Modèle de copie :
Créer une application web

GDWFSCAWEXAIII1A

Ceci est un modèle de copie. N'oubliez pas de renseigner vos prénom/nom, ainsi que le nom et le lien vers le projet.

Vous pouvez bien sûr agrandir les cadres pour répondre aux questions sur la description du projet si nécessaire.

Prénom : Rémi
Nom : Faure

ATTENTION ! PENSEZ À RENSEIGNER VOS NOM ET PRÉNOM DANS LE TITRE DE VOS FICHIERS / PROJETS !

Nom du projet : KGB-STUDI
Lien Github du projet : https://github.com/rf33350/kgb
URL du site : https://kgb-studi-faure.fr/

Description du projet

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions. Dans cette rubrique, le jury cherche à voir comment vous procédez : comment vous organisez votre travail, comment vous réalisez concrètement la tâche ou l'opération pas à pas.
   Utiliser un langage professionnel. Employez-le « je », car vous parlez en votre nom. Vous pouvez écrire au temps présent.

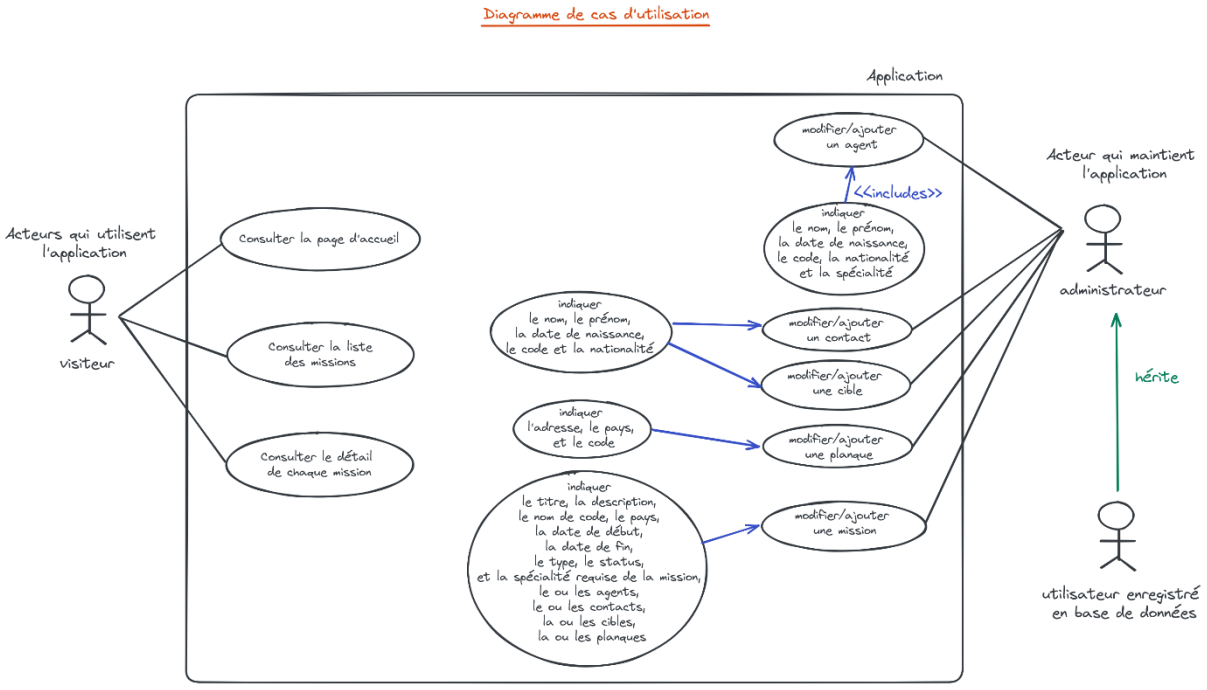
J'ai réalisé un site de gestion des données du KGB.
Afin de réaliser cette application, j'ai tout d'abord conçu la base de données.

1. Conception et création de la base de données :

Pour concevoir la base de données, j'ai réalisé un diagramme de cas d'utilisation, un diagramme de séquences. J'ai également utilisé la méthode Merise pour créer les modèles conceptuel, logique et physique en vue de la création de la base de données elle-même.

1.1 Diagramme de cas d'utilisation :

J'ai tout d'abord réalisé le diagramme de cas d'utilisation de l'application.



Sur ce diagramme j'ai représenté un visiteur qui consulte le site de manière anonyme. Il aura donc la possibilité de :
- Consulter la page d'accueil du site
- Consulter la page de la liste des missions
- Consulter la page de chaque mission

J'ai également représenté l'administrateur, qui a en charge le maintien de l'application. Il aura la possibilité de gérer les agents. Les créer, les modifier, les supprimer. Il aura accès à la page de listing des agents ainsi que la page de détails de chaque agent. De la même manière il pourra gérer les contacts, les cibles, les planques et les missions.

A partir de ce diagramme j'ai créé des diagrammes de séquences.

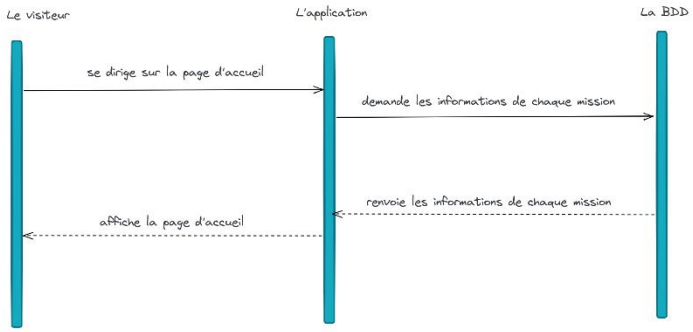
1.2 Diagramme de séquences :

Je me suis conformé aux détails donnés dans l'énoncé de l'examen pour réaliser le diagramme de séquences pour les visiteurs du site :

Diagramme de séquences visiteurs

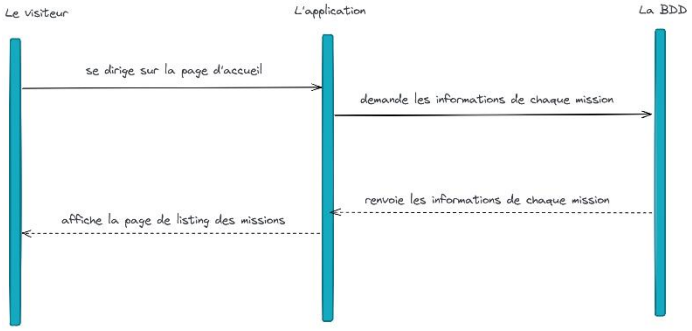
Consulter la page d'accueil

1- Le client se dirige sur la page d'accueil
2- L'application demande les informations de chaque mission à la BDD
3- La BDD renvoie les informations de chaque mission
4- L'application affiche la page d'accueil avec les informations de chaque mission



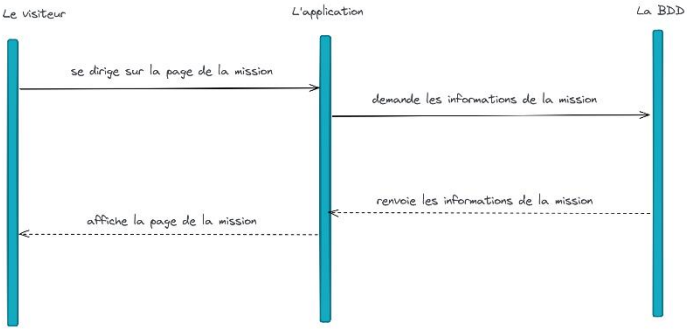
Consulter la page des missions

1- Le client se dirige sur la page d'accueil
2- L'application demande les informations de chaque mission à la BDD
3- La BDD renvoie les informations de chaque mission
4- L'application affiche la page de listing des missions



Consulter la page de chaque mission

1- Le client se dirige sur la page de la mission sélectionnée
2- L'application demande les informations de la mission à la BDD
3- La BDD renvoie les informations de la mission
4- L'application affiche la page de la mission



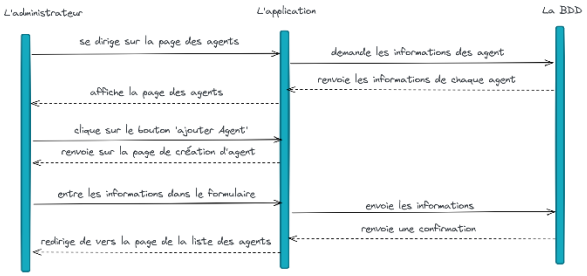
Ce diagramme décrit les opérations entre le visiteur, l'application et la base de données pour chaque interaction utilisateur.

J'ai également réalisé le diagramme de séquences pour les administrateurs du site :

Diagramme de séquences administrateurs

Ajouter un agent

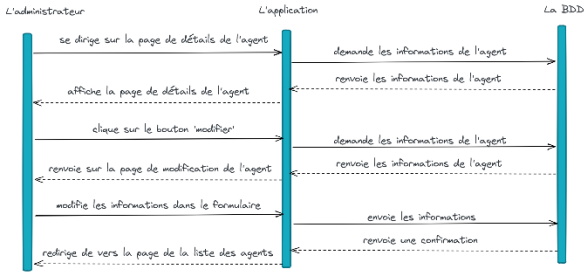
1- L'administrateur se dirige sur la page des agents
2- L'application demande les informations des agents à la BDD
3- La BDD renvoie les informations des agents
4- L'application affiche la page des agents
5- L'administrateur clique sur le bouton 'ajouter Agent'
6- L'application renvoie l'administrateur sur la page de création d'agent
7- L'administrateur entre les informations dans le formulaire
8- L'application envoie les informations à la BDD
9- La BDD renvoie une confirmation
10- L'application redirige l'administrateur de vers la page de la liste des agents



NB: Cette séquence est identique pour les opérations de création:
- des missions
- des contacts
- des cibles
- des planques

Modifier un agent

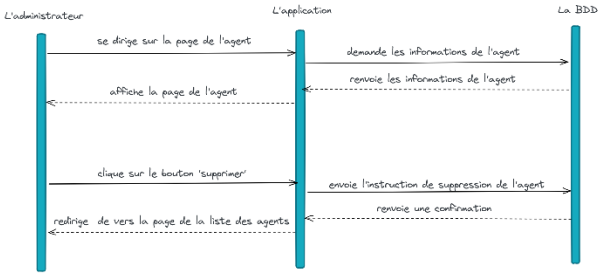
1- L'administrateur se dirige sur la page de détails de l'agent
2- L'application demande les informations de l'agent à la BDD
3- La BDD renvoie les informations de l'agent
4- L'application affiche la page de détails de l'agent
5- L'administrateur clique sur le bouton 'modifier'
6- L'application demande les informations de l'agent à la BDD
9- La BDD renvoie les informations de l'agent
6- L'application renvoie l'administrateur sur la page de modification de l'agent
7- L'administrateur modifie les informations dans le formulaire
8- L'application envoie les informations à la BDD
9- La BDD renvoie une confirmation
10- L'application redirige l'administrateur de vers la page de la liste des agents



NB: Cette séquence est identique pour les opérations de création:
- des missions
- des contacts
- des cibles
- des planques

Supprimer un agent

1- L'administrateur se dirige sur la page de l'agent
2- L'application demande les informations de l'agent à la BDD
3- La BDD renvoie les informations de l'agent
4- L'application affiche la page de l'agent
5- L'administrateur clique sur le bouton 'supprimer'
6- L'application envoie l'instruction de suppression de l'agent à la BDD
7- La BDD renvoie une confirmation
8- L'application redirige l'administrateur de vers la page de la liste des agents



