

Galaxy Swiss Bourdin – Gestion de Visites

Documentation Technique

Cette documentation vise à présenter les axes majeurs de la méthode de conception de GSB Visites, site web donnant accès à une gestion complète de rapports et d'un portefeuille de médecins à ses utilisateurs.

Sommaire:

Présentation du projet GSB – Gestion de visites.....	3
Contexte.....	3
Cahier des charges et attentes	3
Schémas et architecture	4
Diagramme des cas d'utilisation	4
MLD et UML	4
Technologies et méthodes utilisées.....	5
Réalisation du projet.....	8
Connexion	8
Gestion des médecins	10
Recherche de médecin :.....	10
Modifier le médecin :.....	11
Lister les rapports du médecin :.....	12
Gestion des rapports.....	12
Recherche de rapport:	12
Ajouter un rapport:.....	14
Modifier un rapport:	15

Présentation du projet GSB – Gestion de visites

Contexte

L'application web GSB Gestion de visites doit pouvoir permettre aux visiteurs médicaux, représentant l'activité commerciale d'un laboratoire pharmaceutique, de rendre visite à des médecins généralistes de manière régulière. Afin de mener leur mission à bien, ces visiteurs posséderont chacun un portefeuille de médecins unique, contenant une liste prédéterminée de médecins qui leurs sont attribués.

Le visiteur doit, après une connexion via identifiant et mot de passe, avoir accès à de multiples fonctionnalités, à savoir la gestion de ses visites, et la gestion de son portefeuille de médecins.

Cahier des charges et attentes

La partie de gestion des médecins doit permettre au visiteur d'interagir avec son portefeuille.

Le visiteur trouve un/des médecins en saisissant le début de leur nom dans une barre de recherche. Les résultats apparaissent sous forme d'un tableau, listant l'ensemble des médecins correspondant, ainsi que leurs informations (Nom complet, numéro de téléphone, spécialisation...).

En cliquant sur le numéro de téléphone d'un de ses médecins, le système devra composer le numéro automatiquement afin que le visiteur puisse entrer en contact avec le médecin ciblé aisément.

Le visiteur pourra consulter l'ensemble des rapports effectués auprès du médecin, mais également pouvoir modifier les informations le concernant.

La partie de gestion des rapports doit permettre au visiteur de tenir compte de ses visites, en précisant le médecin visité, la date à laquelle la visite a eu lieu, les médicaments prescrits et leur quantité respective, ainsi qu'un motif et un bilan de visite.

Lors de la création d'un rapport, le visiteur devra renseigner l'ensemble des paramètres mentionnés ci-dessus, cependant, des attentes supplémentaires s'ajouteront en ce qui concerne l'ajout de médicaments offerts et la sélection du médecin. En effet, le visiteur doit pouvoir retrouver le médecin visité de manière dynamique, en tapant uniquement le début de son nom. Le visiteur cliquera ainsi sur la proposition correspondante, qui remplira le champ par le nom du médecin souhaité.

La section des médicaments à offrir doit adopter une structure différente des autres champs. Elle fonctionne par paires, et est donc composé d'une liste déroulante, regroupant l'ensemble des médicaments, ainsi qu'un champ permettant au visiteur de préciser la quantité d'échantillons offerts pour chacun des médicaments. Par défaut, aucun champ d'offre ne doit apparaître. Le visiteur peut rajouter des champs en cliquant sur un bouton. Chaque champ sera accompagné d'un bouton permettant son autosuppression. Une limite de cinq champs est prédéfinie. A noter qu'un même médicament ne peut être offert plusieurs fois, et qu'un rapport ne contient pas forcément une offre de médicament, cependant chaque offre spécifiée devra se voir attribuer une quantité.

Le visiteur pourra également modifier le motif et le bilan d'une visite déjà créée.

Par défaut, un tableau contiendra l'ensemble des visites effectuées par le visiteur. Cependant, ces dernières pouvant s'accumuler au fil du temps, il sera possible de limiter le champ de recherche via une barre de navigation. Le visiteur entrera alors une date précise, et le tableau ne contiendra alors que les rapports ayant eu lieu à cette date. Si aucun rapport n'a été effectué à la date choisie, un message explicite devra le signaler.

Le visiteur pourra à tout moment se déconnecter de sa session en cliquant sur un bouton positionné sur la barre de navigation.

