DOCUMENTATION TECHNIQUE

Galaxy Swiss Bourdin

Projet « GSB Gestion Visites »

TABLE DES MATIÈRES

PRÉSENTATION DU PROJET

1.1 – 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.

1.2 – 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.

1.3 – 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.

1.4 – 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 connait 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.

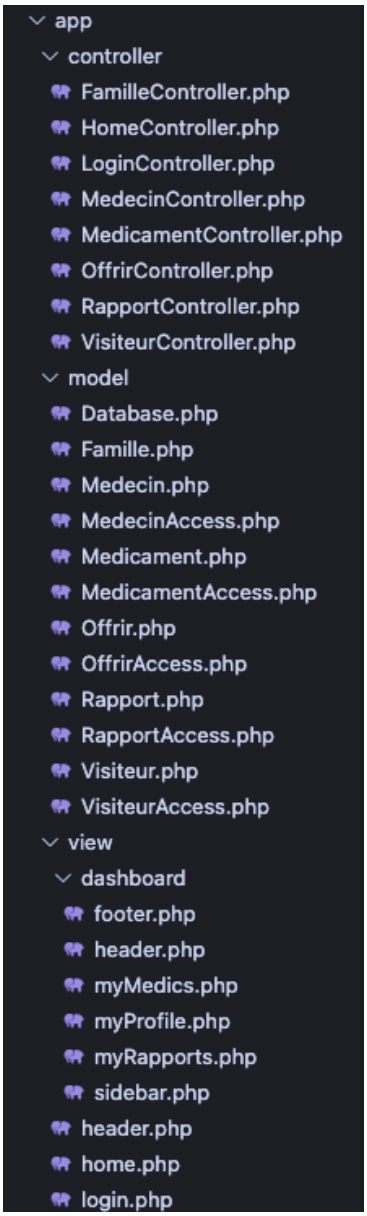
1.5 – Les médecins.

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.

FONCTIONNEMENT DE L'APPLICATION

2.1 – Fonctionnement technique.

2.1.1 – Structure de l'application.



La structure de l'application est telle qu'elle répertorie les différents contrôleurs *(dans le dossier **controller**)*, les différents modèles *(dans le dossier **model**)* ainsi que les différentes vues *(dans le dossier **view**)*.

Chaque fichier travaille sur un ou plusieurs éléments liés à eux. Par exemple, le **RapportController.php** *(contrôleur)* ainsi que le **Rapport.php** et **RapportAccess.php** fonctionnent ensemble pour tout ce qui est lié de près ou de loin aux différents rapports.

Toutes les différentes méthodes liées au sujet du fichier seront alors inscrites dedans et seulement dedans.

Le fichier **Database.php** sera la classe principale utilisant le **PDO**.

Dans la partie « **Vue** » *(dossier view)*, il y a un premier **header.php**, celui-ci est directement visible une fois le dossier **view** ouvert. Il servira donc à la navbar principale qui sera affichée sur la page de connexion et la page d'accueil. Le second **header.php**, quant à lui, sera visible dans le dossier **dashboard**, c'est donc la navbar qui sera affichée constamment quand l'utilisateur sera dans le tableau de bord. Le fichier **sidebar.php** sera également la navbar latérale du tableau de bord, permettant de naviguer facilement entre toutes les différentes pages disponibles. *(Gestion rapports, gestion médecins, etc.)*

FONCTIONNEMENT DE L'APPLICATION

2.1.2 – Modèle MVC

Pour cette application, j'ai utilisé le modèle (ou l'architecture) dit « **Modèle – Vue – Contrôleur** »

Le « **Modèle** » lui, va gérer toute la partie base de données de l'application *(site)*. Son rôle principal est de récupérer les informations « brutes » venant de la BDD *(base de données)*, le les organiser et de les renvoyer directement au **contrôleur**. Donc on y retrouve entre autres les différentes requêtes SQL, qui peuvent permettre de récupérer des informations comme cité plus haut ou bien alors de les envoyer.

