

Titre professionnel CDA projet fil rouge Présentation

AMIO assistance helpdesk

Jalon n°1
développement
d'application



juin 2023

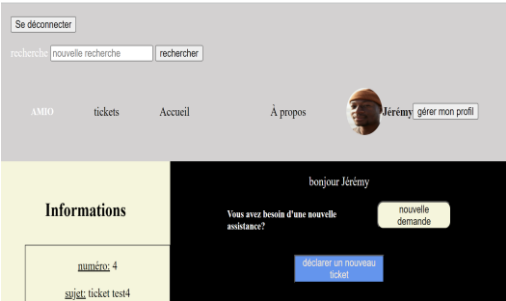
Commenté [AT1]: Ajouter le libellé de l'activité visée dans ce jalon 1

A moins que je le vois pas dans le dossier partagé (à cause de sharepoint je crois) mais, je ne vois pas ton nom, prénom ni la promotion sur la page de garde.

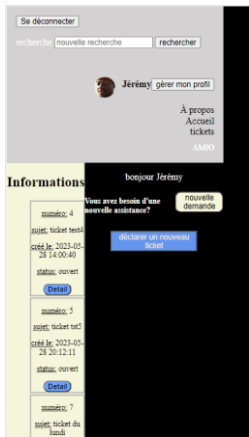
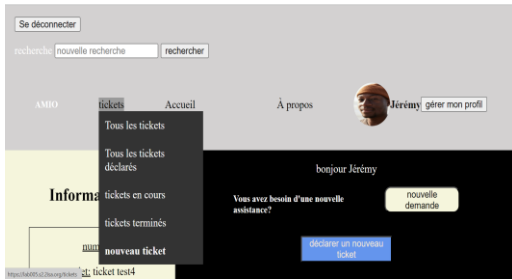
Prévoir d'insérer un pied de page comportant le numéro de page ainsi que nom et prénom, promotion CDA6

Table des matières

Introduction	3
1.1	3
1.2 Contexte pédagogique	3
1.3 Résumé et contexte du projet.....	3
1.4 Objectif :.....	3
2 Spécifications du projet :.....	3
2.1 Spécification des besoins fonctionnels :	3
2.2 Spécification technique.....	4
2.3 Architecture	4
2.4 Environnement de développement	4
2.5 Technologies :.....	4
3 Mise en place de l'environnement technique.....	4
3.1 Module et fonctionnalités.....	4
3.1.1 Utilisation du module Fortify	5
3.1.2	5
3.1.3 Fonctionnalités :	1
4 Réalisation	4
4.1 La base de données.....	4
4.1.1 Création des tables.....	4
4.1.2 Modification des tables	4
4.2 Partie front-end.....	4
4.2.1 Le maquetage.....	5
4.2.2 Balisage des pages.....	6
4.2.3 Le langage de Template blade	7
4.2.4 Le style	8

	9
--	---

Dossier projet



	9
5	Partie back-end:
5.1	Architecture MVC:.....
5.1.1	Schéma du MVC
5.1.2	Le concept du MVC
5.1.3	Exemple pour la récupération de tous les tickets de l'utilisateur
5.2	Éléments de sécurité.....
5.2.1	Requêtes https
5.2.2	Jeton CSRF
5.2.3	Classe DB ::.....
5.2.4	Version à jour du Framework utilisé :
5.2.5	Middleware* :
6	Conclusion.....
6.1	Bilan du projet.....
6.1.1	Problématiques rencontrées :.....
6.1.2	Solutions opposées :
6.2	Apports personnels et professionnels.....
6.3	Perspectives d'améliorations.....

Dossier projet

7	Annexes	9
7.1.1	Quelques requêtes SQL	9
7.2	Schéma de la base de données	11

Introduction

1.1

1.2 Contexte pédagogique

Ce projet intervient au cours de ma formation de Concepteur Développeur d'Application. Cette dernière se réfère à un référentiel officiel. Il requière la validation de compétences. Celle-ci sont réparties dans trois activités respectivement 'développer une application', 'concevoir et développer une application' et 'préparer le déploiement d'une application'.

Ces trois activités en plus de la période de stage en milieu professionnel servent à mettre en application et démontrer les compétences acquises au cours d'un "projet fil rouge". Cette simulation de projet reflétant un projet du métier est partitionné en deux jalons. Nous sommes ici dans le cadre du premier qui doit servir à démontrer les compétences suivantes :

- Gérer son environnement de travail
- Développer une interface utilisateur
- Développer les composants d'une application.

1.3 Résumé et contexte du projet

Dans le cadre d'une mise en place de leur solution d'assistance aux utilisateurs l'association AMIO demande le développement d'une application qui permettra à ses utilisateurs de faire une demande d'assistance sous forme de ticket par le biais de leur compte utilisateur en ligne. Ses demandes sous forme de ticket seront prises en charge par des responsables de dépannage par l'intermédiaire cette fois-ci d'une application desktop liée à la première. Les mises à jour de l'avancée de chaque ticket fait par les dépanneurs seront visibles par le réclamant côté application web via leur compte utilisateur. La communication entre les différents protagonistes se fera à l'aide d'un chat.

1.4 Objectif :

Pour ce premier jalon l'objectif est de fournir au client une première application web afin de permettre aux utilisateurs de créer un compte et de déclarer leur(s) demande(s) d'assistance sur la plateforme demandée par la maîtrise d'ouvrage.

2 Spécifications du projet :

2.1 Spécification des besoins fonctionnels :

- Pour cette application de ce premier jalon, celle-ci devra offrir la possibilité aux utilisateurs de créer et s'authentifier sur leur compte.
- Depuis celui-ci ils devront pouvoir déclarer une demande d'assistance auprès du service technique.

Commenté [AT2]: A la fin d'une ligne de titre, le signe de ponctuation ":" ne se met pas. Ce commentaire est donc à répercuter à plusieurs endroits de ton dossier

Commenté [AT3]: Avant d'introduire le contexte du projet, prévoir une partie avec présentation personnelle succincte et présentation formation CDA

Commenté [AT4]: Utiliser ce début de résumé pour finaliser celui à rendre à part (au format pdf) et qui doit comporter entre 200 et 250 mots. Apparemment, il faudrait ajouter le déroulement technique à ce début de résumé + ajouter liste de mots-clés

Commenté [AT5]: Ajouter ici la liste des compétences mises en œuvre dans ce jalon (reprendre les libellés tels quels)

Dossier projet

- Ces demandes d'assistance feront l'objet d'un suivi par la possibilité d'un échange par messages entre les différents protagonistes de la demande d'assistance en question.

