

# Documentation Technique



<u><i>Table des matières</i></u>
----------------------------------

❖	<i>I. Introduction.....</i>
❖	<i>II. Cahier des charges .....</i>
❖	<i>III. Diagramme des acteurs de l'application.....</i>
❖	<i>IV. Arborescence.....</i>
❖	<i>V. Différentes Fonctionnalités.....</i>

# I. Introduction

---

Le laboratoire **Galaxy Swiss Bourdin (GSB)** est issu de la fusion entre le géant américain Galaxy (spécialisé dans le secteur des maladies virales dont le SIDA et les hépatites) et le conglomérat européen **Swiss Bourdin** (travaillant sur des médicaments plus conventionnels), lui-même déjà union de trois petits laboratoires.

## **Domaine d'étude**

L'entreprise souhaite porter une attention nouvelle à sa force commerciale dans un double objectif qui est d'obtenir une vision plus régulière et efficace de l'activité menée sur le terrain auprès des praticiens.

Les déplacements et actions de terrain menées par les visiteurs engendrent des frais qui doivent être pris en charge par la comptabilité. On cherche à agir au plus juste de manière à limiter les excès sans pour autant diminuer les frais de représentation qui font partie de l'image de marque d'un laboratoire. Chez Galaxy, le principe d'engagement des frais est celui de la carte bancaire au nom de l'entreprise. Chez Swiss-Bourdin, une gestion forfaitaire des principaux frais permet de limiter les justificatifs. Pour tout le reste, le remboursement est fait après retour des pièces justificatives.

Une gestion unique de ces frais et remboursement pour l'ensemble de la flotte visite est souhaitée.

Les visiteurs récupèrent une information directe sur le terrain. Ceci concerne aussi bien le niveau de la confiance qu'inspire le laboratoire que la lisibilité des notices d'utilisation des médicaments ou encore les éventuels problèmes rencontrés lors de leur utilisation, etc. Ces informations ne sont actuellement pas systématiquement remontées au siège, ou elles le sont dans des délais jugés trop longs. Le service rédaction qui produit les notices souhaite avoir des remontées plus régulières et directes. Ceci permettra également au service labo-recherche d'engager des évaluations complémentaires.

L'application permet aux visiteurs médicaux lors de leurs déplacements et à tout moment de saisir et modifier leurs fiches de frais au forfait et hors forfait afin d'être remboursés.

**L'objet de ce document est de définir les spécifications fonctionnelles détaillées de l'application GSB.**

**Les spécifications fonctionnelles détaillées ont pour but de décrire précisément l'activité et toutes les fonctionnalités prévues lors de la phase de conception sont précisées dans ce document en indiquant l'implémentation de ces fonctionnalités dans l'application.**

# II. CAHIER DES CHARGES

---

## **Définition de l'objet**

Le suivi des frais est actuellement géré de plusieurs façons selon le laboratoire d'origine des visiteurs. On souhaite uniformiser cette gestion. L'application doit permettre d'enregistrer tout frais engagé pour l'activité directe (déplacement, restauration et hébergement) et de présenter un suivi daté des opérations menées par le service comptable (réception des pièces, validation de la demande de remboursement, mise en paiement, remboursement effectué).

## **Forme de l'objet**

L'application Web destinée aux visiteurs, délégués et responsables de secteur sera en ligne, accessible depuis un ordinateur. La partie utilisée par les services comptables sera aussi sous forme d'une interface Web.

## **Accessibilité/Sécurité**

L'environnement doit être accessible aux seuls acteurs de l'entreprise. Une authentification préalable sera nécessaire pour l'accès au contenu.

## **Architecture**

L'application respectera une architecture précise à définir.

## **Ergonomie**

Les pages fournies ont été définies suite à une consultation. Des améliorations ou variations peuvent être proposées.

