

# DOSSIER DE PROJET

Hafid HADDAOUI

Développeur Web et Web mobile

Application web Le Quai Antique



# Remerciements

---

Bien que ce soit une formation à distance, et bien que je n'aie pu faire de stage en entreprise, je n'ai pas manqué d'interactions avec les élèves de Studi, ainsi qu'avec les formateurs. Je remercie toute la communauté présente sur le forum de la plateforme Studi, Je remercie tous les élèves, les formateurs, les professionnels présent sur les serveurs discord qui m'ont apporté leur aide et leur soutien.

# Sommaire

---

Remerciement.....	2
Sommaire.....	3
Introduction.....	5
1.Compétences du référentiel couvertes par le projet.....	5
1.1 Développer la partie Front-end d'une application web ou web mobile en intégrant les recommandations de sécurité.....	5
1.2 Développer la partie Back-end d'une application web ou web mobile en intégrant les recommandations de sécurité.....	5
2.Résumé du projet.....	6
Présentation du projet ECF	
1 Le cahier des charges.....	7
1.1 Contexte.....	7
1.2 Front-end.....	7
1.3 Back-end.....	8
2 Spécifications techniques.....	8
2.1 Environnement de travail.....	8
2.2 Front-end.....	9
2.2 Back-end.....	9
Réalisation de l'ECF	
1.Conception de la base de données.....	10
1.1 Le diagramme de classe (ou la méthode Merise).....	10
1.2 Le diagramme de cas d'utilisation.....	12
1.3 Le diagramme de séquence.....	13
2. Développement de l'application.....	14
2.1 Installation.....	14
2.2 Installation et configuration de la base de données.....	15
2.3 Création des entités (Models).....	15
2.4 Migrations et DQL.....	17
2.5 Création des Controller et Repository .....	18
2.6 Création de la Vue (View).....	19
2.7 Création d'un utilisateur.....	21
2.8 Création d'un formulaire d'inscription.....	22
2.9 Création d'un formulaire de connexion.....	23
3. Création d'un administrateur.....	24
3.1 Création d'un panel d'administration .....	25

## Fonctionnalité la plus représentative : La réservation

1. Présentation de la fonctionnalité.....	28
2. Création de la Modale.....	29
3. Création du formulaire.....	29
4. Création des QueryBuilder et capacité maximum du restaurant.....	30
5. Développement du système de réservation dans un Controller.....	31

## Description de la veille sur les vulnérabilités de sécurité

1. Les attaques XSS.....	35
2. Les attaques CSRF.....	35
3. Les injections SQL.....	36

## Description d'une situation de travail et d'une recherche sur un site anglophone.....36

## Extrait du site anglophone.....38

## Conclusion.....40

# Introduction

---

## 1. Compétences du référentiel couvertes par le projet

### 1.1 Développer la partie Front-end d'une application web ou web mobile en intégrant les recommandations de sécurité

- Maquetter une application ( voir wireframe en annexe )
- Réaliser une interface utilisateur web statique et adaptable
- Développer une interface utilisateur web dynamique
- Réaliser une interface utilisateur avec une solution de gestion de contenu ou e-commerce

Le projet doit être livré avec un wireframe et/ou un mockup pour la version mobile ET la version desktop, qui permet dans un premier temps de réfléchir sur la cohérence de l'interface front.

L'expérience utilisateur doit être responsive, c'est-à-dire que l'interface doit s'adapter à toute les tailles d'écran.

L'application doit intégrer des fonctionnalités dynamiques, qui permettrons à l'utilisateur d'exécuter des actions sans rechargement de la page.

Enfin, une interface doit être spécialement conçue pour l'administrateur, elle permettra à ce dernier de gérer le contenu du site.

### 1.2 Développer la partie Back-end d'une application web ou web mobile en intégrant les recommandations de sécurité

- Créer une base de données
- Développer les composants d'accès aux données
- Développer la partie backend d'une application web ou web mobile
- Elaborer et mettre en œuvre des composants dans une application de gestion de contenu ou e-commerce

Pour la base de données j'ai utilisé le service ClearDB MySQL que j'ai intégré à l'application (Add-on) sur Heroku, comme son nom l'indique c'est une base de données de type MySQL.

La partie Back-end a été développée à l'aide du Framework Symfony basé sur le modèle

MVC (Model View Controller).

J'ai utilisé la solution EasyAdmin pour concevoir l'interface d'administration.

## 2. Résumé du projet

Dans le cadre de mon ECF, j'ai eu pour mission la création de A à Z d'une application web vitrine pour un restaurant.

Mes premières réflexions se sont portées sur la conception des diagrammes qui seront plus tard nécessaire à la matérialisation de la base de données, suivi d'une étape de maquettage (wireframe) pour visualiser la partie front de l'application.

Le fonctionnement de l'application est le suivant : l'utilisateur va pouvoir parcourir la carte, les menus, les produits « préférés » des clients, réserver une table, ou encore se créer un compte.

Un autre utilisateur sera « l'hôte d'accueil » pour qui j'ai mis à disposition une solution de gestion de contenu afin qu'il puisse gérer les données du site en toute autonomie.

L'application aura une interface web statique et responsive avec HTML, CSS, et BOOTSTRAP, elle intégrera également des éléments dynamiques avec JAVASCRIPT (système de tri par catégorie pour la carte). L'outil EasyAdmin sera la solution de gestion de contenu utilisée par « l'hôte d'accueil ».

L'application sera conçue sur la base du modèle MVC avec le Framework SYMFONY, elle sera déployée en ligne sur Heroku, et la base de données sera accueillie par le service ClearDB MySQL intégré à Heroku.

Ce projet a été très intéressant, il m'a permis de passer par toutes les étapes de développement de l'application, du maquettage au déploiement de celle-ci.

# Présentation du projet ECF

---

## 1 Le cahier des charges

### 1.1 Contexte

Le Chef Arnaud Michant est un passionné des produits et des producteurs de la Savoie, après avoir ouvert deux restaurants il a décidé d'en ouvrir un troisième et souhaiterait profiter de l'impact positif que peut avoir une application web sur son chiffre d'affaires.

Il veut pouvoir transmettre au travers de cette application l'identité, et les valeurs de sa gastronomie.

### 1.2 Front-end

L'application est composée de plusieurs pages :

- Une page d'accueil qui présente les produits « favoris » des clients, et qui intègre un bouton call-to-action « réserver » ouvrant un formulaire de réservation.
- Une page pour la carte qui présente, sous forme de galerie, tous les produits du restaurant, avec la possibilité d'être triée par catégories par l'utilisateur.
- Une page menu qui présente plusieurs formules, avec chacune leur description, ainsi que leur prix.
- Une page d'inscription, qui permet de créer un compte et d'y ajouter directement des informations par défaut comme le nombre de couverts ou les allergies.
- Une page de connexion
- Une page d'administration pour la gestion du contenu de l'application

Les horaires d'ouverture du restaurant sont affichés au bas de toutes les pages.

## 1.3 Back-end

**Connexion** - Il y a deux utilisateurs sur l'application : « l'hôte d'accueil » (l'administrateur) et le client, les deux ont le même formulaire de connexion, avec pour identifiants une adresse email et un mot de passe.

**Galerie d'image** - Toutes les photos de la galerie ont un titre, les photos et leur titre peuvent être ajoutées, modifiées, ou supprimées.

**La carte** - Chaque produit est présenté avec une photo, un titre, une description, un prix, et tous ces éléments peuvent être ajoutés, modifiés, ou supprimés.

**Les menus** - Chaque menu a un titre, une ou plusieurs formules avec un prix et une description et tous ces éléments peuvent être ajoutés, modifiés, ou supprimés.

**Réservation** - Le formulaire demande de saisir le nombre de couverts, la date, l'heure prévue, et éventuellement la mention des allergies.

Un seuil pour le nombre de couvert peut être fixé.

**Inscription** - Le formulaire d'inscription permet de fixer un nombre de couvert par défaut et mentionner les allergies éventuelles, ce qui permet à l'utilisateur une fois connecté d'avoir un formulaire de réservation prérempli.

## 2. Spécifications techniques

### 2.1 Environnement de travail

**PHP Storm** - Un éditeur de code très complet, idéal pour les projets en PHP/Symfony, il prend également en charge la gestion de bases de données.

**Git & GitHub** - C'est l'outil de versioning que j'ai choisi, il me permet de déposer mon travail sur un dépôt distant, et surtout de pouvoir travailler sur de nouvelles fonctionnalités de mon code en sécurité en utilisant une branche différente. J'utilise uniquement des lignes de commandes sur le terminal Git Bash de mon éditeur PHP Storm.

**Trello** - Un outil essentiel pour ma productivité, qui me permet d'organiser mon travail par tâche et par priorité (à faire, en cours, terminé), on appelle également cette méthode la « Méthode Kanban ».

**Figma** - C'est l'outil de maquettage qui m'a permis de réaliser les wireframes.



**Draw.io** – Un autre outil en ligne que j’ai utilisé pour réaliser mes diagrammes (utilisation, séquence, classe).

## 2.2 Front-end

**HTML (Twig)** – la structure du site en HTML est rendue (Vue) par le moteur de templating Twig intégré à Symfony.

**CSS / Bootstrap** – J’ai jonglé entre le CSS et le Framework CSS Bootstrap selon mes besoins pour la réalisation de la mise en page, et du responsive.

**Javascript** – C’est le langage orienté objet que j’ai utilisé pour ajouter du dynamisme à mon application.

## 2.3 Back-end

**Symfony** – Un Framework PHP complet, basé sur le modèle MVC (Model, View, Controller), il m’a permis de me faciliter la création de mes entités (Model), du rendu de mes pages (View), et de l’interaction avec ma base de données (Controller).

**SecurityBundle** – Bundle intégré à Symfony, il permet la connexion et la navigation de manière sécurisée.

**Doctrine ORM (Object Relational Mapper)** – C’est l’outil intégré à Symfony qui m’a permis de faire le lien entre mes entités (objets) et ma base de données.

**DQL (Doctrine Query Builder)** – C’est le système de requête SQL utilisé par Symfony pour interroger ma base de données.

**EasyAdmin** – J’ai utilisé cet outil pour créer ma solution de gestion de contenu (interface administrateur) de l’application.

**Heroku** – Solution d’hébergement en ligne de mon application

**ClearDB MySQL** – Module (Add-on) que j’ai intégré à mon application en ligne sur Heroku, qui me permet d’avoir une base de données de type MySQL.

