**CUKUROVA UNIVERSITY**

**COMPUTER ENGINEERING**

**E-COMMERCE MIDTERM**

**PROJECT REPORT**

**CAR SALES WEBSITE:**

**MEHA CARS**

**Hande OCAK – 2019556048**

**Melike ARIKAN - 2019556009**

**Introduction**

MEHA Cars has been developed as a platform where users can buy and sell vehicles. This report includes the functions we offer to users on the website and a detailed description of the pages within the site. The MEHA Cars website is equipped with numerous functions and a user-friendly interface to facilitate vehicle buying and selling transactions. From user registration and login processes to adding and managing vehicles, from the pricing offer system to organizing test drives, a wide range of services are provided. Thanks to communication options, users can easily get support if they encounter any problems. With these comprehensive features, MEHA Cars offers an efficient and reliable platform to its users.

**User Functions**

1. **User Registration Panel**
   * Users are required to register to sell vehicles. This panel allows new users to register to the system.
2. **User Login**
   * A user login panel is available for registered users to log in to the system and perform vehicle sales and other transactions.
3. **Vehicle Adding and Posting System**
   * Users can add the brand, model, year, description, and vehicle images of the vehicles they want to sell. They can upload their vehicles to the system with this information and create listings.
4. **Vehicle Status Management**
   * Users can activate or deactivate the vehicles they have uploaded. This feature allows users to temporarily withdraw their vehicles from sale.
5. **Profile Update**
   * Users can update their profile information. This allows them to keep their contact information and other personal details up to date.
6. **Vehicle Viewing and Filtering**
   * Users can view vehicles uploaded by other users and filter these vehicles by brand, model, and year.
7. **Price Sorting System**
   * Vehicles can be sorted by their prices in ascending or descending order. This feature helps users find vehicles that fit their budgets easily.
8. **Price Offer System**
   * Users can make price offers for the vehicles they like. Offers are received using the auction method, and the highest bid determines the winner.
9. **Setting a Date for Test Drive**
   * Users can set a date with the seller to test drive the vehicle they want to buy. This feature allows buyers to get a closer look at the vehicles.
10. **Contact and Support**
    * Users can contact us via the 'Contact Us' page in case of any problems. Options for sending a message, direct calling, and sending an email are available.

**Website Pages**

1. **Home**
   * The home page is the first page users encounter when they enter the site. It contains general information about the site and featured vehicles.

taşıt, araç, kara taşıtı, tekerlek, otomotiv tasarımı içeren bir resim

Açıklama otomatik olarak oluşturuldu

1. **Car Sell**
   * This is the page where users can create listings to sell their vehicles. Brand, model, year, description, and images are added on this page.
2. **Login Page**
   * This is the page where registered users can log in to the system.
3. **Register Page**
   * This is the page where new users can register to the system.
4. **Price Offer Page**
   * This is the page where users can make price offers for vehicles and view existing offers.
5. **Post Car, Car Detail Page, and Update Uploaded Vehicle Details Page**
   * This page allows users to create vehicle listings, view vehicle details, and update listing information.
6. **My Profile - Edit Profile**
   * This page allows users to view and update their profile information.
7. **Create and Update Test Drive**
   * This page allows users to set a date for a test drive and update existing test drive requests.
8. **List Test Drive Requests Page**
   * This page allows users to view and manage their test drive requests.
9. **View Uploaded Vehicles Page**

* This page lists all the vehicles uploaded by users. Filtering and sorting options are available here.

1. **View Shared Vehicles Page**

* This page allows users to see and manage the vehicles they have uploaded.

1. **Contact Us**

* This is the page where users can contact us. Options for sending a message, calling, and sending an email are available.

1. **View All Uploaded Vehicles Page**

* This page allows users to view all the vehicles uploaded to the platform.

1. **About Us**

* This page provides information about MEHA Cars, including the company's mission, vision, and history.

1. **KVKK Text**

* This page explains how users' personal data is protected and processed.

metin, ekran görüntüsü, yazı tipi, sayı, numara içeren bir resim

Açıklama otomatik olarak oluşturulduBefore introducing the codes, we combined the parts found on every web page of our project (such as footer, header, navbar) under the components folder and included them in each page as shown below.

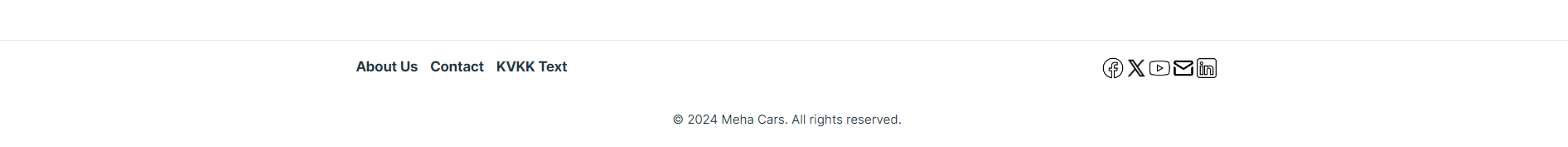
metin, ekran görüntüsü, yazılım, ekran, görüntüleme içeren bir resim

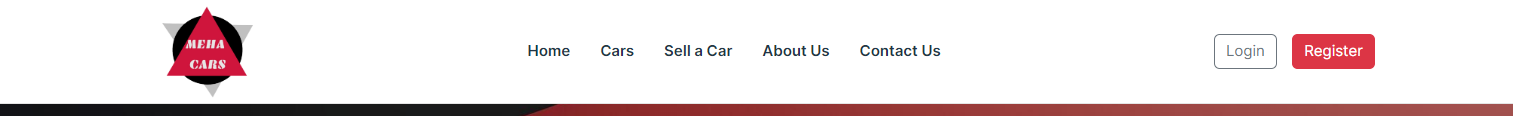
Açıklama otomatik olarak oluşturulduSure, let's take the footer section as an example. Here is how you can create a reusable footer component and include it in your web pages:

