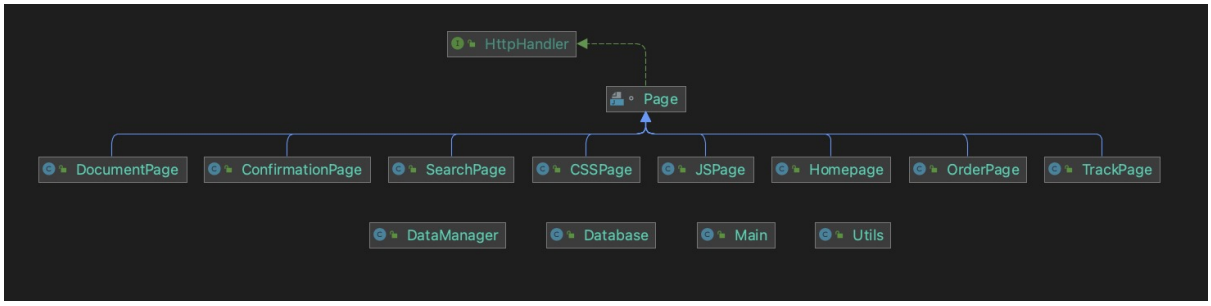
Documentation développeur

Diagramme de classes



Architecture en Package

Nous avons fait le choix de dissocier deux packages :

- `pages` → L'ensemble des classes héritant de la classe `Page`, ainsi que cette dernière. Chaque classe contient une ou plusieurs méthodes afférentes à ses besoins, en terme de HTML & bases de données.
- `tools` → Un package qui comprend :
  - `Database` : une classe personnalisée pour gérer simplement les interactions avec la base de données

- `DataManager` : une classe permettant d'instancier et utiliser Database, ainsi que de transmettre l'unique instanciation de Database vers d'autres classes. Elle permet également d'effectuer d'autres actions, telles qu'insérer du HTML ou exécuter une action de discount automatique à chaque lancement.
- `Utils` : Une classe contenant un ensemble de méthodes statiques utiles aux autres méthodes, comme déterminer si le projet est lancé depuis un fichier Jar, ou lire un fichier JSON.

Création du serveur web

Notre projet est basé sur un serveur Http, utilisant pour cela la classe Java `HttpServer`, qui permet de créer un serveur web de façon simplifiée. Pour cela, on utilise dans la méthode `main` :

```
// Création du serveur
HttpServer server = HttpServer.create(new InetSocketAddress(port), 0);

// Création d'une route (page disponible sur le navigateur)
server.createContext("/route", new Page());

// Démarrage du serveur
server.start();
```

Chaque route est matérialisée par une classe héritant de `Page`, une classe personnalisée implémentant l'interface `HttpHandler`, de `HttpServer`.

Classe `Page`

Chaque classe Page implémente `HttpHandler`. L'objectif de cette classe est de renvoyer du code HTML lorsque la route qui l'appelle, définie dans `Main`, est appelée par le navigateur client. pour cela, elle présente plusieurs attributs :

Attribut ou méthode	Explication
`private String html`	Le code html de l'instance de la classe, qui sera renvoyé vers la page.
`public Page()`	Le constructeur par défaut de la classe. Il initialise le html en faisant appel à un fichier de l'arborescence `base_html.html`.
`public Page(String ... html)`	Un constructeur permettant d'ajouter au html : • Le code du fichier `base_html.html`. • Au sein de la balise `<body>`, chaque code html sous la forme de String passé en paramètre.
`protected String getHtml()`	Un getter qui renvoie le code html contenu dans la variable du même nom.

Attribut ou méthode	Explication
`protected static String insertHTML(String html, String body)`	Une méthode permettant d'insérer dans un code html déjà établi du nouveau code html, entre les balises `<body></body>`.
`protected static Map<String,String> getUrlParams()`	Une méthode permettant d'obtenir sous la forme d'une `Map` les paramètres GET de l'URL de la requête, lors de l'ouverture de la page par le client. Le code original provient de StackOverflow.
`@Override public void handle(HttpExchange exchange) throws IOException`	Une redéfinition de la méthode `handle` de l'interface `HttpHandler`. Elle envoie les headers de la réponse, envoie les octets dans un `OutputStream` et ferme le flux.