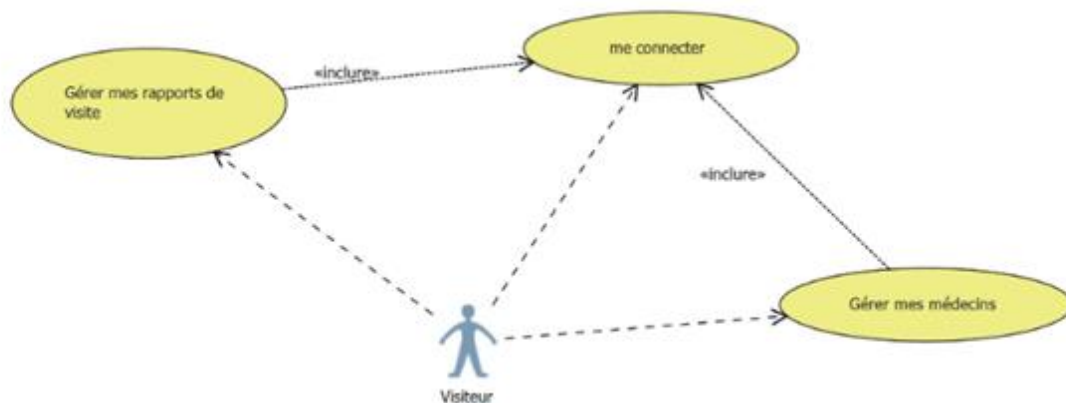
Les opérations d'ajouts, de modifications, ou d'authentications pouvant présenter des erreurs, le système devra tenir compte de la réussite ou non des opérations, et en informer clairement le visiteur.

Schémas et architecture

Afin de mener à bien ce projet, une architecture précise a été définie au préalable. Ce modèle est à respecter scrupuleusement.

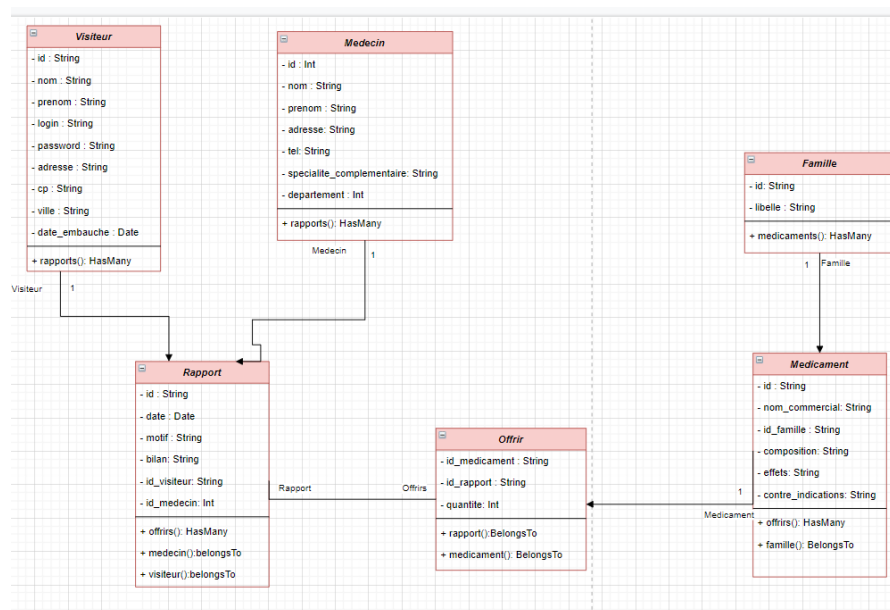
Diagramme des cas d'utilisation

Comme expliqué précédemment, le visiteur devra, après une connexion via identifiant et mot de passe, pouvoir gérer ses rapports de visite, ainsi que son portefeuille de médecin.



MLD et UML

Une base de données étant nécessaire afin de mettre le projet sur pied, un modèle logique de données permet de définir et clarifier les relations entre les tables composant cette base de données.



