



# Universidad **Mariana**

LUXURY COLLECTION

MILTON DAVID DAZA DAZA

SAMUEL FELIPE QUINTERO

JUAN JOSE PORTILLA

JULIAN DAVID ROSALES PORTILLA

Universidad Mariana

Programa de Ingeniería de Sistemas

Cuarto Semestre

San Juan de Pasto

20 de mayo de 2024

LUXURY COLLECTION

MILTON DAVID DAZA DAZA

SAMUEL FELIPE QUINTERO

JUAN JOSE PORTILLA

JULIAN DAVID ROSALES PORTILLA

Docente

WILSON ANDRES CASTILLO CASTRO

Universidad Mariana

Programa de Ingeniería de Sistemas

Cuarto Semestre

San Juan de Pasto

20 de mayo de 2024

## Introducción

"LUXURY COLLECTION" es una aplicación web diseñada para gestionar una colección de autos de lujo, deportivos y clásicos. La plataforma permite a los usuarios registrar, visualizar y gestionar autos, así como actualizar sus credenciales. La aplicación está diseñada para proporcionar una experiencia de usuario fluida y estética, utilizando HTML, CSS y JavaScript. Los datos de los autos y los usuarios se almacenan en localStorage para facilitar la persistencia de datos en el navegador del usuario.

### Estructura del Proyecto:

El proyecto está compuesto por varias páginas HTML, hojas de estilo CSS y scripts JavaScript para manejar la lógica de la aplicación. A continuación, se describe la estructura de los archivos principales:

#### HTML:

- index.html: Página principal de la aplicación.
- login.html: Página de inicio de sesión.
- registro.html: Página de registro de nuevos usuarios.
- agregarAuto.html: Formulario para agregar nuevos autos a la colección.
- mostrarColeccion.html: Página para mostrar la colección de autos.
- actualizar.html: Página para actualizar las credenciales de los usuarios.

#### CSS:

- style.css: Hoja de estilos principal para la aplicación.
- style2.css: Hoja de estilos adicional para páginas específicas.

#### JavaScript:

- login.js: Maneja la lógica de inicio de sesión y registro.
- agregarAuto.js: Maneja la lógica para agregar nuevos autos y previsualizar imágenes.

- `mostrarColeccion.js`: Maneja la lógica para mostrar la colección de autos y realizar operaciones como eliminar y agregar a favoritos.
- `actualizar.js`: Maneja la lógica para actualizar las credenciales de los usuarios.

#### Funcionalidades:

##### Inicio de Sesión y Registro

- Los usuarios pueden registrarse proporcionando su correo electrónico y una contraseña.
- Los datos de los usuarios se almacenan en `localStorage`.
- El inicio de sesión verifica las credenciales ingresadas contra las almacenadas en `localStorage`.

##### Agregar Autos

- Los usuarios pueden agregar nuevos autos a la colección mediante un formulario.
- El formulario incluye campos para la marca, modelo, motor, año, cilindraje y una imagen del auto.
- La imagen seleccionada se previsualiza antes de ser agregada.
- Los datos del auto se almacenan en `localStorage`.

##### Mostrar Colección

- Los autos agregados se muestran en una vista de tarjetas.
- Cada tarjeta muestra la imagen del auto, junto con sus detalles (marca, modelo, motor, año, cilindraje).
- Los usuarios pueden eliminar autos de la colección o agregarlos a favoritos.

##### Actualizar Credenciales

- Los usuarios pueden actualizar su correo electrónico y contraseña.
- La actualización se realiza verificando la existencia del usuario y almacenando los nuevos datos en `localStorage`.

##### Detalles Técnicos

###### 1. HTML

- Uso de elementos semánticos para una estructura clara y accesible.

- Formularios para la entrada de datos y gestión de autos.

## 2. CSS

- Diseño responsivo para asegurar una buena experiencia de usuario en dispositivos móviles y de escritorio.
- Uso de clases CSS para estilizar los elementos y mejorar la apariencia visual de la aplicación.