1. **Footer Section:**
   * A section of HTML that starts with the **<footer>** tag and ends with the **</footer>** tag. This section includes contact links and the copyright part.
2. **Links and Icons:**
   * **<div class="container d-flex justify-content-between">**: This **<div>** element is used to arrange the content inside the footer horizontally. The container and d-flex justify-content-between classes allow us to spread the content blocks towards the edges with equal spacing.
   * The **<ul>** elements inside the first **<div>** element represent unordered lists. We used the unordered list structure to position the icons more easily and neatly, and made the list icons invisible.
   * metin, ekran görüntüsü, yazı tipi, sayı, numara içeren bir resim

     Açıklama otomatik olarak oluşturulduThe **<ul>** elements inside the second **<div>** element contain the social media icons in the footer. These icons direct users to MEHA Cars' social media accounts when clicked. Each icon is defined with an **<img>** tag. The icons are visualized in png format under the /images/user/ directory.
3. **Copyright Information:**
   * **<div class="container text-center mt-4">**: This **<div>** element is used to arrange a section containing the copyright information vertically.
   * **<p>&copy; 2024 Meha Cars. All rights reserved.</p>**: This **<p>** element contains the copyright information. The © symbol represents the copyright sign.
4. **JavaScript Link:**
   * **<script src="..."></script>**: This **<script>** element is used to link the Bootstrap library's JavaScript file to the website. This file enables some interactive features and Bootstrap components to function on the website.

As a result of all these codes, we obtained a footer section like the one below.

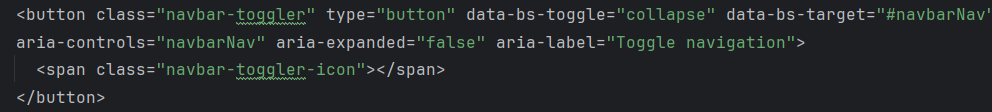


Our navigation section is as follows:

Our navigation section includes our logo in the top left corner and login/register buttons on the left side. We activated the hover feature on the buttons, so when the mouse hovers over the buttons, they turn gray. As a result of our responsive design, our website is responsive to screens of different sizes and takes the form below on smaller screens:

metin, ekran görüntüsü, yazılım, tasarım içeren bir resim

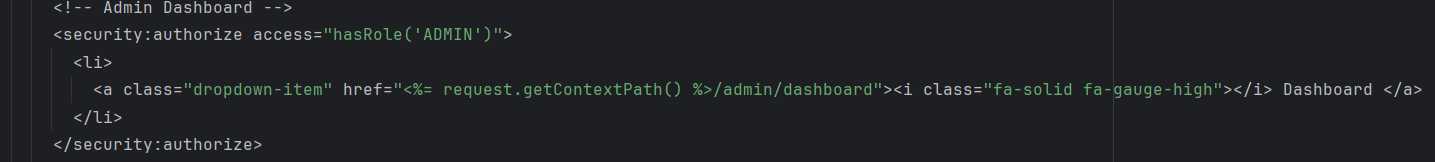
Açıklama otomatik olarak oluşturuldu

1. **Structure of the Navbar:**
   * The navbar starts with the **navbar navbar-expand-lg** class. This indicates that the navbar will expand when the screen size is **lg** (large) or larger. You can adjust the behavior of the navbar according to the screen sizes by changing this class (**lg**, **md**, **sm**, etc.).
   * The **sticky-top** class ensures that the navbar is fixed at the top of the page.
2. **Toggle Button:**
   * When the screen size is reduced, a toggle button becomes visible in the navbar. This button is defined with the **navbar-toggler** class. When the screen is small, users can click this button to open and close the menu.
   * Attributes like **data-bs-toggle**, **data-bs-target**, and **aria-controls** specify which menu the toggle button will open and close.
3. **Menu Items:**
   * The **navbar-collapse** class hides the menu inside the navbar on small screens. This menu opens and closes when the toggle button is clicked.
   * Menu items are located under the **navbar-nav** class. This class formats the menu items as a navbar list.
   * The **ms-auto** and **me-auto** classes align the menu items to the left and right.

metin, ekran görüntüsü, yazılım, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

1. **User Menu:**

Different menu items are displayed when the user is logged in or logged out. In this case, the **<security:authorize>** tags are used to conditionally specify the menu items.

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

metin, tekerlek, kara taşıtı, taşıt, araç içeren bir resim

Açıklama otomatik olarak oluşturulduIf we examine the 'about-us' page, which includes sections like the navbar and footer, the appearance on the browser is as follows:

tekerlek, metin, kara taşıtı, taşıt, araç içeren bir resim

Açıklama otomatik olarak oluşturuldu

This page uses Bootstrap's Carousel feature to increase user interaction and grab users' attention. The Carousel component is one of the ready-made components offered by Bootstrap and is used to create a slideshow. This feature is provided with the following code:

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

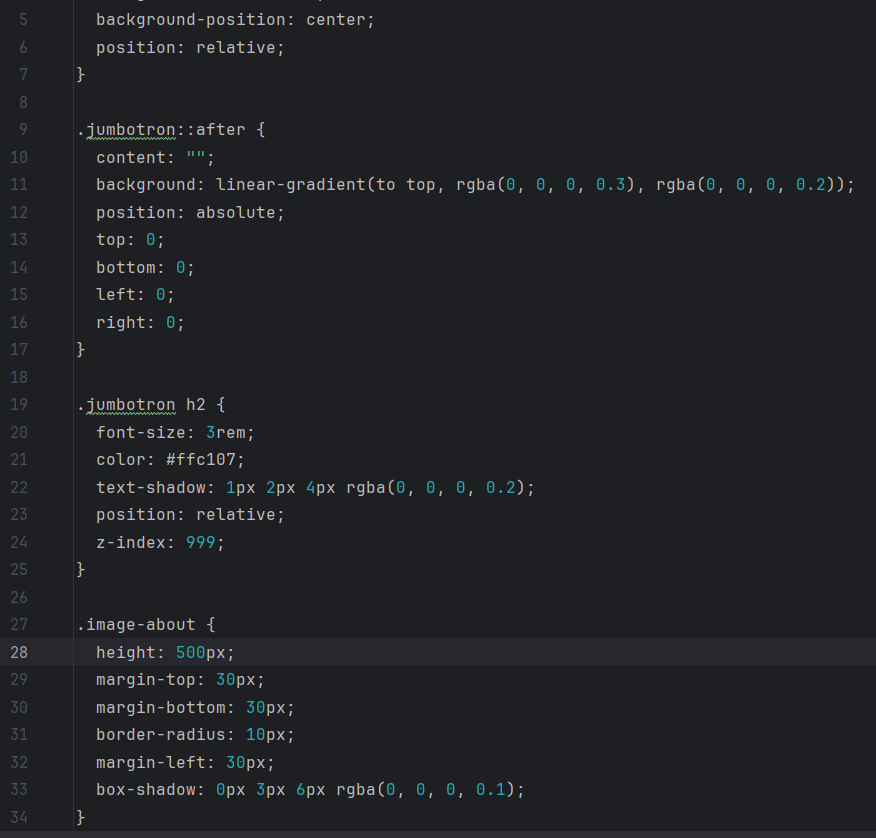
1. **<main>**: This tag represents the main content section of the HTML document. Typically, the most important content of the page is placed here.
2. **<div id="carouselExampleSlidesOnly" class="carousel slide" data-ride="carousel">**: This **<div>** tag creates a carousel. The **carousel** and **slide** classes come from the Bootstrap library and are used to create a slideshow. The **data-ride="carousel"** attribute enables automatic sliding.
3. **<div class="carousel-inner">**: This inner **<div>** tag groups the content of the slides within the carousel.
4. **Slides (carousel-item):**
   * **<div class="carousel-item active">**: Represents the first slide and is activated with the **active** class. This slide is the starting slide of the carousel.
   * **<img src="/images/car/1.jpg" class="d-block w-100" alt="Customer-Centric Approach ">**: This **<img>** tag represents the content of the slide. The **src** attribute specifies the path of the image. The **class="d-block w-100"** ensures the image is displayed as a block and takes up 100% width.
   * **alt="Customer-Centric Approach "**: This attribute specifies the alternative text for the image. If the image cannot be loaded or if the user has disabled images, this text serves as the placeholder.
5. **Other slides**: Unlike the first slide, other slides inside the **carousel-item** class are defined without the **active** class. These slides are displayed sequentially as the carousel is navigated.

The rest of the page is formed by adding the relevant text within divs. Additionally, there are related CSS files containing the CSS codes for the web pages.

metin, ekran görüntüsü, yazı tipi, tasarım içeren bir resim

Açıklama otomatik olarak oluşturuldu

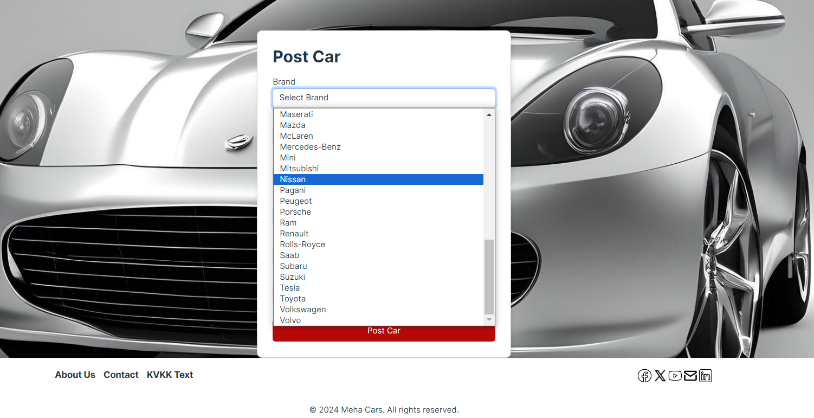
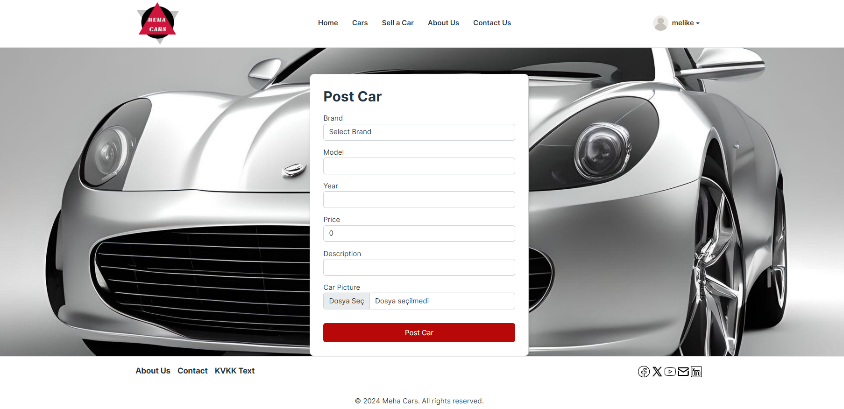
CSS files contain codes that create the layout of the pages. When we take the 'contact-us' page as an example;



