VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY

HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY

WEB PROGRAMMING

ASSIGNMENT REPORT

JOB POSTING WEB

CC01 — SEMESTER 251

GROUP 7

LECTURER: DR. NGUYEN DUC THAI

Student	ID
Phạm Lê Quân	2153749

HO CHI MINH CITY – DEC – 2025

TABLE OF CONTENT

I. Introduction

1. Project Overview

"The Executive Garage" is a dynamic web application designed as a sophisticated online showroom for a luxury and performance car dealership. The platform provides users with an intuitive interface to browse, search, filter, and view detailed information about an exclusive collection of high-end automobiles.

The project's primary goal is to create a seamless and engaging user experience, from initial discovery to detailed vehicle inspection. It serves as a digital bridge between car enthusiasts and their dream vehicles, showcasing top brands like Porsche, BMW, and Mercedes-Benz.

2. Core Features

The application is built with a rich set of features to meet the needs of modern users:

- **User Authentication:** A complete authentication system allowing users to register, log in, and log out. It includes both traditional email/password registration and social login integration with Google and Facebook.
- **Password Management:** Secure password handling with hashing and a "Forgot Password" feature that emails users a new password using the EmailJS service.
- **Dynamic Car Catalog:** A comprehensive catalog of cars that can be filtered by brand and sorted by model name (ascending/descending).
- **Advanced Search:** A powerful search functionality that allows users to find cars by model name, brand, or body style. It includes a live search suggestion feature powered by AJAX for an enhanced user experience.
- **Detailed Vehicle Pages:** Each car has a dedicated page displaying its specifications, description, image gallery, price, and availability across different dealership locations.
- **Responsive Design:** The user interface is fully responsive, ensuring a consistent and accessible experience across desktops, tablets, and mobile devices, built on the Bootstrap 5 framework and CSS’s media.
- **SEO and Accessibility:** The project incorporates 3 basic SEO principles, including meta descriptions, sitemap (*sitemap.xml*, *robots.txt*), and semantic HTML.

3. Technology Stack

Other than the foundational technologies of PHP, HTML, CSS, and JavaScript, this project utilizes:

- **Bootstrap** for responsive UI components.
- **Font Awesome** for icons.
- **HybridAuth** for Social login (Google/Facebook).
- **EmailJS** for emailing clients (e.g., password resets).

II. Design Choices

1. System Architecture

The project's structure is inspired by the Model-View-Controller (MVC) design pattern to ensure a clean separation of concerns, making the codebase organized, scalable, and easier to maintain.

- **Controller** (*/quanswebsite/app/Controllers*)**:** This layer acts as the intermediary between the Model and the View. It receives user requests, processes input, interacts with the Model to fetch or update data, and then passes that data to the appropriate View for rendering. Examples include CarController.php and AuthController.php.
- **Model** (*/quanswebsite/app/Models*)**:** This layer is responsible for all data-related logic. It contains PHP classes (e.g., Car.php, User.php, Brand.php) that interact directly with the MySQL database. They encapsulate SQL queries for creating, reading, updating, and deleting records.
- **View** (*/quanswebsite/app/Views, /quanswebsite/public*)**:** This layer handles the presentation and user interface. It consists of PHP files that render HTML, such as index.php and show.php. Reusable UI components like the header (header.php) and footer (footer.php) are separated into partials for modularity. It also contains the specific car's images.
- **Configuration** (*/quanswebsite/config*)**:** Centralized configuration files for the database connection (*database.php*) and third-party services (*hybridauth_config.php*) are stored here, separate from the application logic.
- **Database** (*/quanswebsite/config*)**:** This contains a script (*script.sql*) to initialize the database with mockdata in it.
- **Public Directory** (*/quanswebsite/public*)**:** This is the web server's document root and the single point of entry for all user-facing pages. It also contains public assets like CSS (*/css/style.css*), helper functions (*/helper*), JavaScript functions (*/js*), general car brand's images (*/uploads*), and some other pages like login, logout, etc.
- **HybridAuth** (*/quanswebsite/vendor*)**:** This stores the HybridAuth library, a third-party dependency used to handle social logins for Google and Facebook.

