



Contexte GSB

Application mobile de gestion des visites et des médecins

Partie 1 :Présentation et mise en place de la page de connexion

Présentation du projet

Présentation du contexte

L'activité à gérer

L'activité commerciale d'un laboratoire pharmaceutique est principalement réalisée par les visiteurs médicaux. En effet, un médicament remboursé par la sécurité sociale n'est jamais vendu directement au consommateur mais prescrit au patient par son médecin.

Toute communication publicitaire sur les médicaments remboursés est d'ailleurs interdite par la loi. Il est donc important, pour l'industrie pharmaceutique, de promouvoir ses produits directement auprès des praticiens.

Les Visiteurs Médicaux

L'activité des visiteurs médicaux consiste à visiter régulièrement les médecins généralistes, spécialistes, les services hospitaliers ainsi que les infirmiers et pharmaciens pour les tenir au courant de l'intérêt de leurs produits et des nouveautés du laboratoire.

Chaque visiteur dispose d'un portefeuille de praticiens, de sorte que le même médecin ne reçoit jamais deux visites différentes du même laboratoire.

Comme tous les commerciaux, ils travaillent par objectifs définis par la hiérarchie et reçoivent en conséquence diverses primes et avantages.

Pour affiner la définition des objectifs et l'attribution des budgets, il sera nécessaire d'informatiser les comptes rendus de visite.

L'activité des visiteurs

L'activité est composée principalement de **visites** : réalisées auprès d'un praticien (médecin dans son cabinet, à l'hôpital, pharmacien, chef de clinique...), on souhaite en connaître la date, le motif (6 motifs sont fixés au préalable), et savoir, pour chaque visite, les médicaments présentés et le nombre d'échantillons offerts. Le bilan fourni par le visiteur (le médecin a paru convaincu ou pas, une autre visite a été planifiée...) devra aussi être enregistré.

Les produits

Les produits distribués par le laboratoire sont des médicaments : ils sont identifiés par un numéro de produit (dépôt légal) qui correspond à un nom commercial (ce nom étant utilisé par les visiteurs et les médecins).

Comme tout médicament, un produit a des effets thérapeutiques et des contre-indications.

On connaît sa composition (liste des composants et quantité) et les interactions qu'il peut avoir avec d'autres médicaments (éléments nécessaires à la présentation aux médecins).

La posologie (quantité périodique par type d'individu : adulte, jeune adulte, enfant, jeune enfant ou nourrisson) dépend de la présentation et du dosage.

Un produit relève d'une famille (antihistaminique, antidépresseur, antibiotique, ...).

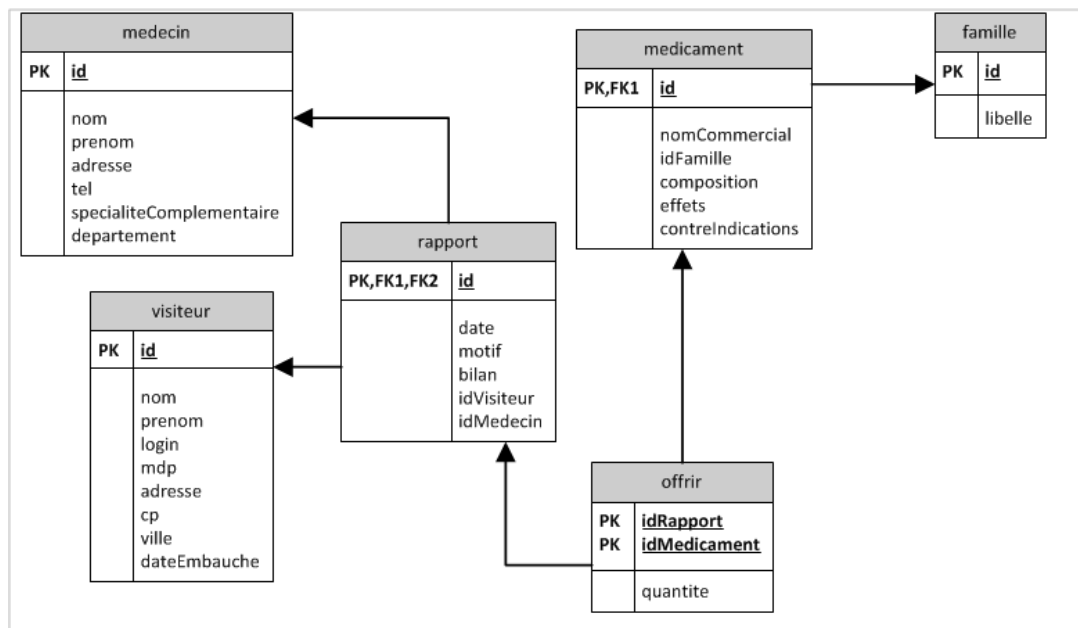
Lors d'une visite auprès d'un médecin, un visiteur présente un ou plusieurs produits pour lesquels il pourra laisser des échantillons.