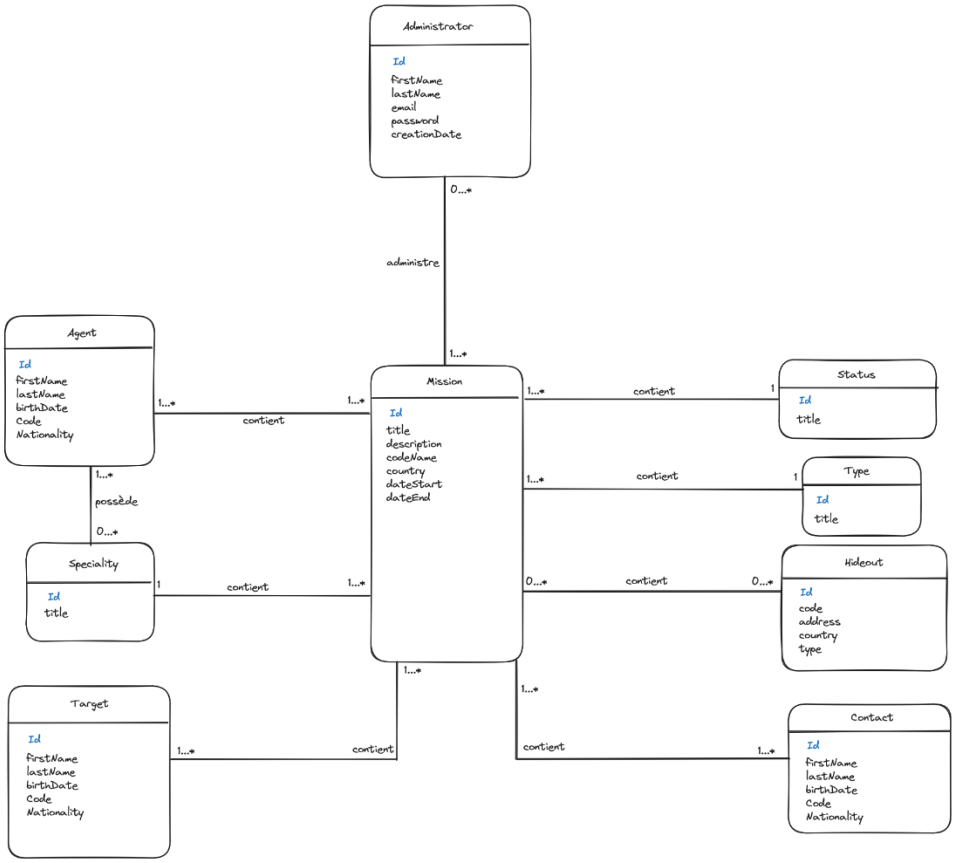
NB: Cette séquence est identique pour les opérations de création:
- des missions
- des contacts
- des cibles
- des planques

Ce diagramme décrit les opérations entre l'administrateur, l'application et la base de données pour la création, la modification ou la suppression d'un agent, d'un contact, d'une cible, d'une planque ou d'une mission.
A partir de ces diagrammes j'ai conçu la base de données de l'application.

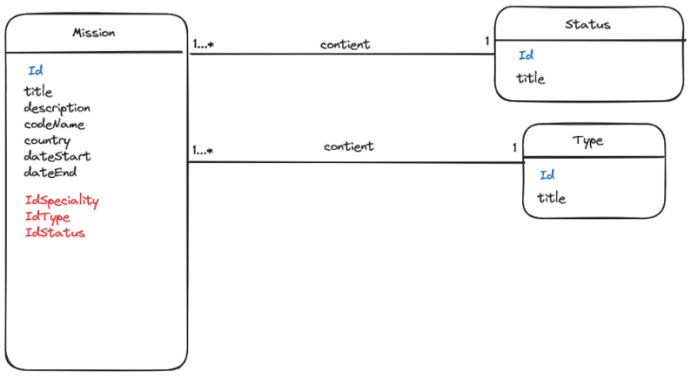
1.3 Conception de la base de données :

J'ai conçu la base de données de l'application en utilisant le logiciel en ligne Excalidraw. J'ai utilisé la méthode Merise pour créer le diagramme de classes. La méthode consiste à réaliser en premier lieu le modèle conceptuel de données, le modèle logique, puis le modèle physique qui représente le plus fidèlement la base de données.

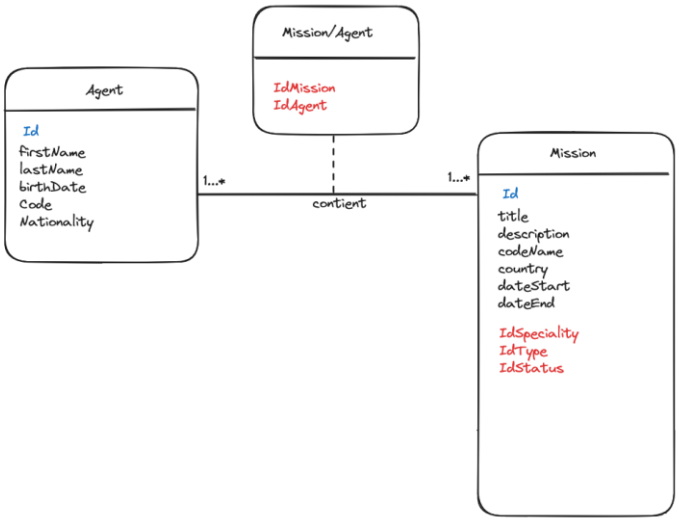
J'ai tout d'abord créé le modèle conceptuel de la base de données. J'ai défini les entités ainsi que leurs propriétés. J'ai ensuite défini les relations et les cardinalités entre les entités. Le modèle conceptuel de données établi est donc le suivant :



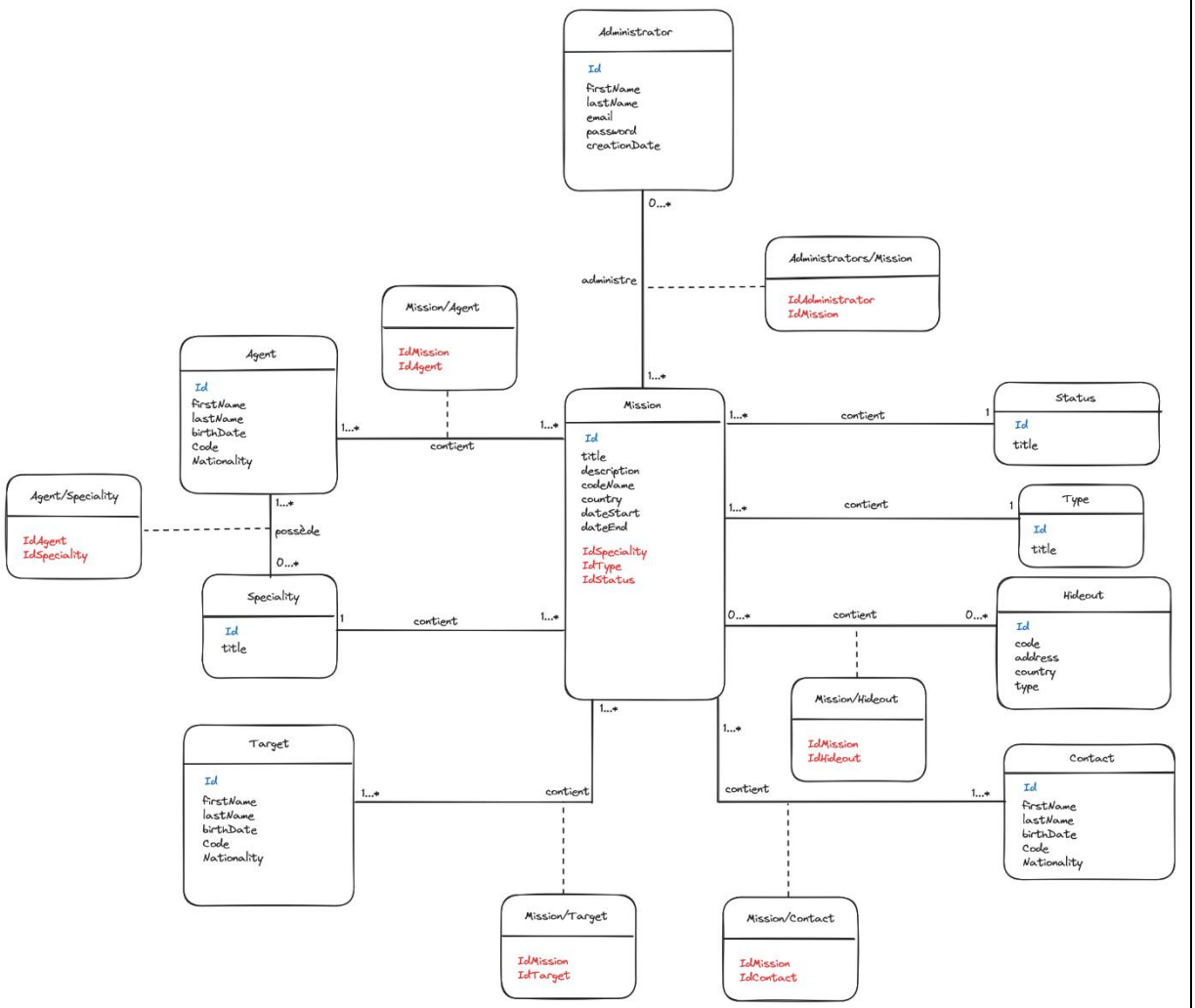
A partir du modèle conceptuel de données et de ses cardinalités, j'ai construit le modèle logique de données. A partir des différentes relations, j'ai ajusté le modèle de cette manière. Pour les relations faibles/fortes, j'ai créé une clé étrangère de l'entité de poids le plus faible dans l'entité de poids le plus fort. Exemple pour la relation Mission/Type et la relation Mission/Status :



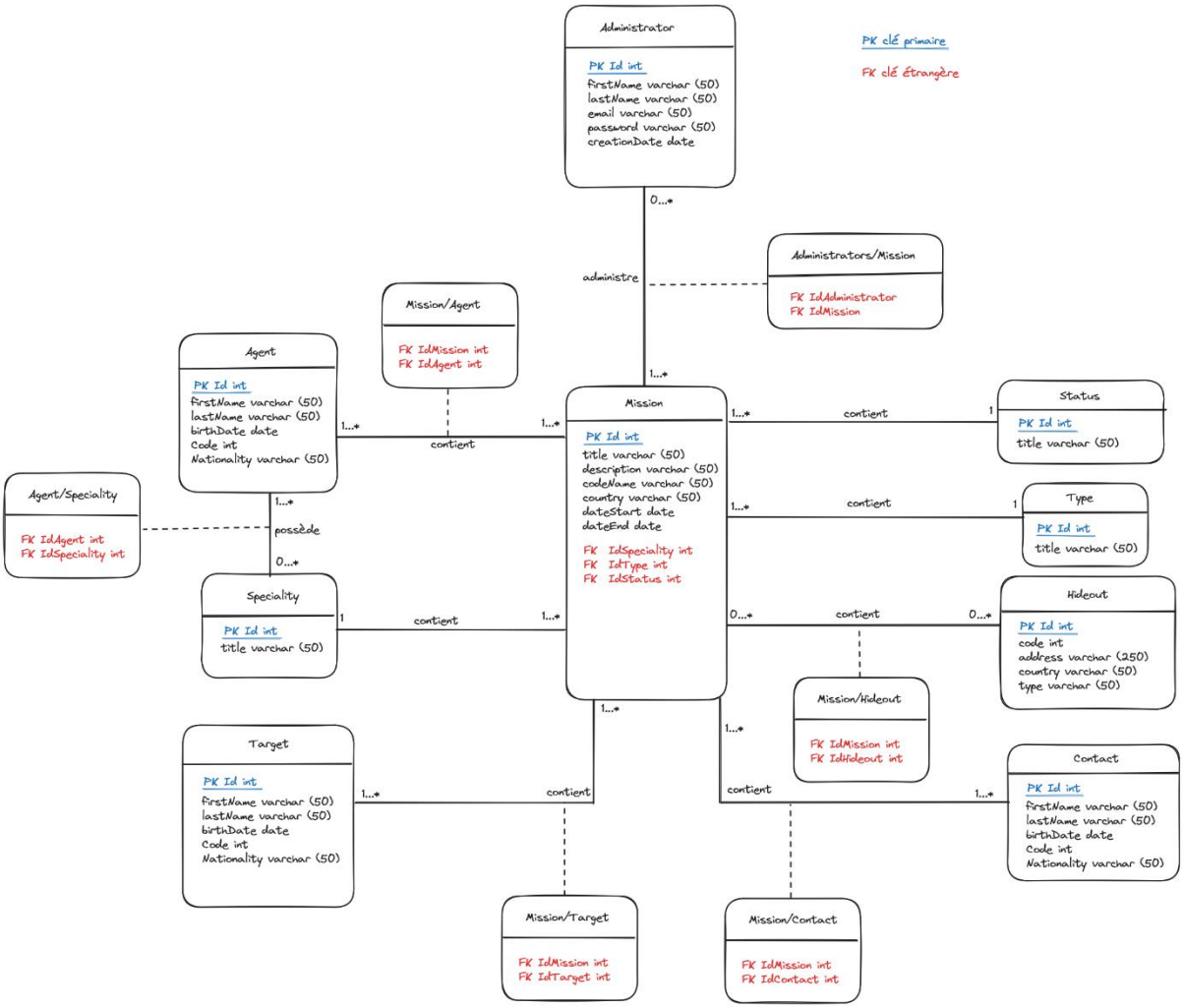
Pour les relations fortes/fortes, j'ai créé une entité supplémentaire ayant pour propriétés les références vers l'identifiant des deux entités concernées.
Exemple pour la relation Mission/Agent :



