pour se connecter, ainsi le useEffect charge le style de la dernière connexion présent dans la table STYLE.

```
const updateStylePage = (data) => {  
  console.log("STYLEEE " + JSON.stringify(data));  
  document.title = data.loginPage.design.title.name;  
  root.style.setProperty('--color1',data.loginPage.design.color.mainColor);  
  root.style.setProperty('--color4',data.loginPage.design.color.secondColor);  
  root.style.setProperty('--color2',data.loginPage.design.color.bgLoginColor);  
}
```

Figure 38 : Méthode updateStylePage

Comme je viens de le dire, le style se met à jour à travers la méthode updateStylePage mais pas que. Cette méthode présentée dans la figure 38 modifie le thème en mettant à jour les trois variables présentes dans mon fichier css et elle met aussi à jour le titre de la page. Il y a également le useState styleData qui permet d'insérer le style dans les composants, en effet ce useState contient l'objet contenant tout le style, ainsi pour charger un image ou le textes dans certains boutons, j'utilise ce useState.

4. Résultat

Suite au développement de ce projet, il a été nécessaire de le tester et de rédiger de la documentation afin de faciliter son utilisation, vous trouverez ainsi dans cette partie, un manuel d'utilisation et les méthodes de tests effectuées.

4.1. Manuel d'utilisation

Afin d'utiliser le projet, il est pour le moment encore nécessaire de le lancer avec nodeJS à l'aide de la commande "npm start". Une fois lancée, on peut quitter l'application en appuyant sur la croix rouge en haut à droite de la fenêtre.

1/ Se connecter

Pour accéder à l'application, il est nécessaire de se connecter. Au démarrage de l'application, on se retrouve sur une page qui propose différents moyens de connexion.

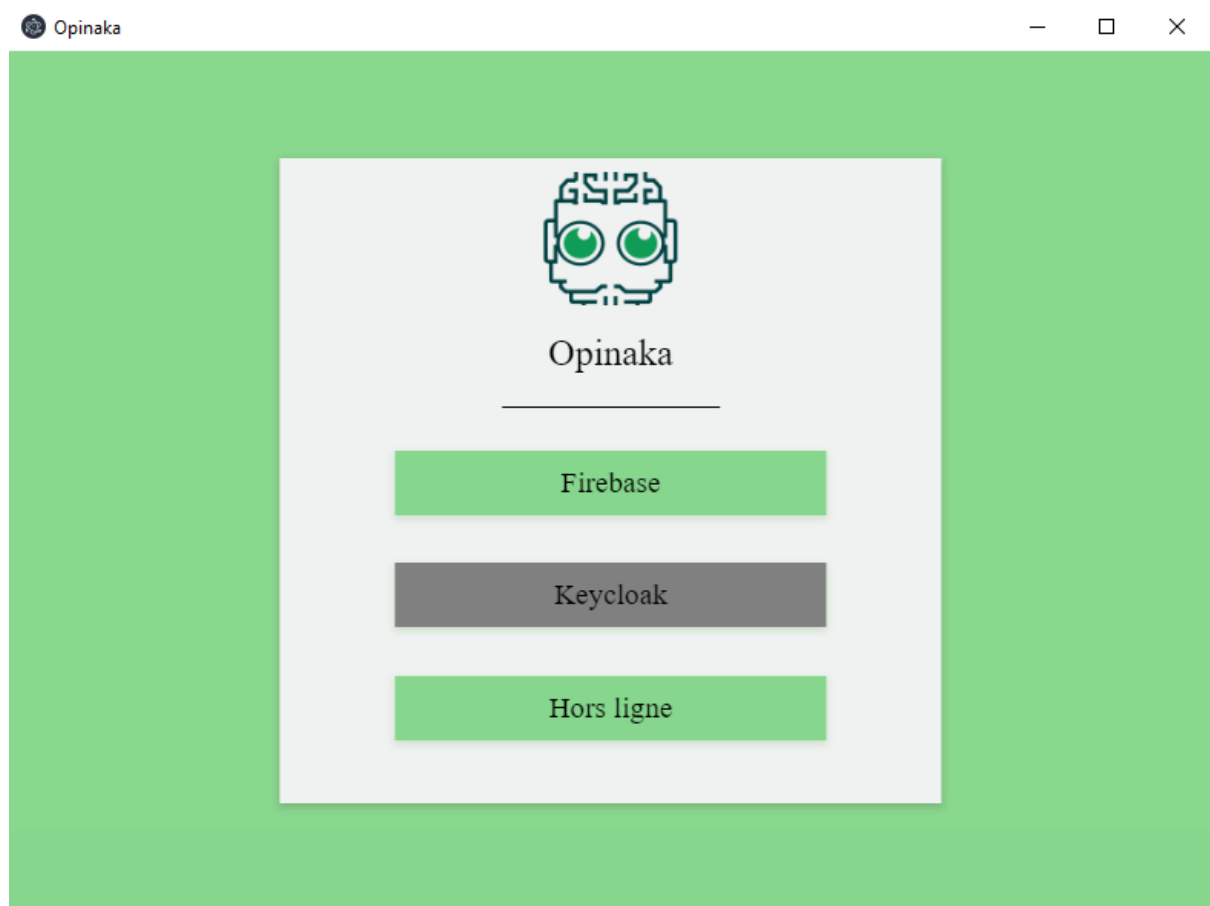


Figure 39 : Page de choix de connexion de l'application

Sur la page présentée dans la figure 39, on peut choisir le moyen de connexion en cliquant sur le bouton en question. Firebase pour se connecter avec Firebase (en ligne) ou Hors ligne pour se connecter en local. Keycloak est pour le moment désactivé car le service de connexion ne fonctionne pas correctement. En choisissant le moyen de connexion, cela nous renvoie sur le formulaire de connexion choisi.

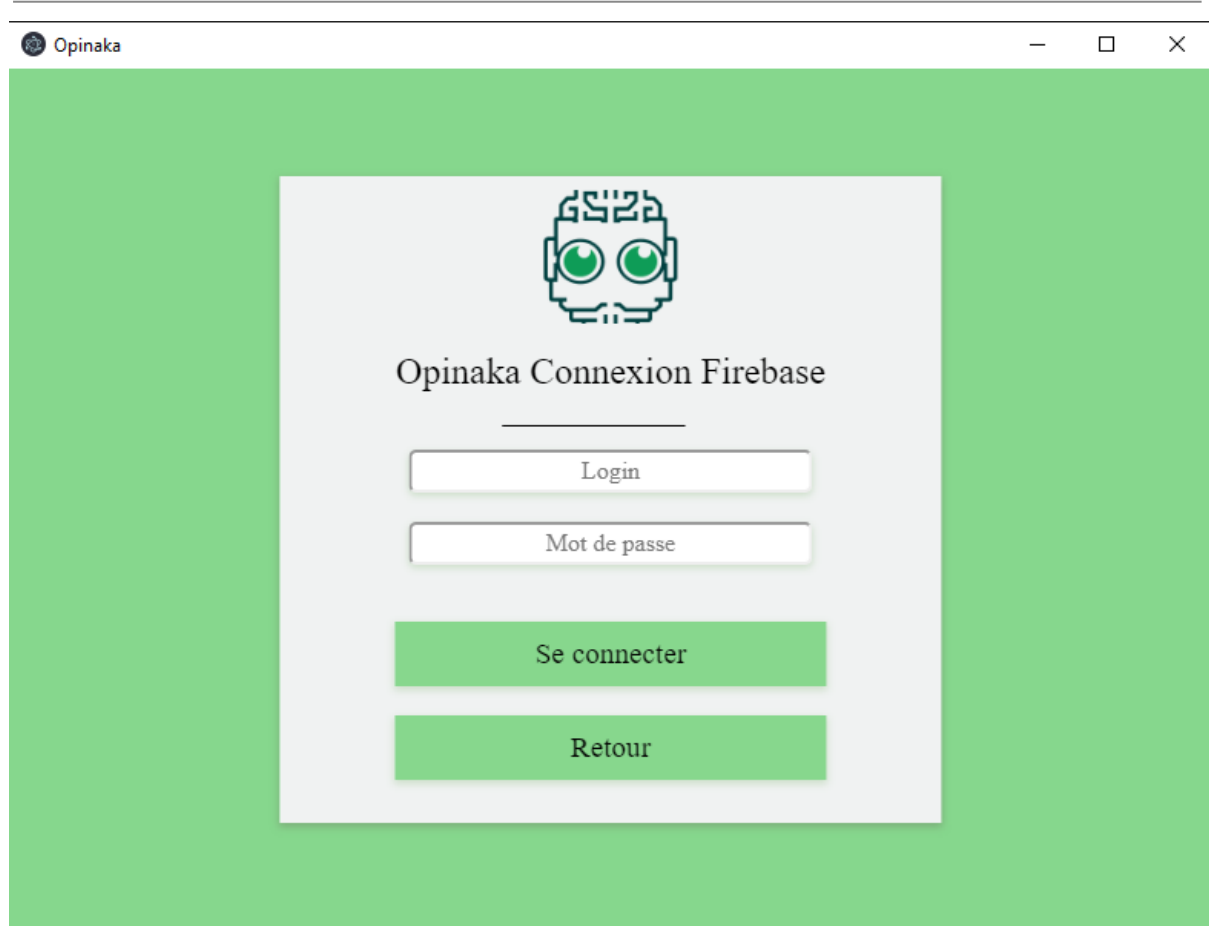


Figure 40 : Formulaire de connexion Firebase de l'application

Dans la figure 40 est présenté le formulaire de connexion pour Firebase, c'est le même formulaire pour la connexion en local. Comme pour tout formulaire, ici, il suffit de rentrer des identifiants valides et cliquer sur le bouton "se connecter" afin d'accéder à l'application. On peut aussi retourner au menu pour choisir son mode de connexion en appuyant sur le bouton "retour".

2/ Utilisation de l'application

Après s'être connecté, on arrive sur l'application contenant le menu de sites.