2.2 Spécification technique

- Langage de programmation : PHP [10.10.17-4](#)
- Framework : Laravel 10.10.1
- SGBD : Maria DB

2.3 Architecture

Pour ce premier jalon l'application web sera développée en respectant d'architecture Model Vue Contrôleur proposé par le Framework Laravel utilisé ici.

2.4 Environnement de développement

le développement de l'application s'effectue directement sur le serveur mis à disposition par le client. J'ai écrit le code avec l'éditeur Visual studio. J'ai versionné ce même code avec l'outil git et utilisé également sa solution de serveur distant GitHub pour pérenniser le contenu de mon avancée dans ce projet.

2.5 Technologies :

- La base de données est gérée par le SGBD Maria DB. J'ai aussi utilisé le logiciel Dbeaver pour lancer mes scripts SQL et visionner le schéma ainsi que l'état de ma base de données, celui-ci est renseigné dans l'annexe de ce dossier. Le logiciel Dbeaver m'a également permis de vérifier le bon fonctionnement de mes différentes interactions avec la base de données et donc de valider le bon fonctionnement des fonctions du modèle sollicité.
-
- Le front-end est développé avec HTML5, CSS3 et animé en JavaScript vanilla.
- La partie back-end est traitée en PHP avec le Framework Laravel 10.8.

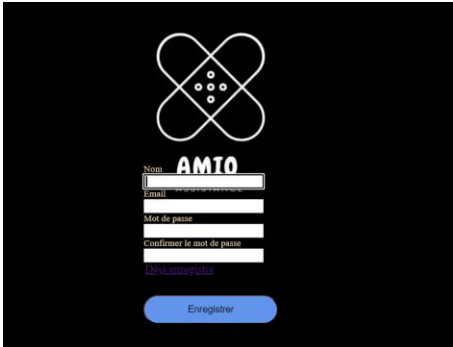
3 Mise en place de l'environnement technique

3.1 Module et fonctionnalités

Concernant la fonctionnalité d'authentification et de création de compte utilisateur elle est gérée côté serveur par la bibliothèque Fortify installée par l'intermédiaire du gestionnaire de paquets Composer.

J'ai adapté l'affichage des pages d'enregistrement et de connexion avec les formulaires que j'avais déjà écrit en les intégrant dans les vues créées à cet effet par l'outil fortify.

Dossier projet



3.1.1 Utilisation du module Fortify

Après avoir installé Fortify grâce au gestionnaire de paquet composer et lancé la commande :

```
php artisan :migrate.
```

A ce moment j'ai pu m'apercevoir que l'outil avait créé une table users, je la modifierai par la suite avec mon script de modification de mes tables afin d'y ajouter trois colonnes nécessaires à mon application (adress, tel et role). Fortify m'aura aussi créé un modèle et un contrôleur pour les users.

```
3 --Ajout des colonnes prenom et tel à la table user créé par fortify:
4 ALTER TABLE users
5 ADD COLUMN prenom varchar(10),
6 - ADD COLUMN tel varchar(10);
7 ADD COLUMN role varchar(10);
```

Figure 1modification de la table users

3.1.2

3.1.3 Fonctionnalités :

3.1.3.1 Création de ticket

- La route

Cette route correspondant à l'adresse /new de l'application appelle la fonction 'store' du contrôleur de la classe de la classe 'ticketController'

```
31 //route /new en post qui appelle la fonction create du contrôleur des tickets avec la fonction create qui
32 Route::post('/new', [ticketsController::class, 'store'])->name('ticket_create')->middleware('auth');
```

Figure 2 route/new

- Le ticketContrôleur

La fonction store du ticketsController détaillée dans la capture ci-dessous.

Elle prend comme paramètre un objet de la classe request. Cette classe permet de récupérer dans cette fonction les éléments envoyés dans la requête http. Cela m'a permis de pouvoir stocker les informations données dans l'input 'sujet' du formulaire. J'ai aussi pu récupérer l'identifiant de l'utilisateur actuellement authentifié visible ligne 76, la date actuelle elle est récupérée grâce au module Carbon qui est une bibliothèque de manipulation de dates. J'ai fait appel à la fonction du modèle qui permet de créer un nouveau ticket en base de données en créant une nouvelle instance de la classe ticketModel afin de faire appel à l'une de ses fonctions, 'store en occurrence détaillé dans la capture d'écran du modèle un peu plus bas.

L'objet Request passé en paramètre de la fonction représente la requête HTTP entrante. Il m'a permis de pouvoir récupérer la valeur de l'input 'sujet' du formulaire et l'identifiant de l'utilisateur authentifié. Ces éléments stockés dans des variables seront passés au modèle qui les insèrera dans la base de données.

```
73 public function store(Request $request)
74 {
75     // Récupération des différentes valeurs des inputs du formulaire dans des variables.Ces variables seront données
76     $sujet = $request->input('sujet');
77     $id_auteur = auth()->user()->id;
78     $cdat = Carbon::now(); // Date actuelle
79     $id_status=1; //Par défaut le ticket est à 1 =ouvert.
80     //Nouvelle instance de la classe ticketModel:
81     $ticketModel = new TicketModel();
82     $res = $ticketModel->store($sujet,$id_auteur,$id_status,$cdat);
83     $ticketModel = new TicketModel();
84     $id = $request->route("idTicket");
85     $status=$ticketModel->get_ticket_status($id);
86     $tickets = $ticketModel->getAllTickets();
87     //Si la création est retournée false on renvoie vers le formulaire de création avec un message d'erreur
88     if (!$res) {
89         return view('new_ticket', ['message' => 'Ticket non créé !', 'tickets' => $tickets, 'status'=>$status]);
90     } //Sinon on renvoi vers le détail du ticket créé avec un message:
```

Le modèle :

Appel de la fonction avec les paramètres lui étant nécessaires

La flèche du dessus entre le contrôleur et le modèle montre que la fonction du contrôleur crée une nouvelle instance du modèle puis utilise une de ses fonctions ('store') du modèle.

```
28 function store($sujet, $id_auteur,$id_status, $cdat)
29 {
30
31
32     try {
33         $id = DB::table('ticket')->insertGetId([
34             'sujet' => $sujet,
35             'id_auteur'=>$id_auteur,
36             'id_status'=>$id_status,
37             'created_dat' => $cdat,
38         ]);
39         return $id;
40     } catch (Exception $e) {
41         return false;
42     }
43     return true;
```

Dossier projet

La fonction attribue à chacune des colonnes de la table ticket des valeurs avec celles des arguments qui lui sont fournis.

