

Universidad Tecnológica Metropolitana



Estructuras de Datos Aplicadas

ISC. Ruth Betsaida Martínez Domínguez, MGTI

Práctica 10-12

- Soberanis Acosta Jimena Monserrat
- Tzec Cauich Alejandra de Jesus

Desarrollo de Software Multiplataforma

Cuarto Cuatrimestre

4°B

Parcial II

Viernes, 27 de octubre de 2023

## PRACTICA 10

Creamos la estructura de la página web utilizando **HTML**. Simula una ventanilla de banco. Incluye botones para agregar y atender clientes, una tabla para mostrar la información de los clientes en espera, un modal para ingresar datos de cliente y otro modal para mostrar detalles del cliente seleccionado. El archivo JavaScript "script.js" proporcionará la funcionalidad necesaria para administrar la cola de clientes y la interacción con el empleado de banco.

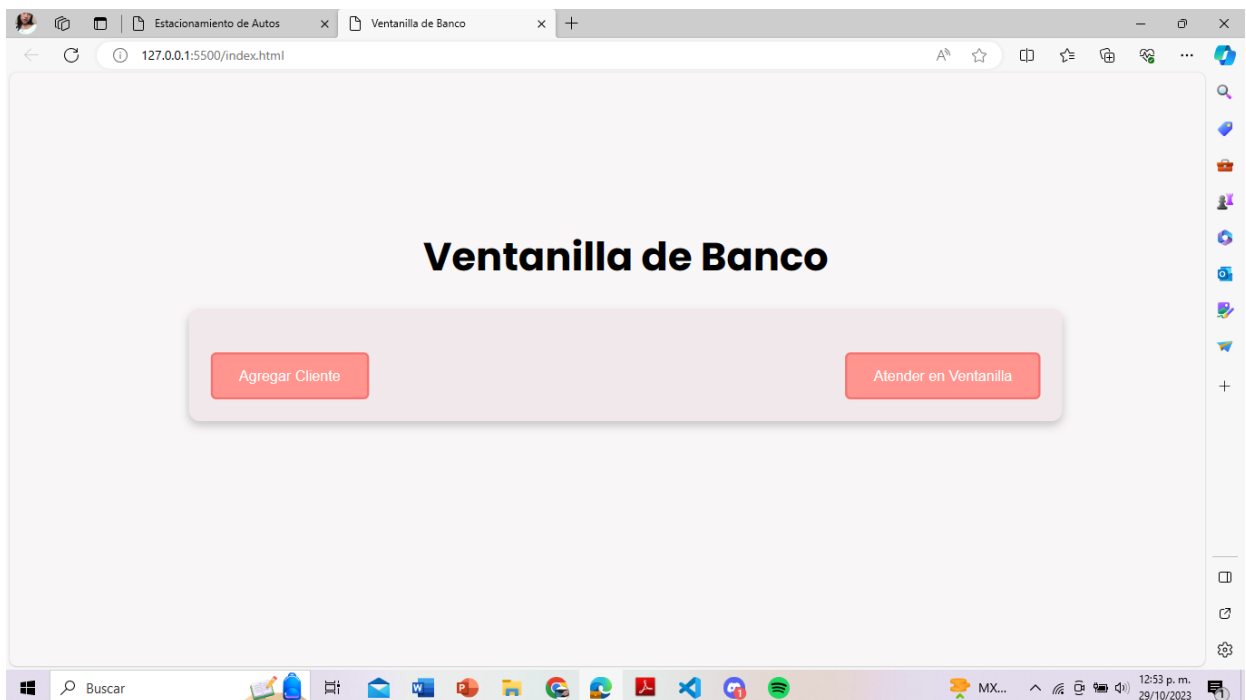
```
index.html x
index.html > html > body > dialog#modal-agregar-cliente > form > select#tipo-movimiento
1 <!DOCTYPE html>
2 <html lang="es">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <link rel="stylesheet" href="style.css">
7   <title>Ventanilla de Banco</title>
8 </head>
9 <body>
10  <h1>Ventanilla de Banco</h1>
11
12  <div class="contenedor-cola">
13    <div class="contenedor-button">
14      <button id="agregar-cliente-btn">Agregar Cliente</button>
15      <button id="atender-cliente-btn">Atender en Ventanilla</button>
16    </div>
17    <table id="tabla-clientes">
18      <thead>
19        <tr>
20          <th>Número de Turno</th>
21          <th>Nombre del Cliente</th>
22          <th>Tipo de Movimiento</th>
23          <th>Hora de Llegada</th>
24        </tr>
25      </thead>
26      <tbody>
27      </tbody>
28    </table>
29  </div>
30
```