Les médecins

Les médecins sont le cœur de cible des laboratoires. Aussi font-ils l'objet d'une attention toute particulière.

Pour tenir à jour leurs informations, les laboratoires achètent des fichiers à des organismes spécialisés qui donnent, les diverses informations d'état civil et la spécialité complémentaire.

Modélisation complète des données



L'application à réaliser

L'entreprise envisage de permettre aux visiteurs de gérer ses visites par l'intermédiaire de son smartphone ou de sa tablette. L'application devrait permettre de réaliser les cas d'utilisation suivants :

Description textuelle des cas d'utilisation

Cas : gérer les rapports de visite

Scénario classique

- 1) Le visiteur demande à créer un nouveau rapport de visite
- 2) Le système retourne un formulaire avec la liste des médecins et des champs de saisie
- 3) Le visiteur sélectionne un médecin à partir de son début de nom, sélectionne la date et remplit les différents champs, sélectionne les médicaments et les quantités offertes et valide
- 4) Le système enregistre le rapport

Scénario étendu : modification d'un rapport

- 5) Le visiteur demande à modifier un rapport
- 6) Le système retourne un formulaire avec une date à sélectionner
- 7) Le visiteur sélectionne la date
- 8) Le système retourne les rapports *que le visiteur a effectués à cette date*
- 9) Le visiteur sélectionne un rapport de visite
- 10) Le système retourne les informations déjà saisies *concernant le motif et le bilan*
- 11) Le visiteur modifie les informations
- 12) Le système enregistre les modifications

Scénario alternatif

- 4.1) Des champs ne sont pas remplis, le système en informe le visiteur, retour à 3

Cas : gérer les médecins

Scénario classique

- 1) Le visiteur demande à voir les informations concernant un médecin
- 2) Le système retourne un formulaire avec un champ de recherche du médecin
- 3) Le visiteur sélectionne un médecin à partir de son début de nom et valide
- 4) Le système retourne les informations personnelles concernant ce médecin

Scénario étendu :

- 5) Le visiteur clique sur le numéro de téléphone du médecin
- 6) Le système compose le numéro
- 7) Le visiteur demande à voir tous les anciens rapports de visite concernant ce médecin
- 8) Le système retourne tous ses rapports
- 9) Le visiteur demande à modifier certains champs concernant des informations du médecin
- 10) Le système enregistre ces modifications.

Mise en place du projet

Création du projet

Commencez par créer un projet React nommé GSB, avec les plugins suivants à installer :

- **react-router-dom** : pour mettre en place le routage : <https://reactrouter.com/en/main/start/tutorial>
- **tailwind** : pour la mise en forme avec la bibliothèque TailwindCSS (attention à bien suivre les instructions) : <https://tailwindcss.com/docs/guides/vite#react>
- **axios** : pour pouvoir communiquer avec des API : <https://axios-http.com/fr/docs/intro>

Remarque : à cette étape, vous pouvez d'ores et déjà supprimer les fichiers App.jsx et App.css, et ne laissez que les éléments nécessaires à Tailwind dans le fichier index.css.

Dans un premier temps créez deux pages : index et accueil. N'oubliez pas de définir leur routes.

Page d'index : Authentification

Formulaire d'authentification :

Avec ce que vous avez déjà vu dans les premiers TP, et en vous inspirants de l'API de Tailwind, essayez de créer la page d'index, qui devra servir de formulaire d'authentification du visiteur :