3.1.3.2 Le postage de messages dans une conversation appartenant au message

1. Le formulaire : dans la page affichant le détail du ticket demandé, la conversation lui étant rattachée est visible.
En bas de cette conversation l'utilisateur peut trouver un formulaire dans lequel se trouve un input de type text qui lui servira à écrire son message le bouton type submit utilisé pour l'envoi déclenche le processus pour le postage du message dans la Dans la page correspondante au détail du ticket voulu l'utilisateur à accès à la conversation appartenant à ce ticket, il à la possibilité de rédiger un message et

- Affichage



- HTML

```
2 <h2><u>détail de la conversation du ticket: {{ $ticket->id }}</u></h2>
3 <button id="msgon" style="margin-bottom: 10px;cursor: pointer;">cacher la conversation</button>
4 <div id="conversation">
5   <ul>
6
7     @foreach ($msg as $ms)
8       <li style="line-height: 0px;">
9         <p><b><u>le: </u></b>{{ $ms->created_at }}</p><b><u>
10          <p>auteur:</u> {{ $ms->name }}</p>
11          </b>
12          <div>
13            <p>{{ $ms->content }}</p>
14          </div>
15        </li>
16      @empty
17      @endforeach
18      <form action="{{route('ticket_detail', ['n'=>$ticket->id])}}" method="post" id="msg_ticket">
19        @csrf
20        <label for="new_msg"> nouveau message</label>
21        <input type="text" name="new_msg" placeholder="écrivez votre message">
22        <button type="submit">envoyer</button>
23      </form>
24    </ul>
25  </div>
```


Dossier projet

2. La route post : elle est appelée lors de la soumission du formulaire en méthode POST pour l'envoi de message visible au-dessous. Cette route fait appel à la fonction 'storemsg' de la classe ticketController dans son chemin apparait l'identifiant du ticket qui servira à lier le message à ce ticket

```
48 //Route pour l'envoi d'un nouveau message:
49 Route::post('ticket/{id}',[ticketsController::class, 'storemsg'])->name('sumition_msg');
50
```

```
150 function storemsg( Request $request) {
151     $newticketModel=new TicketModel();
152     $ticketId= $request->route("id");
153     $contentMessage=$request->input('new_msg');
154     $auteurId=auth()->user()->id;
155     $n_msg=$newticketModel->storemsg($ticketId,$contentMessage,$auteurId);
156
157     return redirect()->route('ticket_detail', ['n' =>$ticketId]);
158 }
159 //La fonction userRight récupère le role de l'utilisateur actuellement auten
1reference|0 overrides
160 function userRight( Request $request) {
161     $user=auth()->user();
162     $NticketController= new User(); //Fait appel au modèle des users pour vérifier
163     $right= $NticketController->currentUser($user);
164     //pour l'aside:
165     $ticketModel = new TicketModel();
166     $tickets = $ticketModel->getalltickets();
167
168     return view('admin', ['user'=>$user, 'tickets'=>$tickets]);
169 }
```

```
75 function storemsg($ticketId, $contentMessage, $auteurId) {
76     try{
77         //Debut Transaction
78         DB::beginTransaction();
79         // Insertion du message
80         DB::insert
81         ["insert into messages
82         (id_auteur, content, created_at) values (?, ?, ?)", [$auteurId, $contentMessage, Carbon::now()]];
83         $idMessage = DB::selectone('select LAST_INSERT_ID() as messageID')->messageID;
84         //Liaison du message et du ticket
85         DB::insert('insert into ticket_message (id_message, id_ticket) values (?, ?)', [$idMessage, $ticketId]);
86         //Fin Transaction
87         DB::commit();
88         return $idMessage;
89     }
90     catch(Exception $exception) {
91         throw $exception;
92         return -1;
93     }
94 }
```

Cette fonction storemsg du Model sollicité par le controller effectue l'insertion du message en base de données. Elle effectue cette tâche en deux temps ce qui justifie le fait que j'ai effectué ses deux étapes dans une transaction.

- La première étape effectue une insertion dans la table message en remplissant les colonnes de celle-ci avec les éléments passés en arguments à la fonction. Ces valeurs sont passées à la requête après sa préparation par un tableau comme le permet la classe DB :: fournie par Laravel. Cette pratique permet de diminuer le risque d'attaques par injection SQL.
- La deuxième étape de cette transaction

4 Réalisation

4.1 La base de données

Le schéma de ma base est disponible dans l'annexe de ce dossier

La base de données sera gérée avec le SGBDR* open source (*Système de Gestion de base de Données relationnelle) Maria DB. Aussi, pour avoir la possibilité d'avoir un accès graphique sur ma base de données et lancer mes différents scripts j'ai utilisé le logiciel Dbeaver. J'ai scindé la création de ma base de données en deux étapes. En Me référant au Modèle Physique de Données j'ai écrit un premier script pour la création des différentes tables sans les contraintes de clefs primaires et clefs étrangères. Cela m'a permis de ne pas avoir à faire aux problèmes de l'ordre dans lequel je créais mes tables. Par conséquent j'ai écrit un deuxième script dédié à l'ajout des contraintes de clefs primaires et de clefs étrangères ainsi qu'à l'ajout des colonnes nécessaires dans la table users créée par Fortify comme nous l'avons déjà vu précédemment.

4.1.1 Création des tables

Ici l'extrait de mon script de création des tables montrant les lignes SQL qui créent la table type_panne avec ses des colonnes (la clef primaire l'entier :id et la varchar : label).

```
18 CREATE TABLE if not exists type_panne (
19     id INT PRIMARY KEY AUTO_INCREMENT,
20     label VARCHAR(20)
21 );
```

Ce type d'instructions seront adaptées pour la création des différentes tables de la base nécessaire au projet.