## **Codage**

Le document "GsbWebTechnique" présente des règles de bonnes pratiques de développement utilisées par le service informatique de GSB pour encadrer le développement d'applications en PHP et en faciliter la maintenance. Les éléments à fournir devront respecter le nommage des fichiers, variables et paramètres, ainsi que les codes couleurs et la disposition des éléments déjà fournis.

## **Environnement**

L'utilisation de bibliothèques, API ou frameworks est à l'appréciation du prestataire.

## **Modules**

L'application présente deux modules : • enregistrement et suivi par les visiteurs • enregistrement des opérations par les comptables

## **Documentation**

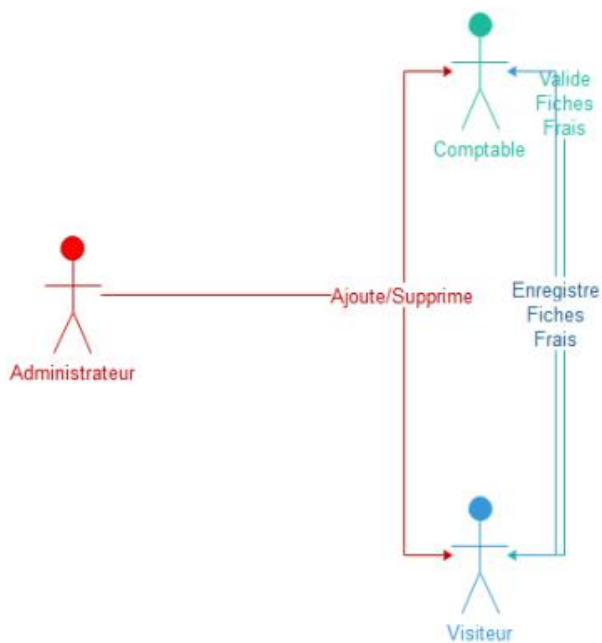
La documentation devra présenter l'arborescence des pages pour chaque module, le descriptif des éléments, classes et bibliothèques utilisées, la liste des frameworks ou bibliothèques externes utilisés

## **Responsabilités**

Le commanditaire fournira à la demande toute information sur le contexte nécessaire à la production de l'application ainsi qu'une documentation et des sources exploitables pour la phase de test : base de données exemple, modélisation... Le prestataire est à l'initiative de toute proposition technique complémentaire. Le prestataire fournira un système opérationnel, une documentation technique permettant un transfert de compétence et un mode opératoire propre à chaque module.

# III. DIAGRAMME DES ACTEURS DE L'APPLICATION

Le diagramme ci-dessous décrit le schéma des acteurs de l'application de gestion des frais.



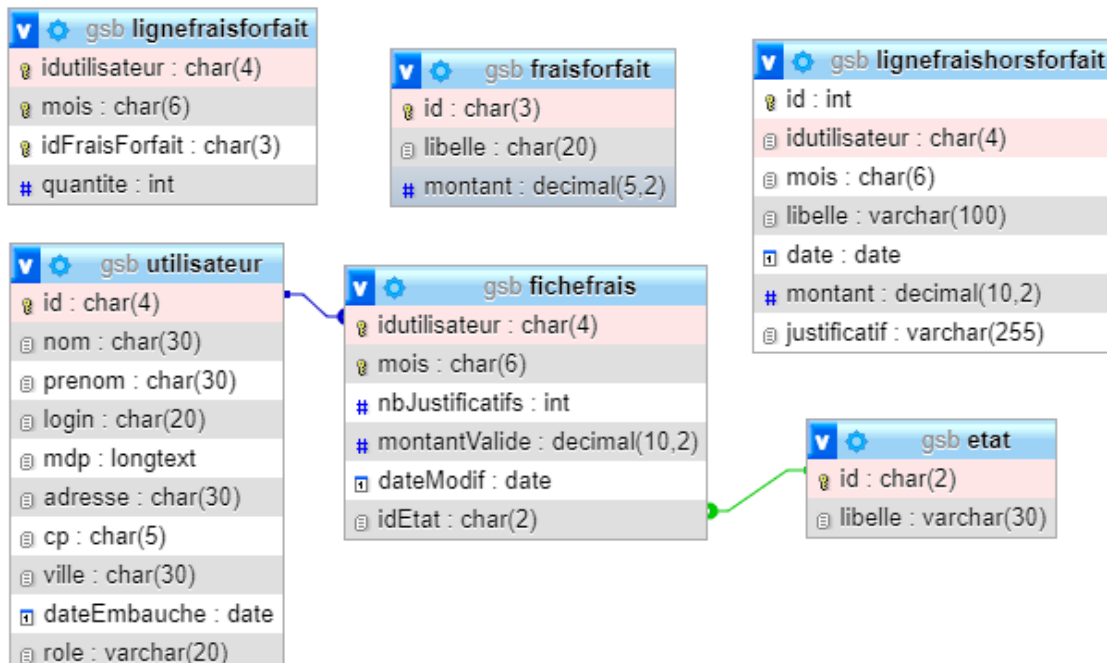
**Les responsabilités de chacun des acteurs de l'application sont décrites ci-dessous :**

