DOSSIER PROJET

Développeur Web et Web Mobile

Session Juin/Juillet 2026

Remerciements :

Ce dossier n'aurait pas été réalisable sans les équipes de la plate-forme Studi, mais également sans mon professeur de soutien Loic Bonin grâce à qui j'ai pu acquérir énormément de bonnes pratiques et compétences. Je les remercie donc pour leurs réponses toujours pertinentes, ainsi que leur accompagnement.

Je remercie également le personnel de l'espace coworking de la MCC de Bourg-en-Bresse qui a été mon espace de travail privilégié.

Enfin je remercie mon épouse Mélissa, ainsi que mes deux filles pour leur soutien dans ce projet de reconversion professionnelle.

Sommaire

Introduction

1 - Compétences du référentiel couvertes par le projet :

Activité type 1 :« Développer la partie front-end d'une application web ou web mobile sécurisée »

- Maquetter des interfaces utilisateur web ou web mobile
- Réaliser des interfaces utilisateur statiques web ou web mobile
- Développer la partie dynamique des interfaces utilisateur web ou web mobile

Le développement de la partie front-end d'une application concerne la partie utilisateur, la partie « visible ». Nous devons la maquetter en respectant différentes étapes. Les langages utilisés pour les interfaces statiques sont le HTML et le CSS, et pour les interfaces dynamiques le JavaScript. Nous devrons respecter le cahier des charges, notamment le responsive, pour apporter une expérience utilisateur optimale et sécurisée.

Activité type 2 : « Développer la partie back-end d'une application web ou web mobile sécurisée »

- Mettre en place une base de données relationnelle
- Développer des composants d'accès aux données SQL et NoSQL
- Développer des composants métier côté serveur

Le développement de la partie back-end d'une application concerne la partie d'administration. Nous réaliserons une interface administrateur, tout en gérant la base de données relationnelle et non relationnelle. Des mécaniques de fonctionnement de l'application seront mises en place grâce au langage PHP à travers le framework Symfony. Nous devrons respecter les bonnes pratiques, notamment en ce qui concerne la sécurité, et mettre en ligne notre application.

2 - Résumé du projet :

Le projet présenté dans ce dossier se nomme « Vite & Gourmand ». C'est un projet factice qui a été réalisé dans le cadre de la formation suivie chez Studi. Il a d'abord été réalisé de façon partielle dans le cadre du rendu ECF (Évaluation en Cours de Formation), avant d'être finalisé pour répondre aux besoins de ce projet et démontrer les compétences requises pour le référentiel.

Je suis donc embauché et missionné par l'entreprise « FastDev » pour réaliser un projet d'application. Ce projet doit répondre aux besoins de l'entreprise « Vite & Gourmand » constituée de deux personnes, Julie et José. Cette entreprise existe depuis 25 ans à Bordeaux, et propose leurs prestations pour tout événement (simple repas comme Noël ou encore Pâques) au travers d'un menu en constante évolution. Les menus sont envoyés par mail aux habitués, mais Julie a eu l'idée d'obtenir une application web permettant d'augmenter leurs visibilités ainsi que proposer leurs menus bien plus facilement et au plus grand nombre.

Il est important de pouvoir se mettre en situation réelle et se poser les bonnes questions. Comment vais-je répondre aux besoins de mes clients, quelles solutions, par mes compétences, puis-je leur apporter.

Nous développerons tout d'abord des maquettes de la solution digitale, des schémas de conceptualisation, qui apporteront une première ébauche à notre client, avant de passer à la réalisation technique, qui devra répondre aux différents critères de bonnes pratique, de sécurité et de maintenabilité des standards actuels. Enfin nous mettrons en ligne notre application. Dans la pratique, toutes ces étapes sont indispensables et doivent être réalisées en étroite collaboration avec le client.

Il est important d'apporter une réponse technique, mais aussi de pouvoir justifier la pertinence de tous les éléments apportés au projet.

3 - Environnement de travail :

Outils de développement

Ce projet a été réalisé en totale autonomie. Je travaille sur PC avec un environnement Windows.

Pour le maquettage j'ai utilisé Figma, qui est une interface extrêmement répandue aujourd'hui pour le maquettage de solution digitale.

Les différents schémas de conceptualisation de base de données ont été eux réalisé avec l'interface draw.io qui permet de générer des schémas relationnels très facilement, et ainsi conceptualiser notre application.

J'ai choisi comme IDE PHP Storm de Jetbrain, pour son ergonomie et sa maturité. De plus, cet IDE a son propre serveur local et on peut y gérer directement la base de données ce qui facilite grandement le développement.

Ressources

Ce projet est basé sur mon examen en cours de formation (ECF), je me suis donc référé à la documentation de ce dernier pour définir mes besoins client, le cahier des charges, ainsi que les exigences techniques.

Mes ressources techniques principales ont été la documentation (w3schools, mozilla, symfony doc...) ainsi que, bien sûr, les cours Studi et particulièrement les lives.

Au niveau des ressources humaines, j'ai bien sûr été appuyé par l'équipe enseignante de Studi, mais également par Mr Loic Bonin qui m'a dispensé des cours de soutien.

Le Projet

1 - Cahier des charges :

Le projet devra comporter une page d'accueil qui aura pour mission de :

- Présenter l'entreprise
- Mettre en avant le professionnalisme de l'entreprise
- Des avis clients validés

Un menu d'application sera également mis en place, il permettra :

- Un retour vers la page d'accueil
- Un accès à tous les menus
- Un accès à la page contact
- Une connexion sécurisée

Un pied de page avec les horaires, les mentions légales et les CGV.

Il y aura également une vue globale des menus, configurable depuis une interface dans le back-office. Tous les éléments du menu pourront être personnalisés.

Cette vue globale doit comporter le titre de chaque menu et sa description. Des filtres doivent être disponibles afin de permettre une recherche.

Une création de compte devra être possible pour tout utilisateur en rentrant les données nécessaires, ainsi qu'une connexion sécurisée via un formulaire.

La commande d'un menu devra également être prise en charge via l'application, en demandant des informations sur la prestation. Il sera possible de visualiser la commande, son état, ainsi que son prix. L'utilisateur devra recevoir un mail de confirmation une fois celle-ci passée.

Un espace utilisateur sera présent, avec un système de visualisation des commandes passées, ainsi que leur état.

Un espace employé/administrateur devra être mis en place pour gérer les commandes, menus, commandes, plats et clients, avec une sécurisation

adéquate.

Enfin, une page contact avec un formulaire d'envoi de mail sera également nécessaire.

2 - Spécifications techniques :

Pour le front-end de notre application, nous avons utilisé les langages **HTML 5** pour la structure de page, ainsi que le **CSS** pour la mise en page statique. Pour ce qui est de l'interface utilisateur dynamique, nous avons utilisé le langage **JavaScript**. Ces choix se justifient car ce sont des technologies très répandues dans le monde du Web aujourd'hui, ce qui fais qu'elles sont largement documentées et sécurisées. La maintenabilité n'en est d'ailleurs que renforcée.

Pour la partie back-end, nous avons utilisé le langage **PHP** à travers le framework **Symfony**, qui nous apporte une structure via le modèle MVC (Modèle, Vue, Contrôleur) qui est également largement utilisée, et donc documentée et sécurisée.

Le framework Symfony nous apporte également énormément de bundles tels que « easyadmin » qui nous permettent d'optimiser notre travail de développement. Ces choix seront justifiés et expliqués dans ce projet.

Notre base de données relationnelle sera elle gérée en **MySQL**, et notre base de données non relationnelle par **MongoDB**.

Pour le déploiement, comme mentionné dans ce dossier, j'étais d'abord parti sur la solution Railway, mais, faisant face à de nombreuses contraintes techniques, je me suis finalement tourné vers la solution **Render**.

Réalisation du projet

1 - Conception de la base de données et schéma MCD (modèle conceptuel de données) :

Lorsque l'on développe un projet Web, il faut pouvoir le conceptualiser. Prendre en compte tous les besoins pour pouvoir apporter une réponse claire, durable, et facile à maintenir à son client.

Le développement ne commence pas à la première ligne de code, il se réfléchit en amont.

La conception de la base de données, et par extension la réalisation du schéma MCD (modèle conceptuel de données) vont être une étape fondamentale dans le processus de création. Elle permettra de faciliter grandement la mise en place du projet. Ce sont un peu les fondations de la future application Web.

Il faudra donc déterminer quels sont les éléments de la base de données, leur nature, leur type, et la relation qu'ils auront entre eux. Cela nous permettra de créer les entités dans notre programme, simplement en suivant ce schéma. Il est important de prendre en compte tous les paramètres possibles, et tous les scénarios, ce qui facilitera la mise à jour de la base de données, et la maintenabilité de l'application.

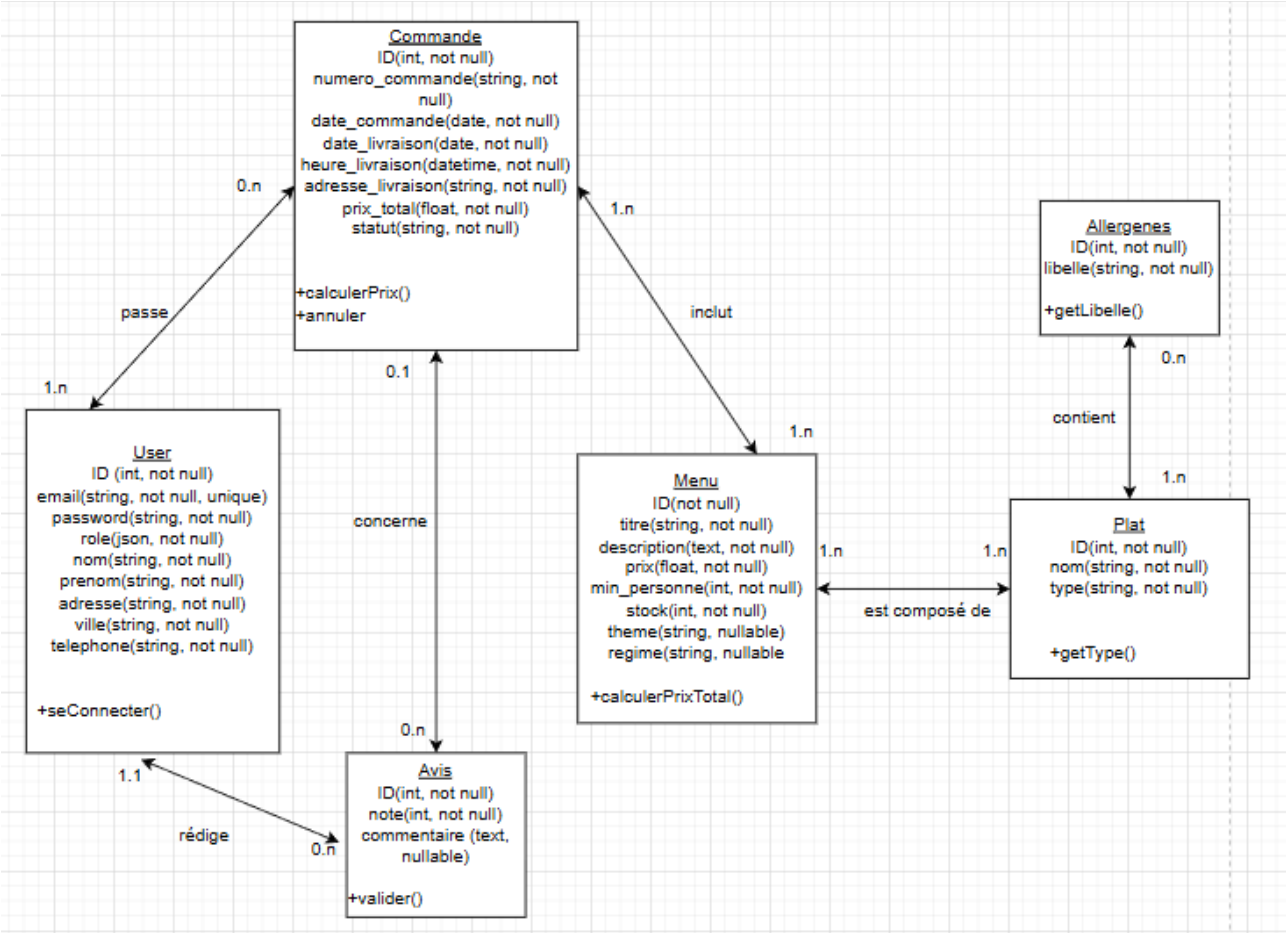
Chaque donnée sera représenté dans une table, avec ses différents champs (fields), ainsi que les méthodes associées. Chaque champ aura un type, et pourra être nul, ou non. C'est exactement ce qui sera déterminé avec le diagramme de classe.

Nous déterminons ensuite les relations que ces données ont entre elles. Il en existe principalement quatre :

- OneToMany
- OneToOne
- ManyToMany
- ManyToOne

Elles permettent de structurer notre base de données, de la rendre « relationnelle ». C'est un élément clé du développement orienté objet.

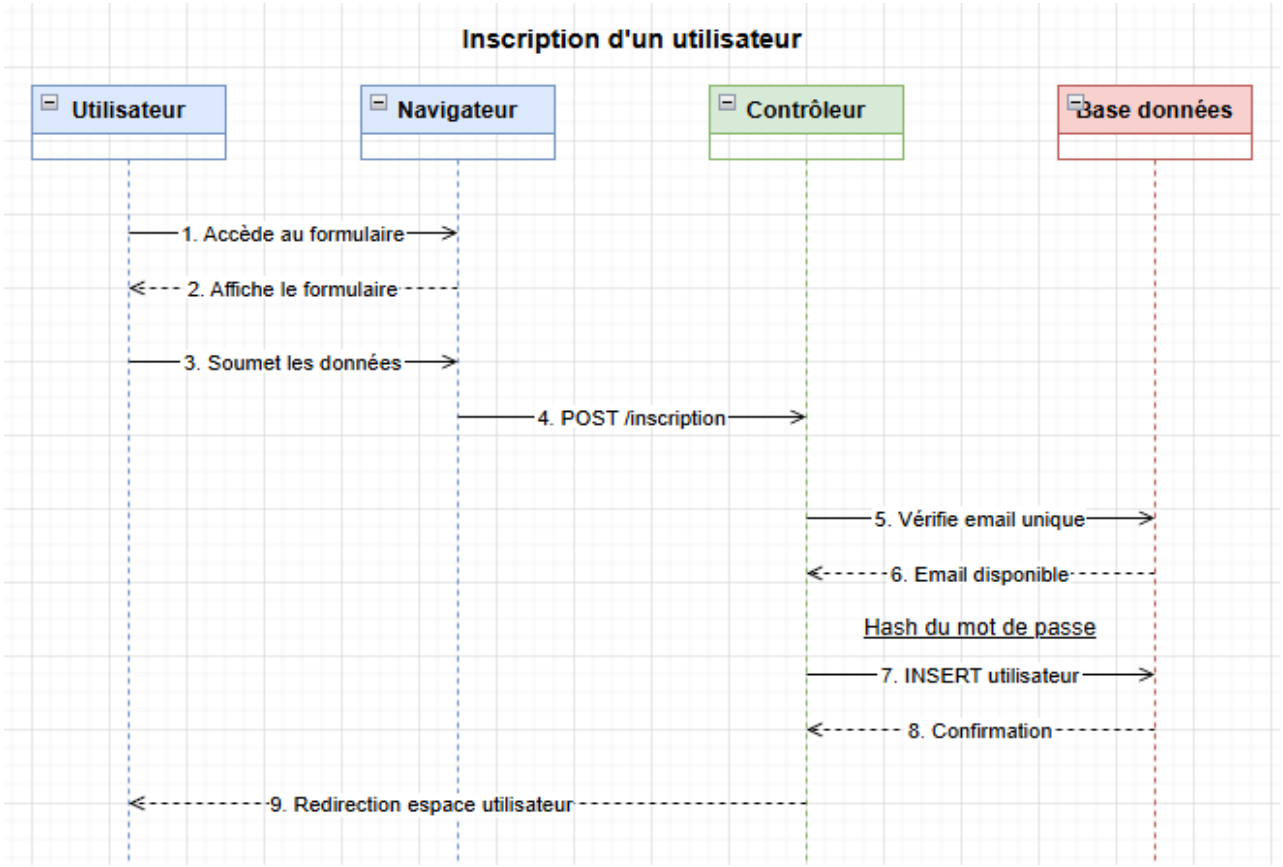
Dans notre cas, nous avons analysé le cahier des charges, et il apparaît que nous avons donc besoin de plusieurs entités distinctes, avec chacune leurs champs, et des relations bien définies. Ceci est donc représenté dans le schéma suivant, que nous appelons **diagramme de classes** :



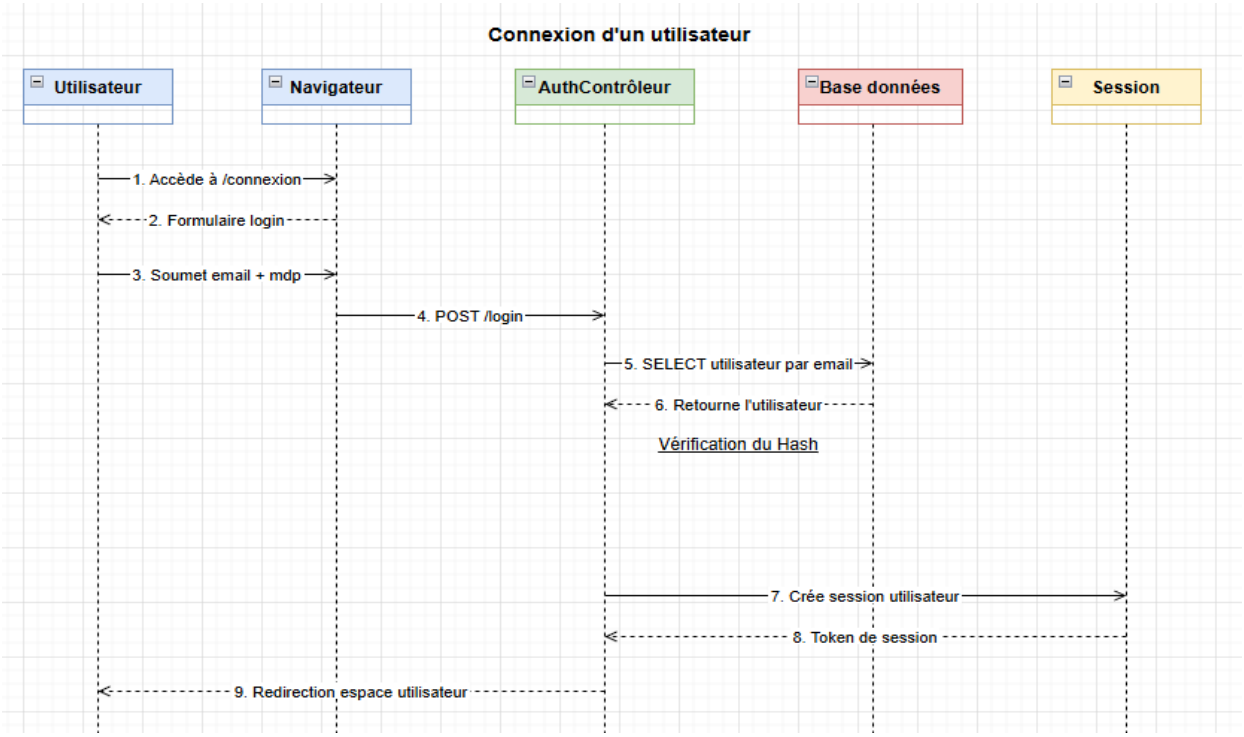
En analysant ces schémas, nous observons que le cahier des charges du client est respecté. Nous avons une idée claire de la base de données relationnelle de notre application, et par extension, de la structure et la logique que nous aurons à développer pour notre application.

Il convient ensuite d'établir des **diagrammes de séquence**, qui auront pour rôle de décrire les différents scénarios, en modélisant les interactions entre les utilisateurs et les composants du système dans un ordre chronologique. Nous avons ici trois diagrammes établis, un pour l'inscription, un pour la connexion et un pour la commande.