4.1.2 Modification des tables

Dans l'exemple suivant extrait de mon script de modification des tables de la base créées dans un premier temps. On peut voir dans l'extrait suivant, l'ajout de la contrainte de la clef étrangère id_auteur qui fait référence à la colonne id de la table users.

```
18 ALTER TABLE ticket
19
20 FOREIGN KEY (id_auteur) REFERENCES users(id);
```

- Ajout des colonnes nécessaire à la table users créé par Fortify.

```
4 ALTER TABLE users
5 ADD COLUMN prenom varchar(10),
6 - ADD COLUMN tel varchar(10);
7 ADD COLUMN role varchar(10);
```

4.2 Partie front-end

Dossier projet

4.2.1 Le maquettage

J'ai effectué le maquettage des différentes pages après en avoir établie la liste selon les demandes nécessaires à l'application attendue par le cahier des charges. J'ai effectué ce travail de structuration graphique et sémantique grâce au logiciel Canvas. J'ai pris le parti de faire le maquettage en mobile first car cela me paraissait plus facile de passer de l'affichage mobile à celui du format desktop que l'inverse. D'autant plus que je tenais que l'application soit le plus possible responsive. Cela ne changeait pas l'organisation sémantique de mes pages



4.2.2 Balisage des pages

J'ai effectué le balisage en HTML5 en me référant aux maquettes effectuées précédemment et les différents éléments écrit en html seront placés pour beaucoup dans des composants blade.php qui seront inclus dans des templates où extension de Template. J'ai inclus ses composants par la suite selon l'utilisation des pages.

Ici le détail du composant blade du tableau affichant tous les tickets de l'utilisateur :

Le thead contient une ligne dans laquelle apparaissent les titres des colonnes. Dans le tbody on peut voir l'instruction @forelse qui est l'équivalent du du blade de la boucle foreach de PHP. Dans cet exemple la variable \$tickets(tous les tickets ressortis de la requête SQL menée par le modèle en base de données) et passée à la vue par le contrôleur avec la fonction view() Cette variable \$tickets est décomposée en variable \$ticket (correspondant à chacun des éléments de \$tickets)chaque élément sera affiché sur une ligne du tableau et chacun de ses attributs dans une colonne dédié.

```

1 | @extends('templates.home')
2 |
3 | @section('content')
4 |
5 | <h3>Tous vos tickets</h3>
6 | <table>
7 |   <thead>
8 |     <tr>
9 |       <th>ID</th>
10 |      <th>SUJET</th>
11 |      <th>STATUS</th>
12 |      <th>créé le</th>
13 |      <th>DETAIL</th>
14 |    </tr>
15 |  </thead>
16 |  <tbody>
17 |    @forelse ($tickets as $ticket)
18 |
19 |      <tr>
20 |        <td>{{ $ticket->id }}</td>
21 |        <td>{{ $ticket->sujet }}</td>
22 |        <td>{{ $ticket->label }}</td>
23 |        <td>{{ $ticket->created_at }}</td>
24 |        <td><button type="button">a href="{{ route('ticket_detail', ['n' => $ticket->id]) }}">Detail</a></button></td>
25 |      </tr>
26 |
27 |    @empty
28 |      <p>Aucun ticket à ce jour</p>
29 |    @endforelse
30 |  </tbody>
31 | </table>
   @endsection

```

4.2.3 Le langage de Template blade

Avec l'apprentissage du langage de Template blade et la compréhension de son fonctionnement je me suis rendu compte que mon design de nécessiterait l'extension de deux templates (home et sign), respectivement celui de la page d'accueil et celui inhérent aux pages de connexion (sgnin et signup).

. Chacun comprenant un yield('content') correspondant à la zone de la page qui s'adaptera au contenu souhaité.

Le langage blade est un langage de Template conçu pour être utilisé avec le Framework PHP Laravel que j'utilise pour cette application. Il permet de créer des vues en HTML. Il permet aussi de définir des sections interchangeables ainsi que la possibilité de définir des composants réutilisables. Il est aussi possible avec blade de définir des conditions et des boucles

Ci-dessous un exemple dans lequel le template [home.blade.php](#) est étendu. La section interchangeable de ce Template est dans cet exemple remplie avec un premier élément (detail_table) et un deuxième sous condition que l'utilisateur qui consulte la page est un rôle d'administrateur, auquel cas lui et le troisième composants (maj_status et msg_display) seront affichés dans la vue.

```
1  @extends('templates.home')
2  @section('content')
3  @include('components.detail_table')
4
5  @if ($user->role== 'admin')
6
7
8  @include('components.maj_status')
9  @else <p>vous n'avez pas les droits pour mettre à jour le status tu ticket</p>
10 @endif
11 @include('components.msg_display')
12 @endsection
```

4.2.4 Le style

J'ai écrit la stylisation de mes pages en deux temps.

En effet ayant déjà effectué l'écriture du style pages par pages avec un css dédié. J'ai choisi de me diriger vers la solution du pré processeur Sass pour intégrer mon css une fois l'apprentissage de laravel commencé car en transformant le fichier app.css de Laravel en app.scss j'ai pu bénéficier du nesting* pour intégrer le css de chaque page plus facilement).

*Exemple de nesting scss :

```
#nesting_container{
  body{
    width: 100%;
    height: 100%;
  }
  header{
    /*properties*/
  }
}
```

Sass m'a aussi permis d'utiliser sa possibilité d'effectuer des opérations mathématiques pour définir les largeurs des cases de mes tableaux (voir ci-dessous :

```
664  table tr td {
665      width: calc(100%/3);
666      height: calc(100%/3);
667      margin: 0;
```

En ajoutant un id #home au body de mon Template [home.blade.php](#) et un id #sign à celui [sign](#) (servant aux pages de connexion et d'enregistrement il me fallait mettre le css de ses pages entre les accolades des sélecteurs respectifs respectivement body #home et body #sign. Lors de l'ajout des sections dans les yield dédiés le style est appliqué aux différents composants qui se trouvent être maintenant les enfants de ces body.

4.2.4.1 Le responsive

Afin de rendre l'affichage de mes pages un maximum responsive, j'ai utilisé la média querie screen avec un break point à 640 pixels.

```
438  @media only screen and (max-width: 640px) {
439      header{
440          width: 100%;
441          height: fit-content;
442      }
```

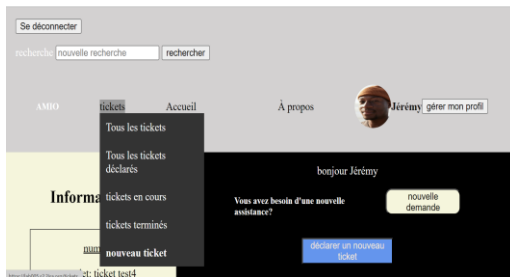
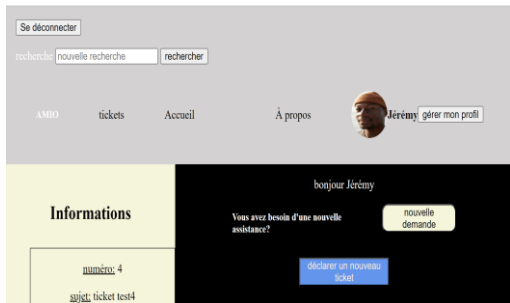
Dossier projet

```
38
39
40         <li><a href="#">À propos</a></li>
41         <li><a href="#"></a>
42             <p>{{ auth()->user()->name }}</p>
43             <button><a href="#">gérer mon profil</a></button>
44         </li>
45     </ul>
46 </nav>
47 </header>
48 @include('components.aside')
49 <main>
50     <p>bonjour {{ auth()->user()->name }}</p>
51     <span>
52         <h1>Vous avez besoin d'une nouvelle assistance?</h1><button type="button"><a href="{{route('new_ticket')}}">nouvelle demande</a></button>
53     </span>
54     @yield('content')
55 </main>
56 @include('components.footer')
57 </body>
58 </html>
59
60 </html>
61
```