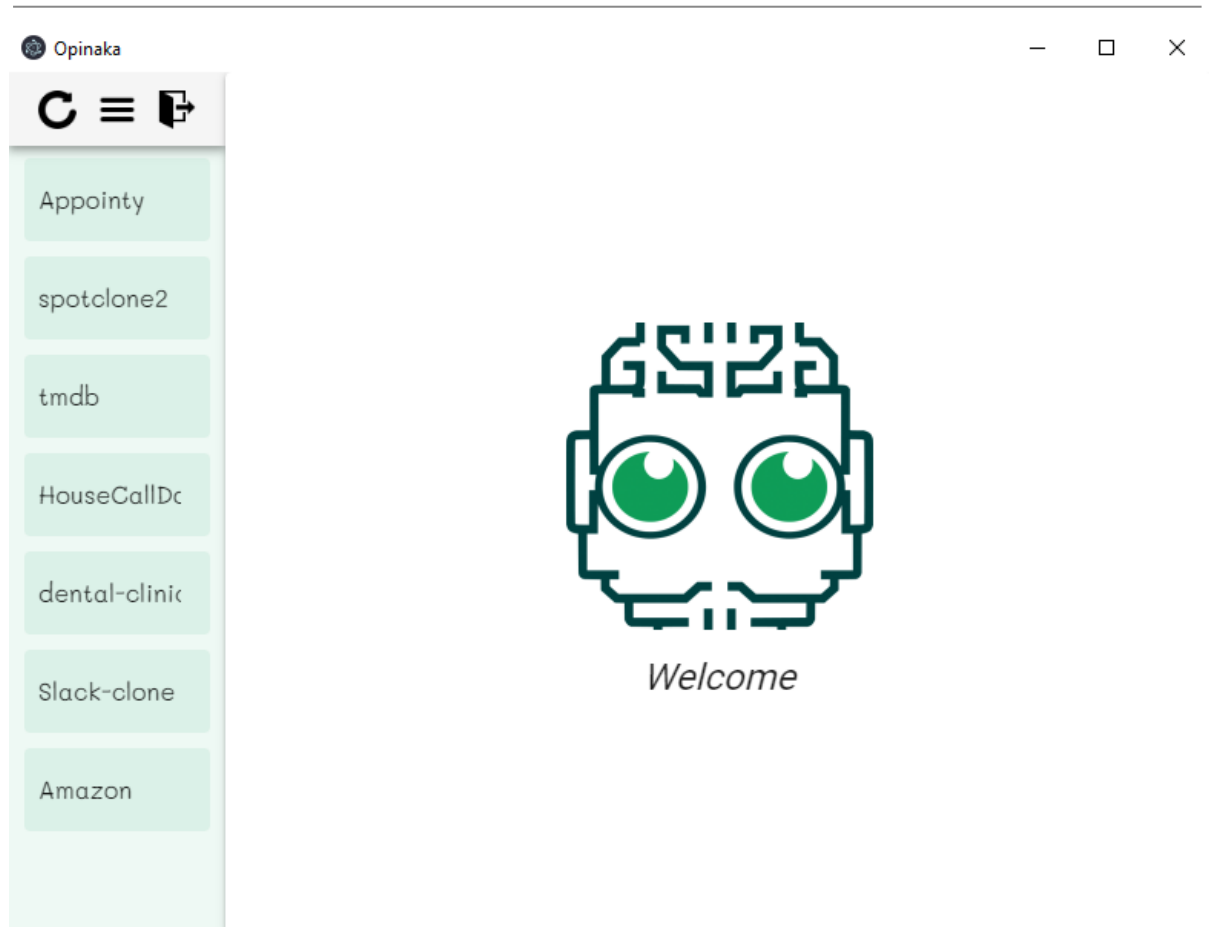


Figure 41 : Page d'accueil de l'application en étant connecté

Sur la page d'accueil de l'application présentée dans la figure 41, il y a un menu sur la gauche contenant tous les sites disponibles sur le compte connecté. Si l'on clique sur l'un d'entre eux, alors il s'affiche sur la partie droite de l'application.

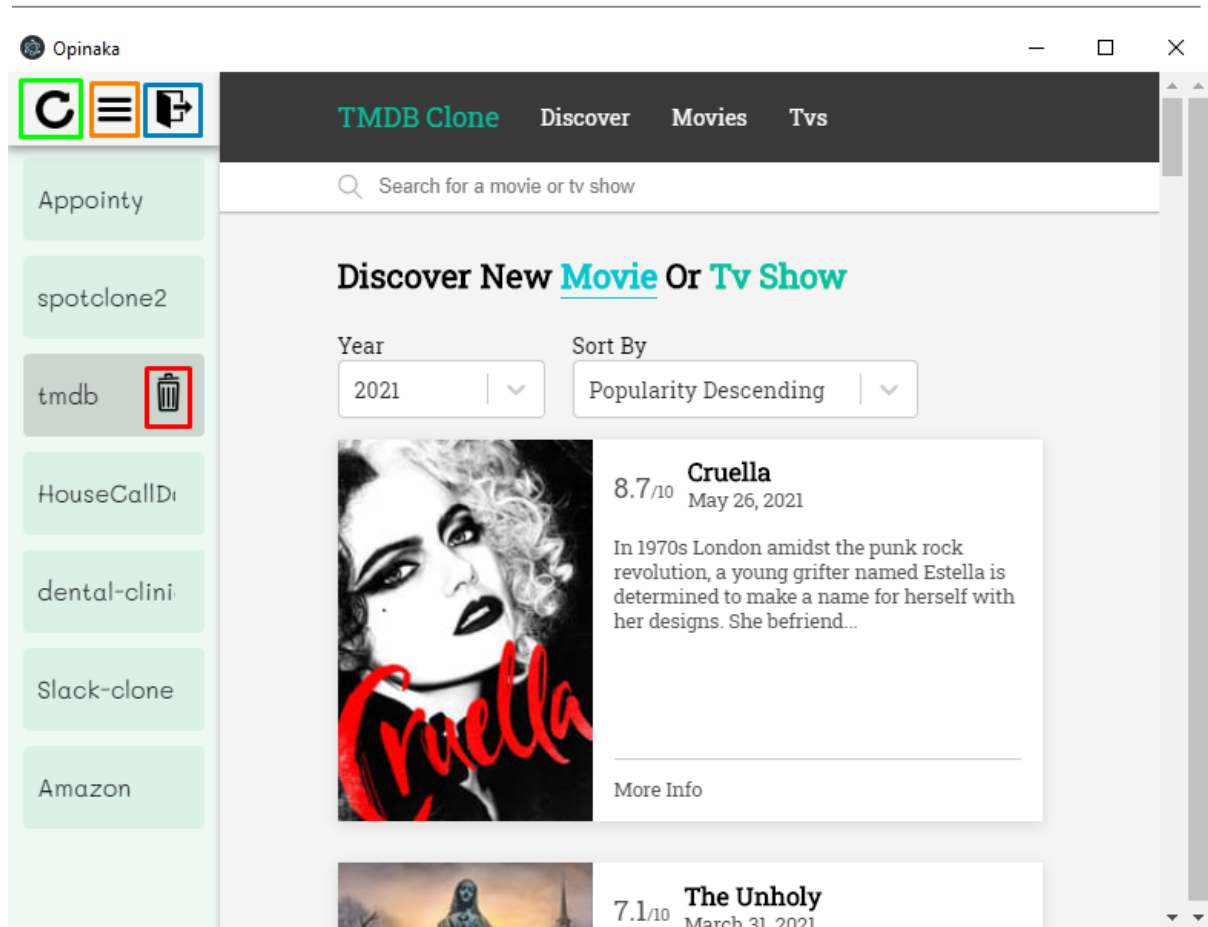


Figure 42 : Affichage d'un site dans l'application (tmdb)

Dans la figure 42, le site sélectionné, tmdb, est affiché. L'utilisateur peut donc naviguer sur le site qui s'affiche et donc faire tout ce qu'il pourrait faire sur un navigateur classique. De plus, dans cette application, il y a quatre boutons. Le premier bouton, encadré en rouge, permet de cacher l'item sélectionné de la liste. Le bouton encadré en vert permet quant à lui de faire réapparaître tous les boutons cachés. Le bouton encadré en orange permet de revenir au menu principal de l'application, donc n'affiche plus de site web sur le côté droit. Enfin, le bouton encadré en bleu permet tout simplement à l'utilisateur de se déconnecter et donc de revenir au formulaire de connexion de l'application.

3/ Ajouter ou modifier des utilisateur ou données dans la base de données

Pour accéder à la base de données, il est nécessaire d'avoir un accès à la console Firebase qui est reliée à l'application. Si l'on a l'accès, alors on peut y retrouver une partie authentification.

Rapport de projet pour l'entreprise Opinaka : Menu personnalisable

Tillier Etienne

The screenshot shows the Firebase authentication console. At the top, there is a search bar with the placeholder text 'Recherchez par adresse e-mail, numéro de téléphone ou ID utilisateur'. To the right of the search bar is a blue button labeled 'Ajouter un utilisateur', which is highlighted with a red rectangle. Below the search bar is a table with the following columns: 'Identifiant', 'Fournisseurs', 'Date de création', 'Dernière connexion', and 'UID utilisateur'. Below the table is a form titled 'Ajouter un utilisateur avec un e-mail/mot de passe'. The form has two input fields: 'E-mail' and 'Mot de passe'. Below the input fields are two buttons: 'Annuler' and 'Ajouter un utilisateur'. Below the form is a table with the following data:

Identifiant	Fournisseurs	Date de création	Dernière connexion	UID utilisateur
etienne@test.com		25 mai 2021	9 juin 2021	2wKnVHjVP7MFzejT6EFKX78GKk...
pierre@test.com		25 mai 2021	28 mai 2021	DRsVXIDrMAa3jXP6illN3rRxiT33
test@test.com		25 mai 2021	28 mai 2021	RF5aMWxGyOUxv5urUifaGAwdZV...
jean@test.com		25 mai 2021	26 mai 2021	f7eifhntNpQs31MsJNiZhNfsTw2
demo@test.com		25 mai 2021	28 mai 2021	yrmHexeaAKb36cFDnOajlL4mi2v1

At the bottom of the table, there is a pagination control showing 'Lignes par page : 50' and '1 - 5 of 5'.

Figure 43 : Authentification firebase (ajout utilisateur)

Dans cette partie authentification présentée dans la figure 43, on peut ajouter un utilisateur en cliquant sur le bouton encadré en rouge. Il faut ensuite entrer un email ainsi qu'un mot de passe qui seront les identifiants pour se connecter à l'application via Firebase. Une fois cela fait, il est possible de se connecter avec ce compte à l'application mais avec des données par défaut car il n'a pas de données associées. Pour ajouter des données à ce compte, il faut accéder à la base de données RealtimeDB de Firebase.



Figure 44 : Base de données (RealtimeDB) de Firebase

Dans cette base de données présentée dans la figure 44, il y a toute la base de données de l'application. Elle est totalement modifiable, on peut ajouter un utilisateur en cliquant sur le "+" encadré en rouge. Il est nécessaire de rentrer un UID existant dans le système d'authentification afin que l'utilisateur enregistré soit pris en compte. De plus, il faut entrer la même structure de données que pour les autres utilisateurs. Également, on peut modifier les données des utilisateurs existant. Pour cela, il faut simplement cliquer sur les données que l'on souhaite modifier et entrer les nouvelles données. Cela se mettra à jour tout seul sur l'application.

4.2. Validation

Durant la réalisation de ce projet, j'ai eu de nombreuses erreurs lors d'installation de nouvelles bibliothèques ou alors par des oublis dans le code. Pour remédier à cela, j'ai gardé la console ouverte sur le côté de l'application, c'est un avantage qu'offre le développement avec Electron. Ainsi, si j'avais du code qui ne s'exécutent pas ou des erreurs, pour déboguer, je faisais des "console.log()" dans la console pour voir s'ils s'affichaient et ainsi je pouvais plus facilement situer où l'erreur était. Lorsque j'ai dû rendre le projet à mon maître de stage, sur sa machine il a eu beaucoup de problèmes pour lancer le projet car c'est un projet avec nodeJS. Les projets avec

cette technologies installent des bibliothèques dans un fichier "node_modules", or lorsqu'on publie son projet, on vide ce dossier car il est trop volumineux et il suffit à l'utilisateur avant de lancer le projet de lancer une commande "npm install" afin d'installer toutes les bibliothèques directement sur sa machine pour que tout fonctionne. Pour ce rendu, une bibliothèque posait problème et j'ai dû la remplacer dans le code afin qu'elle ne soit plus à installer pour que tout fonctionne sur une autre machine que la mienne. Dans ce projet, j'ai aussi travaillé le style de l'application et rendu responsive tout ce que je faisais. Ainsi, j'ai demandé des avis autour de moi pour savoir ce que les personnes pensaient. Cela m'a permis de modifier certaines choses pour les rendre plus esthétiques

5. Rapport d'activité

Pour la bonne réalisation de ce projet, un suivi régulier à été mis en place par mon maître de stage malgré les conditions difficiles en distanciel. Ainsi je faisais part de mon travail très régulièrement à mon maître de stage par divers moyens.

5.1. Méthode de développement et outils

Ce stage s'est déroulé donc à distance, j'étais avec trois autres stagiaires qui faisait le même travail que moi mais nous avons dû rendre un travail indépendamment des autres. Il était recommandé que je communique avec eux pour s'entraider sur nos problématiques qui étaient semblables. J'avais une réunion quasiment tous les jours via des applications de réunion en ligne (Zoom, webex...) où nous faisons le point sur les tâches qui étaient à réalisées. Ainsi chaque stagiaire montrait son travail à l'aide d'un partage d'écran et chacun pouvait réagir et donner son avis. A la fin de la réunion, le maître de stage donnait de nouvelles missions pour la prochaine réunion et c'est comme ça, petit à petit, que le projet avançait. Je travaillais donc de manière agile avec de nouvelles missions ou des modifications sur mon travail à réaliser pour les prochaines réunions. En dehors des réunions, le maître de stage nous avait réuni sur une conversation WhatsApp, c'est dans cette conversation qu'étaient indiquées toutes les informations concernant le stage (réunion, information et messages diverses). De ce fait, même si je pouvais proposer des solutions pour améliorer le projet, les technologies utilisées pour programmer ou pour la base de données nous ont été imposées. La conception a été généralisée pour tous les stagiaires, chacun a donné son avis et ses besoins pour le fonctionnement de son application. Cela résulte à une conception imparfaite pour chacun d'entre nous puisqu'elle doit fonctionner pour chacun d'entre nous mais nous n'avons pas les mêmes besoins. En dehors des réunions, le travail se réalisait en autonomie. Je me documentais sur des vidéos youtube ou sur des sites internet comme StackOverflow pour les erreurs

dans le code. J'ai réalisé le développement de mon projet sur Visual Studio Code ainsi que sur Sublime Text.