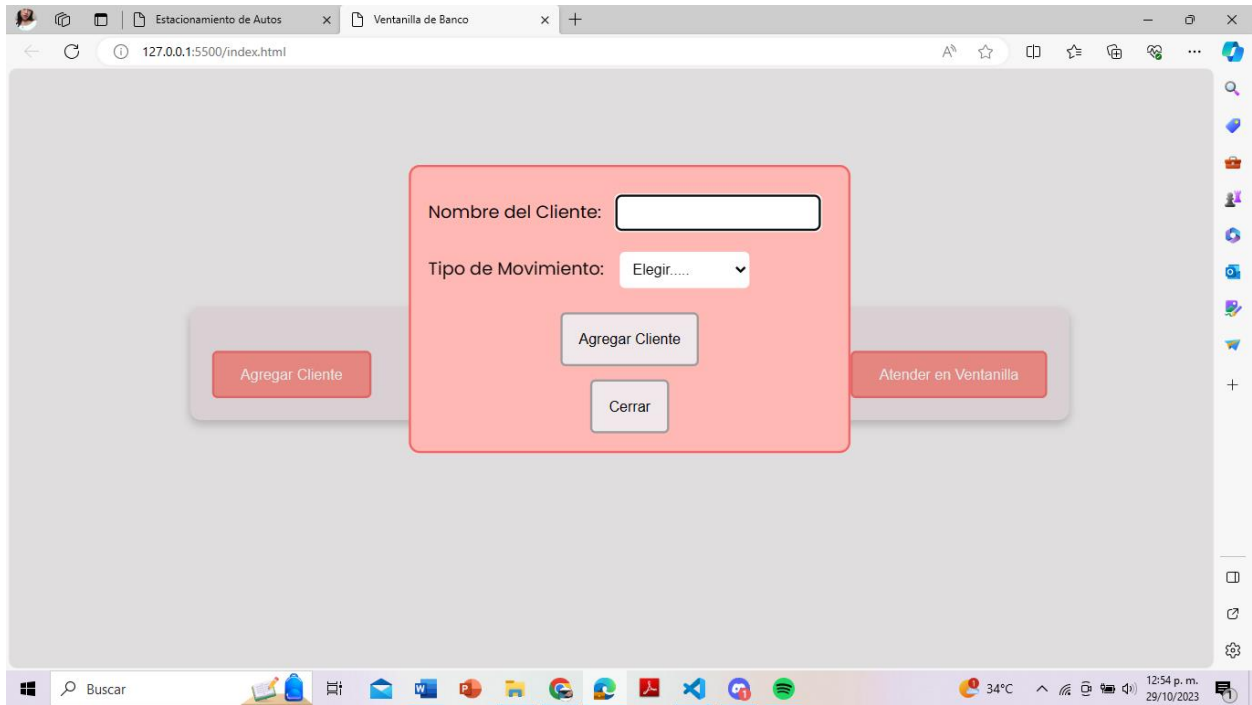
```

30
31     <!-- Modal para ingresar datos del cliente -->
32     <dialog id="modal-agregar-cliente">
33         <form>
34             <label for="nombre-cliente">Nombre del Cliente:</label>
35             <input type="text" id="nombre-cliente" required>
36             <br>
37             <label for="tipo-movimiento">Tipo de Movimiento:</label>
38             <select id="tipo-movimiento" required>
39                 <option value="Depósito">Elegir.....</option>
40                 <option value="Depósito">Depósito</option>
41                 <option value="Retiro">Retiro</option>
42                 <option value="Transferencia">Transferencia</option>
43                 <option value="Consulta">Consulta</option>
44             </select>
45             <br>
46             <button id="enviar-cliente">Agregar Cliente</button>
47             <button id="cerrar-modal">Cerrar</button>
48         </form>
49     </dialog>
50
51     <div id="dialogo" class="dialogo">
52         <div class="contenido-dialogo">
53             <h2>Datos del Cliente</h2>
54             <p id="informacion-cliente"></p>
55             <button id="cerrar-dialogo">Cerrar</button>
56         </div>
57     </div>
58
59     <script type="module" src="script.js"></script>
60 </body>
61 </html>
62