Le modèle logique correspondant à l'application :



Lors de l'élaboration du modèle physique les entités deviennent des tables. J'ai ajouté les types de données aux propriétés. J'ai également ajouté aux identifiants les attributs de clés primaires et clé étrangères.
Le modèle physique de l'application :



Le modèle physique de la base de données étant modélisé, j'ai construit la base de données à proprement parler.

1.4 Création de la base de données :

Au moyen du logiciel Beekeeper Studio, j'ai écrit les lignes de code afin de créer la base de données ainsi que les tables. J'ai également écrit les scripts afin d'injecter des données dans chaque table.

J'ai créé la base de données *kgb* :

```
/*Creation de la base de données kgb*/
CREATE DATABASE IF NOT EXISTS kgb CHARACTER SET utf8mb4 COLLATE
utf8mb4_general_ci;
```

Cette commande permet de créer la base de données, si elle n'existe pas. Je choisis également l'encodage des données en utf8mb4_general_ci, encodage largement utilisé.

J'ai ensuite créé la table *target* (cible) :

```
/*creation de la table target*/
CREATE TABLE kgb.target (
```

J'ai ensuite ajouté l'identifiant *id*, qui est donc une clé primaire. J'ajoute également la fonction *NOT NULL* pour indiquer que cette clé ne doit pas être nulle, et également *AUTO_INCREMENT* de manière à ce que cette clé s'indexe automatiquement. Cela donne:

```
    id int PRIMARY KEY NOT NULL AUTO_INCREMENT,
```

J'ajoute également 3 propriétés, *firstName*, *lastName et nationality* qui sont des chaînes de 50 caractères, ainsi que la propriété *birthDate* qui est une date et la propriété *code* qui est un nombre entier.

```
firstName varchar(50),
    lastName varchar(50),
    birthDate date,
    code int,
    nationality varchar(50)
```

Enfin je précise le moteur de stockage transactionnel engine=INNODB, ce qui permet de sécuriser les requêtes sur la base de données, de garantir l'entièreté de l'exécution de chaque requête et d'améliorer les performances.

Cela donne pour la table *target* :

```
/*creation de la table target*/
CREATE TABLE kgb.target (
    id int PRIMARY KEY NOT NULL AUTO_INCREMENT,
    firstName varchar(50),
    lastName varchar(50),
    birthDate date,
    code int,
    nationality varchar(50)
)engine=INNODB;
```

J'ai rempli la table *target* grâce à la commande SQL « INSERT »

```
/*remplissage de la table target*/
INSERT INTO kgb.target (firstName, lastName, birthDate, code,
nationality)  VALUES ('Remi', 'Faure', '1984-03-04', 1234, 'Française');
INSERT INTO kgb.target (firstName, lastName, birthDate, code,
nationality)  VALUES ('Andrew', 'Tate', '1990-07-22', 4561, 'Anglaise');
INSERT INTO kgb.target (firstName, lastName, birthDate, code,
nationality)  VALUES ('Vladimir', 'Polyty', '1972-12-07', 2133, 'Russe');
```

De la même manière j'ai créé et rempli la table *contact*, *agent* et *hideout* (planque) :

```
/*creation de la table contact*/
CREATE TABLE kgb.contact (
    id int PRIMARY KEY NOT NULL AUTO_INCREMENT,
    firstName varchar(50),
    lastName varchar(50),
    birthDate date,
    code int,
    nationality varchar(50)
)engine=INNODB;
```

```
/*remplissage de la table contact*/
INSERT INTO kgb.contact (firstName, lastName, birthDate, code,
nationality)  VALUES ('Jonathan', 'Devault', '1984-03-04', 789, 'Française');
INSERT INTO kgb.contact (firstName, lastName, birthDate, code,
nationality)  VALUES ('Johnny', 'Wilko', '1990-07-22', 45612, 'Anglaise');
INSERT INTO kgb.contact (firstName, lastName, birthDate, code,
nationality)  VALUES ('Boris', 'Zeveny', '1972-12-07', 213341, 'Russe');
```

```
/*Création de la table agent*/
CREATE TABLE kgb.agent (
    id int PRIMARY KEY NOT NULL AUTO_INCREMENT,
    firstName VARCHAR(50),
    lastName VARCHAR(50),
    birthDate DATE,
    code int,
    nationality VARCHAR(50)
)engine=INNODB;

/*remplissage de la table agent*/
INSERT INTO kgb.agent (firstName, lastName, birthDate, code, nationality)
VALUES ('John', 'Doe', '1990-05-15', 12347895, 'USA');
INSERT INTO kgb.agent (firstName, lastName, birthDate, code, nationality)
VALUES ('Alice', 'Smith', '1985-09-22', 675890, 'Canada');
INSERT INTO kgb.agent (firstName, lastName, birthDate, code, nationality)
VALUES ('Mohamed', 'Ali', '1982-12-10', 9876521, 'Egypt');
INSERT INTO kgb.agent (firstName, lastName, birthDate, Code, nationality)
VALUES ('Maria', 'Garcia', '1994-07-08', 54321, 'Espagne');
```

```
/*creation de la table hideout*/
CREATE TABLE kgb.hideout (
    id int PRIMARY KEY NOT NULL AUTO_INCREMENT,
    address varchar(250),
    country varchar(50),
    code int
)engine=INNODB;

/*remplissage de la table hideout*/
INSERT INTO kgb.hideout (address, country, code)  VALUES ('13, Rue du Temple,
33350 Tillous', 'France', 78999);
INSERT INTO kgb.hideout (address, country, code)  VALUES ('145, 1651 Funchal',
'Portugal', 78998);
INSERT INTO kgb.hideout (address, country, code)  VALUES ('Calle de San
Miguel, 15, 28004 Madrid', 'Espagne', 78997);
```

J'ai ensuite créé et rempli la table *user* pour identifier et gérer les sessions administrateur :

```
/*creation de la table user*/
CREATE TABLE kgb.user (
    id int PRIMARY KEY NOT NULL AUTO_INCREMENT,
    firstName varchar(50),
    lastName varchar(50),
    email varchar(50),
    password varchar(250),
    role varchar(50),
    creationDate date
)engine=INNODB;

/*remplissage de la table user*/
INSERT INTO kgb.user (firstName, lastName, email, password, role,
creationDate)  VALUES ('Remi', 'Admin', 'admin@mail.fr', 'root',
'$2y$10$N6ZKpom5Rgq9n5/EBb9GnOM88Aw5QsVRWU1wA4tizToHT/xyQXOU6','1984-03-04');
```

A noter que pour cette table je sors du modèle physique. C'est le cas uniquement pour cette table.

J'ai ensuite créé les tables *type*, *status* et *speciality* nécessaires à la création de la table *mission* :

```
/*creation de la table status*/
CREATE TABLE kgb.status (
    id int PRIMARY KEY NOT NULL AUTO_INCREMENT,
    title varchar(50)
)engine=INNODB;

/*remplissage de la table status*/
INSERT INTO kgb.status (title)  VALUES ('En préparation');
INSERT INTO kgb.status (title)  VALUES ('En cours');
INSERT INTO kgb.status (title)  VALUES ('Terminé');
INSERT INTO kgb.status (title)  VALUES ('Echec');

/*creation de la table speciality*/
CREATE TABLE kgb.speciality (
    id int PRIMARY KEY NOT NULL AUTO_INCREMENT,
    title varchar(50)
)engine=INNODB;

/*remplissage de la table speciality*/
INSERT INTO kgb.speciality (title)  VALUES ('Renseignement');
INSERT INTO kgb.speciality (title)  VALUES ('Cyber');
INSERT INTO kgb.speciality (title)  VALUES ('Sciences et technologies');
INSERT INTO kgb.speciality (title)  VALUES ('Langues étrangères');
INSERT INTO kgb.speciality (title)  VALUES ('Combat');

/*creation de la table type*/
CREATE TABLE kgb.type (
    id int PRIMARY KEY NOT NULL AUTO_INCREMENT,
    title varchar(50)
```

```
)engine=INNODB;

/*remplissage de la table type*/
INSERT INTO kgb.type (title)  VALUES ('Surveillance');
INSERT INTO kgb.type (title)  VALUES ('Assassinat');
INSERT INTO kgb.type (title)  VALUES ('Infiltration');
```

J'ai ensuite créé la table *mission* :

```
/*Création de la table mission*/
CREATE TABLE kgb.mission (
    id INT PRIMARY KEY AUTO_INCREMENT,
    title VARCHAR(50),
    description VARCHAR(250),
    codeName VARCHAR(50),
    country VARCHAR(50),
    startDate DATE,
    endDate DATE,
    type_id INT,
    status_id INT,
    speciality_id INT,
    FOREIGN KEY (type_id) REFERENCES kgb.type(id),
    FOREIGN KEY (status_id) REFERENCES kgb.status(id),
    FOREIGN KEY (speciality_id) REFERENCES kgb.speciality(id)
)engine=INNODB;
```

J'ai créé dans cette table 3 clés étrangères type_id, status_id, et speciality_id, fidèlement au modèle physique que j'ai créé précédemment par les commandes *FOREIGN KEY* qui retranscrit les clés étrangères et *REFERENCES* qui indique la table à laquelle la clé étrangère fait référence. Ici la clé *type_id* fait référence à la clé primaire de la table *type,* la clé *status_id* fait référence à la clé primaire de la table *status* et la clé *speciality_id* fait référence à la clé primaire de la table *speciality*.