5.2. Bilan critique par rapport au cahier des charges

Comme je l'ai précisé précédemment, c'est moi qui ai rédigé le cahier des charges en cumulant les différentes missions qui m'ont été adressées au fur et à mesure du stage. Ainsi, j'ai réalisé tout ce qui m'a été demandé car pour passer à la mission suivante je devais obligatoirement finir sur ce sur quoi j'étais en train de travailler. La seule chose qui n'était pas dans le cahier des charges mais que j'aurais aimé réaliser, c'est de pouvoir affiner le code. En effet, par manque de temps, je n'ai pas eu le temps de revoir le code en entier. En réalité, je ne l'ai jamais analysé en entier, étant donné que je suis parti d'une application existante, dans un premier temps, j'ai étudié et modifié que ce qui m'était utile car c'était une application complexe et il m'aurait fallu beaucoup de temps pour l'analyser en entier dès le début du stage. Ainsi, même après avoir rendu le projet, il reste des parties dans le code qui font encore référence à l'ancienne application. L'application fonctionne parfaitement et ne se sert pas de ces morceaux de code mais j'aurais aimé avoir eu le temps de m'en occuper afin de rendre ça plus clair.

Conclusion

La mission qui m'a été confiée était la réalisation d'une application qui comporte un système de connexion qui permet d'accéder à un menu de lien internet accessible depuis l'application. L'application se devait d'être simple d'utilisation et d'être esthétique.

Au fur et à mesure des différentes missions, j'ai réalisé le projet en prenant en compte toutes les contraintes demandées. Je suis très heureux du rendu du projet et fier d'avoir eu l'occasion de le réaliser. J'aurais toutefois aimé, avec plus de temps, peaufiner davantage le code afin de le rendre plus clair.

Ce projet pourrait servir à des entreprises comme un logiciel pour le personnel regroupant à l'aide du menu les liens utiles pour l'activité de l'entreprise.

Grâce à ce projet, j'ai pu approfondir mes connaissances en JavaScript et surtout découvrir le framework React. J'ai aussi fait la connaissance de Firebase que je trouve très intéressant ainsi que des bases de données embarquées comme SQLite. Pour finir, la réalisation de ce projet m'a permis de découvrir le monde de l'entreprise, j'ai également développé mes compétences techniques et développé mon travail en autonomie.

Visa du maître de stage

Date et signature :

Le 10 juin 2021



Bibliographie

[1]

Chris Coyer, «Updating a CSS variable with JavaScript», sept. 12, 2018.
<https://css-tricks.com/updating-a-css-variable-with-javascript> (consulté en mai. 2020).

[2]

Priya Pedamkar, «React Login Form»
<https://www.educba.com/react-login-form> (consulté en mai. 2020).

[3]

Christina Holland, «Using Firebase in Electron: Tips and Tricks», nov. 21, 2019.
<https://medium.com/firebase-developers/using-firebase-in-electron-tips-and-tricks-24ac5b44bf5a> (consulté en mai. 2020).

[4]

Vivek Nair, «How to authenticate with Google Oauth in Electron», jan. 10, 2020.
<https://pragli.com/blog/how-to-authenticate-with-google-in-electron> (consulté en avril. 2020).

[5]

The Net Ninja, «Firebase authentication tutorial #2 - How Firebase Auth Works», jan. 02, 2019.
https://www.youtube.com/watch?v=mEF9WRwYDfY&list=PL4cUxeGkcC9jUPIes_B8vRjn1_GapIOPQ&index=3 (consulté en avril. 2020).

[6]

From Scratch - Développement Web, «[React] Authentification avec Facebook et Gmail grâce à Firebase», août. 20, 2020.
https://www.youtube.com/watch?v=au_n76X8bv0 (consulté en avril. 2020).

[7]

marshalllofsound, «Electron Webview», 2017.
<https://www.npmjs.com/package/react-electron-web-view> (consulté en avril. 2020).

[8]

Hong developer, «How to review React native application database with tables», juil. 12, 2019.
<https://stackoverflow.com/questions/57007307/how-to-view-react-native-application-database-with-tables> (consulté en avril. 2020).

[9]

Glenn Stovall, «How to Use useEffect (and other hooks) in class components»,
<https://glennstovall.com/how-to-use-useeffect-and-other-hooks-in-class-components> (consulté en avril. 2020).

6. Annexes

6.1. Sommaire

6. Annexes	55
6.1. Sommaire	55
6.2. Articles	56
6.2.1. Bibliothèques	56
6.2.1.1. Melody	56
6.2.1.2. BaseWeb	57
6.2.2. Clones	58
6.2.2.1. Spotify-Clone	59
6.2.2.2. TMDB Clone	60
6.2.2.3. Spotify-Clone 2	61
6.2.2.4. Slack Clone	62
6.2.2.5. Deer	63
6.2.2.6. HouseCall Doctors	64
6.2.2.7. Dental Clinic App	65
6.2.2.8. Appointy	66
6.2.2.9. ElectronBrowser	67
6.2.2.10. Todoizt	68
6.3. Journal de suivi du stage (non terminé)	68

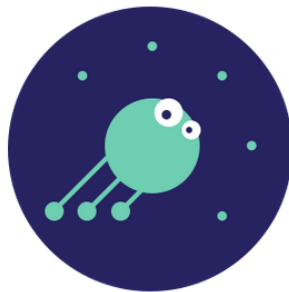
6.2. Articles

6.2.1. Bibliothèques

6.2.1.1. Melody

Melody

par T tienne Tillier | Avr 22, 2021 | Uncategorized | 0 commentaires



Melody

C'est une biblioth que d'interface utilisateur cr  e par Trivago pour la cr  ation d'application en JavaScript. Elle a   t  d velopp  e afin de pouvoir satisfaire les clients du monde entier pour l'entreprise Trivago en r  duisant au maximum le temps d'interactivit  , en r  duisant   galement l'utilisation de la m  moire et pour finir, Melody se veut famili  re pour tous les ing  nieurs afin que ces derniers puissent d  velopper dans de bonnes conditions.

Melody fonctionne avec des mod  les Twig et poss  de un compilateur qui permet de les convertir en « melody-idom ». Il existe aussi des plugins qui permettent de convertir les mod  les Twig autrement, par exemple en JSX pour pouvoir les utiliser avec React.

Les « melody-idom » sont des instructions qui lors de l'ex  cution vont indiquer au DOM    quoi il doit ressembler. Cela permet une ex  cution tr  s rapide en m  moire.

6.2.1.2. BaseWeb

BaseWeb

par T tienne Tillier | Avr 22, 2021 | Uncategorized | 0 commentaires



BaseWeb

Il s'agit d'un framework d'interface utilisateur sous React d velopp  par l' quipe d'Uber afin de simplifier la maintenance et le d veloppement de leurs diff rentes applications. Depuis 2018, il est en open source et offre un langage de conception qui permet la cr ation de site web simplement et rapidement ou m me la r alisation de nouvelles conceptions. Cette biblioth que met   disposition ses composants bas s sur React, ces derniers sont simples   prendre en main et tr s personnalisables.

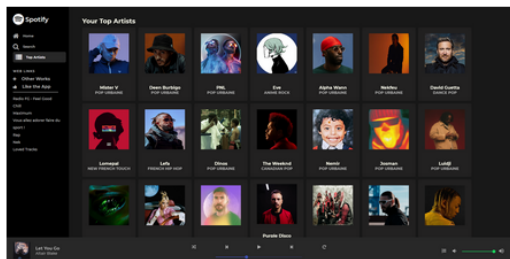
En effet, BaseWeb contient une large gamme de composant, ces derniers font r f rences aux composants individuels HTML. Ils sont tous personnalisable et il est m me possible de cr er des blocs de composants afin de modifier tous les composants  tant dans le bloc. De la m me mani re, un bloc peut contenir un autre bloc. De plus, pour utiliser BaseWeb, seulement les d pendances npm sont n cessaires ce qui permet une grande accessibilit . Evidemment, toutes les applications d'Uber sont maintenant bas es sur cette biblioth que, ils assurent donc la maintenance de cette derni re avec de nombreux tests. L' quipe Uber met aussi   disposition un canal de communication pour sa communaut  afin que celle-ci puisse sugg rer des am liorations de la biblioth que. Enfin, BaseWeb utilise [Styletron](#) pour son style, cela permet de g n rer un style atomique qui n cessite moins de ressources et donc rend l'application plus performante   travers l'optimisation des temps de chargement.

6.2.2. Clones

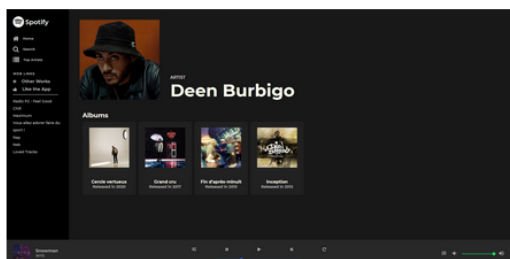
6.2.2.1. Spotify-Clone

Clone Spotify

Il s'agit d'une application permettant d'écouter de la musique où les musiques sont disponibles directement sur l'application.



- Utilise l'API Spotify
- Page de recherche de musique/artiste
- Liste des playlist associée au compte
- Page avec du contenu personnalisé en rapport avec le compte connecté
- Ecoute de musique avec le player
- On peut ajouter des musiques en favoris
- On peut ajouter des musiques en favoris et les retrouver ensuite dans la liste des favoris
- Il y a une page pour chaque artiste/album
- Page des « top artistes »

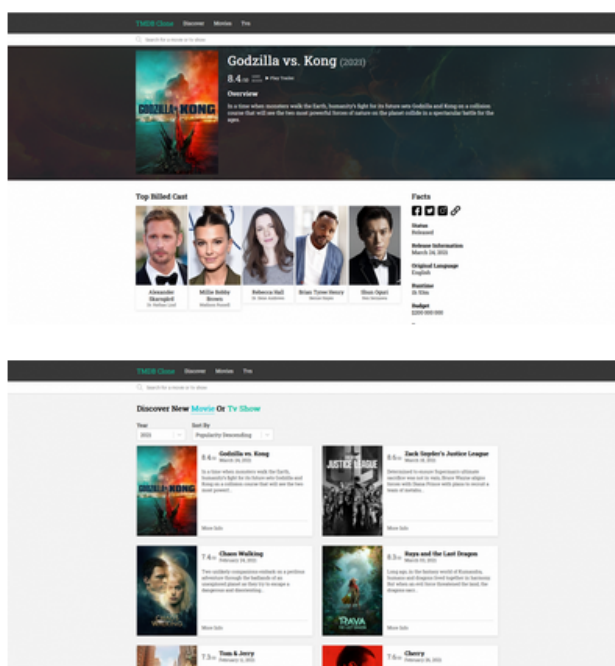


- On ne peut pas créer de playlist
- Il n'y a pas de compte connecté alors qu'on est pourtant connecté à l'API
- Il n'y a pas les podcasts
- Il n'y a pas de bouton retour
- On ne peut pas voir sa liste de titre favoris
- Il faut se connecter à un compte Spotify pour que l'application se connecte à l'API
- Le système de recherche n'est pas optimale
- Les différentes fonctionnalités du menu de connexion sauf la connexion, ne sont pas fonctionnelles

6.2.2.2. TMDB Clone

TMDB Clone

Il s'agit d'une application permettant de rechercher des films et d'avoir plus de détails sur ces derniers.



