



ACSCAPE

DOSSIER PROJET

Titre professionnel :
Développeur Web et
Web Mobile

Julien Mathieu

Stage effectué du 16 Novembre 2022
au 13 Janvier 2023

Site Web de création et de
divertissement sur le thème des
escapes games.



Table des matières

Liste des compétences du référentiel couvertes par le projet	3
Front-end	3
Réaliser une interface web statique et adaptable	3
Réaliser une interface utilisateur web dynamique	3
Maquetter une application	3
Back-end :	4
Créer une base de données	4
Développer les composants d'accès aux données	4
Développer la partie back-end d'une application web ou web-mobile.....	5
Présentation du projet	6
Spécificités techniques.....	6
Réalisations significatives	8
BACK-END	11
Première étape, Composer – Autoloader :.....	12
Seconde étape, le routeur :.....	15
Troisième étape, création des controllers (contrôleur) :.....	19
Quatrième étape, la connexion à la base de donnée MySQL :	23
Cinquième étape, la création des modèles :	25
Sixième étape, les utilisateurs :.....	33
Front-End.....	39
Le jeu :.....	47
Présentation du jeu d'essai sur une fonctionnalité représentative.	54
Description de la veille effectuée pour le projet.	57
Extrait d'un site anglophone utilisé dans le cadre d'une recherche	58
Conclusion	61
Annexe	62

Remerciements

Je tiens à exprimer ma profonde gratitude envers toutes les personnes qui ont contribué à mon parcours de formation en tant que développeur web. Mes collègues de formation ont été un soutien indéfectible, apportant motivation et solidarité avec le pair-programming tout au long de cette expérience enrichissante.

Je tiens également à remercier chaleureusement mon coach, Alain Merruci, pour son accompagnement et sa pédagogie efficace. Il a su s'adapter à mes besoins, vulgariser des concepts complexes et apporter la confiance nécessaire pour réussir. Il a su créer un environnement de travail bienveillant et encourageant, tout en étant exigeant lorsque c'était nécessaire. Il a donné à chacun d'entre nous l'envie d'être curieux et de réussir. Je lui suis extrêmement reconnaissant pour tout ce qu'il a fait pour moi.

Je tiens également à remercier OnlineFormaPro pour m'avoir proposé et accueilli dans cette formation, ainsi que la région Bourgogne-Franche-Comté pour le financement de cette formation. C'est grâce à leur soutien que j'ai pu réaliser ce projet et je leur en suis profondément reconnaissant.

Je vous souhaite une bonne lecture.

Liste des compétences du référentiel couvertes par le projet

Front-end

Réaliser une interface web statique et adaptable

Une interface web statique est une interface qui ne change pas, c'est-à-dire que le contenu affiché est fixe et ne varie pas en fonction de l'utilisateur ou de son action. En revanche, une interface web adaptable est capable de s'adapter à différents écrans et résolutions d'affichage, ce qui permet d'offrir une expérience utilisateur optimale sur tous les appareils. Ce qui implique donc la mise en place de mécanismes de responsive design pour gérer l'affichage de l'interface en fonction de la taille de l'écran.

L'application ACScape comporte plusieurs pages web statiques et toutes sont adaptables. Comme la page d'accueil, la page de la liste des jeux, ou encore la page de présentation d'un jeu. Le contenu de ces pages reste le même pour tous les utilisateurs, mais il peut évoluer en fonction de l'arrivée de nouveaux jeux, nouvelles règles, etc.

Réaliser une interface utilisateur web dynamique

Une interface utilisateur web dynamique est une interface qui change en fonction de l'utilisateur ou de l'action de l'utilisateur. Elle peut par exemple afficher des données différentes en fonction des choix de l'utilisateur

Sur ACScape, il est possible de créer son propre jeu.

Ses fonctionnalités ne sont accessibles seulement si l'utilisateur est enregistré et connecté. Le contenu proposé sera propre à chaque utilisateur. Il est possible de lire, créer, modifier et supprimer uniquement le contenu qui nous est propre.

La page de jeu est accessible à tout utilisateur connecté ou non, mais son contenu est dépendant du jeu choisi.

Les données du jeu sont importés au format JSON afin de pouvoir interagir dynamiquement avec en Javascript pour effectuer le rendu du jeu sans rechargement de page.

Maquetter une application

Mon tuteur m'a fourni une maquette Figma comprenant la version bureau, j'avais de mon côté à réaliser la version mobile. (voir annexe).

Back-end :

Créer une base de données

Une base de données est un système de gestion de données qui permet de stocker, de classer et de manipuler des informations de manière organisée et structurée. Elle est constituée de tables qui stockent les données, de schémas qui définissent la structure des tables et de relations qui permettent de lier les données entre elles.

Elle permet de sécuriser et de protéger les données, en mettant en place des contrôles d'accès et des procédures de sauvegarde.

Dans le cadre de ce projet, j'ai créé une base de données pour stocker les informations nécessaires à l'application.

Pour cela, j'ai utilisé l'outil de gestion de bases de données MySQL, phpMyAdmin.

Avant de mettre en place la base de données, j'ai réalisé un modèle conceptuel de données (MCD) et un modèle logique de données (MLD), qui a permis de définir les structures de tables et les relations entre elles.

Développer les composants d'accès aux données

Dans le cadre de ce projet, j'utilise une architecture MVC (Modèle-Vue-Contrôleur) en PHP, afin de structurer le développement de l'application et de séparer les différentes couches de l'application (données, logique métier, présentation).

Pour construire cette architecture, je suis "parti de zéro" et j'ai développé les différents composants de l'application de manière autonome, sans utiliser de framework tels que Laravel ou Symfony. Cela m'a permis de mieux comprendre le fonctionnement interne de l'architecture MVC.

Voici comment est structurée l'application en utilisant cette architecture :

- **Les modèles** représentent les objets de données de l'application et gèrent les opérations de persistance (lecture, écriture, suppression) dans la base de données.
- **Les vues** sont les éléments de présentation de l'application, qui affichent les données aux utilisateurs et gèrent la mise en forme de l'interface utilisateur.
- **Les contrôleurs** sont les éléments de logique métier de l'application, qui gèrent les interactions avec l'utilisateur et coordonnent les actions de l'application.

En développant l'architecture MVC en PHP "from scratch", j'ai dû mettre en place des solutions pour gérer les différents aspects de l'application (routage, gestion de l'état, sécurité, etc.).

Développer la partie back-end d'une application web ou web-mobile

Pour permettre à l'utilisateur de manipuler les données, comme pour la création d'un jeu, il a fallu concevoir un CRUD (Create, Read, Update, Delete).

Pour concevoir un CRUD, il faut généralement :

- Définir les données à manipuler et la structure de la base de données
- Mettre en place les modèles qui gèrent les opérations de création, de lecture, de modification et de suppression dans la base de données
- Mettre en place les contrôleurs qui gèrent les interactions avec l'utilisateur et coordonnent les actions du CRUD
- Mettre en place les vues qui affichent les données et permettent à l'utilisateur de les manipuler

Présentation du projet

En télétravail pour une période de 8 semaines avec une durée hebdomadaire de 35h chez OnlineFormaPro, j'avais pour projet de créer un site web inspiré du site : rakura.fr

Le projet a pour nom : ACScape, son but est de proposer aux utilisateurs la possibilité de créer leur propre jeu, et de jouer aux jeux présents sur le site.

Ces jeux sont ce que l'on appelle des Escapes Games.

Dans ACScape, les jeux sont de types textuels. Le but est de trouver un code à l'aide d'énigme(s), de suggestion(s), et d'indice(s) pour déverrouiller une salle jusqu'à ouvrir la dernière salle et gagner la partie.

Je devais créer les parties Back-end et Front-end depuis zéro.

- Une partie Back en PHP en programmation orienté objet avec une architecture Modèle Vue Contrôleur, comprenant l'enregistrement et la connexion de l'utilisateur, ainsi qu'un CRUD (Create, Read, Update, Delete) pour l'administration des jeux.
- Une partie Front, où je devais intégrer la maquette Figma réalisée par mon tuteur. La partie jeu est essentiellement développée avec Javascript.

J'ai principalement travaillé en autonomie, en faisant des points réguliers avec mon tuteur.

J'ai au fur et à mesure de l'avancée du projet pu comprendre l'importance de la maintenabilité d'un projet afin d'en assurer son évolution.

Spécificités techniques

ACScape utilise les technologies suivantes :

- **HTML** (HyperText Markup Language) et **CSS** (Cascading Style Sheets) pour structurer et mettre en forme le contenu du site. J'ai utilisé la version "vanilla" de CSS en complément de **Bootstrap5**.
- **JavaScript** pour ajouter de la dynamique et de l'interactivité au site. Il s'agit également de la version "vanilla" de JavaScript.
- **PHP 8** (PHP: Hypertext Preprocessor) pour développer la partie back-end en POO, en suivant une architecture MVC « from scratch »
- **Bootstrap 5**, un framework de développement front-end qui offre un ensemble de composants prédéfinis et de règles de mise en **forme pour faciliter la création de sites web adaptables et responsives**. Bootstrap a grandement facilité l'adaptation aux différents formats d'écran en évitant de faire une importante quantité de « media queries »

à la main.

- **Iconify**, une librairie d'icônes en ligne qui permet de choisir parmi un grand nombre d'icônes vectorielles. J'ai utilisé Iconify pour ajouter de la variété et de la personnalisation à l'interface utilisateur, et pour améliorer l'expérience de l'utilisateur en proposant des éléments visuels uniques et pertinents. Iconify a permis de facilement intégrer les icônes au site en utilisant un simple balisage HTML et en chargeant les icônes depuis leur serveur en ligne
- **GitHub**, une plateforme de développement qui offre un service de gestion de versions en ligne basé sur Git. GitHub offre également des fonctionnalités de collaboration, de suivi de bugs et de gestion de tâches qui nous ont été utiles pour le développement. »
- **Figma**, un éditeur de graphiques vectoriels et un outil de prototypage. Il est principalement basé sur le web

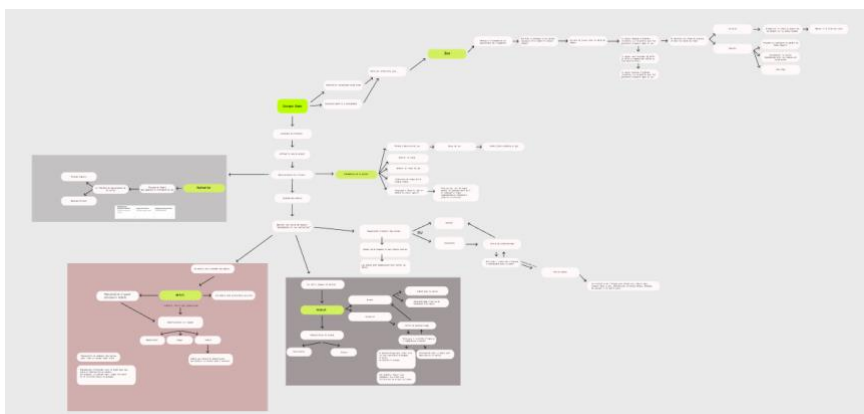
Réalisations significatives

Cette partie représente les réalisations que j'ai dû faire pour ACScape. Elle est découpée en 3 partie. Mise en place du projet, back-end, et front-end.

Mise en place du projet :

Après avoir pris connaissance du projet, et pour en apprendre plus sur le projet j'ai étudié le site dont s'inspire le projet : **rakura.fr**

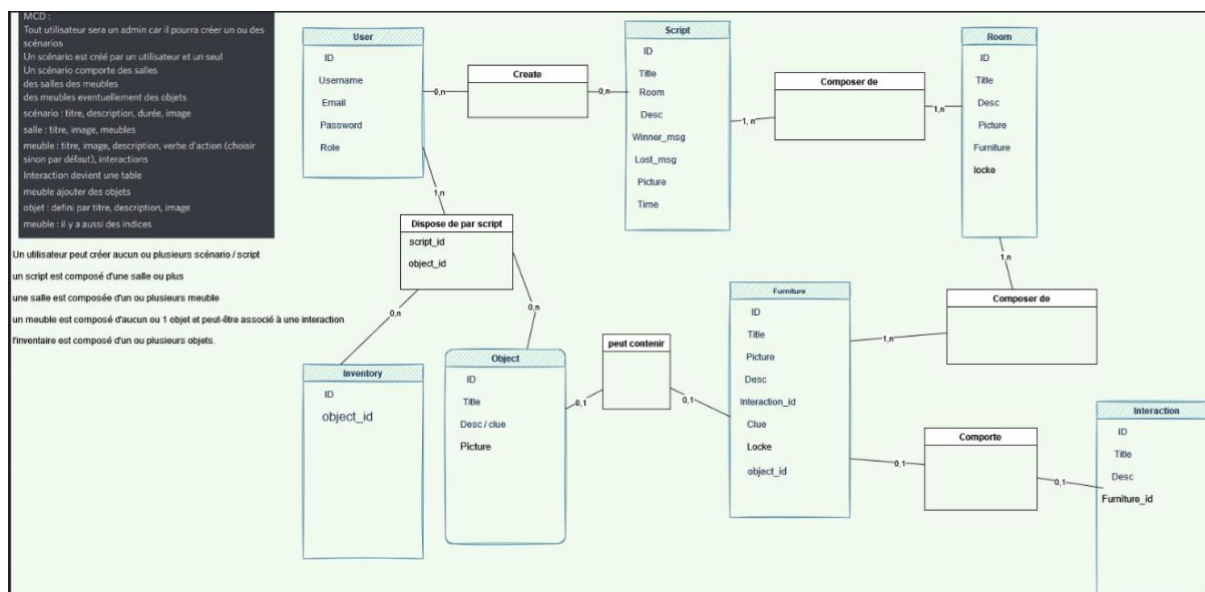
Suite à cette analyse, je réalise un schéma récapitulatif des actions à intégrées.