- Le profil visiteur permet de saisir et consulter des fiches de frais
- Le profil comptable permet de valider, de modifier et de refuser des fiches de frais

**Le profil Administrateur permet d'ajouter, de modifier et de supprimer des utilisateurs**

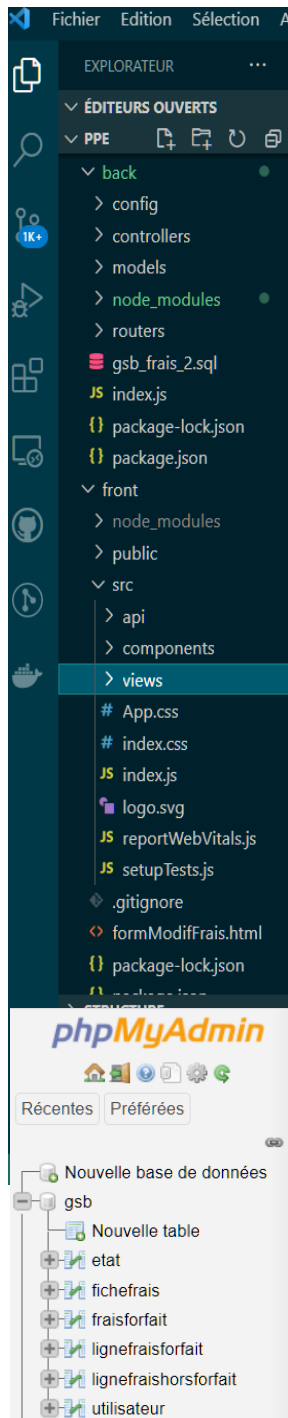
## Modèle logique de données de l'outil

Le diagramme ci-dessous décrit le modèle de données de l'application de gestion de frais. Les informations manipulées dans l'outil n'impactent pas le modèle de données.



# IV. ARBORESCENCE

L'application web a été réalisée avec le **Framework React.js** qui permet de créer des applications web et mobile en disposant d'une multitude de bibliothèque avec une communauté en pleine expansion :



## Back (partie interaction)

**config** : Permet une connexion à la base de données et à la page d'authentification de notre projet

**controllers** : Permet de lancer des actions qui seront réutilisées dans nos composants (gestion de fiches de frais)

**models** : Représente les données avec lesquelles l'utilisateur peut réagir (requête dans la BDD)

**node\_modules** : c'est une collection de fonctions et d'objets JavaScript

**routers** : Permet de définir des URL pour laisser l'utilisateur se déplacer dans le composant approprié au sein du projet