Technologies et méthodes utilisées

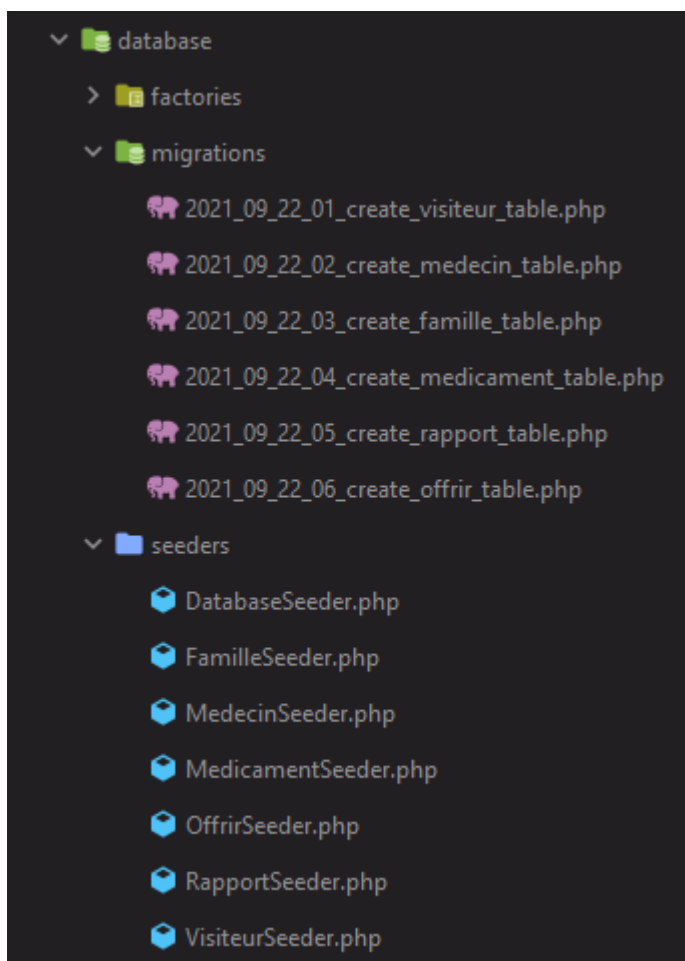
BTS SIO, option SLAM | 2^{ème} année
2021-2022

L'application web, devant suivre une architecture Modèle-Vue-Contrôleur, ainsi qu'une méthode orientée objet, j'ai choisi d'utiliser le Framework Laravel. Laravel est un Framework adoptant le langage PHP, et permettant la mise en place d'une application web plus aisément, proprement et rapidement qu'en utilisant un langage de programmation nativement. Ce Framework a pour avantages d'offrir une gestion des erreurs plus pratique, un chiffrement des mots de passe utilisateur plus perfectionné et actuel que PHP, ou encore de nombreuses fonctions garantissant une sécurité optimale du site web, et ainsi des données qu'elle contient et que les visiteurs vont manipuler.

Le Système de gestion de bases de données utilisé sera MySQL, ce dernier étant par défaut sélectionné pour Laravel, il est le SGBD le plus performant, et donc le plus adéquat, pour ce Framework.

Laravel sert d'intermédiaire entre le développeur et le SGBD, et de part sa méthode de fonctionnement, ne nécessite du développeur aucune requête purement en SQL, car Laravel utilise le principe de migration. Les tables, leurs contraintes, ainsi que leurs données respectives seront précréées au sein même du projet. La structure et les contraintes des tables seront présentes dans des fichiers dits de « migrations », et leurs valeurs dans des « seeders ».

Les contraintes, noms et précisions des tables seront écrites en PHP, bien qu'adoptant une syntaxe rappelant volontairement celle du SQL du fait de ses termes techniques.



La totalité de la structure de la table sera donc contenu dans des fichiers dédiés. La gestion de ces fichiers pourra s'effectuer par invite de commande au travers de commandes spécifiques incluses par Laravel. Une base de données comportant des défauts pourra donc être modifiée bien plus rapidement, et l'instauration de sa structure dans le SGBD(R) se fera rapidement par invite de commande.