## 3. JavaScript

- `localStorage` para almacenar datos de usuarios y autos de manera persistente en el navegador.
- Funciones para manejar la lógica de validación, almacenamiento y manipulación de datos.
- Eventos DOM para manejar interacciones del usuario, como clics en botones y cambios en formularios.

## 5. Mejoras Futuras

### 1. Autenticación y Seguridad

- Implementar una autenticación más segura utilizando sesiones o tokens.
- Encriptar contraseñas antes de almacenarlas en `localStorage`.

### 2. Base de Datos

- Migrar el almacenamiento de datos a una base de datos real (SQL o NoSQL) para mejorar la escalabilidad y seguridad.

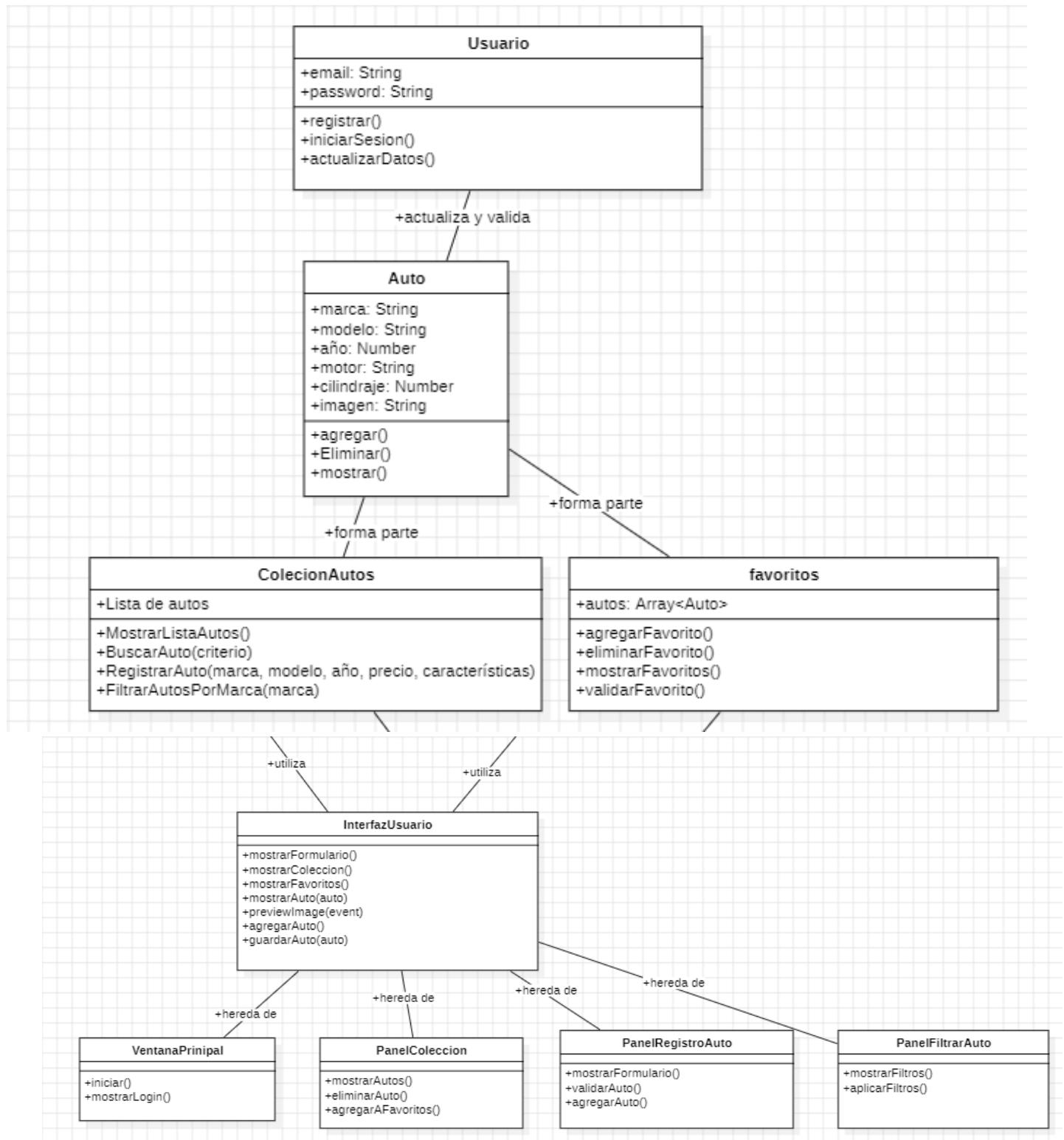
### 3. Interfaz de Usuario

- Mejorar la interfaz de usuario con bibliotecas modernas de UI como React o Vue.js.
- Agregar animaciones y transiciones para una experiencia más dinámica.

### 4. Funcionalidades Adicionales

- Implementar filtros avanzados para buscar autos por diferentes criterios.
- Añadir la capacidad de comparar autos en detalle.
- Implementar una sección de comentarios y valoraciones para los autos.

Diagrama de clases:



## REQUERIMIENTOS FUNCIONALES:

Requisito	Descripción	Entradas	Salidas
R1 - Agregar Auto	El usuario ingresa la información de un nuevo auto y lo agrega a la colección.	Marca, Modelo, Motor, Año, Cilindraje, Imagen (opcional)	El auto se agrega a la colección.
R2 - Buscar Autos	El usuario busca autos por marca y modelo.	Marca, Modelo	Lista de autos que coinciden con la búsqueda.
R3 - Agregar a Favoritos	El usuario agrega un auto de la colección a su lista de favoritos.	Auto a agregar a favoritos	El auto se agrega a la lista de favoritos.
R4 - Ver Favoritos	Se muestran todos los autos almacenados en la lista de favoritos.	No hay.	Lista de autos favoritos.
R5 - Eliminar Auto	Permitir al usuario eliminar un auto de la colección.	Selección del auto a eliminar	El auto seleccionado se elimina correctamente.
R6 - Comparación de Autos	Permitir a los usuarios comparar varios autos.	Selección de autos para comparar	Se muestra una tabla o vista comparativa de los autos.