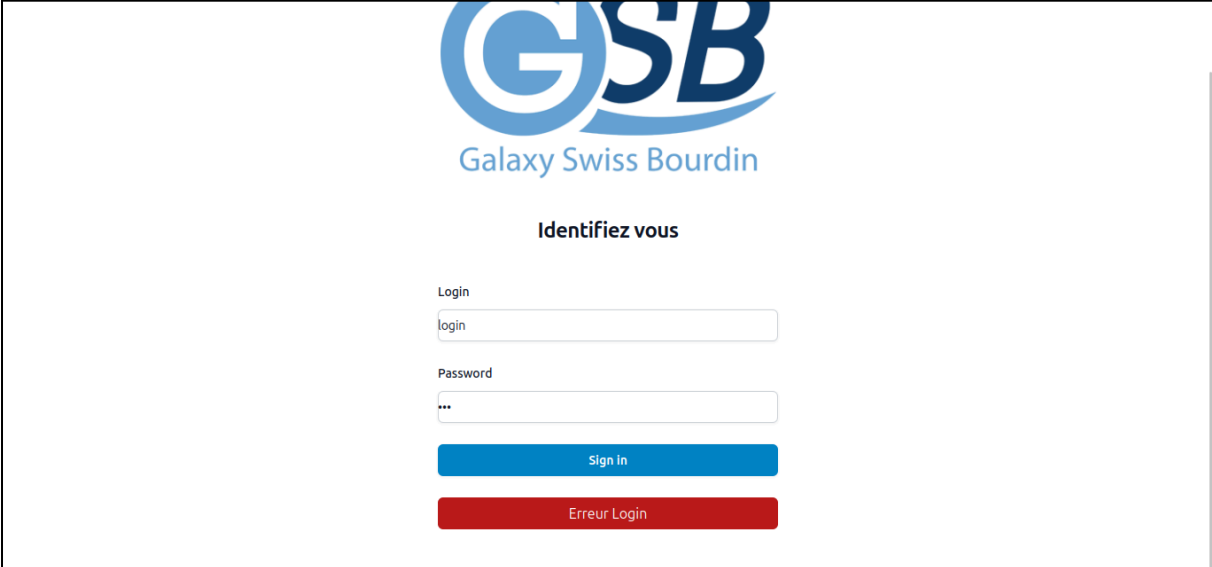


Le formulaire d'authentification est centré sur la page. Il commence par le logo GSB (Galaxy Swiss Bourdin) en bleu. En dessous, le texte 'Identifiez vous' est centré. Il y a deux champs de saisie : 'Login' et 'Password'. Le champ 'Login' est précédé du texte 'Login' et le champ 'Password' est précédé du texte 'Password'. Un bouton 'Sign In' est situé en bas.

Exemple d'une page d'authentification générée à l'aide de Tailwind

Remarque : vous n'êtes pas obligé de respecter cette mise en page, aucune charte graphique n'est imposée.

Vous devez aussi mettre en place une bannière “Erreur d’authentification”, afin d’afficher un message d’erreur en cas de mauvaise saisie. Elle devra être cachée par défaut, et s’afficher qu' en cas de nécessité.



Occupons nous de la partie script. Il y a besoin d’au moins deux variables :

- navigate : Pour la redirection,
- erreurLogin : état qui sera mis à jour, afin d’afficher la bannière d’erreur

Et enfin une fonction :

- connection(event) : fonction qui va s’exécuter lorsque le bouton “Connexion” sera cliqué, et qui va nous permettre de tester le login et le mot de passe.

Travail à faire

Écrire le code du composant *Index*. Testez en lançant votre projet, avec un test pour les identifiants suivants :

- login : aribiA
- mdp : aaaa

En cas de succès, afficher un message “Succès” dans la console, et en cas d’échec affichez la bannière d’erreur.

Connexion à l'API - Axios

Commencer par lire l'annexe 1 qui propose un rappel sur les services REST ainsi que la **présentation du service GSB**.

Utilisation d'un service pour consommer le service REST

Pour « consommer » le service REST fourni dans notre cas par le serveur Apache avec le port 80 par défaut, nous allons créer une classe dédiée, **api** :

```
import axios from 'axios';

export default axios.create({
  // Remplacer localhost par IP de votre serveur si besoin
  baseURL: 'http://172.16.61.61/restGSB'
});
```

Travail à faire

Dans le répertoire "src/", créer un sous-dossier nommé "api", et créer le fichier api.js, avec le code ci-dessus.