2. Database Schema

The database, the_executive_garage, is designed to be relational and efficient. The schema is defined in *database/script.sql*.

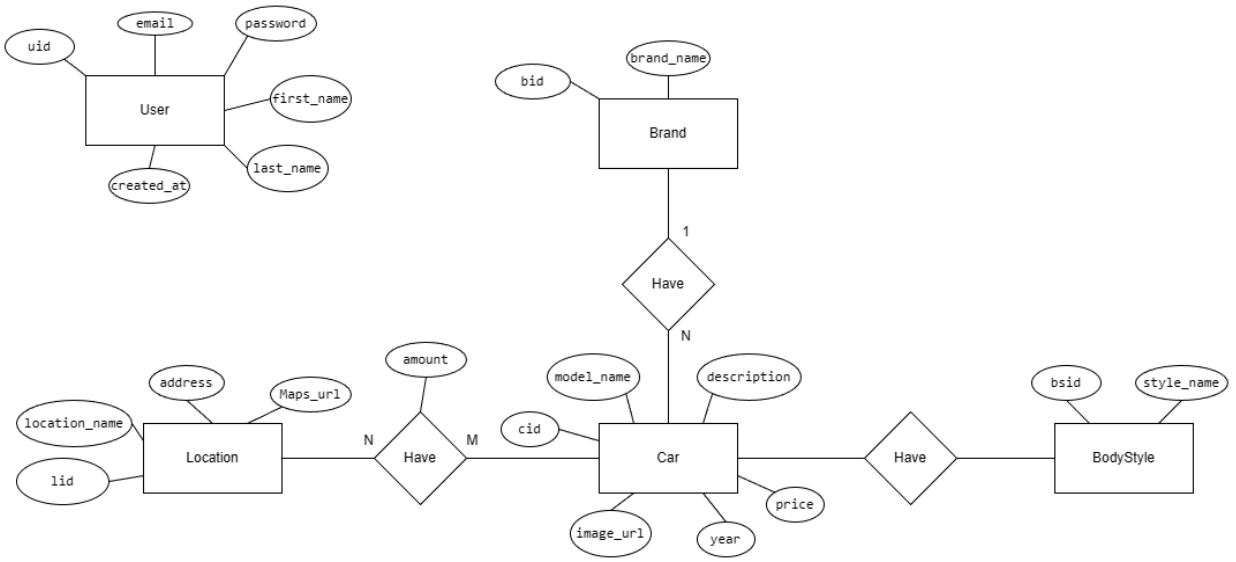

Figure 1: ER diagram (*/quanswebsite/other/ER diagram.png*)

- **User:** Stores user credentials and personal information. The password field is securely hashed.
- **Brand, BodyStyle:** Lookup tables that store brand and body style names, helping to normalize data and avoid redundancy.
- **Car:** The central table containing all vehicle details. It uses foreign keys (brand_id, style_id) to link to the Brand and BodyStyle tables.
- **Location:** Stores information about physical dealership locations, including a Google Maps URL.
- **Inventory:** A junction table that models the many-to-many relationship between Car and Location. It tracks the availability of each car at each dealership, allowing for a flexible inventory management system.

This normalized structure minimizes data duplication and ensures data integrity through the use of foreign key constraints.

Moreover, the reason why i used VARCHAR(255) for User's password is because the hashing I am using is PASSWORD_DEFAULT, not PASSWORD_BCRYPT (Which uses CRYPT_BLOWFISH algorithm to create 60 characters), so it won't always be a fixed 60 characters later on when there are updates in the PHP's PASSWORD_DEFAULT (newer hashing algorithm will be released overtime and assigned in PASSWORD_DEFAULT).

This strategy is deliberately chosen over hardcoding PASSWORD_BCRYPT and using a fixed VARCHAR(60). It ensures the application can automatically adopt stronger security standards as they become available, mitigating the long-term risk of hackers exploiting vulnerabilities in older, potentially outdated hashing algorithms.