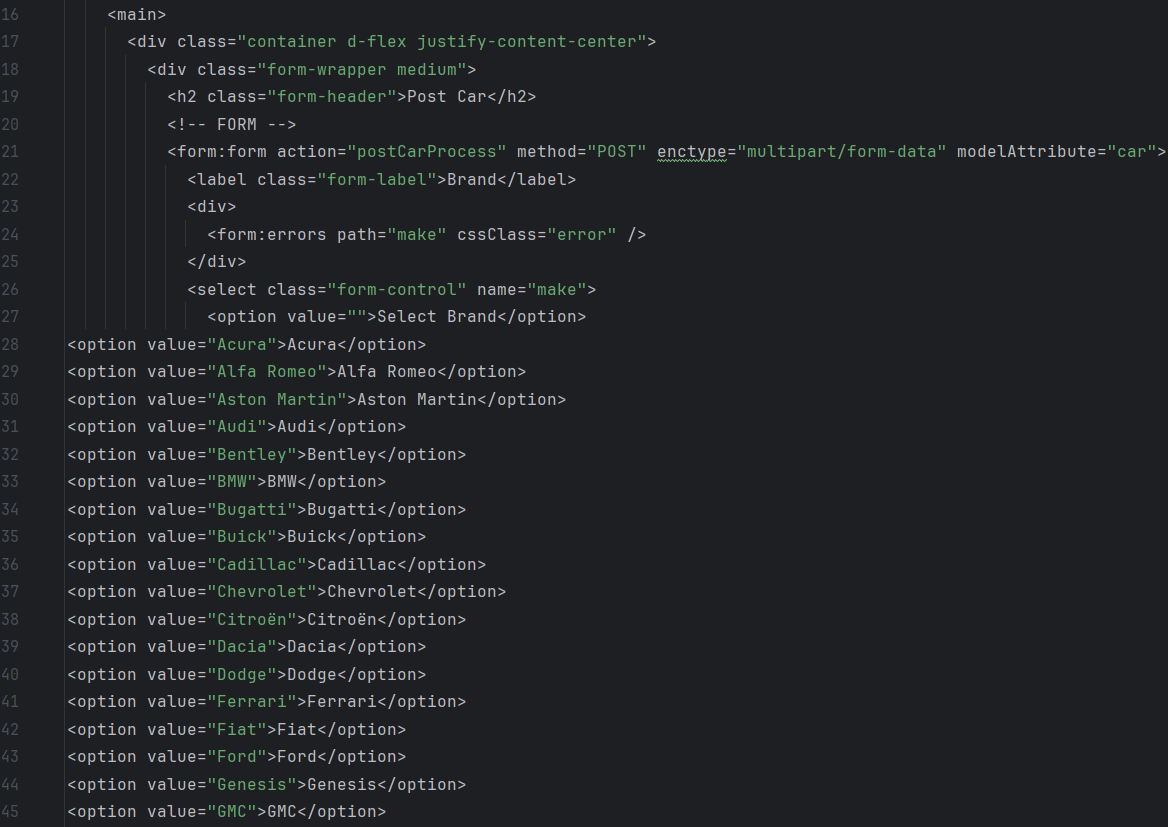
1. **.jumbotron::after**: This CSS rule adds an overlay to the jumbotron component. This overlay adds slight shading and a blur effect to the background via a linear gradient.
   * **content: ""**: Adds content to the jumbotron using the **::after** pseudo-element.
   * **background: linear-gradient(to top, rgba(0, 0, 0, 0.3), rgba(0, 0, 0, 0.2))**: Creates a background color via a linear gradient. This gradient adds transparency and gradually reduces transparency from the top.
   * **position: absolute**: Positions the overlay absolutely, making it cover the jumbotron content.
   * **top: 0; bottom: 0; left: 0; right: 0;**: Ensures the overlay covers the entire jumbotron.
2. **.jumbotron h2**: This CSS rule customizes the heading (h2 element) inside the jumbotron.
   * **font-size: 3rem**: Sets the font size of the heading.
   * **color: #ffc107**: Sets the text color of the heading (a shade of yellow).
   * **text-shadow: 1px 2px 4px rgba(0, 0, 0, 0.2)**: Adds a slight shadow to the heading text, making it more prominent.
   * **position: relative**: Positions the heading relative to its normal position.
   * **z-index: 999**: Ensures the heading has a higher z-index value, placing it above other elements.
3. **.image-about**: This CSS rule defines the style for images with the "image-about" class.
   * **height: 500px**: Sets the height of the image.
   * **margin-top: 30px; margin-bottom: 30px; margin-left: 30px;**: Sets the margins for the top, bottom, and left of the image.
   * **border-radius: 10px**: Rounds the corners of the image.
   * **box-shadow: 0px 3px 6px rgba(0, 0, 0, 0.1)**: Adds a slight shadow below the image, making it stand out.