## Front (partie interface)

**node\_modules** : c'est une collection de fonctions et d'objets JavaScript

**public** : Dossier public de notre site, les fichiers de ce dossier sont accessibles sans restriction.

**src** : Contient nos API et composants

**api** : Permet de communiquer avec la BDD (GET/POST)

**components** : Contient nos différents composants (footer/header/BillList)

**views** : Permet de créer des interfaces utilisateurs interactives avec des pages indépendantes les unes des autres en utilisant des composants

**index.js** : gère le démarrage de l'application, le routage et d'autres fonctions de l'application

**package.json** : Description des bibliothèques utilisées avec le numéro de version

La base de données est gérée sur phpMyAdmin en local. Elle contient 6 tables qui sont liées. Ex : Chaque fiche de frais est liée à un visiteur. Une fiche a elle-même une ou plusieurs lignes de frais forfait ou hors-forfait. La table utilisateur permet de définir les comptes d'accès à notre application web

```
{ package.json > {} scripts
unknown, 2 weeks ago | 1 author (unknown)
{
  "name": "ppe1",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  > (débogage)
  "scripts": {
    "test": "echo \\\"Error: n
    "start": "node index.js"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.17.1",
    "jsonwebtoken": "^8.5.1",
    "mysql": "^2.18.1",
  }
}
```

SCRIPTS NPM

- back\\package.js... 1
  - test - back
  - start - back
- front\\package.js... 1
  - start - front

**Package.json** contient nos dépendances et nos scripts dans notre utilisation. Il contient son nom , sa version , sa description lorsqu'on créer le projet sous **Visual studio** avec la commande **npm init**

Notre commande pour lancer le projet est "**node index.js**" pour la raccourcir , on créer un script start qui permettra d'effectuer la commande "**taper node index.js**" en cliquant simplement sur start.

Nos dépendances sont :

**Express** : Express.js est un framework pour construire des applications web basées sur Node.js<sup>2</sup>. C'est de fait le framework standard pour le développement de serveur en Node.js

**Jsonwebtoken** : Il permet l'échange sécurisé de jetons (tokens) entre plusieurs parties. Cette sécurité de l'échange se traduit par la vérification de l'intégrité des données à l'aide d'une signature numérique.

**Mysql** : MySQL est l'un des systèmes de gestion de bases de données



**NodeJS** est une plateforme logicielle qui permet d'exécuter du JavaScript côté serveur. Il se démarque par l'utilisation d'un système de boucle d'évènement qui permet l'exécution d'opérations de manière asynchrone.

**Nodejs** nous sert également lorsque nous installons les dépendances ou utilisons le terminal pour la majorité des commandes car les commandes "**npm**" garantissent une gestion des dépendances et une interface en ligne de commandes

### Commandes

**Node index.js** : démarre l'index.js

**Npm init** : initialise le projet avec les infos(description , nom ,...)

**Npm install** : installe une dépendance

**Npm uninstall** : désinstalle une dépendance

# V. LES DIFFERENTES FONCTIONNALITES

## Authentification de l'utilisateur

### Se connecter

Pseudo

Mot De Passe

Se connecter

Mot De Passe Oublié

Avant de pouvoir visualiser ses frais, le visiteur doit d'abord s'authentifier à l'aide de son login et de son mot de passe dans le formulaire de connexion. Lorsqu'un utilisateur est authentifié, son nom et son prénom apparaissent dans l'entête.

Lors de la demande de connexion (Quand le bouton "se connecter" a été cliqué avec les informations remplies au préalable)

```
import Login from './views/Login'
import ProtectedRoute from './components/protectedRoute/ProtectedRoute'
const Root = () => {
  return (
    <Router>
      <Switch>
        <Route exact path="/" component={Login}/>
        <ProtectedRoute exact path="/bills" component={Bills} />
        <ProtectedRoute exact path="/create" component={CreateBill} />
      </Switch>
    </Router>
  )
}
```

```
3 import Footer from '../components/footer/Footer'
4 import LoginForm from '../components/login/Login'
5 import React from 'react'
6
7 class Login extends React.Component {
8   constructor(props) {
9     super (props)
10   }
11   render() {
12     return (
13       <div class="container-fluid">
14         <Footer/>
15         <LoginForm/>
16       </div>
17     )
18   }
19 }
20
21 export default Login;
```