Authentification du visiteur - Programmation asynchrone

Nous allons voir dans un premier exemple le cas d'une authentification du visiteur. En effet, notre API nous permet de récupérer les données d'un visiteur, via une méthode `getLeVisiteur()`, avec en paramètre le login et le mot de passe :

```
public function getLeVisiteur($login, $mdp){
    $req = "select id, nom, prenom from visiteur where login = :login and mdp = :mdp";
    $stm = $this->connexion->prepare($req);
    $stm->bindParam(':login', $login);
    $stm->bindParam(':mdp', $mdp);
    $stm->execute();
    $laLigne = $stm->fetch();
    if(count($laLigne)>1)
        return $laLigne;
    else
        return NULL;
}
```

Code de l'API juste à titre informatif

Grâce à Axios, nous allons créer une fonction qui permet de nous connecter dans la base de données, et de vérifier ses données d'authentification.

De nombreuses fonctions sont fournies par les environnements hôtes JavaScript qui vous permettent de planifier des actions *asynchrones*. En d'autres termes, des actions que nous lançons maintenant, mais qui se terminent plus tard.

Autrement dit, ce style de programmation asynchrone est basé sur les "callbacks". Une fonction qui fait quelque chose de manière asynchrone doit fournir un argument callback où nous mettons la fonction à exécuter après qu'elle soit terminée.

Voici un exemple afin de mieux comprendre la programmation asynchrone :

Imaginez que vous êtes un grand chanteur et que les fans vous demandent jour et nuit votre prochaine chanson.

Pour avoir un peu de paix, vous promettez de leur envoyer dès que celle-ci est publiée. Vous donnez à vos fans une liste d'abonnement. Ils peuvent y ajouter leur adresse mail, comme cela, quand le single est sorti, tous les emails reçoivent votre single. Et même si quelque chose arrive, comme un feu dans le studio, et que vous ne pouvez pas sortir le single, ils en seront aussi notifiés.

Tout le monde est content : vous, puisque l'on vous laisse plus tranquille, et vos fans parce qu'ils savent qu'ils ne rateront pas la chanson.

C'est une analogie réelle à un problème courant de programmation :

Un "producteur de code" qui réalise quelque chose mais nécessite du temps. Par exemple :

- *un code qui charge des données à travers un réseau : C'est le "chanteur".*
- *Un "consommateur de code" qui attend un résultat du "producteur de code" quand il est prêt.*
- *Beaucoup de fonctions peuvent avoir besoin de ce résultat. Ces fonctions sont les "fans".*

Une promesse (promise) est un objet spécial en Javascript qui lie le "producteur de code" et le "consommateur de code" ensemble. En comparant à notre analogie c'est la "liste d'abonnement". Le "producteur de code" prend le temps nécessaire pour produire le résultat promis, et la "promesse" donne le résultat disponible pour le code abonné quand c'est prêt.

L'analogie n'est pas la plus correcte, car les promesses en Javascript sont un peu plus complexes qu'une simple liste d'abonnement : elles ont d'autres possibilités mais aussi certaines limitations. Toutefois c'est suffisant pour débiter.

Avant de commencer à coder, un petit rappel de notre appel à l'API :



```
{ "id": "a131", "0": "a131", "nom": "Aribi", "1": "Aribi", "prenom": "Alain", "2": "Alain"
```

Ici, nous faisons appel à la fonction “connexion”, en utilisant la méthode GET , avec deux paramètres :

- login = aribiA
- mdp = aaaa

L'API nous retourne donc l'utilisateur Alain Aribi avec ses informations sous format JSON.

Axios fonctionne de cette manière : il va donc appeler des fonctions de notre API avec les paramètres et recevoir un retour pour voir s'il a trouvé des données ou non.

Écrivons désormais le code pour tester notre login et mot de passe, via la fonction getUser() :

```
/**
 * Appel de notre fonction de manière asynchrone ("async"),
 * Celle-ci va nous retourner une promesse (Promise),
 * ou déclencher une erreur
 */
async function getVisiteur(leLogin, leMdp) {
  /**
   * Try : On tente de se connecter à l'API, en appelant la fonction "connexion"
   * avec la méthode GET, qui prend comme paramètre "login" et "mdp"
   *
   * Catch : Si on arrive pas à atteindre notre API, on affiche un message d'erreur dans la console
   */
  try {
    /**
     * On attend que la promesse se réalise, et renvoie son résultat
     * dans la variable response ("await")
     */
    const response = await api.get('/connexion', {
      params: {
        login: leLogin,
        mdp: leMdp
      },
    });
    return response;
  } catch (error) {
    console.log("ERREUR connexion API")
  }
}
```

Cette fonction est exécutée de manière asynchrone, car il commence à se charger maintenant, mais s'exécute plus tard, lorsque la fonction est déjà terminée.