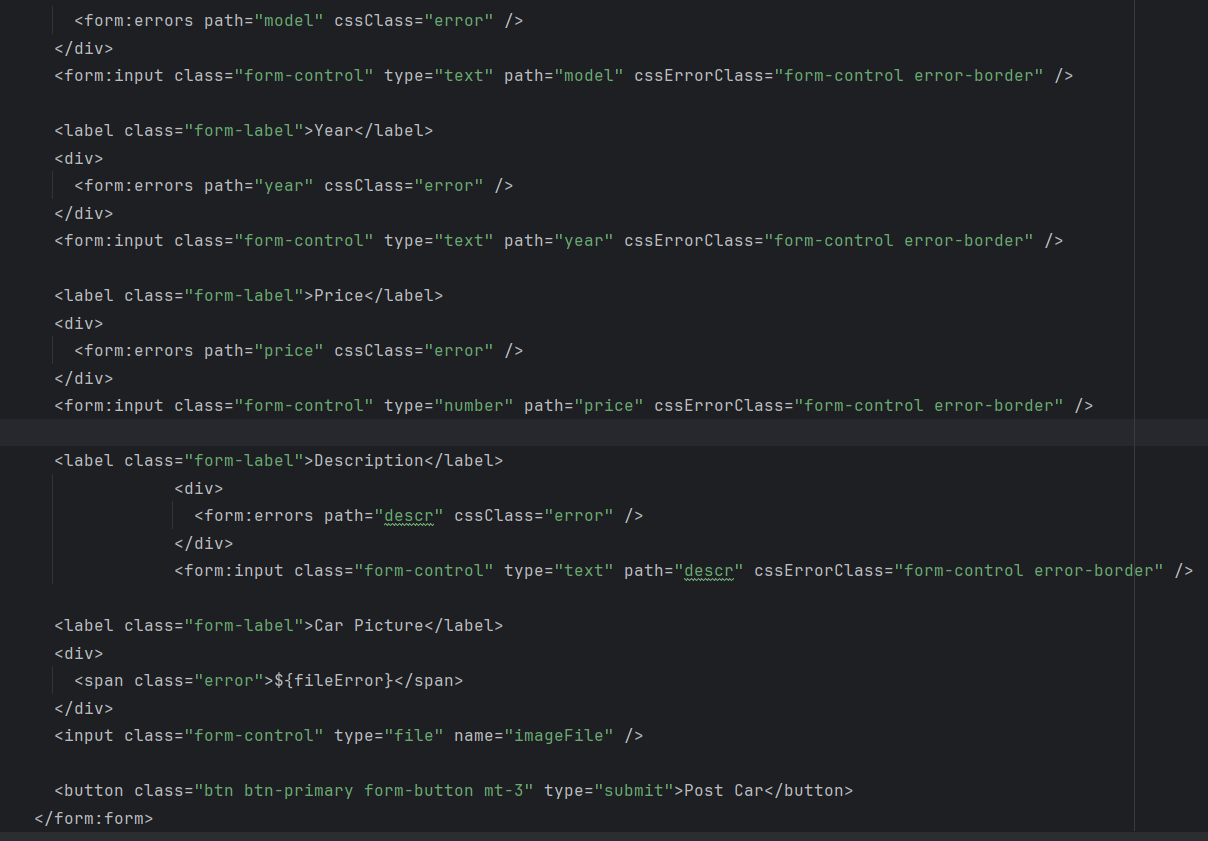
Thus, we adjust the padding, margin, font size, and shadow properties of the pages. This ensures a pleasing appearance for our pages.

This is how the users' car upload page appears:



To create these features, the following codes were used:





1. **<form:form action="postCarProcess" method="POST" enctype="multipart/form-data" modelAttribute="car">**: This **<form:form>** tag uses the Spring Form tag library, a part of the Spring Framework. The form targets a process named "postCarProcess." The form method is "POST," and the enctype attribute is set to multipart/form-data because the form should support file uploads. The modelAttribute attribute indicates that the fields in the form will be bound to a car object.
2. **<select class="form-control" name="make">**: Creates a dropdown menu to select the car's make. It is a select field named "make." The user can select a make, or if "Select Brand" is chosen, no brand is selected.
3. **<form:input class="form-control" type="text" path="model" cssErrorClass="form-control error-border" />**: Creates a text input field for the car's model. The "model" path represents the model field in the car object bound to the form. If an error occurs, the "error-border" class is added to the form field.
4. **<form:input class="form-control" type="text" path="year" cssErrorClass="form-control error-border" />**: Creates a text input field for the car's year. The "year" path represents the year field in the car object bound to the form. If an error occurs, the "error-border" class is added to the form field.
5. **<form:input class="form-control" type="number" path="price" cssErrorClass="form-control error-border" />**: Creates a number input field for the car's price. The "price" path represents the price field in the car object bound to the form. If an error occurs, the "error-border" class is added to the form field.
6. **<form:input class="form-control" type="text" path="descr" cssErrorClass="form-control error-border" />**: Creates a text input field for the car's description. The "descr" path represents the descr field in the car object bound to the form. If an error occurs, the "error-border" class is added to the form field.
7. **<input class="form-control" type="file" name="imageFile" />**: Creates a file upload field for the user to upload the car image. It is a file upload field named "imageFile."
8. **<button class="btn btn-primary form-button mt-3" type="submit">Post Car</button>**: Represents the submit button of the form. It is a primary colored button with the text "Post Car."

**MEGA CARS PROJECT BACKEND SIDE :**

**Technologies Used:**

**Java 11:** The backend part of the project was developed using the Java programming language. Since Java 11 was installed on my computer, this version was used.

**MySQL:** MySQL was used as the relational database management system.

**Spring Boot:** Spring Boot, a module of the Spring Framework, was utilized in the development process.

**Spring Security:** Spring Security was integrated to meet the security needs.

**JPA (Java Persistence API):** JPA was used to map database tables to entities. Hibernate implementation of JPA was preferred.