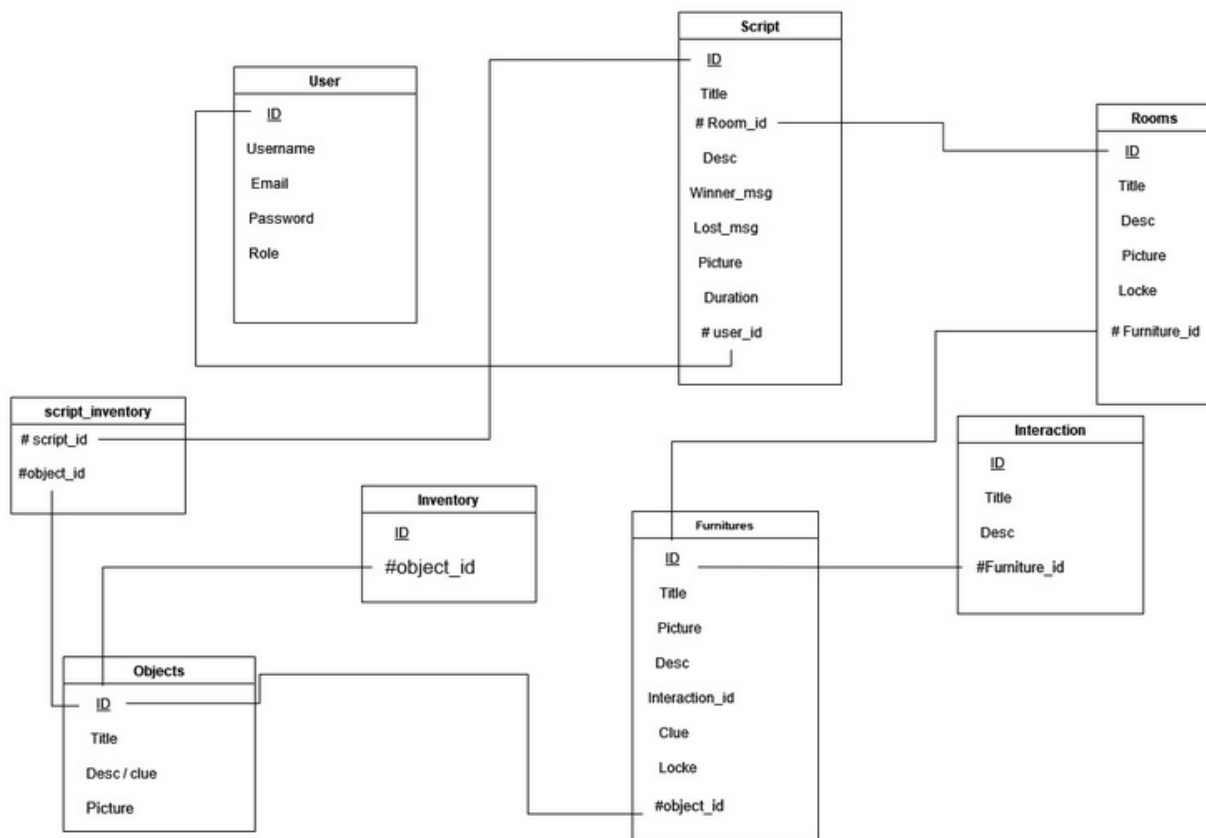
Il est nécessaire de définir les besoins de la BDD (base de données), avant de commencer à coder.

J'ai utilisé l'outil en ligne **app.diagrams.net**. pour réaliser les diagrammes.

J'élabore dans un premier temps le MCD (Modèle Conceptuel des Données).



Puis le MLD (Modèle Logique des Données)



Users :

id Integer PrimaryKey
 Username Varchar(255) not Null Unique
 Email Varchar(255) not Null Unique
 Password Varchar(25) not Null
 Role Enum(user,admin)

Inventory
 id Integer PrimaryKey
 #object_id Foreign Key Reference Objects(id)

Scripts

id Integer PrimaryKey
 Title varchar(255) not Null
 #Room_id integer Foreign Key Reference Rooms(id)
 Desc Text not Null
 Winner_msg varchar(255)
 Lost_msg varchar(255)
 Picture Nullable
 Duration Integer Default 30 Not Null
 #User_id Foreign Key Reference Users(id)

Objects

id Integer PrimaryKey
 Title varchar(255) not Null
 Desc text not Null
 Picture Nullable

Script_inventory

#script_id Foreign Key Reference Scripts(id)
 #Object_id Foreign Key Reference Objects(id)

Rooms

id Integer PrimaryKey
 Title varchar(255) not Null
 Desc Text not Null
 Picture Nullable
 Locke enum(yes,no) Default no Not Null
 #Furniture_id Foreign Key Reference Furnitures(id)

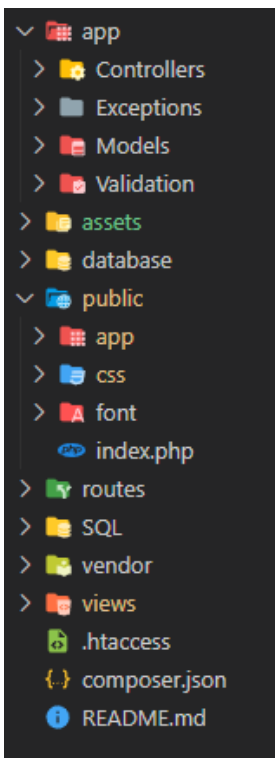
Furnitures

id Integer Primary Key
 Title varchar(255)
 Picture Nullable
 Desc Text
 #interaction_id Foreign Key Reference Interactions(id)
 Clue text nullable
 Locke enum(yes,no) Default no Not Null
 #object_id Foreign Key Reference Objects(id)

Interactions

id integer Primary Key
 Title varchar(255)
 Desc varchar(255)
 #Furniture_id Foreign Key Reference Furnitures(id)

MCD et MLD validés, il est à présent temps de commencer l'architecture Modèle-vue-contrôleur ou MVC. Pour utiliser de manière optimal l'architecture MVC, il faut mettre en place des solutions pour gérer les différents aspects de l'application (autoloader, routage, sécurité, etc.)



Ce qui nous donnera une arborescence finale composée de ces dossiers.

Pour l'essentiel :

Dans Database, les fichiers de connexion à la base de données.

Dans app, nous retrouvons les controllers et les models.

Dans public, les routes, ainsi que les fichiers CCS et JS

Et dans views, nous retrouvons les vues.

BACK-END

Première étape, Composer – Autoloader :

Dans un premier temps, nous allons installer l'autoloader de Composer à l'aide du terminal et de la commande « composer init »

```
PS C:\wamp64\www\acscape> composer init
```

```
Welcome to the Composer config generator
```

```
This command will guide you through creating your composer.json config.
```

```
Package name (<vendor>/<name>) [champ/acscape]:
```

```
Description []:
```

```
Author [GamerBike39 <*****>, n to skip]:
```

```
Minimum Stability []:
```

```
Package Type (e.g. library, project, metapackage, composer-plugin) []:
```

```
License []:
```

```
Define your dependencies.
```

```
Would you like to define your dependencies (require) interactively [yes]?
```

```
Search for a package:
```

```
Would you like to define your dev dependencies (require-dev) interactively [yes]?
```

```
Search for a package:
```

```
Add PSR-4 autoload mapping? Maps namespace "Champ\acscape" to the entered relative path.
```

```
[src/, n to skip]:
```

```
{
    "name": "champ/acscape",
    "autoload": {
        "psr-4": {
            "Champ\\acscape\\": "src/"
        }
    },
    "authors": [
        {
            "name": "GamerBike39",
            "email": "*****"
        }
    ],
    "require": {}
}
```

A cet instant, Composer crée un fichier composer.json à la racine du projet dans lequel je vais définir un autoload.



```
{
  "name": "33642/acscape",
  "authors": [
    {
      "name": "gamerbike",
      "email": "*****@gmail.com"
    }
  ],
  "autoload": {
    "psr-4": {
      "Router\\": "routes/",
      "App\\": "app/",
      "Database\\": "database/"
    }
  },
  "require": {}
}
```

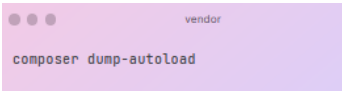
j'utilise **psr-4** (PHP Standards Recommendations.)

Cet autoload a pour rôle d'associer un dossier à un namespace spécifique.

Pour un bon fonctionnement, il est important que le chemin des classes dans le namespace corresponde au nom du namespace.

La recommandation **psr-4** permet à des outils comme Composer de fonctionner et de générer un autoloader à partir des dépendances chargées.

On régénère la liste de toutes les classes qui doivent être incluses dans le projet avec la commande « **composer dump-autoload** » dans le terminal



```
composer dump-autoload
```

composer va générer un dossier « **vendor** » qui comportera un fichier autoload.php qu'il faudra « **require** » dans index.php du dossier public.

```
require '../vendor/autoload.php';
```

Grâce à cet **autoload**, il sera inutile de faire un « **require** » de fichier pour instancier des classes.

Pour respecter les bonnes pratiques, il est recommandé de mettre le fichier index.php dans un dossier public car cela permet de séparer les fichiers accessibles directement à l'utilisateur de ceux qui ne doivent pas l'être.

Le dossier public est généralement utilisé pour stocker les fichiers qui sont accessibles au public, tels que les fichiers CSS, JavaScript et les images. En mettant index.php dans ce dossier, l'utilisateur ne pourra pas accéder aux autres fichiers de l'application, comme les fichiers de configuration ou les fichiers de données sensibles. Cela peut aider à protéger contre les attaques et assurer la sécurité des données.

Je crée ensuite un fichier **.htaccess**

```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^(.*)$ public/index.php?url=$1 [QSA,L]
```

Le fichier **.htaccess** est un fichier de configuration qui peut être utilisé pour modifier les comportements du serveur web Apache. Il redirige les requêtes HTTP, bloque l'accès à certains fichiers ou dossiers, et effectue d'autres opérations de gestion des URL.

La directive « **RewriteEngine On** » active le moteur de réécriture d'URL d'Apache, permettant de rediriger les requêtes vers une autre URL.

La directive « **RewriteCond** %{REQUEST_FILENAME} **!-f** » vérifie si le fichier demandé n'existe pas sur le serveur. Si le fichier n'existe pas, la réécriture d'URL est effectuée en utilisant la règle « **RewriteRule** » qui suit.

La règle « **RewriteRule** ^(.*)\$ **public/index.php?url=\$1 [QSA,L]** » indique au serveur qu'il doit rediriger toutes les requêtes ne correspondant pas à un fichier existant vers le fichier index.php dans le dossier public, en lui passant l'URL complète de la requête en tant que paramètre d'URL "url". Le paramètre [QSA,L] indique au serveur de faire suivre tous les paramètres d'URL existants (query string parameters) et de s'arrêter là (L pour Last).

En résumé, cette configuration permet de rediriger toutes les requêtes qui ne correspondent pas à un fichier existant vers le fichier index.php dans le dossier public, en lui passant l'URL complète de la requête en tant que paramètre d'URL. Cela est utile quand un fichier index.php est utilisé comme point d'entrée pour toutes les requêtes.

Seconde étape, le routeur :

Ce routeur s'inspire de celui de Laravel (framework PHP)

Je crée ensuite un dossier « routes » composé des fichiers suivant :

route.php



```
class Route {

    public $path;
    public $action;
    public $matches;

    public function __construct($path, $action)
    {
        $this->path = trim($path, '/');
        $this->action = $action;
    }

    public function matches(string $url)
    {
        $path = preg_replace('#:([\w]+)#', '([^\./]+)', $this->path);
        $pathToMatch = "#^$path$#";

        if (preg_match($pathToMatch, $url, $matches)) {
            $this->matches = $matches;
            return true;
        } else {
            return false;
        }
    }

    public function execute()
    {
        $params = explode('@', $this->action);
        $controller = new $params[0](new DBConnection(DB_NAME, DB_HOST, DB_USER, DB_PWD));
        $method = $params[1];

        return isset($this->matches[1]) ? $controller->$method($this->matches[1]) : $controller->$method();
    }
}
```

La classe Route possède plusieurs propriétés :

- **path** : c'est le chemin de la route (par exemple, '/posts/:id' pour une route qui affiche un article en particulier).
- **action** : c'est l'action à exécuter lorsque la route est activée. Cela peut être un nom de contrôleur et de méthode séparés par un '@' (par exemple, 'PostController@show').
- **matches** : c'est un tableau contenant les parties de l'URL qui correspondent aux variables définies dans le chemin

La classe Route possède également plusieurs méthodes :

- **__construct()** : c'est le constructeur de la classe, qui est appelé lorsqu'un nouvel objet Route est créé. Il prend en entrée le chemin et l'action de la route et les stocke dans les propriétés de l'objet.

- **matches()** : cette méthode prend en entrée une URL et vérifie si elle correspond au chemin de la route. Si c'est le cas, elle stocke les parties de l'URL qui correspondent aux variables du chemin dans la propriété 'matches' et retourne 'true'. Si l'URL ne correspond pas au chemin, elle retourne 'false'.
- **execute()** : cette méthode exécute l'action de la route en appelant le contrôleur et la méthode associés. Elle peut également passer des paramètres à la méthode si des variables sont définies dans le chemin de la route.

Cette classe Route permet de gérer les URL et de déterminer quelle action doit être exécutée en fonction de l'URL demandée. Par exemple, une route '/login' qui affiche un formulaire de connexion.

Détails de la méthode **matches()** :

```
public function matches(string $url)
{
    $path = preg_replace('#:([\w]+)#', '([^\/]*)', $this->path);
    $pathToMatch = "#^$path$#";

    if (preg_match($pathToMatch, $url, $matches)) {
        $this->matches = $matches;
        return true;
    } else {
        return false;
    }
}
```

1. La première ligne de code remplace les variables du chemin de la route par des motifs de capture de chaîne. Par exemple, si le chemin de la route est '/posts:id', le motif de capture de chaîne sera '([^\/]*)', ce qui signifie qu'il peut y avoir n'importe quelle chaîne (sauf '/') à cet endroit.
2. La deuxième ligne de code crée un motif de comparaison à partir du chemin de la route modifié et de l'URL à vérifier. Le motif de comparaison est encadré par des caractères '#' pour indiquer qu'il s'agit d'un motif de type **'delimiter'**.
<https://www.php.net/manual/fr/regexp.reference.delimiters.php>
3. La fonction 'preg_match' est utilisée pour comparer l'URL à vérifier avec le motif de comparaison. Si l'URL correspond au motif, elle retourne 'true' et stocke les parties de l'URL qui correspondent aux variables du chemin de la route dans un tableau '\$matches'. Si l'URL ne correspond pas au motif, elle retourne 'false'.
<https://www.php.net/manual/fr/function.preg-match.php>
4. Si l'URL correspond au motif, la propriété 'matches' de l'objet Route est mise à jour avec le tableau '\$matches' et la fonction retourne 'true'. Si l'URL ne correspond pas au motif, la fonction retourne 'false'.

En résumé, la fonction 'matches' de la classe Route permet de vérifier si une URL donnée correspond au chemin de la route en utilisant une expression régulière. Si l'URL correspond au chemin, la fonction retourne 'true' et met à jour la propriété 'matches' de l'objet Route.