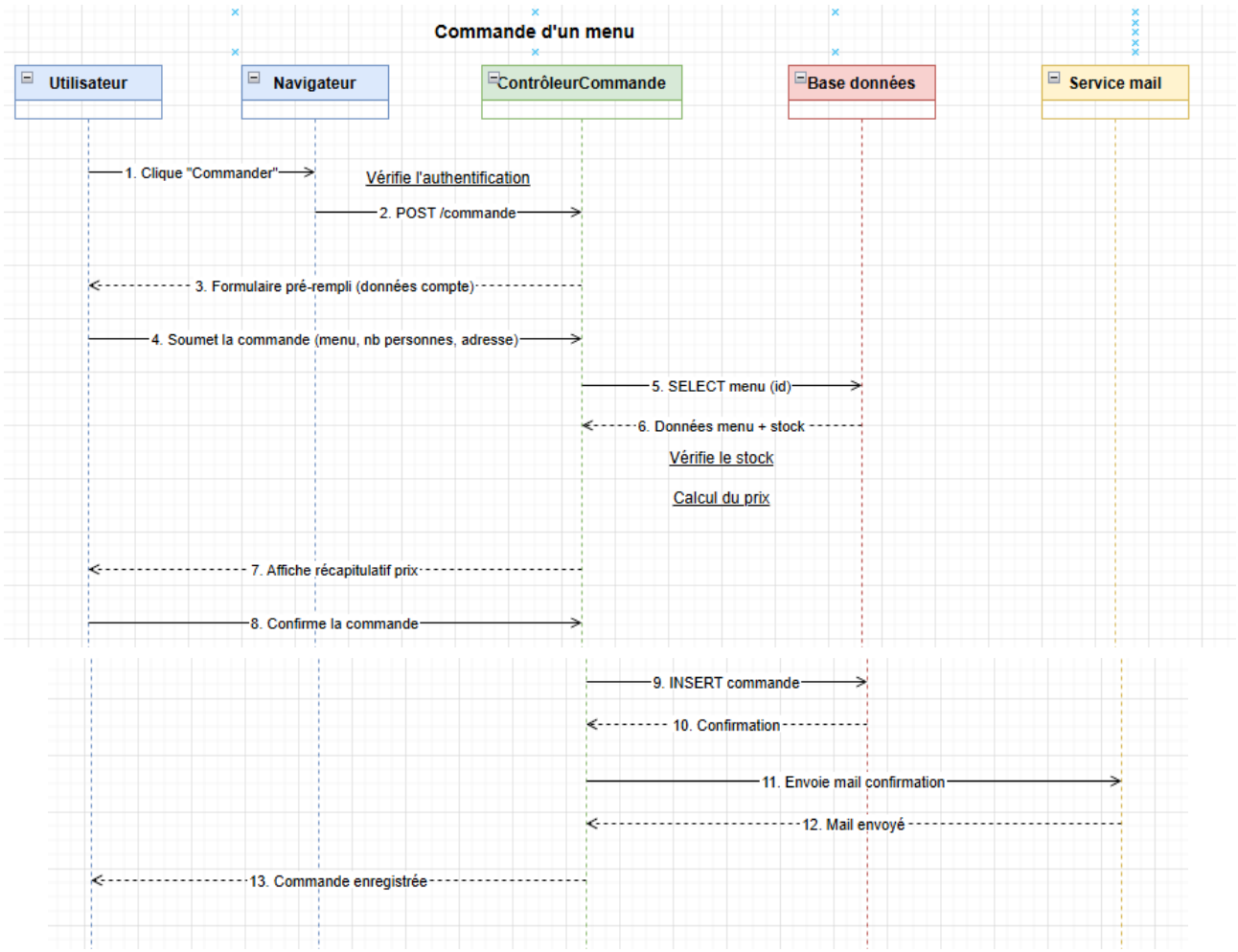
Voici d'abord le diagramme d'inscription, il décrit les étapes quand un utilisateur crée son compte :



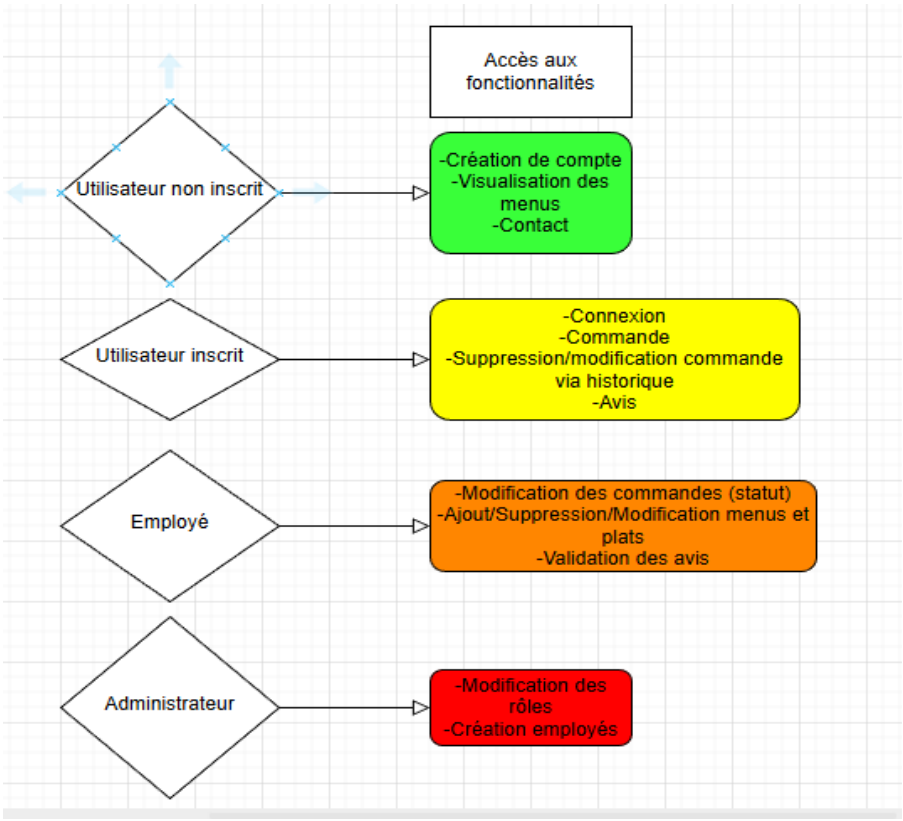
Voici maintenant le diagramme de connexion :



Et enfin de diagramme de commande, c'est le plus complet :



Enfin, il existe un dernier schéma, le diagramme de cas, qui répond à la question « qui fait quoi ? ». Il ne dit pas comment ça fonctionne, juste ce que le système doit permettre. Voici notre diagramme de cas :



2 - Maquettage :

Le maquettage est un étape clé du développement d'une application. Il permet d'avoir une vision claire et précise du résultat final du projet, à la fois pour les développeurs et pour le client.

En général, une charte graphique est, soit définie avec le client, soit fournie par le client. Pour notre projet, j'ai établi une charte graphique, et j'ai créé un logo via IA (cf annexe).

Comme indiqué dans les spécifications technique de notre projet, nous avons utilisé Figma en tant qu'outil de conception de l'interface utilisateur.

Nous allons d'abord réaliser des Wireframes, c'est un peu le « squelette » de chaque page. Elles définissent où est quoi, sans aucun style ni couleur. Cela permet de structurer l'application. Vous retrouverez les Wireframes de la page

d'accueil, de la page menus, et de la page commande en annexe. Notons qu'il est important de faire une version « desktop » et une version « mobile » au minimum. Les « wireframes » doivent être réalisées pour toutes les vues de l'application, et présentées au client avant de passer à l'étape suivante.

Ensuite, il nous faut réaliser les « mockups ». Ce sont des maquettes qui vont se baser sur les « wireframes », mais elles devront être fidèles au résultat final. Il faudra y inclure la charte graphique et du contenu, pour rendre compte du rendu visuel désiré. Vous trouverez en annexe les « mockups » pour cette même page d'accueil, pour la page menus, et également pour la page commande.

Cette étape de maquettage est primordiale pour identifier les besoins et envies du client. Elles permettent de dresser un cap visuel et fonctionnel avant le développement de l'application.

3 – Création du projet :

Initialisation

Maintenant que nous avons structuré et défini notre projet, nous allons pouvoir passer à la réalisation pure. Comme indiqué précédemment, nous utiliserons le langage PHP à travers le framework Symfony pour développer notre application. Nous détaillerons la création du projet grâce à des captures d'écran de code extrait de fichier de code, ou des lignes de commande.

Il est à noter que le téléchargement de PHP, ainsi que Symfony CLI, doit être préalable à l'initialisation du projet.

Nous pouvons télécharger PHP directement sur le site www.php.net, ainsi que Symfony CLI sur www.symfony.com. Il convient de choisir des versions adaptées. PHP a des versions qui ont un support pendant 2 ans, nous avons donc opté pour la version 8.2. Pour ce qui est de Symfony, il vaut mieux choisir une version « LTS » (Long Term Support) qui a un support de 3 ans, c'est pourquoi nous avons choisi la version 7.4. Nous choisissons la version « CLI », car elle apporte énormément de fonctionnalités supplémentaires et très utiles, comme un serveur local par exemple.

Il convient maintenant d'installer « composer ». Mais qu'est ce que « composer », et à quoi cela sert ? « Composer » est un outil de gestion de dépendances en PHP . Il permet de déclarer les bibliothèques dépendantes de

votre projet et se charge de les installer. Car oui, Symfony est un framework qui utilise l'injection de dépendances pour fonctionner. Composer pourra s'installer en téléchargeant le fichier d'installation sur www.getcomposer.org.

Tout d'abord, nous allons initier notre projet grâce à la commande permise par Symfony CLI suivante :

Notre projet est maintenant créé.

Il faut maintenant installer les différents bundles dont nous aurons besoin. En effet, Symfony est un framework PHP qui nous donne énormément d'outils pour nous aider dans le développement de notre application. Voici les principaux bundles installés à l'initiation du projet :

- TwigBundle (permet de gérer les « vues » du projet, nous reviendrons sur la structure « MVC » (modèle, vues, contrôleurs) du projet dans la partie suivante).

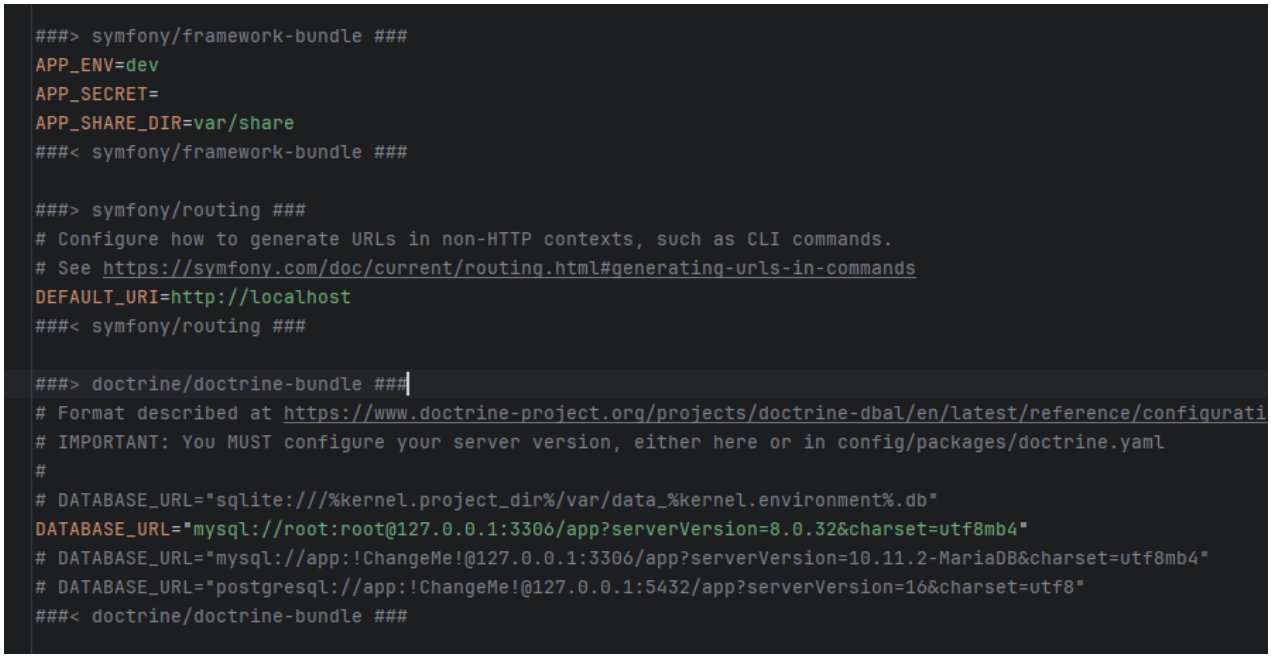
- MakerBundle (permet de créer facilement beaucoup d'éléments nécessaire à notre application, simplement en ligne de commande)

- EasyAdminBundle (ce bundle permet de gérer facilement le back-office, il est particulièrement préconisé dans le type de projet que nous développerons. Nous reviendrons beaucoup plus longuement sur ce bundle plus tard)

- SecurityBundle (permet de gérer la partie « sécurité » de notre application, avec notamment un système de « hash » de mot de passe, un système de gestion de droits utilisateurs, etc...)

Nous devons ensuite configurer notre environnement de travail. Il suffit pour cela de créer un environnement local que nous appellerons « .env.local » et y paramétrer :

- l'environnement en « dev »
- l'URL de la base de données



```
###> symfony/framework-bundle ###
APP_ENV=dev
APP_SECRET=
APP_SHARE_DIR=var/share
###< symfony/framework-bundle ###

###> symfony/routing ###
# Configure how to generate URLs in non-HTTP contexts, such as CLI commands.
# See https://symfony.com/doc/current/routing.html#generating-urls-in-commands
DEFAULT_URI=http://localhost
###< symfony/routing ###

###> doctrine/doctrine-bundle ###
# Format described at https://www.doctrine-project.org/projects/doctrine-dbal/en/latest/reference/configurati
# IMPORTANT: You MUST configure your server version, either here or in config/packages/doctrine.yaml
#
# DATABASE_URL="sqlite:///%kernel.project_dir%/var/data_%kernel.environment%.db"
DATABASE_URL="mysql://root:root@127.0.0.1:3306/app?serverVersion=8.0.32&charset=utf8mb4"
# DATABASE_URL="mysql://app:!ChangeMe!@127.0.0.1:3306/app?serverVersion=10.11.2-MariaDB&charset=utf8mb4"
# DATABASE_URL="postgresql://app:!ChangeMe!@127.0.0.1:5432/app?serverVersion=16&charset=utf8"
###< doctrine/doctrine-bundle ###
```

Ici, l'URL de la base de données est paramétré sur MySQL, avec comme identifiants et mots de passe « root ».

MySQL est un système de gestion de base de données relationnelles. C'est grâce à lui que nous pouvons gérer notre base de données. Cette dernière peut être administrée depuis une interface. Ici, nous utiliserons l'interface de l'IDE PHP Storm.

Enfin, il est nécessaire de versionner l'application à l'aide d'un système de gestion de versions tel que **Git**. Celui-ci permet d'enregistrer méthodiquement les différentes évolutions du code au cours du développement, tout en conservant un historique des modifications. L'ensemble du projet est stocké dans un espace appelé **repository** (ou dépôt).

Dans notre cas, le dépôt distant est hébergé sur **GitHub**, une plate-forme permettant de stocker et partager des dépôts Git.

La méthodologie utilisée est la suivante :

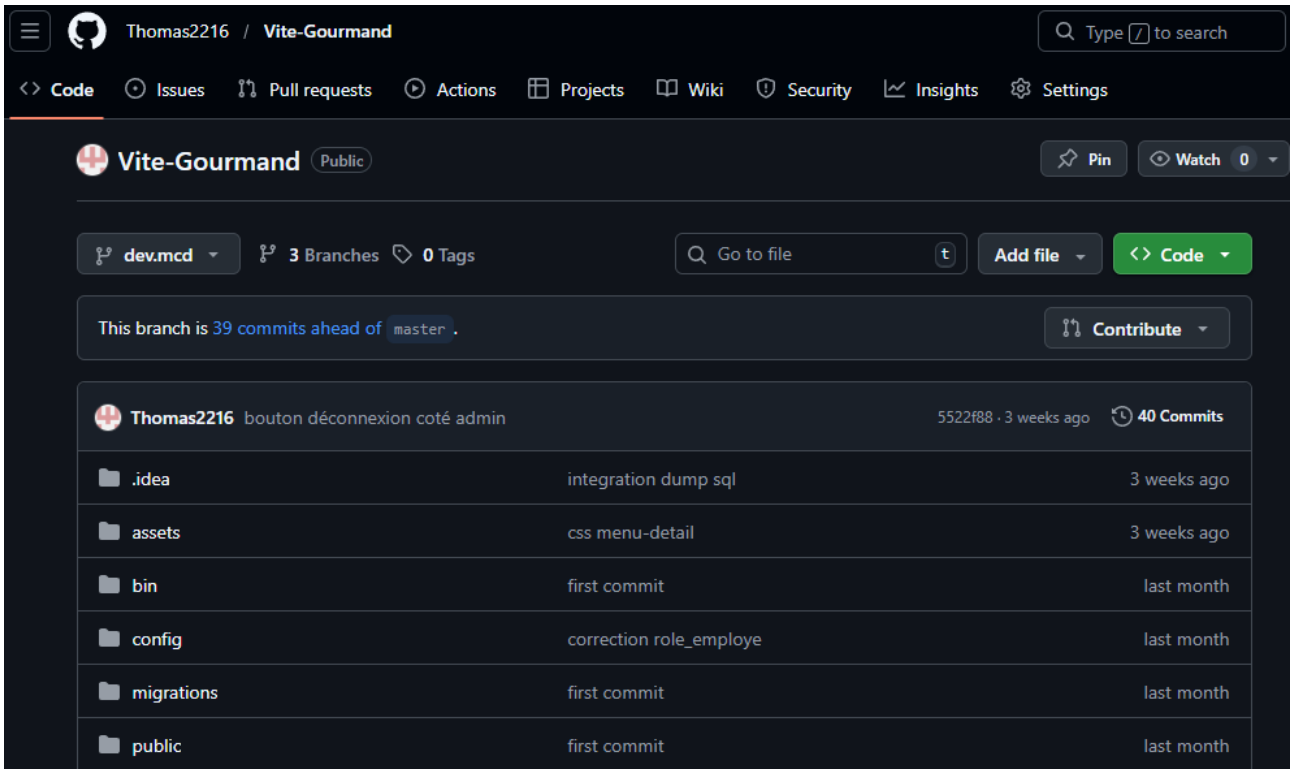
1. Créer un **repository vide** via l'interface de GitHub.
2. **Cloner** ce repository sur la machine locale dans le dossier du projet à l'aide de la ligne de commande. Cela crée une copie locale du dépôt distant.
3. Ajouter les fichiers du projet au suivi de Git avec la commande git add.
4. Créer un **commit**, c'est-à-dire un enregistrement des modifications du code source, avec la commande git commit.
5. Envoyer ces modifications vers le dépôt distant GitHub à l'aide de la commande git push.

Le projet est alors **versionné** : à chaque modification du code, il est possible d'enregistrer une nouvelle version du projet tout en conservant l'historique complet des changements. Cela permet notamment de suivre l'évolution du développement, de revenir à une version précédente si nécessaire et de faciliter le travail collaboratif.