J'ai ensuite rempli la table *mission* :

```
/*remplissage de la table mission*/
INSERT INTO kgb.mission (title, description, codeName, country, startDate,
endDate, type_id, status_id, speciality_id) VALUES ('Mission1', 'Description
de la mission 1', 'Tango Charly', 'Canada', '2020-05-15', '2022-01-15', 1, 1,
1);
INSERT INTO kgb.mission (title, description, codeName, country, startDate,
endDate, type_id, status_id, speciality_id) VALUES ('Mission2', 'Description
de la mission 2', 'Operation Espadon', 'Espagne', '2023-02-12', '2023-03-12',
2, 2, 2);
INSERT INTO kgb.mission (title, description, codeName, country, startDate,
endDate, type_id, status_id, speciality_id) VALUES ('Mission3', 'Description
de la mission 3', 'Tango Charly', 'Norvege', '2021-04-05', '2022-04-05', 3, 3,
3);
```

J'ai ensuite créé et rempli les tables de liaison pour les liaisons fortes, comme *mission_agent*, *mission_contact*, *mission_target*, et *mission_hideout* :

```
/*Création de la table de liaison mission_agent*/
CREATE TABLE kgb.mission_agent (
  mission_id INT,
  agent_id INT,
  FOREIGN KEY (mission_id) REFERENCES kgb.mission (id),
  FOREIGN KEY (agent_id) REFERENCES kgb.agent (id)
)engine=INNODB;

/*remplissage de la table de liaison mission_agent*/
INSERT INTO kgb.mission_agent (mission_id, agent_id) VALUES (1, 1);
INSERT INTO kgb.mission_agent (mission_id, agent_id) VALUES (1, 2);

/*Création de la table de liaison mission_contact*/
CREATE TABLE kgb.mission_contact (
  mission_id INT,
  contact_id INT,
  FOREIGN KEY (mission_id) REFERENCES kgb.mission (id),
  FOREIGN KEY (contact_id) REFERENCES kgb.contact (id)
)engine=INNODB;

/*Remplissage de la table de liaison mission_contact*/
INSERT INTO kgb.mission_contact (mission_id, contact_id) VALUES (1, 1);

/*Création de la table de liaison mission_target*/
CREATE TABLE kgb.mission_target (
  mission_id INT,
  target_id INT,
  FOREIGN KEY (mission_id) REFERENCES kgb.mission (id),
  FOREIGN KEY (target_id) REFERENCES kgb.target (id)
)engine=INNODB;

/*Remplissage de la table de liaison mission_target*/
INSERT INTO kgb.mission_target (mission_id, target_id) VALUES (1, 2);

/*Création de la table de liaison mission_hideout*/
CREATE TABLE kgb.mission_hideout (
  mission_id INT,
  hideout_id INT,
  FOREIGN KEY (mission_id) REFERENCES kgb.mission (id),
  FOREIGN KEY (hideout_id) REFERENCES kgb.hideout (id)
)engine=INNODB;

/*Remplissage de la table de liaison mission_hideout*/
INSERT INTO kgb.mission_hideout (mission_id, hideout_id) VALUES (1, 3);
```

Afin de sauvegarder la base de données à partir du terminal, il faut utiliser la commande *mysqldump :*

```
mysqldump -u root -p kgb > backup_kgb.sql
```

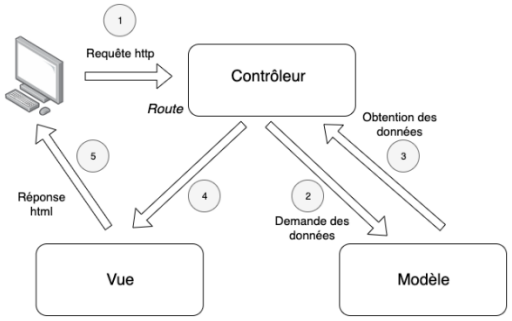
Pour restaurer la base de données, il faudra utiliser la commande inverse :

```
mysql -u root -p kgb < backup_kgb.sql
```

Après avoir créé la base de données, il faut maintenant créer l'application en elle-même.

2. Création de l'application :

Conformément au but du projet, j'ai créé cette application en PHP natif, c'est-à-dire sans Framework (Laravel ou Symfony). J'ai utilisé l'architecture MVC (*Model Vue Controller*).

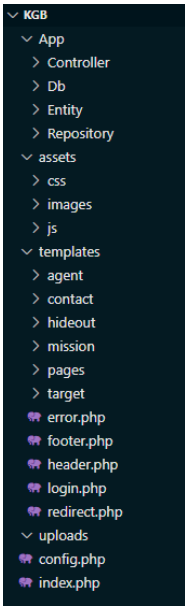


Le modèle représente les données de l'application et les règles métier associées. Il contient les méthodes qui permettent de manipuler ces données, ainsi que les fonctions de validation, de transformation et de persistance.
La vue représente l'interface utilisateur de l'application, c'est-à-dire la manière dont les données sont présentées à l'utilisateur final. Elle est responsable de l'affichage des données, de leur mise en forme et de leur organisation.
Le contrôleur agit comme l'intermédiaire entre le modèle et la vue. Il gère les interactions utilisateur (requêtes), récupère les données du modèle et les transmet à la vue.

Les prochains paragraphes expliquent le code que j'ai créé et quelles fonctions elles assurent au sein de l'application.

2.1 Organisation du code :

J'ai tout d'abord organisé mon code en dossier de cette manière :



Un dossier *App* dans lequel seront placés :
-Les contrôleurs.
-Les entités. Les "entités" font référence aux tables de la base de données.
-Les repositories gèrent l'accès aux données de la base de données.
-Le dossier Db qui gère l'accès à la base de données elle-même.

Un dossier *assets* dans lequel seront placés :
-Le dossier CSS où sont placés les fichiers CSS de customisation de l'affichage des pages créées.
-Le dossier images où sont placées les images utilisées pour l'affichage des pages.
-Le dossier JS où sont placés les fichiers Javascript qui dynamisent les pages et gèrent les événements utilisateur.

Un dossier *templates* dans lequel sont placées les vues. J'ai créé un dossier par entité afin de pouvoir plus facilement repérer les bons fichiers.

2.2 Exemple de fonctionnement, du contrôleur général vers la page d'accueil :

Dans ce paragraphe j'explique le fonctionnement de l'application pour le chargement de la page principale du site. Nous partons du fichier index.php qui est appelé la racine du site pour définir ce qui sera affiché en bout de chaîne.

Tout d'abord le fichier index.php :

```php
<?php
//création de la variable _ROOTPATH_
define('_ROOTPATH_', __DIR__);

//se charge de faire les include
spl_autoload_register();
```

```php
//dépendance, on utilise le Controller qui se trouve dans le namespace App\Controller
use App\Controller\Controller;

//on crée un nouvelle instance de Controller
$controller = new Controller();

//on appelle la fonction route()
$controller->route();
```

Ce fichier effectue les opérations suivantes :
- Initialise la variable _ROOTPATH_
- Appelle un mécanisme qui permet de charger automatiquement les classes au fur et à mesure de leur utilisation, sans avoir à les inclure manuellement à chaque fois
- Appelle une instance du contrôleur *Controller*.php
- Appelle la fonction *route() de Controller.php*

Comme le suggère l'étape précédente j'ai créé le contrôleur *Controller.php*. Il a pour but d'aiguiller l'utilisateur vers un contrôleur plus spécifique. Dans cette classe se trouvent 2 fonctions. La fonction *route()* qui aiguille vers le second contrôleur en fonction des paramètres dans l'URL, et la fonction *render()* qui gère la manière d'appel à la vue, en intégrant ou non des paramètres. Nous allons utiliser cette variable pour transmettre des données à la vue.

Exemple avec l'URL :
http://localhost/kgb/index.php?controller=page&action=home

Le router du contrôleur va comprendre qu'on veut utiliser la route « page ».

```php
try {
            if (isset($_GET['controller'])) {
             switch ($_GET['controller']) {
                 case 'page':
                     $pageController = new HomeController();
                     $pageController->route();
                 break;
```

Dans ce cas-là nous appelons une nouvelle instance d'un autre contrôleur, le *HomeController* et nous effectuons la méthode *route()*.

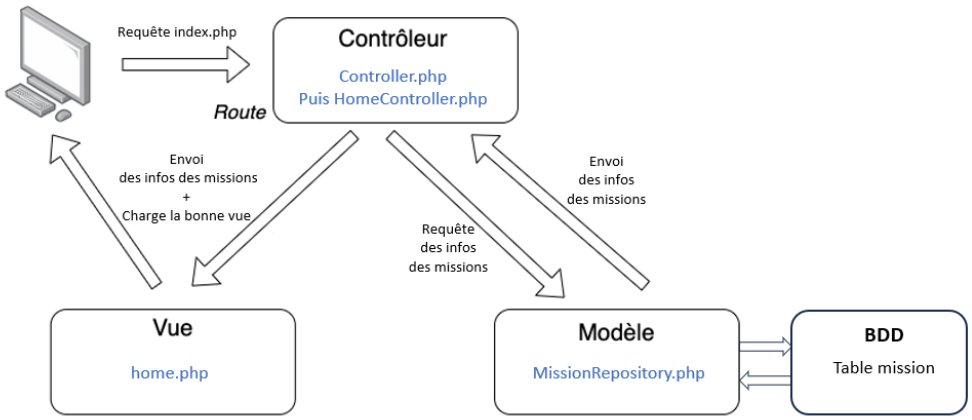
J'ai créé le *HomeController*, qui est une classe fille de *Controller*, contient également une fonction *route()*.

```php
<?php

namespace App\Controller;

use App\Repository\MissionRepository;

class HomeController extends Controller {
    public function route() :void {
        if (isset($_GET['action'])) {
            switch ($_GET['action']) {
```

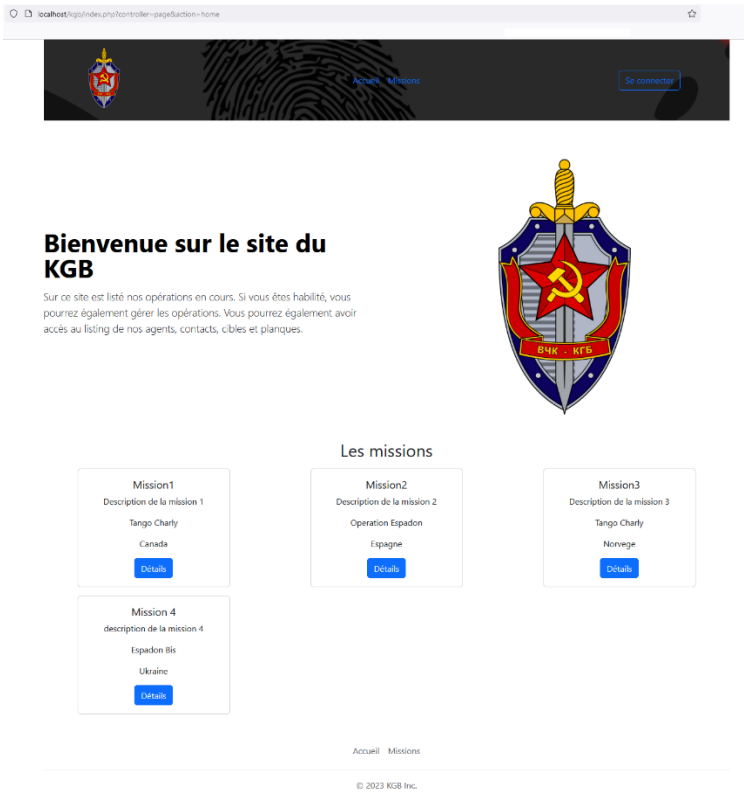
```
                case 'home':
                    $this->home();
                break;
                default:
                    //
                break;
            }
        } else {
            //charger la page d'accueil
            $this->home();
        }
    }

    protected function home() {

        $missionRepository = new MissionRepository();
        $missions = $missionRepository->findAll();

        $params = [
            'missions' => $missions,
        ];

        $this->render('/templates/pages/home.php', $params);
    }
}
```

La fonction *route()* scrute la variable *action* dans l'URL. Si elle est égale à 'home', elle effectue la fonction *home()* de la même classe. Cette fonction a pour but de récupérer toutes les informations des missions en base de données grâce au repository *MissionRepository()* que je détaillerai plus tard. Elle transmets les données des missions à la vue *home.php* et affiche la page *home.php* grâce à la fonction *render()*.