Au-dessus on peut observer une partie de mon Template home avec sa barre de navigation faisant partie du header. En dessous se trouve l'inclusion de mon composant blade aside ligne 48 puis le main contenant mon yield ('content' (zone interchangeable en fonction des pages)) et enfin l'inclusion du footer lui aussi sous forme de composant blade.

4.2.4.2 Affichage de la page d'accueil :

La page d'accueil est le template 'home' sans aucun remplissage de la section 'content'(correspondant au <main> de la page).Ce template reçoit un composant head.blade.php dans lequel est faite la liaison avec le fichier de style sera étendu pour d'autres page de l'application seule la section 'content' sera rempli avec le composant nécessaire.[La page d'accueil est le template](#)



Dossier projet

Se déconnecter

recherche nouvelle recherche rechercher

Jérémy

gérer mon profil

À propos

Accueil

tickets

AMBO

bonjour Jérémy

Vous avez besoin d'une nouvelle assistance?

nouvelle demande

ticket, ticket test2

créé le: 2023-03-28 14:00:40

statut: ouvert

Details

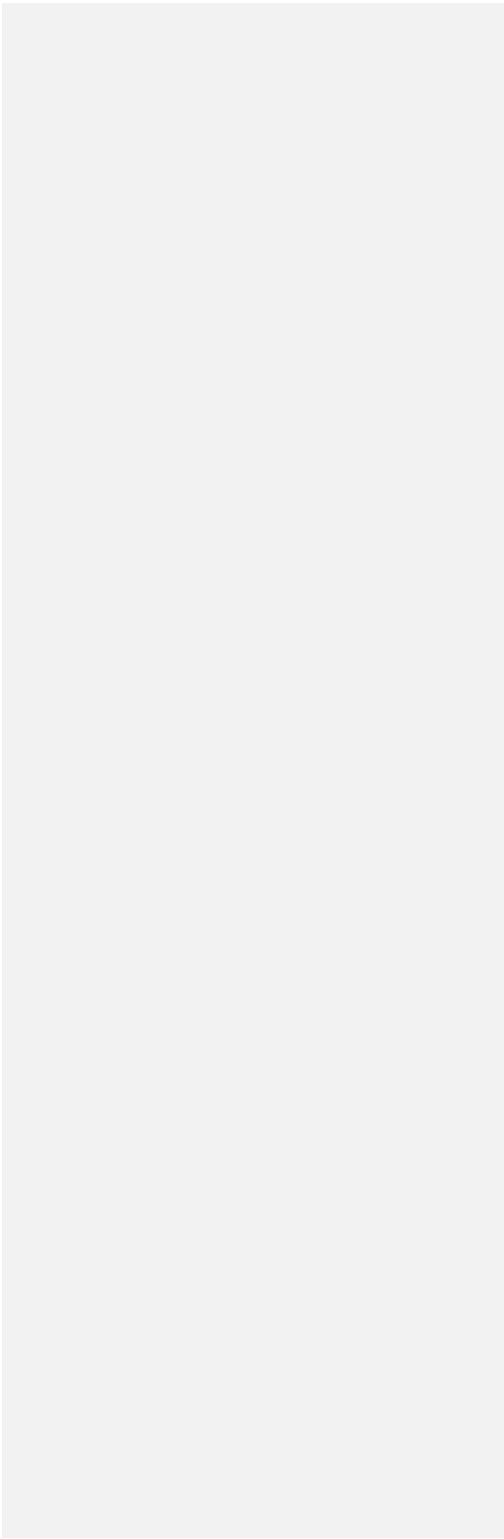
ticket, ticket test3

créé le: 2023-03-28 20:12:11

statut: ouvert

Details

ticket, ticket du lundi

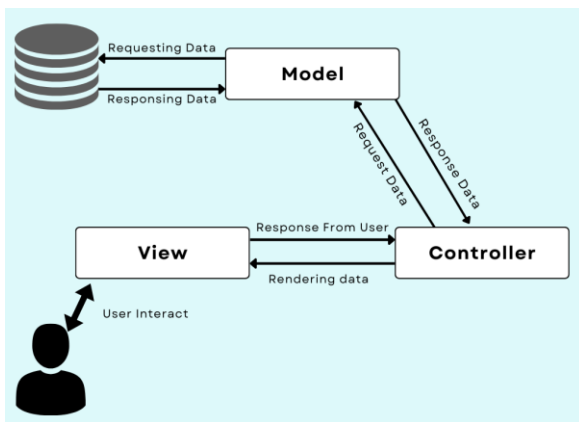


5 Partie back-end:

5.1 Architecture MVC:

- Les routes : pour l’affichage de chaque page correspond une route en méthode GET où POST qui appelle la fonction du contrôleur nommé dans cette route. La fonction retourne la vue correspondante. Si cette vue nécessite l’affichage de données, le contrôleur sollicite une fonction du modèle utilisé. Cette fonction du modèle effectue alors une requête en base de données et retourne le résultat obtenu. Ces données feront l’objet d’un traitement avant d’être retournées où le seront directement.

5.1.1 Schéma du MVC



5.1.2 Le concept du MVC

- Modèle (Model) : le modèle est chargé des interactions avec la base de données. Les requêtes SQL y sont écrites dans des fonctions qui retournent le résultat de leur requête. Le modèle est représenté par une classe.
- Vue(view) : La vue est responsable de la représentation graphique des données récupérées en base de données et transmises par le contrôleur.
- Contrôleur (Controller). Le contrôleur est responsable de la gestion des interactions entre le modèle et la vue. Il traite les requêtes de l'utilisateur. Lui aussi est représenté par une classe. Il sollicite le modèle en lui fournissant des paramètres pour mener des requêtes en base de données et lui retourne les informations qu'il lui faut pour les transmettre à la vue qui l'a elle-même sollicité. Si besoin c'est lui qui est chargé de les manipuler.
- La Vue(View) : Elle est en charge de l’affichage des données qui ont été récupérées en base de données par le modèle et transmises par le contrôleur.