## Elementos del paradigma de Programación Orientada a Objetos:

Elemento	Descripción
<b>Clases</b>	Las clases podrían representar entidades como Auto, Usuario y posiblemente ColeccionAutos. Cada una de estas clases define las características y comportamientos asociados a su respectivo concepto en el sistema.
<b>Objetos</b>	Los objetos son instancias de las clases mencionadas anteriormente. Por ejemplo, cada auto individual en la colección sería un objeto de la clase Auto.
<b>Atributos</b>	Los atributos de las clases podrían ser las propiedades de cada objeto. Por ejemplo, un objeto de la clase Auto podría tener atributos como marca, modelo, motor, año, cilindraje, etc.
<b>Métodos</b>	Los métodos de las clases representarían las acciones que pueden realizar los objetos. Por ejemplo, un método agregarAuto() podría añadir un nuevo auto a la colección, mientras que un método eliminarAuto() podría eliminar un auto existente.
<b>Encapsulación</b>	Aunque JavaScript no maneja la encapsulación de manera tan estricta como otros lenguajes, se pueden utilizar prácticas como el uso de variables privadas y métodos accesibles solo internamente en una clase para lograrla.
<b>Herencia</b>	La herencia permite definir una jerarquía de clases donde las subclases heredan

	comportamientos y atributos de sus superclases. En el proyecto, podría haber una jerarquía de clases como AutoDeportivo y AutoClasico que heredan de la clase base Auto.
<b>Polimorfismo</b>	El polimorfismo permite que diferentes clases respondan al mismo mensaje de manera diferente. Por ejemplo, podrías tener un método mostrarDetalles() en la clase base Auto, que es implementado de manera distinta en las subclases AutoDeportivo y AutoClasico.
<b>Abstracción</b>	La abstracción implica identificar las características esenciales de un objeto y omitir los detalles irrelevantes. En el proyecto, las clases y los objetos representan una abstracción de los autos y usuarios del mundo real, simplificando su representación para su gestión en la aplicación web.