Je résume le code créé par le schéma ci-dessous :



Nous arrivons via ce cheminement selon l'architecture MVC à la page d'accueil :



Après avoir montré la mise en place du MVC dans mon application, je vais expliquer en détail chaque brique qui m'a permis de créer une application fonctionnelle. Je détaillerai dans les paragraphes suivants les entités, les *repositories*, leurs appels par les contrôleurs, et enfin les vues.

2.3 Les entités :

Les entités sont les classes qui correspondent aux tables de la base de données. Afin de pouvoir gérer les tables de la base de données, j'ai créé les entités suivantes :

-Agent.php
-Contact.php
-Mission.php
-Hideout.php
-Target.php
-Speciality.php
-Status.php
-Type.php
-User.php

J'ai également créé les entités correspondant aux tables de liaison, conformément au modèle physique de la méthode Merise :

-AgentSpeciality.php
-MissionAgent.php
-MissionContact.php
-MissionTarget.php
-MissionHideout.php

Un exemple d'entité crée, *Contact.php* :

```php
<?php

namespace App\Entity;

class Contact {
    protected ?int $id = null;
    protected string $firstName;
    protected string $lastName;
    protected \DateTime $birthDate;
    protected int $code;
    protected string $nationality;

    /**
     * Get the value of id
     */
    public function getId()
    {
        return $this->id;
    }

    /**
     * Set the value of id
     *
     * @return  self
     */
    public function setId($id)
    {
        $this->id = $id;

        return $this;
    }

    /**
     * Get the value of firstName
     */
    public function getFirstName()
    {
        return $this->firstName;
    }

    /**
     * Set the value of firstName
     *
     * @return  self
     */
    public function setFirstName($firstName)
    {
```

```
        $this->firstName = $firstName;

        return $this;
    }

    /**
     * Get the value of lastName
     */
    public function getLastName()
    {
        return $this->lastName;
    }

    /**
     * Set the value of lastName
     *
     * @return  self
     */
    public function setLastName($lastName)
    {
        $this->lastName = $lastName;

        return $this;
    }

    /**
     * Get the value of birthDate
     */
    public function getBirthDate()
    {
        return $this->birthDate;
    }

    /**
     * Set the value of birthDate
     *
     * @return  self
     */
    public function setBirthDate($birthDate)
    {
        $this->birthDate = $birthDate;

        return $this;
    }

    /**
     * Get the value of code
     */
```

```
    public function getCode()
    {
        return $this->code;
    }

    /**
     * Set the value of code
     *
     * @return  self
     */
    public function setCode($code)
    {
        $this->code = $code;

        return $this;
    }

    /**
     * Get the value of nationality
     */
    public function getNationality()
    {
        return $this->nationality;
    }

    /**
     * Set the value of nationality
     *
     * @return  self
     */
    public function setNationality($nationality)
    {
        $this->nationality = $nationality;

        return $this;
    }
}
```

J'ai construit cette classe en déclarant tout d'abord les propriétés de la classe en « *protected* », ce qui permet de contrôler l'accès à ces propriétés à l'intérieur de la classe et dans ses classes dérivées (héritage). J'ai également typé chaque paramètre (entier, chaîne de caractères ou date par exemple) de manière à garantir l'intégrité des données. J'ai créé les *getters* et les *setters* qui permettent d'écrire et lire les propriétés.

2.4 Les *Repositories* :

Le *Repositories* sont des classes qui permettent la communication avec la base de données. J'ai créé les *Repositories* de chaque entité. Ainsi chaque entité a un lien avec la base de données.

Je continue avec l'exemple des contacts :
Après avoir créé l'entité *Contact.php* j'ai créé le repository *ContactRepository.php*, qui permet de récupérer les données de la table *contact*.

```php
<?php

namespace App\Repository;

use App\Entity\Contact;
use App\Db\Mysql;

class ContactRepository {

    public function findOneById(int $id){

        $mysql = Mysql::getIsntance();
        $pdo = $mysql->getPDO();
        $query = $pdo->prepare('SELECT * FROM contact WHERE id = :id');
        $query->bindValue(':id', $id, $pdo::PARAM_INT);
        $query->execute();
        $contact = $query->fetch();

        $birthDate = new \DateTime($contact[3]);

        $contactEntity = new Contact();
        $contactEntity->setId($contact[0]);
        $contactEntity->setFirstName($contact[1]);
        $contactEntity->setLastName($contact[2]);
        $contactEntity->setBirthDate($birthDate);
        $contactEntity->setCode($contact[4]);
        $contactEntity->setNationality($contact[5]);
        return $contactEntity;
    }

    public function findAll(){

        $mysql = Mysql::getIsntance();
        $pdo = $mysql->getPDO();
        $query = $pdo->prepare('SELECT * FROM contact');
        $query->execute();
        $contacts = $query->fetchAll();
        $contactEntities = [];

        foreach ($contacts as $contact) {
            $birthDate = new \DateTime($contact[3]);
            $contactEntity = new Contact();
            $contactEntity->setId($contact[0]);
            $contactEntity->setFirstName($contact[1]);
            $contactEntity->setLastName($contact[2]);
```

```
            $contactEntity->setBirthDate($birthDate);
            $contactEntity->setCode($contact[4]);
            $contactEntity->setNationality($contact[5]);

            $contactEntities[] = $contactEntity;
        }

        return $contactEntities;
    }


    public function delete(int $id): bool {

    $mysql = Mysql::getIsntance();
    $pdo = $mysql->getPDO();
    $query = $pdo->prepare('DELETE FROM contact WHERE id = :id');
    $query->bindValue(':id', $id, \PDO::PARAM_INT);
    $result = $query->execute();
    return $result;
    }

    public function update(Contact $contact): bool
    {

        $mysql = Mysql::getIsntance();
        $pdo = $mysql->getPDO();

        $query = $pdo->prepare('UPDATE contact SET firstName = :firstName,
lastName = :lastName, birthDate = :birthDate, code = :code, nationality =
:nationality WHERE id = :id');

        $query->bindValue(':id', $contact->getId(), \PDO::PARAM_INT);
        $query->bindValue(':firstName', $contact->getFirstName(),
\PDO::PARAM_STR);
        $query->bindValue(':lastName', $contact->getLastName(),
\PDO::PARAM_STR);
        $query->bindValue(':birthDate', $contact->getBirthDate()->format('Y-m-
d'), \PDO::PARAM_STR);
        $query->bindValue(':code', $contact->getCode(), \PDO::PARAM_INT);
        $query->bindValue(':nationality', $contact->getNationality(),
\PDO::PARAM_STR);

        $result = $query->execute();

        return $result;
    }


    public function create(Contact $contact) {
```

```
    $mysql = Mysql::getIsntance();
    $pdo = $mysql->getPDO();

    $query = $pdo->prepare('INSERT INTO contact (firstName, lastName,
birthDate, code, nationality) VALUES (:firstName, :lastName, :birthDate,
:code, :nationality)');
    $query->bindValue(':firstName', $contact->getFirstName());
    $query->bindValue(':lastName', $contact->getLastName());
    $query->bindValue(':birthDate', $contact->getBirthDate()->format('Y-m-
d'));
    $query->bindValue(':code', $contact->getCode());
    $query->bindValue(':nationality', $contact->getNationality());

    $query->execute();
    }

}
```

Dans cette classe j'ai créé 5 fonctions qui effectuent des opérations bien précises avec la table *contact* de la base de données :
-La méthode *findOneById()* récupère les infos du contact ayant pour *id* celui qui est donné en entrée.
-La méthode *findAll()* récupère tous les contacts et ses attributs.
-La méthode *delete()* supprime le contact ayant l'*id* transmis de la base de données.
-La méthode *update()* modifie le contact donné en entrée.
-La méthode *create()* crée un nouveau contact.

Pour chaque méthode que j'ai créée :
-Je récupère une instance de la classe Mysql que j'ai créé, je détaillerai cette fonction ultérieurement.

```
$mysql = Mysql::getIsntance();
$pdo = $mysql->getPDO();
```

-Je monte la requête SQL à effectuer.

```
$query = $pdo->prepare('DELETE FROM contact WHERE id = :id');
```

-Afin de protéger la base de données et les données attenantes, j'ai lié la valeur envoyée dans la requête à un type particulier (en l'occurrence un entier).

```
$query->bindValue(':id', $id, \PDO::PARAM_INT);
```

-J'exécute la requête.

```
$query->execute();
```

J'ai effectué ce travail sur les 9 *Repositories* suivant :
-AgentRepository.php
-ContactRepository.php
-MissionRepository.php
-HideoutRepository.php
-TargetRepository.php
-SpecialityRepository.php
-StatusRepository.php
-TypeRepository.php

-UserRepository.php

J'ai également créé les *Repositories* correspondant aux entités de liaison :
-AgentSpecialityRepository.php
-MissionAgentRepository.php
-MissionContactRepository.php
-MissionTargetRepository.php
-MissionHideoutRepository.php

2.5 La classe Mysql.php:

Les *repositories* ont besoin de communiquer avec la base de données. Pour cela on utilise la librairie PDO (PHP Data Object).
https://www.php.net/manual/fr/book.pdo.php
Par soucis d'optimisation du code et des appels à la librairie, j'ai écrit une classe *Mysql.php* de manière à créer une instance de PDO au premier appel de la classe, et de renvoyer l'instance créée lors des appels suivants. J'ai pour cela utilisé le *Design Pattern* du *Singleton* qui convient spécifiquement à ce cas.

La classe *Mysql.php* :

```php
<?php

namespace App\Db;

class Mysql {

    private $db_name;
    private $db_user;
    private $db_password;
    private $db_port;
    private $db_host;

    private $pdo = null;
    private static $_instance = null;


    private function __construct() {
        $conf = require_once _ROOTPATH_.'/config.php';

        if (isset($conf['db_name'])) {
            $this->db_name = $conf['db_name'];
        }
        if (isset($conf['db_user'])) {
            $this->db_user = $conf['db_user'];
        }
        if (isset($conf['db_password'])) {
            $this->db_password = $conf['db_password'];
        }
        if (isset($conf['db_port'])) {
            $this->db_port = $conf['db_port'];
```

```
        }
        if (isset($conf['db_host'])) {
            $this->db_host = $conf['db_host'];
        }
    }

    public static function getIsntance():self {

        if (is_null(self::$_instance)) {
            return self::$_instance = new Mysql();
        } else {
            return self::$_instance;
        }

    }

    public function getPDO():\PDO {

        if(is_null($this->pdo)) {
            $this->pdo = new \PDO('mysql:dbname='.$this->db_name.';charset=utf8;host='.$this->db_host.':'.$this->db_port, $this->db_user, $this->db_password);
        }
        return $this->pdo;
    }
}
```

A noter que cette classe fait appel à un fichier *config.php*, dans lequel j'ai regroupé la configuration de ma base de données. Ce fichier renvoie juste un tableau de valeurs lors de son appel :

```
<?php
    //fichier de config pour la com bdd
    return [
        'db_name' => 'kgb',
        'db_user' => 'root',
        'db_password' => '',
        'db_port' => '3307',
        'db_host' => 'localhost',
    ];
```

2.6 Utilisation des *Repositories* dans les contrôleurs :

Afin de poursuivre l'explication de l'utilité des *repositories*, il faut maintenant expliquer comment les intégrer dans le cadre du MVC. J'ai intégré les *repositories* dans les contrôleurs.

Je poursuis mon exemple d'implémentation des contacts dans mon application.

Dans le *ContactController.php*, j'ai implémenté une fonction *list()* qui permet de récupérer tous les contacts en base de données et ainsi de les afficher lorsqu'on se rend sur la page de listing des contacts :

http://localhost/kgb/index.php?controller=contact&action=list
Comme le montre l'URL on appelle le contrôleur *contact* et on lui indique de faire l'action *list*.
La fonction *list()* du *ContactController.php* :

```php
protected function list() {
        $contactRepository = new ContactRepository;
        $contacts = $contactRepository->findAll();

        $params = [
            'contacts' => $contacts
        ];

        $this->render('/templates/contact/list.php', $params);
    }
```

Cette fonction fait appel à une nouvelle instance de la classe *ContactRepository* pour la communication avec la base de données. Nous utilisons la méthode *findAll()* du *Repository* pour récupérer tous les contacts que j'ai placé dans la variable *$contacts*.