# Réalisation de l'ECF

---

## 1. Conception de la base de données

Comme il est essentiel pour la partie front d'établir une maquette (wireframe et/ou MockUp) afin de matérialiser notre réflexion avant de coder, il est encore plus indispensable de modéliser notre base de données sous forme de diagrammes, avant de créer notre système.

Pour se faire j'ai conçu les trois diagrammes suivants qui chacun ont un rôle différent.

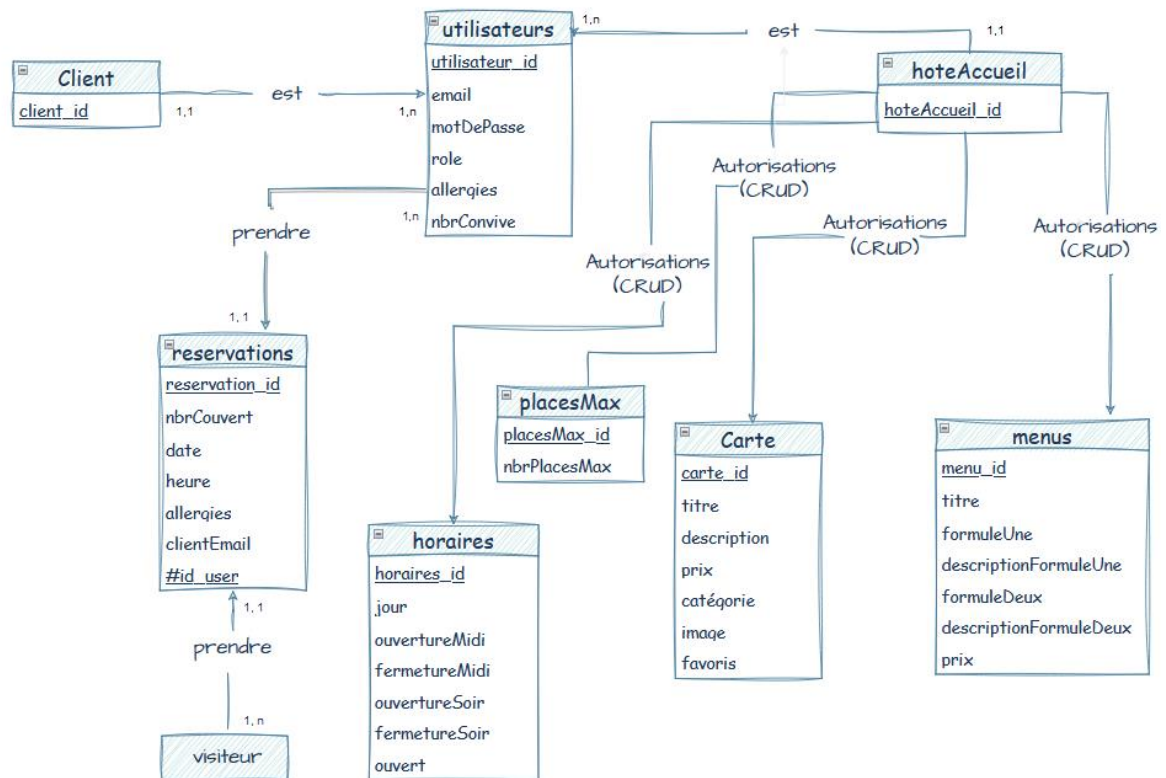
### 1.1 Le diagramme de classe (ou la méthode Merise)

La méthode Merise est une méthode de conceptualisation des données basée sur le modèle MCD (Modèle Conceptuel de Données), son but est de faciliter la communication entre tous les acteurs du processus de développement d'un projet, bien qu'ils soient étrangers au domaine du développement informatique.

Pour ma part elle m'a surtout servi à organiser mon système d'information et éviter de basculer dans un hors sujet.

Grâce à cette méthode j'ai pu déterminer les éléments suivants :

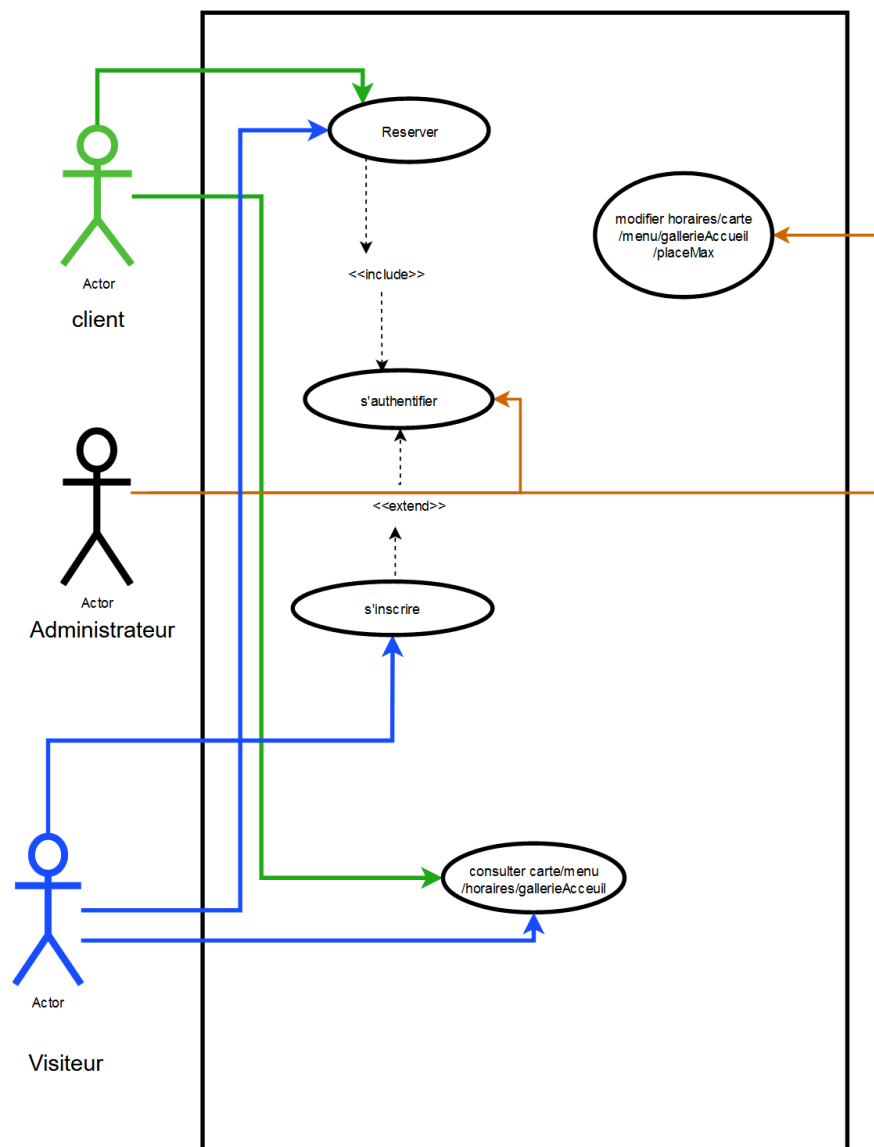
- **Les entités**, qui représentent chacune un ensemble d'information (propriétés). Elles possèdent toute un identifiant « \_id ». *Exemple : l'entité « Carte » possède un identifiant « carte\_id » et plusieurs propriétés comme par exemple « titre »*
- **Les relations**, qui sont les liaisons des entités. *Exemple : un Utilisateur « prend » une réservation, ou une réservation est « prise » par un utilisateur.*
- **Les cardinalités**, c'est le nombre de fois que l'entité participe à une relation, elle est représentée sous la forme « 1,1 », « 1, n » ou « 0, n » *Exemple : un Utilisateur « est » au minimum « 1 » client et au maximum plusieurs (« n ») clients.*



## 1.2 Le diagramme de cas d'utilisation

Le diagramme de cas d'utilisation donne une description globale du comportement de l'application, il identifie les acteurs et leur interaction avec celle-ci.

**Exemple :** l'acteur « visiteur » peut consulter la carte, le menu, les horaires, et la galerie. Mais il peut aussi Réserver sans s'inscrire, ou s'inscrire avant de réserver.

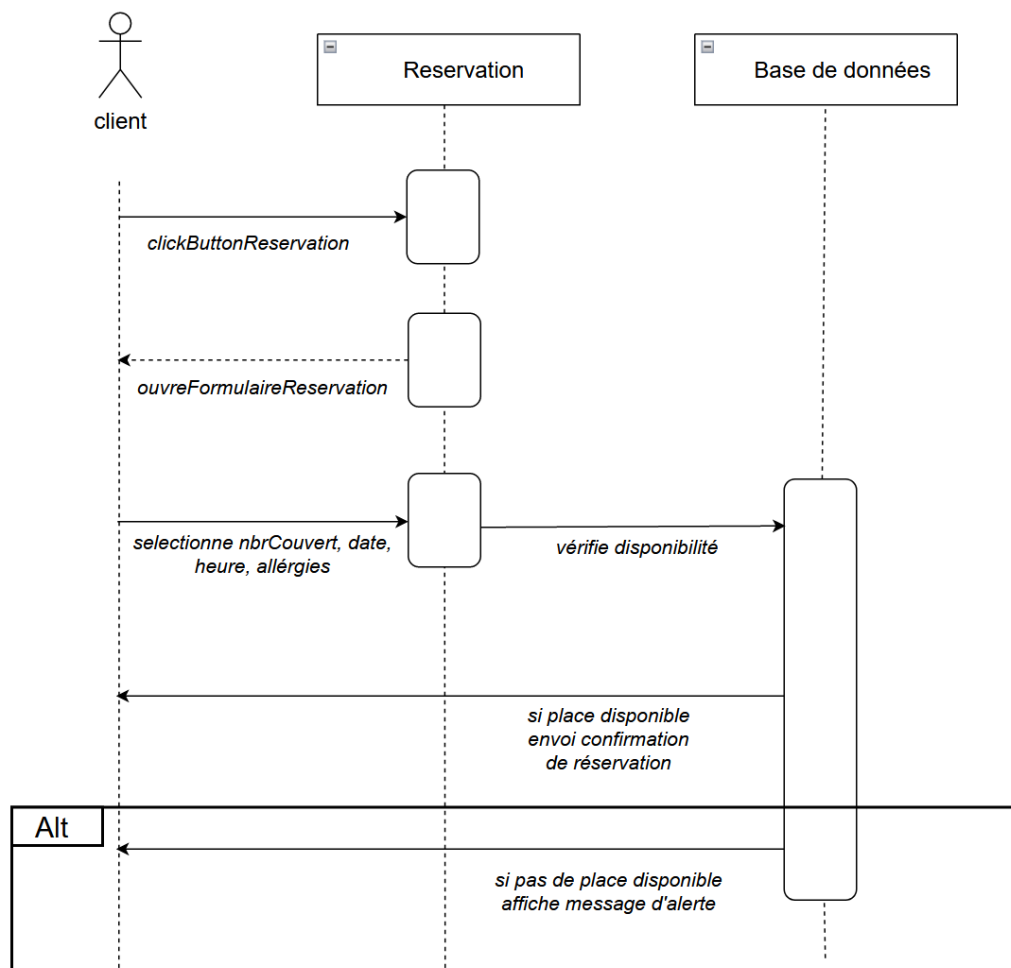


## 1.3 Le diagramme de séquence

Comme son nom l'indique ce diagramme présente une séquence en particulier, mais cette fois ci il est décrit sur une ligne de temps (de haut en bas), grâce à cela on peut prendre une action de l'utilisateur et présenter toutes les interactions qu'elle déclenche les unes après les autres.