Servlets, manejo de métodos de petición HTTP (GET y POST):

En los scripts que se utilizan, no se manejan directamente los servlets ni se realizan peticiones HTTP GET o POST al servidor. Estos scripts se centran en la manipulación del DOM, el manejo de eventos del usuario, la interacción con el almacenamiento local (como localStorage), la validación de formularios y otras operaciones relacionadas con la interfaz de usuario en el navegador web.

Implementación de List, arrayList o LinkedList:

En los scripts, no se hace explícitamente una implementación de List, ArrayList o LinkedList. Sin embargo, en la manipulación de datos de autos, usuarios y favoritos, se utilizan estructuras de datos similares a las listas para almacenar y manipular los objetos correspondientes.

Por ejemplo, en el código se usa localStorage para almacenar y recuperar datos en forma de arrays de objetos JSON. Estos arrays de objetos podrían considerarse como listas de elementos, donde cada elemento representa un auto, un usuario o un favorito.

```

1  var autos = JSON.parse(localStorage.getItem('autos')) || [];
2

```

En este caso, autos es un array que actúa como una lista de autos. Se utiliza JSON para convertir los datos almacenados en localStorage en un formato que se puede manejar fácilmente en JavaScript.

Para agregar un nuevo auto a esta lista:

```

1  autos.push(auto);
2  localStorage.setItem('autos', JSON.stringify(autos));

```

Aquí, auto es un objeto que representa un auto, y se agrega al final de la lista de autos (autos). Luego, se actualiza el



localStorage para reflejar los cambios.


Aunque no se utilicen explícitamente las clases List, ArrayList o LinkedList proporcionadas por lenguajes como Java, el concepto de almacenar y manipular datos en forma de lista se refleja en el código a través del uso de arrays y objetos en JavaScript.

Recorrido simple y doble:

En los scripts, se realizan recorridos simples y dobles sobre los arrays de objetos que representan los autos, usuarios y favoritos almacenados en localStorage.

Un recorrido simple se lleva a cabo cuando se itera sobre un array de elementos en una sola dirección, normalmente de principio a fin o de fin a principio.

Por ejemplo, en el código se itera sobre el array de autos para mostrar cada uno de ellos en la página:




```
1  autos.forEach(function(auto, index) {  
2      // Código para mostrar cada auto en la página  
3  });
```

Aquí, forEach itera sobre cada elemento del array autos, ejecutando la función proporcionada para cada uno de ellos en orden secuencial, desde el primer elemento hasta el último.

Por otro lado, un recorrido doble implica iterar sobre un array de elementos en ambas direcciones, hacia adelante y hacia atrás. En JavaScript, no hay una función integrada para recorridos dobles como en otros lenguajes de programación, pero podrías lograr un recorrido doble escribiendo bucles for anidados o utilizando índices para recorrer el array en ambas direcciones.

Por ejemplo, podrías hacer un recorrido doble para comparar cada par de elementos en el array de autos:



```
1  for (var i = 0; i < autos.length; i++) {  
2      for (var j = i + 1; j < autos.length; j++) {  
3          // Código para comparar autos[i] con autos[j]  
4      }  
5  }
```

Aquí,

el

primer bucle itera sobre todos los elementos del array, mientras que el segundo bucle itera sobre los elementos restantes a partir del siguiente elemento al actual ( $i + 1$ ). De esta manera, se comparan todos los pares de elementos en el array.