Je place les contacts récupérés dans la variable $params qui est utilisée pour la transmission des données à la vue. Dans cet exemple grâce à la méthode *render()* du contrôleur je charge la page list.php des contacts et je transmets également les données à la page.

L'administrateur du site devant lister, modifier, supprimer et créer des missions, agents, contacts, cibles et planques, j'ai créé pour chaque contrôleur les méthodes *list()*, *show()*, *create()*, *update()* et *delete()*. J'ai donc créé ce qu'on appelle communément un CRUD pour chaque entité.

Les contacts étant passés à la vue, j'explique maintenant comment j'ai géré l'affichage des pages.

2.7 Les vues :

J'ai créé 21 vues pour chaque page de visualisation ou d'administration de l'application. En plus de la vue *home.php* vue précédemment, dans le dossier */templates* j'ai créé un dossier *agent*, *contact*, *mission*, *hideout* (planque), et *target* (cible) contenant chacune les vues du même nom :
-list.php
-show.php
-create.php
-update.php
N.B : Je n'ai pas mis de vue pour la fonction *delete()* de chaque contrôleur. J'ai codé cette fonction de telle sorte que lorsqu'on l'appelle en appuyant sur le bouton « supprimer » de chaque entité, il supprime l'entité à l'*id* concernée et redirige directement l'utilisateur vers la page de listing.

Je continue mon exemple précédent des contacts. Je développe ici la vue *list.php* que j'ai créé :

```php
<?php
    require_once _ROOTPATH_.'/templates/header.php';
    require_once _ROOTPATH_.'/templates/redirect.php';
?>

<div class="text-center">
    <h1>
```

```
        Nos Contacts
    </h1>
</div>
<div class="row text-center">
<?php foreach ($contacts as $contact) { ?>
    <div class="col-md-4 my-2 d-flex justify-content-center">
        <div class="card" style="width: 18rem;">
            <div class="card-body">
                <h5 class="card-title"><?php echo  $contact->getFirstName().'
'.$contact->getLastName(); ?></h5>
                <p class="card-text"><?php echo  $contact->getBirthDate()-
>format('d/m/Y'); ?></p>
                <p class="card-text"><?php echo  $contact->getCode(); ?></p>
                <p class="card-text"><?php echo  $contact->getNationality();
?></p>
                <a
href="index.php?controller=contact&action=show&id=<?=  $contact->getId(); ?>"
class="btn btn-primary">Détails</a>
            </div>
        </div>
    </div>
<?php } ?>
</div>
<br>
<div class="text-center">
<a href="index.php?controller=contact&action=create" class="btn btn-
secondary">Créer un nouveau contact</a>
</div>

<?php require_once _ROOTPATH_.'/templates/footer.php'; ?>
```

Ce fichier appelle le code HTML commun à toutes les pages du site grâce à le fonction *require_once*.
Le *header.php* :

```
<?php
if (session_status() == PHP_SESSION_NONE) {
    session_start();
}
?>

<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css"
rel="stylesheet" integrity="sha384-
```

```
9ndCyUaIbzAi2FUVXJi0CjmCapSmO7SnpJef0486qhLnuZ2cdeRhO02iuK6FUUVM"
crossorigin="anonymous">
    <link href="assets/css/kgbstyle.css" rel="stylesheet">
    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/bootstrap.bundle.min
.js" integrity="sha384-
geWF76RCwLtnZ8qwWowPQNguL3RmwHVBC9FhGdlKrxdiJJigb/j/68SIy3Te4Bkz"
crossorigin="anonymous"></script>
    <title>KGB</title>
</head>
<body>
    <div class="container">
        <header class="header-space d-flex flex-wrap align-items-center
justify-content-center justify-content-md-around py-3 mb-4 border-bottom">
        <div class="col-md-3 mb-2 mb-md-0">
            <a href="index.php?controller=page&action=home" class="d-inline-
flex link-body-emphasis text-decoration-none">
            <img src="assets/images/logo_KGB.png" width="60px" alt="logo kgb">
            </a>
        </div>

        <ul class="nav col-12 col-md-auto mb-2 justify-content-center mb-md-
0">
            <li><a href="index.php?controller=page&action=home" class="nav-
link px-2">Accueil</a></li>
            <li><a href="index.php?controller=mission&action=list" class="nav-
link px-2">Missions</a></li>
            <?php
                if (isset($_SESSION['user'])) {
                    echo '
                    <li><a href="index.php?controller=target&action=list"
class="nav-link px-2">Cibles</a></li>
                    <li><a href="index.php?controller=contact&action=list"
class="nav-link px-2">Contacts</a></li>
                    <li><a href="index.php?controller=hideout&action=list"
class="nav-link px-2">Planques</a></li>
                    <li><a href="index.php?controller=agent&action=list"
class="nav-link px-2">Agents</a></li>
                    ';
                }
            ?>
        </ul>

        <div class="col-md-3 text-end">
            <?php
                if (!isset($_SESSION['user'])) {
```

```
                    echo '<a
href="index.php?controller=auth&action=login"><button type="button" class="btn
btn-outline-primary me-2">Se connecter</button></a>';
                } else {
                    echo '<a
href="index.php?controller=auth&action=logout"><button type="button"
class="btn btn-outline-primary me-2">Se déconnecter</button></a>';
                }

            ?>
        </div>
        </header>

        <main>
```

Ce fichier regroupe l'entête de la page html (balises *DOCTYPE*, *html* et *head*), ainsi que le début du corps (balise *body*) du site ainsi que le *header* contenant le logo, la barre de menu et le bouton de connexion.

Est appelé également le fichier *redirect.php* :

```
<?php
if (!isset($_SESSION['role']) || $_SESSION['role'] !== 'admin') {
    // Redirigez l'utilisateur vers la page d'accueil
    header('Location:
http://localhost/kgb/index.php?controller=page&action=home');
    exit();
}
?>
```

C'est un élément de sécurité qui vérifie que l'utilisateur loggé ait les bons droits pour disposer de cette page.

De la même façon le fichier *footer.php* est appelé en fin de page :

```
</main>

        <footer class="py-3 my-4">
            <ul class="nav justify-content-center border-bottom pb-3 mb-3">
                <li class="nav-item"><a
href="index.php?controller=page&action=home" class="nav-link px-2 text-body-
secondary">Accueil</a></li>
                <li class="nav-item"><a
href="index.php?controller=mission&action=list" class="nav-link px-2 text-
body-secondary">Missions</a></li>
                <?php
                if (isset($_SESSION['user'])) {
                    echo '
                    <li><a href="index.php?controller=target&action=list"
class="nav-link px-2 text-body-secondary">Cibles</a></li>
                    <li><a href="index.php?controller=contact&action=list"
class="nav-link px-2 text-body-secondary">Contacts</a></li>
```

```
                    <li><a href="index.php?controller=hideout&action=list"
class="nav-link px-2 text-body-secondary">Planques</a></li>
                    <li><a href="index.php?controller=agent&action=list"
class="nav-link px-2 text-body-secondary">Agents</a></li>
                    ';
                }
            ?>
            </ul>
            <p class="text-center text-body-secondary">© 2023 KGB Inc.</p>
        </footer>
    </div>

</body>
</html>
```

Pour afficher le listing des contacts, j'ai choisi l'affichage par des cartes de la librairie Bootstrap :

```
<div class="text-center">
    <h1>
        Nos Contacts
    </h1>
</div>
<div class="row text-center">
<?php foreach ($contacts as $contact) { ?>
    <div class="col-md-4 my-2 d-flex justify-content-center">
        <div class="card" style="width: 18rem;">
            <div class="card-body">
                <h5 class="card-title"><?php echo  $contact->getFirstName().'
'.$contact->getLastName(); ?></h5>
                <p class="card-text"><?php echo  $contact->getBirthDate()-
>format('d/m/Y'); ?></p>
                <p class="card-text"><?php echo  $contact->getCode(); ?></p>
                <p class="card-text"><?php echo  $contact->getNationality();
?></p>
                <a
href="index.php?controller=contact&action=show&id=<?=  $contact->getId(); ?>"
class="btn btn-primary">Détails</a>
            </div>
        </div>
    </div>
<?php } ?>
</div>
<br>
<div class="text-center">
<a href="index.php?controller=contact&action=create" class="btn btn-
secondary">Créer un nouveau contact</a>
</div>
```

Dans ce fichier je récupère la variable *$contacts* venant du contrôleur. J'utilise une boucle *foreach* pour créer une carte Bootstrap de chaque contact avec les informations principales.
Comme le montre le code je récupère et affiche dans du contenu HTML le prénom, le nom, la date de naissance, le code, la nationalité du contact.
Je crée également un bouton « détail » pour renvoyer l'administrateur vers la page de détail du contact.

Enfin je crée un bouton générique « Créer un nouveau contact » de pour rediriger l'administrateur vers la page de création de contact.
Après avoir expliqué le mécanisme de fonctionnement des vues, je vais maintenant m'attarder sur des fonctions spécifiques de l'application.

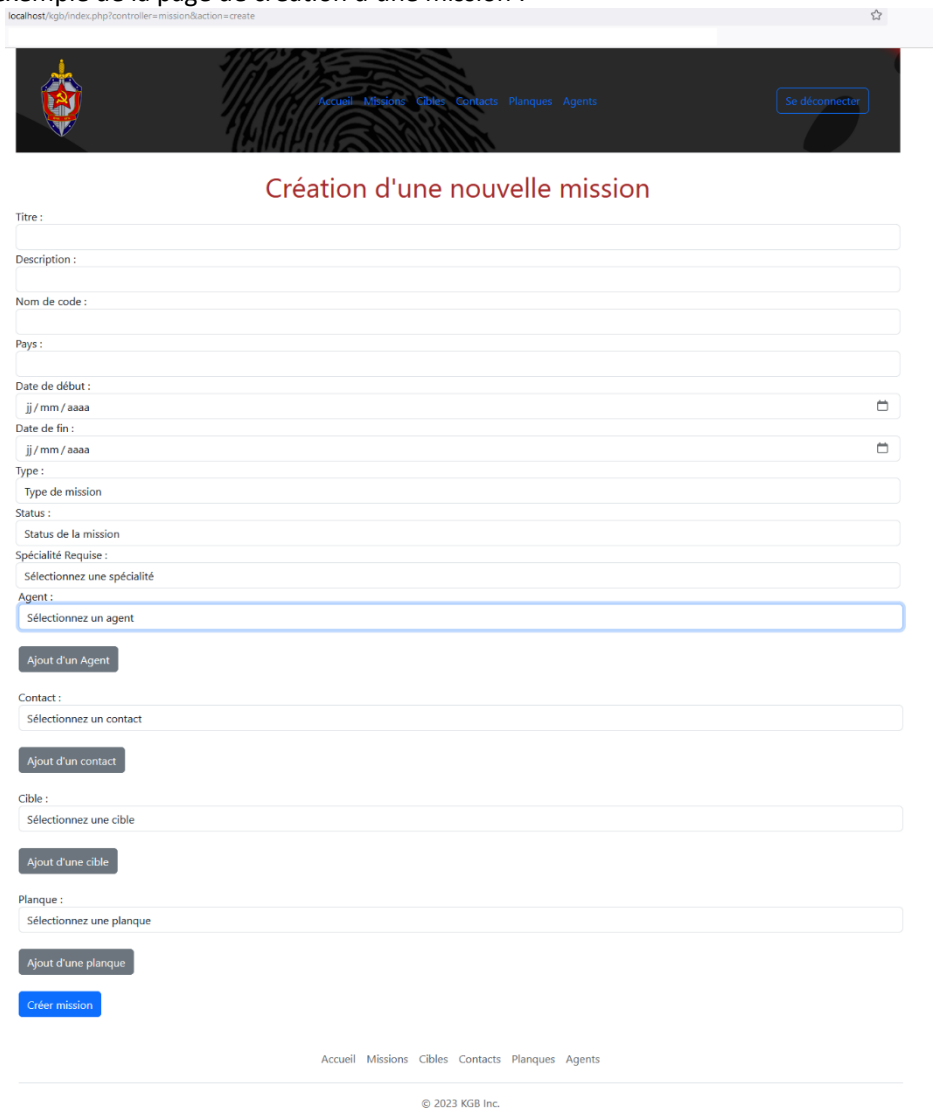
3 Les fonctions spécifiques :

Dans ce paragraphe je vais détailler le comportement de l'application avec les entités de liaison, les repositories et leur impact sur le développement des fonctions complexes.