Il y aura également besoin d'une autre classe nommée Router dans le même namespace.



```
class Router {  
  
    public $url;  
    public $routes = [];  
  
    public function __construct($url)  
    {  
        $this->url = trim($url, '/');  
    }  
  
    public function get(string $path, string $action)  
    {  
        $this->routes['GET'][] = new Route($path, $action);  
    }  
  
    public function post(string $path, string $action)  
    {  
        $this->routes['POST'][] = new Route($path, $action);  
    }  
  
    public function run()  
    {  
        foreach ($this->routes[$_SERVER['REQUEST_METHOD']] as $route) {  
            if ($route->matches($this->url)) {  
                return $route->execute();  
            }  
        }  
  
        throw new NotFoundException("La page demandée est introuvable.");  
    }  
}
```

La classe Router permet de gérer les URL et de rediriger les requêtes vers les actions appropriées. Elle possède plusieurs propriétés et méthodes :

- **url** : c'est l'URL de la requête courante.
- **routes** : c'est un tableau qui contient les routes définies dans l'objet Router, organisées par méthode HTTP (GET ou POST).
- **__construct()** : c'est le constructeur de la classe, qui est appelé lorsqu'un nouvel objet Router est créé. Il prend en entrée l'URL de la requête courante et la stocke dans la propriété 'url'. <https://www.php.net/manual/en/language.oop5.decon.php>
- **get()** et **post()** : ces méthodes permettent d'ajouter des routes à l'objet Router. La méthode **get()** ajoute une route qui sera utilisée pour les requêtes **HTTP GET**, tandis que la méthode **post()** ajoute une route qui sera utilisée pour les requêtes **HTTP POST**. Les routes sont ajoutées sous forme d'objets Route, qui contiennent le chemin de la route et l'action à exécuter.
- **run()** : cette méthode parcourt les routes définies dans l'objet Router et vérifie si l'une d'entre elles correspond à l'URL de la requête courante. Si une route correspond, elle exécute l'action associée à cette route en appelant la méthode '**execute**' de l'objet Route. Si aucune route ne correspond, elle lève une exception '**NotFoundException**' indiquant que la page demandée est introuvable.

Explication détaillée de la fonction **run()**



```
public function run()
{
    foreach ($this->routes[$_SERVER['REQUEST_METHOD']] as $route) {
        if ($route->matches($this->url)) {
            return $route->execute();
        }
    }

    throw new NotFoundException("La page demandée est introuvable.");
}
```

La boucle **foreach** parcourt le tableau de routes de l'objet Router qui correspond à la méthode HTTP de la requête courante (soit GET ou POST). Pour chaque route dans le tableau, la méthode 'matches' de l'objet Route est appelée avec l'URL de la requête courante en tant que paramètre. Si la méthode '**matches**' retourne '**true**', cela signifie que l'URL de la requête correspond à la route en cours de traitement. Dans ce cas, la méthode '**execute**' de l'objet Route est appelée pour exécuter l'action associée à cette route.

La boucle **foreach** s'arrête dès qu'une route correspondante est trouvée et exécutée. Si aucune route ne correspond à l'URL de la requête, la boucle se termine et la méthode 'run' de l'objet Router lève une exception '**NotFoundException**' indiquant que la page demandée est introuvable.

Nous pouvons désormais appeler et créer un nouvel objet router dans index.php du dossier public à l'aide de : `use Router\Router;`

Nous écrivons ensuite la ligne de code : `$router = new Router($_GET['url']);`

pour créer un nouvel objet Router en passant l'URL courante en tant que paramètre. Le paramètre est généralement récupéré à partir de la superglobale PHP `$_GET['url']`, qui contient l'URL de la requête courante.

Le constructeur de la classe Router est appelé avec l'URL en entrée et peut être utilisé pour initialiser l'objet Router et préparer les routes qui seront utilisées par l'application. Par exemple, pour ajouter des routes à l'objet Router, comme ceci :

```
$router->get('/login', 'App\Controllers\UserController@login');
$router->post('/login', 'App\Controllers\UserController@loginPost');
$router->get('/logout', 'App\Controllers\UserController@logout');
$router->get('/register', 'App\Controllers\UserController@register');
$router->post('/register', 'App\Controllers\UserController@registerPost');
```

En bas du fichier index.php j'exécute la méthode `run()` dans un **bloc try-catch**

```
try {
    $router->run();
} catch (NotFoundException $e) {
    return $e->error404();
}
```

Le **bloc try-catch** est utilisé ici pour gérer les erreurs qui peuvent survenir lors de l'exécution de l'objet Router, comme l'absence de route correspondant à l'URL de la requête. Cela permet de protéger l'application contre les erreurs inattendues et de gérer proprement ces erreurs en affichant une page d'erreur ou en redirigeant vers une autre page.

Troisième étape, création des controllers (contrôleur) :

Qu'est-ce qu'un contrôleur :

En architecture MVC, le contrôleur est un composant qui gère les interactions entre l'utilisateur, le modèle (qui gère les données de l'application) et la vue (qui affiche les données à l'utilisateur). Le contrôleur joue le rôle de "pont" entre l'utilisateur et le reste de l'application, en recevant les requêtes de l'utilisateur, en demandant au modèle de récupérer ou de mettre à jour les données nécessaires, et en envoyant ces données à la vue pour être affichées à l'utilisateur.

Voici les principales responsabilités d'un contrôleur dans l'architecture MVC :

- **Gérer les requêtes de l'utilisateur** : le contrôleur reçoit les requêtes de l'utilisateur (par exemple, une requête HTTP GET pour afficher une page) et détermine quelle action doit être exécutée en fonction de l'URL de la requête.
- **Interagir avec le modèle** : le contrôleur peut demander au modèle de récupérer ou de mettre à jour des données en utilisant des méthodes du modèle (par exemple, une méthode 'getPosts' pour récupérer tous les articles d'un blog).
- **Préparer les données pour la vue** : le contrôleur peut préparer les données récupérées du modèle de manière à ce qu'elles soient facilement utilisables par la vue (par exemple, en organisant les données sous forme de tableaux ou d'objets).
- **Appeler la vue** : le contrôleur appelle la vue avec les données préparées et la vue affiche ces données à l'utilisateur.

En résumé, le contrôleur joue un rôle central dans l'architecture MVC en gérant les interactions entre l'utilisateur, le modèle et la vue et en s'assurant que les données sont correctement affichées à l'utilisateur.

Pour des fonctions, requêtes récurrentes, j'utilise le principe d'héritage. Afin d'optimiser le code et augmenter la maintenabilité.

Ces fonctions se trouveront dans, Controller.php et Model.php
Il étendra l'ensemble des contrôleurs à l'aide de « extends »

Comme ceci par exemple : `class RoomController extends Controller {`

Voyons comment est composée la classe abstraite Controller



```
<?php

namespace App\Controllers;

use Database\DBConnection;

abstract class Controller {

    protected $db;

    public function __construct(DBConnection $db)
    {
        if (session_status() === PHP_SESSION_NONE) {
            session_start();
        }

        $this->db = $db;
    }

    public function generateCsrfToken() {
        $random_id = bin2hex(random_bytes(32));
        $csrf_token = hash_hmac('sha256', $random_id, 'secret_key');
        return $csrf_token;
    }

    public function validateCsrfToken($csrf_token) {
        $stored_token = $_COOKIE['csrf_token'];
        if ($csrf_token === $stored_token) {
            return true;
        } else {
            return false;
        }
    }

    protected function view(string $path, array $params = null)
    {
        ob_start();
        $path = str_replace('.', DIRECTORY_SEPARATOR, $path);
        require VIEWS . $path . '.php';
        $content = ob_get_clean();
        require VIEWS . 'layout.php';
    }

    protected function json(string $path, array $data)
    {
        $path = str_replace('.', DIRECTORY_SEPARATOR, $path);
        require VIEWS . $path . '.php';
        header('Content-Type: application/json');
        echo json_encode( $data );
        exit();
    }

    protected function getDB()
    {
        return $this->db;
    }

    protected function isAdmin()
    {
        if (isset($_SESSION['auth']) && $_SESSION['auth'] === 1) {
            return true;
        } else {
            return header('Location: login');
        }
        if ($_COOKIE['csrf_token'] != $_SESSION['csrf']) {
            return header('Location: /login?error=session_expired');
        }
    }
}
```

La classe 'Controller' est une classe abstraite fournissant des méthodes communes aux contrôleurs de l'application. Elle définit une interface pour les contrôleurs, qui doivent étendre de cette classe et implémenter ses méthodes.

Voici les principales méthodes de la classe 'Controller' :

- **__construct()** : Il prend en entrée un objet '**DBConnection**' qui représente une connexion à la base de données et le stocke dans la propriété '**db**'. Si la session n'est pas démarrée, il la démarre.
- **view()** : cette méthode permet de charger une vue et de l'afficher à l'utilisateur. Elle prend en entrée le chemin de la vue à charger et un tableau de paramètres à passer à la vue. La vue est chargée dans un tampon de sortie, puis le contenu du tampon est affiché à l'utilisateur en utilisant le **layout** de l'application (un fichier 'layout.php' qui contient le code HTML de base de l'application, en-tête, contenu et pied de page).
- **json()** : cette méthode permet de charger une vue et de renvoyer les données sous forme de JSON à l'utilisateur. Elle prend en entrée le chemin de la vue à charger et un tableau de données à encoder en JSON les données sont renvoyées avec un en-tête 'Content-Type: application/json'.
- **getDB()** : cette méthode retourne l'objet '**DBConnection**' associé au contrôleur.
- **isAdmin()** : cette méthode vérifie si l'utilisateur est connecté en tant qu'administrateur. Si c'est le cas, elle retourne '**true**', sinon elle redirige l'utilisateur vers la page de connexion.

Arrêtons-nous quelques instants sur la méthode **view()** et décrivons son fonctionnement en 5 étapes :

1. La méthode '**ob_start**' démarre un tampon de sortie. Cela signifie que toutes les données envoyées à la sortie (par exemple, du code HTML) seront mises en tampon au lieu d'être immédiatement affichées à l'utilisateur.
<https://www.php.net/manual/fr/function.ob-start.php>
2. Le chemin de la vue est modifié pour remplacer les points '.' par des séparateurs de répertoire (**DIRECTORY_SEPARATOR**). Cela permet de définir le chemin de la vue de manière à ce qu'il puisse être utilisé sur n'importe quel système d'exploitation (les séparateurs de répertoire étant différents sur Windows et sur Unix).
3. La vue est chargée à l'aide de la fonction '**require**'. Le chemin de la vue est préfixé par le chemin des vues de l'application (VIEWS) pour indiquer où trouver la vue sur le serveur.
4. La méthode '**ob_get_clean**' récupère le contenu du tampon de sortie et le vide. Cela signifie que le contenu de la vue est récupéré et stocké dans une variable '\$content'.
5. Le **layout** de l'application est chargé à l'aide de la fonction '**require**'. Le **layout** est un fichier '**layout.php**' qui contient le code HTML de base de l'application (en-tête, pied de page, etc.).
6. La méthode '**view**' se termine et le contenu de la vue est affiché à l'utilisateur avec le **layout** de l'application.

Pour utiliser la méthode **view()**, j'ai besoin de créer un dossier **views** avec un fichier **layout.php**.

Qu'est-ce qu'un layout ?

Le layout est un fichier qui contient le code HTML de base de l'application, c'est-à-dire le code HTML qui est commun à toutes les pages de l'application (en-tête, pied de page, etc.). Le layout est généralement chargé une seule fois et utilisé comme "cadre" autour des vues de l'application, qui sont chargées à l'intérieur du layout dans la variable « \$content »

Comme ceci :

```
<div class="container-fluid gx-0 back_dark flex-grow-1 position-relative ingame_container_background">
    <?= $content ?>
</div>
```

Quatrième étape, la connexion à la base de donnée MySQL :

Avant tout, pour se connecter à une base de données, il faut la créer.

Pour ce projet, j'ai utilisé **Wamp**, logiciel qui permet de générer un serveur Apache en local, et **phpmyadmin** pour la création de la base de donnée.

Table	Action	Lignes	Type	Interclassement	Taille	Perte
<input type="checkbox"/> furnitures	★	15	InnoDB	latin1_swedish_ci	80,0 kio	-
<input type="checkbox"/> interactions	★	0	InnoDB	latin1_swedish_ci	64,0 kio	-
<input type="checkbox"/> objects	★	8	InnoDB	latin1_swedish_ci	48,0 kio	-
<input type="checkbox"/> rooms	★	16	InnoDB	latin1_swedish_ci	64,0 kio	-
<input type="checkbox"/> scripts	★	7	InnoDB	latin1_swedish_ci	32,0 kio	-
<input type="checkbox"/> users	★	6	InnoDB	latin1_swedish_ci	48,0 kio	-
6 tables	Somme	52	MvISAM	latin1_swedish_ci	336,0 kio	0 o

Une fois la base de données créée je peux établir la connexion à la BDD avec le fichier DBConnection.php dans le dossier database.

Je crée une classe **DBConnection**.



```
class DBConnection {  
  
    private $dbname;  
    private $host;  
    private $username;  
    private $password;  
    private $pdo;  
  
    public function __construct(string $dbname, string $host, string $username, string $password)  
    {  
        $this->dbname = $dbname;  
        $this->host = $host;  
        $this->username = $username;  
        $this->password = $password;  
    }  
  
    public function getPDO(): PDO  
    {  
        return $this->pdo ?? $this->pdo = new PDO("mysql:dbname={$this->dbname};host={$this->host}",  
$this->username, $this->password, [  
            PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,  
            PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_OBJ,  
            PDO::MYSQL_ATTR_INIT_COMMAND => 'SET CHARACTER SET UTF8'  
        ]);  
    }  
}
```

La classe '**DBConnection**' est une classe qui permet de créer et de gérer une connexion à une base de données. Elle utilise la classe '**PDO**' (PHP Data Objects) de PHP pour effectuer des requêtes SQL sur la base de données.

Voici les principaux éléments de cette classe :

- **\$dbname, \$host, \$username, \$password** : ce sont des propriétés qui contiennent respectivement le nom de la base de données, l'adresse du serveur de base de données, le nom d'utilisateur et le mot de passe pour se connecter à la base de données.
- **\$pdo** : c'est une propriété qui contient l'objet '**PDO**' associé à la connexion à la base de données. Elle est initialisée à 'null' par défaut.
<https://www.php.net/manual/fr/class.pdo.php>
- **getPDO()** : cette méthode retourne l'objet '**PDO**' associé à la connexion à la base de données. Si l'objet n'a pas encore été créé, il est créé à l'aide des informations de connexion stockées dans les propriétés de l'objet '**DBConnection**'. La méthode utilise l'opérateur '??' pour retourner l'objet '**PDO**' si elle existe, sinon elle le crée et le stocke dans la propriété '**pdo**' avant de le retourner. Cela permet de ne créer l'objet '**PDO**' qu'au premier appel de la méthode '**getPDO**', ce qui peut être utile pour éviter de surcharger inutilement le serveur de base de données.

En résumé, la classe '**DBConnection**' permet de créer et de gérer une connexion à une base de données en utilisant l'objet '**PDO**' de PHP. Elle fournit une méthode '**getPDO**' qui retourne l'objet '**PDO**' associé à la connexion, créé au besoin.

Cinquième étape, la création des modèles :

Maintenant que notre routeur et la connexion à la base de données sont créés, je peux désormais préparer des requêtes SQL, à l'aide de PDO disponible dans les contrôleurs.

J'ai évoqué précédemment des fonctions, requêtes récurrentes. Nous allons voir comment faire ça avec les modèles.

En fonctions récurrentes, nous avons par exemple, les tâches du CRUD (Create, Read, Update, Delete).