Il convient d'avoir quelques bonnes pratiques lorsqu'on utilise un système de versions :

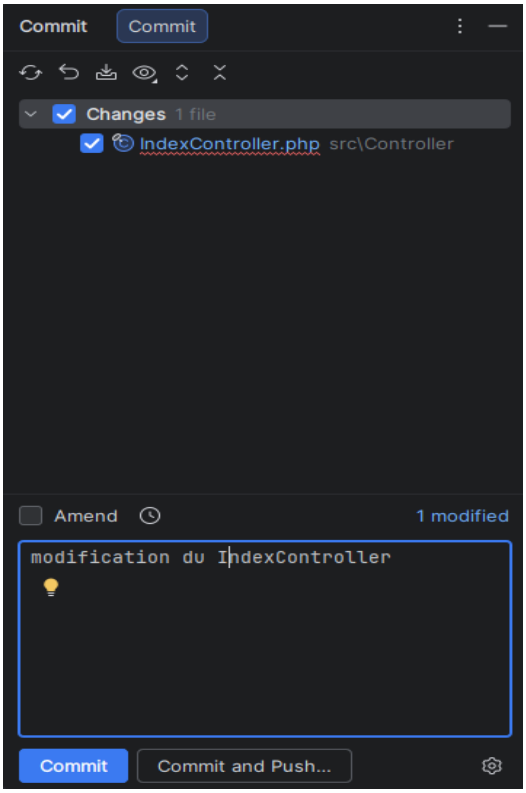
- Il faut travailler sur plusieurs « branches », chacune concernant un domaine bien distinct du développement.
- Il faut nommer tous ses commits, pour pouvoir identifier facilement toute modification de code, et faire des « commits » réguliers.
- **Ne pas versionner** certains dossiers qui peuvent contenir certaines informations qu'il convient de ne pas rendre publiques (comme les fichiers .env, .env.local par exemple). Pour cela, un fichier .gitignore est utilisé afin d'exclure ces éléments du versionnage.

Voici une vue globale de l'interface GitHub, on y voit les différents outils, les branches existantes. Certains dossiers du projet avec les noms et dates des différents « commits » (versions).



Il est à noter que cet outil permet de partager et dupliquer son travail très facilement et de faciliter le travail en équipe en séparant les tâches. L'équipe « Front » pourra par exemple travailler sur la branche « Front », sans interférer avec les autres équipes de développement.

Voici également une vue de l'interface de l'IDE PHP Storm qui permet de gérer le Git. En quelques clics, il est très simple de mettre à jour son dépôt local et/ou distant, ce qui est un gain de temps considérable.



Notre projet est maintenant initié. Nous allons pouvoir commencer le développement à proprement parler.

La structure de notre application respectera le modèle « MVC », très répandu, qui permet d'organiser le code en séparant les responsabilités afin de rendre l'application plus claire, maintenable et évolutive.

Les principes du modèle « MVC » :

- Modèle (gère la logique métier et les données)
- Vue (gère l'interface utilisateurs)
- Contrôleur (fait le lien entre le modèle et la vue)

Pour le développement du code d'une application, il est indispensable de lancer le serveur local, pour pouvoir afficher les vues et faire fonctionner les mécaniques mises en place. Grâce à Symfony CLI, ce serveur est directement intégré au projet.

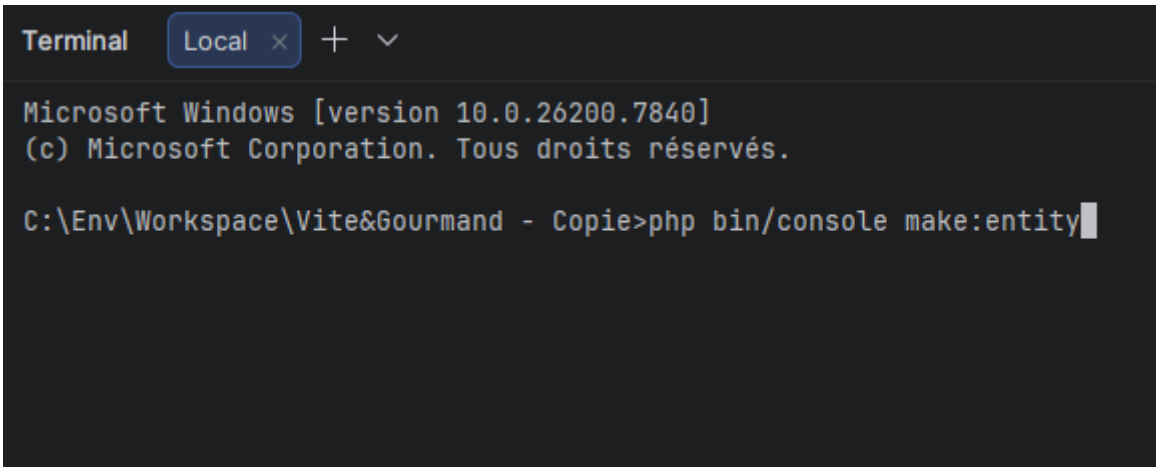
Nous avons maintenant un serveur local fonctionnel.

Développement base de données

Une des premières choses à mettre en place lorsque l'on crée un application est la base de données.

Chaque donnée va être représentée sous forme de tableau, qui lui même sera représenté dans un fichier PHP « entity » dans notre application. Voici comment procéder.

Grâce à notre « MakerBundle », il est très facile de créer notre entité avec la commande suivante :



Une fois notre entité créée, il va falloir la nommer, puis déterminer ses champs, avec leur type, ainsi que si ils sont « nullable ».

```
Class name of the entity to create or update (e.g. BravePuppy):
> Utilisateur

Add the ability to broadcast entity updates using Symfony UX Turbo? (yes/no) [no]:
>

created: src/Entity/Utilisateur.php
created: src/Repository/UtilisateurRepository.php

Entity generated! Now let's add some fields!
You can always add more fields later manually or by re-running this command.

New property name (press <return> to stop adding fields):
> nom

Field type (enter ? to see all types) [string]:
> string

Field length [255]:
> 255

Can this field be null in the database (nullable) (yes/no) [no]:
> no

updated: src/Entity/Utilisateur.php

Add another property? Enter the property name (or press <return> to stop adding fields):
>
```

On peut voir ici toute ces étapes. On note que Symfony a créé notre entité (ici Utilisateur) et l'a placé dans un dossier « Entity », lui même dans le dossier « src » du projet. Il a également créé un « repository ».

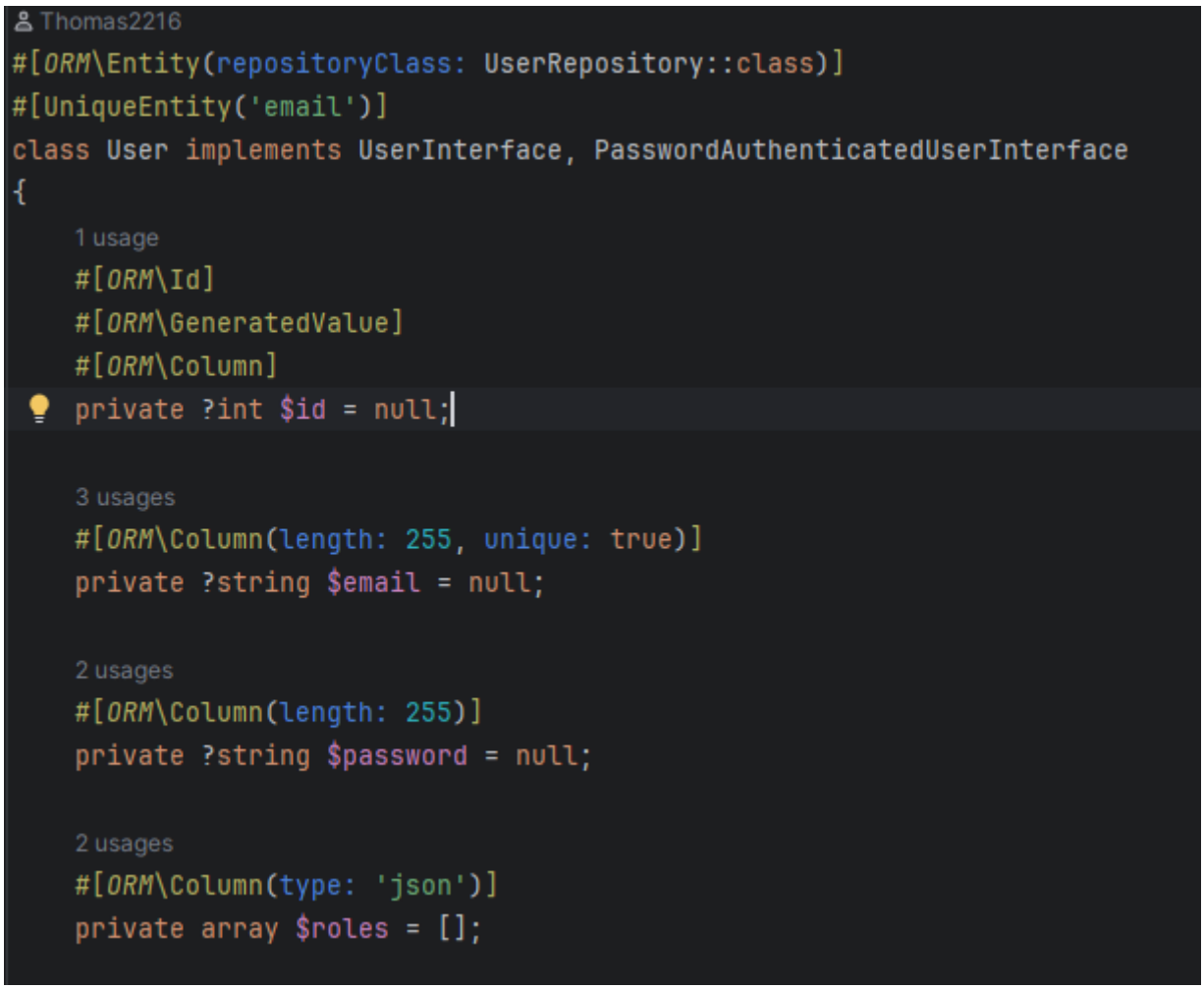
Allons maintenant voir le fichier PHP d'une de nos entité.

Comme vous pouvez le voir ci dessous, une classe a été créée.

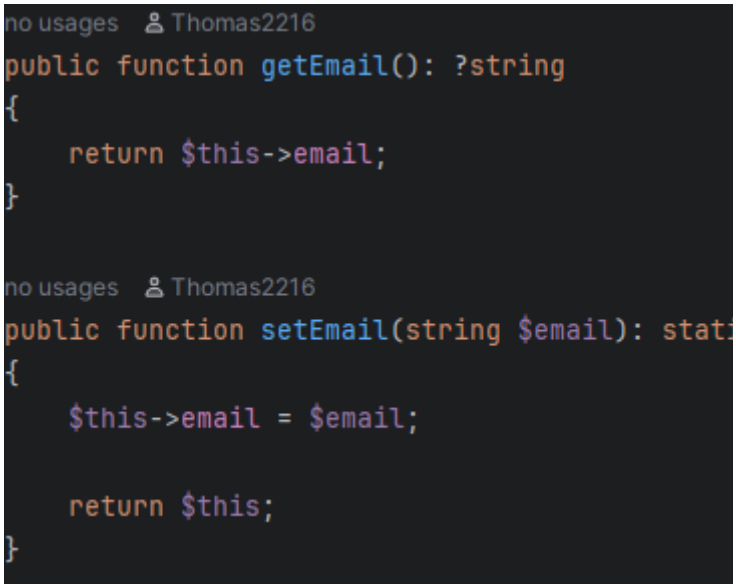
On voit que l'ORM (Object Relation Mapping), dans notre cas il s'agit de Doctrine, est mis en place. Il va permettre de manipuler la base de données. Un ORM transforme des tables SQL en objets PHP et inversement. Cela permet de travailler avec des objets PHP au lieu d'écrire des requêtes SQL à la main.

On note un bloc de code pour chaque champ de l'entité, avec le type, et si le champ est « nullable ».

Le champ « ID » est automatiquement créé au moment de la création de l'entité.



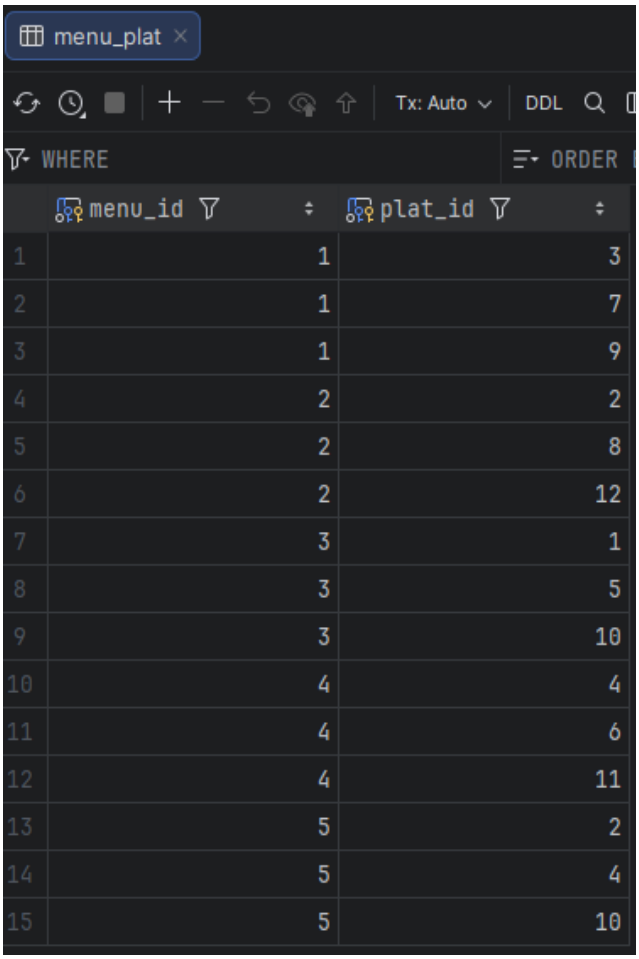
En plus de la définition de ces champs, cette classe contient des fonctions très importantes, communément appelées « getters » et « setters » pour chaque champ. Elles vont permettre de manipuler les données dans le reste de notre application. Elles se présentent comme ceci :



Maintenant que nous avons créé nos entités, il faut déterminer leur relations. Pour ajouter une relation entre deux entités, il faut ajouter un champ sur l'une d'elle, et définir ce champ sur la relation souhaitée. Le résultat dans l'entité se présente comme ceci :

On voit bien ici la relation de type « OneToMany » entre le User et la Commande (UN utilisateur peut avoir PLUSIEURS commandes).

Ici, Doctrine va créer des tables « invisibles » qui représenteront les relations entre nos entités. Ici par exemple une table « menu_plat » qui fait la relation entre nos entités menus et plats grâce à leurs « ID » :



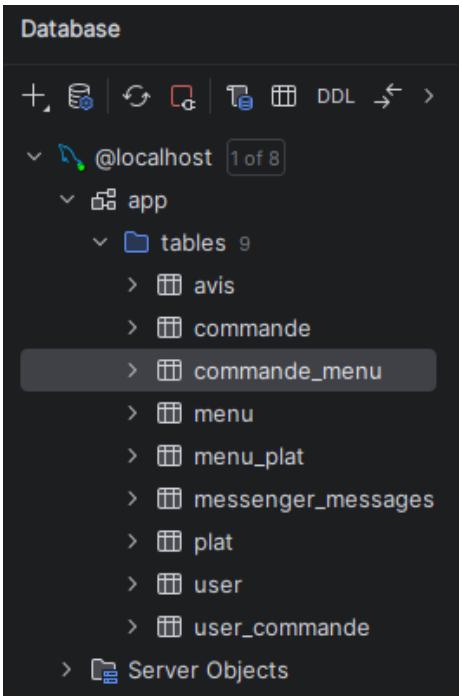
	menu_id	plat_id
1	1	3
2	1	7
3	1	9
4	2	2
5	2	8
6	2	12
7	3	1
8	3	5
9	3	10
10	4	4
11	4	6
12	4	11
13	5	2
14	5	4
15	5	10

Maintenant que nos entités sont créées, c'est à dire que les différents éléments et la structure de notre base de données sont définies dans notre application, il nous faut réellement créer notre base de données. Il existe des commandes très simples pour ceci, comme indiqué ici :

Ainsi nous créons notre base de données. Il faut maintenant y mettre à jour les schémas que nous avons créé juste avant, avec cette commande :

Et voilà ! Nous venons de créer notre base de données pour notre application.

Comme nous pouvons le voir ci-dessous, nous retrouvons notre base de données (avec notamment les tables invisibles qui font le lien entre les tables créées) dans notre IDE.



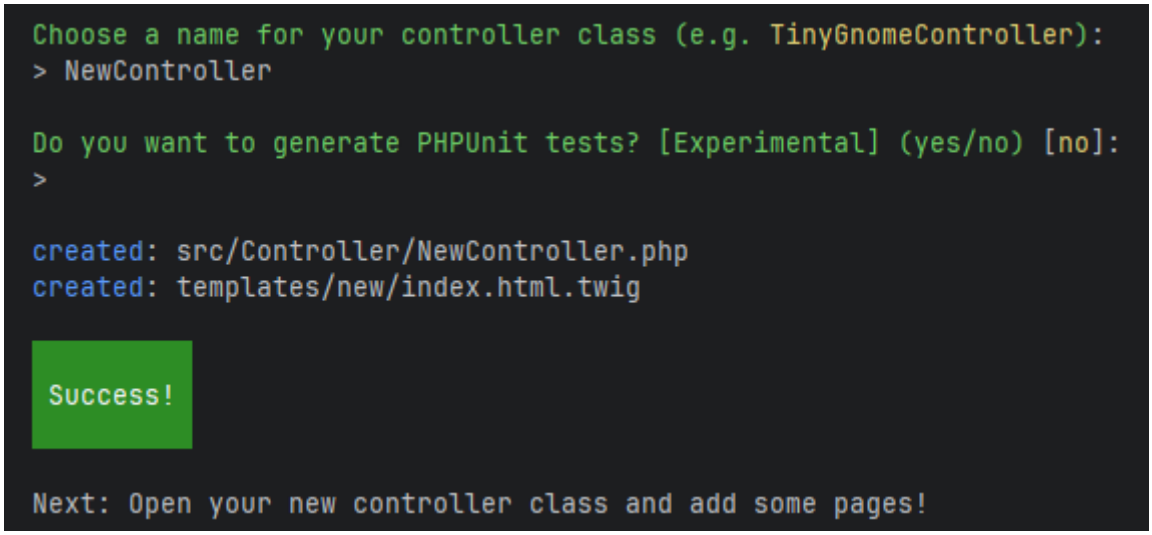
Développement des Routes et Contrôleurs

Il convient ensuite de créer les « contrôleurs », éléments essentiels de notre application. Comme dit précédemment, ces « contrôleurs » vont avoir

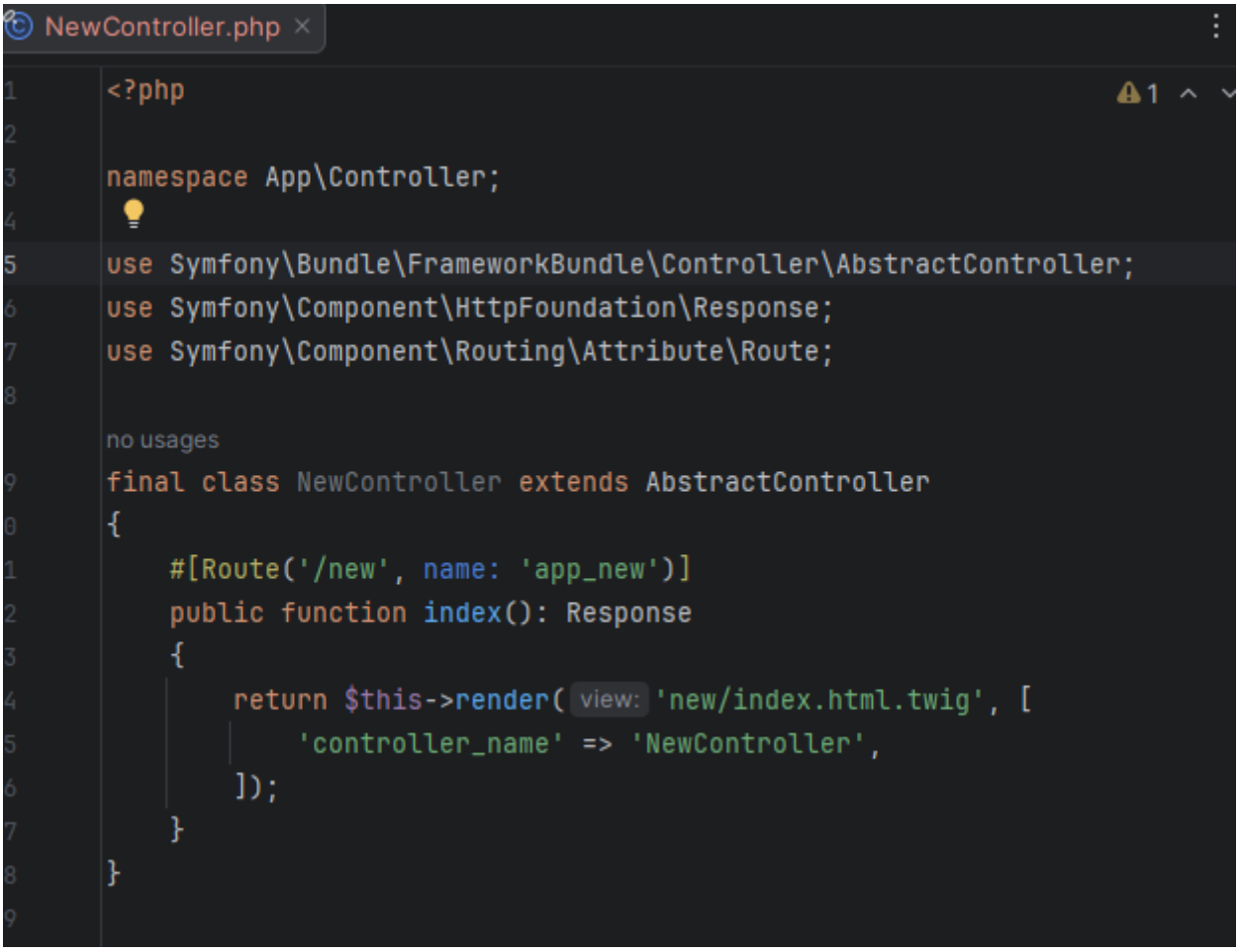
pour rôle de faire le lien entre les vues et la logique métier. Concrètement, c'est un fichier PHP qui va contenir les redirections de route, et le comportement que doit avoir l'application, principalement grâce à des fonctions PHP.