3. User Interface (UI) and User Experience (UX)

The UI design focuses on being clean, modern, and intuitive.

- **Visual Identity:** The color scheme uses a bold yellow (#FFDF00) as the primary accent color against dark and light neutral backgrounds, creating a premium and energetic feel. The 'Inter' font is used for its excellent readability.
- **Responsive Layout:** Built with Bootstrap's grid system and .container class, the layout fluidly adapts to different screen sizes. Custom CSS media queries are also used to change element styles at different resolutions. Special attention was given to the mobile experience, with elements like the navigation collapsing into a toggleable menu and form layouts stacking vertically.
- **Interactive Elements:**
    - **Responsive Navigation:** As seen in header.php, on smaller screens the main navigation bar collapses into a compact dropdown menu. Users can click a toggler icon to expand or collapse the list of navigation links, ensuring full site accessibility without cluttering the view. This is handled by Bootstrap's built-in JavaScript components.
    - **Brand Showcase:** On the homepage, brands are presented as large, clickable image buttons that reveal the brand name on hover, providing a visually engaging way to filter the car list.
    - **AJAX Live Search:** The search bar on the homepage provides instant feedback with a dropdown of suggestions as the user types, reducing friction and speeding up the search process. This is powered by a fetch call in ajaxSearch.js to a PHP endpoint.
    - **Dynamic Sorting:** The car list can be sorted by model name (A-Z or Z-A) by clicking a "Sorting" button on the index.php page. The user's preference is stored in the session and persists across page loads and filters.
    - **Social Login Buttons:** The login.php page offers prominent buttons for Google and Facebook. These buttons use a CSS transition to expand on hover, revealing the full text "Log in with..." which saves space while remaining intuitive.
    - **Interactive Location Links:** On each car's detail page (*show.php*), dealership locations are listed. These are clickable links that open the specific location in Google Maps in a new tab, providing immediate geographical context for the user.
    - **Footer Functionality:** The site footer (*footer.php*) includes social media icons that link to the company's external profiles. It also features a "Back to Top" button, which uses JavaScript (*otherEffects.js*) to provide a smooth scrolling animation to the top of the page, improving navigation on long pages.

III. Implementation Details

1. Authentication System

The authentication flow is a critical component of the application.

- **Registration (*signup.php*):** New users can create an account by providing their first name, last name, email, and password. The AuthController.php handles the request, hashes the password using password_hash(), and saves the new user to the database via the User model.
- **Login (*login.php*):** Users can log in with their email and password. The controller verifies the credentials using password_verify(). Upon success, the user's data is stored in the $_SESSION superglobal to maintain the logged-in state across the site.
- **Social Login (*social_login.php*, *social_callback.php*):**
  - The user clicks a "Log in with Google/Facebook" button on the login.php page, which directs them to *social_login.php* with the provider name.
  - *social_login.php* initializes the HybridAuth library and redirects the user to the provider's authentication page.
  - After the user authorizes the application, the provider redirects them back to *social_callback.php*.
  - The callback script retrieves the user's profile information (first name, last name, email) from the provider.
  - It then checks if a user with that email already exists in the database.
    - If the user exists, they are logged in.
    - If not, a new account is automatically created for them with a secure, randomly generated password, and then they are logged in.
- **Password Reset (*forgot_password.php*):**
  - A user enters their email address.
  - The system generates a new random password and hashes it.
  - The hashed password is saved to the user's record in the database.
  - The page then uses JavaScript (*resetPassword.js*) to make a client-side call to the EmailJS service, sending the new, unhashed password directly to the user's email address. This offloads the email-sending responsibility from the PHP backend.

2. Car Listing and Search

The core of the application is the car catalog.

- **Pagination:** To handle a large number of cars efficiently, all car listings (all cars, by brand, or search results) are paginated. The CarController.php calculates the total number of pages and fetches only the cars needed for the current page ($carsPerPage = 6), using LIMIT and OFFSET in the SQL queries.