- Utilise l'API TMDB
- Page de recherche de films et séries télé
- Tri par catégorie pour la recherche
- Page détaillée pour chaque film
- Vidéo trailer pour chaque film disponible
- Liste des acteurs de chaque film
- Recommandations de films
- Les films sont notés
- Réseaux sociaux de chaque film

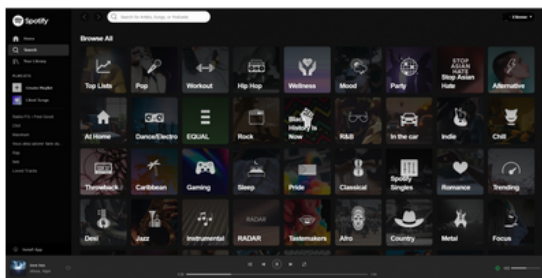


- Il n'y a pas de page pour les acteurs
- On ne peut pas noter les films
- Il n'y a pas de système de connexion

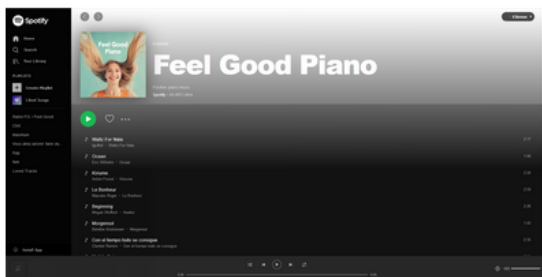
6.2.2.3. Spotify-Clone 2

Clone Spotify

Il s'agit d'une application permettant d'écouter de la musique où les musiques sont disponibles directement sur l'application.



- Utilise l'API Spotify
- Page de recherche de musique/artiste
- Liste des playlist associée au compte
- Page avec du contenu personnalisé en rapport avec le compte connecté
- Page profil du compte connecté
- Ecoute de musique avec le player
- On peut ajouter des musiques en favoris et les retrouver ensuite dans la liste des favoris
- Le clone est reconnu par l'API et donc il est possible de faire basculer l'écoute sur le clone depuis un téléphone et vice-versa

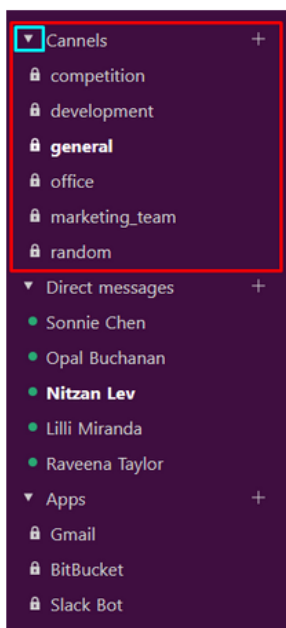


- Il n'y a pas de page pour les artistes
- On ne peut pas noter les films
- L'image du profil ne s'affiche pas
- Il n'y a pas les podcasts
- Bouton de retour en arrière non fonctionnel
- Il faut se connecter à un compte Spotify pour que l'application se connecte à l'API

6.2.2.4. Slack Clone

Slack-clone

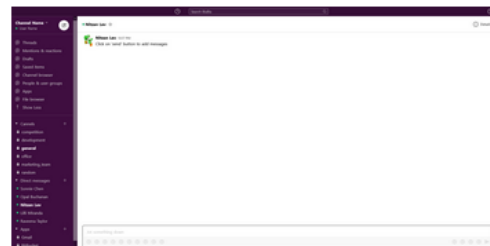
Il s'agit d'une application permettant de discuter avec des personnes sur différents canaux.



- interface de menu
- Boutons qui permettent de cacher des éléments du menu



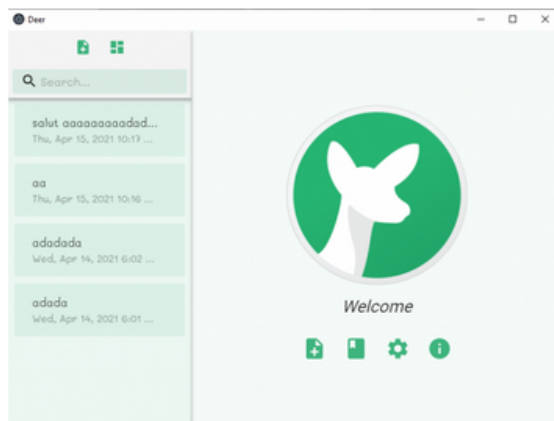
A part les boutons afin de réduire le menu, rien ne fonctionne sur l'application.



6.2.2.5. Deer

Deer

Il s'agit d'une application permettant de sauvegarder des notes



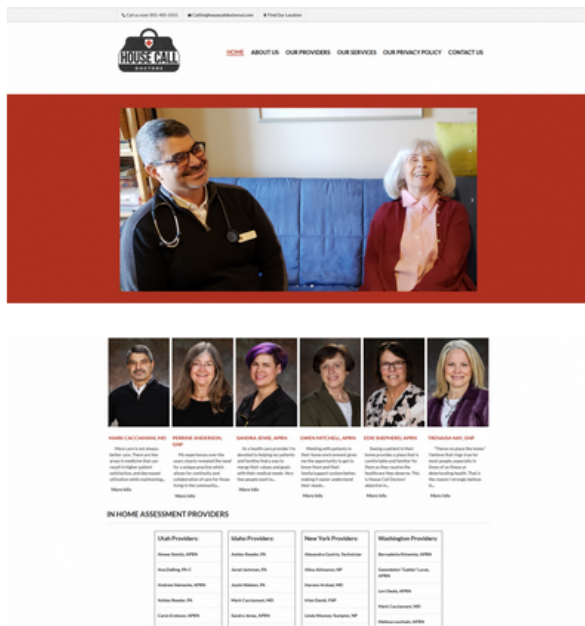
- Personnalisation du texte dans les notes
- Ajout et suppressions de notes
- Interface agréable
- Fonction de recherche de note
- Changement de langue possible
- Animation de transition de page

Aucun point négatif rencontré

6.2.2.6. HouseCall Doctors

HouseCall Doctors

Il s'agit d'un site web permettant de contacter des docteurs et d'avoir plus d'info à propos de ces derniers



- Menu en haut de page fonctionnel
- Liste des médecins
- Popup disponible pour plus d'info sur le médecin en question
- Intégration de GoogleMap
- Page contact

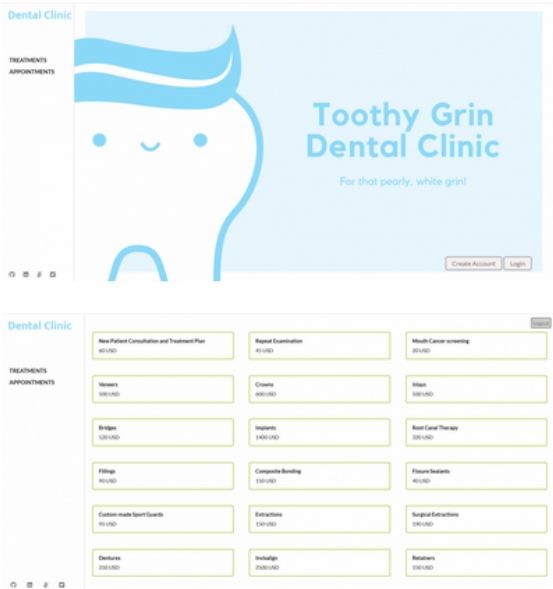


- Pas de profil médecin
- Pas de prise de rendez-vous
- Pas de connexion à une base de donnée
- Formulaires pas très esthétiques

6.2.2.7. Dental Clinic App

Dental Clinic App

Il s'agit d'une application permettant de prendre rendez-vous pour des prestation avec un médecin.



- Donnée en JSON
- Formulaire pour choisir le médecin ainsi que la date du rendez-vous
- Liste des prestations cliquable pour accéder au formulaire de rendez-vous
- Page de connexion/inscription
- Barre menu statique (à gauche)
- Page pour retrouver ses rendez-vous
- Animation de chargement entre les pages

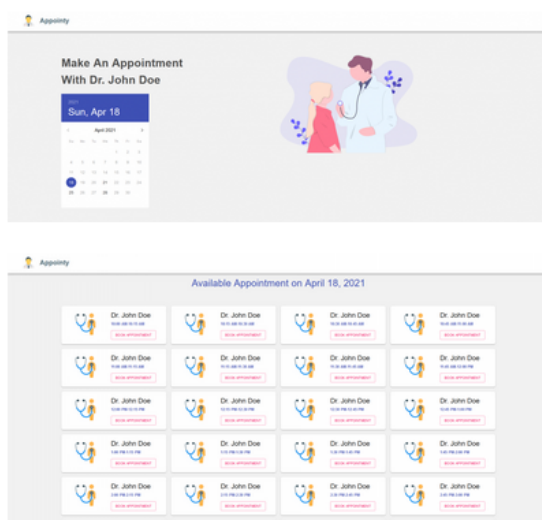


- Pas de profil médecin

6.2.2.8. Appointy

Appointy

Il s'agit d'une interface graphique permettant de prendre des rendez-vous avec des médecins



- Donnée en JSON
- Calendrier pour choisir le jour du rendez-vous
- Liste des créneaux disponibles en fonction du jour et des disponibilités du médecin
- Page contact
- Barre statique (en haut)
- Petit formulaire pour la prise de rendez-vous



- On ne voit pas les rendez-vous que l'on prend
- Pas de profil médecin
- Une fois un créneau pris, il est toujours disponible
- Bouton pour prendre rendez-vous pas très esthétique

6.2.2.9. ElectronBrowser

ElectronBrowser

Il s'agit d'un navigateur sous Electron qui permet de naviguer sur internet.



- Permet de faire des recherches sur internet
- Peut avoir plusieurs onglets
- Bouton retour et d'actualisation
- Bouton pour réduire, fermer ou agrandir la page
- Bouton pour revenir à la page d'accueil
- Assez bien responsive



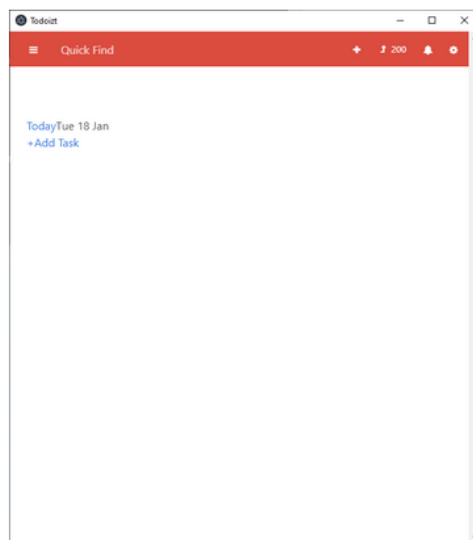
- Navigateur non à jour, l'accès à certains sites est restreint
- Navigateur non recommandé pour la navigation car sûrement mal sécurisé



6.2.2.10. Todoizt

Todoizt

Il s'agit d'une application qui permet de stocker des tâches et de mettre des alarmes de rappel pour ces dernières.



- interface de menu
- Animation lorsqu'on clique sur la barre de recherche



Il n'y a que la barre de recherche qui est intéressante sur cette application.