Nous allons donc créer notre contrôleur avec cette commande, toujours grâce au « makerbundle » :

Il est ensuite simplement demandé un nom à notre contrôleur et si nous souhaitons mettre en place des tests unitaires.



On note la création du contrôleur, mais également d'un template Twig, c'est un modèle de page vers laquelle renvoie le contrôleur, mais nous y reviendrons plus tard. Étudions maintenant le contrôleur créé :



On voit donc qu'une classe « NewController » a été créée. On observe les injections de dépendances (ici les fichiers précédés de « use »).

Dans cette classe, nous avons l'attribut #[Route('/new', name : 'app_new')] qui détermine le chemin de l'URL, ainsi que le nom de cette route.

Ensuite nous avons la fonction « index » qui va retourner (« return ») la vue associée, ici 'new/index.html.twig'.

Concrètement, l'application va recevoir une requête, et va appeler l'action du contrôleur pour générer la réponse. Ici c'est une redirection de page mais, comme nous le verrons plus tard, cela peut être des actions bien plus complexes.

Dans le cas de notre projet, nous allons utiliser ce processus pour créer

les différentes routes que nous stockerons principalement dans le contrôleur nommé « IndexController ».

Si il y a une logique plus poussée pour le fonctionnement de la page, (comme par exemple une authentification, une restriction d'accès par rapport aux droits, une logique d'affichage de liste dynamique, etc...) nous créerons un contrôleur dédié à cette page. C'est un choix de méthode de développement que de bien compartimenter les contrôleurs en fonction de leur rôle.

Vous avez dans les captures suivantes quelques exemples de définition de routes dans « l'IndexController »:

En fonction des routes, c'est à dire des « chemins URL », l'application renverra la vue correspondante.

Manipulation des données (CRUD) et présentation du bundle EasyAdmin

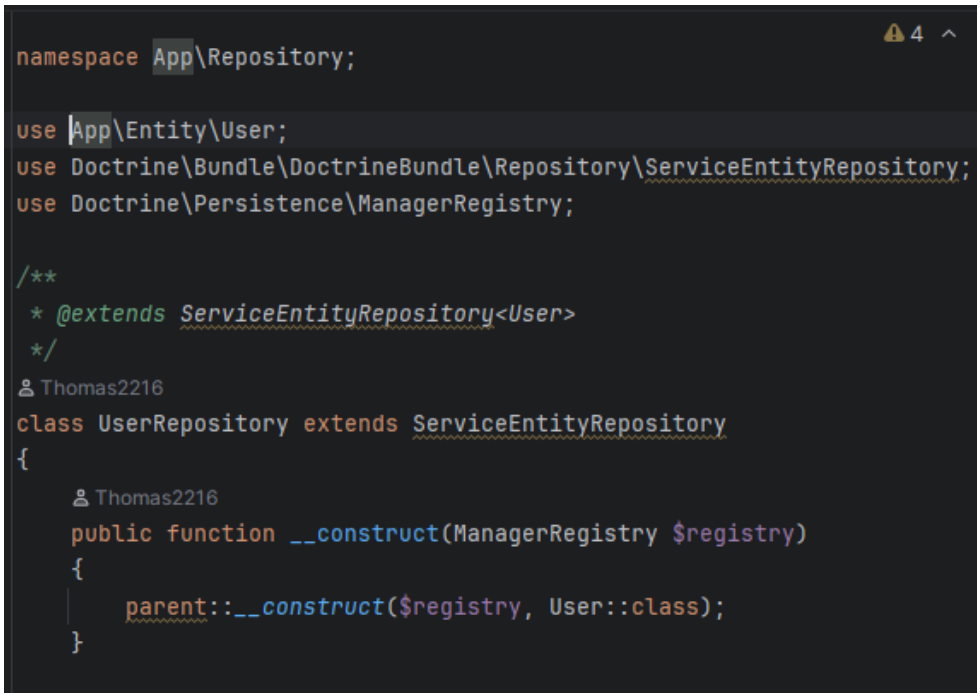
Pour développer notre application, il convient de mettre en place des fonctions qui vont nous permettre de manipuler nos données, que ce soit pour la création (Create), l'affichage (Read), la mise a jour (Update) ou la suppression (Delete) de données. Nous utiliserons l'acronyme « CRUD » pour définir cet ensemble d'actions. Ces actions peuvent très bien servir les fonctionnalités coté client (la création/suppression d'une commande dans notre cas), que coté administrateur (la modification d'un plat, d'un prix par exemple).

Nous allons dans un premier temps nous intéresser au « back-office » de notre application.

Nous pourrions imaginer, dans ce cas, un contrôleur que nous appellerions « UserCrudController », qui couvrirait toute les fonctions de

manipulation de l'entité « User ».

Lorsque nous avons créé nos entités, nous avons vu que Symfony a également créé un « repository » associé. Ces « repository » sont des classes Doctrine qui vont nous permettre d'accéder à nos données (Read). Elles se présentent comme ceci :



Dans notre contrôleur, nous pouvons faire appel à ce repository. Cela se présenterait comme ceci :

On voit bien dans cette fonction « index » que l'EntityManager va récupérer et retourner nos données via le repository.

Notons que nous appelons la methode « findAll » mais qu'il existe d'autres méthodes comme « findBy »(qui permet de récupérer plusieurs résultats par critère(s)), « findOneBy » (qui permet de récupérer un résultat via un ou plusieurs critères), ou « find » (qui permet de récupérer un résultat en passant un argument unique à la méthode, comme un identifiant par exemple).

Grâce à cette fonction, nous pouvons récupérer des données, ce qui nous permet de remplir la fonction « READ » de notre contrôleur « CRUD ».

Intéressons nous maintenant à la création d'une donnée. Il faut pour cela créer une fonction qui va, récupérer l'interface EntityManager, instancier l'objet à enregistrer (nous définirons ici manuellement ses attributs, même si généralement pour un contrôleur CRUD, on récupère un objet via un formulaire), une persistance de l'objet, et application de la modification en base de données via le « flush ». Voici comment la fonction se présente :

```php
#[Route('/user/create', name: 'app_user_create')]
public function create(EntityManagerInterface $em):Response
{
    $user = new User();
    $user->setNom( nom: "Valle");
    $user->setPrenom( prenom: "Thomas");
    $em->persist($user);
    $em->flush();

    return $this->redirectToRoute( route: 'app_user_index');

}
```

Grâce à cette fonction, nous avons pu créer un nouveau « user » et l'enregistrer en base de données, ce qui nous permet de remplir la fonction « CREATE » de notre contrôleur « CRUD ».

Pour la mise à jour des données, il y a quelques différences. On va ici récupérer l'objet via son ID, nous ne l'instancions pas. Il n'est pas nécessaire de persister avant l'enregistrement car l'objet existe déjà en base de données.

```php
#[Route('/user/update', name: 'app_user_update')]
public function update(int $id, EntityManagerInterface $em):Response
{
    $repository = $em->getRepository(User::class);
    $user = $repository->find($id);
    $user->setNom( nom: "Valle");
    $user->setPrenom( prenom: "Thomas");
    $em>flush();

    return $this->redirectToRoute( route: 'app_user_index');

}
```

Grâce à cette fonction, nous avons mis à jour notre objet «user » en base de données. Nous avons donc rempli la fonction « Update » de notre contrôleur « CRUD ».

Enfin, intéressons nous à la suppression de données. Nous allons récupérer un objet et utiliser la fonction « remove » pour préparer doctrine à la suppression.

```php
#[Route('/user/delete', name: 'app_user_delete')]
public function delete(int $id, EntityManagerInterface $em):Response
{
    $repository = $em->getRepository(User::class);
    $user = $repository->find($id);
    $em->remove($user);
    $em->flush();

    return $this->redirectToRoute( route: 'app_user_index');
}
```

Grâce à cette fonction, nous avons supprimé un objet en base de données . Nous avons donc rempli la fonction « Delete » de notre contrôleur

« CRUD ».

En programmation orientée objet (POO), les fonctions « CRUD » sont indispensables à la manipulation de données, et donc au bon fonctionnement de notre application.

Comme indiqué plus haut, Symfony bénéficie de bundles qui apportent des outils facilitant le développement, mais surtout qui apportent robustesse, sécurité et maintenabilité à notre application. Ces outils sont un gain de temps considérable pour le développeur, et donc d'efficacité.

C'est pourquoi, pour la gestion du back-office de notre projet, j'ai choisi d'utiliser **le bundle EasyAdmin 5** qui va gérer les manipulations de données coté admin. Il est bien sur possible de personnaliser totalement cet interface d'administration.

Pour créer cet interface, vu que nous avons déjà installé le bundle, il suffit de taper en ligne de commande :

```
Microsoft Windows [version 10.0.26200.8037]
(c) Microsoft Corporation. Tous droits réservés.

C:\Env\Workspace\Vite&Gourmand - Copie>php bin/console make:admin:dashboard
```

Nous avons maintenant notre interface d'administration, avec un contrôleur nommé « DashboardController ». Il faut donc en définir la route, ce que nous faisons ici :

```
Thomas2216
#[AdminDashboard(routePath: '/admin', routeName: 'admin')]
class DashboardController extends AbstractDashboardController
{
    Thomas2216
    public function index(): Response
    {
        return $this->redirectToRoute( route: 'admin_user_index');
    }
```

Il est également possible de personnaliser ce tableau de bord, comme ceci :

```php
no usages  Thomas2216
public function configureDashboard(): Dashboard
{
    return Dashboard::new()
        ->setTitle( title: 'Vite Gourmand')

        ->setLocales(['fr']);
}
```

Maintenant que nous avons notre tableau de bord, il faut créer nos contrôleurs « CRUD », ce qui se fait très facilement en ligne de commande, puisque nous avons déjà créé nos entités.

```
Microsoft Windows [version 10.0.26200.8037]
(c) Microsoft Corporation. Tous droits réservés.

C:\Env\Workspace\Vite&Gourmand - Copie>php bin/console make:admin:crud

 Which Doctrine entity are you going to manage with this CRUD controller?:
  [0] App\Entity\Avis
  [1] App\Entity\Commande
  [2] App\Entity\Menu
  [3] App\Entity\Plat
  [4] App\Entity\User
 >
```

On voit ensuite que Symfony nous demande à quelle entité associer le contrôleur.

Notre contrôleur « CRUD » est maintenant créé, avec tous les champs correspondants à l'entité que nous avons préalablement créée. Il se présente comme ceci :

```php
class UserCrudController extends AbstractCrudController
{
    no usages  Thomas2216
    public static function getEntityFqcn(): string
    {
        return User::class;
    }


    no usages  Thomas2216
    public function configureFields(string $pageName): iterable
    {
        return [
            Emailfield::new( propertyName: 'email'),
            TextField::new( propertyName: 'password', label: 'Mot de passe'),
            TextField::new( propertyName: 'nom'),
            TextField::new( propertyName: 'prenom', label: 'Prénom'),
            TextField::new( propertyName: 'adresse', label: 'Adresse postale'),
            TextField::new( propertyName: 'ville'),
            TelephoneField::new( propertyName: 'telephone', label: 'Téléphone'),
            ArrayField::new( propertyName: 'roles', label: 'Role')->setPermission( permission: 'ROLE_ADMIN'),

        ];
    }

}
```

On note qu'il est très facile de modifier des champs grâce à la fonction « configureFields ». Sur l'exemple ci-dessus, on note une modification du champ « rôles » pour qu'il puisse n'être modifié que par un utilisateur avec le 'ROLE_ADMIN '.

Ces contrôleurs vont permettre de remplir toutes les fonctions d'un contrôleur « CRUD » classique.

Nous allons retrouver tous ces contrôleurs sur le « DashboardController », où nous pouvons voir l'arborescence du back-office :

```php
no usages  Thomas2216
public function configureMenuItems(): iterable
{
    yield MenuItem::linkToDashboard( label: 'Tableau de Bord', icon: 'fa fa-table');
    yield MenuItem::linkToCrud( label: 'Utilisateurs', icon: 'fas fa-user', entityFqcn: User::class);
    yield MenuItem::linkToCrud( label: 'Menus', icon: 'fas fa-list', entityFqcn: Menu::class);
    yield MenuItem::linkToCrud( label: 'Commandes', icon: 'fas fa-eur', entityFqcn: Commande::class);
    yield MenuItem::linkToCrud( label: 'Plats', icon: 'fas fa-cutlery', entityFqcn: Plat::class);
    yield MenuItem::linkToRoute( label: 'Retour à Vite & Gourmand', icon: 'fa fa-home', routeName: 'app_accueil');
    yield MenuItem::linkToLogout( label: 'Déconnexion', icon: 'fas fa-sign-out-alt');
}
```

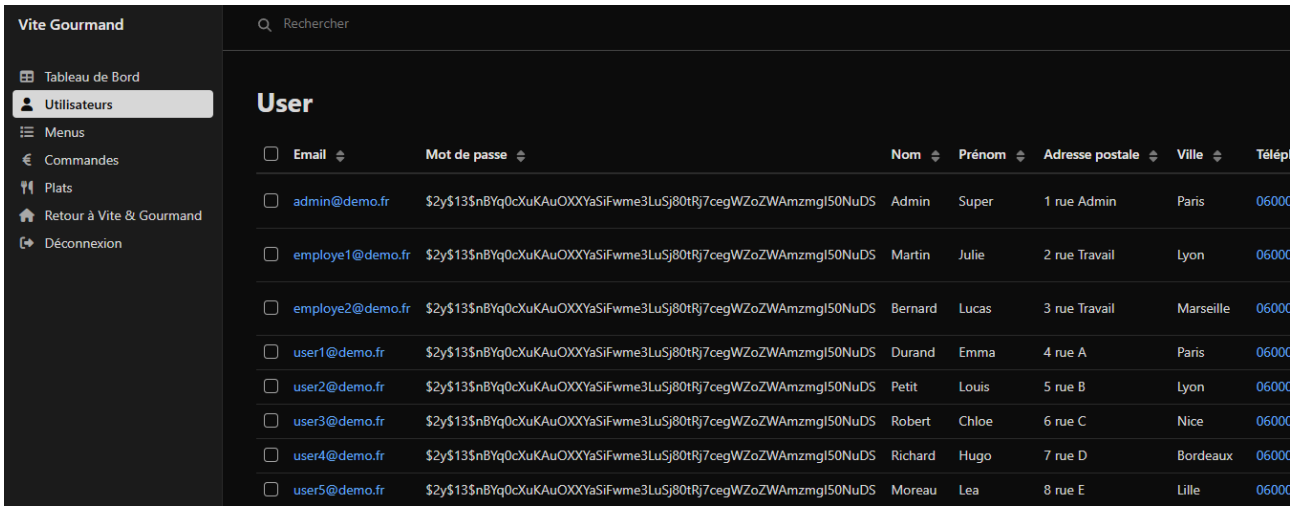
Nous avons pu y ajouter un retour aux vues de l'application, toujours utiles pour les administrateurs, ainsi qu'un bouton de déconnexion.

Enfin, et c'est extrêmement important, il faut restreindre les droits d'accès à ce back-office, pour qu'il ne soit accessible qu'aux employés et à l'administrateur. Cela se fait dans le fichier de configuration « security.yaml » comme ceci :

```
# Note: Only the *first* matching rule is applied
access_control:
    -   path: ^/admin
        roles: [ROLE_ADMIN , ROLE_EMPLOYE]
    # - { path: ^/profile, roles: ROLE_USER }
```

Ainsi, notre espace administrateur est sécurisé.

Sur la capture suivante, vous aurez un aperçu de l'interface administrateur. Sur la gauche, l'accès aux différentes entités, et sur la droite la base de données.



Nous avons donc notre back-office, configuré et sécurisé, et qui correspond aux demandes du client.

Requêtes SQL

Même si la plupart des bases de données se gèrent aujourd'hui via des

SGBD (Système de Gestion de Base de Données) de plus en plus ergonomique et facile d'accès, il convient de savoir utiliser le langage SQL (Structured Query Language) pour effectuer des requêtes dans sa base de données.

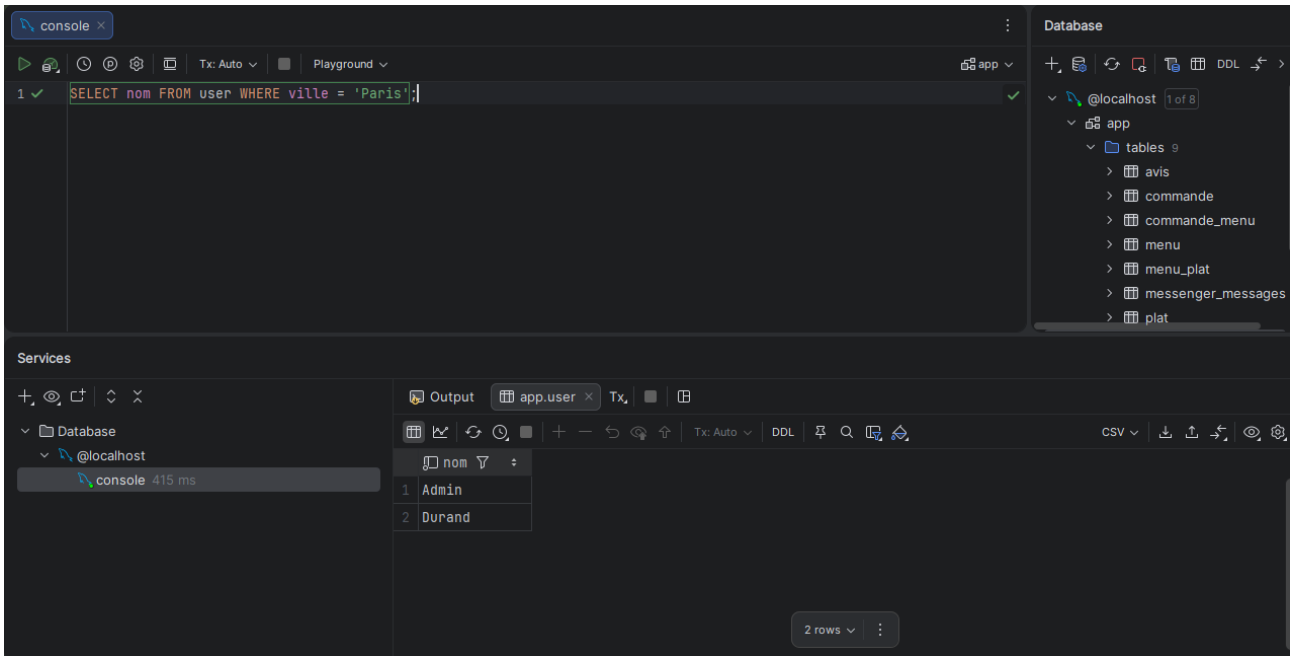
Ce langage a plusieurs intérêts majeurs, comme la recherche d'informations précises, le tri de données ou encore le filtre de résultats.