La « **Vue** » sera l'affichage de l'application sur le navigateur, elle ne fera presque aucun calcul et elle se contentera juste de récupérer les seules variables qui lui seront utiles à l'affichage *(ex : Nom prénom d'un utilisateur, etc.)*. On y retrouve essentiellement du code HTML mais aussi un peu de PHP *(comme des boucles ou des conditions très simples)*.

Le « **Contrôleur** » va être la passerelle entre la **vue** et le **modèle**, c'est par ici que la vue pourra demander à récupérer des informations, le contrôleur va alors traiter la demande et la vérifier pour ensuite aller questionner le modèle et récupérer les informations initialement demandées. C'est un peu le « chef d'orchestre » de cette architecture. Il gère la partie logique du code et les décisions, cette partie d'ailleurs contient seulement du PHP.

FONCTIONNEMENT DE L'APPLICATION

2.1.3 – Le pattern des classes et leurs fonctionnement.

Partie contrôleur :

Durant cette partie, je vais décrire le fonctionnement « **basique** » d'une classe contrôleur dans mon application, elle aura pour fonctionnalité principale d'exécuter des méthodes, avec notamment des « **getter** » mais aussi des « **setter** » qui prendront la forme de requêtes **SQL** envoyées directement à la couche **modèle** de mon architecture **MVC**. Cette classe est nommée « **RapportController** », j'ai décidé de la prendre en exemple car c'est celle qui a le plus de méthode concrète et différente permettant de mieux comprendre le fonctionnement global de mon code et de ma manière de concevoir l'application. Enfin nous verrons aussi très rapidement le « **LoginController** » pour que vous puissiez savoir comment j'ai organisé mon système de connexion au site.

Fichier « **RapportController.php** » :

```php
<?php
    // Namespaces & Uses
    namespace App\controller;
    use App\model\OffrirAccess;
    use App\model\RapportAccess;
```

Le début de chaque **contrôleurs** comme le **RapportController.php**, **MedecinController.php**, **LoginController.php**, et autre commencent par l'ajout d'un « **namespace App\controller ;** », cela permet d'ajouter le fichier actuel au « groupe » des contrôleurs, et de ce fait, l'ajout d'un contrôleur n'est plus nécessaire grâce à un « **include** » n'est plus nécessaire, car tous les **contrôleurs** seront en soit déjà « **réunis** » au même endroit, ce qui fait qu'ils pourront être utilisé dans n'importe quel autre **contrôleurs**.

```php
// Class
    class RapportController {
        private static $_instance = null;
        public static $page;
        public static $totalPages;

        private function __construct() {
            self::checkRapport();
        }
```

Une fois cela fait, nous créons alors la classe, ici « **RapportController** » *(Il faut obligatoirement que les noms des classes correspondent **exactement** au nom du fichier.)*, et nous pouvons y attribuer les différentes variables. Par ailleurs, celle qui restera constamment et qui sera également en **private** c'est la variable **static** nommée « **$_instance** », qui sera définie à **null** *(Elle sera par la suite utilisée)*. Et nous allons exécuter la méthode appelée ici « **checkRapport()** » qui va permettre de vérifier si un rapport à soit été ajouté, modifié ou bien recherché par date.

```php
private static function checkRapport() : void {
            // L'ajout d'un rapport
            if(isset($_POST['rapport_Motif'])) {
                $medic = $_POST['rapport_Medic'];
                $motif = $_POST['rapport_Motif'];
                $bilan = $_POST['rapport_Bilan'];
                $date = $_POST['rapport_Date'];
                $convertedDate = self::dateToEnglishFormat($date);

                RapportAccess::addRapport($convertedDate, $motif, $bilan,
$_SESSION['id_VISITOR'], $medic);

                if(!empty($_POST['rapport_Medication']) &&
!empty($_POST['rapport_Quantity'])) {
                    foreach(array_combine($_POST['rapport_Medication'],
$_POST['rapport_Quantity']) as $key => $val) {
                        $lastRapport = RapportAccess::getLastRapport();
                        OffrirAccess::addOffrir($lastRapport, $key, $val);
                    }
                }
            }
```

Le « **checkRapport()** » sera donc une méthode dite « **privée** », et de plus nous allons grâce à des conditions, vérifier grâce à **$_POST** le type de rapport qui a été renvoyé *(Si c'est par exemple un **$_POST** concernant l'ajout d'un rapport, il va*

alors le vérifier et donc voir que c'est bien un ajout, il exécutera donc le reste du code).