6.3. Journal de suivi du stage (non terminé)

(1ère mission : 2 jours | Recherche bibliographique sur les bibliothèques de type front-end existantes) : Afin de réaliser cette mission, j'ai fait des recherches sur GitHub et principalement sur des sites d'entreprises connues (Netflix, Uber, etc...). A travers ces recherches, j'ai découvert de grandes bibliothèques comme [BaseWeb](#) ou encore [Melody](#) qui est un peu moins connue. De la même manière, j'ai aussi fait la découverte de nombreuses petites bibliothèques comme [Vue Carousel](#) par exemple. Il s'agit d'une bibliothèque pour VueJs qui permet l'implémentation de carrousel sur un site web. Effectivement, grâce à cette Vue-Carousel, on peut ajouter des « slides » ou des « carrousels » dans n'importe quel composant VueJs. Les diapositives dans le carrousel sont très flexibles car on peut y mettre toutes sortes de composants VueJs. Cette bibliothèque est utilisée par l'application Just-Eat

(2ème mission : 5 jours | Recherche de clone UI Javascript d'application type TMDB,Netflix,Uber,Spotify...) : Comme pour la première mission, généralement, mes

recherches se sont déroulées sur GitHub. J'ai commencé par me focaliser sur les clones Spotify, j'ai trouvé de nombreux clones dont la majorité qui ne fonctionnaient pas ou qui étaient incomplets. J'ai tout de même pu en relever deux qui sortent du lot et qui sont donc fonctionnels ([le premier clone](#), [le deuxième clone](#)). Ces deux clones fonctionnent avec l'API de Spotify, ce qui est pratique pour tester l'interface de l'application sans se prendre la tête avec une base de donnée. Après les clones spotify, je me suis mis à la recherche de clone TMDB, pareillement que pour les clones Spotify, j'ai eu beaucoup de résultats mais seulement parmi tout cela, seulement un seul de retenu ([clone TMDB](#)). Il utilise tout comme les précédents clones l'API TMDB. Pour finir sur cette mission, j'ai recherché des clones d'application de santé car nous voulions adapté un clone pour réaliser un site dans le domaine médical avec des médecins, patients etc... J'ai donc recherché des clones de Doctolib ou encore de ZocDoc, mais il n'en existe pas de disponible gratuitement en JavaScript, en réalité, il existe même des demandes rémunérées pour la réalisation de ce genre de clone. Donc j'ai dû me contenter de plus petits clones trouvés sur github comme [Appointy](#), [Dental Clinic App](#) ou encore [HouseCall Doctors](#).

(3ème mission : 2 jours | Mettre les clones sur un serveur Xampp) : Pour réaliser cette tâche, j'ai évidemment créer les builds de chaque clone. Le problème est que les builds étaient non fonctionnelles pour la plupart, j'ai dû tous les modifier (les URL des scripts et images surtout), cela a permis le fonctionnement de certains sur Xampp mais pour d'autres, je n'ai toujours pas trouvé de solution.

(4ème mission : 3 jours | Recherche de clone UI Electron JS et de clone de Slack) : De la même manière que pour les missions 1 et 2, j'ai réalisé mes recherches sur GitHub. Au tout départ, j'ai recherché toutes sortes de clones ou programmes faits sur Electron JS, j'ai trouvé énormément d'applications mais la grande majorité sont incomplètes ou ne fonctionnent pas. J'ai tout de même retenu quelques applications comme [Un navigateur sur electron](#), ou alors [Todoizt](#), il s'agit d'une application non finie qui possède une animation de barre de recherche intéressante. En effet, dans ces recherches, il faut se contenter parfois de petits détails car c'est assez rare de tomber sur un clone parfait et fonctionnel. J'ai tout de même réussi à trouver une application terminée et fonctionnelle : [Deer](#). Il s'agit d'une application qui permet de prendre des notes avec un texte personnalisable et de les sauvegarder. Elle comporte aussi de belles animations et est faite sur React JS. Je reparlerai plus loin de cette application car elle va me suivre tout au long de mon stage. Enfin, nous avons aussi pour mission de trouver des clones de Slack, après avoir fouiller partout sur internet et majoritairement sur GitHub, j'ai fini par avoir une dizaine de clone mais seulement un seul qui sort du lot : [Slack-clone](#).

(5ème mission : 2 jours | Recherche de bibliothèque qui permettent la création de menu à partir de donnée en JSON) : Au départ de cette mission, j'avais mal compris le sujet, je pensais qu'il fallait trouver une bibliothèque permettant l'implémentation d'une sidebar personnalisable à partir de donnée en JSON, c'est donc un peu plus complexe que ce qui était demandé. J'ai donc trouvé qu'une bibliothèque qui répondait approximativement à la question : [custom-sidebar](#). Ensuite, après avoir bien compris le sujet, j'ai trouvé un [site](#) qui présente une manière de faire en ReactJS mais cela ne m'a pas convaincu et j'ai finalement trouvé une [solution](#) sur un forum qui répondait parfaitement à la problématique.

Effectivement, cette solution est faite pour ReactJs, elle fonctionne avec des données en JSON et comprend les sous-menus.

(6ème mission : 4 jours | Implémenter un menu avec des données en JSON sur un clone ElectronJs) : Pour commencer cette mission, j'ai déjà du me mettre à la recherche d'une application ou d'un clone que j'avais trouvé au préalable et qui pourrait correspondre à ce qui était demandé. J'ai pensé qu'il serait bon que cette dernière contienne déjà un menu, j'ai donc cherché et n'ayant pas eu beaucoup de clones ou applications avec un menu, je me suis contenté de [Deer](#), l'application contient un « panel » qui contient les notes stockées dans la base de donnée, je me suis dit que cela pourrait convenir pour un menu. De plus, cela faisait déjà un bon moment que j'étais en recherche de menu et que je n'en trouvais aucun de plus pertinent que celui-ci. Suite à ma décision, j'ai cherché à comprendre comment l'application fonctionne. Pour cela, j'ai dû regarder des tutoriels sur React car l'application est basée sur cette technologie avec redux. Après m'être assez informé sur la partie menu de l'application, je me suis aperçu que je n'allais pas intégrer une bibliothèque car je pouvais plutôt adapter directement le menu de l'application afin qu'il soit formé à partir de donnée au format JSON. Au final, j'avais donc mon fichier JSON qui contient toutes les notes de l'application qui étaient affichées dans le menu. Le fichier JSON intervenait uniquement dans l'affichage du menu, sinon c'était la base de donnée (PouchDB) qui était requise pour toutes les autres actions. Le but de ce menu n'était pas de stocker des notes comme l'application le faisait déjà, mais plutôt de pouvoir accéder via ce menu à des réseaux sociaux. Jusque là, je n'avais que modifier la lecture des données présente dans les méthodes d'affichage du menu afin que cela puisse comprendre les donnée en JSON et les afficher. Mais pour accéder aux réseaux sociaux à la place des notes stockées, il a fallu que je modifie les méthodes liées directement à l'utilisation du menu comme le click par exemple afin de rediriger vers une page web.

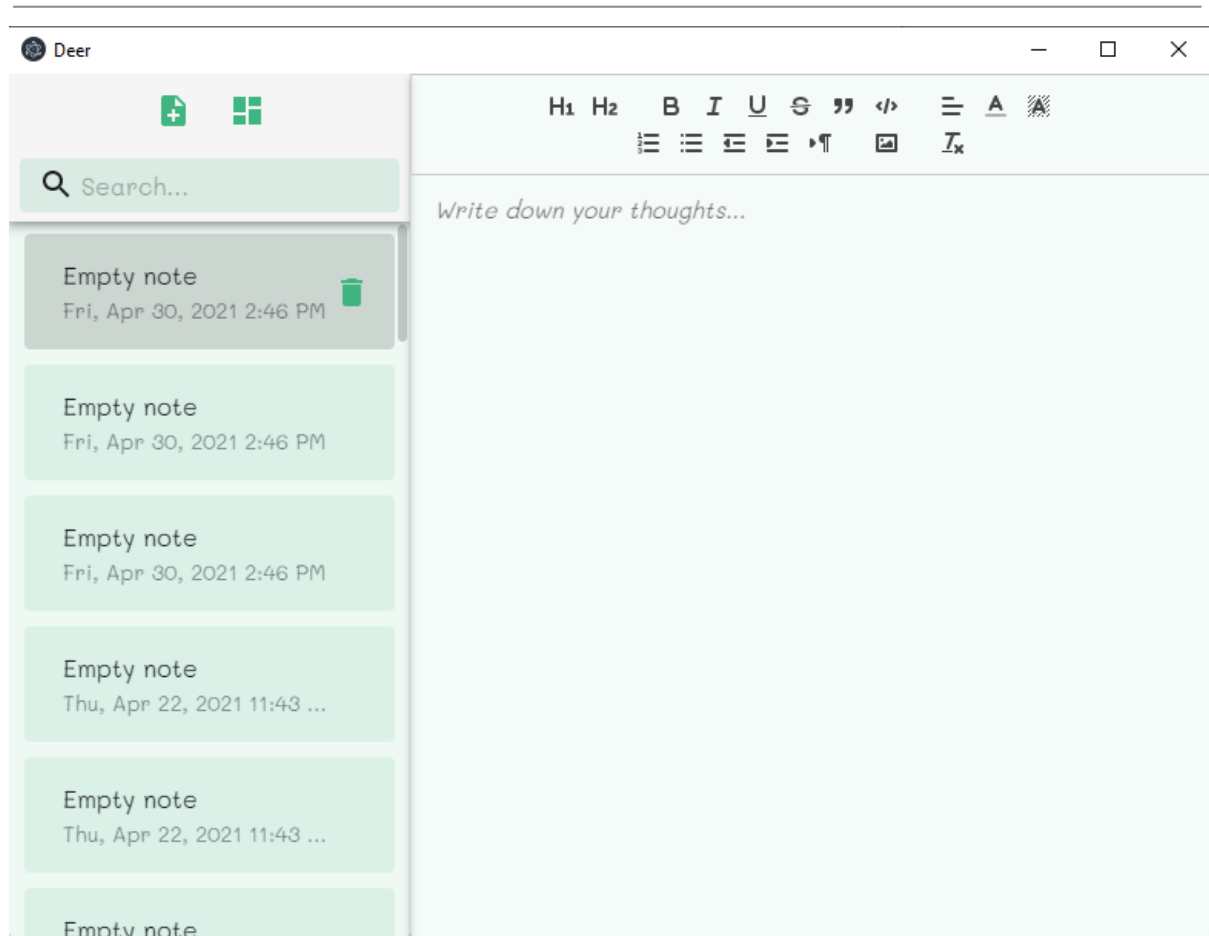



Figure 1 : Application Deer au départ

(7ème mission : 3 jours | Stocker les données JSON dans une base de donnée embarquée SQLite) : J'ai de bonnes bases en SQL mais je ne connaissais pas SQLite, je me suis donc documenté à son sujet et j'ai trouvé ça plutôt simple à implémenter une fois tout importé. Seulement, les importations ne se sont pas passées comme prévu, en effet, l'application sur laquelle je travaille est sur Electron et a plus de deux ans donc toutes les dépendances sont plutôt vieille et ça se voit surtout sur le code en React où ils ont utilisé des classes alors qu'aujourd'hui on utilise plutôt les fonctions fléchées pour les composants. J'ai donc eu beaucoup d'erreurs de dépendance, j'ai passé plus d'une journée à régler majoritairement ces erreurs. Mais au final, j'ai eu mon menu fonctionnel avec sa base de donnée SQLite pour son affichage.

Rapport de projet pour l'entreprise Opinaka : Menu personnalisable

Tillier Etienne



The screenshot shows a SQLite database interface. At the top, there's a header with 'SQLite' and some icons. Below it, a query editor shows the command: `SELECT * FROM MENU`. The results are displayed in a table with a 'json' header. Each row contains a JSON object representing a menu item.

json
<code>{"_id":"45e40796-1159-4894-828d-2764c47c0dd9","title":"spotclone2","status":"1"}</code>
<code>{"_id":"09e089e4-e36f-4b4a-b801-be9c83bc7f0e","title":"tmdb","status":"1"}</code>
<code>{"_id":"93aedab0-7087-4200-9eef-47dbe1e324f5","title":"Appointy","status":"1"}</code>
<code>{"_id":"a13a58a6-b963-45c6-8085-e7e2973214c8","title":"dental-clinic","status":"1"}</code>
<code>{"_id":"75beb380-160c-409f-ac36-043c7d04d698","title":"HouseCallDoc","status":"1"}</code>
<code>{"_id":"45e40796-1159-4894-828d-2764c445c0dd9","title":"Slack-clone","status":"1"}</code>

Figure 2 : Table menu de la base de donnée SQLite

On peut remarquer que la table menu ne contient qu'un seul attribut qui est en fait une chaîne (VARCHAR) qui correspond à du JSON où pour chaque ligne est stocké un item du menu avec son nom, son id pour son fonctionnement avec l'application et enfin un status que j'aborderai plus tard. Cela permet à partir d'une SELECT dans la base de donnée depuis l'application de récupérer des objets en JSON qu'on peut « parser » afin de pouvoir travailler avec eux en JavaScript. Dans mon cas, je me sers du « title » pour l'afficher sur le menu, l'id sert pour le fonctionnement de l'application et le status pour la visibilité de l'item.