Il est à noter que pour notre projet, nous utilisons une base de données de type MySQL, mais il en existe d'autres, comme PostgreSQL, Oracle, SQLite.

Dans le cas de notre application, la base de données fictive a été entièrement créée grâce à des requêtes SQL.

Le principe du langage SQL est qu'il va utiliser des mots-clefs pour donner des instructions. Nous allons effectuer quelques exemples sur notre base de données.

Tout d'abord nous allons simplement extraire une donnée, ici nous souhaitons extraire (« SELECT ») le nom des utilisateurs (« FROM user ») qui vivent à Paris (« WHERE ville = 'Paris' »). Voici le résultat :



On obtient donc deux personnes qui vivent à Paris.

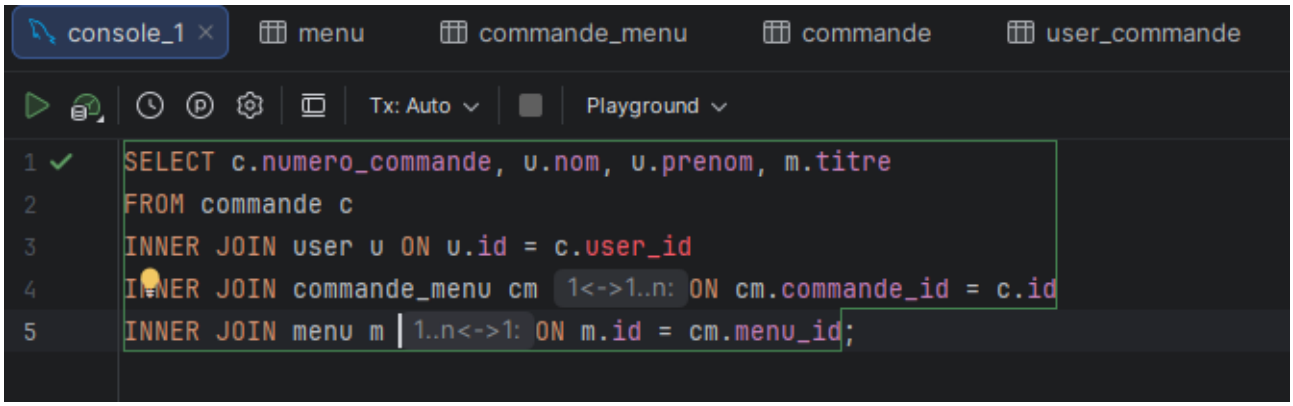
Les jointures sont l'une des fonctionnalités les plus puissantes du langage SQL. Elles permettent de combiner les données de plusieurs tables en une seule requête, en se basant sur une relation commune entre ces tables (généralement une clé étrangère). Sans les jointures, il serait impossible d'exploiter pleinement une base de données relationnelle.

Il existe plusieurs types de jointures. Les plus courantes sont :
INNER JOIN : retourne uniquement les lignes qui ont une correspondance dans les deux tables. C'est la jointure la plus utilisée.

LEFT JOIN : retourne toutes les lignes de la table de gauche, même si elles n'ont pas de correspondance dans la table de droite. Les colonnes de droite seront alors NULL.

RIGHT JOIN : l'inverse du LEFT JOIN. Toutes les lignes de la table de droite sont retournées, même sans correspondance à gauche.

Il est à noter qu'on utilise très régulièrement pour ces requêtes des alias pour simplifier leur écriture et leur compréhension. Voici un exemple :



Reprenons cette requête :

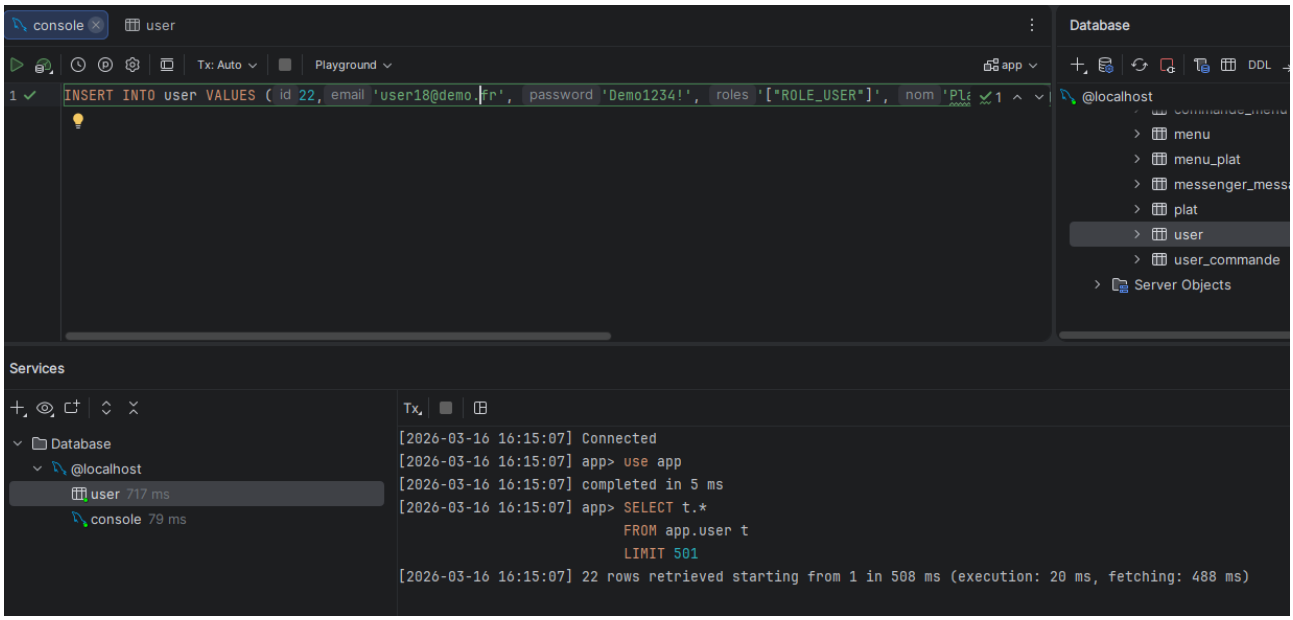
- Ici le « SELECT » sélectionne les champs qui nous intéressent, ceux que nous voulons récupérer. On note les préfixes qui sont les alias des tables dans lesquelles aller chercher ces champs.
- Le « FROM » est notre table principale, alias c.
- Le premier « INNER JOIN » est notre première jointure, elle fait le lien entre l'ID de la table « user » (clé primaire) et l' « user_id » (clé étrangère). Elle joint les utilisateurs qui ont passé commande.
- Le second « INNER JOIN » est notre deuxième jointure. Elle joint la table intermédiaire « commande_menu » qui fait le lien, comme son nom l'indique, entre la commande et les menus. On la relie à commande via « c.id »
- Le dernier « INNER JOIN » est notre dernière jointure. Elle joint la table « menu » via la table « commande_menu ». Cela nous permet de récupérer le titre des menus.

Cette requête permet de récupérer le détail des commandes en croisant les informations de trois tables. C'est bien plus puissant et rapide que de faire des requêtes séparées.

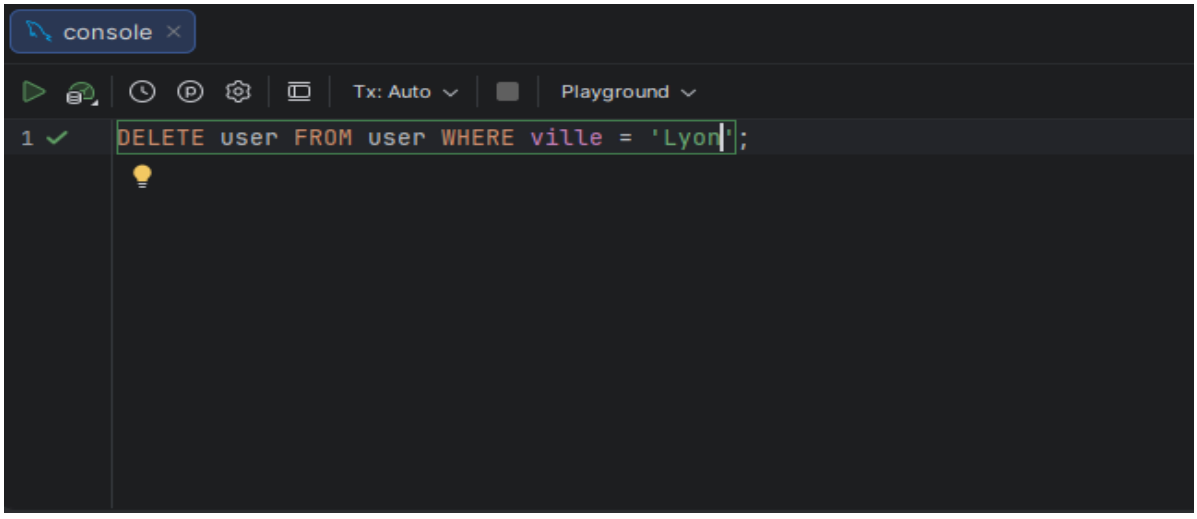
Dans notre application, ces requêtes sont générées automatiquement par

Doctrine ORM. Cependant, maîtriser le SQL brut est indispensable pour comprendre ce qui se passe en arrière-plan, déboguer des problèmes de performance, et justifier ses choix techniques.
Il est également possible, mis à part l'extraction de données, d'insérer, ou supprimer des données.

Dans l'exemple suivant, nous insérons (« INSERT INTO ») dans la table « user » les valeurs (« VALUES ») qui suivent, comme indiqué . On voit que la requête a bien été effectuée avec succès.



Maintenant voici une requête de suppression (« DELETE ») avec un critère de sélection.



Requêtes NoSQL

Dans le cadre de notre application, il est pertinent de développer une base de données dite « non-relationnelle », plus exactement « NoSQL » (Not Only SQL). La base de données non relationnelle adopte des modèles de stockage plus flexibles. Ils sont donc complémentaires des modèles « SQL » qui gèrent des données bien précises avec des relations complexes.

Dans notre cas (une application commerciale), il est judicieux d'utiliser le « NoSQL » pour, par exemple, la gestion de données statistiques, dont la structure peut être variable et le volume est potentiellement plus élevé. Ici nous avons choisi « MongoDB » qui est la base « NoSQL » de type document la plus répandue. Les données sont stockées en format JSON, ce qui apporte une grande flexibilité.

Dans Symfony, MongoDB est intégré via le bundle « doctrine ODM »(Object Document Mapper). Nous allons donc créer un document, l'équivalent de l'entité en SQL :

```php
<?php

namespace App\Document;

use Doctrine\ODM\MongoDB\Mapping\Annotations as ODM;

1 inheritor  Thomas2216
#[ODM\Document(collection: "commande_stats")]
class CommandeStat
{
    1 usage
    #[ODM\Id]
    private ?string $id = null;

    2 usages
    #[ODM\Field(type: "string")]
    private string $menuTitre;

    2 usages
    #[ODM\Field(type: "int")]
    private int $nombrePersonnes;

    2 usages
    #[ODM\Field(type: "float")]
    private float $prixTotal;
```

```php
2 usages
#[ODM\Field(type: "string")]
private string $periode;

2 usages
#[ODM\Field(type: "date")]
private \DateTime $dateCommande;

// Getters et Setters

no usages  Thomas2216
public function getId(): ?string
{
    return $this->id;
}

1 usage  Thomas2216
public function getMenuTitre(): string
```

On voit ici qu'on retrouve les champs (mappés avec #[ODM\Field...]), ainsi que les getters et setters.

On doit ensuite paramétrer ses variables dans notre .env.local :

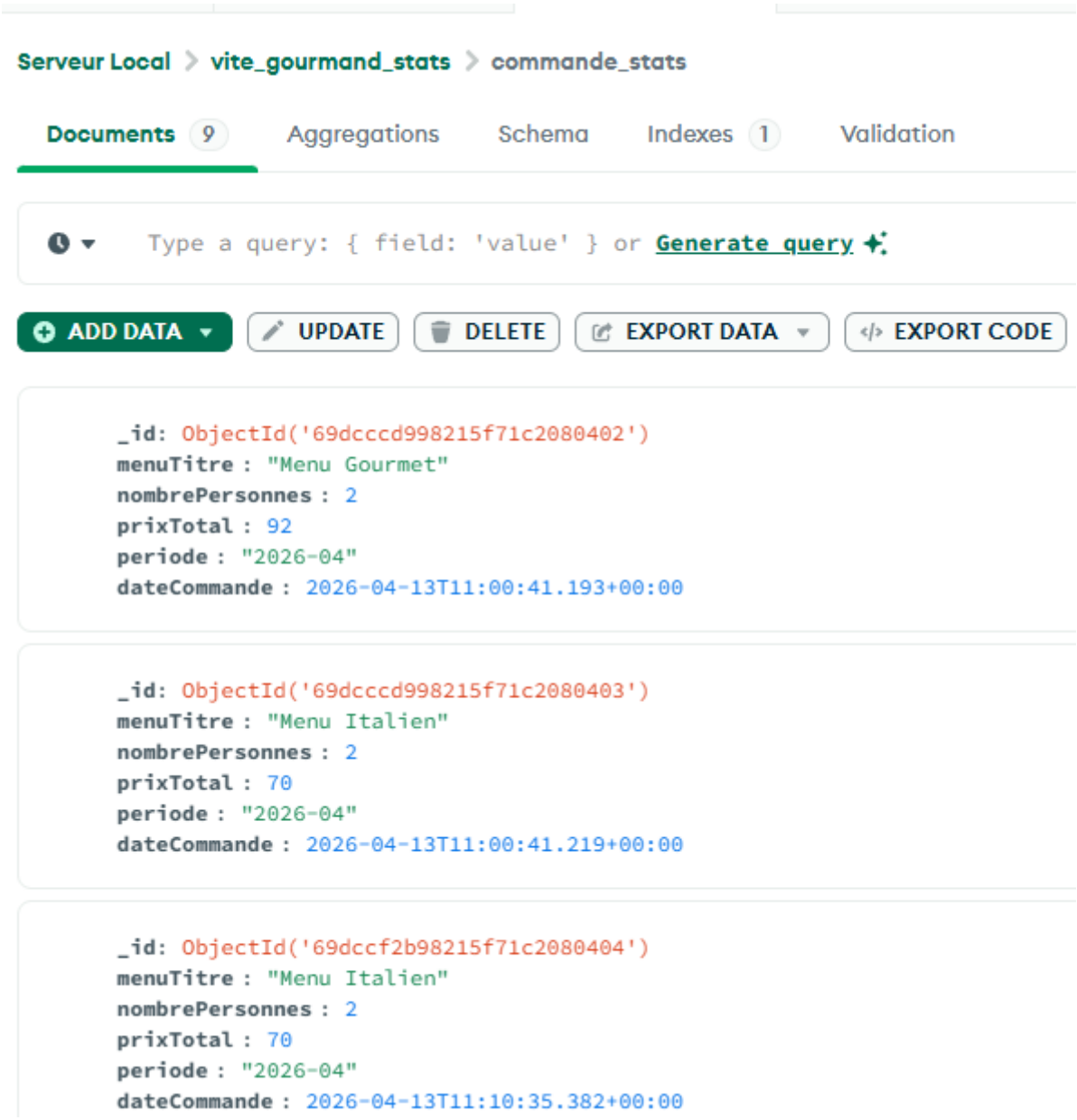
```
###> config mongoBD ###
MONGODB_URI=mongodb://localhost:27017
MONGODB_DB=vite_gourmand_stats
###< config mongoDB ###
```

Ensuite il convient d'intégrer notre logique d'enregistrement des données au contrôleur des commandes :

```php
try {
    foreach ($panier as $menuId => $quantite) {
        $menuItem = $em->getRepository(Menu::class)->find($menuId);
        if ($menuItem) {
            $stat = new CommandeStat();
            $stat->setMenuTitre($menuItem->getTitre());
            $stat->setNombrePersonnes($nombrePersonnes);
            $stat->setPrixTotal( prixTotal: $menuItem->getPrix() * $nombrePersonnes / 100);
            $stat->setPeriode((new \DateTime())->format( format: 'Y-m'));
            $stat->setDateCommande(new \DateTime());
            $dm->persist($stat);
        }
    }
    $dm->flush();
} catch (\Exception $e) {
}
```

(On note ici « dm » qui est le « DocumentManager » (équivalent de l'EntityManager des entités en SQL) mais les données sont envoyées ici à MongoDB et non à MySQL).

Ainsi, on retrouvera donc ces données dans notre interface MongoDB sous la forme de collections :



Ici, une collection (commande_stats) va regrouper plusieurs enregistrements, on parle donc de document (l'équivalent des lignes en SQL). La grande différence avec le SQL est qu'ici, le document est un objet JSON .

Ces données peuvent ensuite être exploitées dans le back-office. Dans le cas de notre application, nous avons mis en place un affichage de statistiques accessibles à l'administrateur, via la personnalisation d'un template Twig :

Statistiques des commandes

MENUS COMMANDÉS

MENU	NOMBRE DE COMMANDES	CHIFFRE D'AFFAIRES
Menu Gourmet	2	460,00 €
Menu Italien	4	560,00 €
Menu Healthy	2	192,00 €
Menu Express	1	76,00 €

Ces indicateurs sont très pertinent pour faciliter l'analyse de l'activité de l'entreprise Vite & Gourmand.

Personnalisation des templates Twig

Maintenant que nous avons pu mettre en place le « back-office » de notre application, nous allons pouvoir nous pencher sur la partie « visible » par n'importe quel utilisateur. Il y a à la fois un aspect statique, qui va être géré par de la mise en page HTML et CSS, mais également une partie dynamique, notamment d'affichage, des mécaniques de commandes qui vont agir sur la base de données. C'est tous ces points que nous allons aborder dans cette partie.

L'aspect « vue » du modèle « MVC est primordial. Nous allons tout

d'abord aborder la structure d'une page statique « classique » avant de voir comment elle va être utilisée dans notre framework Symfony, notamment via les templates Twig.

Le Langage HTML :

Le HTML (HyperText Markup Language) est un langage de balisage utilisé pour structurer une page Web. Ce n'est pas un langage de programmation, mais plutôt un langage qui va servir à décrire le contenu de la page Web.

Il va organiser le contenu des pages avec des balise comme :
<h1></h1> pour un titre
 pour un paragraphe de texte
<img> pour une image

Il existe une grande quantité de balises. On parle de balise ouvrante au début du contenu et de balise fermante à la fin. Certaines balises sont « auto-fermantes » (comme la balise <img> dans notre exemple) et ne nécessitent pas de balise fermante.

Il est important de préciser qu'il est possible de passer des attributs aux balises, comme des lien, des chemins vers des images , etc.

Enfin, le langage HTML respecte une structure bien définie. Nous allons avoir une partie <head></head> qui va contenir toutes les « métadonnées », comme le titre de la page, le lien vers la mise en page CSS ou Javascript. Ces données n'apparaissent que très peu sur le contenu de la page, mais sont essentielles à son bon fonctionnement.

Voici un exemple très simple d'un contenu de notre balise <head></head> :

```
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <title>Ma première page</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
```

Elle est précédée du type de document (html), ainsi que de la langue. On y trouve ensuite l'encodage de la page (ici UTF-8), le titre de la page, ainsi que le lien vers la stylisation de page (ici un fichier CSS de notre application).

Vient ensuite la balise <body></body> qui va contenir le contenu de

notre page. Elle se divise généralement en trois partie. Le <header></header> (en-tête), le <main></main> (le contenu principal), et le <footer></footer> (le pied de page. On pourrait imaginer la structure d'une page très simple comme ceci :

```
<body>

    <header>
        <h1>Bienvenue sur mon site</h1>
    </header>

    <main>
        <p>Ceci est le contenu principal de la page.</p>
    </main>

    <footer>
        <p>© 2026 Mon site</p>
    </footer>

</body>
```

Le langage HTML est relativement simple, mais c'est la structure de base de toute page Web. Voyons maintenant comment gérer le style « statique » de nos pages, grâce au langage CSS.