Exemple de structure de table : La table « Rapports »

```
public function up()
{
    Schema::create( table: 'rapports', function (Blueprint $table) {
        $table->integer( column: 'id', autoIncrement: 11);
        $table->date( column: 'date')->nullable()->default( value: NULL);
        $table->string( column: 'motif', length: 100)->nullable()->default( value: NULL);
        $table->string( column: 'bilan', length: 100)->nullable()->default( value: NULL);
        $table->char( column: 'visiteur_id', length: 4);
        $table->unsignedInteger( column: 'medecin_id');
        $table->foreign( columns: 'medecin_id')
            ->references( columns: 'id')
            ->on( table: "medecins");
        $table->foreign( columns: 'visiteur_id')
            ->references( columns: "id")
            ->on( table: "visiteurs");
    });
}
```

On remarque que l'ensemble des colonnes sont spécifiés, ainsi que le type des données qu'elles prennent en charge respectivement. Il est possible, après déclaration de ces colonnes, d'y ajouter des contraintes, principalement des clefs étrangères dans ce cas précis.

```
public function run()
{
    Rapport::create( [
        'id'=>1,
        'date'=>'2017-01-02',
        'motif'=>'positif',
        'bilan'=>'visiteannuelle',
        'visiteur_id'=>'b16',
        'medecin_id'=>'963'
    ] );

    Rapport::create( [
        'id'=>2,
        'date'=>'2016-07-02',
        'motif'=>'Demande du médecin',
        'bilan'=>'Très aimable maintenir un contact régulier',
        'visiteur_id'=>'a93',
        'medecin_id'=>'4'
    ] );

    Rapport::create( [
        'id'=>3,
        'date'=>'2016-07-02',
        'motif'=>'Demande du médecin',
        'bilan'=>'Trop pressé',
        'visiteur_id'=>'a93',
    ] );
}
```

Le « seeder » regroupe l'ensemble des données à insérer dans la table. Il prendra la forme d'un tableau PHP basique, mais correspond finalement à une requête INSERT INTO. Laravel opte pour une méthode de stockage des données proche des fichiers .JSON, et écarte les requêtes d'insertion classiques que peut offrir SQL.

Cette méthode a pour avantages de permettre à l'utilisateur une meilleure gestion de la structure de sa base de données, ce dernier pouvant revenir en arrière et récupérer une forme précédente ou ultérieure, tout en y injectant les données instantanément.

Laravel va créer automatiquement les classes métier correspondantes à chaque table. Ces classes métiers ne nécessiteront ni constructeur, ni getter ou setter, Laravel prenant en charge ces fonctionnalités de lui-même. La majorité des opérations s'effectuera donc dans les contrôleurs, ce qui représente un réel gain de temps pour le développeur.

La relation entre les tables et les modèles se fait automatiquement grâce à l'« Active Record Pattern », c'est-à-dire via une similitude dans les noms des documents. Ainsi, la table « rapports »

sera automatiquement lié à un modèle « Rapport ». Cette méthode, bien que pratique, contraint les fichiers à adopter un nom très précis, un modèle nommé « Rapports » ne sera, par exemple, pas relié à la table rapports.

L'ensemble des fichiers de vues sont appelés « Blade », ils agissent comme des fichiers PHP plus performants, incluant des fonctionnalités ajoutées par Laravel, comme des protections contre les attaques CSRF, ou encore la possibilité d'intégrer des variables PHP à un script JavaScript.

Réalisation du projet

Connexion

La connexion est la première étape d'accès aux ressources que propose GSB Visites. Afin de mener le système de connexion à bien, l'ensemble des redirections et affichages passent par un même fichier de route, appelé « web ». On y définit ensuite le type de requête (POST, GET...), les informations contenues dans l'URL, puis l'action à effectuer.

Exemple des routes pour la connexion :

```
18
19 Route::get( uri: 'login', [SessionController::class, 'create'])->middleware( middleware: 'guest');
20 Route::post( uri: 'login', [SessionController::class, 'store'])->middleware( middleware: 'guest');
21 Route::get( uri: 'logout', [SessionController::class, 'destroy'])->middleware( middleware: 'checkLog');
22
```

On remarque que deux redirections existent pour « login », cependant, une est allouée à la méthode GET, à savoir un accès standard à la page, et l'autre à la méthode POST, qui correspond à l'envoi d'un formulaire de connexion. Il est également possible d'ajouter à tout cela un « middleware ». Le middleware est un intermédiaire qui sert, de manière générale, à vérifier une permission, ou à limiter l'accès à un contenu. Ici, le middleware « guest » empêche un utilisateur connecté de se connecter, et le middleware « checkLog » empêche un utilisateur non-connecté de se déconnecter.