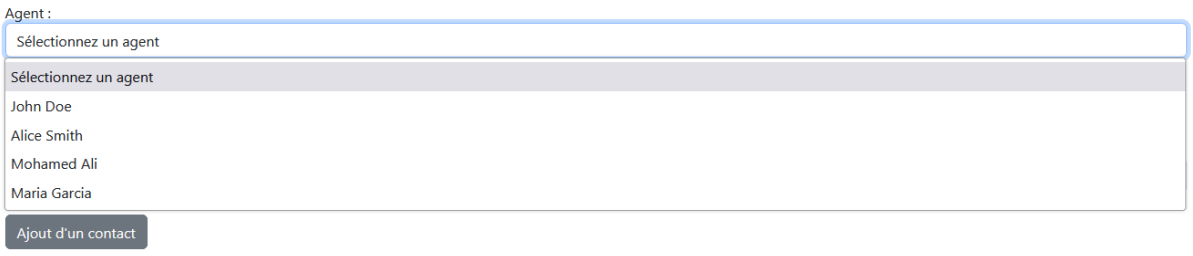
3.1 Liaison entre les entités *Mission* et *MissionAgent.php* :

Une liaison forte entre deux entités implique comme dans cet exemple que lorsqu'on va créer, modifier, ou supprimer une mission, nous devons travailler sur l'entité *Mission*. Mais cela implique aussi des répercussions sur l'entité *MissionAgent*. J'explique le code spécifique que j'ai implémenté dans le cas suivant.

Je prends l'exemple de la page de création d'une mission :



Lors du remplissage du formulaire de création de mission, lorsque je clique sur « Sélectionner un agent », Le formulaire doit faire apparaître en option de choix les agents que j'ai en base de données :



Pour cela j'ai implémenté le code suivant dans la méthode *create()* du *MissionControlleur.php* :

```php
$agentRepo = new AgentRepository();
$agents = $agentRepo->findAll();

/*code intermédiaire*/
```

```php
$params = [
                    /*autres params avant*/
                    'agents' => $agents,
                    /*autres params après*/
                ];

$this->render('/templates/mission/create.php', $params);
```

Dans la vue je procède ainsi :

```php
<div class="form-group">
        <label for="agents[]">Agent :</label>
        <select name="agents[]" id="agents" class="form-control">
            <option value="">Sélectionnez un agent</option>
            <?php foreach ($agents as $agent) : ?>
                <option value="<?php echo $agent->getId(); ?>"><?php echo
$agent->getFirstName().' '.$agent->getLastName(); ?></option>
            <?php endforeach; ?>
        </select>
    </div>
```

Pour chaque agent récupéré de la base de données je crée une balise option proposant le prénom et le nom de l'agent. A noter que dans le champs *value* de l'option, j'injecte l'*id* de l'agent. Ce qui va faciliter le traitement par la suite.
Lorsqu'on soumet le formulaire, cet *id* de l'agent choisi est remonté vers le contrôleur. Le contrôleur scrute si le formulaire a été soumis par la fonction :

```php
if (isset($_POST['createMission'])) {
```

Si le formulaire est complété alors j'effectue le code suivant :

```php
$mission = new Mission();

                $mission->setTitle($_POST['title']);
                $mission->setDescription($_POST['description']);
                $mission->setCodeName($_POST['codeName']);
                $mission->setCountry($_POST['country']);
                $mission->setStartDate(new \DateTime($_POST['startDate']));

                if ($_POST['endDate']){
                    $mission->setEndDate(new \DateTime($_POST['endDate']));
                }

                $mission->setType_id($_POST['type']);
                $mission->setStatus_id($_POST['status']);
                $mission->setSpeciality_id($_POST['speciality']);

                $missionRepository = new MissionRepository();
                $missionRepository->create($mission);
```

A partir de la variable *$_POST* le contrôleur récupère les valeurs saisies dans le formulaire et les injecte à une nouvelle instance de l'entité *mission*. Le contrôleur crée une nouvelle instance du *MissionRepository*, ce qui permet d'appeler la fonction *create()* et donc de créer la mission en base de données.

Il faut également mettre à jour la table *MissionAgent*. Pour cela je récupère le tableau des agents sélectionnés dans le formulaire.

```
$selectedAgents = $_POST['agents'];
```

N.B. : J'ai conçu l'application pour que plusieurs agents puissent être affectés à une mission, je développerai cette fonction ultérieurement.

```
if(!is_null($selectedAgents)) {

                    foreach ($selectedAgents as $selectedAgent) {
                        $mission_agentRepo = new MissionAgentRepository();
                        $mission_agentRepo->create($createdMissionId,
$selectedAgent);
                    }

                }
```

Selon le code ci-dessus, Pour chaque agent affecté à la mission, l'application crée une nouvelle instance du *MissionAgentRepository*, et ce qui permet de créer une nouvelle ligne dans la table, en affectant l'*id* de la mission créée et l'*id* de l'agent sélectionné, qui provient du champs *value* précédemment cité.

De la même manière lorsqu'on supprime une mission de la base de données, il faut impérativement d'abord supprimer les liaisons qui existent entre la mission et les autres tables. Pour cela je détaille la fonction *delete()*.

```
try {
            if(isset($_GET['id'])) {

                $id = (int)$_GET['id'];

                $missionAgentRepository = new MissionAgentRepository();
                $missionAgentRepository->deleteByMissionId($id);

                $missionContactRepository = new MissionContactRepository();
                $missionContactRepository->deleteByMissionId($id);

                $missionTargetRepository = new MissionTargetRepository();
                $missionTargetRepository->deleteByMissionId($id);

                $missionHideoutRepository = new MissionHideoutRepository();
                $missionHideoutRepository->deleteByMissionId($id);

                $missionRepository = new MissionRepository();
                $missionRepository->delete($id);
                $this->list();

            } else {
```

```
                throw new \Exception('id introuvable');
            }
        } catch (\Exception $e) {
            $this->render('/templates/error.php', [
                'error'=> $e->getMessage(),
            ]);
        }
```

J'ai implémenté une logique de *try/catch* sur cette fonction de manière à gérer les différentes erreurs qui peuvent survenir. Selon la mission sélectionnée, si on appuie sur le bouton « supprimer » le contrôleur active cette méthode. Il ouvre une instance des *repositories* de liaisons fortes comme *MissionAgent*, *MissionContact*, *MissionTarget* et *MissionHideout*. Pour chaque instance il appelle la fonction *deleteByMissionId*(), méthode que j'ai créé pour chaque *repository* et qui supprime les lignes qui ont comme *id* de mission l'*id* passé en entrée de fonction. Les dépendances de liaisons fortes étant supprimée, le code supprime ensuite la mission. Enfin il redirige l'administrateur vers la page de listing des missions.

Pour les 4 liaisons fortes j'ai géré la création, le listing, la mise à jour et la suppression des différentes lignes en base de données.

3.2 Javascript / Ajout de plusieurs agents/contacts/cibles/planques :

En étudiant les attentes du client vis-à-vis de l'application, il est apparu le fait de résoudre le problème d'ajout de plusieurs agents, contacts, cibles et planques lors de la création d'une mission. Comme nous l'avons vu précédemment, j'ai géré l'affichage du formulaire de la page pour pouvoir sélectionner les agents de la base de données. Il en est de même pour les contacts, cibles et planques. J'ai ajouté une structure *select* pour chaque choix. Je décris ici la solution que j'ai mis en place pour répondre au besoin de l'application.

Après chaque structure *select*, j'ai ajouté une balise *div* vide ayant pour *id* *agentContainer*, *contactContainer*, *targetContainer* et *hideoutContainer*.

J'ai ensuite créé 4 fichiers javascript *addAgent.js, addContact.js, addTarget.js* et *addHideout.js*.

Ces fichiers étant très similaires, je décris le fonctionnement du fichier *addAgent.js* :

J'ai ajouté une fonction *AjouterAgent()*

```
function ajouterAgent() {
```

A l'intérieur de celle-ci, je récupère la *div agentContainer* :

```
const agentContainer = document.getElementById('agentContainer');
```

Je recrée en javascript une balise *div* qui est exactement celle qui contient la balise select vue précédemment.

```
// Création du nouvel élément div
    const agentDiv = document.createElement('div');
    agentDiv.classList.add('form-group');
    // Création du label
    const label = document.createElement('label');
    label.setAttribute('for', 'agents[]');
    label.textContent = 'Agent :';
    // Création du select
    const select = document.createElement('select');
```

```
    select.setAttribute('name', 'agents[]');
    select.setAttribute('id', 'agents');
    select.classList.add('form-control');
    // Création de l'option par défaut
    const defaultOption = document.createElement('option');
    defaultOption.setAttribute('value', '');
    defaultOption.textContent = 'Sélectionnez un agent';
    // Ajout de l'option par défaut au select
    select.appendChild(defaultOption);
```

Je récupère tous les noms d'agents de la vue :

```
const agentsSelect = document.getElementById('agents');
    const allAgents = Array.from(agentsSelect.options)
        .filter(option => option.value !== '')
        .map((option, index) => ({
            id: option.value,
            name: option.textContent.trim()
          }));
```

Pour tous les agents créés, je crée une balise *option*, à laquelle j'affecte dans le champs value l'*id* de l'agent et dans le champ de proposition son nom suivi de son prénom.

```
allAgents.forEach(agent => {
        const option = document.createElement('option');
        option.setAttribute('value', agent.id);
        option.textContent = agent.name;
        select.appendChild(option);
    });
```

Ayant construit tous les éléments, j'ai ensuite affecté les éléments aux bonnes balises *div* :

```
// Ajout du label et du select à la div
    agentDiv.appendChild(label);
    agentDiv.appendChild(select);

    // Ajout de la div à la div agentContainer
    agentContainer.appendChild(agentDiv);
```

Pour finir, à chaque ajout de balise permettant de sélectionner un nouvel agent, j'adjoins un bouton permettant de supprimer la div d'ajout, si on ne veut finalement pas ajouter d'autre agent :

```
const removeBtn = document.createElement('button');
    removeBtn.classList.add('btn', 'btn-sm', 'btn-danger');
    removeBtn.textContent = 'Supprimer';
    removeBtn.addEventListener('click', () => {
        agentDiv.remove();
    });

    agentDiv.appendChild(removeBtn);
```

Une fois cette fonction ajoutée, il faut l'appeler dans la vue :

```
<script src="/kgb/assets/js/addAgent.js"></script>
```

Pour appeler cette fonction, j'ai placé un bouton dans la vue :

```
<button type="button" class="btn btn-secondary" onclick="ajouterAgent()">Ajout
d'un Agent</button>
```

J'utilise l'attribut *onclick* de la balise HTML *button*.

J'utilise cette fonction dans la vue de création ainsi que celle de modification des missions.