Pour éviter de se répéter dans chacune des classes, je crée une classe parent Model.php contenant les fonctions génériques



```
abstract class Model {  
  
    protected $db;  
    protected $table;  
  
    public function __construct(DBConnection $db)  
    {  
        $this->db = $db;  
    }  
  
    public function all()  
    {  
        return $this->query("SELECT * FROM {$this->table} ORDER BY id DESC");  
    }  
}
```

Ce code définit une classe abstraite appelée **Model**, qui est destinée à être étendue par d'autres classes de modèle.

La classe **Model** possède deux propriétés protégées: **\$db** et **\$table**. La propriété **\$db** est une instance de la classe **DBConnection**, qui représente une connexion à une base de données. La propriété **\$table** est le nom de la table de base de données associée au modèle.

La classe **Model** définit également un constructeur qui prend en paramètre une instance de **DBConnection** et l'enregistre dans la propriété **\$db**.

Enfin, la classe **Model** définit une méthode **all()** qui retourne tous les enregistrements de la table associée au modèle, triés par ordre décroissant de leur identifiant (**id**). Cette méthode **all()** sera utilisée dans tous les contrôleurs.



```
public function findById(int $id): Model  
{  
    return $this->query("SELECT * FROM {$this->table} WHERE id = ? ", [$id],  
true);  
}
```

Il est important de définir la méthode **query()** pour mieux comprendre ce qui se passe.



```
public function query(string $sql, array $param = null, bool $single = null)
{
    $method = is_null($param) ? 'query' : 'prepare';

    if (
        strpos($sql, 'DELETE') === 0
        || strpos($sql, 'UPDATE') === 0
        || strpos($sql, 'INSERT') === 0 ) {

        $stmt = $this->db->getPDO()->$method($sql);
        $stmt->setFetchMode(PDO::FETCH_CLASS, get_class($this), [$this->db]);
        return $stmt->execute($param);
    }

    $fetch = is_null($single) ? 'fetchAll' : 'fetch';

    $stmt = $this->db->getPDO()->$method($sql);
    $stmt->setFetchMode(PDO::FETCH_CLASS, get_class($this), [$this->db]);

    if ($method === 'query') {
        return $stmt->$fetch();
    } else {
        $stmt->execute($param);
        return $stmt->$fetch();
    }
}
```

La méthode **query()** est une méthode générique qui permet d'exécuter une requête SQL sur la base de données associée au modèle. Elle prend en entrée une chaîne de caractères **\$sql** qui représente la requête à exécuter, ainsi qu'un tableau **\$param** (optionnel) qui contient les paramètres de la requête. Elle prend également en entrée un booléen **\$single** (optionnel) qui indique si on souhaite récupérer un seul enregistrement ou l'ensemble des enregistrements correspondant à la requête.

La méthode détermine d'abord si la requête est une requête de modification (**INSERT**, **UPDATE**, **DELETE**) ou une requête de sélection (**SELECT**). Si c'est une requête de modification, elle prépare et exécute la requête en utilisant la méthode **prepare()** de l'objet **PDO** associé à la base de données, et retourne le résultat de l'exécution (**true** si la requête a été exécutée avec succès, **false** sinon). Si c'est une requête de sélection, la méthode prépare et exécute la requête en utilisant soit la méthode **query()** (si **\$param** est **null**) soit la méthode **prepare()** de l'objet **PDO**, puis retourne soit tous les enregistrements correspondant à la requête (si **\$single** est **null**) soit un seul enregistrement (si **\$single** est **true**).

En résumé, la méthode **query()** permet d'exécuter n'importe quelle requête SQL sur la base de données associée au modèle, et de retourner le résultat sous forme d'objets de la classe courante (ou de son extension).

Revenons un instant sur les contrôleurs.

A partir des méthodes **all()**, et **findById()**, je peux à présent, m'en servir dans les contrôleurs et envoyé le résultat à une vue. Avec une route définie dans index.php

Prenons les exemples pour la page d'accueil :



```
public function index()  
{  
    $script = new Script($this->getDB());  
    $script = $script->all();  
    return $this->view('home.index', compact('script'));  
}
```

cette fonction envoie à la vue index, l'ensemble des enregistrements dans la table scripts dans un tableau **\$params**



```
public function show(int $id)  
{  
    $script = new Script($this->getDB());  
    $script = $script->findById($id);  
    $scriptAll = $script->all();  
    return $this->view('home.show', compact('script', 'scriptAll'));  
}
```

Cette fonction nous permet d'envoyer à la vue show.php l'enregistrement qui correspond à l'**id** que l'on souhaite. Généralement après un clic sur l'élément ou une recherche.

Pour enfin afficher les vues, il faut les définir à travers le routeur.

Dans index.php, j'inscris donc ces lignes :



```
$router->get('/index', 'App\Controllers\BlogController@index');  
$router->get('/show/:id', 'App\Controllers\BlogController@show');
```

Je fais donc appel à l'objet **router** qui attends **une url, un contrôleur et sa méthode**. Ces deux derniers sont séparé par un @

Avec la méthode **execute()** de la classe **Route**, les enregistrements sont retournées à la vue sous forme d'un tableau d'objet disponible avec la variable **\$params**.



```
$scripts = $params['script'];
```

Par exemple, dans la vue index, dans laquelle tous les enregistrements de la table « scripts » sont retournés. Je stocke dans la variable \$scripts ce que me renvoie la méthode all exécuté par

le contrôleur.

Avec une boucle **ForEach** il est possible de parcourir tous les éléments qui nous sont renvoyés.



```
<?php foreach ($scripts as $game) : ?>
    <div class="game col-10 col-md-3">
        
        <div class="card_game_content">
            <div class="title_game">
                <p class="m-0"><?= $game->title ?></p>
            </div>
            <div class="parameters_game d-flex align-items-center">
                <div class="diffucilty" data-difficulty="<?=
$game->difficulty ?>">
                    <iconify-icon data-id="<?= $game->id ?>"
icon="ri:lock-line"></iconify-icon>
                    <iconify-icon data-id="<?= $game->id ?>"
icon="ri:lock-line"></iconify-icon>
                    <iconify-icon data-id="<?= $game->id ?>"
icon="ri:lock-line"></iconify-icon>
                    <iconify-icon data-id="<?= $game->id ?>"
icon="ri:lock-line"></iconify-icon>
                    <iconify-icon data-id="<?= $game->id ?>"
icon="ri:lock-line"></iconify-icon>
                </div>
                <div class="time_game d-flex justify-content-center
align-items-center gap-3">
                    
                    <p class="m-0 white"><?= $game->duration?></p>
                </div>
            </div>
            <a class="link_game" href="/show/<?= $game->id ?>"></a>
        </div>
    </div>
<?php endforeach; ?>
```

Je peux dans la boucle accéder aux valeurs de l'objet et les afficher avec « **echo** »

ici par exemple pour afficher le titre. `<p class="m-0"><?= $game->title ?></p>`

\$game est défini à l'initiation de la boucle **foreach** de cette manière

```
<?php foreach ($scripts as $game) : ?>
```

Revenons dans Model.php

Regardons désormais la méthode **create()**



```
public function create(array $data, ?array $relations = null)
{
    $firstParenthesis = "";
    $secondParenthesis = "";
    $i = 1;

    foreach ($data as $key => $value) {
        $comma = $i === count($data) ? "" : ", ";
        $firstParenthesis .= "{$key}{$comma}";
        $secondParenthesis .= ":{key}{$comma}";
        $i++;
    }

    return $this->query("INSERT INTO {$this->table} ($firstParenthesis)
VALUES($secondParenthesis)", $data);
}
```

La méthode **create()** est une méthode qui permet d'insérer un nouvel enregistrement dans la table associée au modèle. Elle prend en entrée un tableau **\$data** qui contient les données à insérer, sous forme de couples clé-valeur, et un tableau **\$relations** (optionnel) qui contient des informations sur les relations entre les différentes tables de la base de données.

La méthode commence par boucler sur le tableau **\$data** et construit deux chaînes de caractères utilisées pour construire la requête SQL d'insertion. La première chaîne, **\$firstParenthesis**, contient la liste des colonnes de la table à insérer, séparées par des virgules, et la seconde chaîne, **\$secondParenthesis**, contient des marqueurs de paramètres nommés, sous forme de **:nom_de_colonne**, également séparés par des virgules. Ces chaînes sont construites de manière à ce que leur nombre et leur ordre correspondent à ceux des colonnes de la table et des valeurs de **\$data**.

Ensuite, on appelle la méthode **query()** en lui passant en entrée la requête SQL d'insertion construite à partir des chaînes **\$firstParenthesis** et **\$secondParenthesis**, ainsi que le tableau **\$data** qui sera utilisé pour remplacer les marqueurs de paramètres nommés. La méthode **query()** s'occupera alors d'exécuter la requête et de retourner le résultat.

En résumé, la méthode **create()** permet d'insérer facilement un nouvel enregistrement dans la table associée au modèle, en utilisant une requête préparée avec des marqueurs de paramètres nommés pour protéger la base de données contre les injections SQL.

Revenons dans un contrôleur pour voir comment cette méthode est utilisée.

Par exemple dans le contrôleur **RoomController** (pour un gain de place, la partie concernant

l'upload de photos n'est pas compris dans cet extrait de code)



```
public function createRoom()  
{  
    $this->isAdmin();  
  
    $room = new Room($this->getDB());  
    $result = $room->create([  
        'title' => $_POST['title'],  
        'description' => $_POST['description'],  
        'picture' => time().'_' . $_FILES['picture']['name'],  
        'padlock' => $_POST['padlock'],  
        'n_room' => $_POST['n_room'],  
        'unlock_word' => $_POST['unlock_word'],  
        'clue' => (isset($_POST['clue'])) ? $_POST['clue'] : null,  
        'clue2' => (isset($_POST['clue2'])) ? $_POST['clue2'] : null,  
        'clue3' => (isset($_POST['clue3'])) ? $_POST['clue3'] : null,  
        'reward' => (isset($_POST['reward'])) ? $_POST['reward'] : null,  
        'user_id' => $_POST['user_id'],  
        'script_id' => $_POST['script_id'],  
    ]);  
    return header('Location: /admin/game');  
}
```

La méthode récupère les données nécessaires à la création de la salle de jeu dans les variables **\$_POST**, qui sont des variables globales contenant les données envoyées via un formulaire HTML de type "POST". Ces données sont stockées dans un tableau qui est passé en entrée de la méthode **create()** de l'objet **\$room**. La méthode **create()** s'occupera alors d'insérer ces données dans la table de la base de données correspondant aux salles de jeu.

Enfin, la méthode redirige l'utilisateur vers la page d'administration des jeux en utilisant l'en-tête HTTP "**Location**".

En résumé, la méthode **createRoom()** permet de créer une nouvelle salle en récupérant les données du formulaire HTML de création et en les insérant dans la base de données à l'aide du modèle de données **Room**. Elle vérifie également que l'utilisateur courant est bien un administrateur avant de permettre la création de la salle.

Il est nécessaire d'avoir une méthode qui permet de mettre à jour un enregistrement. Pour ceci, j'utilise la méthode **update()**



```
public function update(int $id, array $data, ?array $relations = null)
{
    $sqlRequestPart = "";
    $i = 1;

    foreach ($data as $key => $value) {
        $comma = $i === count($data) ? "" : ', ';
        $sqlRequestPart .= "{$key} = :{$key}{$comma}";
        $i++;
    }

    $data['id'] = $id;

    return $this->query("UPDATE {$this->table} SET {$sqlRequestPart} WHERE id
= :id AND user_id = {$_SESSION['user_id']} ", $data);
}
```

La méthode **update()** est une méthode permettant de mettre à jour un enregistrement dans la table associée au modèle. Elle prend en entrée un entier **\$id** qui représente l'identifiant de l'enregistrement à mettre à jour, un tableau **\$data** qui contient les nouvelles valeurs à enregistrer, sous forme de couples clé-valeur, et un tableau **\$relations** (optionnel) qui contient des informations sur les relations entre les différentes tables de la base de données.

La méthode commence par boucler sur le tableau **\$data** et construit une chaîne de caractères qui sera utilisée pour construire la requête SQL de mise à jour. Cette chaîne, appelée **\$sqlRequestPart**, contient la liste des colonnes à mettre à jour et leurs nouvelles valeurs, sous forme de **nom_de_colonne = :nom_de_colonne**, séparées par des virgules. Les marqueurs de paramètres nommés sont utilisés pour protéger la base de données contre les injections SQL.

Ensuite, la méthode ajoute l'identifiant de l'enregistrement à mettre à jour au tableau **\$data**, sous la forme d'un couple clé-valeur **'id' => \$id**.

Enfin, la méthode appelle la méthode **query()** en lui passant en entrée la requête SQL de mise à jour construite à partir de la chaîne **\$sqlRequestPart**, ainsi que le tableau **\$data** qui sera utilisé pour remplacer les marqueurs de paramètres nommés. La méthode **query()** s'occupera alors d'exécuter la requête et de retourner le résultat.

En résumé, la méthode **update()** permet de mettre à jour facilement un enregistrement dans la table associée au modèle, en utilisant une requête préparée avec des marqueurs de paramètres nommés pour protéger la base de données contre les injections SQL. La méthode vérifie également que l'enregistrement à mettre à jour appartient bien à l'utilisateur courant, grâce à la variable **\$_SESSION['user_id']**.

Le contrôleur est sensiblement le même que pour la méthode **create()**

Une différence cependant au niveau de la vue.

Si il n'est pas nécessaire d'envoyer des données à la vue **create**, il est obligatoire d'envoyer le paramètre associé à l'enregistrement que l'on veut éditer.



```
public function edit(int $id)
{
    $this->isAdmin();

    $room = (new Room($this->getDB()))->findById($id);
    $furnitures = (new Furniture($this->getDB()))->all();

    return $this->view('admin.room.edit', compact('room', 'furnitures'));
}
```

la méthode **edit()** du contrôleur room envoie les données de la **table rooms** en fonction de l'**id**. Ceci nous permet dans la vue **edit** d'afficher les valeurs que l'on souhaite mettre à jour.

Nous afficherons donc la valeur des inputs à l'aide d'un **echo**.

Exemple pour le titre.



```
<input type="text" name="title" id="title" class="form-control"
        value="<?=htmlspecialchars($room->title) ?>" required>
```

Maintenant que les principales méthodes sont présentées. Je vais expliquer comment se passe la partie enregistrement d'utilisateur, et les paramètres associés.

Vous avez pu remarquer qu'une méthode **isAdmin()** (page.21) était employée pour tout ce qui fait appel à un **create()**, **update()** et **destroy()**.