Le Langage CSS :

Le CSS (Cascading Style Sheets) est un langage utilisé pour mettre en forme notre page. Comme nous l'avons vu précédemment, il est relié à notre page HTML via une balise <link> que nous trouverons dans le <head></head>. (Il est à noter qu'il est tout à fait possible de passer du CSS à un élément HTML via un attribut « style » que nous pouvons passer dans la balise ouvrante de l'élément). Il va gérer les couleurs des différents éléments, la police, la mise en page, les espacements. Tout le code CSS devra être placé dans un fichier CSS de notre application. Voici un exemple très simple de code CSS :

```
h1 {
    color: blue;
    font-size: 24px;
}
```

On observe qu'il y a donc un sélecteur (ici « h1 ») qui va déterminer l'élément ciblé. Puis des propriétés (ici « color » et « font-size ») qui vont

déterminer ce que nous voulons personnaliser. Et enfin des valeurs (ici « blue » et « 24px ») qui vont assigner des valeurs à ces propriétés.

Nous aurons donc ici un titre qui prendra la couleur bleue et dont les caractères auront une taille de 24 pixels.

Pour sélectionner des éléments précis, on peut ajouter comme attributs des identifiants (sélecteur unique) ou des classes (sélecteurs multiples) dans les balises ouvrantes HTML. Cela se rédige comme ceci :

```css
/* Sélecteur de classe */
.texte {
    color: green;
}

/* Sélecteur d'ID */
#titre {
    color: red;
    font-size: 30px;
}
```

Enfin, il est important de parler de la mise en page que nous pouvons appliquer grâce au CSS. Les principales utilisées aujourd'hui sont la mise en page « FlexBox », qui va aligner les élément verticalement ou horizontalement, et « Grid » qui va pouvoir mettre en page de façon plus globale, sur plusieurs dimensions. Mais comment cela fonctionne ?

```css
.container {
    display: flex;
    justify-content: center;
    align-items: center;
}
```

Ici nous allons passer des propriété au sélecteur d'élément « .container » qui vont permettre d'aligner et centrer l'élément HTML grâce au display Flex.

```css
.container {
    display: grid;
    grid-template-columns: 1fr 1fr;
}
```

Ici, nous nous servons du display Grid pour séparer nos éléments en

deux colonnes.

Notons que ces « displays » sont régulièrement utilisés ensemble. Ils sont complémentaires.

Tous ces éléments sont vraiment la base de toute page Web, il est donc important pour tout développeur de les maîtriser.

Pour notre projet, nous avons choisi de développer la partie CSS en partie avec le framework Bootstrap, que l'on peut décrire comme une « boîte à outils » prête à l'emploi pour le design. Ce framework utilise des classes. Il contient des composants déjà prêts, comme des cards, des navbar par exemple, mais prend aussi en charge le « responsive design ».

Pour bénéficier de la magie de ce framework, il est nécessaire d'ajouter dans le <head></head> un lien CDN comme ceci :

```
{% block stylesheets %}
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.8/dist/css/bootstrap.min.css">
    <link rel="stylesheet" href="{{ asset('styles/app.css') }}">
{% endblock %}
```

Nous avons parlé précédemment du « responsive design ». C'est un élément essentiel pour toute application moderne, quand on sait que les utilisateurs peuvent aujourd'hui la consulter aussi bien sur un écran d'ordinateur, une tablette ou un mobile. Cet aspect est vraiment primordial. Le CSS va nous permettre de répondre à ce besoin. Tout d'abord grâce à « bootstrap », comme indiqué ci-dessus, qui va inclure des classes qui vont permettre de rendre notre contenu responsive. Prenons un exemple :

```
<body>
    <header>
        <nav class="navbar navbar-expand-lg bg-body-tertiary">
            <div class="container-fluid">
                <img class="logo" src="{{ asset('images/logo_2.png') }}" alt="Logo Vite & Gourmand"

                <button class="navbar-toggler" type="button"
                        data-bs-toggle="collapse"
                        data-bs-target="#navbarSupportedContent"
                        aria-controls="navbarSupportedContent"
                        aria-expanded="false"
                        aria-label="Toggle navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>

                <div class="collapse navbar-collapse" id="navbarSupportedContent">
                    <ul class="navbar-nav mx-auto mb-2 mb-lg-0">
                        <li class="nav-item">
                            <a class="nav-link" href="{{ path('app_accueil') }}">Accueil</a>
```

Nous voyons que dans le code de notre barre de navigation, il y a un bouton qui a pour classe « navbar-toggler ». Ce bouton est un menu burger qui va apparaître lorsque l'écran se réduit. C'est un des gros avantages de bootstrap, ses classes sont conçues de façon responsive.

Un autre aspect du responsive est ce que l'on appelle les « media queries ». Ce sont des classes CSS qui vont déterminer la stylisation des éléments en fonction de la taille de l'écran du navigateur. Cela se présente comme ceci :

```css
@media (max-width: 768px) {

    .nav-link {
        font-size: 16px;
        padding: 10px;
    }
    .navbar-nav {
        gap: 10px;
    }

    .presentation,
    .presentation-menu {
        padding: 40px 20px;
        font-size: 16px;
        line-height: 28px;
    }
```

Ici, le CSS dit « si l'écran fait moins de 768px, applique ces style-là ». Dans la pratique, il est judicieux d'utiliser les classes bootstrap et les « media queries » de façon complémentaire. C'est ce que nous avons fait pour notre projet « Vite & Gourmand ». Nous avons géré en « media queries » ce que bootstrap ne pouvait pas gérer pour ce qui est du responsive.

Symfony utilise un moteur de templates appelé « Twig ». Il va nous permettre de structurer notre code HTML, et d'y ajouter des variables, des boucles, des conditions, sans mettre du PHP dans nos templates. Cela rend le code plus lisible et organisé.

Nous avions noté que, à la création de nos contrôleurs, une page Twig associée était automatiquement créée. Cela peut-être pratique dans certains cas, mais cela nécessite en général un restructuration des pages Twig pour respecter l'arborescence de notre application.

En général, chaque template Twig « étend » une page « base html » qui va contenir elle tout le contenu redondant de notre application, comme le header et le footer par exemple. Cela se présente comme ceci :

base.html.twig

```
<!DOCTYPE html>
<html lang="fr">
    <head>
        <meta charset="UTF-8">
        <title>{% block title %}{% endblock %}</title>

        {% block javascripts %}
            {% block importmap %}{{ importmap('app') }}{% endblock %}
        {% endblock %}

        {% block stylesheets %}
            <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.8/dist/css/bootstrap.min.css">
            <link rel="stylesheet" href="{{ asset('styles/app.css') }}">
        {% endblock %}

    </head>
    <body>
        <header>
            <nav class="navbar navbar-expand-lg bg-body-tertiary">
                <div class="container-fluid">
                    <img class="logo" src="{{ asset('images/logo_2.png') }}" alt="Logo Vite & Gourmand" width="200" height="200">
                    <div class="collapse navbar-collapse" id="navbarSupportedContent">
                        <ul class="navbar-nav mx-auto mb-2 mb-lg-0">
                            <li class="nav-item">
                                <a class="nav-link" href="{{ path('app_accueil') }}">Accueil</a>
                            </li>

                            <li class="nav-item">
                                <a class="nav-link" href="{{ path('app_menus') }}">Menus</a>
                            </li>

                            <li class="nav-item">
                                <a class="nav-link" href="{{ path ('app_contact') }}">Contact</a>
                            </li>
                            {% if app.user is null %}
                                <li class="nav-item">
                                    <a class="nav-link" href="{{ path('app_login') }}">Connexion</a>
                                </li>
                            {% elseif is_granted('ROLE_ADMIN') or is_granted('ROLE_EMPLOYE') %}
                                <li class="nav-item">
                                    <a class="nav-link" href="{{ path('admin') }}">Espace Administrateur</a>
                                </li>
                            {% else %}
                                <li class="nav-item">
                                    <a class="nav-link" href="{{ path('app_user') }}">Votre Espace</a>
                                </li>
                            {% endif %}
```

```
                <div>
                    <a href="mailto:contact@viteetgourmand.fr" class="text-footer">Mail</a>
                </div>

                <div>
                    <a href="{{ path('app_mentions') }}" class="text-footer">Mentions légales</a>
                </div>

                <p>© 2026 Vite & Gourmand. Tous droits réservés.</p>
            </footer>
    </body>
</html>
```

```
                    </ul>
                </div>
            </div>
        </nav>
    </header>

    {% block body %}
    {% endblock %}

        <footer class="footer">
            <div>
                <img src="{{ asset('images/facebook-icon.png') }}" width="24" alt="logo fbk">
                <a href="https://www.facebook.com/" class="text-footer">Facebook</a>
            </div>

            <div>
                <img src="{{ asset('images/instagram-icon.png') }}" width="24" alt="logo insta">
                <a href="https://www.instagram.com/" class="text-footer">Instagram</a>
            </div>
```

Nous pouvons noter beaucoup de chose sur notre page « base.html.twig ». Tout d'abord que la structure de base d'une page HTML est respectée. On retrouve bien tous les éléments nécessaires dans la balise <head></head>, mais que ces éléments sont définis avec une syntaxe particulière de type {% … %}. Dans d'autres parties de la page on note des éléments qui sont définis avec des doubles accolades {{…}}.
Cette syntaxe particulière permet d'inclure dans notre fichier Twig des variables :

```
<img class="logo" src="{{ asset('images/logo_2.png') }}"
```

Comme ici un chemin d'accès à une image placée dans les assets de l'application.

Mais elle permet aussi et surtout d'inclure des blocks réutilisables, comme ici :

```
<title>{% block title %}{% endblock %}</title>
```

(ici nous pourrons via ce « block title » définir le titre de chacune de nos pages indépendamment des données de la balise <head></head>).

Enfin elle permet d'inclure de la logique dans notre code HTML, comme ici :

```
{% if app.user is null %}
    <li class="nav-item">
        <a class="nav-link" href="{{ path('app_login') }}">Connexion</a>
    </li>
{% elseif is_granted('ROLE_ADMIN') or is_granted('ROLE_EMPLOYE') %}
    <li class="nav-item">
        <a class="nav-link" href="{{ path('admin') }}">Espace Administrateur</a>
    </li>
{% else %}
    <li class="nav-item">
        <a class="nav-link" href="{{ path('app_user') }}">Votre Espace</a>
    </li>
{% endif %}
```

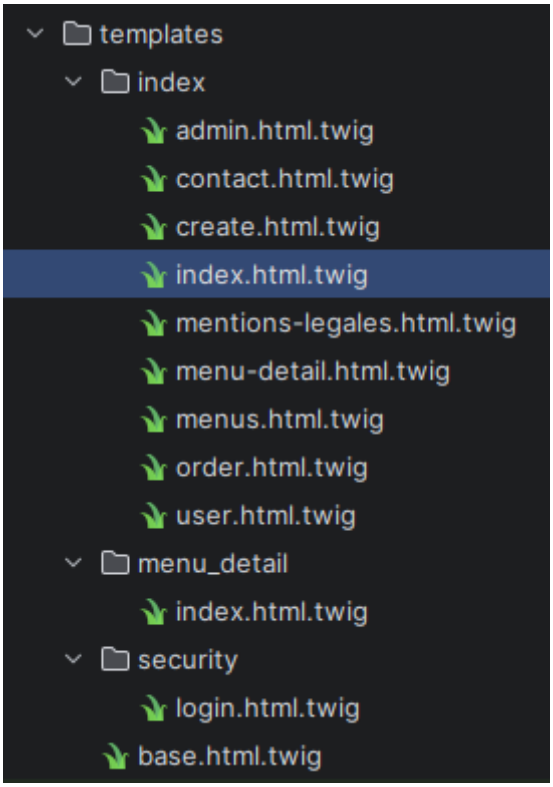
On voit bien ici que nous avons inclus une condition qui va modifier l'affichage en fonction de quel, et si l'utilisateur est connecté.

Il est à noter que sur ce template Twig, un stylisation bootstrap est utilisée, pour la Navbar, on y voit les classes Bootstrap directement dans les attributs des éléments HTML.

L'un des avantages de Twig, est que nous pouvons étendre cette base pour l'inclure à toute nos pages, et parallèlement remplir les blocks pour personnaliser chaque page, comme ceci :

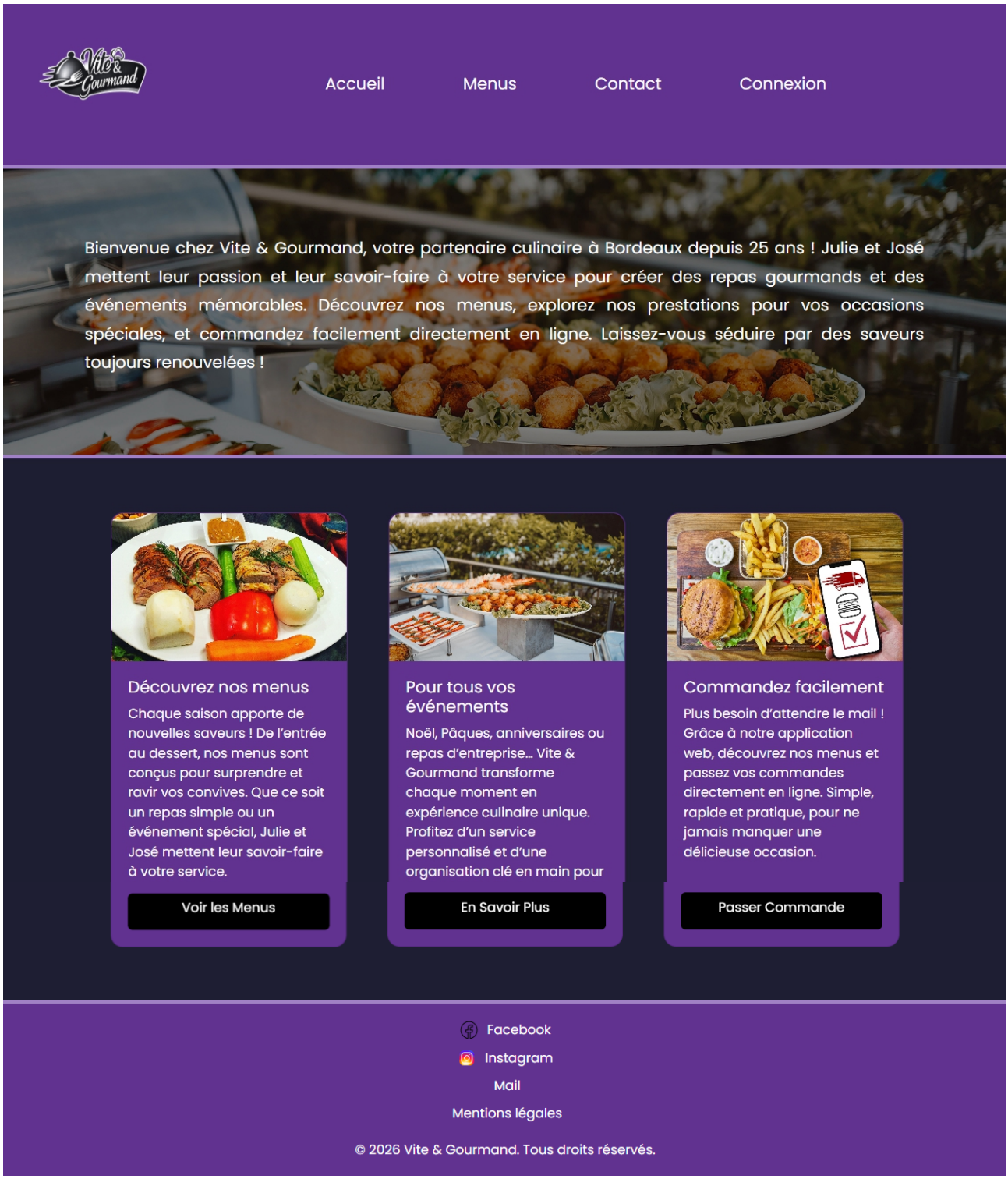
```
{% extends 'base.html.twig' %}

{% block title %}Vite & Gourmand{% endblock %}
```

Tous ces fichier twig sont placés dans un dossier « templates » dans l'arborescence de notre application comme ceci :



Nous venons de présenter la logique du moteur de templates Twig, qui va gérer l'affichage des vues sur notre application.

Voici un aperçu du rendu final de la page d'accueil de l'application :



Les formulaires

Nous avons vu que pour le back-office, les échanges avec la base de données se faisaient via notre bundle EasyAdmin, mais qu'en est-il si c'est un utilisateur qui doit avoir accès à ces données ? Pour une connexion ou une prise de commande par exemple, l'utilisateur va avoir besoin d'accéder à la base de données de façon sécurisée et performante. Ces opérations vont se faire via des formulaires (de connexion, de création de commande, etc.).

Le formulaire va être un élément essentiel d'une application moderne. Dans notre cas, nous montrerons quelques exemples concrets utilisés pour le bon fonctionnement de notre projet.

Grâce à Symfony, la création et gestion des formulaires est simplifiée. En effet, en quelques lignes de code, nous pouvons générer des formulaires et les rattacher à des entités, comme vous pouvez le voir ici :

```
C:\Env\Workspace\Vite&Gourmand - Copie>php bin/console make:form

 The name of the form class (e.g. OrangeGnomeType):
 > OrderType

 The name of Entity or fully qualified model class name that the new form will be bound to (empty for none):
 > Commande

 created: src/Form/OrderType.php

  Success!
```

On voit ici qu'on doit déterminer le nom de notre formulaire (en Symfony, on utilise la syntaxe « Type » pour désigner un formulaire). On peut également déterminer une entité avec laquelle relier notre formulaire (ici l'entité « Commande »).

Nous allons maintenant prendre l'exemple d'un cas concret, en l'occurrence la création d'un utilisateur via un formulaire, pour notre application.

Nous avons donc créé un formulaire que nous retrouvons dans un fichier PHP, ici « UserType ». Mais que contient ce fichier PHP ?

On voit que nous avons une classe « Usertype » qui contient différentes fonctions dont un fonction qui va déterminer l'architecture de notre formulaire en fonction des champs que nous allons lui passer (champs qui sont issus de notre entité correspondante).

Nous avons l'intention d'utiliser ce formulaire dans une vue Twig appelée « create.html.twig », dans le but qu'un utilisateur puisse créer un compte utilisateur.

Voici comment se présente cette vue, en terme d'architecture :

```twig
{% extends 'base.html.twig' %}

{% block title %}Créer son profil{% endblock %}

{% block body %}
    <h1 class="create-title">Créez votre Profil</h1>

    <div>
        {{ form_start(form, {'attr': {'class': 'creation-form'}}) }}
        {{ form(form) }}
        {{ form_end(form) }}
    </div>

{% endblock %}
```

On peut voir que le code de cette page est très simple. Mais alors, où est le lien entre notre formulaire, et celle-ci ? Tout simplement dans le contrôleur. Notons que nous avons fait le choix de mettre cette logique dans le contrôleur « SecurityController » qui va regrouper toute la logique de connexion, création de compte, etc de notre application. Observons donc maintenant ce qui se passe dans ce dernier :

```
Thomas2216
#[Route('/create', name: 'app_create', methods: ['GET', 'POST'])]
public function create(HttpFoundationRequest $request, EntityManagerInterface $em): Response
{
    // Crée un nouvel utilisateur
    $user = new User();
    $user->setRoles(['ROLE_USER']); // rôle par défaut

    // Crée le formulaire
    $form = $this->createForm( type: UserType::class, $user);
    $form->add( child: 'submit',  type: SubmitType::class, [
        'label' => 'Créer',
        'attr' => ['class' => 'btn btn-primary'],
    ]);

    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        // Hasher le mot de passe avant de persister
        $hashedPassword = $this->passwordHasher->hashPassword(
            $user,
            $user->getPassword() // le mot de passe en clair saisi dans le formulaire
        );
```

```
            $user->setPassword($hashedPassword);

            $em->persist($user);
            $em->flush();

            return $this->redirectToRoute( route: 'app_user');
        }

        return $this->render( view: 'index/create.html.twig', [
            'form' => $form->createView(),
        ]);
    }
}
```

On a tout d'abord notre Route qui passe les méthodes « GET » et « POST », qui, comme leur nom l'indique, vont permettre de récupérer et envoyer des données au serveur.