Le mot “async” devant une fonction signifie une chose simple : une fonction renvoie toujours une promesse (type de données Promise). Les autres valeurs sont enveloppées dans une promesse résolue automatiquement.

Travail à faire

Compléter votre fichier page index avec le code ci-dessus.

Et enfin nous allons compléter la méthode appelée lors de la soumission du formulaire, qui va appeler la méthode `getVisiteur`, avec les données saisies par l'utilisateur.

Un objet promesse permet le lien entre l'exécuteur (le “code produit” ou “chanteur”) et les fonctions consommatrices (les “fans”), lesquels recevront un résultat ou une erreur. Ces fonctions consommatrices peuvent s'abonner (subscribed) en utilisant les méthodes `.then`, `.catch`.

Le premier argument `.then` est une fonction qui se lance si la promesse est tenue, et reçoit le résultat en paramètre.

Le deuxième argument `.catch` (facultatif) est une fonction qui se lance si la promesse est rompue, et reçoit l'erreur en paramètre.

Donc dans notre cas, cela donne :

```
/**
 * Ici nous allons appeler la fonction getUser,
 * et voir si nous allons récupérer notre "promesse"
 */
getVisiteur(form.get("login"), form.get("password"))
  .then((response) => {
    /**
     * Si tout est OK, alors on redirige vers la page d'accueil,
     * en transmettant les informations de l'utilisateur connecté.
     * Sinon, on va afficher la bannière "Erreur Login"
     */
    if (response.data != null) {
      console.log(response.data);
      /**
       * Si il y a un erreur de saisie, on affiche la bannière d'erreur
       */
    } else {
      setErreurLogin(true)
    }
  });
```

Travail à faire

1) Compléter votre page Index, et vérifiez que la fonction se lance bien au clic de la souris, pour cela ouvrez votre console web et vérifiez si :

- Il y a bien un log d'affiché sur la console, avec les informations de l'utilisateur si tout est correct,
- Une erreur, avec la bannière qui s'affiche

```
▼ Object 3 LoginView.vue:40  
  0: "a131"  
  1: "Aribi"  
  2: "Alain"  
  3: "8 rue des Charmes"  
  4: "46000"  
  5: "Cahors"  
  adresse: "8 rue des Charmes"  
  cp: "46000"  
  id: "a131"  
  nom: "Aribi"  
  prenom: "Alain"  
  ville: "Cahors"  
  ► [[Prototype]]: Object
```

Cas de succès de connexion avec les bons identifiants

```
Erreur VM1660 LoginView.vue:34
```

Cas d'échec avec les mauvais identifiants

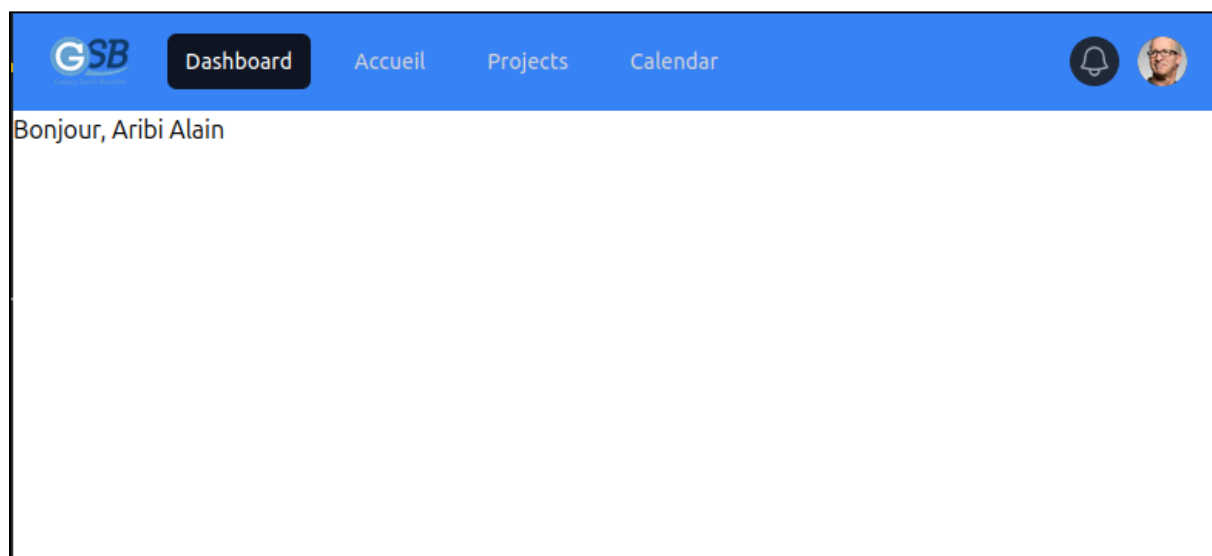
Redirection vers une page d'accueil du visiteur