5.1.3 Exemple pour la récupération de tous les tickets de l'utilisateur

En suivant le processus d'appel dans le MVC :

La route : Dans l'exemple suivant on peut voir la route '/tickets' en GET qui fait appel à la fonction 'allTickets' du ticketsController.

```
//route vers la page de tous les tickets.
Route::get('/tickets', [ticketsController::class, 'allTickets'])->name('all_tickets');
```

1. Le contrôleur :

Voici ici le détail de la fonction du contrôleur appelé dans la route au-dessus :

```
95 public function allTickets()
96 {
97
98     $ticketModel = new TicketModel();
99     $tickets = $ticketModel->getAllTickets();
100
101     return view('all', ['tickets' => $tickets]);
102 }
```

Cette fonction crée une nouvelle instance de la classe ticket Model dans laquelle existe une fonction qui sera appelée en tant qu'attribut de l'instance \$ticketModel.

2. Le modèle effectue la requête en base de données :

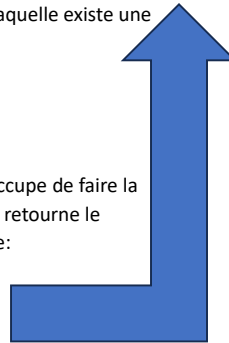


Le modèle : La fonction du modèle getAllTickets est détaillée ci-dessous s'occupe de faire la requête SQL à la base de données lors de son appel par le contrôleur et lui retourne le résultat afin qu'il la transmette à la vue qui en a besoin pour son affichage:

```
public function getAllTickets()
{
    return DB::select('select * from ticket;');
}
```

La requête précédente retourne tous les éléments de la table ticket. On remarque ici comme dans mes autres requêtes visible dans l'annexe que j'ai utilisé la classe DB :: mise à disposition par laravel, en effet elle sécurise les requêtes en base de données en offrant la possibilité de préparation des requêtes avec des valeurs de paramètre. Cela peut permettre de se prémunir des attaques par injections SQL. Ci-dessous on observe une requête qui utilise elle aussi la classe DB :: mise à disposition par laravel. J'utilise la possibilité de passer des valeurs par l'intermédiaire de paramètres qui est une pratique permettant de se prémunir des attaques par injections SQL.

En suivant une requête effectuée par la fonction prend un paramètre qui se trouve être l'identifiant de l'utilisateur dont on recherche les tickets fermés



Dossier projet

```
139 public function close_tickets($userid)
140 {
141     return DB::select('select *from ticket
142     join status on ticket.id_status=status.id
143     join users on ticket.id_auteur= users.id
144     WHERE status.id = ? AND users.id = ?', [3,$userid]);
145 }
```

3. L'affichage après la première requête :

Ici on peut constater que l'utilisateur 30.05.23 qui compte trois tickets lui appartenant : tous ont le statu 'ouvert' Ce statu est d'ailleurs donné par défaut au ticket lors de leur création.

bonjour user name 30.05.23

Vous avez besoin d'une nouvelle assistance?

Tous vos tickets

ID	SUJET	STATUS	CRÉE LE	DETAIL
33	exemple ticket1	ouvert	2023-06-11 17:44:34	<input type="button" value="Detail"/>
34	exemple ticket 2	ouvert	2023-06-11 17:44:58	<input type="button" value="Detail"/>
35	exemple ticket3	ouvert	2023-06-11 17:45:22	<input type="button" value="Detail"/>

Dans le cas où aucun ticket n'est présent un message l'indiquant est affiché pour prévenir l'utilisateur grâce à une condition écrite avec blade entre les lignes 26 à 28 ici.

```
19 <tr>
20     <td>{{ $ticket->id }}</td>
21     <td>{{ $ticket->sujet }}</td>
22     <td>{{ $ticket->label }}</td>
23     <td>{{ $ticket->created_dat }}</td>
24     <td><button type="button"><a href="{{ route('ticket_detail', ['n' => $ticket->id]) }}">
25 </tr>
26 @empty
27     <p>Aucun ticket à ce jour</p>
28 @endforelse
```

Dossier projet

Dossier projet

1. Route paramétrée :

Les routes peuvent prendre un paramètre afin de devenir plus spécifique. Ce paramètre sera l'objet d'un traitement par le contrôleur

```
37 // route pour afficher le détail d'un ticket:
38 Route::get('/ticket/{n}', [ticketsController::class, 'detailTicket'])->name('ticket_detail');
```

- Ici la route prend un paramètre appelé n. Il sera défini et traité dans le contrôleur et servira aussi dans le modèle lors de la requête pour conditionner le son résultat.

```
38 ket/{n}', [ticketsController::class, 'detailTicket'])->name('ticket_detail');
```

- Le contrôleur : Il utilisera la fonction getone_ticket qui prendra n comme paramètre. Il passera les éléments récupérés à la vue grâce à une variable donnée comme valeur dans un tableau associatif lui-même utilisé comme paramètre dans la fonction view() qui prend deux paramètres, en premier le nom de la vue à retourner et en deuxième les variables nécessaires à la vue. Ces variables sont placées dans un tableau associatif comme dans l'exemple ci-dessous qui montre le retour de la vue 'detail' accompagnée des variables nécessaires à son bon affichage.

```
public function detailTicket($n)
{
    $ticketModel = new ticketModel();
    $ticket = $ticketModel->getone_ticket($n);
    $tickets = $ticketModel->getallTickets();
    // récupère tous les messages du ticket et les donner à la vue.
    $msg = $ticketModel->searchmsg($n);
    //Récupération des droits de l'utilisateur pour l'affichage de la mise à jour du status du ticket
    $user = auth()->user();
    $NticketController = new TicketModel();
    $right = $NticketController->CurrentUser($user);

    return view('detail', ['ticket' => $ticket, 'tickets' => $tickets, 'msg'=>$msg, 'user'=>$user]);}

```

- Le modèle : Il récupère tous les éléments de la table ticket et filtre le résultat en fonction de n. La fonction retournera donc seulement les éléments correspondants à la condition du WHERE. La variable n'est pas passée directement dans la requête mais est remplacée par un point d'interrogation et est donnée dans un tableau en deuxième paramètre. Cela sécurise la requête

```
function getone_ticket($n)
{
    return DB::selectOne('select * from ticket where ID =?;', [$n]);
}
```

Dossier projet

3. Les routes en post : elles seront utilisées lors de l'envoi d'une requête post sur le chemin correspondant à celle de la route. Contrairement aux routes en GET elles ne sont pas tant présentes pour délivrer une page mais plus pour récupérer de données d'un formulaire soumis sur 'url qu'elle indique.