**Maven:** Maven was used as the package manager for the Java project

**Detailed Explanation:**

The backend part of the project was developed using the Java programming language, with Java 11 being the version installed on my computer. During the application development process, Spring Boot, a part of the widely used and developer-friendly Spring Framework, was employed. Spring Boot enables the rapid and efficient development of standalone, production-ready applications.

MySQL was chosen as the relational database due to its high performance and wide usage. To manage database operations more efficiently and easily, Java Persistence API (JPA) was utilized. JPA allows for mapping between database tables and Java classes, and for this mapping, the Hibernate implementation of JPA was used. Hibernate provides a robust and flexible implementation for JPA.

To ensure the security layer of the application, Spring Security was used. Spring Security offers a comprehensive security service to meet the security requirements in applications and played a crucial role in securing the project.

For project management and dependency control, Maven was employed. Maven simplifies the process from project configuration to compilation and testing. With Maven, project dependencies can be easily managed, and the processes of compiling and testing the project can be automated.

These technologies facilitated the rapid, secure, and efficient development of the project's backend.

**LAYERS OF THE PROJECT**

**ENTITY LAYER**

The entity layer contains the 'entities' of the project, which correspond to the tables in the database. The entities in this layer include:

* metin, ekran görüntüsü, yazı tipi içeren bir resim

  Açıklama otomatik olarak oluşturuldumetin, ekran görüntüsü, yazı tipi, tasarım içeren bir resim

  Açıklama otomatik olarak oluşturulduCar
* CarBidding
* CarPicture
* ProfilePicture
* Role
* TestDrive
* UserAccount
* UserProfile

These entities are mapped to the respective tables in the database, facilitating the interaction between the application and the data storage.

**Car Entity:** In the Car class, we used the **@Entity** annotation from the Java Persistence API to indicate that the Car class corresponds to a database table. To name this table, we used the **@Table** annotation and set the table name to **tb\_car** using the name parameter.

We defined a field named **idCar** of type Integer to hold the ID parameter of the Car class. To specify that this field is an ID, we used the **@Id** annotation from the Java Persistence API. Additionally, to enable automatic incrementation of the ID, we used the **@GeneratedValue** annotation with the strategy parameter set to **GenerationType.IDENTITY**. This way, MySQL's **AUTO\_INCREMENT** feature is utilized to automatically generate an incrementing value for the ID with each new record.

Furthermore, using the **@Column** annotation and its name parameter, we specified the column name in the database for our field. This ensures that the field in our class correctly maps to the corresponding column in the database table.

This configuration ensures that the Car entity is accurately mapped to the database table, with proper handling of the primary key and column names.

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu**Car.java file:**

In the Car class, the same annotations were used for the make, model, and description fields. By setting the nullable parameter of the **@Column** annotation to false, we ensured that these fields cannot be null. To prevent these fields from being left blank, we used the **@NotBlank** and **@NotEmpty** annotations, which provide a message to inform the user if these fields are not filled out.

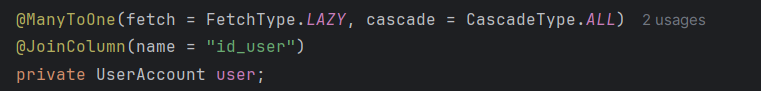
Additionally, to prevent the price field from having a negative value, we used the **@Positive** annotation. This annotation includes a message parameter that informs the user if a negative value is entered.

metin, ekran görüntüsü içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Database Relationships:** JPA annotations were used to establish relationships between database tables. A many-to-one relationship (one user can have many cars) was created between the userAccount table and another table. The **@ManyToOne** annotation was utilized for this purpose. The fetch parameter was set to **FetchType.LAZY**. This ensures that the associated object is loaded only when accessed, thus providing performance benefits.

This configuration helps in making database queries and memory usage more efficient since associated objects are loaded into memory only when needed. This enhances the overall performance of the application.



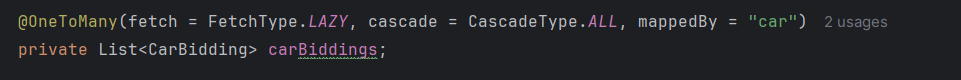
The **CascadeType.ALL** parameter has been provided to the cascade attribute. This ensures that all operations performed on this field are applied to the associated object as well.A join operation between related tables has been performed using the **@JoinColumn** annotation. The name parameter specifies the name of the column to which this field is associated.

Additionally, a **one-to-one relationship** has been established between the car table and the **carPicture** table using the @OneToOne annotation.The **FetchType.EAGER** parameter has been provided to the fetch attribute. This ensures that when the object is loaded, the associated object is also loaded immediately.The cascadeType parameter is again set to **CascadeType.ALL**.The mappedBy parameter is set to car, indicating that the car field of the CarPicture class is used.

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

A one-to-many relationship has been established between the **carBidding** table and another table using the **OneToMany** annotation. This relationship allows for multiple carBiddings to be associated with a single car.



metin, ekran görüntüsü içeren bir resim

Açıklama otomatik olarak oluşturulduA no-argument constructor, as well as getter and setter methods, have been defined for the Car class. This allows for object creation and access to and modification of the fields

**UserAccount Entity:** In the UserAccount class, user account information is stored. It has been transformed into a database table using the **@Entity** annotation. The **@Table** annotation's **name** parameter specifies the table name as **tb\_user**. The features mentioned earlier are similarly applied here.

The **@Size** annotation is used to define the minimum and maximum character counts for the **username** and **password** fields. The **message** parameter is used to inform the user if these character limits are not met.