Fichier `base_html.html`

Il s'agit d'un fichier html auquel la classe `Page` accède via la méthode `Utils.` *`getFileAsString(String path)`*. Il présente plusieurs éléments :

- Un code html standard avec une `<head>` et plusieurs balises `<meta>`, ainsi qu'un `<body>` vide.
- Un appel aux pages de style (CSS) et de script (JavaScript) :

```
<link rel="stylesheet" href="style.css">
<script defer src="script.js" type="text/javascript" async="false"></script>
```

Classes héritant de `Page`

Chaque classe héritant de la classe Page va, au moins faire appel à `super()` dans son propre constructeur pour avoir un code html correspondant à la base commune, et :

- Va proposer une ou plusieurs méthode(s) qui lui sont propres pour rendre les informations que la page doit fournir (par exemple, une méthode faisant appel à la base de données pour afficher les voitures disponibles en html, dans la page d'accueil)
- Va redéfinir la méthode `handle` si besoin est pour gérer les paramètres GET de l'URL.

Cas des classes `CSSPage` & `JSPage`

Les classes CSSPage et JSPage sont particulières car elles servent uniquement à rendre les fichiers `style.css` et `script.js`. Elles sont utilisées dans chaque page via les balises html vues plus haut.

Ainsi, elles vont simplement être constituées d'un appel à `super()` prenant en entrée le fichier de style ou de script.

Elles ne proposent aucune méthode qui leur est propre ni ne redéfinissent `handle`.

Cas des autres classes

En accord avec la disposition des routes dans la classe `Main` via `HttpServer` comme vu précédemment, les classes héritant de `Page` sont les suivantes :

Classe	Route	Méthodes	Redéfinition de `handle`
CSSPage	/style.css		Non
JSPage	/script.js		Non
HomePage	/	`protected static String showAvailableVehicles(String search)`	Non
SearchPage	/search		Oui
OrderPage	/order	`private static String showPurchaseForm(int vehicleId)`	Oui
ConfirmationPage	/confirmation	`private static Map<String, Object> parseOrderParams(Map<String,String> params)` `private static void insertOrderInDb(Map<String,String> params) throws Exception` `private static String showOrderConfirmation(Map<String,String> params)`	Oui
TrackPage	/track	`private static String showTrackPage(String email)`	Oui
DocumentPage	/sale_certificate, /registration_request, /order_sheet	`public static void createDocumentsPagesContexts(HttpServer server)` `private String showDocumentPage(int orderId) public DocumentPage(String documentType) throws InvalidAttributeValueException` *	Oui

📌 Il est à noter que la méthode `handle` est redéfinie si et seulement si les paramètres GET de la requête sont exploités.

📌 L'ensemble des méthodes ci-dessus sont *private* car elles ne sont utilisées que dans leur propre classe, à l'exception de :

- `showAvailableVehicles`, en *protected* car utilisée dans *`Homepage`* & *`SearchPage`*.
- `createDocumentsPagesContexts`, en *public* car appelé dans *`Main`* pour créer les endpoints correspondants *(voir tableau ci-dessous)*

📌 Certaines pages proposant des méthodes, ont un accès à la base de données, que nous évoquerons plus bas dans ce document. Ce dernier prends toujours la forme suivante :

```
private static final Database db = DataManager.getDb();
```

Le fonctionnement des méthodes ci-dessus est détaillé dans le tableau ci-dessous :

Méthode	Classe	Explication
`showAvailableVehicles`	Homepage	Récupère l'ensemble des données des véhicules de la base de données. Si un String de recherche est précisé, seul les données contenant la recherche sont récupérées. Si la réponse de la base de données est vide, renvoie un titre précisant au client qu'aucun véhicule n'est disponible. Sinon, crée un code HTML dans une boucle contenant, pour chaque modèle de véhicule, les informations correspondant et un lien d'achat.
`showPurchaseForm`	OrderPage	Récupère depuis la base de données les informations relatives au modèle dont l'ID est passé en paramètre GET. Présente au client ces informations via une liste en HTML. Enfin, récupère le formulaire d'achat contenu dans le fichier `order_form.html`, et y ajoute les informations du véhicule à acheter (par exemple, en bornant le champ de sélection de nombre de véhicules au maximum du stock). Ce même formulaire permet de confirmer l'achat.
`parseOrderParams`	ConfirmationPage	Crée une Map dans laquelle sont ajoutés les éléments des paramètres GET issus du formulaire d'achat, dont la syntaxe a été corrigée. Par exemple, une adresse e-mail est transmise sous la forme *prenom%40serv.com*, au lieu de *prenom@serv.com*. Cette fonction corrige ça. Certaines informations, telles que les taxes, sont récupérées depuis la base de données et ajoutées à la Map. Enfin, certaines informations telles que le prix du véhicule après la prise en compte d'un crédit sont calculées au sein de cette fonction.