Dans l'exemple ci-dessous la route en post sur le chemin *new* appellera la méthode *store* du contrôleur des tickets.

```
Route::post('/new', [ticketsController::class, 'store'])->name('avion_create')->middleware('auth');  
//route/new en get pour fournir le formulaire de création de ticket:  
Route::get('/new', [ticketsController::class, 'form'])->name('new_ticket')->middleware('auth');
```

5.2 Éléments de sécurité

5.2.1 Requêtes https

Même si pour ce premier jalon nous n'avons pas eu à gérer la configuration du serveur. Je note tout de même ici que le serveur hébergeant mon application exécute les requêtes en utilisant le protocole HTTPS.

5.2.2 Jeton CSRF

Dans mes formulaires j'ai placé un jeton CSRF (Cross Site Request Forgery) ce jeton est une mesure de sécurité qui permet de s'assurer que les requêtes soumises à un serveur web sont authentiques et ne sont pas falsifiées par un tiers malveillant. Cela sécurise l'application et lui permet de se prémunir contre les attaques type CSRF.

Dans l'exemple suivant on peut remarquer que j'ai placé cet élément de sécurité sur la ligne n°7 :

```

4      <h4>Déclarer un nouveau ticket</h4>
5      
6      <form action="" method="post">
7          @csrf
8          <label for="sujet">sujet du ticket</label>
9          <input type="text" name="sujet" id="sujet" placeholder="sujet du ticket">
10
11         <label for="coments">commentaires</label>
12         <input type="text" placeholder="commentaires" id="coments">
13         <button type="submit">valider</button>
14
15     </form>

```

5.2.3 Classe DB ::

Comme je l'ai expliqué dans un précédent chapitre sur mes requêtes SQL dans mon Model. J'ai utilisé la classe DB :: mise à disposition par laravel pour interroger la base de données. L'utilisation de cette dernière permet de diminuer l'exposition aux attaques par injections SQL grâce à la possibilité qu'offre cette classe de préparer la requête en lui passant des valeurs par paramètres

```

function getone_ticket($n)
{
    return DB::selectOne('select * from ticket where ID =?;', [$n]);
}

```

5.2.4 Version à jour du Framework utilisé :

Pour le développement de mon application j'ai utilisé une version à jour du Framework la 10.10.1. Cela diminue le risque de failles de sécurité.

5.2.5 Middleware* :

Certaines de mes routes sont équipées du morceau de code pour sécuriser leur accessibilité

```

52 Route::get('/admin', [ticketsController::class, 'userRight'])->name('admin')->middleware('auth');

```

Dossier projet

Un ***middleware** est une couche intermédiaire entre la requête http et la réponse http d'une application Laravel. Dans le cas que j'expose dans la capture ci-dessus. Le middleware "auth" créé par fortify dans le fichier dédié est utilisé afin de vérifier si l'utilisateur qui demande la route en question est bien connecté sur l'application avant de pouvoir accéder à cette route. Dans le cas contraire Laravel redirigera vers la page de connexion.

Ci-dessous le détail de ce middleware construit par Fortify dans le fichier :

/home/lab005/s2/www/fil-rouge/app/Http/Middleware/RedirectIfAuthenticated.php:

```
13  /**
14   * Handle an incoming request.
15   *
16   * @param \Closure(\Illuminate\Http\Request): (\Symfony\Component\HttpFoundation\Response) $next
17   */
18  0 references | 0 overrides
19  public function handle(Request $request, Closure $next, string ...$guards): Response
20  {
21      $guards = empty($guards) ? [null] : $guards;
22
23      foreach ($guards as $guard) {
24          if (Auth::guard($guard)->check()) {
25              return redirect(RouteServiceProvider::HOME);
26          }
27      }
28      return $next($request);
```

Je n'ai personnellement créé aucun middleware avec la commande `php artisan make:middleware` pour ce premier jalon. J'y ai simplement porté une attention à titre informatif lors de mes recherches sur les éléments de sécurités et après qu'il nous ait été conseillé d'écrire ses instructions sur certaines de nos routes.

6 Conclusion

6.1 Bilan du projet

6.1.1 Problématiques rencontrées :

Lors du développement de cette première application, j'ai logiquement été amené à rencontrer quelques problématiques telles que :

- La compréhension de la transmission des variables entre les différentes couches du MVC.
- La syntaxe des média quéries en .scss. En effet , je souhaitais que mon application soit autant que faire se peut responsive seulement je ne connaissais la syntaxe uniquement pour les média quéries en .css mais pas en .scss.

6.1.2 Solutions opposées :

Toutefois malgré ces désagréments liés comme souvent dans les périodes d'apprentissage, j'ai malgré tout su y opposer des stratégies de solutionnement.

- Concernant la transmission des variables au travers des différentes couches du MVC afin de bien comprendre et de renforcer ces concept dans mon esprit, je me suis efforcé à trouver des fonctionnalités qui nécessites d'utiliser la gymnastique du MVC comme par exemple l'ajout de la barre de recherche dans la barre de navigation ou la possibilité de mettre à jour.

Dossier projet

- le statu d'un ticket n à partir d'un formulaire utilisable seulement par les utilisateur ayant les droits nécessaires.
- J'ai aussi sollicité l'appuis de l'équipe pédagogique pour m'aider à assimiler ses nouveaux concepts
- Au sujet de la syntaxe de média quéries en .scss, il m'a fallu mener des recherches dans la documentation de Sass : <https://sass-lang.com/documentation/>
- Concernant le documentations, celle de Laravel aura été une aide pour ce projet et la compréhension de sa bonne utilisation. <https://laravel.com/docs/10.x/readme>
- Dans le futur en continuant de progresser et d'apprendre il me viendra à penser que j'aurais pu et du ajouter telle et telle fonctionnalités. A ce moment-là je reprendrai l'amélioration grâce entre autre à la possibilité d'accès au projet que permet GitHub et à mes nouvelles connaissances et capacités.

6.2 Apports personnels et professionnels

A travers ce projet j'ai pu me rendre compte que toute la période que j'ai passé par le auparavant à travailler le front-end en autodidacte à porté ses fruits et m'a permis d'aborder ce projet avec un peu plus de sérénité. J'ai aussi été très content de pouvoir enfin interagir avec une base de données et d'apprendre l'utilisation d'un framework . Mener ce projet m'a permis de constater que malgré tout je suis capable de réaliser et comprendre des choses. Reste à moi de m'en contenter.