Vous pouvez d'ailleurs voir que dans cette méthode, nous ajoutons d'abord le Rapport grâce à la méthode « **addRapport()** » directement créée dans la DAO *(**D**ata **A**ccess **O**bject)* Rapport nommée « **RapportAccess** ». Dans ce cas précis c'est pour l'ajout des différents échantillons, car dans cette table *(Table « **Offrir** »)*, il est demandé l'**ID** du rapport auquel le visiteur a attribué les échantillons, pour cela j'ai donc d'abord créé le rapport, et j'ai pu ensuite récupérer l'**ID** de celui-ci dans la méthode qui suit, nommée « **addOffrir()** » qui fait partie de la DAO **Offrir** grâce à la méthode « **getLastRapport()** », car en effet pour simplement savoir à quel **ID** de rapport les échantillons ont été donnés, il suffit de prendre le dernier en date qui a été créé, et comme le rapport est créé juste avant l'ajout des échantillons, ça sera obligatoirement le rapport en question.

```
// Édition d'un rapport
            if(isset($_POST['rapport_Motif_edit']) &&
isset($_POST['rapport_Bilan_edit'])) {
                $id_rapport = $_GET['editRapport'];
                $motif = $_POST['rapport_Motif_edit'];
                $bilan = $_POST['rapport_Bilan_edit'];

                RapportAccess::updateRapport($id_rapport, $motif, $bilan);
            }

            // Rechercher d'un rapport par date
            if(isset($_POST['rapport_EditDate'])) {
                header('Location: ./?action=myRapports&page=' . $_GET['page'] .
'&date=' . self::dateToEnglishFormat($_POST['rapport_EditDate']));
            }

            // Suppression d'un rapport
            if(isset($_GET['delete'])) {
                RapportAccess::deleteRapport($_GET['delete']);

                header('Location: ./?action=myRapports&page=' . $_GET['page']);
            }
```

Pour la suite du code, cela est sensiblement pareil, nous vérifions d'abord dans quelle condition nous devons exécuter les différentes parties de code grâce à la méthode **$_POST**, et ensuite on enchaîne avec le code.

```
public static function dateToEnglishFormat(string $date) {
            $convertedDate = date('Y-m-d', strtotime($date));
```

```
            return $convertedDate;
        }

        public static function dateToFrenchFormat(string $date) {
            $convertedDate = date('d-m-Y', strtotime($date));

            return $convertedDate;
        }
```

Il faut savoir, que pour les méthodes ci-dessus, qu'elles ont été créées pour un affichage plus « **français** », car en effet, dans les BDD, la date est toujours affichée en modèle « **middle-endien** », c'est-à-dire sous cette forme : **MM/JJ/AAAA**, mais en base de données, c'est sensiblement la même chose si ce n'est que l'année passe avant, donc sous cette forme : **AAAA-JJ-MM**, grâce donc aux méthodes ci-dessus, il est possible de retranscrire automatiquement une date qui était à l'origine en norme américaine (cf. « **middle-endien** ») en norme française (« **little-endien** »).

```
public static function getEveryRapportFrom($id_visitor) : array {
            // Paginations variables
            $limit = 10;
            $allRecords = RapportAccess::getTotalRecords($_SESSION['id_VISITOR']);
            self::$totalPages = ceil($allRecords / $limit);
            self::$page = (isset($_GET['page']) && is_numeric($_GET['page']) ) ?
$_GET['page'] : 1;
            $paginationStart = (self::$page - 1) * $limit;

            // Get every rapport from a visitor ID
            $rapport_collection = RapportAccess::getEveryRapportFrom($id_visitor,
$paginationStart, $limit);

            // Return the objects array
            return $rapport_collection;
        }

        public static function getEveryRapportOfAMedic($id_medic) : array {
            $rapports = RapportAccess::getEveryRapportOfAMedic($id_medic);

            return $rapports;
        }
```