```

Este es la forma en la que se ve el maquetado con estilo de la página.





Creamos el primer archivo llamado **ColaBancaria.js**. dentro de este archivo está la clase **ColaBancaria** gestiona una cola de clientes en una ventanilla de banco. Puede insertar nuevos clientes en la cola y atender al cliente en la parte delantera. Si la cola está llena, muestra una alerta, y si no hay clientes en la cola, también muestra una alerta. Además, registra el tiempo de espera de un cliente atendido y lo muestra en la alerta.

```

JS ColaBancaria.js X
JS ColaBancaria.js > ...
1  export default class ColaBancaria
2  {
3      constructor(tamanoMaximo)
4      {
5          this.cola = [];
6          this.tamanoMaximo = tamanoMaximo;
7      }
8
9      insertarCliente(turno, nombre, tipo, horaLlegada)
10     {
11         if (this.colaLlena())
12         {
13             alert("La cola está llena. No se pueden agregar más clientes.");
14             return;
15         }
16
17         this.cola.push({ turno, nombre, tipo, horaLlegada });
18     }
19
20     atenderCliente()
21     {
22         if (this.colaVacia())
23         {
24             alert("No hay clientes en la cola.");
25             return null;
26         }
27
28         const clienteAtendido = this.cola.shift();
29         const horaActual = new Date();
30         const tiempoEspera = (horaActual - clienteAtendido.horaLlegada) / 1000;
31         alert(`Cliente ${clienteAtendido.nombre} atendido. Tiempo de espera: ${tiempoEspera.toFixed(2)} segundos.`);
32         return clienteAtendido;

```

En la misma clase se creamos los métodos **colaVacia()** y **colaLlena()** permiten verificar si la cola de clientes está vacía o llena, respectivamente. El primero devuelve true si no hay clientes en la cola, y el segundo devuelve true cuando la cola ha alcanzado su capacidad máxima. Estas funciones son útiles para controlar el flujo de clientes en la ventanilla de banco y mostrar alertas cuando sea necesario.

```

35     colaVacia()
36     {
37         return this.cola.length === 0;
38     }
39
40     colaLlena()
41     {
42         return this.cola.length >= this.tamanoMaximo;
43     }
44 }
45

```

Luego creamos el segundo archivo llamado **Interfaz.js**. Dentro del archivo creamos funciones que permiten mostrar y ocultar un modal que se utiliza para agregar clientes en

la ventanilla de banco. La función **"abrirModalAgregarCliente"** muestra el modal, y la función **"cerrarModalAgregarCliente"** lo cierra, brindando una interacción amigable con el usuario en la página web.

```
JS Interfaz.js x
JS Interfaz.js > colaBancaria.colas.forEach() callback
1 export function abrirModalAgregarCliente()
2 {
3   const modalAgregarCliente = document.getElementById('modal-agregar-cliente');
4   modalAgregarCliente.showModal();
5 }
6
7 export function cerrarModalAgregarCliente()
8 {
9   const modalAgregarCliente = document.getElementById('modal-agregar-cliente');
10  modalAgregarCliente.close();
11 }
12
```