Méthode	Classe	Explication
`insertOrderInDb`	ConfirmationPage	Utilise `parseOrderParams` pour insérer la commande et le nouvel utilisateur associé le cas échéant, dans la base de données. La fonction vérifie la validité de la carte bancaire et renvoie une erreur si celle-ci n'est pas valide. Elle vérifie ensuite que le nombre de véhicule commandés n'est pas supérieur au stock disponible (et renvoie une erreur en cas de besoin). La fonction vérifie également si un utilisateur associé à l'email donné existe, et si tel est le cas associe la commande avec cet utilisateur. Dans le cas contraire, elle en crée un nouveau. Enfin, la fonction exécute deux requêtes SQL : - Une première pour insérer la commande dans la base de données grâce aux paramètres de `parseOrderParams`. - Une seconde pour décrémenter le stock de véhicules du modèle donnée du nombre de véhicules commandés.
`showOrderConfirmation`	ConfirmationPage	Fonction appelée lors de l'accès à la page de confirmation, via le formulaire d'achat, elle invoque `insertOrderInDb` via un `try...catch` et informe l'utilisateur via HTML si une erreur est survenue. Dans le cas contraire, après avoir inséré la commande, elle présente (avec une anonymisation pour la carte bancaire) les données de la commande à l'acheteur, et lui propose plusieurs liens pour suivre sa commande et obtenir ses documents.
`showTrackPage`	TrackPage	Via un email passé en paramètre, la fonction récupère depuis la base de données l'ensemble des commandes associées au client lui même associé à l'email donné en paramètre. Les informations de chaque commande sont ensuite affichées via un code html, issu d'un template dans un fichier .html.
`createDocumentsPagesContexts`	DocumentPage	Méthode appelée dans la classe `Main`, au lancement du programme. Crée trois endpoints via le serveur passé en paramètre (instancié dans `Main`). Chaque endpoint correspond à un document spécifique rendu disponible au client (sale_certificate, registration_request, order_sheet). Une variable présentant les endpoints autorisés est créée dans la classe : `protected final static String[] allowedDocumentTypes = {"sale_certificate", "registration_request", "order_sheet"}` La méthode vérifie que les endpoints créés correspondent à ces derniers pour éviter toute confusion dans le code.

Méthode	Classe	Explication
`showDocumentPage`	DocumentPage	Envoie une requête à la base de données pour récupérer l'ensemble des informations pour la commande dont l'id `orderId` est passé en paramètre, via une clause sql `where`. L'ensemble des informations, via une double jointure, est inclus, dont les informations de commande, sur le client et le véhicule acheté. En cas d'absence de résultat, informe l'utilisateur via un `try…catch`. Pour chaque véhicule (via une boucle `for` itérant sur la quantité de véhicule commandés), crée un `fieldset` html englobant le contenu du fichier html du document demandé. Il y a autant de documents que de véhicules commandés. Retourne le code html ainsi créé.
`DocumentPage` *(constructeur)**	DocumentPage	Constructeur prenant en entrée un type de document inclus dans `allowedDocumentTypes`. Appelle `super()`, puis vérifie que le paramètre documentType est bien inclus dans `allowedDocumentTypes`. Dans le cas inverse, renvoie une erreur. Attribue enfin cette valeur à une constante `private`, utilisée dans `showDocumentPage` afin de générer (via le fichier html correspondant) le document demandé.

📌 * Seul le constructeur de `DocumentPage` est ici détaillé car étant plus spécifique que les autres constructeurs des autres classes.