Maintenant, nous voyons des « **getters** », l'un utilise un système de **pagination** qui vous sera possible de totalement retranscrire dans une autre page en faisant un simple copier/coller si ce n'est qu'il faudra donc changer les DAO auxquels vous voulez accéder. Celui-ci fonctionne avec une limite que je lui ai attribué *(Ici 10*

rapports maximum à afficher) et je récupère tous les rapports que l'utilisateur a pu rédiger grâce à son ID qui est stockée dans une variable **$_SESSION** qui a été auparavant définie dans le fichier « **LoginController.php** ». Je calcule le nombre total de pages qu'il y aura à la fin en divisant simplement le nombre de rapports trouvés par la limite que j'ai souhaité définir et je continue mon code à partir de ça.

Le second « **getter** » est celui que vous retrouverez dans tous les **contrôleurs**, c'est le plus basique. J'envoie à mon modèle une demande, en lui demandant par exemple dans ce cas précis de récupérer tous les rapports concernant un médecin ayant pour **ID** le paramètre rentré dans la méthode et je stocke ça dans une variable que je retournerais grâce à un « **return** ».

```
    public static function getInstance() : object {
        if(is_null(self::$_instance)) {
            self::$_instance = new RapportController;
        }

        return self::$_instance;
    }

    public function render() : void {
        $this->activePage = 'myRapports';

        include_once 'app/view/dashboard/header.php';
        include_once 'app/view/dashboard/sidebar.php';
        include_once 'app/view/dashboard/myRapports.php';
        include_once 'app/view/dashboard/footer.php';
    }
```

La méthode **getInstance()** permet de récupérer l'instance de la classe, elle va créer directement l'objet qui sera instancié par la suite dans la variable statique appelée **$_instance**. Nous pourrons donc directement appeler les méthodes de cette-même classe autre part en utilisant le **getInstance()**. Exemple : **RapportController::getInstance()->render()**; Le render permettant donc d'inclure *(et de faire afficher par la même occasion)* les fichiers rentrés.

Fichier « **LoginController.php** » :

```
// Class
    class LoginController {
        # Variables
        private static $_instance = null;

        # Functions
        private function __construct() {
```

```
            self::checkLogin();
        }

        private static function checkLogin() : void {
            if(isset($_POST['login_VISITEUR']) && isset($_POST['mdp_VISITEUR'])) {
                $postLogin = $_POST['login_VISITEUR'];
                $postPass = $_POST['mdp_VISITEUR'];

                self::login($postLogin, $postPass);
            }
        }
```

De la même manière que le « **RapportController** », j'ai donc créé une méthode « **__construct()** » ayant à l'intérieur un **checkLogin()** permettant de vérifier si un **$_POST** *(celui venant donc du formulaire de connexion)* avait été envoyé, si oui alors il exécute la méthode « **login()** » disponible ci-dessous.

```
private static function login($login, $pass) : void {
            if(!isset($_SESSION)) { session_start(); }

            $visitor = VisiteurAccess::getByLoginAndPass($login, $pass);

            if(!empty($visitor)) {
                $visitorDB_Login = $visitor->getLogin();
                $visitorDB_Pass = $visitor->getMDP();
```

En premier lieu, je vérifie si une variable de **$_SESSION** a déjà été engagée, si ce n'est pas le cas alors je le fais. Je récupère ensuite dans la variable « **$visitor** » l'objet visiteur comportant exactement le même login ET mot de passe. Biensûr, celui-ci ne peut être un login venant d'un compte et le mot de passe venant d'un autre compte, il vérifie bien que celui-ci appartient bien à un seul et même compte.