6.3 Perspectives d'améliorations

En vérifiant les déférentes contraintes demandées, j'ai pu m'apercevoir trop tardivement que j'ai oublié deux choses :

- De placer une colonne "date limite » » à ma table des tickets. Cela peut être effectué avec un ALTER TABLE.
- De créer la page tableau de bord. J'imagine cette page comme affichant le nombre de ticket en BDD pour chaque statut d'avancement des tickets.

Pour la faire j'utiliserais ma fonction 'UserRight pour afficher son contenu uniquement pour les rôles 'admin'. Je peux créerais des fonctions RES semblables à celles que j'ai déjà pour chaque statut de ticket mais sans la contrainte WHERE cette fois-ci afin de récupérer tous les tickets de chaque statut sans tenir compte de l'utilisateur. Je les aurais compté à l'aide de la méthode count() sur les tickets retournés dans un tableau par la fonction du modèle effectuant la requête correspondante pour indiquer aux admins l'état le nombre d' assistances des clients par statut.

_ Je pourrais aussi indiquer aux admins le temps moyen aloué pour le traitements des tickets en utilisant une boucle foreach sur la liste de tous les tickets retourné par le modèle puis utiliser la méthode avg() sur les différences entre la date limite et la date de création des tickets .

_ Je pourrais faire une chose semblable pour le temps moyen passé à la résolution des tickets par les dépanneurs en faisant la différence entre la date de création et la date fin du ticket cette fois-ci. La comparaison entre ses deux moyennes celle du temps aloué et celle du temps passé peut être effectué

Dossier projet

afin d'indiquer aux admin les performances du service client d'AMIO. Je pourrais combler ses oublis et ultérieurement sur une nouvelle branche du projet et effectuer un 'git merge' lorsque tout sera prêt et entendu avec le client et validé.

_J'ai aussi effectué quelques recherches pour que l'utilisateur puisse mettre à jour sa photo de profil via une page dédiée à la gestion de profil. Cette fonctionnalité fera l'objet :

- d'une nouvelle fonction contenant un update de la table users se trouvant dans le modèle(des users). Cela après avoir
- rajouté une colonne dédiée dans la table users,
- placé un input type 'file' dans un formulaire,
- récupérer cette entrée avec le contrôleur grâce à l'objet request en l'associant à l'utilisateur actuellement authentifié, le contrôleur rangera la nouvelle photo dans un dossier dédié et définira le chemin (pour l'utiliser dans la balise).
- Ci-dessous, le stockage dans une variable correspondant au chemin de l'image soumise dans le formulaire si la requête http contient un fichier provenant de l'input nommé 'photo'

```
if (request()->hasFile('photo')){  
    $path = request()->file('photo')->store('profile', 'public');  
} else {  
    $path = 'avatar.jpg';  
}
```

J'ai effectué cette recherche sur stackoverflow en écrivant dans la barre de recherche : "*how to update a profile picture in laravel application*". En cherchant parmi les résultats j'y ai trouvé une proposition de marche à suivre. Je l'ai analysé et grâce à mes nouvelles connaissances j'ai réussi à comprendre les actions qui y sont menées pour réaliser cette feature. Elle aussi pourra faire l'objet d'une nouvelle branche de mon repo Git ultérieurement.

7 Annexes

7.1.1 Quelques requêtes SQL

- La requête select suivante récupère les différentes colonnes de la table ticket nécessaire à l'affichage de ces derniers. La fonction à laquelle elle appartient se fait passer un paramètre et le paramètre est la variable contenant la valeur de ce que l'utilisateur a soumis dans la barre de recherche. Cette valeur est utilisée dans ma requête afin d'être comparée avec un like. L'éventualité d'une saisie en lettres capitale est envisagée avec la méthode UPPER(). J'ai aussi effectué une gestion d'exception. Si la requête venait à échouer la fonction retournera 'false' autrement elle retournera true.

```

1 reference | 0 overrides
47 function Ticket_search( $search)
48 {
49 //La fonction effectue une requête SQL qui sélectionne dans la table ticket les différentes colonnes nécessaires de la table ticket et qui compare la valeur la valeur soumise dans l'input, je joins la table sta
50 try {
51     return DB::select('select ticket.id,ticket.sujet,ticket.created_at, status.label from ticket
52     join status on ticket.id_status= status.id
53     where ticket.id = :search or sujet like UPPER(:searchLike) or sujet like :searchLike2 order by ticket.id;', ['search'=>$search, 'searchLike'=>'%. $search. %', 'searchLike2' => '% . $search . %']);
54 }
55 catch (Exception $e) {
56     return false;
57 }
58 return true;
59
60 }

```

- Requête update : Cette requête est écrite dans le contexte de la mise à jour du statut d'un ticket

```

113 //Fonction qui met à jour le status du ticket en base de données:
1 reference | 0 overrides
114 public function ticket_update($idTicket, $nstatus)
115 {
116     return DB::update("update ticket set id_status = ? where id = ?", [$nstatus, $idTicket]);
117 }
2 references | 0 overrides
118 function get_ticket_status($n)
119 {
120     return DB::selectOne('select label
121     from status join ticket on ticket.id_status=status.id
122     where ticket.id=?;', [$n]);
123 }

```

- Exemple de requête dans une transaction :

Dans l'exemple suivant puisque j'avais plusieurs requêtes pour l'envoi d'un message dans la conversation d'un ticket, j'ai effectué ses deux requêtes dans une transaction.

- Exemple d'insert

Dossier projet

```
75 function storemsg($ticketId, $contentMessage, $auteurId) {
76     try(
77         //Debut Transaction
78         DB::beginTransaction();
79         // Insertion du message
80         DB::insert('insert into messages (id_auteur, content, created_at) values (?, ?, ?)', [$auteurId, $contentMessage, Carbon::now()]);
81         $idMessage = DB::selectone('select LAST_INSERT_ID() as messageID')->messageID;
82         //Liaison du message et du ticket
83         DB::insert('insert into ticket_message (id_message, id_ticket) values (?, ?)', [$idMessage, $ticketId]);
84         //Fin Transaction
85         DB::commit();
86         return $idMessage;
87     )
88     catch(Exception $exception) {
89         throw $exception;
90         return -1;
91     }
92 }
```

Dossier projet

7.2 Schéma de la base de données