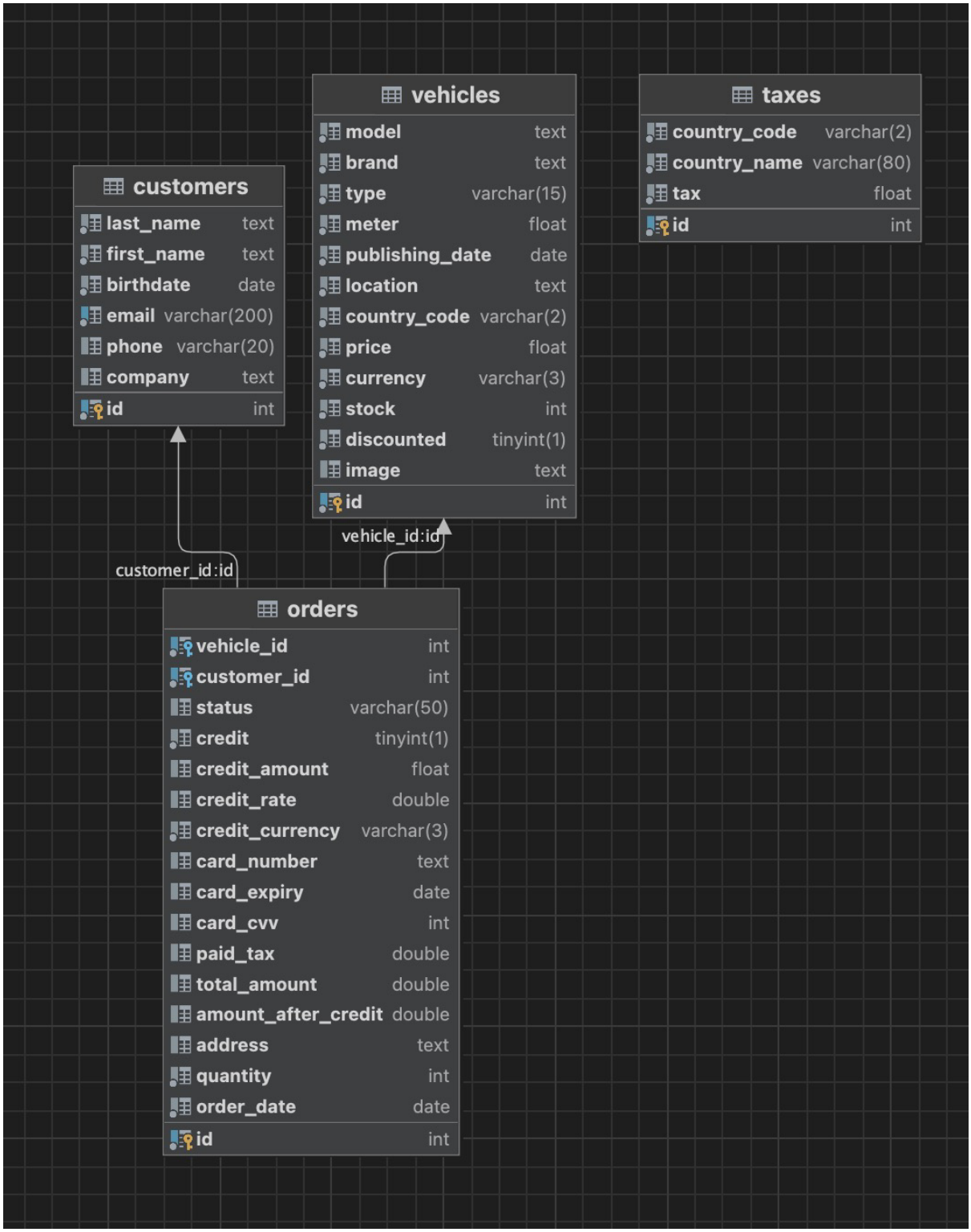
Il vérifie en effet l'intégrité du paramètre `documentType` en entrée.

Cela est nécessaire pour s'assurer que l'appel aux fichiers html effectués dans `showDocumentPage`, ainsi que la consistance des endpoints (créés via `createDocumentsPagesContexts`) utilisés dans d'autres classes, soient respectés.

Gestion des données

Base de données MySQL

Notre base de données est modélisée par le diagramme de classe suivant :



La table **orders** concentre l'ensemble des informations de la commande, incluant des informations calculées en Java via `parseOrderParams` et `insertOrderInDb`. Elle est liée à la table **customers**, qui contient les informations de chaque utilisateur. Une contrainte d'unicité sur l'email est appliquée, étant donné que nous l'utilisons notamment dans Track pour identifier un client. Elle est également liée à la table **vehicles**, contenant le type de véhicules (contrainte de valeur "scooter" ou "car"), avec des informations telles que la localisation, le

code du pays (pour le calcul de taxes), le stock, si le modèle est en promotion ou non, ou encore la date de publication.