Laravel offre une méthode très singulière en ce qui concerne la gestion des erreurs et la vérification des critères d'un formulaire. Voici un exemple en ce qui concerne la page de connexion :

```
public function store(){
    $attributes = request()->validate([
        'login' => ['required', 'alpha_dash', 'min:2', 'max:20'],
        'pass' => ['required', 'min:2', 'max:20']
    ]);

    /* Échec de la connexion */
    if (!Auth::attempt(['login'=>$attributes['login'], 'password'=>$attributes['pass']])){
        return back()
            ->withInput()
            ->withErrors(['login' => 'L\'identifiant ou le mot de passe spécifié est incorrect']);
    }

    /* Succès de la connexion */
    session()->regenerate(); /* Contre les attaques "Session Fixation" */
    return redirect( to: '/' )->with('event_success', 'Connexion réussie, bienvenue <b>'.Auth::user()['prenom'].' '.Auth::user()['nom'].'</b>.' )->withInput();
}
```

Ici, on constate la mise en pratique de la fonction « request validate » que Laravel offre. Cette fonction permet de définir des critères pour chaque champ du formulaire. Ainsi, on constate que le champ de l'identifiant et du mot de passe sont obligatoires, qu'ils ont chacun un minimum et un maximum de caractères, et que le login ne peut contenir d'espaces. Il est important de restreindre l'utilisateur à la fois directement sur le formulaire, mais aussi dans le contrôleur, car les restrictions basiques qu'offre HTML peuvent être contournés assez simplement.

Une fonction prédéfinie par Laravel « attempt », appartenant à la classe d'Authentification, permet de vérifier si l'identifiant et le mot de passe correspondent à un utilisateur présent dans la base de données. Afin d'utiliser cette méthode, Laravel requiert des mots de passe chiffrés au sein de la base de données, ce qui renforce encore une fois la sécurité et la protection des données personnelles.

La fonction bcrypt() permet de chiffrer les mots de passe rapidement.

Chiffrement :

```
'login'=>'dandre',
'password'=>bcrypt( value: 'oppg5'),
```

Valeur stockée dans la base de données :

om	login	password
	aribiaA	\$2y\$10\$izs1F0QVeCsEgztVHs0Zv.Qe4N7RwhmJhLwf8HQTyiN/6QxC0dkSe
	dandre	\$2y\$10\$ay9XR.aRx1RH4/WkmBoSW.VfqC1RqMKf/j0HDqW6bs2Nbhs3mS5py

En cas d'échec lors de la tentative de connexion, la page ramène l'utilisateur en arrière. La fonction withErrors() permet d'indiquer dans la vue les erreurs responsables de l'échec d'authentification. La fonction withInputs() permet de garder en mémoire les valeurs entrées précédemment par l'utilisateur. Ainsi, en cas d'échec, l'identifiant sera toujours présent, et ne devra pas être écrit à nouveau.

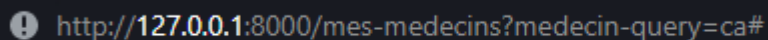
Les messages d'erreurs par défaut étant en Anglais, un fichier permet de les modifier, ce qui est nécessaire par soucis de compréhension. On attribue à chaque restriction un message d'erreur qui viendra remplacer celui qui était par défaut.

```
'custom' => [
  'login' => [
    'required' => 'L\'identifiant est obligatoire.',
    'max' => 'L\'identifiant spécifié est trop long.',
    'min' => 'L\'identifiant spécifié est trop court.',
    'alpha_dash' => 'L\'identifiant ne peut contenir que des caractères alpha-numériques et tirets.',
  ],
  'pass' => [
    'required' => 'Le mot de passe est obligatoire.',
    'max' => 'Le mot de passe spécifié est trop long.',
    'min' => 'Le de passe spécifié est trop court.',
  ],
],
```

Gestion des médecins

Recherche de médecin :

Une fois arrivé sur la page correspondante, le visiteur doit pouvoir retrouver une liste de médecins par une chaîne de caractère que ce dernier peut entrer dans une barre de recherche dédiée. Une fois une valeur entrée, une méthode GET permettra de renvoyer cette valeur, qui sera comprise dans l'URL. Exemple pour « ca » :