3.3 Système d'authentification :

Le cahier des charges indique également qu'un utilisateur non connecté pouvait :
-consulter sur la page principale de l'application
-consulter sur la page de listing des missions
-consulter sur la page de détail de chaque mission

Les autres vues créées doivent être accessibles uniquement par l'administrateur du site, car elles contiennent des actions critiques sur la base de données et l'application.

Pour cela, je me suis éloigné du modèle physique de base de données, et j'ai créé une entité *User* :

```php
<?php

namespace App\Entity;

class User {
    protected ?int $id = null;
    protected string $firstName;
    protected string $lastName;
    protected string $email;
    protected string $password;
    protected string $role;
    protected \DateTime $creationDate;



    /**
     * Get the value of id
     */
    public function getId()
    {
        return $this->id;
    }

    /**
     * Set the value of id
     *
     * @return  self
     */
    public function setId($id)
```

```
    {
        $this->id = $id;

        return $this;
    }

    /**
     * Get the value of firstName
     */
    public function getFirstName()
    {
        return $this->firstName;
    }

    /**
     * Set the value of firstName
     *
     * @return  self
     */
    public function setFirstName($firstName)
    {
        $this->firstName = $firstName;

        return $this;
    }

    /**
     * Get the value of lastName
     */
    public function getLastName()
    {
        return $this->lastName;
    }

    /**
     * Set the value of lastName
     *
     * @return  self
     */
    public function setLastName($lastName)
    {
        $this->lastName = $lastName;

        return $this;
    }

    /**
     * Get the value of email
```

```
     */
    public function getEmail()
    {
        return $this->email;
    }

    /**
     * Set the value of email
     *
     * @return  self
     */
    public function setEmail($email)
    {
        $this->email = $email;

        return $this;
    }

    /**
     * Get the value of password
     */
    public function getPassword()
    {
        return $this->password;
    }

    /**
     * Set the value of password
     *
     * @return  self
     */
    public function setPassword($password)
    {
        $this->password = $password;

        return $this;
    }

    /**
     * Get the value of role
     */
    public function getRole()
    {
        return $this->role;
    }

    /**
     * Set the value of role
```

```
     *
     * @return  self
     */
    public function setRole($role)
    {
        $this->role = $role;

        return $this;
    }

    /**
     * Get the value of creationDate
     */
    public function getCreationDate()
    {
        return $this->creationDate;
    }

    /**
     * Set the value of creationDate
     *
     * @return  self
     */
    public function setCreationDate($creationDate)
    {
        $this->creationDate = $creationDate;

        return $this;
    }
}
```

Cette entité possède un prénom, un nom, un email, un mot de passe, un rôle et une date de création. Il possède donc les paramètres requis pour une identification.

J'ai également créé un *UserRepository.php* pour communiquer avec la table *user* de la base de données que j'ai précédemment créée.

J'ai également créé un contrôleur qui permet à l'administrateur de s'identifier, le *AuthController.php*. Ce contrôleur est enclenché à l'appui sur le bouton de connexion dans le *header* :

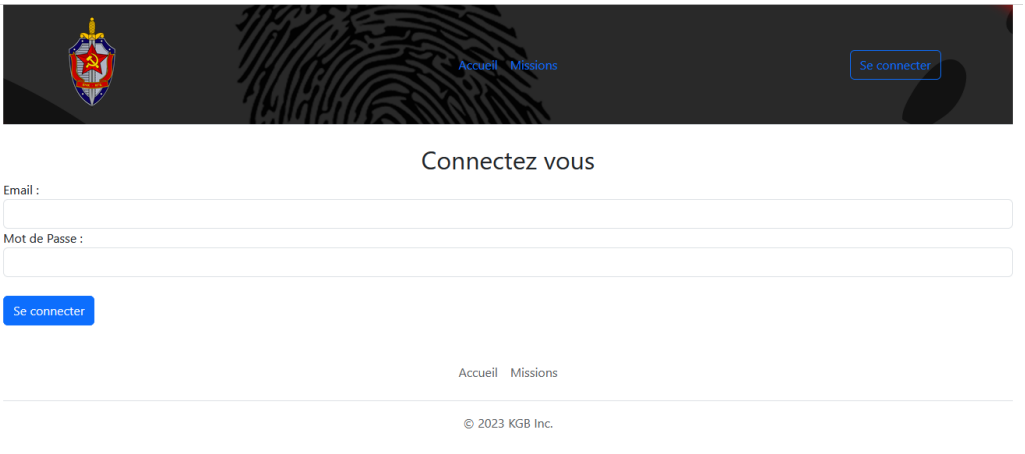
```
<?php
if (!isset($_SESSION['user'])) {
      echo '<a href="index.php?controller=auth&action=login"><button
type="button" class="btn btn-outline-primary me-2">Se connecter</button></a>';
} else {
      echo '<a href="index.php?controller=auth&action=logout"><button
type="button" class="btn btn-outline-primary me-2">Se
déconnecter</button></a>';
}
```

```
?>
```

Si un utilisateur est connecté, j'active le bouton de déconnexion. Il activera la fonction *logout()* du *AuthController*.
Si un utilisateur est déconnecté, j'active le bouton de connexion. Il activera la fonction *login()* du *AuthController*.

La fonction *route()* du contrôleur permet de scruter les 2 actions permises, c'est-à-dire le *login* (connexion) et le *logout* (déconnexion).

```php
 public function route() :void {

        try {
            if (isset($_GET['action'])) {
                switch ($_GET['action']) {
                    case 'login':
                        $this->login();
                    break;
                    case 'logout':
                        $this->logout();
                    break;
                    default:
                        throw new \Exception('L\'action '.$_GET['action'].'
n\'existe pas');
                    break;
                }
            } else {
                //si pas d'argument on charge la page d'accueil
                header("Location: index.php?controller=page&action=home");
            }
        } catch (\Exception $e) {
            $this->render('/templates/error.php', [
                'error'=> $e->getMessage(),
            ]);
        }
    }
```



```php
if (isset($_POST['loginUser'])) {

                $userRepository = new UserRepository;

                $email = $_POST['email'];
                $password = $_POST['password'];

                $user = new User;
                $user = $userRepository->findOneByEmail($email);

                if (isset($user)) {
                    // Vérifier le mot de passe hashé
                    $hashedPassword = $user->getPassword();

                    if (password_verify($password, $hashedPassword)) {
                        // Authentification réussie
                        session_start();
                        $_SESSION['user'] = $user;
                        $_SESSION['role'] = $user->getRole();

                        $missionRepository = new MissionRepository();
                        $missions = $missionRepository->findAll();

                        $params = [
                            'missions' => $missions,
                        ];

                        $this->render('/templates/pages/home.php', $params);
                    } else {
                        // Mot de passe incorrect, afficher un message
d'erreur ou effectuer une action appropriée
                        $errors[] = 'Mot de passe ou email incorrect. Veuillez
réessayer.';
                        $this->render('/templates/login.php', ['errors' =>
$errors]);
                    }
                }
```

Si le formulaire a été soumis, l'application crée une nouvelle instance du *UserRepository*. J'utilise sa fonction *findOneByEmail()* pour récupérer les information du compte lié à une adresse mail spécifique. Si j'arrive à récupérer un utilisateur lié à cet email, je vérifie le mot de passe par la fonction *password_verify()* qui prend en charge l'algorithme de hash (encrypté). Si le mot de passe est correct, j'assigne les données de l'utilisateur aux variables *$_SESSION['user']* et *$_SESSION['role']*. Je récupère les informations des missions et je redirige vers la page *home.php*.
Si le formulaire n'a pas été validé, je redirige l'utilisateur vers la page de *login*.

Dans la fonction *logout()* :

```
protected function logout()
{
    session_start();
    session_destroy();

    unset($_SESSION);

    header("Location: index.php?controller=page&action=home");
}
```

Je détruis la session et je remets à zéro la variable *$_SESSION*. Je redirige l'utilisateur vers la page *home.php*.

Comme vu précédemment, dans le *header* et le *footer* l'application scrute la variable *$_SESSION*. Si l'utilisateur est connecté et s'il est administrateur, alors dans les menus apparaissent les liens d'administration des agents, cibles, planques et contacts. Dans la page de listing des missions, l'administrateur aura accès au bouton de création d'une mission.

```
<?php
                if (isset($_SESSION['user']) && $_SESSION['role'] == 'admin')
{
                    echo '
                    <div class="text-center">
                    <a href="index.php?controller=mission&action=create"
class="btn btn-secondary">Créer une nouvelle mission</a>
                    </div>
                    ';
                }
            ?>
```

Si je m'étais cantonné à cela, les visiteurs auraient eu les liens de cachés mais en trouvant le bon *slug* ou la bonne URL, il aurait été possible d'accéder aux pages d'administration. Pour protéger les pages d'un accès involontaire d'un visiteur à ces pages, j'ai créé un fichier *redirect.php*. Dans ce code :
Si la variable *$_SESSION['role']* n'a pas été initialisée ou si le rôle n'est pas « admin », l'utilisateur est automatiquement redirigé vers la page d'accueil.

```
<?php
if (!isset($_SESSION['role']) || $_SESSION['role'] !== 'admin') {
    // Redirigez l'utilisateur vers la page d'accueil
    header('Location:
http://localhost/kgb/index.php?controller=page&action=home');
    exit();
}
?>
```

J'appelle cette fonction à chaque page où c'est nécessaire.

```
<?php
    require_once _ROOTPATH_.'/templates/header.php';
    require_once _ROOTPATH_.'/templates/redirect.php';
?>
```

Les pages critiques sont donc uniquement accessibles par les administrateurs.

Conclusion :

Après avoir conçu les diagrammes de cas d'utilisation et de séquences, j'ai conçu par la méthode MERISE les modèles conceptuel, logique et physique de la base de données. J'ai construit la base de données en utilisant les requêtes SQL appropriées.

J'ai construit l'application KGB en PHP natif via une architecture MVC. Elle permet à un visiteur non enregistré d'atterrir sur la page d'accueil, de lister et parcourir chaque mission. En plus de ces 3 possibilités, l'administrateur pourra créer, lister, voir, mettre à jour et supprimer chaque mission, agent, contact, cible et planque.

2. Précisez les moyens utilisés. Expliquez tout ce dont vous avez eu besoin pour réaliser vos tâches : langages de programmation, frameworks, outils, logiciels, documentations techniques, etc...

Pour réaliser cette évaluation j'ai utilisé l'environnement de développement VSCode avec les plugins nécessaires pour le PHP (*PHP extension pack* et *PHP Getters and Setters*).
J'ai donc utilisé PHP 8.1.1 et la programmation orientée objet de PHP pour construire les différentes classes l'application. J'ai construit l'application en utilisant la structure MVC (*Model Vue Controller*).
J'ai utilisé le HTML pour créer les vues.
J'ai utilisé la librairie Bootstrap pour la mise en forme de la partie *Front-end* du site.
J'ai utilisé le Javascript pour dynamiser le site, qu'il soit réactif aux interactions utilisateur.
J'ai utilisé Xammp comme serveur web local pour réaliser et tester le site.
J'ai Utilisé Excalidraw pour concevoir la base de données.
J'ai utilisé le langage SQL et le logiciel Beekeeper Studio pour construire la base de données.

3. Contexte. Les noms des organismes, entreprises ou associations, dans lesquels vous avez exercé vos pratiques
   NB: Pour le cas des exercices et évaluations demandées sur la plateforme Studi, il s'agit de...Studi.

Ce projet a été réalisé dans le cadre des évaluations demandées sur la plateforme Studi.

4. Informations complémentaires (*facultatif*)

Les informations de connexion pour se logger en administrateur à l'application : mail : admin@mail.fr et mdp « root ».
En testant l'application, je n'arrive pas à visualiser les mission 1 et 4. Aucun problème avec les autres missions. Cette fonctionnalité marchait parfaitement en local.

Par manque de temps (je dois rendre très rapidement mon dossier professionnel), je n'ai pas implémenté les règles métier (correspondance pays de la mission et pays de la planque par exemple).