Ensuite nous avons une fonction « create », dans laquelle nous allons créer un nouvel utilisateur via la variable $user. Notons que nous définissons un rôle par défaut, nous y reviendrons dans la partie « Sécurité » de ce projet. Nous créons ensuite notre formulaire, en appelant notre fameux « UserType » en argument de la méthode « createform » qui va réellement créer notre formulaire. Ensuite nous ajoutons la méthode « add » qui va valider les informations saisies.

Ensuite nous avons la méthode « handlerequest » qui va récupérer les données de la requête, et gérer la soumission et la validation du formulaire.

Puis nous avons une condition qui vérifie que le formulaire est bien soumis et valide. Notons que nous avons un système de hash de mots de passe mais nous y reviendrons dans la partie « Sécurité » de ce dossier. Nos données vont ensuite être « persistées » (préparées à l'envoi) , puis « flushées » (envoyées en base de données).

Enfin la redirection se fait sur la page utilisateur si la création d'utilisateur est valide, sinon sur la page de création d'utilisateur si ce n'est pas le cas.

Nous avons ainsi vu la création et l'utilisation de formulaires en PHP/Symfony.

Requêtes AJAX/ API Fetch

Il est maintenant temps d'aborder un élément essentiel et très répandu dans la conception d'applications modernes, ce sont les interfaces utilisateurs dynamiques. En effet, l'utilisateur doit pouvoir interagir avec le DOM (faire une requête réseau), sans que cela soit bloquant pour la navigation, et plus précisément sans recharger la page. C'est ce que l'on appelle une requête « asynchrone », ou AJAX (Asynchronous Javascript And XML).

Notons que AJAX est l'acronyme historique, mais que de nos jours, les données sont traitées presque exclusivement en format JSON, plus léger et facile à manipuler que le XML d'origine.

Dans le cadre de notre application, nous allons utiliser les requêtes asynchrones pour gérer l'affichage dynamique du panier lorsque l'utilisateur passe une commande (mise à jour instantanée du prix total, des quantités, des frais de livraison). Mais concrètement, comment cela se passe ?

Tout d'abord il faut « préparer » le template, la vue. Nous allons créer des vide, ou à compléter, qui se mettront à jour de façon dynamique, et qu'il faudra identifier avec des « ID » pour pouvoir les récupérer dans notre fichier de logique javascript.

```
{# ── COLONNE DROITE – RÉCAP STICKY ──────────────── #}
<div class="cmd-col-right">
    <div class="cmd-recap">
        <h2 class="cmd-recap-title">Votre commande</h2>

        <ul id="menu-list" class="cmd-recap-list"></ul>

        <div id="detail-prix" class="cmd-recap-detail">
            <div class="cmd-recap-divider"></div>
            <p id="sous-total-menus" class="cmd-recap-line">Sous-total menus : –</p>
            <p id="ligne-reduction" class="cmd-recap-line cmd-recap-reduction">Réduction 10 % : –</p>
            <p id="ligne-frais" class="cmd-recap-line">Frais de livraison : –</p>
            <div class="cmd-recap-divider"></div>
            <p id="prix-total" class="cmd-recap-total">Total : –</p>
        </div>

        {{ form_row(form.submit) }}
    </div>
</div>
```

Ici, on a notre qui va représenter notre panier avec les différents champs, bien identifiés, qui doivent être mis à jour (« sous-total-menus », « ligne-reduction »...).

Il faut ensuite construire notre fichier javascript, qui sera placé dans nos « assets ». Nous allons établir plusieurs choses, tout d'abord déclarer les constantes des éléments qui nous intéressent, grâce à leur « ID ». Cela permet de pointer exactement sur ces éléments.

```
const btnAddItem : HTMLElement        = document.getElementById( elementId: "btn-add-item");
const menuList : HTMLElement          = document.getElementById( elementId: "menu-list");
const totalPrix : HTMLElement         = document.getElementById( elementId: "prix-total");
const selectMenus : HTMLElement       = document.getElementById( elementId: "liste-menus-a-selectionner");
const adresseInput : HTMLElement      = document.getElementById( elementId: 'commande_adresse');
const fraisLivraisonEl : HTMLElement  = document.getElementById( elementId: 'frais-livraison');
```

Ensuite nous allons avoir un écouteur d'événement qui va dire « si il se passe ça » (comme un clic, un survol d'un champ, etc.), essaye de faire ceci (try), sinon fais cela (catch). Le bloc « try » contient le code à exécuter. Si une erreur survient (problème de serveur, réponse invalide …), le bloc « catch »

l'intercepte et permet de gérer l'erreur proprement.

Ici, vous pouvez voir que nous écrivons dans le code que, au clic sur le bouton add item, on essaye de récupérer une réponse réseau grâce à l'API fetch() de manière asynchrone. Le « corps » de la réponse attendue est ensuite détaillé (le type, le format, etc). Le mot-clé « await » suspend l'exécution de la fonction jusqu'à ce que la réponse du serveur arrive. Pendant ce temps, le reste de la page reste entièrement utilisable, c'est le principe de l'asynchrone. Puis le panier est mis à jour grâce à la méthode « rafraichirPanier » déclarée plus haut.

```
btnAddItem.addEventListener( type: "click",  listener: async () : Promise<void>  =>
```

```
try {
    const reponse : Response  = await fetch( input: '/add_menu',  init: {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify( value: { menuId: menuId, action: 'add', nombrePersonnes: nombrePersonnes })
    });

    if (!reponse.ok) throw new Error("Erreur serveur : " + reponse.status);

    const data = await reponse.json();
    rafraichirPanier(data);

} catch (erreur) {
    menuList.innerHTML = `<strong>Oups !</strong> Impossible d'ajouter le menu.<br><small>${erreur.message}</small>`;
    menuList.style.display = 'block';
    menuList.style.color = '#d9534f';
    console.error("Détail complet de l'erreur :", erreur);
}
```

Ainsi, la réponse est gérée de manière non-bloquante. Pendant la requête réseau, le navigateur reste entièrement actif.

L'API fetch() est la méthode moderne pour réaliser des requêtes HTTP asynchrones en Javascript. On l'utilise ici en lui passant notamment la route « /add_menu » qui va être la route de notre logique métier pour ajouter un menu. On la retrouve dans un contrôleur dédié, détaillé ici :

```php
Thomas2216
#[Route('/add_menu', name: 'add_menu', methods: ['GET', 'POST'])]
public function addItem(HttpFoundationRequest $request, EntityManagerInterface $em): JsonResponse
{
    $data = json_decode($request->getContent(), associative: true);
    $menuId = $data['menuId'] ?? null;

    $session = $request->getSession();
    $panier = $session->get(name: 'panier', []);

    $action = $data['action'] ?? 'add';
    $nombrePersonnes = $data['nombrePersonnes'] ?? 1;

    if ($action == 'add') {
        if (isset($panier[$menuId])) {
            $panier[$menuId] = $nombrePersonnes;
        } else {
            $panier[$menuId] = $nombrePersonnes;
        }
    } elseif ($action == 'remove') {
        if (isset($panier[$menuId])) {
            $panier[$menuId]--;
            if ($panier[$menuId] <= 0) {
                unset($panier[$menuId]);
            }
        }
    }

    $session->set('panier', $panier);

    $totalItems = 0;
    $totalPrix = 0;
    $detailPanier = [];

    foreach ($panier as $menuId => $quantite) {
        $menu = $em->getRepository(Menu::class)->find($menuId);
        if ($menu) {
            $sousTotalMenu = $menu->getPrix() * $quantite;

            $totalItems += $quantite;
            $totalPrix  += $sousTotalMenu;
```

```
                $totalItems += $quantite;
                $totalPrix  += $sousTotalMenu;

                $detailPanier[] = [
                    'menuId'    => $menuId,
                    'quantite'  => $quantite,
                    'titre'     => $menu->getTitre(),
                    'prix'      => $menu->getPrixFormate(),
                    'sousTotal' => $sousTotalMenu / 100,
                    'minPersonne' => $menu->getMinPersonne(),
                ];
            }
        }

        $totalPrix = $totalPrix / 100;

        return $this->json([
            'totalItems'   => $totalItems,
            'totalPrix'    => $totalPrix,
            'detailPanier' => $detailPanier,
            'action'       => $action,
        ]);
    }
}
```

On a donc notre fonctionnalité qui ajoute le menu, elle prend en argument les outils de requêtes HTTP, et l'entitymanager de Symfony qui lui permettront de manipuler les entités à travers les requêtes. On note ici une logique de mise à jour de données, avec notamment un « foreach » qui gèrera chaque mise à jour du panier. La réponse attendue est évidemment en JSON, et le DOM est mis à jour de façon dynamique.

Grâce à cette implémentation , le panier se met à jour en temps réel, offrant une expérience fluide et moderne.

Docker

Dans le cadre du développement d'une application, nous pouvons rencontrer un problème majeur : il faut pouvoir garantir que l'application fonctionne de la même manière sur n'importe quelle machine. En effet, un développeur tierce qui va cloner notre projet doit installer manuellement PHP, MySQL, Nginx et configurer les bonnes versions. Cela peut facilement engendrer des problèmes, et vite devenir fastidieux.

Heureusement, Docker nous apporte une solution pour pallier à ce problème. Il va empaqueter notre application et toutes ses dépendances dans des conteneurs (on parle de conteneurisation). Ces conteneurs sont des unités légères et autonomes qui vont contenir tout ce dont l'application a besoin pour

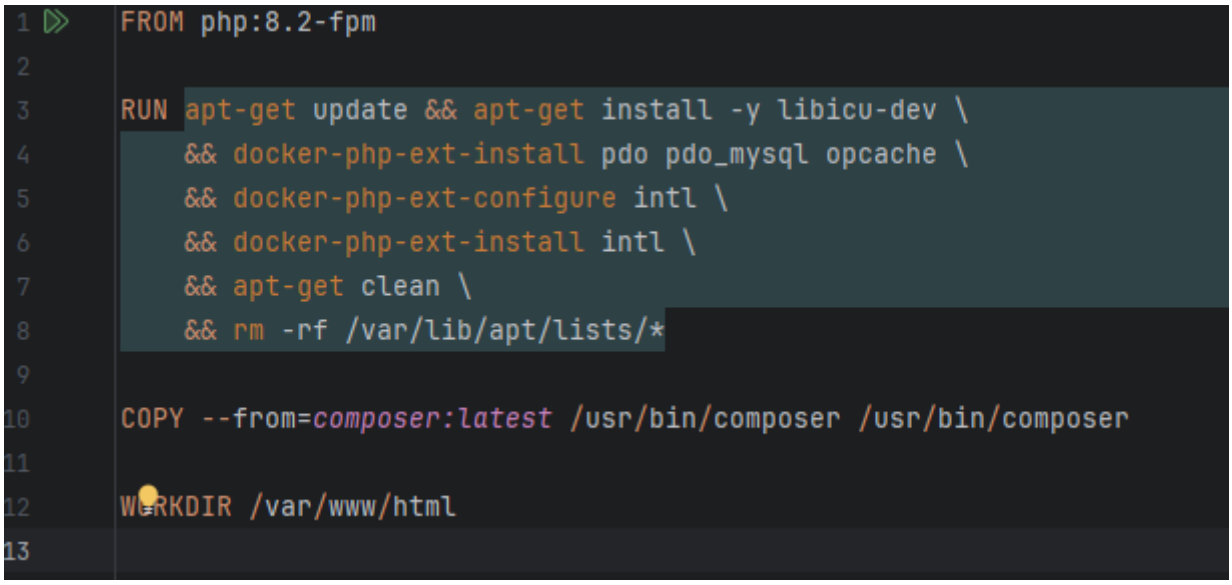
bien fonctionner, peu importe l'environnement.

Dans le cas de notre application, chaque conteneur va contenir un « service » différent, qui seront en lien grâce à un réseau virtuel géré par Docker. Ces services sont au nombre de quatre :

- php (exécute Symfony/PHP)
- nginx (gère les requêtes HTTP)
- database (gère la base de données)
- mailer (gère les emails)

Il est à noter que pour chaque service il sera déterminé une « image » qui est un environnement complet figé. Elle peut être issue de la bibliothèque Docker (comme ici MySQL ou Nginx) ou personnalisée (comme ici php).

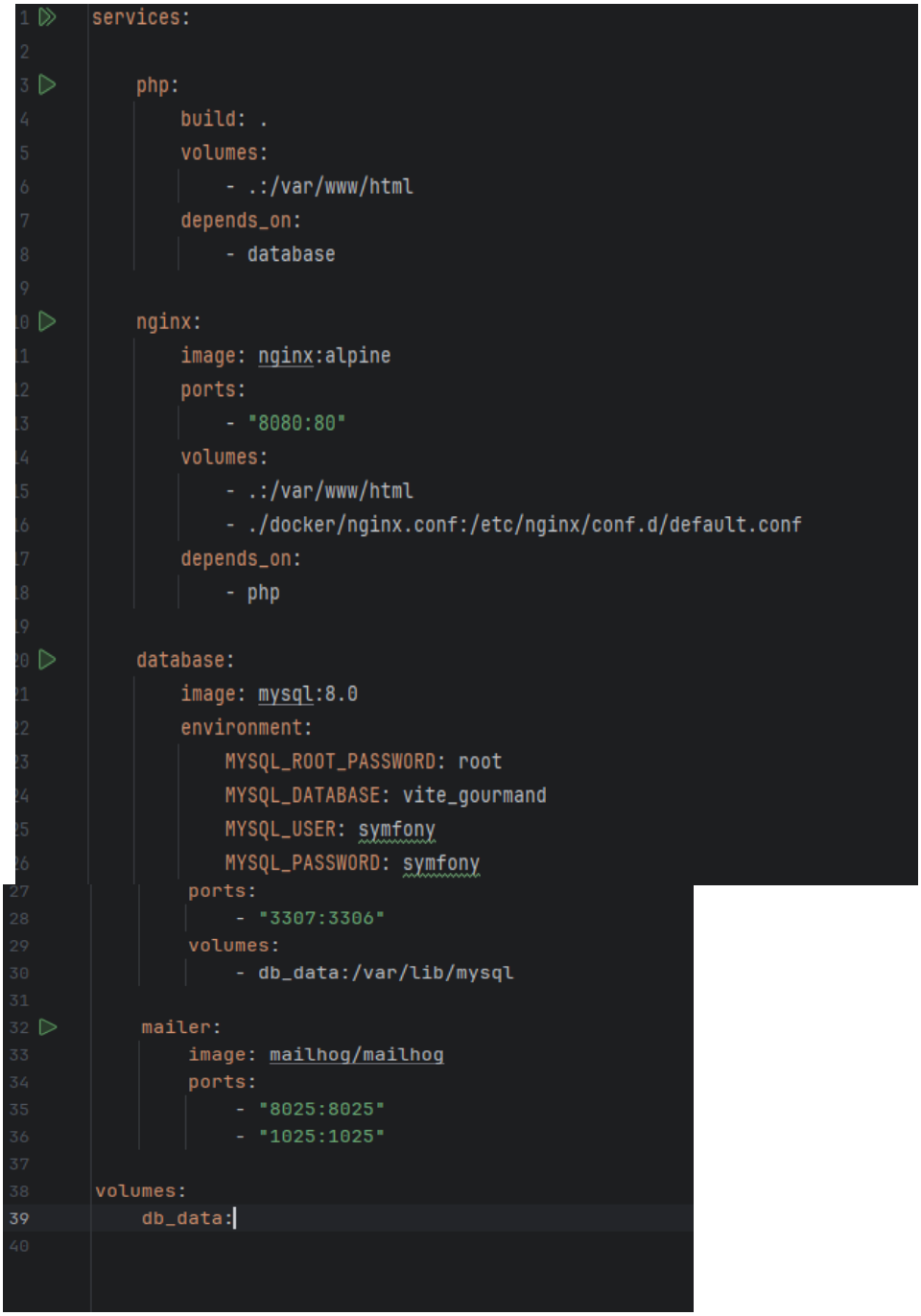
Docker nécessite bien sur une configuration. Tout d'abord le « Dockerfile » qui va déterminer la « recette » de construction de l'image PHP personnalisée.



On voit qu'on part de l'image officielle php:8.2-fpm disponible sur le Dockerhub, et on installe ensuite les dépendances nécessaires ainsi que les extensions. On intègre ensuite composer, et enfin, on détermine un dossier de travail.

Ensuite nous devons configurer Nginx, qui est notre serveur qui va gérer toutes les requêtes HTTP. La configuration est quasiment toujours la même, et disponible sur la documentation officielle de Symfony.

Le troisième et dernier fichier à créer est le fichier « docker-compose.yml », c'est le fichier central qui va orchestrer nos quatre services. Il définit comment ils sont construits et comment ils communiquent :

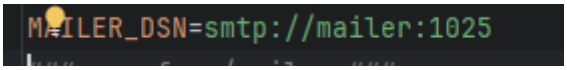


```yaml
services:

    php:
        build: .
        volumes:
            - .:/var/www/html
        depends_on:
            - database

    nginx:
        image: nginx:alpine
        ports:
            - "8080:80"
        volumes:
            - .:/var/www/html
            - ./docker/nginx.conf:/etc/nginx/conf.d/default.conf
        depends_on:
            - php

    database:
        image: mysql:8.0
        environment:
            MYSQL_ROOT_PASSWORD: root
            MYSQL_DATABASE: vite_gourmand
            MYSQL_USER: symfony
            MYSQL_PASSWORD: symfony
        ports:
            - "3307:3306"
        volumes:
            - db_data:/var/lib/mysql

    mailer:
        image: mailhog/mailhog
        ports:
            - "8025:8025"
            - "1025:1025"

volumes:
    db_data:
```

Les « volumes » jouent un rôle important, car ils vont déterminer la persistance des données, les ports exposent les services sur la machine locale, et enfin le « depends_on » va déterminer l'ordre de démarrage. Toute cette

configuration est disponible sur la documentation Docker. Elle est simplement à adapter à notre projet.

Enfin, il convient de mettre à jour notre .env.local :



Nous pouvons maintenant créer nos images grâce à la commande « docker compose -d --build », et nous voyons ainsi nos 4 conteneurs.

Un développeur qui reprend ce projet pourra ainsi, en clonant le dossier, et en faisant un « docker compose up -d –build » avoir à la fois le code source et tout son environnement (PHP, Nginx, MySQL et MailHog) fourni par les 4 conteneurs. Peu importe qu'il soit sur Windows, Mac ou Linux. Peu importe qu'il ait PHP ou non installé sur sa machine. Peu importe sa version de MySQL locale.

C'est exactement la promesse de Docker : « It works on my machine » devient « It works on every machine ».

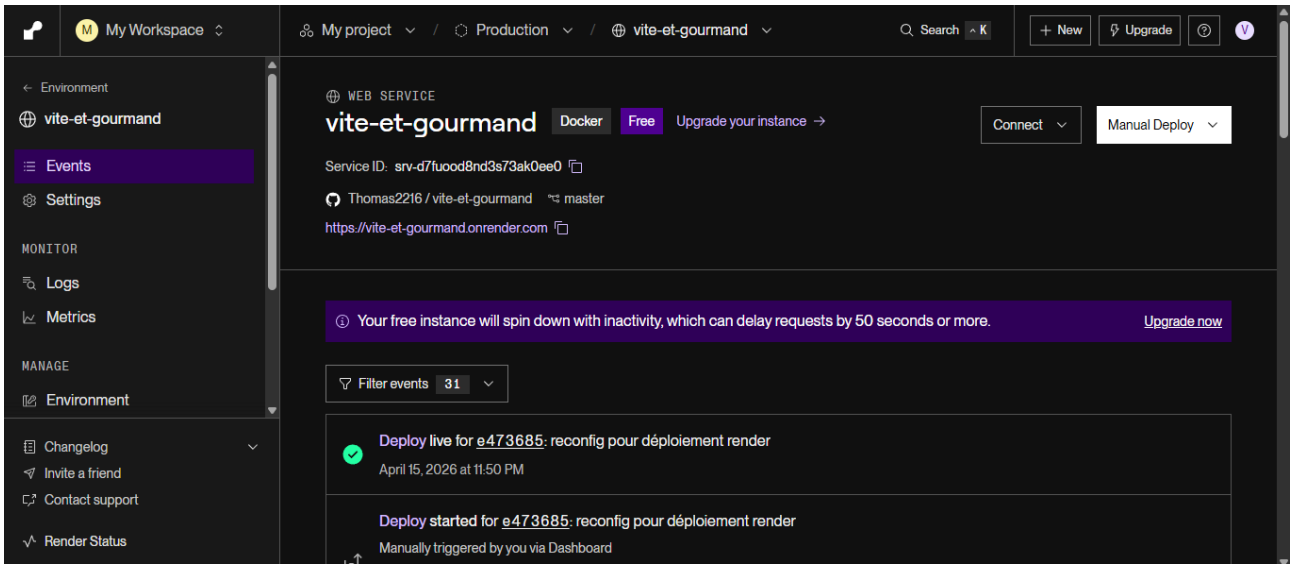
Mise en ligne

Une fois le développement de l'application réalisé, il convient de la rendre publique, c'est la mise en ligne (ou déploiement). C'est l'étape finale de notre projet, mais souvent pas la plus simple ! Il faut choisir une plateforme d'hébergement, configurer l'environnement de production, et s'assurer que l'application fonctionne correctement. Dans mon cas, cela a été relativement compliqué, ce que je vais vous détailler ici.

Pour le choix de la plateforme d'hébèrgement, il s'est initialement porté sur la plateforme Railway, qui me semblait être une bonne décision. Cependant, des incompatibilités techniques ont rendu ce choix impossible, notamment car l'extension PHP mongodb, indispensable au bon fonctionnement de l'application, ne pouvait pas être installée via Railway, ou tout du moins cela nécessitait des connaissances bien au delà de mes capacités. J'ai donc décidé de migrer vers Render, et ce choix s'est avéré payant, car la mise en ligne s'est effectué bien plus facilement.

Render est une plateforme cloud, qui gère très bien les déploiements via

Docker, ce qui offre un contrôle total sur l'environnement, ce qui nous permet d'installer les dépendances nécessaires au bon fonctionnement de notre application. Son interface est, comme vous le voyez, très simple :



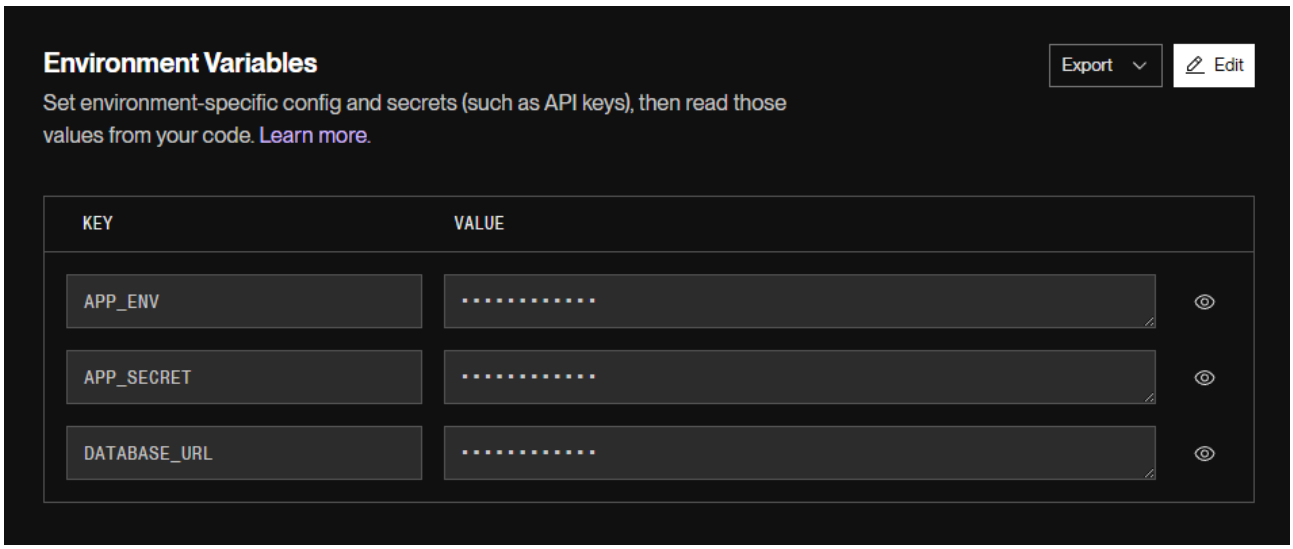
En production, avec Render, l'approche Docker est légèrement différente, car un seul conteneur suffit au déploiement. On a donc mis à jour notre Dockerfile en conséquence :

```
FROM php:8.2-fpm

RUN apt-get update && apt-get install -y \
    libicu-dev libssl-dev git unzip libpq-dev \
    && docker-php-ext-install pdo pdo_mysql pdo_pgsql opcache \
    && docker-php-ext-configure intl \
    && docker-php-ext-install intl \
    && pecl install mongodb \
    && docker-php-ext-enable mongodb \
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/*

COPY --from=composer:latest /usr/bin/composer /usr/bin/composer

WORKDIR /app
COPY . .

RUN composer install --ignore-platform-reqs --no-scripts --optimize-autoloader --no-interaction

RUN php bin/console importmap:install
RUN php bin/console asset-map:compile

EXPOSE 8080

CMD ["sh", "-c", "php bin/console doctrine:schema:update --force --complete && php -S 0.0.0.0:8080 -t
```

On retrouve l'image de base PHP, les extensions des différentes bases de données, la configuration des assets (importmap, asset-map). Une commande de mise à jour des tables de données, et du lancement du serveur PHP intégré est là pour compléter.

Nous avons du changer de service de base de données pour passer sur PostgreSQL proposé par la plateforme. Nous avons pour ce faire utilisé l'URL de connexion complète fournie par render. Et c'est pour cela que nous avons intégré la commande de création de table dans le CMD du Dockerfile, nous contournons ainsi le problème.

Il nous faut ensuite configurer les variables d'environnement. Nous les configurons directement sur Render :



Enfin, le déploiement automatique se fait via GitHub. Une fois la connexion avec notre dépôt faite, chaque push sera pris en compte par Render et un redéploiement sera fait. L'application est désormais accessible publiquement à l'adresse https://vite-et-gourmand.onrender.com .Il est à noter que le HTTPS est activé par défaut sur Render, ce qui garantit que toutes les communications entre le navigateur de l'utilisateur et le serveur sont chiffrées.

Le déploiement ne se déroule jamais de la bonne manière du premier coup. Il faut paramétrer bon nombre de fichiers, notamment le dockerfile, et l'adapter aux différents problèmes rencontrés. Sur la plupart des solutions de déploiement, il existe un système de log pour identifier ces problèmes. A l'aide des informations fournies par ces logs et la documentation Render, Docker et Symfony, cela nous permet d'arriver à notre objectif et réaliser le déploiement.

Note : *Suite à la suspension automatique de la base de données gratuite sur Render, j'ai migré l'application sur un VPS OVH sous Ubuntu 25.04, en reconfigurant manuellement Nginx comme serveur web, PHP-FPM pour l 'exécution du code, PostgreSQL et Mongo DB comme bases de données, et en sécurisant le déploiement avec un certificat SSL via Certbot. L'application est accessible via l'URL* https://vite-et-gourmand.thomas-valle.dev/ *.*

4 - Sécurité :

La sécurité est un aspect fondamental du développement d'une application Web. C'est une notion qui est vraiment transversale au projet et ne se limite pas à telle ou telle fonctionnalité. Symfony nous propose déjà, de façon native, beaucoup d'outils et de mécanismes de sécurité. Nous détaillerons ici différentes choses mises en place dans notre application pour répondre à ce besoin.

- Hash des mots de passe

Le stockage des mots de passe en base de données ne peut pas se faire en clair pour des raisons évidentes de sécurité. Symfony propose donc une fonctionnalité pour « hasher » les mots de passe, c'est à dire les remplacer par une série de caractères basée sur l'algorithme « bcrypt ». On retrouve sa configuration dans le « security.yaml » :

```yaml
security:
    # https://symfony.com/doc/current/security.html#registering-the-user-hashing-passwords
    password_hashers:
        Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'
```

Lors de la création d'un compte, le mot de passe d'un utilisateur est donc hashé avant d'être persisté en base de données. Ce hashage est irreversible. Il est impossible de retrouver le mot de passe original à partir du hash.

```php
if ($form->isSubmitted() && $form->isValid()) {
    $hashedPassword = $this->passwordHasher->hashPassword(
        $user,
        $user->getPassword()
    );
    $user->setPassword($hashedPassword);

    $em->persist($user);
    $em->flush();
```

– La protection CSRF

Une attaque CSRF (Cross Site Request Forgery) consiste à piéger un utilisateur connecté et exécuter une action à son insu, comme par exemple la soumission d'un formulaire. Symfony propose, de façon native, une protection des formulaires grâce à des « tokens » CSRF. Un « token » est unique et temporaire, il est associé à l'utilisateur, sa session, son adresse IP, etc. Si un token est absent ou invalide au moment d'une requête, celle-ci est refusée. Dans nos formulaires, cette protection est activée par défaut.

– La gestion des rôles utilisateurs

Notre application distingue trois niveaux d'accès : ROLE_USER (utilisateur standard), ROLE_EMPLOYE (employé de Vite et Gourmand), et ROLE_ADMIN (administrateur, accès complet aux fonctionnalités). Ces rôles sont hiérarchiques et héritent les uns des autres. Cette hiérarchie est définie dans le security.yaml :

```
role_hierarchy:
    ROLE_EMPLOYE: ROLE_USER
    ROLE_ADMIN: [ ROLE_EMPLOYE, ROLE_USER ]
```

Par défaut, un utilisateur qui crée un compte va hériter d'un ROLE_USER :

```
no usages  Thomas2216
public function getRoles(): array
{
    $roles = $this->roles;
    $roles[] = 'ROLE_USER';

    return array_unique($roles);
}
```

– Protection des routes avec IsGranted

La gestion des rôles ne suffit pas si les routes ne sont pas protégées. On attribue donc à nos routes un attribut PHP #[IsGranted] qui vérifie le rôle de l'utilisateur connecté avant d'exécuter la fonction du contrôleur. Si la condition

n'est pas remplie, il convient de gérer la chose via notre contrôleur.

```
#[AdminDashboard(routePath: '/admin', routeName: 'admin')]
class DashboardController extends AbstractDashboardController
```

– HTTPS en production

Le protocole HTTPS garantit que les échanges sont chiffrés entre l'utilisateur et le serveur. Il est mis en place automatiquement, dans notre cas, via Render (comme mentionné plus haut). Il ne nécessite donc pas de configuration complémentaire.

– Validation des données de formulaires

La validation des données des formulaires permet d'éviter l'injection de données malveillantes en définissant des contraintes, soit dans les entités, soit dans les formulaires. Nous avons principalement utilisé :

- NotBlank : vérifie qu'un champ n'est pas vide)
- Length : contrôle la longueur minimale et maximale d'une chaîne
- Email : vérifie que le format est bien celui d'une adresse email

```
->add( child: 'nom', type: TextType::class, [
    'constraints' => [
        new NotBlank(message: 'Le nom est obligatoire.'),
        new Length(max: 255, maxMessage: 'Le nom ne peut pas dépasser {{ limit }} caractèr
    ],
```

– La protection contre les failles XSS

Une faille XSS (Cross-Site Scripting) consiste à injecter du code malveillant dans une page Web, qui sera ensuite exécuté par le navigateur, comme du JavaScript par exemple. Cela se produit souvent sur les champs de formulaire.

Symfony protège nativement de ce type d'attaque grâce au moteur de template Twig qui va automatiquement échapper toutes les variables affichées. Concrètement, si une variable contient du code HTML ou JavaScript, celui ci sera affiché comme du texte et ne sera jamais exécuté.

Les contraintes symfony constituent une couche de protection supplémentaire protégeant des attaques XSS, comme ici :

```
->add( child: 'nom', type: TextType::class, [
    'constraints' => [
        new NotBlank(message: 'Le nom est obligatoire.'),
        new Length(max: 255, maxMessage: 'Le nom ne peut pas dépasser {{ limit }}
    ],
])
```

5 - Conclusion :

La réalisation du projet Vite & Gourmand m'a permis de mettre en pratique l'ensemble des compétences attendues dans le cadre de la formation « Développeur Web et Web Mobile ». De la conception initiale jusqu'à la mise en ligne, chaque étape a représenté un défi technique et pédagogique.

Sur le plan de la conception, la modélisation des données, les diagrammes, le maquettage, m'ont appris à structurer un projet avant d'écrire la moindre ligne de code. Cette démarche est indispensable en milieu professionnel.

Sur le plan technique, le développement avec Symfony m'a permis de maîtriser les fondamentaux du framework : le modèle MVC, l'ORM Doctrine, le moteur de template Twig, la gestion des formulaires, et la sécurité. La mise en place du back-office avec EasyAdmin, les requêtes SQL, les requêtes AJAX, et l'intégration de MongoDB pour les statistiques illustrent la diversité des compétences mobilisées.

La mise en ligne a également été une expérience enrichissante. Les difficultés rencontrées, notamment avec les problèmes de compatibilité, m'ont conduit à approfondir ma compréhension des environnements Docker et les spécificités de chaque plateforme d'hébergement.

Ce projet m'a également sensibilisé aux bonnes pratiques, notamment le versionnage avec Git et la sécurité. Cette mise en situation est précieuse pour aborder le marché du travail.

En conclusion, le projet Vite & Gourmand m'a permis de couvrir l'ensemble des activités types du référentiel.

Annexes

Logo Vite & Gourmand :

Charte graphique :

Charte Graphique – Vite & Gourmand

Cette charte graphique définit l'identité visuelle de l'application web « Vite & Gourmand », afin d'assurer une communication cohérente, élégante et reconnaissable.

1. Présentation de l'entreprise

Vite & Gourmand est une entreprise bordelaise fondée il y a 25 ans par Julie et José. Elle propose des prestations culinaires pour tous types d'événements, avec des menus évolutifs, gourmands et accessibles.

2. Valeurs et positionnement

- Convivialité
- Qualité artisanale
- Simplicité et accessibilité
- Modernité au service de la tradition

3. Palette de couleurs

Couleur	Usage	Code HEX
Violet principal	Identité, titres, éléments clés	#5B2D8B
Violet clair	Arrière-plans secondaires	#9B7BC2
Noir	Texte principal	#000000
Gris foncé	Textes secondaires	#3A3A3A
Blanc	Fond principal	#FFFFFF

4. Typographies

Titres : Police sans-serif moderne (ex : Montserrat, Poppins)
Textes courants : Police lisible et sobre (ex : Open Sans, Roboto)
Règles : Hiérarchisation claire, tailles et contrastes respectés.

5. Iconographie et visuels

- Icônes simples et modernes
- Photographies de plats authentiques et appétissantes
- Privilégier des images lumineuses avec une touche chaleureuse

6. Ton et style rédactionnel

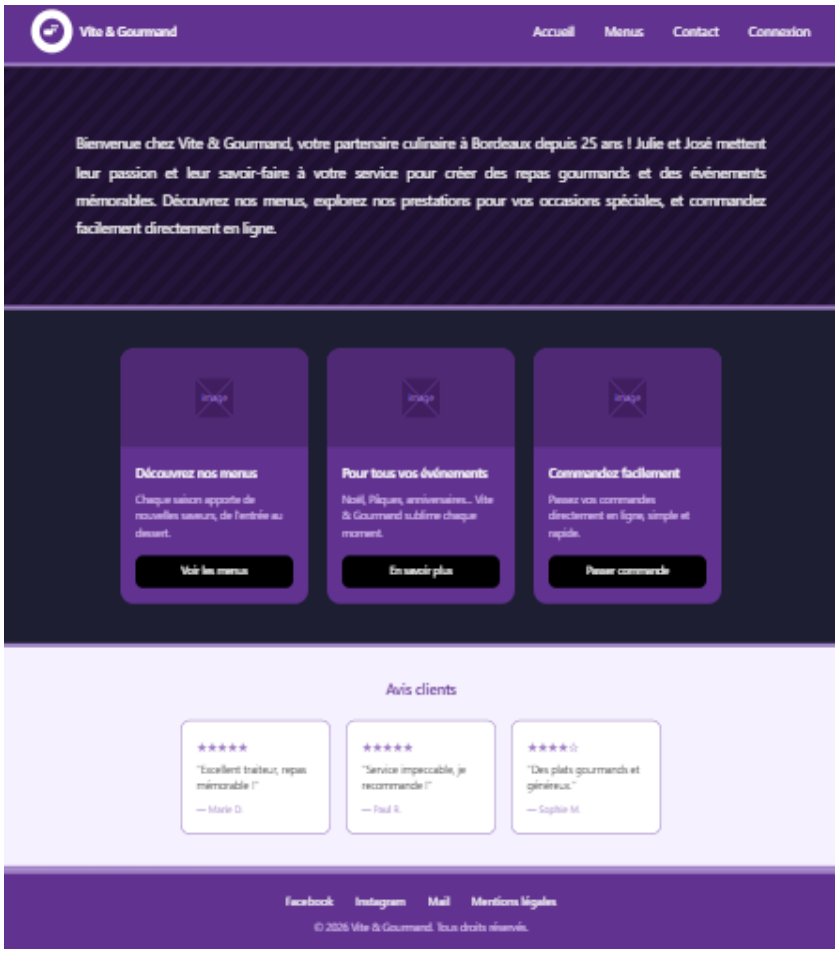
Le ton doit être chaleureux, convivial et accessible. Le vocabulaire reste simple, gourmand et positif, afin de refléter l'esprit familial de Vite & Gourmand.

7. Utilisation dans l'application web

- Menus clairs et lisibles
- Boutons violets pour les actions principales
- Fond clair avec accents violets et noirs pour la lisibilité

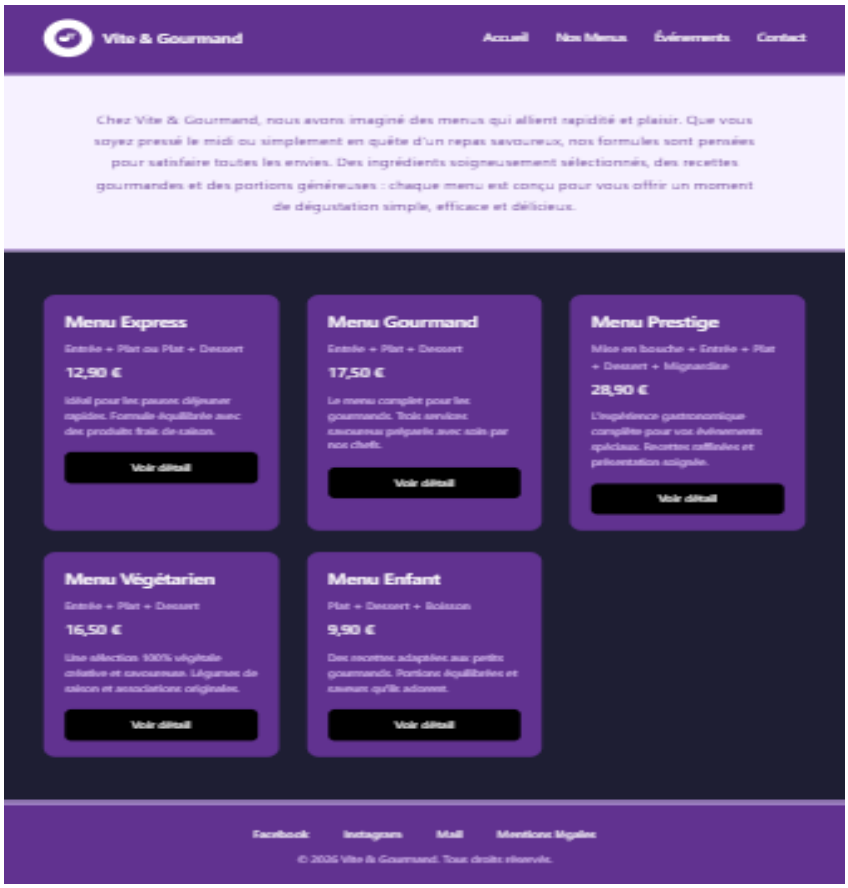
Wireframes Page Accueil :

Mockups Page Accueil :



Wireframes Page Menus :

Mockups Page Menus :



Wireframes Page Commande :

Mockups Page Commande :