On constate que la partie « medecin-query » possède à présent pour valeur « ca ». Le contrôleur récupère cette valeur, et répertorie l'ensemble des médecins ayant un nom commençant par la valeur. Si la valeur retournée n'est pas nulle, il y a donc au moins un médecin, et on affiche un tableau comportant les données liées au médecin, sinon, on affiche un message indiquant qu'aucun résultat ne correspond.

```
if ($query == ""){
  return back()
  ->withErrors(['empty' => 'Veuillez remplir le champ avant l\'envoi du formulaire.']);
} else {
  return view( view: 'medecins')->with('msq', $medecins);
}
```

Si le champ est vide, on indique à l'utilisateur qu'il doit absolument remplir le champ. Sinon, on passe une variable contenant l'ensemble des médecins à la vue. Une boucle au sein de la vue permettra de lister les médecins et de les placer dans un tableau HTML.

Modifier le médecin :

Il sera possible, après cette recherche, de modifier les détails concernant les médecins apparaissant dans le tableau. L'identifiant du médecin sera présent dans l'URL, afin qu'il puisse être récupéré.

Exemple pour le médecin Anne-Ange CANO, médecin n°146

L'ensemble des champs du formulaire seront préremplis par les données déjà existantes concernant le médecin.

```
<label for="adress" class="block mb-2 uppercase font-bold text-xs text-gray-700">
  Adresse
</label>
<input type="text" name="adress" class="border border-gray-400 p-2 w-full" value="{{ $medecin->adresse }}">
```

On constate que la valeur allouée au champ provient d'une variable passée à la vue. Cette variable contient le médecin correspondant à l'id présent dans l'URL, comme mentionné ultérieurement.

Il est possible de directement récupérer une valeur souhaitée en indiquant la variable, suivit d'une flèche « -> » et du nom de la colonne de la table. Cette méthode équivaut à un getter traditionnel, mais est géré par Laravel, ce qui facilite encore une fois le travail du développeur, et simplifie le code. Le champ d'adresse aura donc pour valeur l'adresse actuelle du médecin.

Une fois validé, de la même manière que pour la connexion, l'ensemble des champs du formulaire vont passer par une vérification. Voici un exemple un peu plus poussé de vérification pour le téléphone du médecin :

```
/*
- Le regex vérifie que le numéro commence bien par "0": (0)
- Le regex contraint les valeurs à être comprises entre 0 et 9: [0-9]
- Le regex vérifie qu'au moins 9 valeurs suivront le "0" de départ: {9}
*/
'phone' => ['required', "unique:medecins,tel,$leMedecin->id", 'regex:/^(0)[0-9]{9}/', 'max:10'],
```

Comme expliqué dans le code, la fonction de regex permet de garantir que le numéro de téléphone spécifié possède une forme correcte. Le critère « unique » permet de vérifier que le numéro de téléphone est unique, et que personne d'autre ne l'utilise. Cependant, si on ne modifie pas le médecin, le numéro reste le même, et est donc déjà attribué à lui-même. Il faut donc préciser que le numéro doit être unique, excepté pour le médecin actuel.

Une fois la vérification validée, on remplace les valeurs actuelles par les nouvelles. Une fonction de vérification de différence entre la valeur du champ et celle de la bdd peut être intégrée, afin d'esquiver toute opération ou communication avec la BDD futile.

Ici, on insère (ou écrase) le département et la spécialité que si ces dernières sont réellement renseignées, et ne sont donc pas vides. Le numéro de téléphone est quant à lui systématiquement remplacé, la vérification précédente empêchant tout problèmes liés à son caractère unique.

```
$leMedecin->adresse = $inputs['address'];
if (!is_null($inputs['department'])) {
    $leMedecin->departement = $inputs['department'];
}
$leMedecin->tel = $inputs['phone'];
if (!is_null($inputs['spe'])) {
    $leMedecin->specialitecomplementaire = $inputs['spe'];
}
```

Lister les rapports du médecin :

Afin de lister les rapports d'un médecin, une simple requête prenant comme paramètre l'id du médecin placé dans l'URL permet de les récupérer. On classe les rapports du plus récent au plus ancien.

```
$visites = Rapport::where( column: 'medecin_id', $id)
    ->orderBy( column: 'date', direction: 'desc')->get();
```