Dentro del mismo archivo están otras funciones que permiten agregar clientes a la cola de la ventanilla de banco, mostrar información detallada sobre los clientes y cerrar el diálogo. La primera función recopila datos del nuevo cliente, los agrega a la cola, y muestra información del cliente en un diálogo. La segunda función cierra el diálogo que muestra la información del cliente en la página. Estas funciones son esenciales para la interacción del usuario en la aplicación de ventanilla de banco.

```
12
13 export function enviarCliente(colasBancarias, entradaNombreCliente, seleccionTipoMovimiento)
14 {
15   if (colasBancarias.colasLlena())
16   {
17     alert("La cola está llena. No se pueden agregar más clientes.");
18     return;
19   }
20
21   const turno = colasBancarias.colas.length + 1;
22   const nombre = entradaNombreCliente.value;
23   const tipo = seleccionTipoMovimiento.value;
24   const horaLlegada = new Date();
25   colasBancarias.insertarCliente(turno, nombre, tipo, horaLlegada);
26   actualizarTablaCola(colasBancarias);
27   cerrarModalAgregarCliente();
28
29   entradaNombreCliente.value = '';
30   seleccionTipoMovimiento.value = 'Depósito';
31
32   const informacionCliente = `Número de Turno: ${turno}, Nombre del Cliente: ${nombre}, Tipo de Movimiento: ${tipo}, Hora
33   document.getElementById('informacion-cliente').textContent = informacionCliente;
34   document.getElementById('dialogo').style.display = 'block';
35 }
36
37 export function cerrarDialogo()
38 {
39   document.getElementById('dialogo').style.display = 'none';
40 }
41
```

Las funciones permiten atender al cliente en la ventanilla de banco y actualizar la tabla que muestra la cola de clientes. La función **"atenderCliente"** extrae al cliente que está siendo atendido, y si hay un cliente para atender, actualiza la tabla. La función

"**actualizarTablaCola**" borra y vuelve a llenar la tabla con información de la cola de clientes, mostrando los datos actualizados en la página. Estas funciones son esenciales para el funcionamiento y la visualización de la cola de clientes en la aplicación de ventanilla de banco.

```
41
42 export function atenderCliente(colaBancaria)
43 {
44   const clienteAtendido = colaBancaria.atenderCliente();
45   if (clienteAtendido)
46   {
47     actualizarTablaCola(colaBancaria);
48   }
49 }
50
51 export function actualizarTablaCola(colaBancaria)
52 {
53   const cuerpoTabla = document.querySelector('tbody');
54   cuerpoTabla.innerHTML = '';
55
56   colaBancaria.cola.forEach(cliente =>
57   {
58     const fila = cuerpoTabla.insertRow();
59     const celda1 = fila.insertCell(0);
60     const celda2 = fila.insertCell(1);
61     const celda3 = fila.insertCell(2);
62     const celda4 = fila.insertCell(3);
63     celda1.innerHTML = cliente.turno;
64     celda2.innerHTML = cliente.nombre;
65     celda3.innerHTML = cliente.tipo;
66     celda4.innerHTML = cliente.horaLlegada.toLocaleTimeString();
67   });
68
69   document.getElementById('tabla-clientes').style.display = 'table';
70 }
```

Por ultimo, se está el archivo **script.js**. Este archivo importa módulos para gestionar la cola de clientes y la interfaz de usuario. Luego, establece constantes y crea una instancia de la cola bancaria. También, obtiene referencias a elementos HTML y agrega oyentes de eventos para mostrar y cerrar un modal que permite ingresar datos de clientes en la ventanilla de banco. Esta interacción forma parte de una aplicación de simulación de una ventanilla de banco en la página web.

```
JS ColaBancaria.js JS Interfaz.js JS script.js X
JS script.js > addEventListener('click') callback
1
2 import ColaBancaria from './ColaBancaria.js';
3 import * as UI from './Interfaz.js';
4
5 const TAMANO_MAXIMO_COLA = 10;
6 const colaBancaria = new ColaBancaria(TAMANO_MAXIMO_COLA);
7
8 const botonAgregarCliente = document.getElementById('agregar-cliente-btn');
9 const modalAgregarCliente = document.getElementById('modal-agregar-cliente');
10 const entradaNombreCliente = document.getElementById('nombre-cliente');
11 const seleccionTipoMovimiento = document.getElementById('tipo-movimiento');
12
13 botonAgregarCliente.addEventListener('click', () =>
14 {
15     modalAgregarCliente.showModal();
16 });
17
18 document.getElementById('cerrar-modal').addEventListener('click', () =>
19 {
20     modalAgregarCliente.close();
21 });
22
```