Figure 3 : Menu affiché à partir de la base de donnée

(8ème mission : 3 jours | Insérer dans le menu les applications/clones trouvés dans les missions précédentes et les afficher sur l'application Electron) : Comme on peut le constater sur la figure 3, le menu contient déjà les applications stockées dans la base de donnée SQLite au format JSON. Les applications ne sont pas réellement stockées dans la base de donnée, seulement un nom est stocké en rapport avec l'application en question. J'ai créé un dossier dans mon application où je stocke tous les builds des applications ou clones que je veux afficher dans l'application, ainsi par le nom stocké dans la base de donnée, je peux accéder au fichier en question et lancer le build. Premièrement, j'affichais le build dans une fenêtre externe à Electron jusqu'à ce que tout fonctionne pour ensuite travailler sur l'intégration dans Electron. Pour cela, il fallait intégrer une vue qui permet d'afficher une page web. Or mon projet est sur Electron et React et ne comportait jusque là aucune vue, je me suis donc renseigné sur comment intégrer une vue dans cette application. Après de nombreuses erreurs d'intégrations, j'ai trouvé une bibliothèque compatible avec mon projet que j'ai donc intégré afin d'afficher les applications à la place de l'éditeur de texte présent jusque là.

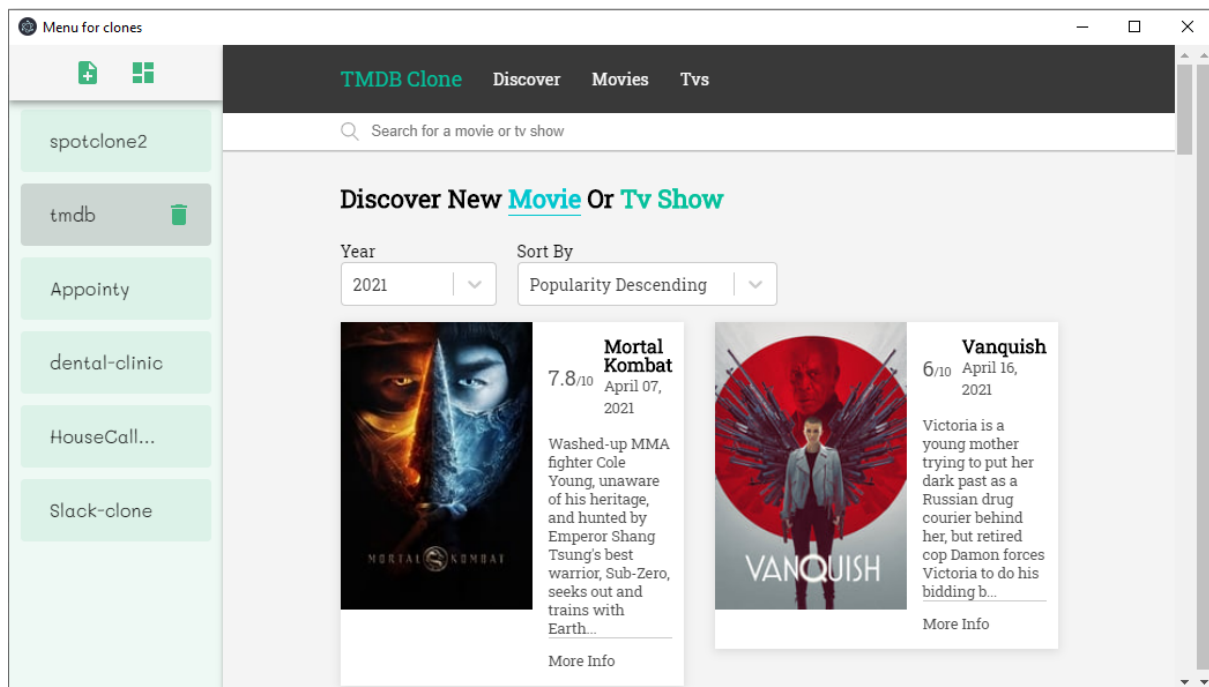


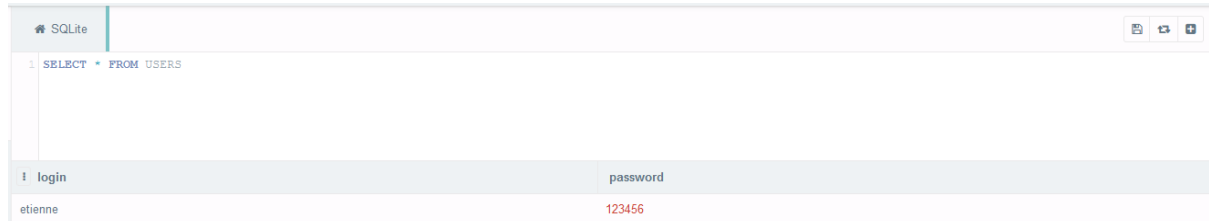
Figure 4 : Intégration de la vue dans l'application affichant un clone de TMDB

Au clique sur un item du menu, la vue affiche l'application sur laquelle on peut ensuite naviguer. Ainsi, il est possible de naviguer entre toutes les applications présentes dans le menu à travers la vue.

Rapport de projet pour l'entreprise Opinaka : Menu personnalisable

Tillier Etienne

(9ème mission : 1 jour | Créer une page de connexion qui permet d'accéder à l'application où les utilisateurs sont stockés sur une base de donnée SQLite) :



The screenshot shows a SQLite database interface. At the top, there's a header with 'SQLite' and some icons. Below it, a query editor shows the SQL statement: `SELECT * FROM USERS`. Below the query editor, a table displays the results of the query. The table has two columns: 'login' and 'password'. There is one row of data with the values 'etienne' and '123456'.

login	password
etienne	123456

Figure 5 : Base de donnée SQLite pour les utilisateurs

Pour cette mission, j'ai commencé par utiliser la base de donnée déjà existante pour le menu où j'ai juste rajouté une table contenant les utilisateurs avec leur mot de passe. Ensuite, j'ai ajouté les méthodes de connexion dans mon fichier de base de donnée et le tour était joué pour cette partie là. Il ne manquait plus qu'un visuel où l'utilisateur pourrait rentrer ses identifiant pour se connecter à l'application, pour cela j'ai été sur GitHub chercher une interface open-source et je l'ai adapté à ma situation ce qui n'était pas bien complexe. Cette interface était en JS pure et non en React, donc j'ai utilisé des IPC (Inter-Process Communication) pour qu'une fois la connexion validée sur la page en JS pure, cela envoie un message au processus principal d'electron (main) afin qu'il change l'url de la fenêtre en indiquant celui de l'application basé sur du React.

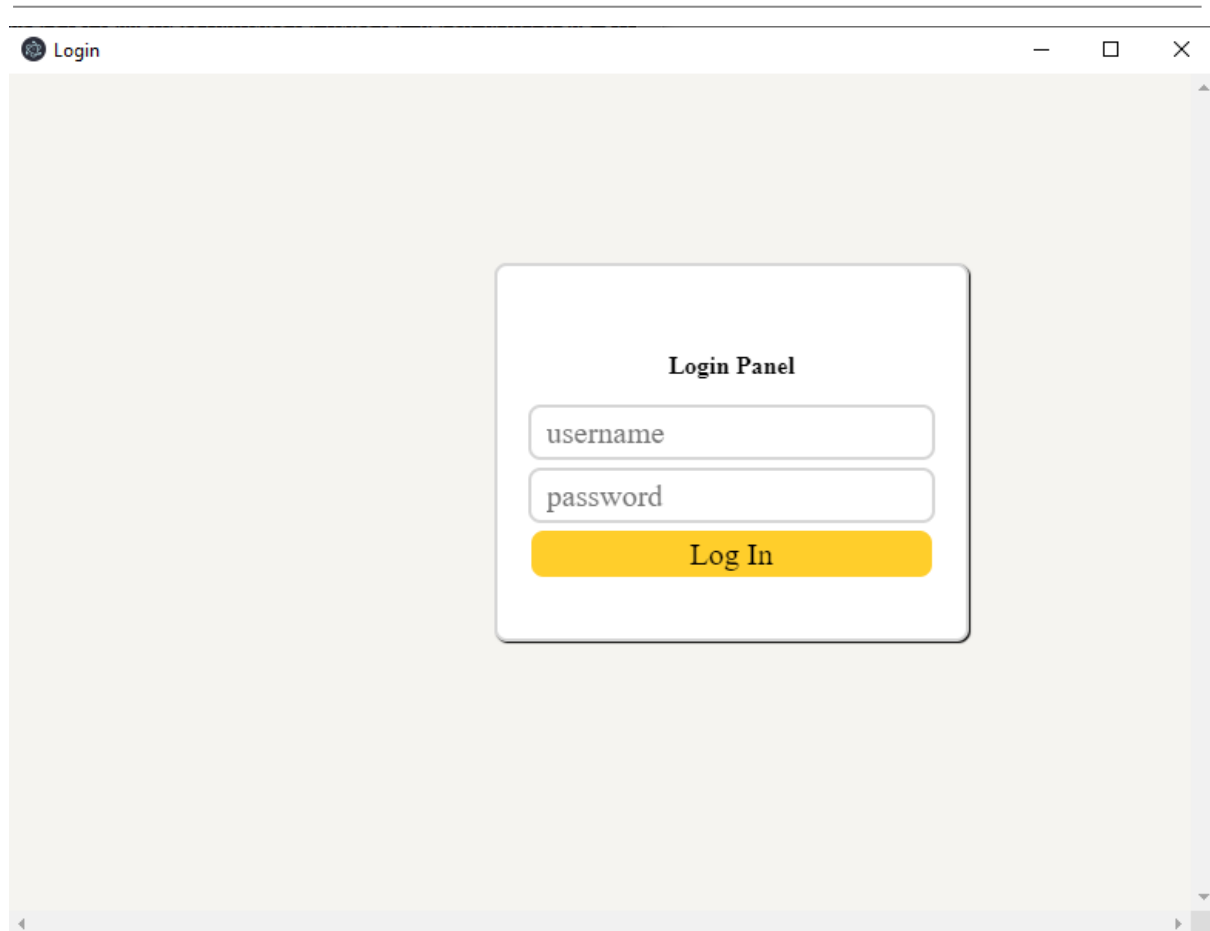


Figure 6 : Interface de connexion de l'application

Ainsi avec l'interface présentée dans la figure 6, les données rentrées à l'intérieur sont vérifiées par des méthodes accédant à la base de donnée SQLite pour valider ou non la connexion à l'application.

(10ème mission : 3 jours | Ajouter et adapter certaines fonctionnalités présentes dans l'application Deer pour l'utilisation du menu) : Pour la bonne réalisation de ces tâches, j'ai dû me familiariser davantage avec le code présent dans l'application car jusque-là je ne m'étais focalisé que sur la partie menu. J'ai commencé par réduire la largeur du menu à la demande de mon maitre de stage, ce qui à ma grande surprise ne fut pas si simple que ça car il s'agit d'un composant de la bibliothèque « material-ui » et je ne trouvais pas où était rangé le style de ce composant. J'ai donc fini par trouver un moyen afin d'écrire par-dessus le style déjà attribué à ce composant afin que la largeur soit réduite.