Dans **index.js** , on importe la route Login a la racine de notre site web d'où le "/" depuis le répertoire contenant **views/login**

Dans le répertoire **views/login** On assemble le **footer**, le formulaire de connexion et la mise en forme. On importe également le composant login depuis le répertoire **components/login/login**

```

import './Login.css'
import React from 'react'
import { withRouter } from 'react-router-dom'
import { getToken } from '../api/auth'

You, seconds ago | 2 authors (You and others)
class Login extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      mdp: '',
      login: ''
    }
  }

  handleChange(e) {
    let { name, value } = e.target
    this.setState({
      [name]: value
    })
  }

  async login() {
    try {
      let { decoded, token } = await getToken({ login: this.state.login, mdp: this.state.mdp })

      if (decoded) {
        console.log(decoded)
        localStorage.setItem('id', decoded.id)
        localStorage.setItem('token', token)
        this.props.history.push('/bills')
      }
    } catch (e) {
      console.error(e)
    }
  }
}

```

Dans le répertoire **components/login/login** on crée un formulaire qui permettra de créer une interface à l'instance de connexion.

**This.state** permet de sauvegarder les valeurs tapées par l'utilisateur .

**Handlechange(e)** est déclenché à chaque frappe pour mettre à jour l'état local React, la valeur affichée restera mise à jour au fil de la saisie.

**Async login** permet de réaliser une promesse ( si la valeur n'est pas rejetée .L'id et le token valide sera envoyé et la page suivante sera lancée )

On importe **getToken** depuis le répertoire **api/auth** qui enverra les données de l'utilisateur vers le serveur

On cherche avec la variable **searchByLogin** si le mot de passe coïncide avec le mot de passe de l'utilisateur dans la base de données, le **token** est créé et l'accès est garanti. L'utilisateur est à présent connecté

```

var jsonwebtoken = require('jsonwebtoken')

const createAccessToken = (user) => { // creation d'un token
  return new Promise((resolve, reject) => {
    if (user.id === undefined) reject({error: 'Invalid Credentials'})
    else {
      const signedUser = {
        email: user.login,
        id: user.id
      }

      jsonwebtoken.sign(signedUser, "ppe", {expiresIn: '1d'}, (error, token) => {
        if (error) reject(error)
        resolve(token)
      })
    }
  })
}

module.exports = createAccessToken

```

```

const userModel = require('../models/users.model')
const createAccessToken = require('../config/token')
const searchByLogin = (request, response) => {

  const { login, mdp } = request.body
  userModel.searchByLogin(login, async(err, result) => {
    try {
      if (err) throw err
      else {
        if (mdp == result[0].mdp) {
          const token = await createAccessToken(result[0])
          response.json({ token })
        } else {
          response.status(403).send({ error: 'Forbidden' })
        }
      }
    } catch (e) {
      response.status(403).send({ error: 'Forbidden' })
      console.error(e)
    }
  })
}

module.exports = {
  searchByLogin
}

```



### Gestion de fiches de frais

Après s'être connecté, notre utilisateur peut modifier ses fiches de frais existantes ou en ajouter/supprimer

id utilisateur	Mois	Justificatifs	Montant	Date de modification	Etat	Action
1	202011	1	50.00	2020-11-18T00:00:00.000Z	VA	MODIFIER
1	202105	5	70.00	2021-04-04T00:00:00.000Z	CR	MODIFIER



L'interface se présente ainsi et permet à l'utilisateur de renseigner les informations notamment les nuits/repas et le kilométrage pour se faire rembourser ses frais dans le forfait ou hors forfait.