A one-to-one relationship has been established with the **UserProfile** table using the **@OneToOne** annotation. Additionally, one-to-many relationships have been established with the **Role**, **Car**, and **CarBidding** tables using the **@OneToMany** annotation.

metin, ekran görüntüsü, yazılım, işletim sistemi içeren bir resim

Açıklama otomatik olarak oluşturuldu

Parameterized and no-argument constructors, as well as getter and setter methods, have been defined:

metin, ekran görüntüsü içeren bir resim

Açıklama otomatik olarak oluşturuldu

metin, ekran görüntüsü içeren bir resim

Açıklama otomatik olarak oluşturuldu

**CarBidding Entity:** A many-to-one relationship has been established with both the ‘**UserAccount’** and ‘**Car’** tables using the **@ManyToOne** annotation.

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

**CarPicture Entity**: A one-to-one relationship is established with the ‘**Car’** table using the **@OneToOne** annotation.

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

**UserProfile Entity:** In the UserProfile class, the user's personal information is stored. It is specified as a database table with the **@Entity** annotation. The **@Table** annotation's **name** parameter specifies the table name in the database corresponding to the class.

metin, ekran görüntüsü, yazılım içeren bir resim

Açıklama otomatik olarak oluşturuldu

A one-to-one relationship is established with the **UserAccount** and **ProfilePicture** tables. Both parameterized and no-argument constructors, as well as **getter** and **setter** methods, are defined.

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

**ProfilePicture Entity:** A **one-to-one** relationship is established with the **UserProfile** table.

**metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu**

**metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu**metin, ekran görüntüsü, yazılım içeren bir resim

Açıklama otomatik olarak oluşturuldu**Role Entity:** The Role class is responsible for storing user roles. A **many-to-one** relationship is established with the UserAccount table.

**metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturulduTestDrive Entity:** In the TestDrive class, the requests for test drives that users have scheduled are stored. The **pattern** parameter provided in the **DateTimeFormat** annotation ensures that the date entered by users is stored in the **year/month/day** format. **Many-to-one** relationships are established with the **UserAccount** and **Car** tables.

**metin, ekran görüntüsü, yazılım içeren bir resim

Açıklama otomatik olarak oluşturuldu**

**REPOSITORY LAYER**

The repository layer, also known as the **data access** layer, is responsible for performing database operations in the application.

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

For each entity, a repository interface has been created in the repository layer. Each interface extends **JpaRepository**. JpaRepository already contains many predefined methods for database operations, so classes extending it can use these methods (such as save, findById, findAll...).

When we need functionality beyond the methods provided by JpaRepository, we can write our own methods or create our own queries.

**CarRepository :** The CarRepository interface extends **JpaRepository**. JpaRepository is a parameterized interface. The first parameter represents the **entity** class on which database operations will be performed. In this repository, we provide our Car class as the entity. The second parameter is the data type of the id field of this class, which is int in the case of the Car class.

In addition to the default **CRUD** operations provided by JpaRepository, Car repository requires custom methods. These methods are defined using the **@Query** annotation with the **nativeQuery = true** parameter, allowing us to write native **SQL** queries for our methods.

In the **searchCar** method, the keyword provided as a parameter is searched in the **tb\_car** table for matches in the make, model, or year fields using **LIKE** operator.

The **searchCarByPriceRange** method also utilizes a native SQL query. It searches for records in the **tb\_car** table where the price column value is greater than or equal to the low value and less than or equal to the high value provided as parameters.

The **searchCarByKeywordAndPriceRange** method combines the functionalities of the previous two methods. It uses a native SQL query to retrieve records from the database that satisfy both the keyword search and the price range criteria.

The FeaturedCars method also employs a native SQL query to fetch three randomly selected records from the tb\_car table where the status is set to **ACTIVE**.

metin, ekran görüntüsü, yazılım içeren bir resim

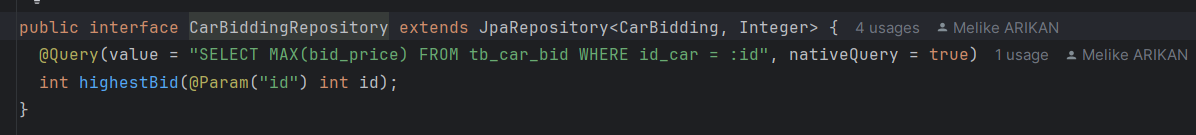
Açıklama otomatik olarak oluşturuldu

**UserRepository :** In addition to the **jpa** methods, the **findByUsername** method is defined in the user reposiory interface. In this method, jpa returns a **UserAccount** according to the username parameter.

**metin, ekran görüntüsü içeren bir resim

Açıklama otomatik olarak oluşturuldu**

**CarBiddingRepository :** In **CarBiddingRepository**, a method named **highestBid** is defined, and a **native SQL** query is used. This method shows the highest bid price made for the car with the specified id in the **tb\_car\_bid** table.

****

**RoleRepository:** In the Role repository, a method named **findByRole** is defined. This method searches the database based on the role parameter provided and returns the role.Similarly, the **CarPictureRepository**, **ProfilePictureRepository**, **TestDriveRepository**, and **UserProfileRepository** interfaces are extended from **JpaRepository**, and no additional methods beyond **JpaRepository** methods are defined.

**metin, yazı tipi, ekran görüntüsü içeren bir resim

Açıklama otomatik olarak oluşturuldu**

**CONTROLLER LAYER**

The controller layer in a web application's backend is responsible for handling incoming user requests. It receives HTTP requests, processes them, and returns the appropriate responses. The controllers typically contain methods that match specific URL routes, and these methods handle the requests.