Une fois notre API fonctionnelle, donc maintenant nous allons mettre en place une redirection vers une page d'accueil pour le visiteur :

Travail à faire

1) Réadapter votre méthode getLogin() dans la page de connexion, de sorte que :

- Lorsqu'il y a un succès de connexion, rediriger vers la page d'Accueil,
- Dans la page d'accueil, afficher le contenu de la variable utilisateur, contenu dans le store
- Avoir la bannière d'erreur en cas d'echec



Exemple de la page accueil après succès de login

ANNEXES

Annexe 1 : API REST

Une API, qu'est-ce que c'est ?

Une **API (application programming interface ou « interface de programmation d'application »)** est un ensemble de définitions et de protocoles qui facilite la création et l'intégration de logiciels d'applications. Elle est parfois considérée comme un contrat entre un fournisseur d'informations et un utilisateur d'informations, qui permet de définir le contenu demandé au consommateur (l'appel) et le contenu demandé au producteur (la réponse). Par exemple, l'API conçue pour un service de météo peut demander à l'utilisateur de fournir un code postal et au producteur de renvoyer une réponse en deux parties : la première concernant la température maximale et la seconde la température minimale.

En d'autres termes, lorsque vous souhaitez interagir avec un ordinateur ou un système pour récupérer des informations ou exécuter une fonction, une API permet d'indiquer au système ce que vous attendez de lui, afin qu'il puisse comprendre votre demande et y répondre.

Vous pouvez vous représenter une API comme un médiateur entre les utilisateurs ou clients et les ressources ou services web auxquels ils souhaitent accéder. Pour une entreprise, c'est aussi une solution pour partager des ressources et des informations, tout en maintenant un certain niveau de sécurité, de contrôle et d'authentification, en déterminant qui est autorisé à accéder à quoi.

Autre avantage des API : vous n'avez pas besoin de connaître le fonctionnement exact de la mise en cache, c'est-à-dire de savoir comment vos ressources sont récupérées ni d'où elles proviennent.

REST, qu'est-ce que c'est ?

REST (*representational state transfer*) est un style d'architecture logicielle définissant un ensemble de contraintes à utiliser pour créer des services web. Les services web conformes au style d'architecture REST, aussi appelés services web **RESTful**, établissent une interopérabilité entre les ordinateurs sur Internet. Les services web REST permettent aux systèmes effectuant des requêtes de manipuler des ressources web via leurs représentations textuelles à travers un ensemble d'opérations uniformes et prédéfinies sans état. L'architecture REST a été créée par l'informaticien Roy Fielding.

Une API REST (également appelée API RESTful) est donc une interface de programmation d'application (API ou API web) qui respecte les contraintes du style d'architecture REST et permet d'interagir avec les services web RESTful..

Nous allons illustrer cette architecture à l'aide d'exemples.

1.a Le service Open Data de la ville de Paris : <https://opendata.paris.fr/pages/home/>

La ville de Paris propose une API nommée OpenData, qui retourne les informations sur les différents services de la ville de Paris, sous format JSON :

The screenshot shows the Paris Data OpenData portal. It features a navigation bar with links like Accueil, Les données, Paris en chiffres, L'API, La licence, La démarche, Cartographie, and Les algorithmes. The main content area displays a grid of dataset cards. Each card includes a title, a brief description, the modification date, the producer, the license, and the number of records. For example, the 'Vélib - Vélos et bornes - Disponibilité temps réel' card shows it was modified on 21 October 2022, produced by AutoLib Vélib Métropole, under an Open Database License (ODBL), with 1,442 records. Other cards include 'Subventions aux associations accordées ou non', 'Ascenseurs / Escalators - Télé-surveillance temps réel', 'Vélib' - Localisation et caractéristique des stations', 'Belib' - Points de recharge pour véhicules électriques', 'Arbres', 'Travaux perturbants la circulation', and 'Fontaines à boire'.