Les fiches de frais seront enregistrées dans la Base de données. Le comptable n'aura plus qu'à opérer pour gérer et rembourser ses fiches de frais avec également la possibilité d'acquisition de fichier envoyer par le praticien

#### MODIFICATION DE FRAIS

Type	Quantité	Montant	Total
Nuitées	<input type="text" value="1"/>	80€	80€
Repas	<input type="text" value="2"/>	25€	50€
Kilométrage	<input type="text" value="3"/>	0,62€	1.8599999999999999€

#### MODIFICATION DE FRAIS HORS FORFAIT

Date	Libelle	Montant	Justificatifs
<input type="text" value="15/11/2020"/> 	<input type="text" value="restaurantds"/>	<input type="text" value="70,00"/> €	<input type="text" value="Choisir un fichier"/> <input type="text" value="Aucun fichier choisi"/> 

```
import React from 'react'
import './BillsList.css';
import * as fromBillsApi from '../api/bills'
import Modal from 'react-bootstrap4-modal'

You, seconds ago | 2 authors (unknown and others)

class BillsList extends React.Component {

  constructor(props) {
    super(props)

    this.state = {
      bills: [],
      visible: false,
      rows: [],
      nightsQty: "",
      Kilometrage: "",
      Repas: "",
      dateHF : "",
      libelleHF : "",
      montantHF : ""
    }
  }
}
```

```
  handleChange(e){
    e.preventDefault()
    let name = e.target.name
    this.setState({
      [name]: e.target.value
    })
  }

  async postFiche(){
    let Kilometrage = await fromBillsApi.postBills({idutilisateur: 'a1
    let Repas = await fromBillsApi.postBills({idutilisateur: 'a131', m
    let nightsQty = await fromBillsApi.postBills({idutilisateur: 'a131
  }
}
```

```
export const postBills = async () => {
  let response = await fetch('http://localhost:3001/fiches/IdFraisForfait', {
    method: 'POST',
    headers: {
    }
  })
  let bills = await response.json()
  return bills
}
```

```
  async update() {
    let id = localStorage.getItem('id')
    await fromBillsApi.putLigneFraisForfait(id, this.state.mois, this.state.idFraisForfait)
    this.setState({
      visible : !this.state.visible
    })
  }
}
```

```
  async componentDidMount() {
    let bills = await fromBillsApi.getBills()
    this.setState({ bills: bills.result }, () => console.log(this.state))
  }

  showModal() {
    this.setState({
      visible: !this.state.visible
    })
  }
}
```

```
  addRow() {
    this.setState({
      rows: [...this.state.rows, {name: '', date: '', qty: '', files: ''}]
    })
  }
}
```

```
  removeRow(i) {
    let newRows = this.state.rows
    newRows.splice(i, 1)
    this.setState({
      rows: newRows
    })
  }
}
```

Tout d’abord, nous avons créé un tableau (<th>) dans notre fichier Billlist.js . Nous avons donc notre tableau qui apparait sous la forme ci-dessus. Ensuite, nous récupérerons les entrées des utilisateurs (Les nuits, Kilométrage, Repas...) avec “this.state”

Comme précédemment **Handlechange(e)** est déclenché à chaque frappe pour mettre à jour l’état local React, la valeur affichée restera mise à jour au fil de la saisie.

**Async postFiche()** permet de réaliser une promesse qui permet d’envoyer les données dans la base de donnée si l’utilisateur rentre des données et les valide dans les champs .