La lecture du diagramme de séquence ci-dessous décrivant la fonctionnalité « Réservation » se fait donc simplement de haut en bas.

**Exemple :** *Le client clique sur le bouton réservation, ce qui ouvre un formulaire de réservation, peut ensuite remplir le formulaire, qui communique à son tour avec la base de données etc...*



## 2. Développement de l'application

### 2.1 Installation

Avant d'installer Symfony qui sera notre Framework pour le développement de mon application, il est nécessaire d'avoir les prérequis suivants installés sur mon ordinateur :

- **PHP 7.1 ou version ultérieure**
- **MySQL** ou un autre serveur de base de données compatible avec Symfony, je recommande **PhpMyAdmin**
- Un serveur Web (par exemple Apache ou Nginx), je recommande **Xampp**.
- **Composer**

Si tous les prérequis sont installés on peut ouvrir le terminal dans **PHP Storm** et vérifier si Symfony est prêt à être installé grâce à la commande ***symfony check:requirements***

Je lance ensuite la commande ***symfony new my\_project\_directory --version="6.2.\*" --webapp*** qui va installer la version complète de Symfony dans le répertoire que j'aurai indiqué dans la commande.

J'utilise alors le package manager **Composer** préalablement installé pour créer mon projet grâce à la commande suivante :

```
composer create-project symfony/skeleton:"6.2.*" my_project_directory
```

```
cd my_project_directory
```

```
composer require webapp
```

Le projet est maintenant initialisé, à noter que cette méthode initialise par défaut un projet **Git**, il n'y a donc pas besoin de lancer la commande ***git init***, cependant il faudra toujours créer soit même un repository distant sur **GitHub** et faire la connexion avec le projet local.

Enfin, après avoir veillé à être positionné sur le répertoire de notre projet on peut lancer l'application avec la commande ***symfony server:start*** et y accéder sur notre navigateur via l'URL ***http://localhost:8000/***

## 2.2 Installation et configuration de la base de données

Le type de base de données que j'ai choisi est le **MySQL**, le langage backend utilisé pour interagir avec est le **SQL**.

J'ai choisi d'utiliser dans un premier temps l'interface **PHPMyAdmin** pour seulement avoir une vue sur ma base de données, je n'ai en aucun cas utilisé ses fonctionnalités pour créer, lire, mettre à jour, ou supprimer des données (CRUD). Nous verrons plus tard comment j'ai procédé pour cela.

L'installation d'un serveur local (ici **Xampp**) est également nécessaire pour prendre en charge notre application et en même temps avoir accès directement à l'interface PHPMyAdmin.

Une fois ceci fait, l'outil qui va nous permettre dans **Symfony** d'interagir avec la base de données est **Doctrine**, il en existe d'autres, mais tous sont ce qu'on appelle des **ORM** (Object Relational Mapper).

Je lance la commande ***composer require symfony/orm-pack*** qui va installer l'ORM, ainsi que la commande ***composer require --dev symfony/maker-bundle*** qui va installer un bundle me permettant de générer du code facilement.

Je peux à présent configurer la connexion avec la base de données dans le fichier **.env**, je décommente l'URL de la base de données **MySQL** et y insère les informations nécessaires (Nom de la BDD, adresse locale, version et nom du SGBD...)

```
30 # DATABASE_URL="sqlite:///kernel.project_dir%/var/data.db"
31 DATABASE_URL="mysql://root:@127.0.0.1:3306/le_Quai_Antique?serverVersion=10.4.25-MariaDB"
32 # DATABASE_URL="postgresql://app:!ChangeMe!@127.0.0.1:5432/app?serverVersion=14&charset=utf8"
33 ###< doctrine/doctrine-bundle ###
```

Je peux maintenant créer une base de données grâce à **Doctrine** avec la commande ***symfony console doctrine:database:create***

Tout est en place pour créer nos entités (tables) définies précédemment dans notre diagramme de classe.

## 2.3 Création des entités (Models)

Pour créer mes entités j'utilise la commande ***symfony console make:entity*** du **MakerBundle**, j'ai répondu ensuite à quelques questions sur la nature de mon entité désirée (nom, nom de la propriété, type...).

Comme je l'ai dit plus haut le **MakerBundle** va me générer du code facilement, et c'est ce qu'il

fait ici car une fois mon entité créée, il va en réalité générer lui-même une classe PHP dans un dossier **/Entity**, que je peux modifier si je le veux.

***Exemple :** l'entité Carte a ses champs définis comme ceci ...*

```
1  <?php
2
3  namespace App\Entity;
4
5  use ...
6
7
8
9
10
11
12  #[ORM\Entity(repositoryClass: CarteRepository::class)]
13  #[Vich\Uploadable]
14  class Carte
15  {
16      #[ORM\Id]
17      #[ORM\GeneratedValue]
18      #[ORM\Column]
19      private ?int $id = null;
20
21      #[Vich\UploadableField(mapping: 'photos', fileNameProperty: 'imageName')]
22      private ?File $imageFile = null;
23
24      #[ORM\Column(type: 'string', nullable: true)]
25      private ?string $imageName = null;
26
27      #[ORM\Column(length: 255)]
28      private ?string $titre = null;
29
30      #[ORM\Column(length: 255)]
31      private ?string $description = null;
32
33      #[ORM\Column]
34      private ?int $prix = null;
35
36
37      #[ORM\Column(nullable: true)]
38      private ?bool $favoris = null;
39
40      #[ORM\Column(length: 255)]
41      private ?string $categorie = null;
42  }
```

Et ses **getter** et **setter** comme ceci...

```
public function getId(): ?int
{
    return $this->id;
}

public function getTitre(): ?string
{
    return $this->titre;
}

public function setTitre(string $titre): self
{
    $this->titre = $titre;

    return $this;
}

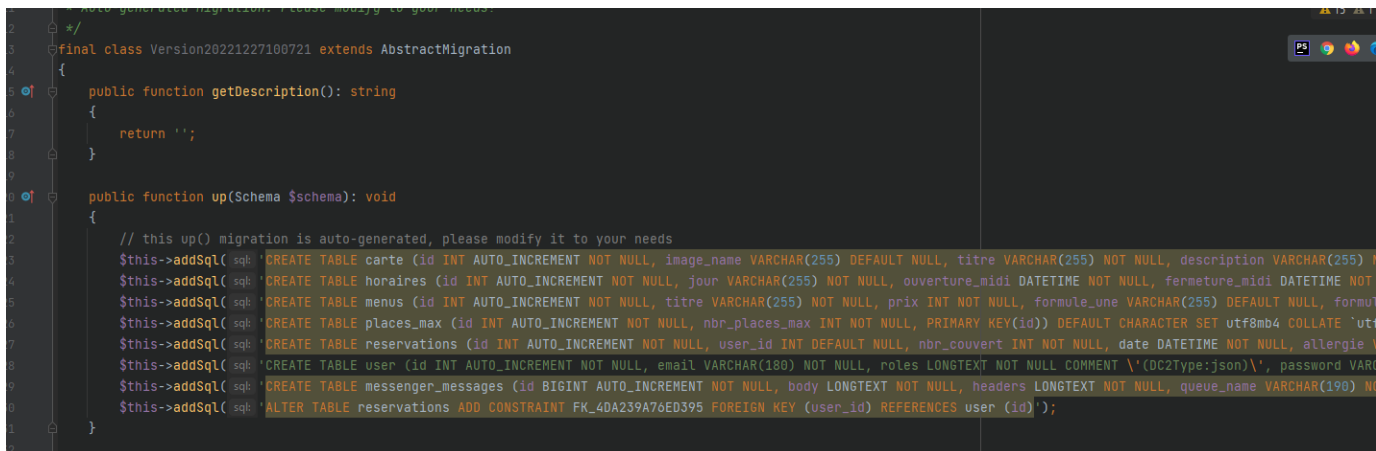
public function getDescription(): ?string
{
    return $this->description;
}
```



## 2.4 Migrations et DQL

A ce stade l'entité n'est pas créée en tant que table dans la base de données, l'étape de la migration va s'en charger. Pour « écrire » dans la base de données et créer notre première table, qui sera le reflet de notre entité il devrions normalement faire nos requêtes avec le langage **SQL** nous-même, cependant **Doctrine** et le **MakerBundle** vont se charger eux même d'établir ces requêtes avec « langage » propre, c'est ce qu'on appelle le **DQL** (Doctrine Query Language).