Enfin, la table **taxes** est ici pour le calcul des taxes, et est utilisée directement dans le code Java sans liaison dans la base de données. Les taxes sont choisies aléatoirement, et non en lien avec une législation.

Classe `Database`

La classe Database a pour but de simplifier les appels SQL à la base de données en une seule méthode query, qui prendrait un unique String en entrée représentant la requête à exécuter. Ses attributs et méthodes sont le suivants :

Attribut	Explication
`private final String protocol ; private final String host ; private final int port ; private final String db_name ; private final String username ; private final String password ;`	Attributs nécessaires à la connexion à la base de données. Ils sont private et final car ils ne sont utiles que pour stocker les données utilisées dans `query()`.
`public Database (String protocol, String host, int port, String db_name, String username, String password)`	Constructeur unique de la classe prenant en entrée une valeur par constante de classe et attribuant la valeur à chaque constante passée en paramètre.
`public ArrayList<HashMap<String, String>> queryWithException (String sql) throws Exception`	Définie le driver de connexion, instancie la connexion grâce à `DriverManager.getConnection()`, via les valeurs des constantes de classe, ainsi qu'un `Statement` de requête. Si la requête passée en paramètre contient à son début "insert", "update" ou "delete", alors `stmt.executeUpdate()` est exécuté et la méthode renvoie une `ArrayList` vide. Sinon, la méthode exécute la méthode via `stmt.executeQuery(sql)`, et récupère les objets `ResultSet` (contient les données) et `ResultSetMetaData` (contient les noms des colonnes, et d'autres données connexes) correspondants. Enfin, retourne un appel à `rsToDict` avec ces deux valeurs.)
`public ArrayList<HashMap<String, String>> query (String sql)`	Retourne `queryWithException` via un `try…catch`, avec en cas d'erreur une impression de l'erreur et le retour d'une liste vide. Il s'agit de la méthode la plus utilisée (`queryWithException` étant plus utilisé dans les cas où on a besoin d'effectuer un `try…catch` plus spécifique)

Attribut	Explication
`private ArrayList<HashMap<String, String>> rsToDict (ResultSet rs, ResultSetMetaData metaData)`	Instancie une ArrayList vide, et itère dans le `ResultSet` via `rs.next()`. Pour chaque valeur, instancie une `HashMap` et via les métadonnées, parcours les colonnes et associe pour chaque colonne son nom et la valeur associée. À la fin de la boucle, ajoute la `HashMap` à la `ArrayList` et la retourne.

Classe `DataManager`

L'objectif de cette classe est de centraliser l'ensemble des moyens d'interaction avec :

- La base de données
- Le fichier `config.json`

Il s'agit d'une classe ne contenant que des méthodes statiques, n'ayant pas vocation à être instanciée.

Elle présente ainsi les attributs et méthode suivants :

Attribut	Explication
`private static final HashMap settings`	`HashMap` correspondant au fichier JSON config.json, dont le contenu a été obtenu via `Utils.readJsonFile`.
`private static final HashMap db_settings`	`HashMap` provenant de la constante `settings`, contenant les paramètres spécifiques de la base de données.
`private static final Database db`	Instanciation unique de la classe Database, prenant en entrée les informations correspondantes du fichier `config.json` contenues dans `db_settings`.
`public static Database getDb()`	Getter renvoyant l'objet `Database` instancié précédemment. Utile pour accéder à la base de données depuis d'autre classes comme les classes de pages évoquées plus haut (d'où la visibilité `public`).
`private static final double defaultCreditRate`	Le taux de crédit par défaut, tel que configuré dans `config.json`.
`private static final double defaultDiscountedPrice`	Le pourcentage de la valeur du véhicule par défaut, correspondant à sa valeur de solde, tel que configuré dans `config.json`.
`public static double getDefaultCreditRate()`	Getter renvoyant le taux de crédit par défaut.

Attribut	Explication
`public static void automaticDiscountOnOldVehicles()`	Méthode invoquée dans Main (au démarrage du programme), exécutant une requête SQL qui actualise le prix de chaque véhicule en fonction de `defaultDiscountedPrice`, où leur date de publication est supérieure à deux mois.
`public static void automaticOrderStatusRefresh()`	Méthode invoquée dans Main (au démarrage du programme), exécutant une requête SQL qui change le statut de la commande de “confirmé” à “livré” si son statut n’est pas “livré” et que la date de commande est supérieure à deux mois.