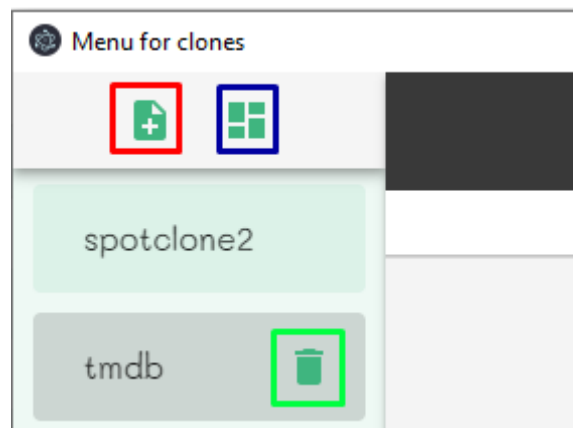


Figure 7 : Les boutons de l'application

Tous les boutons présents dans la figure 7 étaient déjà présents dans l'application de base. Le bouton encadré en rouge servait à la base à ajouter une nouvelle note, le bouton encadré en bleu à revenir au menu et le bouton encadré en vert à supprimer une note existante. Dans cette mission, il m'a été demandé de changer le bouton encadré en rouge ainsi que celui encadré en vert. Ce dernier ne doit plus supprimer les items du menu, ici les applications, mais seulement les cacher pour l'utilisateur. Le bouton en rouge quant à lui, doit permettre d'afficher tous les items du menu, même ceux cachés au préalable. Comme vous l'aurez certainement compris, c'est de là d'où provient le fameux « status » dans la table menu présent dans la base de donnée. Effectivement, si le status d'un item contient la valeur 1 (valeur par défaut), alors ce dernier sera affiché dans le menu, et au contraire, s'il contient la valeur 0, alors il ne sera pas affiché mais restera dans la base de donnée. Donc le bouton encadré en vert permet tout simplement de passer la valeur du status de l'item sélectionné à 0 et le bouton encadré en rouge permet de passer la valeur du status de chaque items présent dans la table menu à 1. Cette adaptation n'a pas été des plus simples car comme précisé précédemment, une base donnée (PouchDB) était déjà présente sur l'application et c'est elle qui gère tout sur l'application, sauf l'affichage du menu que j'avais déjà adapté. Ce qui fut en particulier compliqué, c'est qu'un composant React contient l'id de l'item sélectionné sur le menu et il le fait travailler dans beaucoup de méthodes en relation avec la base de donnée originelle de l'application. Donc même si je ne voulais que modifier les méthodes en rapport avec ma problématique, j'aurais obtenu des erreurs. J'ai donc lié tous le fonctionnement de l'application qui provient du menu à ma base de donnée ce qui m'a ensuite permis d'adapter sans trop de problèmes les boutons encadrés.

(11ème mission : 1 jours | Faire deux versions de l'app) : Étant donné que dans le menu est stocké des liens et des clones, il m'a été demandé de créer une version de l'application qui contient les clones et une autre qui contient les liens. Pour arriver à cela, j'ai évidemment copier l'application et j'ai simplement adapter en fonction de la version de l'application le lien src de la webview. De la même manière, j'ai adapté la base de donnée afin qu'elle ne stocke que des liens ou que des clones. La version pour les clones doit être en offline et donc ne dispose pas d'une page de connexion contrairement à la version pour les liens qui elle dispose d'une page de connexion.

(12ème mission : 2 jours | Implémentation de Firebase) : Avant de commencer cette mission, je ne connaissais rien à Firebase. Je me suis donc documenté sur internet et à l'aide de vidéo YouTube, ensuite j'ai pu créer un projet Firebase pour chaque version de l'application. A l'aide de ces projets Firebase, il m'est possible de stocker en online des données comme le menu des applications. Également, Firebase propose un système d'authentification, ce dernier permet de se connecter via un email ainsi qu'un mot de passe mais on peut aussi se connecter avec des réseaux sociaux comme Twitter, Facebook etc.. Dans un premier temps, je me suis focalisé sur les authentifications avec React, j'ai trouvé un bon moyen d'implémenter proprement le module d'authentification proposé par Firebase avec tout les types d'authentification que je souhaitais. J'ai mal compris ce que le maître de stage voulait, et j'ai donc proposé plusieurs moyens d'authentification comme Facebook et GitHub. Après la remarque de mon maître de stage, j'ai laissé seulement l'authentification par l'adresse mail. Il est donc possible désormais de se connecter en online par Firebase pour les deux versions. Pour la versions offline, elle contient toujours une page de connexion reliée à la base de donnée afin de pouvoir se connecter sans connexion internet. Une fois l'authentification complètement implémentée, j'ai pu me diriger sur la base de donnée (RealtimeDB). Il s'agit d'une base de donnée qui est finalement un objet en JSON et qui une fois lu dans le programme se transforme automatiquement en objet JavaScript. Le menu qui était stocké dans SQLite était déjà au format JSON, donc cela a été pratique pour le copier dans la base de donnée Firebase. Pour rendre compatible les applications avec cette nouvelle base de donnée, j'ai du implémenter de nouvelles méthodes qui font appel à cette base de donnée afin d'en récupérer les données et de pouvoir ensuite travailler avec dans le programme, de plus, j'ai aussi du revoir pas mal de mes anciennes méthodes, comme celles pour cacher ou faire réapparaître des éléments du menu car pour la version online, la base de donnée avait totalement changée. Pour cette partie là, j'ai rencontré quelques difficultés car je ne connaissais pas la documentation de Firebase pour la base de donnée, mais tout comme pour l'authentification, en prenant le temps de regarder la documentation et avec des vidéos, j'ai pu régler ces problèmes assez rapidement. Ainsi, les deux versions de l'application fonctionnent avec Firebase pour l'authentification comme pour accéder aux données, malgré que pour la versions locale, il existe encore la base de donnée SQLite pour la connexion ainsi que pour le menu.

(13ème mission : 4 jours | Synchronisation des bases de données pour la version locale) : Comme précisé ci-dessus, la version locale fonctionne avec deux bases de données, une locale (SQLite) et une online (Firebase). Ainsi, pour cette mission, il m'a été demandé de synchroniser les deux bases de données si l'utilisateur possède une connexion internet. Cela signifie de copier les données de la base de données Firebase dans la base de donnée SQLite. De cette manière, l'application locale, comme son nom l'indique, peut être utilisé à son plein potentiel seulement en local. Afin de réaliser cette mission, j'ai du évidemment créer des méthodes afin de récupérer les données dans Firebase pour les copier dans SQLite après la suppressions des données dans les tables synchronisées. Cette partie ne fut pas très compliquée car je l'avais déjà fait plus ou moins au préalable pour l'utilisation de mes anciennes méthodes. Cependant, cela ne m'a pas empêché d'avoir rencontré des problèmes, en effet, il faut que la synchronisation se fasse quand l'utilisateur se connecte à l'application. Cela signifie qu'elle doit se faire juste avant que cela affiche le contenu du menu, donc il y a un très court instant précis. Et ce qui est très important dans tout ça, c'est qu'il faut que le programme attende d'avoir synchroniser pour afficher

l'application, sinon ça veut dire que les données ne seront pas ou que partiellement copier dans la base de données SQLite. Pour remédier à cette problématique, après m'être documenté, j'ai choisi de procéder avec un système de promesse. J'ai donc adapté mes méthodes de fetching et de copie avec des promesses afin que chaque méthode attende une autre afin qu'il y ait un ordre et qu'aucune ne brule les priorités. Ainsi, la méthode de lancement de l'application attend que toutes les synchronisations soient faites pour se lancer. Afin de se coordonner pour l'entreprise, je devais avoir la même structure de donnée sur Firebase que mes collègues stagiaire. Cela fut compliqué pour moi étant donné que l'application de base que j'ai adapté fonctionne différemment de celle de mes collègues. Effectivement, chaque élément du menu contient un « id », de plus mon application ne contient pas de sous-menu (cela aurait pu être implémenté, mais cela aurait pris beaucoup de temps pour un résultat très moyen car ce n'est pas moi qui ai fait le design de l'application et il aurait fallu que j'adapte ce dernier afin d'y intégrer les sous-menus). Dans cette mission, également, je devais permettre l'accès à l'application pour plusieurs utilisateurs, c'est à dire créer des comptes et un système de connexion. Les utilisateurs doivent être stockés en local et aussi sur Firebase et donc comme les menus, être aussi synchronisés à la connexion d'un utilisateur. Ainsi un utilisateur, peut se connecter sur la version locale avec trois possibilités : Firebase, KeyCloak ou alors via la base de données SQLite. Pour ce qui est de KeyCloak, je rencontre un problème lors de la connexion, cela affiche une erreur. Sur internet, ils disent que ce n'est pas compatible avec Electron, donc je laisse le moyen de connexion pour le moment, mais ce dernier ne fonctionne pas. Enfin, afin de savoir s'il faut faire la synchronisation ou non, pour ne pas avoir d'erreur, il faut savoir si l'utilisateur est connecté à internet. Pour cela, ne sachant pas comment remédier à cela, j'ai cherché sur internet des bibliothèques permettant d'implémenter une solution, ainsi le programme vérifie et exécute la synchronisation si les conditions sont respectées.

(14ème mission : 3 jours | Changement d'architecture de la base de données) : Jusque là, j'avais pour mon application, plusieurs comptes pour se connecter à l'application en offline comme en online afin que ces derniers puissent accéder à un menu commun. Désormais, il m'a été demandé que pour chaque utilisateur, il existe un menu, donc chaque utilisateur doit posséder son propre menu. Pour arriver à cela, il a donc fallu changer toute la structure de la base de données. Dans Firebase, comme les authentifications se font avec un système particulier qui n'est pas relié à la base de données, il faut récupérer l'UID de connexion et parallèlement avoir ajouté les utilisateurs à la base de données en fonction de leur UID. C'est à dire, qu'on peut inscrire un utilisateur sur le système d'authentification de Firebase, mais il ne sera pas pour autant dans la base de données. Il faut donc l'ajouter manuellement dans cette dernière et on l'ajoute avec son UID comme nœud dans la base de données.

Rapport de projet pour l'entreprise Opinaka : Menu personnalisable

Tillier Etienne

Recherchez par adresse e-mail, numéro de téléphone ou ID utilisateur					Ajouter un utilisateur	↺	⋮
Identifiant	Fournisseurs	Date de création	Dernière connexion	UID utilisateur ↑			
etienne@test.com	✉	25 mai 2021	27 mai 2021	2wKnVHjVP7MFzejT6EFKX78GKk...			
pierre@test.com	✉	25 mai 2021	26 mai 2021	DRsVXIDrMAa3jXP6iilN3rRxiT33			
test@test.com	✉	25 mai 2021	27 mai 2021	RF5aMWxGyOUxv5urUifaGAwdZV...			
jean@test.com	✉	25 mai 2021	26 mai 2021	f7eifhntNpQs31MsJNiZhNfsTw2			
demo@test.com	✉	25 mai 2021	27 mai 2021	yrmHexeaAKb36cFDnOajlL4mi2v1			

Lignes par page : 50 1 – 5 of 5 < >

Figure 8 : Authentification Firebase

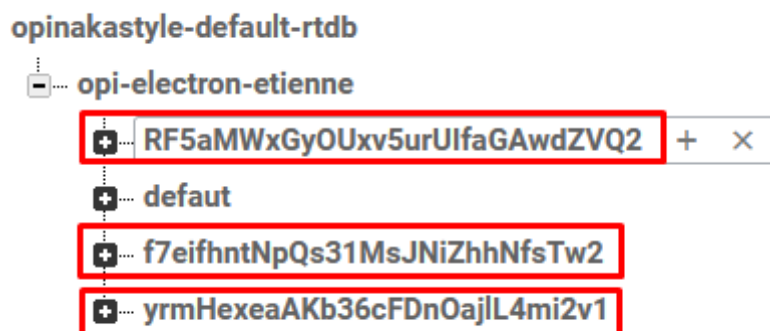


Figure 9 : Realtime DataBase Firebase (UID)

Ainsi, sur les figure 8 et 9, on remarque facilement que les UID de l'authentification Firebase sont retranscrites pour certaines dans la base de données comme nœud utilisateur. Cela signifie, que les utilisateurs ayant leur UID renseigné dans la base de données auront un compte pour se connecter en local car la base de donnée est copiée lors de la synchronisation. Ils auront également accès à un menu personnalisé.