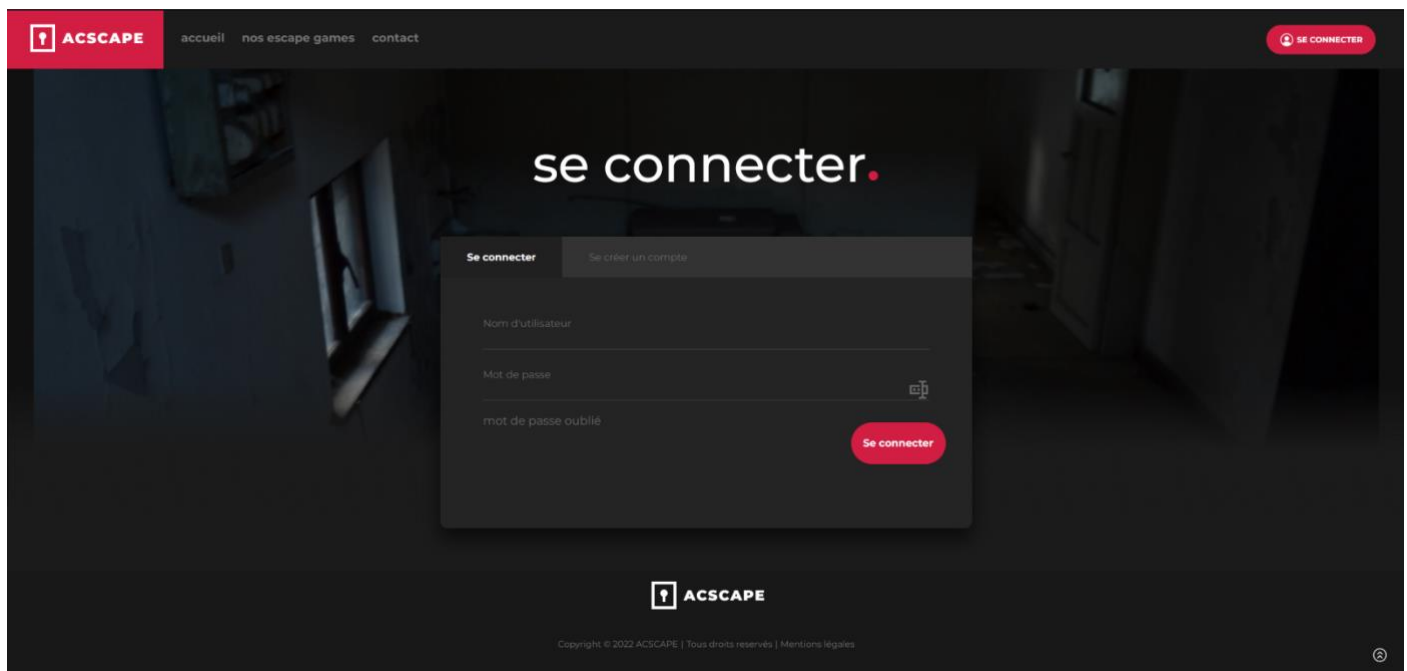
Celle-ci vérifie que l'utilisateur dispose des droits pour administrer du contenu.

Sixième étape, les utilisateurs :

J'ai besoin d'un modèle (User.php) et d'un contrôleur (UserController.php) pour gérer les utilisateurs. Avec des vues associées pour afficher et récupérer les données entrées par l'utilisateur.

Le modèle est à peu près identique aux autres modèles.

Le contrôleur sera plus complet avec en complément un **token** et le **hachage du mot de passe**.



J'utilise la méthode **registerPost()** qui permet de récupérer et enregistrer les données de l'utilisateur à son enregistrement



```
public function registerPost()
{
    $token = bin2hex(random_bytes(32));

    $validator = new Validator($_POST);
    $errors = $validator->validate([
        'username' => ['required', 'min:3'],
        'email' => ['required', 'email'],
        'password' => ['required', 'min:6'],
    ]);

    if ($errors) {
        $_SESSION['errors'][] = $errors;
        header('Location: login');
        exit;
    }

    $user = new User ($this->getDB());
    $email = $user->getByEmail($_POST['email']);
    $username = $user->getByUsername($_POST['username']);
    if ($email) {
        $_SESSION['errorMail'][] = 'Cet email est déjà utilisé';
        header('Location: login?error=email');
        exit;
    }
    if ($username) {
        $_SESSION['errorUsername'][] = 'Ce nom d\'utilisateur est déjà
utilisé';
        header('Location: login?error=username');
        exit;
    }

    $user->create([
        'username' => $_POST['username'],
        'password' => password_hash($_POST['password'], PASSWORD_BCRYPT),
        'role' => 1,
        'email' => $_POST['email'],
        'token' => $token
    ]);

    return header('Location: login');
}
```

Cette méthode commence par générer un jeton aléatoire de 32 octets de long, qui sera utilisé pour vérifier l'authenticité de l'adresse email de l'utilisateur lors de la confirmation de son inscription.

La méthode utilise ensuite une instance de la classe **Validator** pour valider les données envoyées via le formulaire d'inscription, qui sont stockées dans la variable globale **\$_POST**. Les règles de validation spécifiées sont :

- le champ "**username**" est requis et doit contenir au moins 3 caractères ;
- le champ "**email**" est requis et doit être une adresse email valide ;

- le champ "**password**" est requis et doit contenir au moins 8 caractères.

Si les données ne sont pas valides, la méthode stocke les erreurs de validation dans la variable de session **\$_SESSION['errors']** et redirige l'utilisateur vers la page de connexion avec un message d'erreur.

Ensuite, la méthode crée une nouvelle instance de la classe **User**. Elle vérifie ensuite si l'adresse email ou le nom d'utilisateur fournis par l'utilisateur sont déjà utilisés par un autre utilisateur en appelant les méthodes **getByUserMail()** et **getByUsername()** de l'objet **\$user**.

Si les données sont valides, la méthode appelle **create()** de l'objet **\$user** pour insérer un nouvel enregistrement.

Regardons ce que nous apporte la classe **Validator** avec la méthode **validate()**



```
public function validate(array $rules): ?array
{
    foreach ($rules as $name => $rulesArray) {
        if (array_key_exists($name, $this->data)) {
            foreach ($rulesArray as $rule) {
                switch ($rule) {
                    case 'required':
                        $this->required($name, $this->data[$name]);
                        break;
                    case substr($rule, 0, 3) === 'min':
                        $this->min($name, $this->data[$name], $rule);
                        break;
                }
            }
        }
    }

    return $this->getErrors();
}
```

La méthode **validate()** permet de valider des données en fonction de règles de validation spécifiées. Elle prend en entrée un tableau **\$rules** qui contient les règles de validation à appliquer, sous forme de couples clé-valeur où :

- la clé est le nom du champ à valider
- la valeur est un tableau de règles de validation sous forme de chaînes de caractères.

La méthode commence par boucler sur le tableau **\$rules** et, pour chaque champ à valider, elle vérifie si la valeur associée à ce champ existe dans le tableau de données à valider, stocké dans la propriété **\$data** de l'objet.

Si la valeur existe, la méthode boucle sur le tableau de règles de validation et appelle la méthode de validation appropriée en fonction de la règle spécifiée.

Enfin, la méthode retourne le tableau des erreurs de validation stocké dans la propriété **\$errors** de l'objet, ou **'null'**

Le **password** est évidemment haché avant d'être enregistré dans la base de donnée.

Quand un utilisateur est enregistré, une ligne s'ajoute dans la table **users** comme ceci :

Éditer	Copier	Supprimer	30	token	\$2y\$10\$FVfMeM.PycvmHlgRsag5y.1A5K2hzU4hMnEGjDLZg...	4fc5f3b5b206a2ac5006e7b98218050b869ee9a9cf9162d8	1	2023-01-02 16:25:14
Éditer	Copier	Supprimer	32	gamerbike	\$2y\$10\$V9cq.lcX7QZWXDitvmq22OtT5J1E8UcJ1F6m2XxDYMI...	15aff1f80d47872fda64cc0895400b1bdea72dde1319af8190...	1	2023-01-03 15:23:58

Enregistré dans la base de donnée, l'utilisateur peut maintenant se connecter, pour cela j'utilise la méthode **login()** et **loginPost()**



```
public function login()
{
    $csrf_token = $this->generateCsrfToken();
    setcookie('csrf_token', $csrf_token, [
        'expires' => time() + 7200,
        'samesite' => 'strict',
        'secure' => true
    ]);
    return $this->view('auth.login', compact('csrf_token'));
}
```

La méthode **login()** commence par appeler la méthode **generateCsrfToken()** pour générer un jeton **CSRF** (Cross-Site Request Forgery) aléatoire, qui sera utilisé pour protéger le site Web contre les attaques de type CSRF.

Ensuite, la méthode utilise la fonction native de PHP **setcookie()** pour définir un cookie HTTP avec le nom "**csrf_token**" et la valeur du jeton CSRF généré précédemment. Le cookie est configuré avec un délai d'expiration de 7200 secondes (2 heures), une option "**SameSite**" de valeur "**strict**" qui empêche le cookie d'être envoyé avec des requêtes effectuées depuis un autre site Web, et une option "**secure**" qui indique que le cookie ne doit être envoyé que si la connexion est sécurisée (via HTTPS). Dans le cas d'un développement en local en http, « **secure** » est commenté.

Enfin, la méthode utilise la méthode **view()** de l'objet courant pour afficher la vue "auth.login" avec la variable **\$csrf_token** passée en tant que variable compactée. La vue "auth.login" est une vue qui affiche le formulaire de connexion, avec un champ caché qui contiendra le jeton CSRF afin de protéger le formulaire contre les attaques de type CSRF.

Les données du formulaire de la vue login seront envoyés à l'aide de la méthode **loginPost()**



```
public function loginPost()
{
    if (!$this->validateCsrfToken($_POST['csrf_token'])) {
        return header('Location: /login?error=invalid_csrf_token');
    }

    $validator = new Validator($_POST);
    $errors = $validator->validate([
        'username' => ['required', 'min:3'],
        'password' => ['required']

    ]);

    if ($errors) {
        $_SESSION['errors'][] = $errors;
        header('Location: /login');
        exit;
    }

    $user = (new User($this->getDB()))->getByUsername($_POST['username']);

    if (password_verify($_POST['password'], $user->password)) {
        $_SESSION['auth'] = (int) $user->role;
        $_SESSION['user_id'] = (int) $user->id;
        $_SESSION['token'] = $user->token;
        return header('Location: index');
    } else {
        return header('Location: /login?error=error');
    }
}
```

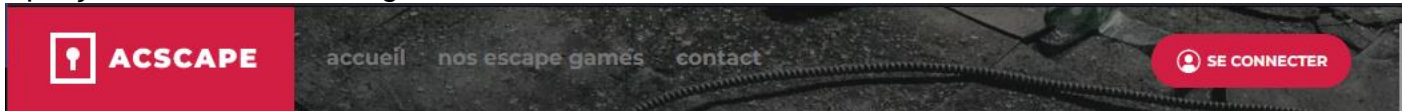
Si le **token_csrf** est valide et que l'ensemble des données correspondent à la base de données alors l'utilisateur est connecté au site.

Septième étape, les restrictions :

Maintenant qu'un système de connexion est établi, il est possible d'afficher ou non des parties du site en fonction du statut de connexion.

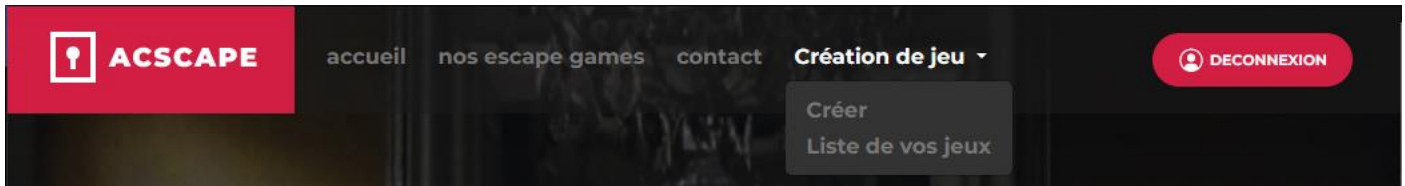
Les jeux sont accessibles à tout utilisateur non connecté. La partie création (admin) est disponible uniquement pour un utilisateur enregistré et connecté.

Aperçu de la barre de navigation avec le statut « invité »



la barre de navigation se compose : du logo, d'un menu et un bouton pour se connecter.

Si l'utilisateur est connecté :



Une option supplémentaire (menu dropdown) apparaît : Création de jeu et le bouton se connecter se change en bouton de déconnexion.

Ce changement est issu de conditions dans layout.php.



```
<?php if (isset($_SESSION['user_id'])) : ?>
    <li class="nav-item dropdown">
        <a class="nav-link dropdown-toggle" href="#" id="navbarDropdown" role="button"
            data-bs-toggle="dropdown" aria-expanded="false">
            Création de jeu
        </a>
        <ul class="dropdown-menu" aria-labelledby="navbarDropdown">
            <li><a class="dropdown-item" href="/admin/posts">Créer</a></li>
            <li><a class="dropdown-item" href="/admin/script">Liste de vos jeux</a></li>
        </ul>
    </li>
<?php endif; ?>
```

La variable de session `$_SESSION['user_id']` est enregistré au moment du login,



```
$_SESSION['user_id'] = (int) $user->id;
```

Elle permet de vérifier si l'utilisateur est bien connecté et si oui, alors on affiche le menu dropdown de création de jeu.