On utilise la commande ***symfony console make:migration*** pour cela, ce qui va générer les requêtes dans le dossier **/migration**



```
2  */
3  final class Version20221227100721 extends AbstractMigration
4  {
5      public function getDescription(): string
6      {
7          return '';
8      }
9
10     public function up(Schema $schema): void
11     {
12         // this up() migration is auto-generated, please modify it to your needs
13         $this->addSql('CREATE TABLE carte (id INT AUTO_INCREMENT NOT NULL, image_name VARCHAR(255) DEFAULT NULL, titre VARCHAR(255) NOT NULL, description VARCHAR(255) NOT NULL, PRIMARY KEY(id))');
14         $this->addSql('CREATE TABLE horaires (id INT AUTO_INCREMENT NOT NULL, jour VARCHAR(255) NOT NULL, ouverture_midi DATETIME NOT NULL, fermeture_midi DATETIME NOT NULL, PRIMARY KEY(id))');
15         $this->addSql('CREATE TABLE menus (id INT AUTO_INCREMENT NOT NULL, titre VARCHAR(255) NOT NULL, prix INT NOT NULL, formule_une VARCHAR(255) DEFAULT NULL, formule_deux VARCHAR(255) DEFAULT NULL, PRIMARY KEY(id))');
16         $this->addSql('CREATE TABLE places_max (id INT AUTO_INCREMENT NOT NULL, nbr_places_max INT NOT NULL, PRIMARY KEY(id)) DEFAULT CHARACTER SET utf8mb4 COLLATE `utf8mb4_unicode_ci`');
17         $this->addSql('CREATE TABLE reservations (id INT AUTO_INCREMENT NOT NULL, user_id INT DEFAULT NULL, nbr_couvert INT NOT NULL, date DATETIME NOT NULL, allergie VARCHAR(255) NOT NULL, PRIMARY KEY(id), INDEX(user_id), FOREIGN KEY (user_id) REFERENCES user (id))');
18         $this->addSql('CREATE TABLE user (id INT AUTO_INCREMENT NOT NULL, email VARCHAR(180) NOT NULL, roles LONGTEXT NOT NULL COMMENT \'(DC2Type:json)\', password VARCHAR(255) NOT NULL, PRIMARY KEY(id))');
19         $this->addSql('CREATE TABLE messenger_messages (id BIGINT AUTO_INCREMENT NOT NULL, body LONGTEXT NOT NULL, headers LONGTEXT NOT NULL, queue_name VARCHAR(190) NOT NULL, PRIMARY KEY(id))');
20         $this->addSql('ALTER TABLE reservations ADD CONSTRAINT FK_4DA239A76ED395 FOREIGN KEY (user_id) REFERENCES user (id)');
21     }
22 }
```

*Un exemple de requête DQL, ici toute les tables sont prêtes à être créées.*

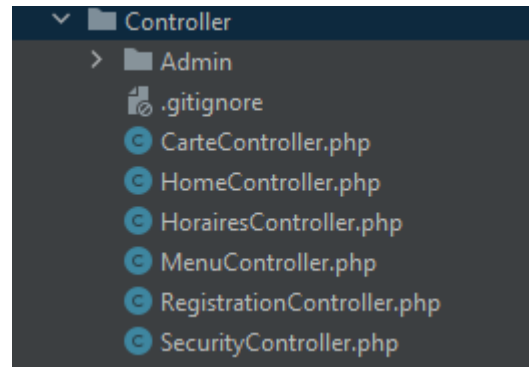
Attention, à ce stade la requête est seulement « prête » mais pas envoyée, pour cela on lance la commande ***symfony console doctrine:migrations:migrate***

L'avantage de ce système c'est que l'on garde toutes les migrations en mémoire, on peut ainsi retourner vers une version précédente de la migration.

## 2.5 Création des Controller et Repository

Après avoir créé les modèles comme le pattern MVC évoqué plus haut le requiert, je me suis attelé à créer les Controller, ceux-ci sont les composant dans lesquels je développe mon code. Je crée plusieurs Controller, chacun ayant pour rôle de développer une partie de l'application.

**Exemple :** je créer un Controller pour la page d'accueil, un autre pour le menu, les horaires, l'inscription etc...



Encore une fois un Controller n'est autre qu'une class PHP, prenons l'exemple du Controller **CarteController** ci-dessous qui doit se charger d'afficher les produits de la carte du restaurant, il est composé en partant du haut :

- D'un **namespace**, autrement dit le nom du chemin donné à ce Controller dans le système
- Des **services** utilisés par ce Controller
- De la **Classe** elle-même dans laquelle on retrouve une ou plusieurs fonction (action), chaque fonction à une **Route** qui servira à appeler cette fonction via une **requête Http**. Dans chaque fonction je développe mon code, et chaque fonction va retourner quelque chose, ici elle va renvoyer un rendu (Render) dont on se servira pour créer notre Vue (Model **Vue** Controller)

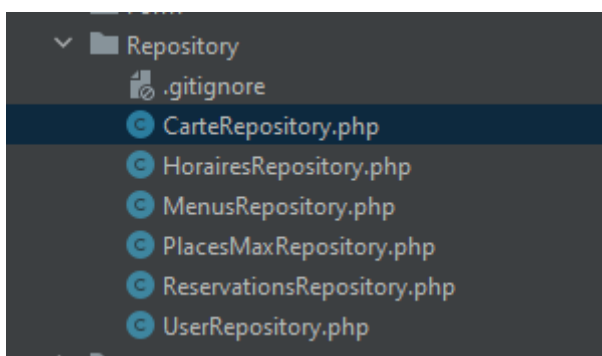
```
1 <?php
2
3 namespace App\Controller;
4
5 use App\Repository\CarteRepository;
6 use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
7 use Symfony\Component\HttpFoundation\Response;
8 use Symfony\Component\Routing\Annotation\Route;
9
10 class CarteController extends AbstractController
11 {
12
13     // Action pour afficher entrées, plats, desserts
14     #[Route('/carte', name: 'app_carte')]
15     public function index(CarteRepository $carteRepository): Response
16     {
17         // Récupération tout le contenu de la table carte en base de données
18         $plat = $carteRepository->findAll();
19         // Affiche la vue Twig avec le contenu de la table carte
20         return $this->render('carte/index', [
21             'plats' => $plat,
22         ]);
23     }
24 }
```

Comme je l'ai dit plus haut, ce Contrôleur doit se charger d'afficher les produits de la carte, mais pour cela il a besoin d'interroger la base de données, et de récupérer les produits de la table **Carte**, lorsque nous avons créé plus tôt nos entités, celles-ci ont généré automatiquement un **Repository**.

Un **Repository** est une classe qui permet justement de gérer les données d'une table

spécifique dans la base de données. Il permet de faire des opérations CRUD (create, read, update, delete) sur les données de cette table, ainsi que de définir des méthodes de requête personnalisées (**Query Builder**).

J'ai donc appelé dans mon Contrôleur mon **CarteRepository** et utilisé la fonction *findAll()* du **Repository** qui va récupérer toutes les données de la table **Carte**, je mets le résultat dans une variable et la passe en paramètre de la fonction *render()* et y indique également l'emplacement de ma **Vue**.



## 2.6 Création de la Vue (View) et du système de tri par catégorie

Pour la vue j'ai utilisé le moteur de template par défaut de **Symfony** qui est **Twig**. Il permet d'organiser ses pages par block (body, title, footer...) et propose de nombreuses fonctions (boucles, filtres...).

J'ai donc créé ma mise en page comme je le souhaite, et comme dans n'importe quel fichier HTML, j'y ajoute ensuite mes données de la manière ci-dessous, j'utilise une boucle **for** qui va chercher chaque plat (produit) dans mon tableau plat que j'ai récupéré précédemment avec la fonction *findAll()* du **Repository**.

Enfin je n'ai plus qu'à afficher ou je veux les données que je souhaite précisément, en récupérant le champ souhaité avec *{{plat.titre}}* par exemple pour afficher le titre du plat.

```
{% for plat in plats %}
    {% if plat.categorie == "entrée" %}
        <div id="card">
            
            <div class="card-info d-flex flex-column justify-content-center align-items-center">
                <h3 class="card-title">{{ plat.titre }}</h3>
                <p class="description">{{ plat.description }}</p>
                <p class="price">{{ plat.prix }}€</p>
            </div>
        </div>
    {% endif %}
{% endfor %}
```

Autre exemple des possibilités de **Twig**, j'ai également utilisé une condition **if** qui me permet de n'afficher que les produits dans la catégorie « entrée ».

```
{% if plat.categorie == "entrée" %}
```

Pour le responsive, j'ai utilisé des media queries :

```
#card {
  display: flex;
  flex-direction: row;
  flex-wrap: wrap;
  margin: 20px;
  -webkit-box-shadow: 1px 1px 9px -3px #C2C2C2;
  box-shadow: 1px 1px 9px -3px #C2C2C2;
  width: 600px;
  height: 300px;
  transition: transform 0.3s;
}

@media screen and (max-width: 610px) {
  #card {
    display: flex;
    flex-direction: column;
    align-items: center;
    flex-wrap: nowrap;
    margin: 20px auto;
    -webkit-box-shadow: 1px 1px 9px -3px #C2C2C2;
    box-shadow: 1px 1px 9px -3px #C2C2C2;
    width: 350px;
    height: 600px;
    transition: transform 0.3s;
  }
}
```

Concernant la carte j'ai dû mettre en place un système de tri par catégorie (entrées, plats, desserts), pour cela j'ai utilisé javascript, j'ai donc créé une feuille de script javascript **script.js**.

J'y ai développé mon code dans lequel j'ai ciblé les éléments HTML que je voulais rendre dynamique. A ces éléments j'ai ajouté un évènement click (eventListener).

```

// système filtre par catégories

// récupère-les buttons
const filterButtons = Array.from(document.querySelectorAll( selectors: '.btn-filter'));

// event listener
filterButtons.forEach(button => {
  button.addEventListener('click', e => {
    // récupère la valeur de l'attribut data-filter du button
    const filter = e.target.dataset.filter;
    // récupère-les éléments à modifier par la suite
    const items = document.querySelectorAll( selectors: '.categorie-section');

    // tout afficher si all est sélectionné
    if (filter === 'all') {
      items.forEach( callbackfn: item => item.style.display = 'block');
    } else {
      // sinon cacher tous les éléments
      items.forEach( callbackfn: item => item.style.display = 'none');

      // et ne montrer que celui qui est sélectionné, pour ça on sélectionne tous les attributs data-category dont la valeur correspond
      // à la catégorie sélectionnée par l'utilisateur et on affiche les éléments.
      const selectedItem = document.querySelector( selectors: `[data-category="${filter}"]`);
      selectedItem.style.display = 'block';
    }
  });
});

```

## 2.7 Création d'un utilisateur

L'application donne la possibilité de créer un compte utilisateur et un compte administrateur. Le compte utilisateur va permettre au client d'entrer des informations par défaut lors de son inscription, et ces informations prérempliront le formulaire de réservation afin de faire gagner du temps à l'utilisateur.

Le compte administrateur est pour l'Hôte d'accueil, lui seul y aura accès, il aura un panel d'administration à sa disposition pour faire appel aux opérations CRUD (**C**reate **R**ead **U**pdate **D**ele) qu'il désire sur les entités de la base de données.

Première étape, j'ai eu besoin d'installer le **SecurityBundle** toujours à l'aide de **Composer** et la commande ***composer require symfony/security-bundle***.