Je vois ensuite grâce à la fonction « **empty()** » si la variable « **$visitor** » n'est pas vide, si ce n'est pas le cas c'est qu'elle a donc renvoyé quelque chose, sinon dans le cas contraire je renvoie vers la page de connexion et une erreur s'affiche. Celle-ci s'affiche car je modifie légèrement l'Url de renvoie, maintenant elle comportera le paramètre « **&error** » qui va permettre à la page de détecter si une erreur est arrivée.

```
        $visitorDB_ID = $visitor->getID();
                    $visitorDB_FName = $visitor->getNom();
                    $visitorDB_LName = $visitor->getPrenom();
                    $visitorDB_Address = $visitor->getAdresse();
                    $visitorDB_CP = $visitor->getCP();
                    $visitorDB_City = $visitor->getVille();
                    $visitorDB_DE = $visitor->getDateEmbauche();

                    # Set $_SESSION variables from databases values
                    $_SESSION['login_VISITOR'] = $visitorDB_Login;
                    $_SESSION['pass_VISITOR'] = $visitorDB_Pass;

                    $_SESSION['id_VISITOR'] = $visitorDB_ID;
                    $_SESSION['fname_VISITOR'] = $visitorDB_FName;
                    $_SESSION['lname_VISITOR'] = $visitorDB_LName;
                    $_SESSION['address_VISITOR'] = $visitorDB_Address;
                    $_SESSION['cp_VISITOR'] = $visitorDB_CP;
                    $_SESSION['city_VISITOR'] = $visitorDB_City;
                    $_SESSION['de_VISITOR'] = $visitorDB_DE;

                    header('Location: .');
                }
```

Sinon je récupère toutes les valeurs renvoyées par la base de données, je les mets dans des variables pour que cela soit plus lisible, je les attribues dans les différentes variables **$_SESSION** et je renvoie l'utilisateur sur la page d'accueil grâce au « **header()** ».

```
public static function logout() : void {
            if(isset($_SESSION['login_VISITOR']) && isset($_SESSION['pass_VISITOR']))
{
                session_destroy();

                header('Location: .');
            }
        }
```

Finalement, pour le système de déconnexion, je prends soin de bien détruire toutes les variables **$_SESSION** existantes grâce à la méthode « **session_destroy()** » et je renvois ensuite vers une page « **.** » *(Qui est gérée grâce à « **Index.php** », car celui-ci va gérer les différentes « **?action=** » qui seront présentes dans l'URL, et il se permettra donc de rediriger l'utilisateur selon des conditions bien précises, comme par exemple la vérification pour savoir si celui-ci est connecté avant de vouloir accéder à la page des rapports, s'il ne l'est pas alors on renvoie directement vers la page de connexion)*.

Fichier « **Index.php** » :

```php
// Every action switch case
    switch($action) {
        case 'myProfile':
            if(VisiteurController::isConnected()) {
                VisiteurController::getInstance()->render();
            } else {
                LoginController::getInstance()->render();
            }
            break;

        case 'myMedics':
            if(VisiteurController::isConnected()) {
                MedecinController::getInstance()->render($action);
            } else {
                LoginController::getInstance()->render();
            }
            break;

        case 'myRapports':
            if(VisiteurController::isConnected()) {
                RapportController::getInstance()->render($action);
            } else {
                LoginController::getInstance()->render();
            }
            break;
```

Voici un court exemple du code disponible dans le « **Index.php** »

FONCTIONNEMENT DE L'APPLICATION

2.1.3 – Le pattern des classes et leurs fonctionnement.

Partie modèle :

Durant cette partie, je vais décrire le fonctionnement « **basique** » de deux classes modèles dans mon application, elles auront pour fonctionnalité principale d'exécuter des méthodes, mais notamment des « **getter** », même si des « **setter** » seront tout à fait visible.