Il est important de préciser, et cela s'applique également à la partie de modification et de recherche, que seul le visiteur possédant le médecin dans son portefeuille peut avoir accès à l'ensemble de ces informations. Afin de restreindre l'accès aux autres visiteurs, il est important de passer par un middleware, qui vérifiera le lien entre médecin et visiteur.

```
/* Cette requête sera exécutée juste avant d'accéder au contenu du contrôleur qui lui est alloué */

/* Permet de récupérer l'id de la page (id du médecin) */
$id = $request->route()->parameters()['medecin'];
$verifRapports = Rapport::where( column: 'medecin_id', operator: '=', value: "$id")
    ->where( column: 'visiteur_id', operator: '=', auth()->user()->id)
    ->limit( value: 1)
    ->get();

/* Si la requête SQL ne retourne aucun résultat, alors le compte n'est pas lié au médecin, et n'a donc pas accès au dossier */
if (empty($verifRapports[0])) {
    abort( code: Response::HTTP_FORBIDDEN); /* Erreur 403 */
}
```

Comme expliqué ici, on récupère l'id du médecin contenu dans l'URL, et on la compare à celle de l'utilisateur actuellement connecté, via la fonction `auth()` que fournit Laravel. La requête peut renvoyer soit un résultat, soit une valeur nulle. Si la valeur est nulle, alors le visiteur ne possède aucun médecin ayant cet identifiant dans son portefeuille, et ne doit pas avoir accès au contenu.

Gestion des rapports

Recherche de rapport:

Une fois le visiteur sur la page de gestion des rapports, ce dernier se retrouve face à un tableau contenant l'ensemble des rapports qu'il a effectué. Ses rapports pouvant s'amasser rapidement, il peut être inconfortable pour le visiteur de devoir chercher dans ce tableau un rapport précis, c'est pourquoi une barre de recherche basée sur la date du rapport est intégrée. Une fois renseignée, le tableau ne sera rempli que par les rapports effectués à une date correspondante à celle entrée par le visiteur.

```
if (request( key: 'rapport-query')) {
  if ($query == "") {
    return back()
    ->withErrors(['empty' => 'Veuillez remplir le champ avant l\'envoi du formulaire.']);
  } else {
    $visites = Rapport::where( column: 'visiteur_id', auth()->user()->id)
    ->where( column: 'date', operator: '=', value: "$query")
    ->get();
  }
} else {
  $visites = Rapport::where( column: 'visiteur_id', auth()->user()->id)
  ->orderBy( column: 'date', direction: 'DESC')
  ->get();
}
```

Une requête par défaut sélectionne l'ensemble des rapports, c'est celle qui s'effectue au chargement de la page. Cependant, une autre requête vient la supplanter si une requête est effectuée. Cette nouvelle requête exigera que la date doive correspondre à la date indiquée par le visiteur.

Le tableau, peu importe s'il provient d'une recherche ou non, indique l'ensemble des informations concernant un rapport, cela concerne également les offres de médicaments, ainsi que le nom complet du médecin concerné. Afin de mener cela à bien et de respecter une méthode dite « objet », Laravel, comme d'autres Frameworks, met en lumière le système de relations entre tables.

Il est possible de relier des modèles entre eux, à partir des clefs étrangères des tables à laquelle ils correspondent. Nous prendrons tout naturellement l'exemple de la classe « Rapport » dans ce cas :

```
public function medecin(){
  return $this->belongsTo( related: Medecin::class); /* Le rapport détient un médecin */
}

public function offrirs(){
  return $this->hasMany( related: Offrir::class); /* Le visiteur à pu recevoir plusieurs médicaments */
}
```

On définit dans la classe Rapport les relations impliquant la table « rapports ». On sait qu'un rapport contient un et un seul médecin, mais qu'il peut contenir plusieurs offres de médicaments.

La méthode de dépendance pourrait s'apparenter à celle d'un Modèle Conceptuel de Données, ou l'on y précise les cardinalités. Les précisions devront se refléter sur les classes auxquelles « Rapport » est à présent liée. Exemple pour « Medecin » :

```
public function rapports(){
    return $this->hasMany( related: Rapport::class); /* Le médecin détient plusieurs rapports */
}
```

La classe métier « Médecin » contient bien le reflet de cette méthode. On précise donc dans la classe Rapport qu'elle appartient à la classe Médecin, et on précise dans un second temps que la classe Médecin possède plusieurs rapports. Le respect de la syntaxe concernant les pluriels est une fois de plus nécessaire.