J'ai ensuite créé une entité User grâce au **MakerBundle** avec la commande ***symfony console make :user***, je réponds à plusieurs questions concernant le type d'identifiant par exemple (nom, email ...), et pour la sécurité, **Symfony** me propose de hasher le mot de passe lui-même. Une fois le questionnaire validé, **Symfony** va créer une entité **User.php** et un fichier de **configuration config/packages/security.yaml**.

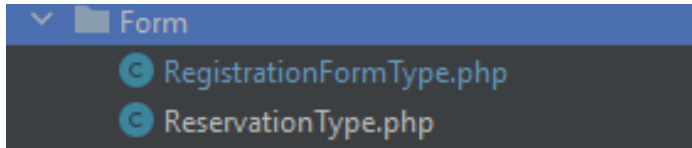
Je finis par faire une migration avec les commandes suivantes :

***Symfony console make : migration***

***Symfony console doctrine: migrations: migrate***

## 2.8 Création d'un formulaire d'inscription

Seconde étape, je crée un formulaire d'inscription avec la commande ***symfony console make :registration-form***, et après avoir validé toutes les questions pour la création d'un formulaire d'inscription, **Symfony** va créer un **RegistrationFormType.php** dans un dossier **/Form** qui accueillera par la suite tous nos formulaires.



Ci-dessous le formulaire **RegistrationFormType.php** qui utilise le service **FormBuilderInterface**, permet de construire son formulaire en ajoutant des champs et en les configurant.

```
<?php

namespace App\Form;

use ...

class RegistrationFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $builder
            // Ajout des champs du formulaire et leur configuration.
            ->add( child: 'email')
            ->add( child: 'agreeTerms', type: CheckboxType::class, [
                'mapped' => false,
                'constraints' => [
                    new IsTrue([
                        'message' => 'You should agree to our terms.',
                    ]),
                ],
            ])
            ->add( child: 'plainPassword', type: PasswordType::class, [
                // instead of being set onto the object directly,
                // this is read and encoded in the controller
                'mapped' => false,
                'attr' => ['autocomplete' => 'new-password'],
                'label' => 'Mot de passe',
                'constraints' => [
                    new NotBlank([
                        'message' => 'Please enter a password',
                    ]),
                ],
            ]),
    }
}
```

Je pourrai à ce stade l'utiliser tel quel, mais je préfère améliorer la sécurité de ce dernier. En effet, par défaut **Symfony** va ajouter une contrainte de longueur du mot de passe et obligera l'utilisateur à entrer un mot de passe, mais à cela j'ai ajouté une contrainte **Regex** sur le mot de passe qui m'a permis d'améliorer la sécurité.

La contrainte **Regex** que j'ai ajouté ci-dessous oblige l'utilisateur à entrer un mot de passe de 8 caractères avec une majuscule, un minuscule, un chiffre et un caractère spécial

```
new Regex( pattern: '/^(?=.*?[A-Z])(?=.*?[a-z])(?=.*?[0-9])(?=.*?[#?!@$%^&*~]).{8,}$/',  
    message: "Il faut un mot de passe de 8 caractère avec 1 majuscule, 1 minuscule, 1 chiffre et 1 caractère spécial")  
],
```

## 2.9 Création d'un formulaire de connexion

Maintenant que j'ai créé mon User et mon formulaire d'inscription, il me faut maintenant créer le formulaire de connexion.

J'entre la commande ***symfony console make :auth***, je réponds aux questions et **Symfony** va alors créer un fichier **LoginAuthenticator.php** dans un nouveau dossier **/Security** qui est une classe, et qui va me permettre de configurer l'authentification, comme par exemple la redirection après que la connexion soit faite.

**Symfony** va aussi générer un fichier **login.html.twig** (ci-dessous) dans un nouveau dossier **/templates/security** pour afficher le formulaire.

```
{% extends 'base.html.twig' %}  
  
{% block title %}Log in!{% endblock %}  
  
{% block body %}  
    <div class="container w-75 h-100 mt-4 d-flex flex-column align-items-center justify-content-center">  
        <form method="post">  
            {% if error %}  
                <div class="alert alert-danger">{{ error.messageKey|trans(error.messageData, 'security') }}</div>  
            {% endif %}  
  
            {% if app.user %}  
                <div class="mb-3">  
                    You are logged in as {{ app.user.userIdentifier }}, <a href="{{ path('/logout') }}">Logout</a>  
                </div>  
            {% endif %}  
  
            <h1 class="h3 mb-3 font-weight-normal">Se connecter</h1>  
            <label for="inputEmail">Email</label>  
            <input type="email" value="{{ last_username }}" name="email" id="inputEmail" class="form-control" autocomplete="email" required autofocus>  
            <label for="inputPassword">Mot de passe</label>  
            <input type="password" name="password" id="inputPassword" class="form-control" autocomplete="current-password" required>  
  
            <input type="hidden" name="_csrf_token" value="{{ csrf_token('authenticate') }}">  
            >  
  
            <div class="checkbox mb-3">  
                <label>  
                    <input type="checkbox" name="_remember_me"> Se souvenir de moi  
                </label>  
            </div>  
  
            <button class="btn btn-lg btn-primary mt-4" type="submit">  
                Soumettre  
            </button>  
        </form>  
    </div>
```

Pour en revenir aux possibilités que **Twig** nous propose, une fois qu'un utilisateur est connecté il faudra bien sûr que cela soit indiqué dans la barre de navigation, en lui proposant de se déconnecter. Pour cela il faut jouer avec l'affichage **Twig** et utiliser la fonction ***is\_granted()*** qui permet d'indiquer en paramètre dans quel cas on doit afficher tel bloc de code , voici un exemple :

```
{% if is_granted('IS_AUTHENTICATED_FULLY') %}
    <li class="nav-item">
        <a class="nav-link" href="{{path('/logout')}}">Se déconnecter</a>
    </li>
{% endif %}
```

Ici, si l'utilisateur est connecté, alors on affiche le lien « se déconnecter » dont la route active la fonction **logout** du **Securitycontroller**.

### 3. Création d'un administrateur

Maintenant que nous avons un User qui va pouvoir consulter le contenu du site, il nous faut un administrateur (l'Hôte d'accueil) qui va gérer tout ce contenu dans une interface de gestion de contenu (panel d'administration).

Je crée l'administrateur via commande **SQL** en suivant les étapes suivantes :

1. J'ouvre mon terminal et accède au répertoire d'installation de MySQL, situé dans **C:\xampp\mysql\bin** pour moi qui utilise [Xampp](#)
2. Je me connecte à **MySQL** en utilisant mes informations de connexion (nom d'utilisateur et mot de passe) :

```
mysql -u username -p
```

3. Lorsque je suis invité à entrer mon mot de passe, je le tape et le valide. S'il n'y a pas de mot de passe j'appuie sur entrée simplement.
5. Je sélectionne la base de données à laquelle je souhaite accéder en utilisant la commande **USE**:

```
USE nom_de_la_base_de_données;
```

6. Création d'un utilisateur avec le rôle admin en utilisant la commande **INSERT INTO**



```
INSERT INTO user (username, password, roles) VALUES ('admin',  
'$2y$10$TKh8H1.PfQx37YgCzwiKb.KjNyWgaHb9cbcoQgdIVFiyg7B77UdFm',  
['ROLE_ADMIN']);
```

*Note : Pour hascher le mot de passe j'ai utilisé l'utilitaire en ligne [Bcrypt](#)*

### 3.1 Création du panel d'administration

Pour la création du panel d'administration j'ai opté pour l'outil **EasyAdmin**, un bundle Symfony qui permet de créer des interfaces d'administration pour les applications web. Il simplifie la création de pages d'administration en fournissant des fonctionnalités telles que la gestion des entités, la création de formulaires et la visualisation des données en utilisant un simple fichier de configuration.

Pour l'installer j'utilise la commande ***composer require easycorp/easyadmin-bundle***.

Lorsque l'installation est terminée je lance la création d'un dashboard avec la commande ***symfony console make : admin :dashboard***.

A ce stade **Symfony** a généré un dossier **/src/Controller/Admin** dans lequel je vais retrouver un fichier **DashboardController.php** qui va me permettre de personnaliser la page d'accueil de l'interface d'administration en y ajoutant du contenu

```
<?php  
  
namespace App\Controller\Admin;  
  
use ...  
  
class DashboardController extends AbstractDashboardController  
{  
  
    #[Route('/admin', name: 'admin')]  
    public function index(): Response  
    {  
        return $this->render( view: 'admin/dashboard');  
    }  
  
    public function configureDashboard(): Dashboard  
    {  
        return Dashboard::new()  
            ->setTitle( title: 'Mon tableau de bord');  
    }  
}
```

*Exemple de Dashboard*

Il faut à présent du contenu à gérer dans ce Dashboard, ce que nous appelons les CRUD Controllers, c'est-à-dire les Controller qui vont nous permettre de gérer une entité

(créer, lire, modifier, supprimer).

Prenons l'exemple de l'entité **Carte**, On procède avec la commande ***symfony console make :admin :crud***, on me demande alors de choisir l'entité à gérer, je termine la création du CRUD, ce qui va me générer un fichier **CarteCrudController.php**

```
<?php

namespace App\Controller\Admin;

use ...

class CarteCrudController extends AbstractCrudController
{
    public static function getEntityFqcn(): string
    {
        return Carte::class;
    }

    public function configureFields(string $pageName): iterable
    {
        return [
            TextField::new('titre'),
            TextField::new('description'),
            IntegerField::new('prix'),
            ChoiceField::new('categorie')->setChoices([
                'entrée' => 'entrée',
                'plat' => 'plat',
                'dessert' => 'dessert'
            ]),
            BooleanField::new('favoris'),
            // TextField::new('imageFile')->setFormType(VichImageType::class),
            ImageField::new('imageName')->setBasePath('path: '/uploads/photos')->setUploadDir('uploadDirPath: 'public/uploads/photos'),
        ];
    }
}
```

*Je peux ajouter les champs que je veux dans ce fichier et les configurer.*

Pour finir je dois intégrer mon **CRUD** à mon Dashboard comme ceci :



# Fonctionnalité la plus représentative : La réservation

---

## 1. Présentation de la fonctionnalité

La fonctionnalité réservation a été la plus importante et la plus complexe me concernant. L'utilisateur, qu'il soit un simple visiteur ou un client avec un compte, peut prendre une réservation en indiquant les éléments suivants :

**Le nombre de couvert** - il est limité par jour et par service ( midi et soir )  
c'est une information qui est prérempli si celle-ci a été entrée lors de l'inscription

**La date**

**L'heure** – il y a deux plages horaires , pour le midi et pour le soir , à intervalle de 15 min ,  
l'utilisateur peut réserver jusqu'à 1h avant la fermeture du restaurant.