La variable **postBills** permet d’envoyer la requête dans la Base de données quand l’utilisateur a enregistré ses fiches de frais

**Async update()** est une fonction qui permet de mettre a jour les modifications entrées par l’utilisateur

**showModal()** permet de faire apparaitre la fenêtre lorsqu’on clique sur le bouton de création de fiche de frais

**addRow()** permet de rajouter une ligne avec les champs ouverts pour l’utilisateur.

**RemoveRow()** permet de supprimer tout simplement une ligne avec son utilisation du **.splice**

```

export const getLigneFraisHorsForfait = async (id, mois) => {
  let response = await fetch('http://localhost:3001/fiches/lignefraishorsforfait/'+ id + ' '
    method: 'GET',
    headers: {
      'Accept' : 'application / json',
      'Content-Type' : 'application/json'
    }
  ))
  let ligneFraisHorsForfait = await response.json()
  return ligneFraisHorsForfait
}

export const postLigneFraisForfait = async (ligneFraisForfait) => {
  let response = await fetch('http://localhost:3001/fraisforfait/lignefraisforfait/new', {
    method: 'POST',
    headers: {
      'Accept' : 'application / json',
      'Content-Type' : 'application/json'
    },
    body: JSON.stringify(ligneFraisForfait)
  })
  let ligneFraisForfaits = await response.json()
  return ligneFraisForfaits
}

```

Dans le répertoire **src/api/bills** on définit la méthode **GET** qui permet de récupérer les données depuis la base de donnée pour les afficher a l'utilisateur pour pouvoir les modifier à sa guise puis la méthode **POST** qui permet d'envoyer les nouvelles données sur la base de données lors d'une création , d'une modification ou d'une suppression

```

const updateFiche = (idutilisateur, mois, fiche) => {
  var query = 'UPDATE fichefrais SET nbJustificatifs = ' + fiche.nbJustificatifs + ' WHERE idutilisateur = ' + idutilisateur + ' AND mois = ' + mois
  var values = [fiche.nbJustificatifs, idutilisateur, mois]
  connection.query(query, values, callback)
}

const deleteFiche = (idutilisateur, mois, fiche) => {
  var query1 = 'DELETE FROM fichefrais WHERE idutilisateur = ' + idutilisateur + ' AND mois = ' + mois
  connection.query(query1, idutilisateur, mois, callback)
}

const search = (idutilisateur, mois, callback) => {
  connection.query('SELECT * FROM fichefrais WHERE idutilisateur = ' + idutilisateur + ' AND mois = ' + mois, callback)
}

```

Dans **model/fichesdefrais.model** on définit les variables contenant les requêtes utilisées pour mettre a jour la base de données pour les modifications/suppressions/mises a jour

```

const deleteLigneFraisForfait = (request, response) => {
  ficheModel.deleteLigneFraisForfait(request.params.id, request.params.mois, response)
  if (err) response.json(err)
  else response.json({result})
}

//Fonction Ligne Hors forfait
const addLigneFraisHorsForfait = (request, response) => {
  const body = request.body
  ficheModel.addLigneFraisHorsForfait(body, (err, result) => {
    if (err) response.json(err)
    else response.json({result})
  })
}

```

Dans **controller.fichedefrais.controller**, on définit l'utilisation de la variable couplée avec une action créer dans le **model** ce qui va permettre de modifier, supprimer ou modifier différents types de fiches de frais . Si l'action n'est pas menée à sa réalisation, on obtiendra une erreur

## Gestion de fiches de frais

La création d'un nouvel utilisateur se passe tout d'abord dans la base de données .