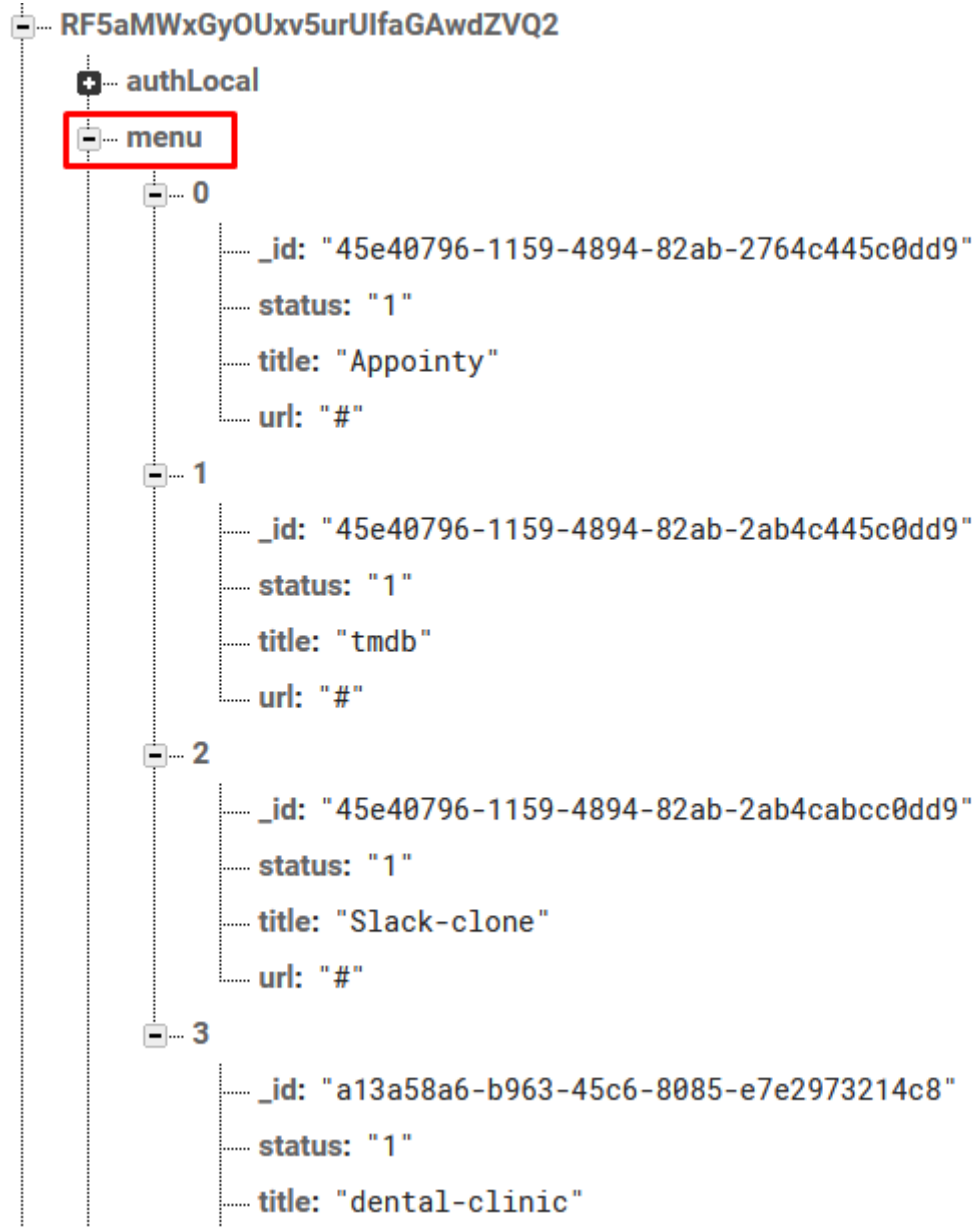


Figure 10 : Realtime DataBase Firebase Menu utilisateur

Avec la figure 10, on voit que chaque utilisateur possède son menu, ici on n'en voit qu'un car cela aurait été trop volumineux d'en afficher davantage. Pour faire fonctionner cette base de données avec l'application, il a déjà fallu que je m'attaque à l'architecture de la base de données SQLite, car en effet elle ne pouvait pas rester comme avant.

Rapport de projet pour l'entreprise Opinaka : Menu personnalisable

Tillier Etienne

1 SELECT FROM MENUS		
id	json	jsonStyle
RF5aMWxGyOUxv5urUlffaGAwdZVQ2	[{"_id":"45e40796-1159-4894-82ab-2764c445c0dd9","status":"1","title":...	{"indexPage":{"barreNav":{"background":"darkblue"},"body":{"backgrou...
default	[{"_id":"45e40796-1159-4894-82ab-2764c445c0dd","status":"1","title":...	{"indexPage":{"barreNav":{"background":"darkblue"},"body":{"backgrou...
f7eifmtNpQs31MsJNiZhNfsTw2	[{"_id":"45e40796-1159-4894-82ab-2764c445c0dd","status":"1","title":...	{"indexPage":{"barreNav":{"background":"darkblue"},"body":{"backgrou...
ymrHexeaAKb36cFDnOajlL4mi2v1	[{"_id":"75beab80-160c-409f-ac36-043c7d04d698","status":"1","title":...	{"indexPage":{"barreNav":{"background":"darkblue"},"body":{"backgrou...

Figure 11 : Base de donnée SQLite (table menus)

Dans la figure 11, on remarque qu'il y a plusieurs attributs, on fera abstraction sur « jsonStyle » pour le moment. L'attribut « id » correspond à l'UID des utilisateurs enregistrés dans la base de données Firebase (après synchronisation) et l'attribut « json » correspond au menu de l'utilisateur au format JSON comme dans la précédente table « menu ». Ainsi, il n'y a pas besoins de se connecter à internet pour pouvoir accéder à son menu car tout est stocké localement. Dans le cas où l'utilisateur se connecte via Firebase, alors je récupère dans un useState React la valeur de son UID et je la transmet de composant en composant afin qu'elle puisse être utilisée partout dans le programme. De cette manière, après la synchronisation, l'affichage du menu dépend d'une méthode qui prend l'UID en paramètre et affiche le menu présent dans l'attribut « json » de la table « MENUS » là où l'UID est égale à l'id de la table.

```
const selectAll = (callback,id) => {
  console.log("id selected = " + id);
  let sql = "SELECT json FROM MENUS WHERE id = ?";
  db.get(sql, [id], (error, row) => {
    var temp = [];
    if (row) {
      let data = JSON.parse(row.json);
      for (let i = 0; i < data.length; i++){
        temp.push(data[i]);
      }
      console.log("temps = " + temp);
      callback(temp);
    }
  });
}
```

Figure 12 : Méthode « selectAll »

Dans la figure 12 est représentée la méthode « selectAll » qui permet de sélectionner dans la base de données SQLite un menu en fonction de l'id entré en paramètre et de transmettre ce menu dans une méthode callback indiquée aussi en paramètre. De cette manière, j'ai fait en sorte que l'id en paramètre soit l'UID de l'utilisateur connecté et le callback est une méthode d'affichage du menu dans l'application. Seulement, jusqu'à présent, lorsqu'on se connecte localement à l'application, on accédait au menu commun et on ne possédait pas d'UID.

(15ème mission : 2 jours | Un compte local pour chaque utilisateur) : Il m'a été demandé de fournir un compte local pour chaque utilisateur. Ce compte est indépendant du

Rapport de projet pour l'entreprise Opinaka : Menu personnalisable

Tillier Etienne

compte en ligne sur Firebase mais pas totalement. En effet, il doit avoir des identifiants personnalisables depuis la base de données Firebase mais en même temps, il doit permettre d'accéder au même menu. Pour cela, en plus de stocker pour chaque utilisateur Firebase un menu, on stocke également un compte local qui sera synchronisé dans une table user SQLite.

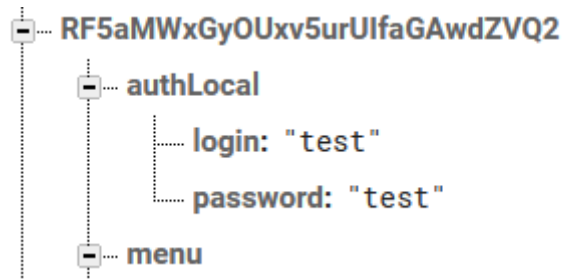


Figure 13 : Realtime DataBase Firebase AuthLocal

1 SELECT * FROM USERS

login	password	menuId
jean	jean	ymHexeaAKb36cFDnOajL4mi2v1
demo	demo	f7eihtntNpQs31MsJNiZhhtNfsTw2
test	test	RF5aMWxGyOUxv5urUlfaGAwdZVQ2
default	default	default

Figure 14 : SQLite table « USERS »

Ainsi, la connexion locale se sert de la table « USERS » présentée dans la figure 14, cette table est synchronisée de la même manière que la table des menus. J'ai également mis dans cette table l'attribut « menuId » afin que si l'utilisateur se connecte en local, le programme récupère l'id afin d'afficher le menu en question présent dans la table « MENUS ».

(16ème mission : 1 jour | Un compte par défaut) : Comme je l'ai dit précédemment, Firebase propose un système d'authentification qui n'a rien à voir avec la base de données RealtimeDB. Donc lorsque j'utilise ce système d'authentification dans mon application, il vérifie dans les comptes présent dans ce module et non pas dans la base de données. Or, il est possible qu'un compte soit ajouté dans ce module sans pour autant que son UID soit présent dans la base de données Firebase et donc non synchronisés dans SQLite. C'est pourquoi il m'a été demandé de créer un compte avec pour id « défaut », afin que si l'authentification par Firebase est confirmée mais que son UID n'est pas présent dans la base de données Firebase, alors la connexion se fera avec l'id « défaut » et donc aura accès à un menu par défaut. On peut voir cet id dans la figure 9. J'ai donc adapter ma méthode de connexion avec firebase afin qu'elle vérifie avant de transmettre son UID dans le programme que son UID est tout d'abord bien présent dans la base de données sur Firebase. Si ce n'est pas le cas, alors l'utilisateur se connectera mais avec l'id « défaut » et

donc aura accès à un menu commun avec tous les utilisateur bénéficiant de cet id. Comme tous les UID stockés sur la base de données Firebase, ils possèdent une authentification locale, un menu ainsi qu'un style dont je parlerai plus tard. Je dois faire en sorte que la connexion en locale en utilisant les identifiants de connexion de l'utilisateur par défaut ne soit pas possible car cette connexion est uniquement réservée pour les utilisateurs se connectant par Firebase. Donc même si une authentification locale pour l'id défaut est présente dans la base de données SQLite, j'ai bloqué sa connexion en vérifiant l'id de connexion à chaque tentative, s'il s'agit de « défaut », alors cela indique une erreur.

Résumé

Ce rapport concerne le déroulement et la réalisation d'un projet réalisé à l'occasion d'un stage en entreprise effectué durant le 4ème semestre de mon année universitaire 2020 / 2021 à l'IUT informatique de Montpellier. Pour davantage de précision, il s'agit d'un projet développé en JavaScript sous Electron avec le framework React qui permet l'accès à un menu personnalisable où des sites web sont accessibles directement depuis l'application. Il comporte aussi un système de connexion qui lui permet d'accéder au menu. Ce projet contient une base de données locale SQLite ainsi qu'une base de données en ligne Firebase. Ce rapport détaille également comment a été géré le projet. Il contient un cahier des charges, un rapport technique, un manuel d'utilisation ainsi qu'un rapport d'activité.

Mots clés : JavaScript, React, Electron, Firebase, SQLite, menu personnalisable, connexion

Abstract

This report is about the development and the realization of a project realized during the 4th semester of my academic year 2020 / 2021 at the Montpellier's Computer Science IUT. For more precision, it is a project developed in JavaScript under Electron with the React framework which allows the access to a customizable menu where websites are accessible directly from the application. It also includes a connection system that allows it to access the menu. This project contains a local SQLite database as well as an online Firebase database. This report also details how the project was managed. It contains a specification, a technical report, a user manual and a progress report.

Keywords : JavaScript, React, Electron, Firebase, SQLite, customizable menu, login