Pour les pages admins, la méthode `isAdmin()` vérifie que l'utilisateur a le bon statut, et si le cookie `csrf_token` correspond bien. Si tout est bon, l'utilisateur a accès à la partie « création de jeu ».



```
protected function isAdmin()
{
    if (isset($_SESSION['auth']) && $_SESSION['auth'] === 1) {
        return true;
    } else {
        return header('Location: login');
    }
    if ($_COOKIE['csrf_token'] != $_SESSION['csrf']) {
        return header('Location: /login?error=session_expired');
    }
}
```

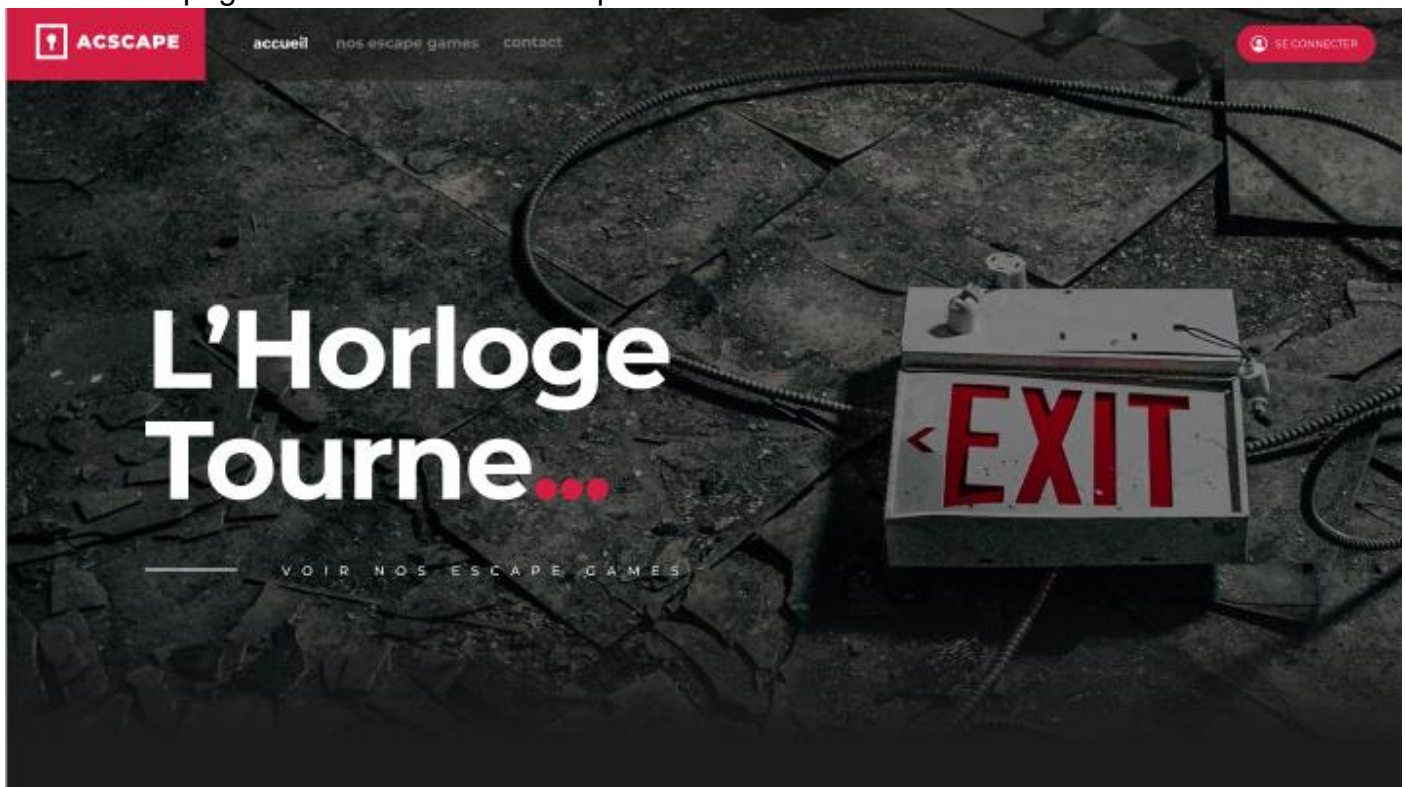
Front-End

Maintenant que nous avons vu l'essentiel de la partie back-end, je vais vous présenter la partie front-end.
Cette partie s'attardera sur ce qui se passe dans le dossier « **views** » et en particulier sur la partie « **jeu** »

la mise en page, les vues :

Pour la mise en page, je me réfère au travail de mon tuteur sur la maquette faite avec Figma.
J'extrait de la maquette l'ensemble des images et des icônes.
J'utiliserai le **framework CSS Bootstrap** pour le responsive et divers composants.
Et du **CSS Custom** pour tout ce qui n'est pas possible de faire avec Bootstrap.

Extrait de la page d'accueil issu de la maquette :



Comme on peut le voir, l'image d'arrière-plan comporte un filtre.
Je reproduis ce filtre à l'aide d'un **gradient linéaire en CSS** et la propriété **background-image**



```
background-image: linear-gradient(180deg, rgba(27, 27, 27, 0) 0%, #1B1B1B 100%), url(../../assets/front/home/homeTop.webp);
```

Pour optimiser la charge réseau, **l'image est encodé en .webp**
Ce format permet une compression de plus de 80% sur cette image.

Le titre « L'Horloge tourne » dispose de trois points rouges.
Je prends l'initiative de faire une petite animation sur ceux-ci, pour donner un effet de temps qui passe.
Pour faire cette animation, j'utilise les propriétés **CSS animation** et **keyframes**



```
.little_red_circle>span:nth-child(1) {  
  animation: pulsation 2s infinite;  
}  
  
.little_red_circle>span:nth-child(2) {  
  animation: pulsation 2s infinite;  
  animation-delay: 0.2s;  
}  
  
.little_red_circle>span:nth-child(3) {  
  animation: pulsation 2s infinite;  
  animation-delay: 0.4s;  
}
```

Je sélectionne séparément les différents « points rouges » afin de pouvoir leur attribuer à chacun un délai différent

```
@keyframes pulsation {  
  0% {  
    transform: scale(1);  
    opacity: 1;  
  }  
  
  50% {  
    transform: scale(1.2);  
    opacity: 0;  
  }  
  
  100% {  
    transform: scale(1);  
    opacity: 1;  
  }  
}
```

Ce **keyframes** appelle l'animation

Elle se répète infiniment pour passer d'une **opacity** 1 à 0, et un **scale** de 1 à 1.2.

Pour éviter de faire trop de média queries, je choisis d'utiliser une nouvelle propriété **css** : **clamp()**
Par exemple pour le titre :



```
.title {  
  font-size: clamp(3.125rem, 1.9737rem + 6.1404vw, 7.5rem);  
  font-style: normal;  
  font-weight: 700;  
}
```

Pour les paramètres de la propriété clamp de font-size je me suis aidé de l'outil en ligne : <https://clamp.font-size.app>



The screenshot shows the 'Font-size Clamp Generator' interface. It has a dark background with white text. The title 'Font-size Clamp Generator' is at the top, followed by the subtitle 'Generate linearly scale font-size with clamp()'. Below this, there are four input fields arranged in a 2x2 grid. The first row is for 'Minimum viewport width' (500 PX) and 'Maximum viewport width' (900 PX). The second row is for 'Minimum font size' (16 PX) and 'Maximum font size' (48 PX). At the bottom, there is a text box containing the generated CSS code: 'font-size: clamp(1rem, -1.5rem + 8vw, 3rem);'. To the right of the code box is a clipboard icon.

Font-size Clamp Generator
Generate linearly scale font-size with clamp()

Minimum viewport width: 500 PX
Maximum viewport width: 900 PX
Minimum font size: 16 PX
Maximum font size: 48 PX

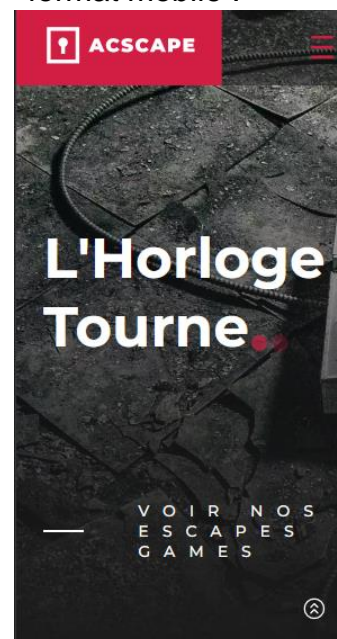
font-size: clamp(1rem, -1.5rem + 8vw, 3rem);

Il permet de définir la largeur en pixel minimum et maximum du viewport et de la taille de la police. Et compiler ses valeurs pour la propriété clamp()

format bureau :



format mobile :



Pour la suite de la page d'accueil, j'utilise les classes de **Bootstrap** pour rendre rapidement la page **adaptable** et **responsive** :

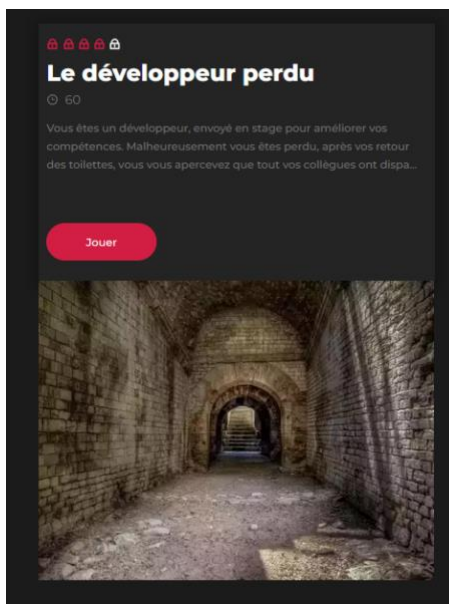


```
<div class="first_game d-flex flex-column flex-lg-row justify-content-center my-3">
```

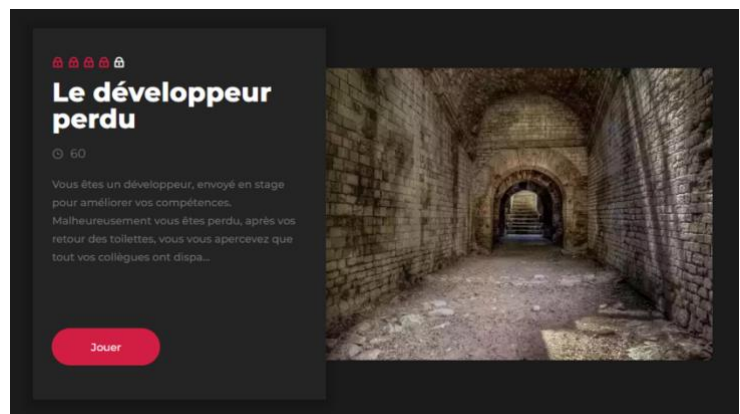
Le **frameworks Bootstrap** étant mobile first, les classes qui correspondent au format mobile n'ont pas de « suffixe ».

Dans ce cas, **flex-column** ajoute la propriété **CSS flex-direction : column** de base jusqu'à ce que l'on lui indique un changement de taille d'écran.

Ici il s'agit de **flex-lg-row**, qui peut se traduire par : *à partir d'une taille minimum de 992px de large, la direction en colonne change pour passer à une direction en ligne.*



Format mobile



Format Bureau

Le contenu de ce premier jeu présenté est issu de la base de donnée.

Ces données sont appelées depuis un contrôleur qui nous renvoie ce qui nous intéresse dans une vue

Les données arrivent sous la forme d'un tableau d'objet, je choisis de les enregistrer dans une variable pour une meilleure lisibilité du code.

```
$script = $params['script'];|
```

Avec l'opérateur de l'objet en PHP « -> » il est possible d'accéder aux propriétés de l'objet. Par exemple



```
<div class="title_game">
  <h3><?= $script[0]->title ?></h3>
</div>
```

Pour récupérer le titre du script index 0 dans le conteneur div, titre du jeu.



```
<div class="diffuculty my-2" data-difficulty="<?= $script[0]->difficulty ?>">
  <iconify-icon icon="ri:lock-line"></iconify-icon>
  <iconify-icon icon="ri:lock-line"></iconify-icon>
  <iconify-icon icon="ri:lock-line"></iconify-icon>
  <iconify-icon icon="ri:lock-line"></iconify-icon>
  <iconify-icon icon="ri:lock-line"></iconify-icon>
</div>
```

En ce qui concerne l'affichage de la difficulté, j'aurais besoin de **Javascript** pour colorer les cadenas, en récupérant la valeur renvoyée en PHP dans l'**attribut data-difficulty**



```
const difficultyDivs = document.querySelectorAll('.diffuculty');

for (const difficultyDiv of difficultyDivs) {
  const icons = difficultyDiv.querySelectorAll('iconify-icon');
  const difficulty = difficultyDiv.getAttribute('data-difficulty');
  for (let i = 0; i < difficulty; i++) {
    icons[i].classList.add('red');
  }
  for (let i = difficulty; i < icons.length; i++) {
    icons[i].classList.add('white');
  }
}
```

Je commence par sélectionner tous les conteneurs div de la page qui ont la **classe .diffuculty**. Ensuite j'utilise une boucle pour parcourir chacun des éléments définis dans la constante vue précédemment.

Je déclare ensuite à nouveau une nouvelle constante icons pour sélectionner les éléments icones présent à l'intérieur de difficultyDiv.

Je fais la même chose pour récupérer la valeur de l'attribut « **data-difficulty** », ce qui me permet d'avoir la valeur associée de la difficulté à chaque conteneur.

Et donc en fonction de la valeur de **data-difficulty** récupérer les éléments prennent la classe « **red** » qui correspond à un **stroke = « red »** en CSS. Et j'ajoute ensuite la classe « **white** » pour les cadenas suivant.

La page « nos escapes games » est composée sur à peu près le même schéma.

Cependant les jeux n'étant pas choisis de manière arbitraire contrairement à la page d'accueil, j'utilise une boucle **forEach** pour parcourir l'ensemble des données renvoyées par le contrôleur.

(les **** représentent du code non affiché dans l'exemple)



```
<?php foreach ($scripts as $game) : ?>
    *****
    <p class="m-0"><?= $game->title ?></p>
    *****
<?php endforeach; ?>
```

Une différence dans une boucle **forEach**, il n'est pas obligatoire de déclarer un index comme précédemment.

Pour la page description du jeu (show).

Je récupère les données du jeu à afficher depuis la méthode **show** du contrôleur associé, mais également avec l'**id** de l'élément passé par le lien sur la page précédente.



```
<a class="link_game" href="/show/<?= $game->id ?>"></a>
```

Comme on peut le voir le contrôleur attend une **id** :



```
public function show(int $id)
{
    $script = new Script($this->getDB());
    $script = $script->findById($id);
    $scriptAll = $script->all();
    return $this->view('home.show', compact('script', 'scriptAll'));
}
```

Il ne faut pas oublier de le préciser à la définition de la route :



```
$router->get('/show/:id', 'App\Controllers\HomeController@show');
```

Avec toute ces informations il est donc possible de cibler précisément un élément et d'afficher ses propriétés.

Comme un seul élément est renvoyé avec la méthode **findById**, il n'y pas besoin d'indexer, ni de parcourir une boucle.

Je peux donc afficher les données sous cette forme :



```
<p><?= $script->description ?></p>
```

Pour afficher les autres jeux en bas de page de manière plus ludique et facilement responsive je fais le choix d'utiliser un carrousel à l'aide la **librairie Splide JS**.

<https://splidejs.com/>

Pour expliquer ce choix, cette librairie est très légère, sans dépendance, simple d'utilisation et parfaitement adaptée au mobile.