**Les allergies** – l'utilisateur peut indiquer les éventuelles allergies, c'est une information  
qui est prérempli si celle-ci a été entrée lors de l'inscription

Le formulaire de réservation peut être affiché via un **Call-To-Action** sur la page d'accueil, et j'ai choisi de l'afficher sous forme de modale.

Après validation du formulaire un message « **flash** » est affiché pour confirmer la réservation ou signaler s'il n'y a plus de place disponible :



Merci, votre réservation a bien été prise en compte



## 2. Création de la Modale

J'ai créé la modale avec Bootstrap et l'ai intégré dans le Template de la page d'accueil comme ceci.

```
<button type="button" class="btn btn-info btn-lg m-5" data-bs-toggle="modal" data-bs-target="#modal1">Reservez</button>

{# Modal reservation#}

<div class="modal fade" id="modal1">
  <div class="modal-dialog">
    <div class="modal-content">
      <div class="modal-header">
        <h5 class="modal-title" aria-labelledby="modal title">
          Reservez
        </h5>
        <button class="btn-close" data-bs-dismiss="modal" aria-label="close"></button>
      </div>
      <div class="modal-body" aria-describedby="content">

        <p>
          {{ form_start(form, {'autocomplete' : 'off'}) }}
          {{ form_row(form.nbrCouvert) }}
          {{ form_row(form.date) }}
          {{ form_row(form.heure, {'empty_data' : '14:00'}) }}
          {{ form_row(form.allergie) }}
          {{ form_end(form) }}
        </p>
      </div>
    </div>
  </div>
</div>
```

## 3. Création du formulaire

Je crée le formulaire avec la commande ***Symfony console make :form***, je lui donne un nom, et choisis la classe à laquelle ce formulaire sera liée.

**Symfony** me génère alors un **ReservationType.php** dans le dossier **/src/Form**.

```

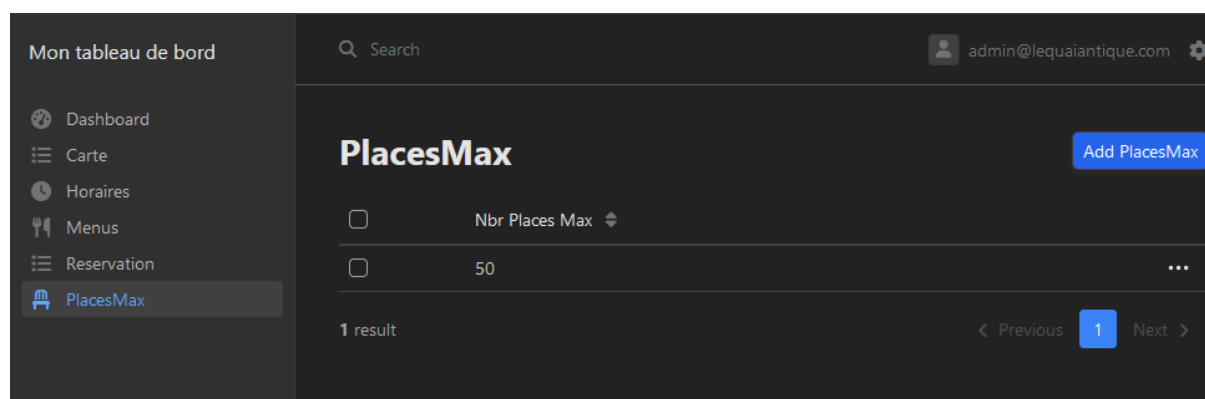
class ReservationType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $builder
            ->add( child: 'nbrCouvert', type: IntegerType::class, [
                'label' => 'Nombre de couvert'
            ])
            ->add( child: 'date', type: DateType::class, [
                'placeholder' => [
                    'year' => 'Year',
                    'month' => 'Month',
                    'day' => 'Day',
                ],
                'model_timezone' => 'Europe/Paris',
                'data' => new \DateTime(),
            ])
            ->add( child: 'heure', type: TimeType::class,
                [
                    'input_format' => 'H:m',
                    'input' => 'datetime',
                    'widget' => 'choice',
                    'hours' => ['12', '13', '14', '19', '20', '21'],
                    'minutes' => ['00', '15', '30', '45']
                ]
            )
            ->add( child: 'allergie', type: TextType::class, [
                'required' => false,
                'label' => 'Eventuelles allergies',
            ])
    }
}

```

*Je défini mes champs, et leur ajoute des options tel qu'un label, un format pour la date, l'heure, ou encore faire en sorte qu'un champ comme les allergies ne sois pas obligatoire avec l'option « required » sur false.*

## 4. Création des QueryBuilder et capacité maximum du restaurant

Le système de réservation m'a demandé de fixer une limite pour le nombre de couvert par jour, et ceci pour le service du midi et du soir. Cette limite peut être fixée par l'administrateur, j'ai donc créé une entité **PlaceMax.php** et je lui créé un **PlacesMaxCrudController.php**



*L'administrateur peut fixer le nombre de place maximum du restaurant*

Lorsque le formulaire de réservation sera soumis, il faudra vérifier s'il y a des places disponibles selon le choix de la date et de l'heure de l'utilisateur. Pour cela j'ai décidé de créer deux **QueryBuilder**, un pour compter le nombre de couvert à une date donnée pour le service du midi et un autre pour le service du soir.

```
// Compte le nombre de couverts à une date donnée pour le service du midi
public function countNbrCouvertDateMidi(string $date, string $heure ): int
{
    $qb = $this->createQueryBuilder( alias: 'r')
    ->select( select: 'SUM(r.nbrCouvert)')
    ->where( predicates: 'r.date = :date')
    ->andWhere('r.heure < :heure')
    ->setParameter( key: 'date', $date)
    ->setParameter( key: 'heure', $heure);

    return (int) $qb->getQuery()->getSingleScalarResult();
}
```

*Ce QueryBuilder va se charger de retourner un nombre qui sera la somme des nombres de couvert à la date et l'heure qu'on lui passera en paramètre (pour le service du midi).*

## 5. Développement du système de réservation dans un Controller

J'ai développé la logique du système de réservation dans le **HomeController**, car ma réservation étant intégré dans une modale elle se trouve donc dans la même action qui récupère et affiche les plats favoris.

Voici les étapes de développement :

- Je récupère le gestionnaire d'entités

```
// Récupère le gestionnaire d'entité
$entityManager = $managerRegistry->getManager();
```

- Je récupère le nombre de couvert maximum fixé dans l'entité **PlaceMax** dont j'ai parlé plus haut.

```
// Récupère le nombre de couverts maximum fixé en base de données dans la table PlacesMax
$maxReservationPerDay = $placesMaxRepository->findOneBy(['id' => '1']); // Méthode pour récupérer
$maxReservationPerDayValue = $maxReservationPerDay->getNbrPlacesMax();
```

- Je crée une instance de l'entité **Réservation**

```
// Création d'une nouvelle instance de l'entité Reservations
$reservation = new Reservations();
$reservation->setDate(new \DateTime()); // Permet de mettre une date par défaut au formulaire de réservation
$reservation->setHeure(new \DateTime()); // Permet de mettre une heure par défaut au formulaire de réservation
```

- Je relie cette instance au formulaire de **ReservationType.php** que j'ai créé

```
// Création du formulaire et liaison avec l'entité correspondante
$form = $this->createForm( type: ReservationType::class, $reservation);
```

- A présent je peux récupérer les données du formulaire

```
// Recuperation des données du formulaire
$form->handleRequest($request);
$data = $form->getData();
$reservationDate = $data->getDate();
$reservationHeure = $data->getHeure();
$nbrCouvertSelectionne = $data->getnbrCouvert();
```

- Maintenant je fais appel au **QueryBuilder** que j'ai créé en y passant en paramètre la date et l'heure saisies dans le formulaire par le client. Grâce à ça j'ai le nombre de couvert en base de données correspondant à la date et l'heure que l'utilisateur a choisi.

```
// Recuperation du nombre de couverts à une date sélectionnée pour le service du midi
$nbrCouvertMidi = $reservationsRepository->countNbrCouvertDateMidi($reservationDate, $reservationHeure );

// Recuperation du nombre de couverts à une date sélectionnée pour le service du soir
$nbrCouvertSoir = $reservationsRepository->countNbrCouvertDateSoir($reservationDate, $reservationHeure);
```

Je n'ai plus qu'à traiter le formulaire comme je le souhaite. Décortiquons les éléments qui composent le traitement du formulaire :



**Les conditions** – le formulaire doit être soumis, valide (champs obligatoire, types de caractères et de champs), et il doit y avoir assez de place à la date et l'heure sélectionnée par l'utilisateur.

```
// Vérifie si formulaire valide, et si assez de place à la date et l'heure sélectionnée
if ($form->isSubmitted()
    && $form->isValid()
    && $maxReservationPerDayValue >= ($nbrCouvertMidi + $nbrCouvertSelectionne)
    && $maxReservationPerDayValue >= ($nbrCouvertSoir + $nbrCouvertSelectionne))
```

Si les conditions ne sont pas remplies je retourne un message **flash** pour alerter l'utilisateur qu'il n'y a plus de place à la date et à l'heure qu'il a sélectionné.

```
// Sinon affiche un message d'erreur.
elseif ($maxReservationPerDayValue < ($nbrCouvertMidi + $nbrCouvertSelectionne) || $maxReservationPerDayValue < ($nbrCouvertSoir + $nbrCouvertSelectionne) ) {
    dd($nbrCouvertMidi, $nbrCouvertSoir);
    $this->addFlash( type: 'full', message: 'Il n\'y a plus de place disponible à cette date');
    return $this->redirectToRoute( route: '/' );
}
```

**La persistance des données** - j'utilise l'**entityManager** pour « **persister** » les données recueillies (pas encore envoyée à ce stade) et les envoyer en BDD ( flush ), j'en profite pour ajouter un message **flash** qui s'affichera dans la **Vue** pour confirmer à l'utilisateur que sa réservation a bien été prise en compte .

```
// Enregistrement en base de données et affichage d'un message de confirmation
$entityManager->persist($reservation);
$entityManager->flush();
$this->addFlash( type: 'success', message: 'Merci, votre réservation a bien été prise en compte');
return $this->redirectToRoute( route: '/' );
```