- **Filtering and Sorting:** The index.php page checks for $_GET parameters like brand_id to filter the results. A $_SESSION variable (carNameSorting) tracks the user's preferred sort order (ASC/DESC), which persists across page loads.
- **AJAX Live Search:**
  - The *ajaxSearch.js* script listens for input events on the search box.
  - On each keypress, it sends a fetch request to a dedicated PHP endpoint, *ajax_search.php*.
  - This endpoint performs a LIKE query against the database for car models, brands, and body styles.
  - It returns a JSON array of matching cars.
  - The JavaScript then dynamically creates and displays a list of suggestion links below the search bar.

3. Car Details View

The show.php page provides an in-depth look at a single vehicle.

- **Data Fetching:** The page retrieves the car's ID (cid) from the URL's query parameters.
  - The CarController.php is called to execute showCar($cid). This primary method fetches the core vehicle data from the Car table, including its model name, price, year, description, and image URL. It also retrieves the foreign keys for brand_id and style_id.
  - Once the primary car information is retrieved, it calls BrandController->getBrandNameById() to get the brand name (e.g., "Porsche") and BodyStyleController->getBodyStyleById() to get the body style name (e.g., "Coupe").
- **Inventory Display:** A separate query is made to fetch the car's inventory status from the Inventory table. This returns a list of all locations where the car is available and the quantity at each, which is then displayed as a list of clickable links pointing to the location on Google Maps.

IV. Lessons Learned

1. Challenges and Solutions

- **Initial Misunderstanding of the MVC Pattern:** Although I understood the MVC concept theoretically, translating it into a practical folder and code structure was an initial challenge. At first, I misinterpreted the Controller's role, believing it only managed the main content area of a page. This led to an incorrect implementation where the View layer communicated directly with the Model to fetch data.

  The turning point came from observing the structure of a group assignment project. Seeing a practical, working example of the MVC pattern provided the clarity that theoretical knowledge alone could not. It was a key learning opportunity that demonstrated how the Controller should act as the central intermediary. Armed with this new understanding, I was able to refactor my own project to follow the correct flow: a request goes to the Controller, which then calls the Model for data, and finally passes that data to the View for rendering. This was a critical step in building a properly structured and maintainable application.

- **Implementing Social Login:** When implementing Social Login, I had to configure both Google Cloud Console (https://console.cloud.google.com/) for Google login and Meta for Developers (https://developers.facebook.com/) for Facebook login. For both providers, I faced many challenges due to the lack of comprehensive tutorials and the complexity of their official guides, especially for PHP and HTML integration.

  For Google, in the "Clients" section, I needed to set:

  - Authorized JavaScript origins to http://localhost
  - Authorized redirect URIs to http://localhost/quanswebsite/public/social_callback.php (for the social login callback after authorization)

  For Facebook, I discovered that when the app is in "Development" mode, it works with http, so I did not need to host the site with https for testing. In the "Use case" section, I had to:

  - **Set Permissions and features:** Mark email as "action" and ensure public_profile is already in "action" state

    - **In Settings:** Set Valid OAuth redirect URIs to http://localhost/quanswebsite/public/social_callback.php (for the social login callback after authorization)
- **Secure Credential Handling:** Storing API keys and secrets (like those for HybridAuth and EmailJS) directly in version-controlled files is a security risk. The solution was to place sensitive keys in a separate *hybridauth_config.php*, *emailjs_config.php* files and add these files, along with the vendor/ directory, to .gitignore. The README.md instructs developers to create this file locally.

  Additionally, direct accesses to *hybridauth_config.php* and *emailjs_config.php* are explicitly blocked via the .htaccess file. This prevents external users from accessing the file and potentially exploiting the API keys, further strengthening the application's security posture.

- **Client-Side Emailing:** Implementing a "Forgot Password" feature was a challenge. The initial attempt involved using PHP's native mail() function, but it proved unreliable in the local XAMPP environment, which is a common problem as it requires a properly configured mail server (SMTP) that is not set up by default.