Comme expliqué dans la documentation de la librairie :



```
<section class="splide" aria-label="Basic Structure Example">
  <div class="splide__track">
    <ul class="splide__list">
      <li class="splide__slide">Slide 01</li>
      <li class="splide__slide">Slide 02</li>
      <li class="splide__slide">Slide 03</li>
    </ul>
  </div>
</section>
```

Il faut définir une **section** avec une **classe**, et ensuite respecter la structure **div>ul>li**. Ce qui donnera pour mon cas, la structure suivante :



```
<section class="splide mx-auto slide_game_index" aria-label="slide_game">
  <div class="splide__track">
    <ul class="splide__list">
      <?php foreach ($games as $game) : ?>
      <?php if ($game->id != $_SESSION['scriptId']) : ?>
      <li class="game splide__slide">
        
        <span class="filter"></span>
        <div class="card_game_content d-flex">
          <div class="d-flex align-items-end h-100 justify-content-between p7">
            <div class="d-flex flex-column">
              <div class="title_game_index">
                <p class="m-0"><?= $game->title ?></p>
              </div>
              <div class="diffcully" data-difficulty="<?= $game->difficulty ?>">
                <iconify-icon data-id="<?= $game->id ?>" icon="ri:lock-line"></iconify-icon>
                <iconify-icon data-id="<?= $game->id ?>" icon="ri:lock-line"></iconify-icon>
                <iconify-icon data-id="<?= $game->id ?>" icon="ri:lock-line"></iconify-icon>
                <iconify-icon data-id="<?= $game->id ?>" icon="ri:lock-line"></iconify-icon>
                <iconify-icon data-id="<?= $game->id ?>" icon="ri:lock-line"></iconify-icon>
              </div>
            </div>
            <div class="time_game d-flex justify-content-center align-items-center gap-3">
              <iconify-icon icon="mdi:clock-time-three-outline" style="color: #717171;">
              </iconify-icon>
              <p class="m-0 duration_color"><?= $game->duration?></p>
            </div>
          </div>
          <a class="link_game" href="/show/<?= $game->id ?>"></a>
        </li>
      <?php endforeach; ?>
    </ul>
  </div>
</section>
```

Ensuite en **Javascript**, une fois le **DOM** chargé à l'aide de l'écouteur d'évènement «**DOMContentLoaded** », j'indique les paramètres nécessaires au lancement du script



```
document.addEventListener('DOMContentLoaded', function () {  
    var splide = new Splide('.splide', {  
        type: "loop",  
        perPage: 1,  
        gap: "1rem",  
    });  
    splide.mount();  
});
```

Voilà pour l'essentiel des pages de présentations.

Le jeu :

Je vais dans cette partie décrire de manière non exhaustive, le script JS du jeu.

Tout d'abord pour faire marcher le jeu sans rechargement de page, il a fallu faire des choix techniques.

Afin d'éviter des requêtes serveurs, avec des appels **AJAX** et pour un soucis de simplicité et d'efficacité, j'ai opté pour envoyer les données nécessaires au jeu sous le format **JSON**.

Pour générer et envoyer le **JSON**, j'inscris une méthode **json()** dans mon contrôleur principal.



```
protected function json(string $path, array $data)  
{  
    $path = str_replace('.', DIRECTORY_SEPARATOR, $path);  
    require VIEWS . $path . '.php';  
    header('Content-Type: application/json');  
    echo json_encode( $data );  
    exit();  
}
```

J'utilise la fonction « **header** » pour définir le type de contenu de la réponse comme étant « **application/json** » afin d'indiquer au navigateur le type de fichier.

J'utilise ensuite la fonction **json_encode** pour convertir les données en format **JSON**.

Le contrôleur **InGameController** héritant de **Controller** j'utilise la méthode précédemment définie.

Comme ceci :



```
public function jsonGame()
{
    $room = new Room($this->getDB());
    $room = $room->allByScriptIdByNroom($_SESSION['scriptId']);
    $furniture = new Furniture($this->getDB());
    $furniture = $furniture->allByFurnitureId($_SESSION['scriptId']);
    $script = new Script($this->getDB());
    $script = $script->allByScriptId($_SESSION['scriptId']);

    $data = [
        'room' => $room,
        'furniture' => $furniture,
        'script' => $script];

    return $this->json('json.data', compact('data'));
}
```

Le **JSON** emprunte une route comme une page « normale », n'ayant pas de Middleware comme sur Laravel par exemple.



```
$router->get('/json/data', 'App\Controllers\InGameController@jsonGame');
```

Maintenant que j'ai dans un **JSON** toute les données nécessaires pour créer le jeu, il est temps de récupérer ces données et les exploiter à l'aide de **Javascript**.

Pour cela j'utilise l'**API Fetch**.



```
async function getData() {
    try {
        const response = await fetch('/json/data');
        if (response.ok) {
            const data = await response.json();
            dataGlobal = data.data;

            return dataGlobal;
        } else {
            console.error('Erreur lors de la récupération des données :', response.statusText);
        }
    } catch (error) {
        console.error('Erreur lors du chargement des données :', error);
    }
}
```

La fonction **getData** est utilisé pour récupérer les données à partir d'une URL ici : /json/data

J'utilise l'objet « **fetch** » pour envoyer une demande **GET** à l'URL spécifiée.

Cette fonction utilise « **async** » pour gérer des appels réseau asynchrone ainsi que « **try / catch** »

« **try** » pour recevoir la réponse si le chargement est bon, et « **catch** » dans le cas contraire.

Comme c'est une requête asynchrone il est nécessaire d'attendre la réponse en utilisant « **await** »

Je stocke le résultat dans un tableau déclaré avec la variable **dataGlobal**.
dataGlobal est déclarée dans la portée globale, comme ceci :

```
let dataGlobal = [];
```

Je définirais au fur et à mesure de mon avancé dans le code les variables qui me seront nécessaires dans la portée globale afin de pouvoir les utiliser comme bon me semble.

A ce moment, je vais avoir besoin de parcourir l'ensemble des salles à afficher à l'utilisateur, en définissant les conditions, de si celle-ci sont ouvertes ou fermées.

Pour parcourir l'ensemble des salles j'utilise **une boucle for**, comme ceci :



```
for (let i = 0; i < dataGlobal.room.length; i++)
```

Pour afficher les éléments retournés, j'utilise la méthode **createElement** pour créer une liste avec « **li** » ainsi que la méthode **appendChild** pour ajouter un nœud à la fin de la liste des enfants d'un nœud parent spécifié. Ici le nœud parent est **roomList**



```
li_room = document.createElement('li');  
li_room.classList.add('rooms_list_item', `nb-${i}`);  
roomsList.appendChild(li_room);
```

Maintenant que nos salles sont créés dans des éléments li, il faut que j'apporte les conditions qui permettront de différencier la modal qui sera ouverte au click en fonction de si elles sont fermées ou ouvertes.

Pour générer une modal au clic, il faut ajouter un écouteur d'évènement « click »



```
roomsArray[i].addEventListener('click', function (roomClick) {
```

A chaque clic sur un élément de **roomsArray**, il se produira un évènement.
Je dois donc ici poser les conditions nécessaires à la différenciation.

Dans le cas où la salle est ouverte :



```
if (dataGlobalUnlock[0].room[i]['padlock'] == "no")
```

Si l'on clique sur une salle ouverte, la salle est considérée comme active, et le background du jeu

doit s'adapter à l'image de la salle.



```
document.querySelector('body').style.backgroundImage = `url(/assets/pictures/rooms/${dataGlobalUnlock[0].room[i].picture})`;
```

Ici l'image récupérée viendra changer le background de « body »

Puis encore une fois à l'aide de la méthode « **createElement** », je crée une modale sur le modèle de Bootstrap 5 pour afficher le contenu attendu.

La section salle ouverte ne comporte pas beaucoup plus de fonctionnalités, celle-ci étant ouverte, il est inutile d'y apporter les fonctions propres à un déverrouillage.

Passons à la section des salles fermées.

Pour cette partie, j'ai besoin d'une nouvelle condition :



```
if (dataGlobal.room[i]['padlock'] == "yes" &&  
roomsArray[i].classList.contains('room_unlock_open') == false) {
```

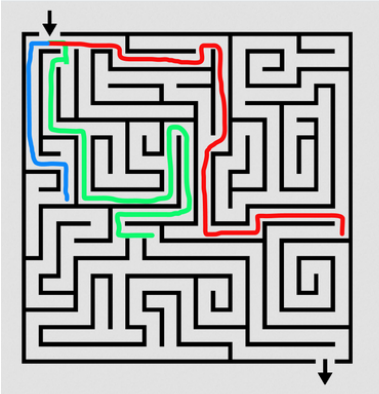
A l'intérieur de cette condition, il va se passer plusieurs choses.

La première étant d'afficher une modale différente de celle proposée pour les salles ouvertes.


Toujours avec « **createElement** » je crée une modale avec Bootstrap 5.

Celle-ci devra contenir l'image de la salle, le contenu associé au fait qu'elle soit fermée. C'est-à-dire : **description**, **indice**, et un **input** pour inscrire la réponse.

Le Labyrinthe de la Tombe



Un étrange mécanisme est gravé sur la porte, on dirait un labyrinthe. Des chemins de couleurs différentes sont déjà tracés, lequel prendre ? Est-ce la solution pour déverrouiller cette porte ?



3

Indice 1

Indice 2

Indice 3

la solution semble être la couleur, mais quelque chose ne va pas

Les indices entraînent des pénalités de temps à l'ouverture. Une pénalité de temps est également appliquée après 3 mauvaises réponses.

Les pénalités sont affichées en dessous de l'input pour rentrer les réponses.

Et entre chaque indice, une intervalle de temps avant de pouvoir ouvrir le prochain.

Si l'on souhaite enregistrer l'indice quelque part, il est possible de copier son contenu sur un « post-it » pour le retrouver plus tard, en fonction de la complexité du jeu créé par l'utilisateur.

le post-it depuis cette icône récupère le titre de la salle et le contenu de l'indice :

✖
Le Labyrinthe de la Tombe

indice n°1
la solution semble
être la couleur, mais
quelque chose ne va
pas

Il est ensuite possible de le « **grab** » attraper, pour le positionner où on le souhaite.



Pour différencier les indices des salles, et éviter une **propagation** de la fonction de pénalité de temps, il fallait que je trouve une solution pour indiquer qu'un ou plusieurs indice avaient déjà été ouvert.



```
clue_show1.addEventListener('click', function () {  
  
    clue_show_content1.classList.toggle('dnone');  
    this.classList.toggle('rotate_5deg');  
    clue_show_content2.classList.add('dnone');  
    clue_show2.classList.remove('rotate_5deg');  
    clue_show_content3.classList.add('dnone');  
    clue_show3.classList.remove('rotate_5deg');  
    clue_show_content1.innerHTML = `

<iconify-icon class="copy" icon="clarity:copy-to-clipboard-line"></iconify-icon></button>`;  
  
    if (t == 0 && !dataGlobalUnlock[0].room[i].clue1penalty) {  
        dataGlobalUnlock[0].room[i].clue1penalty = "yes";  
        t++;  
        penalty.classList.add('topToBottom');  
        penalty.textContent = "-30 sec";  
        penalty_info.textContent = '';  
        penalty_info.classList.remove('fade');  
        function_penalty_info()  
        countdown = countdown - 30;  
        removeTopttoBottom();  
    }  
  
    if (!dataGlobalUnlock[0].room[i].clue1Found) {  
        intervalFunction(function () {  
            dataGlobalUnlock[0].room[i].clue1Found = "yes";  
            tooltipList[0].hide();  
            tooltipList[0].disable();  
  
            interval = 10;  
        });  
    }  
}


```

Après avoir essayé de donner des attributs aux indices, cette solution s'avéra être un échec, en effet les attributs sont redéclarés à chaque nouvelle modale. La méthode la plus optimale qui me soit venue a donc été d'inscrire une nouvelle propriété dans mon objet.

Et donc si l'élément ne contient pas cette propriété au click, alors je l'ajoute. De cette manière, chaque indice est bien celui de la salle associée.

En ce qui concerne l'input, je fais une vérification, comparaison de ce qui est entrée par l'utilisateur avec le bon code contenu dans l'objet.



```
if (rooms_unlock_key.value == dataGlobal.room[i]['unlock_word']) {
```

Pour une meilleure expérience utilisateur, je m'assure qu'il ne puisse pas rentrer un résultat « vide » avec :



```
if (rooms_unlock_key.value == '')
{
    e.preventDefault();
}
```

Comme je suis dans un évènement au click, je m'assure également de prévenir l'envoi de « vide » avec le clavier.



```
document.addEventListener('keydown', function (e) {
    if (e.key == 'Enter')
        *****
        if (rooms_unlock_key.value == '') {
            e.preventDefault();
            return;
        }
        *****
});
```

Les salles peuvent être composées de meubles que l'on peut fouiller.

Pour explorer la liste des meubles, j'avais d'abord commencé avec une boucle for.

Ce qui a eu pour effet de me poser de nombreux problèmes, en particulier avec des retours « undefined ».

C'est qu'un peu plus tard après des recherches que j'ai compris qu'étant donné que je triais les meubles à afficher, et que ma boucle for retournait l'ensemble des meubles. Je me retrouvais avec des éléments « undefined »

Par exemple, si ma liste contient 10 meubles, mais qu'après mon trie je n'en retourne que 3. Les 7 restants sont lus comme « undefined » provoquant des problèmes en chaînes dans les écouteurs d'évènements, l'affichage etc...

Je revois donc la méthode, et j'opte pour une boucle **forEach**. J'aurais également pu opté pour écarter les résultats « **undefined** » de la **boucle for**, mais **forEach** me paraissait plus adapté.



```
dataGlobal.furniture.forEach(furniture => {  
    furniture.unlock_word = furniture.unlock_word.toLowerCase();  
    if (furniture.room_id === roomID) {
```

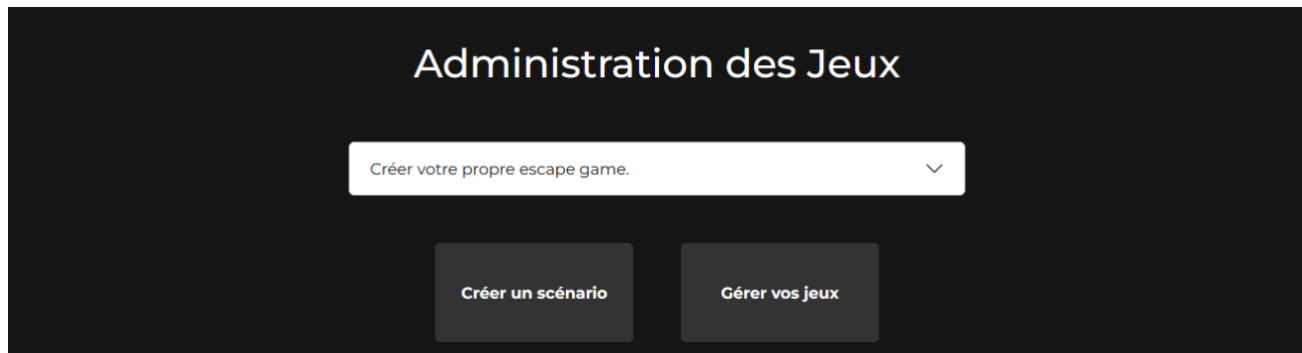
La méthode **forEach** résout tous les problèmes rencontrés précédemment avec une **boucle for**. Je peux ainsi y passer des conditions et il me retourne que les éléments qui correspondent.

La suite est à peu de chose près similaire aux salles.

Créations de modale en fonction de si le meuble est verrouillé ou non, et les interactions qui vont avec.


Présentation du jeu d'essai sur une fonctionnalité représentative.

Pour ce jeu d'essai, j'ai choisi de vous montrer la création du scénario d'un jeu
Il faut tout d'abord se connecter pour accéder à la création d'un jeu.



Une fois connecté, il est possible d'accéder à la partie administration.
Le bouton, « créer un scénario » conduit à un formulaire.