📌 Ces deux dernières méthodes sont publiques car appelées dans `Main`.

Fichier `config.json`

Utiliser un fichier de configuration permet d’éviter une recompilation du code en cas de changement de base de données, ou de valeurs pour d’autres aspects.

Notre fichier de configuration prend la forme suivante :

```
{
    "database": {
        "protocol": "mysql",
        "host": "localhost",
        "port": 3306,
        "db_name": "java_cars",
        "username": "root",
        "password": "root"
    },
    "default_credit_rate": 0.05,
    "default_discounted_price": 0.8,
    "images": {
        "logo": "https://www.pixenli.com/image/0LcOf6Qp"
    }
}
```

Fonctions transverses

Un ensemble de fonctions transverses utiles à la bonne exécution du programme sont recensées de visibilité `public` et `static` dans la classe `Utils`.

En voici un aperçu :

Fonction	Description	Fonctionnement

Fonction	Description	Fonctionnement
`String getLastSubstring(String input, int index)`	Renvoie les derniers caractères d'un `String` passé en entrée, dont l'espacement avec la fin du `String` correspond au paramètre `index`.	On utilise `substring()` et `length()` pour renvoyer la sortie attendue.
`String getFileAsString(String path)`	Permet d'obtenir le contenu d'un fichier dans l'arborescence du projet / du système via son chemin d'accès absolu ou relatif, sous la forme d'un `String`.	Ce code est en partie inspiré de StackOverflow. Si le programme est exécuté depuis un .jar (via `isRunningFromJar()`) : On récupère via `ClassLoader` le chemin absolu d'exécution du fichier .jar. On instancie ensuite un `Scanner` qui prend en entrée un nouveau `FileReader`, lui même prenant en entrée la concaténation du `path` (variable d'entrée) et du chemin absolu d'exécution du .jar. Enfin, on utilise un `StringBuilder` auquel on ajoute chaque ligne ainsi qu'un espace du `Scanner` ci-dessus. On retourne enfin le `StringBuilder` via `.toString()`. Sinon, On utilise : `return Files.readString(Path.of(path))` *(cette méthode ne fonctionne pas dans un fichier .jar, seulement en exécution via un IDE)* Le code ci-dessus est exécuté dans un `try…catch`, où en cas d'erreur celle-ci est affichée dans la console, avec le statut de `isRunningFromJar()` et un `String` vide est retourné.
`boolean isRunningFromJar()`	Retourne `true` si le programme est exécuté depuis un fichier .jar, `false` sinon.	On récupère le protocole de la classe courante (`Utils`) via `Utils.class.getResource("Utils.class")).getProtocol()`. Si le protocole est égal à "jar" ou "rsrc", il s'agit d'une exécution depuis un fichier .jar. On renvoie donc cette condition booléenne.
`String anonymizeCardNumber(String cardNumber)`	Renvoie le numéro de carte bancaire en entrée sous la forme "1234 xxxx xxxx 4567"	On utilise `substring(0,4)` et `getLastSubstring()`, en concaténation avec " xxxx xxxx ", pour obtenir le résultat attendu.

Fonction	Description	Fonctionnement
`String anonymizePhoneNumber (String phoneNumber)`	Renvoie le numéro de téléphone passé en entrée sous la forme “06 xx xx xx xx 12”.	On utilise le même procédé que `anonymizeCardNumber`.
`String capitalizeFirstLetter (String input)`	Renvoie la même chaîne de caractère passée en entrée, avec la première lettre mise en majuscule.	On utilise la concaténation de deux `.substring()`, le premier étant la première lettre, et `.toUpperCase()`.
`String numberToString (Number number)`	Renvoie le nombre sous la forme d’un `String` (par exemple, 4.3 devient “4.3”).	On renvoie la concaténation du nombre et d’un `String` vide.
`String escapeSQLChars (String input)`	Remplace dans un `String` les valeurs ‘ et “ par des `String` vides.	On renvoie le même String qu’en entrée, avec les valeurs ‘ et “ supprimées via `.replace()`.
`HashMap readJsonFile (String path)`	Retourne le contenu d’un fichier .json sous la forme d’une `HashMap`.	On utilise la librairie `com.fasterxml.jackson.databind.ObjectMapper`, pour transformer le contenu d’un `String` à la structure d’un objet `JSON` en `HashMap`. Pour obtenir ce `String`, on utilise `getFileAsString`.