Fichier « **Rapport.php** » :

```php
<?php
    // Namespaces & Uses
    namespace App\model;

    // Class
    class Rapport {
        # Variables
        private int $id;
        private string $date;
        private string $motif;
        private string $bilan;
        private string $idVisiteur;
        private string $idMedecin;

        # Constructor
        public function __construct(int $id, string $date, string $motif, string
$bilan, string $idVisiteur, string $idMedecin) {
            $this->id = $id;
            $this->date = $date;
            $this->motif = $motif;
            $this->bilan = $bilan;
            $this->idVisiteur = $idVisiteur;
            $this->idMedecin = $idMedecin;
        }
```

Voici le schéma basique d'une classe métier, elle est ajoutée à la même manière que les **contrôleurs** à un « **namespace** » mais cette fois utilisée que par les modèles.

Nous lui attribuons toutes ses variables privées, et nous créeons donc le « **__construct()** » permettant de faire un système de « **setter** » sur ces variables pour les définirs comme elles ont été rentrées lors de l'utilisation de cette classe *(ex : **new Rapport(1, '2002-13-01', 'Motif', 'Bilan', 'a131', '238')**)*. À la manière des **contrôleurs**, le fonctionnement est le même sur toutes les classes métiers, comme précisé c'est un « **pattern** » à suivre.

```
# Functions
        public function getID() : int { return $this->id; }
        public function getDate() : string { return $this->date; }
        public function getMotif() : string { return $this->motif; }
        public function getBilan() : string { return $this->bilan; }
        public function getIDVisiteur() : string { return $this->idVisiteur; }
        public function getIDMedecin() : string { return $this->idMedecin; }
```

Et enfin, nous créons les méthodes publiques permettant de récupérer ces différentes variables qui auront été définies lors de la création d'un objet utilisant cette classe. Il est important que vous utilisez un typage pour les méthodes *(ainsi que pour les variables, qu'elles soient dans un **contrôleur** ou un **modèle**)*, cela rend le code plus lisible et une erreur apparaîtra si la méthode vous retourne pas ce qui est censé être donné, et c'est très utile.

Fichier « **Database.php** » :

```
<?php
    //
    // Class name: Database
    //

    // Namespaces & Uses
    namespace App\model;
    use PDO;

    // Class
    class Database {
        # Variables
        private static $conn_DB = null;
        private static $host_DB = '127.0.0.1';
        private static $name_DB = 'gsb_gestion_visites';
        private static $user_DB = 'root';
        private static $pass_DB = '';

        # Functions
        protected static function getPDO() : object {
            if(is_null(self::$conn_DB)) {
```

```
                $connectPDO = new PDO('mysql:host=' . self::$host_DB . ';dbname=' .
self::$name_DB . ';charset=utf8', self::$user_DB, self::$pass_DB);
                $connectPDO->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

                self::$conn_DB = $connectPDO;
            }

            return self::$conn_DB;
        }
```

Voici une présentation du fichier « **Database.php** », et cela vous montre quel pattern j'utilise pour la création d'un code me permettant d'utiliser la base de données. Je l'ajoute donc au « **namespace** » des modèles et je prends soin d'utiliser la librairie « **PDO** » me permettant d'accéder à une base de données et d'y exécuter différentes requêtes. Dans cette classe « **Database** », à la même manière qu'une méthode « **getInstance()** » je créer une méthode nommée ici « **getPDO()** » me permettant de réutiliser l'objet **PDO** que j'aurais créé auparavant et qui utilise directement tous les bons paramètres au lieu de devoir le refaire entièrement à chaque nouvelle requête.

```
protected static function query($statement) : object {
            $stat = self::getPDO()->query($statement);

            return $stat;
        }

        protected static function prepare($statement, $attributes) : array {
            $stat = self::getPDO()->prepare($statement);
            $stat->execute($attributes);

            return $stat->fetchAll();
        }

        protected static function request($statement, $attributes) : void {
            $stat = self::getPDO()->prepare($statement);

            $stat->execute($attributes);
        }
```