J'ai aussi ajouté le mail de l'utilisateur à la réservation si l'utilisateur est connecté, s'il n'est pas connecté l'utilisateur sera enregistré comme « visiteur », pour ce faire j'ai utilisé la fonction privée suivante :

```
// Fonction pour vérifier s'il y a un utilisateur connecté, ou s'il s'agit seulement d'un visiteur.
private function getUserOrGuestIdentifier(Security $security): string
{
    if ($security->getUser() !== null) {
        // si utilisateur connecté, retourne son email
        return $security->getUser()->getUserIdentifier();
    } else {
        // si pas d'utilisateur connecté, retourne 'visiteur'
        return 'visiteur';
    }
}
```

```
// Recupération de l'email du client
$mailUser = $this->getUserOrGuestIdentifier($security);
$reservation->setClientEmail($mailUser);
```

**Autre Fonctionnalités** - J'ai également fait en sorte que l'utilisateur, si il est connecté, retrouve les informations qu'il a entrée par défaut lors de son inscription ( allergies, nombres de convives ). Pour cela j'utilise le service **Security** qui me permet de récupérer l'utilisateur avec la méthode **getUser()**, je récupère alors les données du champ **allergie** avec **getAllergies()** et du champs **nbrDeConvives** avec **getNbrConvive()**, pour finir j'utilise directement les setters **setNbrCouvert()** et **setAllergie()** de l'entité **Réservation** pour enregistrer les données.

```
// Récupère l'information enregistrée par défaut (Nombre de convives/allergies) par l'utilisateur connecté lors de son inscription
if($this->isGranted( attribute: 'IS_AUTHENTICATED_FULLY') || $this->isGranted( attribute: 'ROLE_ADMIN')){
    $user = $security->getUser();
    $allergieUser = $user->getAllergies();
    $nbrCouvertUser = $user->getNbrConvive();
    $reservation->setNbrCouvert($nbrCouvertUser); |
    $reservation->setAllergie($allergieUser);
}
```

Le formulaire est maintenant terminé et fonctionnel.

The screenshot shows a web form titled "Reservez" with a close button (X) in the top right corner. The form contains the following fields:

- Nombre de couvert**: A dropdown menu currently showing the value "0".
- Date**: Three dropdown menus for month, day, and year, showing "Feb", "7", and "2023" respectively.
- Heure**: Two dropdown menus for hour and minute, showing "13" and "00" respectively, separated by a colon.
- Eventuelles allergies**: A text input field.
- Soumettre**: A green button at the bottom left of the form.

# Description de la veille sur les vulnérabilités de sécurité

---

De nos jours il est primordial d'identifier les vulnérabilités de sécurité de son application et de les résoudre. Pour cela j'ai d'abord cherché à identifier les attaques les plus répandues, puis une fois avoir compris leur fonctionnement, j'ai recherché les mesures possibles pour s'en prémunir.

Je me suis concentré sur trois attaques qui sont parmi les plus répandues et dont le site de **owasp.org** en fait la mention dans son top 10 des vulnérabilités de sécurité :

## 1. Les attaques XSS

**Fonctionnement** – se produit lorsqu'un attaquant injecte du code malveillant, tel que du JavaScript, dans une page web affichée à un utilisateur. Lorsqu'un utilisateur visite la page, le code malveillant s'exécute sur son navigateur et peut causer des dommages tels que le vol d'informations confidentielles ou la modification du comportement de la page web.

**Solution** – pour se prémunir de ces attaques on peut valider les entrées utilisateur à l'aide des contraintes de validation définies pour les formulaires.

Par exemple on peut définir une contrainte pour s'assurer que le nom d'utilisateur ne contient pas de caractères spéciaux, que l'adresse e-mail est valide etc.

La contrainte **REGEX** que j'ai présentée plus haut en est une.

## 2. Les attaques CSRF

**Fonctionnement** – C'est une technique qui consiste à forcer un utilisateur à exécuter une action malveillante sur un site Web à son insu. Lorsqu'un utilisateur est connecté à un site web vulnérable et un site web malveillant, ce dernier peut envoyer une requête au site Web vulnérable, qui pensera qu'elle a été envoyée par l'utilisateur.

**Solution** – Des **tokens CSRF** uniques sont générés par **Symfony** pour chaque formulaire, ainsi le site sécurisé peut vérifier que la requête provient bien d'un formulaire soumis par l'utilisateur lui-même.

### 3. Les injections SQL

**Fonctionnement** – Consiste à envoyer des requêtes SQL au serveur pour modifier ou voler des données.

**Solution** – c'est l'ORM **Doctrine** de **Symfony** qui empêche ce genre d'injections. Il se charge lui-même de réaliser des requêtes SQL grâce à son propre système de requête : le **DQL** ( **Doctrine Query Language** )

## Description d'une situation de travail et d'une recherche sur un site anglophone

---

Lorsque j'ai dû faire en sorte que les horaires s'affichent sur tous les pages au niveau du footer, j'ai été confronté à un problème.

En effet j'ai d'abord créé mon Controller qui va me permettre d'aller chercher les horaires fixés en base de données comme ceci :

```
class HorairesController extends AbstractController
{
    // Action pour afficher les horaires
    #[Route('/horaires', name: 'app_horaires')]
    public function afficherHoraires(HorairesRepository $horairesRepository): Response
    {
        // Récupère et affiche les horaires en base de données
        $horaire = $horairesRepository->findAll();
        return $this->render( view: 'partials/footer.html.twig', [
            'horaires' => $horaire,
        ]);
    }
}
```

Le rendu se fait donc sur mon footer, mais pour appeler cette action il faudrait utiliser l'URL **/horaires** pour afficher ces horaires, ce qui n'aurait pas de sens car je souhaite afficher mes horaires sur toutes les pages.

J'ai donc tapé sur Google, la recherche « *Symfony display data in footer on all pages* » et cela m'a fait découvrir le « **Embedded Controller** »

for rendering to the client. When we installed **symfony** the ...

✓ <https://symfony.com> › templating Traduire cette page

## How to Embed Controllers in a Template (Symfony 4.1 Docs)


In other words, the controller of **any page** that **displays** that sidebar must make the same **database** query and pass the list of articles to the included template ...

Vous avez consulté cette page 3 fois. Dernière visite : 14/02/23

### Recherches associées

footer symfony      twig symfony documentation  
symfony renderview in service      symfony javascript twig  
include twig symfony      symfony email template

# How to Embed Controllers in a Template

 Edit this page

**Warning:** You are browsing the documentation for [Symfony 4.1](#), which is no longer maintained.  
Read [the updated version of this page](#) for [Symfony 6.2](#) (the current stable version).

*Comme à chaque fois que mes recherches me mènent à la doc Symfony je veille à bien me retrouver sur la page à jour.*

En lisant la documentation je comprends que le concept de « **Embedded Controller** » qui veut dire Controller « intégré » ou « embarqué » va me permettre de récupérer mon Controller **afficherHoraires()** et de l'appeler sur chaque page au niveau du footer.

Pour se faire je vais sur la Template de base et l'intègre là où je souhaite mettre le footer :

```
<body>

    {{ include('partials/header.html.twig') }}

    {% block body %}
    {% endblock %}

#      integration d'un controller pour afficher footer dynamique sur toute les pages (via outils 'embedded controller')#
    {{ render(path('app_horaires')) }}
```

Un peu comme pour le header ci-dessus que j'appelle sur toute les pages grâce à la fonction **{{ include() }}**, la fonction **{{ render() }}** va me permettre d'appeler non pas une page Twig, mais un Controller.

# Extrait du site anglophone

---



## Embedding Controller

*Including template fragments is useful to reuse the same content on several pages. However, this technique is not the best solution in some cases.*

*Imagine that the template fragment displays the three most recent blog articles. To do that, it needs to make a database query to get those articles. When using the **include()** function, you'd need to do the same database query in every page that includes the fragment. This is not very convenient.*

*A better alternative is to **embed the result of executing some controller** with the **render()** and **controller()** Twig functions.*

*First, create the controller that renders a certain number of recent articles:*

```
// src/Controller/BlogController.php
namespace App\Controller;

use Symfony\Component\HttpFoundation\Response;
// ...

class BlogController extends AbstractController
{
    public function recentArticles(int $max = 3): Response
    {
        // get the recent articles somehow (e.g. making a database query)
        $articles = ['...', '...', '...'];

        return $this->render('blog/_recent_articles.html.twig', [
            'articles' => $articles
        ]);
    }
}
```

*Then, create the `blog/_recent_articles.html.twig` template fragment (the `_` prefix in the template name is optional, but it's a convention used to better differentiate between full templates and template fragments):*

```
{# templates/blog/_recent_articles.html.twig #}
{% for article in articles %}
    <a href="{{ path('blog_show', {slug: article.slug}) }}">
        {{ article.title }}
    </a>
{% endfor %}
```

Now you can call to this controller from any template to embed its result:

```
{# templates/base.html.twig #}

{# ... #}
<div id="sidebar">
    {# if the controller is associated with a route, use the path() or url() functions
    {{ render(path('latest_articles', {max: 3})) }}
    {{ render(url('latest_articles', {max: 3})) }}

    {# if you don't want to expose the controller with a public URL,
    use the controller() function to define the controller to execute #}
    {{ render(controller(
        'App\\Controller\\BlogController::recentArticles', {max: 3}
    )) }}
</div>
```



# Traduction

## Contrôleurs intégrés

L'inclusion de fragments de modèles est une méthode pratique pour réutiliser le **même contenu sur plusieurs pages**. Cependant, cette technique n'est pas toujours la meilleure solution.

Prenons l'exemple d'un fragment de modèle qui affiche les trois articles de blog les plus récents. Pour y parvenir, il doit effectuer une requête dans la base de données pour récupérer ces articles. Si nous utilisons la fonction ***include()***, cette même requête devrait être effectuée sur chaque page qui inclut le fragment. Cela peut vite devenir fastidieux.

Une alternative plus efficace consiste à intégrer le résultat d'un contrôleur en utilisant les fonctions Twig ***render()*** et ***controller()***.