À propos de la réalisation du projet

Répartition du travail entre les membres du groupe

Notre code étant hébergé sur GitHub, nous avons pu aisément collaborer sur ce projet. Cependant, aucune répartition n’a été arrêtée à priori. Celle-ci a évoluée en fonction de l’avancement du projet. On a chacun une idée de ce qu’on fait, et on demande à l’autre d’ajouter des fonctionnalités ou d’effectuer des tests. On avance comme cela en parallèle. En analysant avec le recul, on peut écrire :

- Fares a principalement contribué sur la création du serveur web, l’implémentation de HttpHandler et la création de la classe Page, ainsi que d’une majorité des classes héritées. Il a également créé une partie des documents HTML ainsi que des éléments de style en CSS.

- Loïc a conçu et créé la base de données, ainsi que son connecteur en Java. Il a également créé les Utils, pour lire un fichier en String ou en HashMap, ou déterminer si le programme est exécuté depuis un .jar, entre-autres. Il a également créé certaines pages (dont DocumentPage), et contribué à d'autres, notamment en ce qui concerne la connexion à la base de données et la transformation de ces dernières.

Difficultés rencontrées

Les plus grandes difficultés que nous avons rencontré sont :

- L'export en .jar, passant par Eclipse (nous développons sous IntelliJ) ne fonctionnait pas toujours. Cela a été résolu par un ajout des librairies dans le Build Path de Eclipse, manuellement et localement.
- L'utilisation de `getFileAsString` depuis un fichier .jar. L'utilisation de `Files.readString(Path.of(path))` ne fonctionnait pas, il a été nécessaire de trouver une nouvelle solution, qui n'a pas été aisée à trouver. En effet, il était nécessaire de calculer les chemins absolus des fichiers depuis la racine du disque pour que le .jar puisse les détecter, et d'utiliser un `Scanner` pour y accéder.
- La détection du .jar a été difficile car, utilisant le protocole d'exécution (souvent "jar" depuis un .jar, et "file" depuis un IDE), des erreurs étaient renvoyées. Cela était dû au fait que Eclipse exporte les .jar avec un protocole spécifique : "rsrc".
- La création du connecteur de base de données `Database` a été compliquée car il était nécessaire d'itérer de multiple fois dans le `ResultSet` et le `ResultSetMetaData` pour pouvoir lier, dans des `HashMap`, les noms des colonnes de leur valeur. Nous avions souhaité faire cela pour simplifier les accès aux données dans le code.
- Certaines double jointures de la base de données rendaient difficile l'accès à certaines valeurs, notamment les `id`. Cela a été résolu par un usage de `as` en SQL.
- Obtenir l'`id` en paramètre GET pour DocumentPage pouvait poser problème si l'id n'existait pas dans la base de données. Un `try…catch` a résolu ce problème.

> 📌 Il n'y a pas eu de problématiques non-techniques majeures (telles qu'un conflit entre les membres).

Ce qui n'a pas été fait

- La création d'un script SQL calculant automatiquement les taxes lors de l'achat d'un véhicule n'a pas été fait. Les taxes sont calculées en Java.
- Nous n'avons pas pu créer un système d'utilisateur et de connexion avec un identifiant / mot de passe. À la place, lors d'un achat, l'utilisation d'une adresse e-mail existante lie

l’achat à l’utilisateur existant associé. De même, un utilisateur peut suivre ses commandes et télécharger ses documents via son e-mail depuis la page `track`.

- Nous n’avons pas créé de système type “ORM” (Object-Relational Mapping), tel que des classes `Car extends Vehicle` ou `Motorcycle extends Vehicle`. Nous avons préféré utiliser des requêtes SQL pour afficher les informations et interagir avec la base de données directement.
- Notre projet suit une architecture propre à notre IDE IntelliJ. Ainsi, il n’a pas été possible d’utiliser `javac` pour compiler les fichiers .java en .class dans notre `makefile`. Ainsi, l’exécution make ne permet que d’exécuter le projet en gérant les dépendances, si et seulement si les fichiers .class sont déjà compilés.
- Une gestion plus granulaire des véhicules n’a pas été possible. Actuellement, seuls les modèles sont différenciés dans la base de données. Chaque commande implique une quantité de véhicules du même modèle, sans possibilité de distinguer chaque véhicule. Les numéros d’immatriculation sont actuellement générés aléatoirement.