Voici le fonctionnement des requêtes, j'ai créé différentes méthodes pour différentes requêtes. La première, « **query()** » permet simplement d'exécuter une requête et de retourner ce qu'elle renvoie sans utiliser de système de « **BindValue()** », c'est-à-dire que dans cette méthode, il n'est pas prit en compte les conditions « **WHERE** » d'une requête SQL. Elle est donc créée pour les requêtes du type : « **SELECT * FROM Rapport ;** ». La méthode « **prepare()** » quant à elle est prévue pour la même chose,

mais en y ajoutant des conditions. Finalement, la méthode « **request()** » permet d'envoyer des requêtes ne nécessitant pas de retour de valeur(s), comme des **INSERT** ou **UPDATE** mais qui nécessite soit des conditions, soit des variables, comme pour le **INSERT** où il est obligatoire d'informer le système des valeurs que l'on veut rajouter.

Fichier « **RapportAccess.php** » :

```php
<?php
    //
    // Class name: RapportAccess
    //

    // Namespaces & Uses
    namespace App\model;
    use PDO;

    // Class
    class RapportAccess extends Database {
        # Functions
        public static function getTotalRecords(string $id_visitor) : int {
            $query = self::query("SELECT COUNT(id) AS ID FROM Rapport WHERE idVisiteur
= '" . $id_visitor . "';");
            $int = 0;

            foreach($query as $rows) {
                $int = $rows['ID'];
            }

            return $int;
        }
```

Voici la présentation basique d'un modèle DAO. Nous l'ajoutons donc au « **namespace** » des modèles et nous utilisons la librairie **PDO**. Les classes DAO sont toujours des héritages de la classe « **Database** » qui a été montrée plus haut, car nous devons pouvoir utiliser les méthodes de cette même classe. Et voici une méthode « **getter** » permettant de récupérer le nombre total de rapports qui ont été créés.

```php
public static function getRapportByID(int $id_rapport) : object {
            $request = self::prepare('SELECT * FROM Rapport WHERE id = :id_rapport
ORDER BY date DESC;', array(':id_rapport' => intval($id_rapport)));

            if(!empty($request)) {
```

```
                return new Rapport($request[0]['id'], $request[0]['date'],
$request[0]['motif'], $request[0]['bilan'], $request[0]['idVisiteur'],
$request[0]['idMedecin']);
            } else {
                return null;
            }
        }

        public static function getEveryRapportFrom(string $id_visitor, int $start, int
$end) : array {
            $request = self::prepare('SELECT * FROM Rapport WHERE idVisiteur =
:idVisiteur ORDER BY date DESC LIMIT ' . $start . ', ' . $end . '',
array(':idVisiteur' => $id_visitor));
            $collection = array();

            if(!empty($request)) {
                foreach($request as $rows) {
                    $collection[$rows['id']] = new Rapport($rows['id'], $rows['date'],
$rows['motif'], $rows['bilan'], $rows['idVisiteur'], $rows['idMedecin']);
                }
            }

            return $collection;
        }
```

Et enfin, voici deux méthodes dont une renvoie directement un objet, car la requête utilisée est « **précise** », c'est-à-dire qu'on veut un rapport dont l'**ID** est exactement celui rentré en paramètre, donc il ne peut y en avoir qu'un seul. *(Méthode : **getRapportByID()**)* et une deuxième permettant de renvoyer une collection d'objet *(Tableau (array) contenant des objets)*, ici la méthode « **getEveryRapportsFrom()** » servant à récupérer tous les rapports venant d'un visiteur ayant pour **ID** celui rentré en paramètre.

COMPOSANTS PRINCIPAUX

Les composants principaux de cette applications sont au nombre de 3.