Ainsi, par cette méthode, il est possible de récupérer l'ensemble des informations liées au rapport par une seule et même variable, facilitant sa mise en page dans la vue.

```
<td>{{ $visite->motif }}</td>
<td>{{ $visite->date }}</td>
<td class="visits-categ-1">{{ $visite->medecin->prenom." ".$visite->medecin->nom }}</td>
<td class="visits-categ-2">
    @foreach($visite->offrirs as $offrir )
        {{ $offrir->medicament->nomCommercial }}<br>
    @endforeach
</td>
<td class="visits-categ-2">
    @foreach($visite->offrirs as $offrir )
        {{ $offrir->quantite }}<br>
    @endforeach
</td>
```

De simple boucles foreach() permettent de parcourir les collections correspondantes, et de récupérer les valeurs demandées.

Ajouter un rapport:

L'ajout de rapport se basant sur plusieurs champs dynamique, cette partie contient une part importante de code en JavaScript, contrairement au reste de l'application web.

Étant donné que le visiteur peut se voir attribuer de multiples médicaments, il est important de lui laisser gérer le nombre de champs disponibles.

Comme mentionné ultérieurement, Les fichiers de vue Blade offrent la possibilité d'intégrer des variables PHP dans du code JavaScript, nous allons donc définir l'ensemble des variables dans ce fichier de vue, puis les scripts nécessaires dans des fichiers à part.

```
];
let medicName = [
    @foreach($medicaments as $medicament)
        "{{ $medicament->nomCommercial }}",
    @endforeach
];
```

Par exemple, on récupère ici l'ensemble des noms commerciaux des médicaments, qu'on stocke dans une variable javascript. Les variables seront accessibles depuis les fichiers JavaScript dédiés.

Afin de gérer le nombre de champs offrir, on va attribuer à chaque champ un numéro. Un nouveau champ, prenant la valeur du dernier +1 pourra être ajouté tant que cette valeur ne dépasse pas le nombre maximal, ici 5.

Sachant qu'un champ à n'importe quelle position peut être supprimé par l'utilisateur, il est important de décrémenter de 1 chaque champ ayant une valeur plus haute que celui qui est enlevé.

```
/* Pour le bouton, on modifie à la fois la valeur passée à la fonction Js, et son id */
allRemoveButtons[i].setAttribute( qualifiedName: 'id', value: 'remove-'+(categId.match(regex) - 1));
allRemoveButtons[i].setAttribute( qualifiedName: 'onclick', value: 'supprimerVisite('+(categId.match(regex) - 1)+'')');
```

On décrémente de 1 la valeur correspondante, à la fois sur l'id HTML de la section, et sur l'action du bouton de suppression.

Le formulaire devait également comprendre un champ de recherche de médecin dynamique, permettant à l'utilisateur de trouver un médecin par le début de son nom, et d'en prendre la valeur en cliquant dessus. La méthode utilisée ici est de stocker l'ensemble des médecins dans une variable javascript, et de comparer le début de chacun par la valeur entrée dans la barre de recherche. Si cela correspond, on affiche la valeur en l'incorporant dans le code HTML.

```
valueArray = valueArray.map((contenu :string )=>{
  /* Les propositions seront contenues dans des balises <li>, afin que l'affichage sur la page s'effectue correctement */
  return contenu = '<li>'+contenu+'</li>';
});
```

```
suggestions.innerHTML = listeValeurs;
```

La fonction innerHTML permet d'ajouter des balises HTML depuis le script. Par défaut, la recherche est une liste vide, cependant, si une recherche correspond, on l'ajoute à l'HTML en la mettant entre des balises de liste « ». Les valeurs apparaîtront donc dans l'HTML, et s'actualiseront en temps réel.

Modifier un rapport:

La méthode de modification d'un rapport est factuellement très similaire à celle adoptée pour modifier les informations d'un médecin. On passe le bilan et le motif de la visite ciblée à la vue, qui seront les valeurs des champs par défaut.

```
<label for="motive" class="block mb-2 uppercase font-bold text-xs text-gray-700">
  Motif
</label>
<input type="text" name="motive" class="border border-gray-400 p-2 w-full" value="{{ $leRapport->motif }}">
```

Une vérification de différence entre la valeur des champs et le contenu de la base de données est encore une fois effectuée, afin d'éviter les opérations inutiles le plus possible.