  Instead of undertaking complex server configuration or integrating a heavy backend mailer library, EmailJS provided a clever solution. It offloads the email sending responsibility to the client-side, where JavaScript calls the EmailJS API directly, as seen in header.php and resetPassword.js. This simplified the backend architecture significantly but required careful handling of the public API key on the frontend to ensure it was exposed securely.

- **XAMPP MySQL Failure:** A significant technical hurdle was encountered when the MySQL service in XAMPP began to shut down unexpectedly (Error: MySQL shutdown unexpectedly), making database access impossible and halting development. After researching the issue, a solution was found in a community guide (https://www.youtube.com/watch?v=tYUXvUy_oyU) which involved a manual restoration of MySQL's core data files. The specific recovery steps were as follows:
    - Navigate to the */mysql* folder within the XAMPP installation directory.
    - Create a backup copy of the data folder.
    - Enter the original */mysql/data folder*.
    - Delete the *mysql*, *performance_schema*, *phpmyadmin*, and *test* subfolders.
    - Delete all files except for the *ibdata1* file and the folders corresponding to the project's databases.
    - Navigate to the */mysql/backup* folder.
    - Copy all contents from the backup folder except for the *ibdata1* file.
    - Paste the copied files and folders back into the */mysql/data* folder.

This intricate process successfully restored the MySQL service without data loss. It underscored the importance of being able to troubleshoot not just the application code, but also the underlying development environment itself.

2. What Went Well

- **Successful MVC Implementation:** The project's greatest success was the final implementation of the MVC-like architectural pattern. Despite initial confusion, observing a practical example was a turning point that led to a successful refactoring of the codebase. The final structure, with its clear separation of concerns between Models, Views, and Controllers, made the application significantly easier to navigate, debug, and extend.
- **Effective Integration of Third-Party Services:** The project demonstrates a strong ability to solve complex problems by integrating external services. Instead of getting blocked by the complexities of OAuth or SMTP server configuration, pragmatic solutions were found. HybridAuth was successfully configured to handle the entire social login flow, and EmailJS was cleverly used to implement a password reset feature without a backend mailer, showcasing resourceful problem-solving.
- **Robust Security Foundations:** Proactive steps were taken to secure the application. The decision to isolate API keys and sensitive credentials in separate config files, exclude them from version control via .gitignore, and block direct web access using .htaccess establishes a strong security posture. This multi-layered approach shows a solid understanding of protecting sensitive data.
- **Component-Based Views and UX:** Breaking the UI into reusable partials (header.php, footer.php) streamlined development and ensured a consistent look and feel. Furthermore, the implementation of the AJAX live search feature significantly improved the user experience, demonstrating the power of combining frontend JavaScript with a simple PHP backend to create a dynamic and responsive interface.

3. Future Improvements

- **Refactor to a Full Framework:** Having successfully implemented a manual MVC structure, the next logical step would be to migrate the project to a full-fledged PHP framework like Laravel or Symfony. This would provide more robust features out-of-the-box, such as an ORM (like Eloquent), a powerful routing engine to create "pretty URLs" (solving the .htaccess goal), and built-in security features like CSRF protection.
- **Expand Social Login Options:** Building on the successful integration of Google and Facebook login, the application could be enhanced by adding more social login providers. Integrating professional networks like LinkedIn or other popular platforms like X (formerly Twitter) would broaden the user base

and offer more convenience for different user demographics. This would involve extending the existing HybridAuth implementation.

- **Refine CSS and Component Styling:** The current styling uses a mix of a dedicated stylesheet, Bootstrap classes, and inline styles (e.g., in header.php). A future task would be to refactor this by moving all inline styles into the style.css file and creating a more consistent design system. This would improve maintainability and ensure a cleaner separation between structure (HTML) and presentation (CSS).
- **Implement Unit and Integration Testing:** To improve code quality and ensure long-term stability, introducing a testing framework like PHPUnit would be invaluable. Writing automated tests for the business logic within the Controllers and Models would help prevent regressions and catch bugs before they reach production.