Exemple avec une station Vélib :

```

4:
datasetid: "velib-disponibilite-en-temps-reel"
recordid: "b3ff0ce89086a0598de2377dd852aed45fb589a3"
fields:
  name: "Lacépède - Monge"
  stationcode: "5110"
  ebike: 3
  mechanical: 0
  coordonnees_geo:
    0: 48.84389286531899
    1: 2.3519663885235786
  duedate: "2022-10-21T08:50:02+00:00"
  numbikesavailable: 3
  numdocksavailable: 17
  capacity: 23
  is_renting: "OUI"
  is_installed: "OUI"
  nom_arrondissement_communes: "Paris"
  is_returning: "OUI"
  geometry:
    type: "Point"
    coordinates:
      0: 2.3519663885235786
      1: 48.84389286531899
  record_timestamp: "2022-10-21T09:09:00.456Z"

```

On peut connaître ainsi, le nombre de places totales (23), le nombre de vélos disponibles (3) entre autres.

Toutes les applications Vélib (SmartPhone souvent) utilisent ce service REST. Il en est de même pour de nombreuses autres applications (météorologique, géographique, Amazon,...).

2.a. Notre service REST GSB

Dans notre API REST, nous allons utiliser la norme **C.R.U.D** pour réaliser nos opérations d'interactions avec la base de données, à savoir :

Create

- **POST** (ou **PUT**) : Création d'une ressource.

Read

- **GET** : Récupération d'une ressource ou d'une collection.

Update

- **PUT** (ou **PATCH**) : Remplacement d'une ressource ou d'une collection.

Delete

- **DELETE** : Suppression d'une ressource ou d'une collection.

Nous pouvons maintenant utiliser ce service à partir d'URL ; voici quelques exemples :

- Les informations sur le médecin d'id 45.



```
{ "id": "45", "0": "45", "nom": "CORTES", "1": "CORTES", "prenom": "Gilles", "2": "Gilles", "adresse": "65 rue des oiseaux ARROUT 09800", "3": "65 rue des oiseaux ARROUT 09800", "tel": "0578097401", "4": "0578097401", "specialitecomplementaire": "MEDECINE APPLIQUEE AUX SPORTS", "5": "MEDECINE APPLIQUEE AUX SPORTS", "departement": "9", "6": "9" }
```

- Les informations sur le rapport d'id 200.



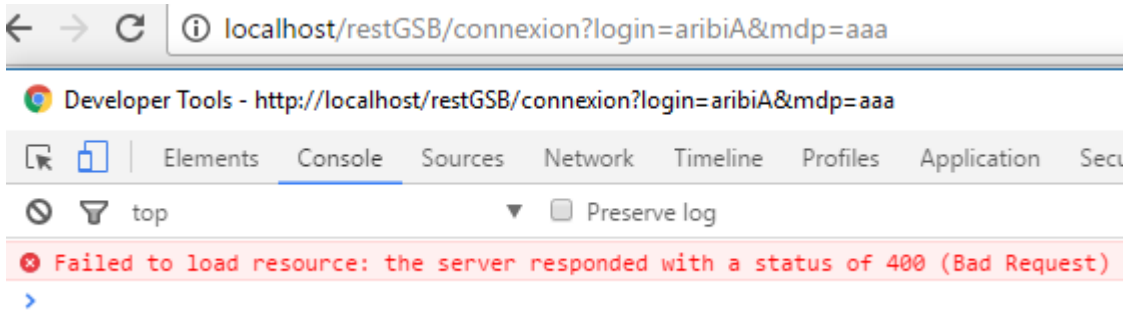
```
{ "id": "200", "0": "200", "date": "2016-07-11", "1": "2016-07-11", "motif": "Prise de contact", "2": "Prise de contact", "bilan": "Peu int\u00e9ressant", "3": "Peu int\u00e9ressant", "idVisiteur": "e52", "4": "e52", "idMedecin": "625", "5": "625" }
```

- Les informations d'un visiteur dont on fournit le login et le mot de passe.



```
{"id":"a131","0":"a131","nom":"Aribi","1":"Aribi","prenom":"Alain","2":"Alain"
00 000 000 000000 0 0000 00 0000 00 0000
```

- Un retour de la classe 400 si les informations de connexion ne sont pas valides.



Remarque : le format retourné par ce service est le JSON.