La liste initial des jeux pour le compte connecté ressemble à :




Le développeur perdu

Vous êtes un développeur, envoyé en stage pour améliorer vos compétences. Malheureusement vous êtes perdu, après vos retour des toilettes, vous...

Editer

Supprimer




échapper à la tombe du Pharaon

Vous êtes un archéologue qui a réussi à pénétrer dans la tombe du Pharaon, mais vous vous êtes fait surprendre par des gardiens qui ont verrouillé...

Editer

Supprimer



Le Laboratoire de Louis Pasteur




Vous êtes un groupe de scientifiques qui avez été invités à visiter le célèbre laboratoire de Louis Pasteur. Cependant, lorsque vous arrivez, v...

Editer

Supprimer

et côté base de données :

id	title	difficulty	description	winner_msg	lost_msg	picture	duration	user_id
57	Le Laboratoire de Louis Pasteur	3	Vous êtes un groupe de scientifiques qui avez été ...	Bravo, vous avez permis aux scientifiques d'accomp...	Domage, vous n'avez pas pu libérer les scientifiq...	1672756200_pasteur.jpg	60	32
58	échapper à la tombe du Pharaon	3	Vous êtes un archéologue qui a réussi à pénétrer d...	Bravo vous avez réussi à vous échapper !	Vous avez succomber aux pièges de la pyramide.	1672995121_scriptBg.jpg	60	32
60	Le développeur perdu	4	Vous êtes un développeur, envoyé en stage pour am...	Vous avez réussi à vous échapper et retrouver vos ...	Domage, il fallait mieux suivre les cours de la f...	1673253343_img_game.webp	60	32

lection :  Editer  Copier  Supprimer  Exporter

Une fois cliqué sur le bouton, la page affiche un formulaire :

Création d'un scénario

Nom du scénario

test jeu d'essai

Description


test jeu d'essai

Message de victoire

bravo test concluant

Message de défaite

dommage test echoué



Difficulté à

Très Facile





Durée en minutes

28

Créer

Une fois le formulaire validé et envoyé, une ligne s’ajoute bien à la base de donnée

id	title	difficulty	description	winner_msg	lost_msg	picture	duration	user_id
57	Le Laboratoire de Louis Pasteur	3	Vous êtes un groupe de scientifiques qui avez été ...	Bravo, vous avez permis aux scientifiques d'accomp...	Dommmage, vous n'avez pas pu libérer les scientifiq...	1672756200_pasteur.jpg	60	32
58	échapper à la tombe du Pharaon	3	Vous êtes un archéologue qui a réussi à pénétrer d...	Bravo vous avez réussi à vous échapper !	Vous avez succomber aux pièges de la pyramide.	1672995121_scriptBg.jpg	60	32
60	Le développeur perdu	4	Vous êtes un développeur, envoyé en stage pour am...	Vous avez réussi à vous échapper et retrouver vos ...	Dommmage, il fallait mieux suivre les cours de la f...	1673253343_img_game.webp	60	32
64	test jeu d'essai	1	test jeu d'essai	bravo test concluant	dommmage test echoué	1673792325_DALL-E 2023-01-08 00:58:20 - hopital ef...	28	32


action :  Éditer  Copier  Supprimer  Exporter

Et côté front, tout c'est bien ajouté :

Administration des scénarios

En savoir plus

AJOUTER UN SCÉNARIO




test jeu d'essai

test jeu d'essai...

Editer

Supprimer




Le développeur perdu

Vous êtes un développeur, envoyé en stage pour améliorer vos compétences. Malheureusement vous êtes perdu, après vos retour des toilettes, vou...

Editer

Supprimer




échapper à la tombe du Pharaon

Vous êtes un archéologue qui a réussi à pénétrer dans la tombe du Pharaon, mais vous vous êtes fait surprendre par des gardiens qui ont verroui...

Editer

Supprimer



Le Laboratoire de Louis Pasteur

Vous êtes un groupe de scientifiques qui avez été invités à visiter le célèbre laboratoire de Louis Pasteur. Cependant, lorsque vous arrivez, v...

Editer

Supprimer

Description de la veille effectuée pour le projet.

Je n'ai à proprement parlé, pas vraiment fait de veille lié à ce projet. Mais je pratique une veille continue depuis que j'ai commencé la formation développeur web.

En particulier sur des plateformes sociales comme Twitter, Twitch, Youtube et Discord.

Mais aussi sur certains sites d'informations numériques et tech, généraliste ou non, comme :

<https://www.developpez.com/>
<https://www.numerama.com/>
<https://www.awwwards.com/>
<https://www.nextinpact.com/>
<https://korben.info/>
<https://tympanus.net/codrops/>
<https://css-tricks.com/>
<https://medium.com/>

Concernant les personnes que je suis sur plusieurs plateformes.

- [Grafikart](#)
- [BastiUi](#)
- [Lior Chamla](#)
- [SilverLeDev](#)
- [Micode](#)
- [Nord Coders](#)
- [Benjamin Code](#)
- [William Traoré](#)
- [Deafmute](#)

Sans oublier des podcasts sur Spotify par exemple :

- [Double slash](#)
- [TPMC](#)

Extrait d'un site anglophone utilisé dans le cadre d'une recherche

Avec l'utilisation de **Bootstrap**, il y a beaucoup de propriétés **CSS** qui sont inutilisées. J'ai fait des recherches pour savoir si il était possible d'épurer le **CSS** de ces classes pour gagner du temps de chargement.

The screenshot shows a Google search interface with the query "how to clean unused css". The search results are as follows:

- Search Results:** Environ 1840 000 résultats (0,42 secondes)
- First Result:**
 - URL: <https://wp-rocket.me> > ... > Largest Contentful Paint
 - Title: [How to Remove \(or Reduce\) Unused CSS on WordPress](#)
 - Snippet: If you want to remove unused CSS entirely, you can **use a tool such as PurifyCSS to find out how much CSS file size can be reduced**. Once you get the CSS code you should eliminate, you have to remove it manually from the page. If you want to deep dive into a manual solution, you can read the CSS-tricks in-depth article.
- Second Result:**
 - URL: <https://www.keycdn.com> > blog > re...
 - Title: [How to Remove Unused CSS for Leaner CSS Files - KeyCDN](#)
 - Snippet: 4 mars 2019 — **How to remove unused CSS** manually# · Open Chrome DevTools · Open the command menu with: cmd + shift + p · Type in "Coverage" and click on the "Show ...
- Third Result:**
 - URL: <https://css-tricks.com> > how-do-you-...
 - Title: [How Do You Remove Unused CSS From a Site?](#)
 - Snippet: 19 nov. 2019 — To **clean CSS** with multiple pages just use that tool for all pages with different layouts. Then merge all exported **CSS** files to one. Then the ...
- Recherches associées:**
 - purgecss
 - css cleaner
 - remove unused css online
 - wordpress remove unused css
 - remove unused css visual studio
 - webpack remove unused css

J'ai choisi la seconde réponse d'un site que je connais bien et que je sais fiable et intéressant.

Voici un extrait de la page de **css-tricks**.

🔗 The motivation

I imagine the #1 driver for the desire to remove unused CSS is this:

You used a CSS framework (e.g. Bootstrap), included the framework's entire CSS file, and you only used a handful of the patterns it provides.

I can empathize with that. CSS frameworks often don't provide simple ways to opt-in to only what you are using, and customizing the source to work that way might require a level of expertise that your team doesn't have. That might even be the reason you reached for a framework to begin with.

Say you're loading 100 KB of CSS. I'd say that's a lot. (As I write, this site has ~23 KB, and there are quite a lot of pages and templates. I don't do anything special to reduce the size.) You have a suspicion, or some evidence, that you aren't using a portion of those bytes. I can see the cause for alarm. If you had a 100 KB JPG that you could compress to 20 KB by dropping it onto some tool, that's awesome and totally worth it. **But the gain in doing that for CSS is even more important because CSS is loaded in the head and is render blocking.** The JPG is not.

Traduction :

« J'imagine que le souhait numéro 1 est le désir de supprimer le CSS inutilisé.

Vous utilisez un framework, par exemple Bootstrap, et vous incluez le fichier CSS complet, et vous voulez seulement utilisé ce qui vous est utile .

Je peux comprendre cela, et les frameworks CSS n'offrent souvent pas de moyens simples pour utiliser seulement le nécessaire et sélectionner et customiser ce qui est utile pour le travail peut demandé un certain niveau d'expertise. Cela peut même être une des raisons du choix du framework

Si vous chargez 100kb de CSS, je vous dirais que c'est beaucoup (alors que j'écris ces lignes, ce site a environ 23 KB et il y a beaucoup de pages et de modèles, je ne fais rien de spécial pour réduire la taille)

Vous avez quelques doutes, suspicions que vous n'utilisez pas tout ce qui est chargé. Je peux voir la cause de votre attention.

Si vous avez un JPG de 100kb et que vous voulez le compresser jusqu'à 20kb, vous n'avez qu'à le déposer dans un outil et c'est génial, et véritablement rentable. Mais le gain pour le CSS est encore plus important parce que le CSS est chargé dans l'en-tête de votre page et bloque le rendu, ce que ne fait pas un JPG

The biggest flaw with PurifyCSS is its lack of modularity. However, this is also its biggest benefit. PurifyCSS can work with any file type, not just HTML or JavaScript. PurifyCSS works by looking at all of the words in your files and comparing them with the selectors in your CSS. Every word is considered a selector, which means that a lot of selectors can be erroneously consider used. For example, you may happen to have a word in a paragraph that matches a selector in your CSS.

PurgeCSS fixes this problem by providing the possibility to create an extractor. An extractor is a function that takes the content of a file and extracts the list of CSS selectors used in it. It allows a perfect removal of unused CSS.

Traduction :

Le plus gros défaut de PurifyCSS est son manque de modularité. Cependant, c'est aussi son plus grand avantage. PurifyCSS peut fonctionner avec n'importe quel type de fichier, pas seulement HTML ou JavaScript. PurifyCSS fonctionne en examinant tous les mots de vos fichiers et en les comparant avec les sélecteurs de votre CSS. Chaque mot est considéré comme un sélecteur, ce qui signifie que de nombreux sélecteurs peuvent être considérés à tort comme utilisés. Par exemple, il se peut que vous ayez un mot dans un paragraphe qui correspond à un sélecteur dans votre CSS.

PurgeCSS corrige ce problème en offrant la possibilité de créer un extracteur. Un extracteur est une fonction qui prend le contenu d'un fichier et extrait la liste des sélecteurs CSS utilisés dans celui-ci. Il permet une suppression parfaite des CSS inutilisés.

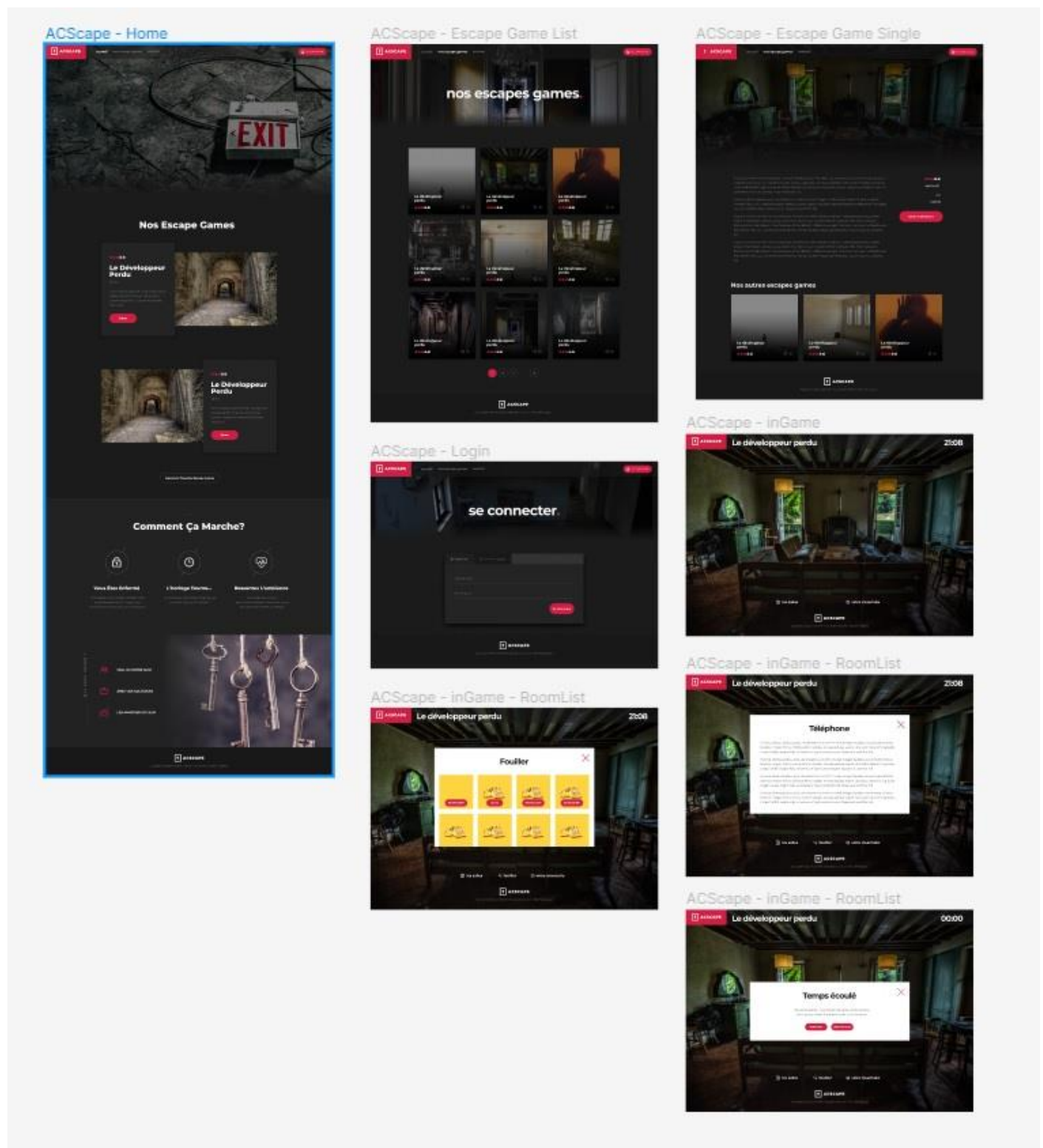
Conclusion

En conclusion, ce projet m'a permis de mettre en pratique les connaissances acquises durant ma formation et de découvrir de nouveaux horizons. Il m'a donné un véritable coup de boost pour poursuivre dans cette voie passionnante et découvrir de nouvelles technologies et outils. Je suis convaincu que cette expérience me sera bénéfique pour l'avenir, et j'ai hâte de poursuivre mon parcours en tant qu'auto-entrepreneur ou prestataire pour des agences ou des clients. Je suis confiant dans mes capacités et enthousiaste à l'idée de relever de nouveaux défis professionnels dans les années à venir.

Annexe

Maquette :

Bureau :



Mobile :



Base de données :