L'administrateur ou comptable si les droits lui sont accordés pourra a sa guise ajouter des nouveaux utilisateurs en renseignant simplement les champs dont les plus importants (login,mdp)



```
const addUser = (user, callback) => {
  connection.connect()
  var query = 'INSERT INTO utilisateur (id, nom, prenom, login, mdp) VALUES (' + user.id + ', ' + user.nom + ', ' + user.prenom + ', ' + user.login + ', ' + user.mdp + ')'
  // EXECUTION DE LA REQUETE
  connection.query(query, [values], (err, result) => {
    if (err) callback(err)
    else callback(null, result)
  })
  connection.end()
}
```

```
const updateUser = (id, user, callback) => {
  connection.connect()
  var query = 'UPDATE utilisateur SET nom = ' + user.nom + ', prenom = ' + user.prenom + ', login = ' + user.login + ', mdp = ' + user.mdp + ' WHERE id = ' + id
  connection.query(query, [values], (err, result) => {
    if (err) callback(err)
    else callback(null, result)
  })
  connection.end()
}
```

```
const addUser = (request, response) => {
  var body = request.body
  userModel.addUser(body, (err, result) => {
    if (err) response.json(err)
    else response.json({result})
  })
}
```

```
unknown, 2 weeks ago • first commit
/* request.params.id RECUPERE ID SUR POSTMAN */
const updateUser = (request, response) => {
  var body = request.body
  userModel.updateUser(request.params.id, body, (err, result) => {
    if (err) response.json(err)
    else response.json({result})
  })
}
```

```
const deleteUser = (request, response) => {
  userModel.deleteUser(request.params.id, (err, result) => {
    if (err) response.json(err)
    else response.json({result})
  })
}
```

Comme précédemment, tout se passe dans le **model** et le **Controller**.

Dans **model/users.model** on définit les variables contenant les requêtes utilisées pour mettre à jour la base de données pour les modifications/suppressions/mises à jour de données d'utilisateur

Dans **controller.users.controller**, on définit l'utilisation de la variable couplée avec une action créer dans le model ce qui va permettre de **modifier, supprimer** ou modifier les différents utilisateurs Si l'action n'est pas menée à sa réalisation, on obtiendra une erreur

## Routing

Le routage permet la redirection de pages en pages au sein d'une application Web.

```
router.put('/:id', userController.update)

router.delete('/delete/:id', userController.delete)

router.get('/', userController.searchAll)

module.exports = router
```

```
unknown, 2 weeks ago | 1 author (unknown)
const express = require('express')
const authenticationController = require('./controllers/authenticationController')
let router = express()

router.post('/', authenticationController.login)

module.exports = router
```

```
// CREATION
router.post('/', ficheController.addFiche)
router.post('/lignefraisforfait/new', ficheController.addFraisForfait)
router.post('/lignefraishorsforfait/new', ficheController.addFraisHorsForfait)

// MODIFICATION
router.put('/:id:mois', ficheController.updateFiche)

// SUPPRESSION
router.delete('/delete/:id:mois', ficheController.deleteFiche)

// Modifier BDD
//router.put('/:id:mois', ficheController.updateFiche)

router.put('/lignefraisforfait/:id:mois', ficheController.updateFraisForfait)
router.put('/lignefraishorsforfait/:id:mois', ficheController.updateFraisHorsForfait)
//Supprimer données
router.delete('/lignefraisforfait/delete/:id:mois', ficheController.deleteFraisForfait)
router.delete('/lignefraishorsforfait/delete/:id:mois', ficheController.deleteFraisHorsForfait)

You, seconds ago • Uncommitted changes

// EXPORT DU ROUTEUR POUR POUVOIR L'APPELER
module.exports = router
```

Le routage peut aussi permettre certaines actions comme dans l'utilisation de fonctions pour répondre aux requêtes avec un certain chemin en retour

Le composant **protected route** sera chargé de vérifier si nous sommes connectés avant de donner accès au site web, sinon il ne redirigera pas les utilisateurs sur une autre page

```
import Bills from './views/bills';
import reportWebVitals from './reportWebVitals';
import CreateBill from './views/CreateBill';
import {
  BrowserRouter as Router,
  Route,
  Switch
} from 'react-router-dom'

import Login from './views/Login'
import ProtectedRoute from './components/protectedRoute'

const Root = () => {
  return (
    <Router>
      <Switch>
        <Route exact path="/" component={Login}/>
        <ProtectedRoute exact path="/bills" component={Bills}/>
        <ProtectedRoute exact path="/create" component={CreateBill}/>
      </Switch>
    </Router>
  )
}

ReactDOM.render(
  <React.StrictMode>
    <Root />
  </React.StrictMode>,
  document.getElementById('root')
```