Controllers do not usually implement business logic; instead, they delegate tasks to appropriate service layers and perform necessary data processing. Controllers often validate incoming data, invoke business logic services, generate results, and send HTTP responses back to clients.

In summary, the controller layer acts as an intermediary that processes incoming requests and generates appropriate responses for a web application.

**CarController** : The CarController class is annotated with **@Controller**, indicating to Spring that this class is a controller. Additionally, it maps requests to the "**/cars**" URL path.The **CarController** class injects two services (**CarService** and **UserCarService**) automatically using **@Autowired.** These services are used to support the business logic of this controller.

Here are some significant methods in this controller class:

* **carPage**: This method handles GET requests to the "/cars" URL. It retrieves all cars using **CarService** and adds them to a model object. Then, it returns a template named "cars" to display the list of cars.
* **carDetails**: This method handles GET requests to the "/cars/{make}/{model}/{year}/{id\_car}" URL. It retrieves the car based on the given car ID (**id\_car**) using **CarService** and adds it to the model object. Additionally, it retrieves the highest bidding amount using **UserCarService** and adds it to the model object.
* **searchCar**: This method handles GET requests to the "/cars" URL with a **keyword** parameter for searching cars. It searches for cars using the **keyword** parameter with **CarService** and adds the results to a model object.
* **searchCarByPriceRange**: This method handles GET requests to the "/cars" URL with **low** and **high** parameters for searching cars within a price range. It searches for cars within the specified price range using **CarService** and adds the results to a model object.
* **listCarsSortedByPriceAsc** **and** **listCarsSortedByPriceDesc**: These methods are used to sort cars by price in ascending or descending order. They sort the cars using **CarService** and add the results to a model object.

This controller class receives requests, processes them, retrieves necessary data, and forwards the results to appropriate templates for rendering user interface pages related to cars in the web application.

**metin, ekran görüntüsü içeren bir resim

Açıklama otomatik olarak oluşturuldu**metin, ekran görüntüsü içeren bir resim

Açıklama otomatik olarak oluşturuldu

**LoginController** : The LoginController class defines a controller class that handles GET requests to the "**/login**" URL path. When a user accesses the login page, this method is triggered. Inside the method, a new **UserAccount** object is created, potentially representing user login credentials. This object is then added to the model with the attribute name "**user**". Finally, the method returns "**login**", indicating that the associated "**login**" template will be rendered. Overall, this controller serves the login page to users accessing the "**/login**" URL path.

metin, ekran görüntüsü, yazılım, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu Note: We wrote other controllers in similar ways, you can find their codes on Github.

**SERVICE LAYER**

The service layer in a software application manages the business logic and operations. It includes functions such as business logic implementation, data processing, external integrations, and transaction management. This layer interacts with the controller layer (which handles web requests) and the repository layer (which communicates with the database). By separating concerns, service layers promote modularity, maintainability, and testability in code. Keeping business logic separate from data access improves code organization and maintenance.

**CarService** : The CarService interface contains methods for performing various operations related to cars. Here's a brief explanation of each method:

* **getCarById(int id):** Retrieves a car by its unique identifier.
* **listCar():** Retrieves a list of all cars.
* **searchCar(String keyword):** Searches for cars based on a given keyword.
* **searchCarByPriceRange(int low, int high):** Searches for cars within a specified price range.
* **searchCarByKeywordAndPriceRange(String keyword, int low, int high):** Searches for cars based on a keyword and within a specified price range.
* **featuredCars():** Retrieves a list of featured cars.
* **findAllByOrderByPriceAsc():** Retrieves all cars ordered by price in ascending order.
* **findAllByOrderByPriceDesc():** Retrieves all cars ordered by price in descending order.

This interface serves as a contract for classes that provide car-related services. Implementing classes will define the actual logic for these methods, enabling the application to perform operations on cars.

**metin, ekran görüntüsü, yazılım içeren bir resim

Açıklama otomatik olarak oluşturuldu**

**UserService :** This interface declares methods for performing various operations related to users. Here's a brief explanation of each method:

* **saveUser(UserAccount user, UserProfile profile):** Saves a user account and its associated profile.
* **findByUsername(String username):** Finds a user account by username.
* **getUserLogin():** Retrieves the currently logged-in user account.
* **saveImage(MultipartFile file, UserProfile profile) throws Exception:** Saves a profile image. MultipartFile represents the uploaded file.
* **editUserProfile(UserProfile profile):** Edits a user profile.
* **getProfile(int idProfile):** Retrieves a user profile by a specific profile ID.

This interface abstracts the functionality required to perform user-related operations. Implementing classes will provide the actual logic for these methods, allowing the application to interact with users effectively.

**metin, ekran görüntüsü, yazı tipi, yazılım içeren bir resim

Açıklama otomatik olarak oluşturuldu**

**Note:** We wrote other services in a similar way. You can access the codes on Github.

metin, yazı tipi, sayı, numara, ekran görüntüsü içeren bir resim

Açıklama otomatik olarak oluşturuldu**DATABASE**

The tables we use in the database are:

* tb\_cars
* tb\_car\_bid
* tb\_car\_picture
* tb\_profile\_picture
* tb\_role
* tb\_test\_drive
* tb\_user
* tb\_user\_profile

**Images of the 'column' and 'related tables' information of the tables taken from the database :**

metin, ekran görüntüsü, yazı tipi, tasarım içeren bir resim

Açıklama otomatik olarak oluşturuldu

**metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu**

**metin, ekran görüntüsü, yazı tipi, makbuz içeren bir resim

Açıklama otomatik olarak oluşturuldumetin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu**