|  |
| --- |
| Studi |
| Dossier de Projet |
| Site web garage |

|  |
| --- |
| BOGDANOVIC HUGO  03/04/2024 |

# Sommaire

[Sommaire 1](#_Toc163036685)

[Introduction 2](#_Toc163036686)

[1. Compétences couvertes par le projet 2](#_Toc163036687)

[a. Développer la partie front-end d’une application web ou web mobile en intégrant les recommandations de sécurité 2](#_Toc163036688)

[b. Développer la partie back-end d'une application web ou web mobile en intégrant les recommandations de sécurité 2](#_Toc163036689)

[2. Résumé du projet 3](#_Toc163036690)

[Projet ECF 4](#_Toc163036691)

[1. Cahier des charges 4](#_Toc163036692)

[2. Spécifications techniques 5](#_Toc163036693)

[Réalisation du projet 6](#_Toc163036694)

[1. Conception de la base de données 6](#_Toc163036695)

[2. Développement du Back-end 7](#_Toc163036696)

[a. Initialisation du projet 7](#_Toc163036697)

[b. Création des Model 9](#_Toc163036698)

[c. Définition des routes 12](#_Toc163036699)

[d. La gestion de contenu 14](#_Toc163036700)

# Introduction

## Compétences couvertes par le projet

### Développer la partie front-end d’une application web ou web mobile en intégrant les recommandations de sécurité

* Maquetter une application.
* Réaliser une interface utilisateur web statique et adaptable.
* Développer une interface utilisateur web dynamique.
* Réaliser une interface utilisateur avec une solution de gestion de contenu ou e-commerce.

L’application doit être accessible par des ordinateurs et des téléphones par l’interface web de manière adaptable. J’ai fait le choix d’utiliser bootsrap comme framework css.

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

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

Le front devant communiqué avec une base de donnée le choix du framework symfony (PHP), leader du marché actuel, m’a permis une abstraction grâce à son puissant service d’ORM et de migration grâce à l‘intégration de Doctrine nativement.

J’ai également décidé d’utiliser le SGBD (Système de Gestion de Base de Donnée) PostgreSQL, lui aussi leader du marché, pour son adoption massive favorisant le développement de correction de bug et une communauté active.

## Résumé du projet

Mon projet est l’ECF proposé avec pour sujet une commande d’un site pour un garage.

La commande était un site permettant à un particulier de voir les différents services proposés par le garage (Réparation de véhicule, Vente de véhicule, Estimation…) ces services doivent être modifiable par un administrateur (le gérant du garage) et il doit pouvoir les ajouter ou les supprimer.

Le site doit également proposer une présentation des différentes voitures d’occasion disponible à l'achat (sans proposer de moyen de paiement en ligne). Comme pour les services, les voitures doivent être modifiables, ajustables ou supprimables pour un administrateur.

Le contact client étant au centre de leurs intérêts, le garage a également demandé un système d’avis intégré au site sous forme de note/commentaire.

Les visiteurs peuvent envoyer un mail au garage demandant une information sur une voiture, un service ou autre en passant par un formulaire de contact.

En parallèle, l'administrateur doit pouvoir ajouter un nouvel employé en lui créant un compte.

Un employé peut saisir des témoignages.

Le client impose l’utilisation de certaines technologies: Une SGBD SQL et le site en PHP avec le framework symfony (7.06).

# Projet ECF

## Cahier des charges

Le client, Vincent Parrot, nous a contacter pour développer son premier site web en lien avec son garage.

L’application est hébergée sur mon serveur privé, front comme back.

Travaillant seul j’ai décidé d’utiliser un manageur de version comme Git plutôt que de m’imposer un développement en Agile qui aurait été lourd et sans grand intérêt dans ma situation.

A son arrivé le visiteur doit avoir une vue concise du garage, avec une brève description ainsi que quelques un des services proposés. Un visiteur peut contacter le garage par mail, soit en passant par la page de contact soir en cliquant sur boutons « En savoir plus » présent sur certains éléments (comme les services par exemple).

Un visiteur doit pouvoir visionner les différentes voitures proposées à la vente et doit pouvoir les trier en fonction du prix, du kilométrage et/ou de l’année de mise en circulation du véhicule, cela sans avoir à rechargé la page.

Un employé du garage doit également avoir un compte, lui permettant de se connecter pour avoir une gestion, réduite, sur le site. Un employé doit pouvoir ajouter, modifier et supprimer des voitures quand elles arrivent à la vente ou lorsqu’elles sont vendues. Il doit également pouvoir ajouter des témoignages de client.

Il y a, au moins, un administrateur pour le site, M.Parrot et potentiellement un, ou plusieurs, webmaster. Ces derniers doivent pouvoir ajouter, modifier et supprimer des services et des voitures, ainsi que des témoignages. L’administrateur a également la main sur les horaires d’ouverture du garage (horaire semaine, samedi et dimanche)

## Spécifications techniques

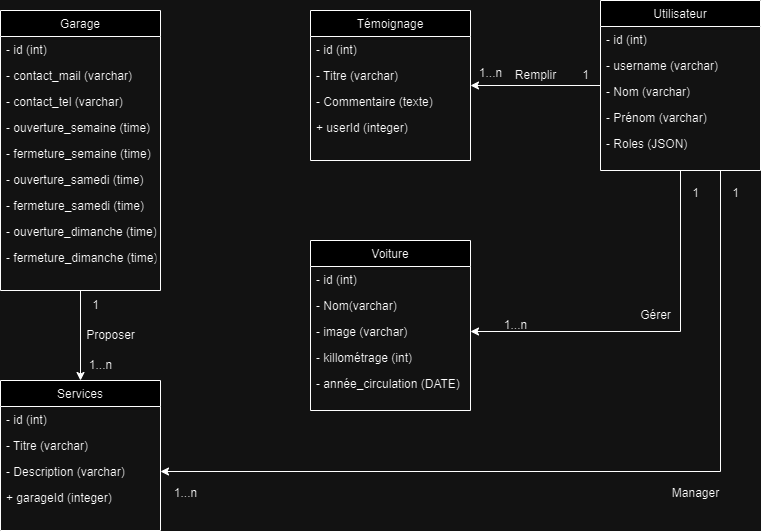
* Environnement de travail :
  + Utilisation du framework Symfony utilisant php 8.4 (ou supérieur) comme base permettant un développement full-stack avec son moteur html Twig intégré.
  + Outil Git pour la gestion de versions avec la plateforme GitHub.
  + DBVisualizer pour pouvoir requêter et avoir une vision sur la base de donnée sans avoir à passer par l’invité de commande Postgres.
* Partie Front :
  + Twig, un moteur HTML permettant la création de template et l’insertion de variable via le renderer de Symfony.
  + Occasionnellement du JavaScript et du JQuerry.
* Partie Back :
  + Symfony, framework php leader depuis le retour en force de php (avec php 7) il permet le développement d’application via le principe MVT (Model – View – Template). Un Model représente une donnée dans la base, une View représente les interactions avec ce model (ajout, modification, suppression, récupération) et enfin le Template est l’interface IHM, ici la page web.
  + Doctrine, ORM (Objet Relational Mapping) pour avoir une abstraction avec la base de donnée.
  + Postgres, SGBD (Système de Gestion de Base de Donnée) comme serveur SQL (Structured Querry Language) pour le stockage des données.

# Réalisation du projet

### Conception de la base de données

En utilisant les informations données par M.Parrot j’ai listé toutes les potentielles entités ainsi que leur relations les unes aux autres. Un visiteur par exemple n’a pas de compte, il n’est donc pas stocké dans la base. Un utilisateur (employé) lui est stocké dans une table, il interagit avec les voitures et les avis ect…

J’ai donc réaliser un MCD, donnant les entités ainsi que leurs relations et une cardinalité a ces relations.



Une fois le MCD prêt j’ai commencé la création du back-end

## Développement du Back-end

### Initialisation du projet

Pour commencer j’ai dû créer le projet Symfony et généré son arborescence.

J’ai utilisé l’application Symfony CLI, un outil développer par l’équipe de Symfony pour rendre l’initialisation des projets plus simple et donne également des outils aux développeurs.

D’abord cette commande pour s’assurer que j’avais les dépendances requises :

$ symfony check:requirements

Ensuite si tout va bien nous pouvons initialiser le projet :

$ symfony new ECF-Garage --version="7.0.\*" –webapp

Ici j’initialise donc un nouveau projet symfony, nommé ECG-Garage utilisant la version 7.0.\* (il prendra la dernière version 7.0 sortie) avec le flag webapp indiquant que c’est une application full stack, important automatiquement des package comme Twig. Sans se flag nous aurions le squelette d’une API.

M.Parrot voulant également que les voitures aient des photos de présentation j’ajoute 2 autres packages non-inclus de base :

* VichUploader
* LiipImagine

Ces deux packages permettent de grandement faciliter l’upload, le stockage et la configuration des uploads. Pour les ajouter au projet :

$ composer require vich/uploader-bundle

$ composer require liip/imagine-bundle

Pour vich il faut ensuite aller dans le fichier de configuration pour ajouter une clause dans app/config/config.yml :

vich\_uploader:

db\_driver: orm

Cela permet au package de savoir qu’elle base de données l’application utilise, et la valeur ORM lui indique de passer par Doctrine.

Une fois cela prêt il faut configurer le .env à la racine du projet.

### Création des Model

Pour fonctionner Symfony se base sur les entités, elles représentent nos données. On peut voir une entité comme une ligne dans une table de notre BDD.

Je viens donc créer chaque entité :

* Utilisateur
* Voiture
* Garage
* Service
* Temoignage

Tout cela en utilisant symfony CLI (ou artisan si l’outil n’est pas installé)

$ symfony console make:entity

Grace au MakerBundle, inclus dans notre configuration de base, symfony est capable de générer via l’invité de commande nos entités et nos contrôleurs.



On vient définir tous nos champs avec des décorateurs Doctrine permettant de définir les colonnes, leurs types, leurs attributs (taille d’un champ texte) ect…

Notre Utilisateur hérite de UserInterface et de PasswordAuthenticator, imposant l’implémentation de certaines fonctions

Les différents imports :

Le repository matérialise le lien avec la base

Doctrine est l’ORM

Et Security est présent car c’est une class User, c’est grâce à ça que l’on peut hasher le mot de passe

Je fais la même chose pour les autres entités avec leurs champs.

Une fois toutes les entités prêtes je peux lancer la commande :

$ symfony console make:migration

Me permettant de générer un fichier de migration au format php

Doctrine va automatiquement générer du PostgreSQL pour alimenter notre base.

Les séquences sont là pour identifier les id auto-incrémentés, les CREATE TABLE créer nos tables en fonctions des champs dans les entités, les CREATE INDEX permettent de faire les relations entre les tables.

Enfin quand le fichier de migration est juste on peut lancer la commande symfony pour exécuter la migration

$ symfony console doctrine:migrations:migrate

Cela va appliquer la migration générée à notre base PostgreSQL.

### Définition des routes

Dans symfony les routes sont définis directement dans les controllers.

Le routeur de Symfony fonctionne avec des décorateurs comme Doctrine.

On ajoute donc le decorateur :

#[Route(‘lien/de/la/route’, name : ‘nom\_de\_la\_route’)]

Grace à ça le routeur va attribuer le lien à la fonction automatiquement. La valeur de ‘name’ sert en interne pour appeler la route sans avoir a taper le lien. Par exemple on pourra dans une page appeler la fonction twig path() en passant en paramètre le nom de la route, Twig va de son coter interroger le routeur Symfony pour générer le bon lien.



Dans cette exemple je prends la fonction d’enregistrement pour mes voitures.

On remarque le décorateur #[Route] avec le lien /voiture/register et le nom app\_voiture\_register.

Donc sur mon site, quand j’irais sur le lien /voiture/register j’appellerais la fonction register de la classe voiture.

### La gestion de contenu

Pour la gestion de contenu je me suis inspiré du paradigme CRUD (Create, Read, Update, Delete). Tous mes contrôleurs on les fonctions suivantes :

* Register (ou Ajouter) : Permettant de persister une donnée rentrée via l’interface utilisateur
* Update (ou Modifier) : Permettant de mettre à jour un enregistrement spécifique
* Delete (ou Supprimer) : Permettant de supprimer de la base un enregistrement

En utilisant ces fonctions et une stratégie de rôle pour les utilisateurs chacun peut gérer le contenu des pages.

Exemples pour les services :



La fonction AjouterService, prenant en paramètres l’EntityManagerInterface et la une Request. L’EntityManageInterface sert à avoir accès a la base de donnée, que ce soit pour retrouver le garage :

$garage = $entityManager->getRepository(Garage::class)->findOneBy(['id' => '1']);

Ou pour ecrire dans la base:

$entityManager->persist($service);

Ce que fais la fonction est « simple », d’abord je crée un nouvel objet Service vide. Je récupère le garage pour pouvoir lier le service avec ce dernier.

Je crée un formulaire (avec la fonction createForm)

Si le formulaire est rempli et renvoyé correctement (soumis et valide) alors on reprend notre objet vide Service et on lui assigne ses champs, le garage avec setGarage en utilisant le garage créer juste avant, le nom et la description en reprenant les infos du formulaire.

Une fois l’objet complet on l’écrit dans la base avec persist() et on flush() l’EntityManager pour pouvoir le réutiliser.

Enfin je redirige vers la page d’accueil des services avec redirectToRoute.

Si le formulaire n’a pas été renvoyé cependant je créer la vue avec le formulaire, c’est également notre état de base permettant d’avoir toujours le formulaire d’afficher la première fois.

La fonction updateService est très similaire, en effet je fais la même chose que pour la ajouterService à la différence près que cette fois ma variable $service n’est pas créer vide mais via la fonction findOneBy() permettant de trouver dans la base le service souhaiter en fonction de son ID. Ensuite la logique est la même que pour créer. La fonction createForm va automatiquement alimenter les champs du service avec les données déjà existantes.

Pour deleteService c’est plus concis. Je récupère le service que je veux supprimer, j’affiche tout de même le service à supprimer via un formulaire et si l’utilisateur confirme alors on supprime l’enregistrement de la base.

Cette logique est répliquée dans les Contrôleurs pour les voitures et les témoignages. Je ne l’ai néanmoins pas appliquée pour les horaires (n’ayant implémenter que le update) et pour le garage en lui-même qui n’a aucune de ses fonctions. La logique étant que ce le garage par exemple ne change pas et ne peut pas être supprimé et que les horaires ne peuvent-être que modifier.

### Prise de contact

Le client a demandé à ce que soit implémenter un moyen pour les visiteurs de prendre contact avec le garage. J’ai donc créer une fonction contact dans mon contrôleur du garage.

J’ai pris la décision de gérer l’auto-complétion du message de contact en back. Donc quand le visiteur clique sur un bouton « en savoir plus » situé par exemple sur les voitures il est redirigé automatiquement vers la page de contact avec un message prédéfinis indiquant le message (J’aimerais plus d’info sur la voiture xxx). Ainsi le visiteur n’a plus qu’a indiqué son mail et appuyé sur « Envoyer ». Il peut également faire le choix de remplir le message comme il le souhaite (sans avoir la possibilité de changer le mail du garage.

Pour l’envoi de mail j’utilise une combinaison du mailer de Symfony et d’un client SMTP le serveur hébergeant le site. Mais le service de mail peut également être pris en charge via un client mail tier.