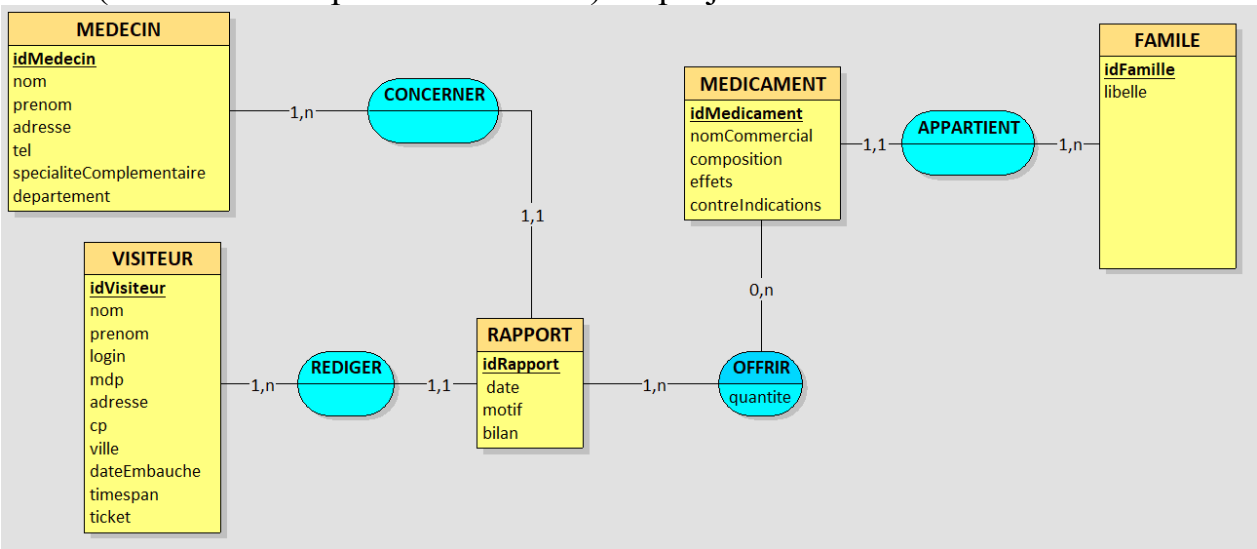
J'ai tout d'abord utilisé « **composer** » pour ce projet, celui est un logiciel gestionnaire de dépendances libre. Il a été écrit en PHP et permet aux utilisateurs *(moi et vous)* de déclarer et d'installer les bibliothèques dont le projet principal a besoin.

J'ai également utilisé une architecture dite « **MVC** » comme cité plus haut, me permettant grâce à des fichiers de bien répertorier les différents travaux que ceux-ci sont censés effectuer. Par exemple, la vue pour l'affichage dans le dossier « **view** », le contrôleur servant de passerelle dans le dossier « **controller** », etc.

Et enfin, j'ai utilisé le framework « **Bootstrap** » pour la partie CSS du site, car n'ayant pas le temps de le faire **Bootstrap** m'a alors été utile plus d'une fois.

DIAGRAMMES & UML

MCD (**M**odèle **C**onceptuel de **D**onnées) du projet :



MLD (**M**odèle **L**ogique de **D**onnées) du projet :

MEDECIN = (idMedecin *INT*, nom *VARCHAR(30)*, prenom *VARCHAR(30)*, adresse *VARCHAR(80)*, tel *VARCHAR(15)*, specialiteComplementaire *VARCHAR(50)*, departement *INT***);**
VISITEUR = (idVisiteur *CHAR(4)*, nom *CHAR(30)*, prenom *CHAR(30)*, login *CHAR(20)*, mdp *CHAR(20)*, adresse *CHAR(30)*, cp *CHAR(5)*, ville *CHAR(30)*, dateEmbauche *DATE*, timespan *BIGINT*, ticket *VARCHAR(50)***);**
FAMILE = (idFamille *VARCHAR(10)*, libelle *VARCHAR(80)***);**
RAPPORT = (idRapport *INT*, _date *DATE*, motif *VARCHAR(50)*, bilan *VARCHAR(50)*, *#idMedecin, #idVisiteur***);**
MEDICAMENT = (idMedicament *VARCHAR(30)*, nomCommercial *VARCHAR(80)*, composition *VARCHAR(100)*, effets *VARCHAR(100)*, contreIndications *VARCHAR(100)*, *#idFamille***);**
OFFRIR = (*#idRapport, #idMedicament*, quantite *INT***);**

UML (**U**nified **M**odeling **L**anguage) du projet :