Tout d'abord, nous créons un contrôleur qui affiche un certain nombre d'articles récents :

Ensuite, nous créons un fragment de modèle ***blog/\_recent\_articles.html.twig*** (le préfixe ***\_*** dans le nom du modèle est facultatif, mais il est souvent utilisé pour mieux différencier les modèles complets des fragments de modèle) :

```
{# templates/blog/_recent_articles.html.twig #}
{% for article in articles %}
    <a href="{ path('blog_show', {slug: article.slug}) }}">
        {{ article.title }}
    </a>
{% endfor %}
```

*A présent vous pouvez appeler ce Contrôleur à partir de n'importe quel template.*

```
{# templates/base.html.twig #}

{# ... #}
<div id="sidebar">
    {# if the controller is associated with a route, use the path() or url() functions
    {{ render(path('latest_articles', {max: 3})) }}
    {{ render(url('latest_articles', {max: 3})) }}

    {# if you don't want to expose the controller with a public URL,
    use the controller() function to define the controller to execute #}
    {{ render(controller(
        'App\\Controller\\BlogController::recentArticles', {max: 3}
    )) }}
</div>
```

## Conclusion

---

Comme pour les ECF d'entraînement j'ai beaucoup apprécié ce projet, il englobe tous les aspects du développement d'une application, bien que je n'aie pas connu de stage en entreprise, ce genre de projet me rend confiant quant à une future expérience en entreprise, même si je débiterai bien sûr sans être « expert » dans aucune des compétences présente dans cet ECF, je sais que j'aurai les bonnes bases pour apprendre, progresser.

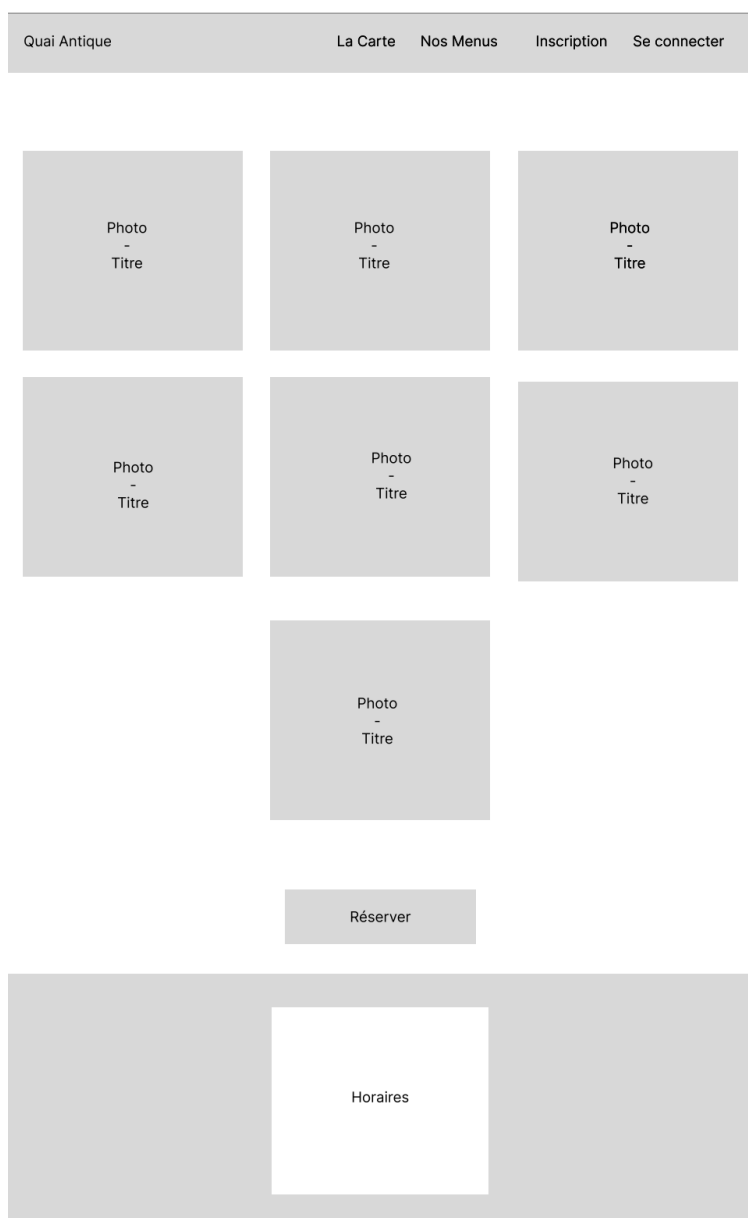
Toutes les difficultés que j'ai rencontrées m'ont pris beaucoup de temps à les résoudre, mais j'ai compris que ce n'était pas du temps perdu, bien au contraire j'ai inconsciemment développé des réflexes, une persévérance, et une manière de penser en toute autonomie, et c'est je pense, ce qu'on attend d'un développeur.



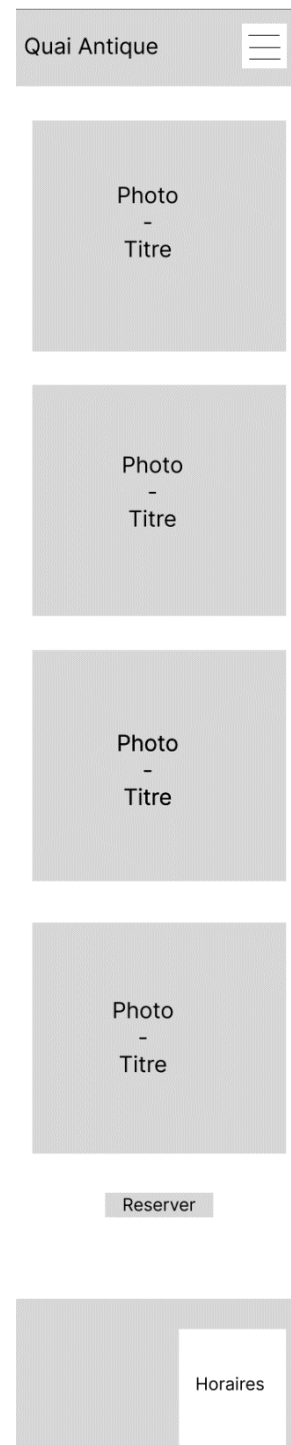
# Annexe

---

## Exemple wireframes desktop



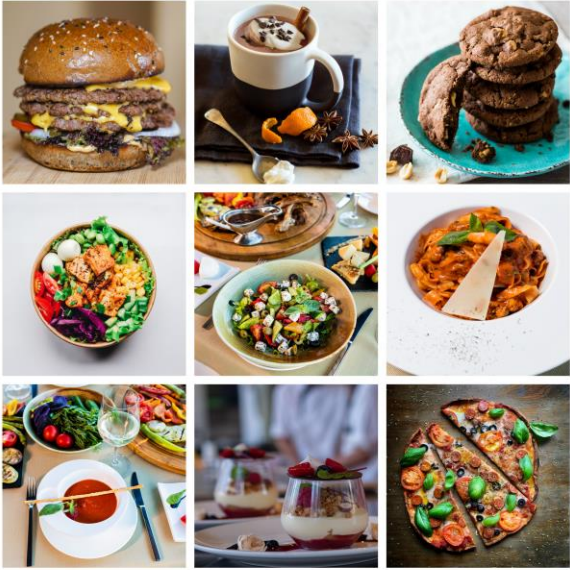
## Exemple wireframes mobile



Le Quai Antique

La CarteMenuS'inscrireSe connecter

Le Chef Michant  
vous souhaite la bienvenue



Reservez

LE QUAI ANTIQUE

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Quibus enim, harum quod repellat ullam vitae. A ad animi aspernatur at delectus dolores ea eros esse explicabo facilis ipsum (ipsum iusto laborum maiores natus repellat reprehenderit, rerum saepe sed ut veniam)

HORAIRES

Lundi : 14:00-15:00 | 19:00-22:00  
 Mardi : 12:00-15:00 | 19:00-22:00  
 Mercredi : 12:00-15:00 | 19:00-22:00  
 Jeudi : 12:00-15:00 | 19:00-22:00  
 Vendredi : 12:00-15:00 | 19:00-21:00  
 Samedi : 12:00-15:00 | 19:00-22:00  
 Dimanche : Ferme

ADRESSE

Le Quai Michant  
 83 Rue Mouffetard,  
 75005 Paris

© 2022 Copyright : Hafid HADJAGUI - Otacon

## La carte

## Desserts



# Page de connexion

Se connecter

Email

Mot de passe

☐ Se souvenir de moi

Soumettre

## Panel d'administration (EasyAdmin)

Mon tableau de bord

Dashboard

Carte

Horaires

Menus

















Reservation

PlacesMax

Search

Carte

Add Carte

<input type="checkbox"/> Titre	Description	Prix	Categorie	Favoris	Image Name	
<input type="checkbox"/> Burger	Burger, 3 steak 100gr, tomate, salade, cheedar	10	plat	<input checked="" type="checkbox"/>		...
<input type="checkbox"/> Mini croq'pané	fromage, panure, herbes	6	entrée	<input type="checkbox"/>		...
<input type="checkbox"/> Café gourmand	Café, crème de vanille, cannelle, zest d'orange	5	dessert	<input checked="" type="checkbox"/>		...
<input type="checkbox"/> Cookies tout choco	Cookies, chocolat, noix	7	dessert	<input checked="" type="checkbox"/>		...
<input type="checkbox"/> Pancake	Pancake, framboise, sirop d'érable	8	dessert	<input type="checkbox"/>		...
<input type="checkbox"/> Salade	Salade asiatique	7	entrée	<input checked="" type="checkbox"/>		...
<input type="checkbox"/> Framboisier	Framboisier, genoise tendre, crème de vanille	5	dessert	<input type="checkbox"/>		...
<input type="checkbox"/> Entrecôte	Entrecôte, sauce crème, champignons	15	plat	<input type="checkbox"/>		...
<input type="checkbox"/> Entrée du chef	une composition spécialement conçu par le chef	9	entrée	<input type="checkbox"/>		...
<input type="checkbox"/> Velouté	Carottes, pommes de terre, herbes de provences	11	plat	<input type="checkbox"/>		...
<input type="checkbox"/> Salade Grec	Feta, cocombre, tomate, olive	8	entrée	<input checked="" type="checkbox"/>		...
<input type="checkbox"/> Plateau de légumes	légumes grillées, aubergine, tomate, piment doux, poivrons, her...	8	plat	<input type="checkbox"/>		...
<input type="checkbox"/> Pâtes du chef	Pâte maison, tomate, fromage, basilic, olives	13	plat	<input checked="" type="checkbox"/>		...
<input type="checkbox"/> Crevette marinées	Crevette, herbes, citron	8	plat	<input type="checkbox"/>		...
<input type="checkbox"/> Taboulet Quinoa	Quinoa, 4 légumes, basilic, citron	9	entrée	<input type="checkbox"/>		...
<input type="checkbox"/> Purée de tomate	tomate, basilic, huile d'olive	8	entrée	<input checked="" type="checkbox"/>		...