El código define eventos que responden a las acciones del usuario en la simulación de la ventanilla de banco. Cuando se hace clic en el botón "Agregar Cliente," se recopilan datos del cliente, se actualiza la cola de clientes y se muestra información en un diálogo. Al hacer clic en el botón "Cerrar" dentro del diálogo, este se oculta. Estas interacciones permiten la gestión de la cola de clientes y la visualización de la información del cliente en la aplicación de ventanilla de banco.

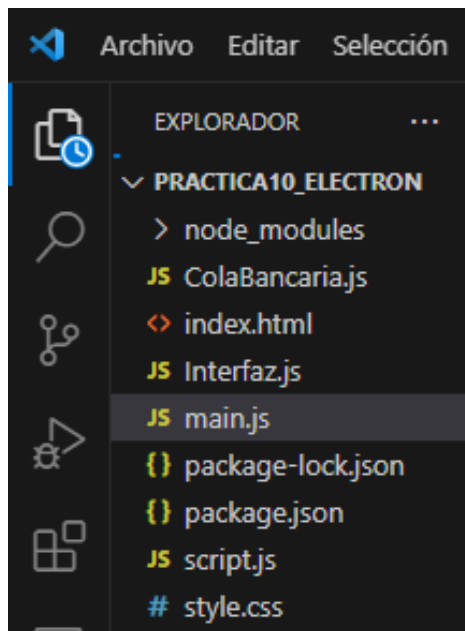
```
23 document.getElementById('enviar-cliente').addEventListener('click', () =>
24 {
25     if (colaBancaria.colaLlena()) {
26         alert("La cola está llena. No se pueden agregar más clientes.");
27         return;
28     }
29
30     const turno = colaBancaria.cola.length + 1;
31     const nombre = entradaNombreCliente.value;
32     const tipo = seleccionTipoMovimiento.value;
33     const horaLlegada = new Date();
34     colaBancaria.insertarCliente(turno, nombre, tipo, horaLlegada);
35     UI.actualizarTablaCola(colaBancaria);
36     modalAgregarCliente.close();
37     entradaNombreCliente.value = '';
38     seleccionTipoMovimiento.value = 'Depósito';
39
40     const informacionCliente = `Número de Turno: ${turno}<br>Nombre del Cliente: ${nombre}<br>Tipo de Movimiento: ${tipo}<br>Hora de Llegada: ${horaLlegada.toISOString().slice(0, 10)}<br>Hora de Salida: ${horaLlegada.toISOString().slice(0, 10)}<br>`;
41     document.getElementById('informacion-cliente').innerHTML = informacionCliente;
42     document.getElementById('dialogo').style.display = 'block';
43 });
44
45 document.getElementById('cerrar-dialogo').addEventListener('click', () =>
46 {
47     document.getElementById('dialogo').style.display = 'none';
48 });
49
```

Por último, cree una constante que al usuario atender al cliente de la ventanilla de banco cuando hace clic en el botón correspondiente. Si hay un cliente en la cola, se le atiende y se actualiza la tabla de la cola en la página.

```
49
50 const botonAtenderCliente = document.getElementById('atender-cliente-btn');
51 botonAtenderCliente.addEventListener('click', () => {
52     const clienteAtendido = colaBancaria.atenderCliente();
53     if (clienteAtendido) {
54         UI.actualizarTablaCola(colaBancaria);
55     }
56 });
57
```

Esta página web lo convertirá a una aplicación de escritorio, con un framework llamado **electrón**.

Desde mi cd de sistemas realicé todos los pasos para poder convertirlo al igual que el visual tuve que crear archivos.

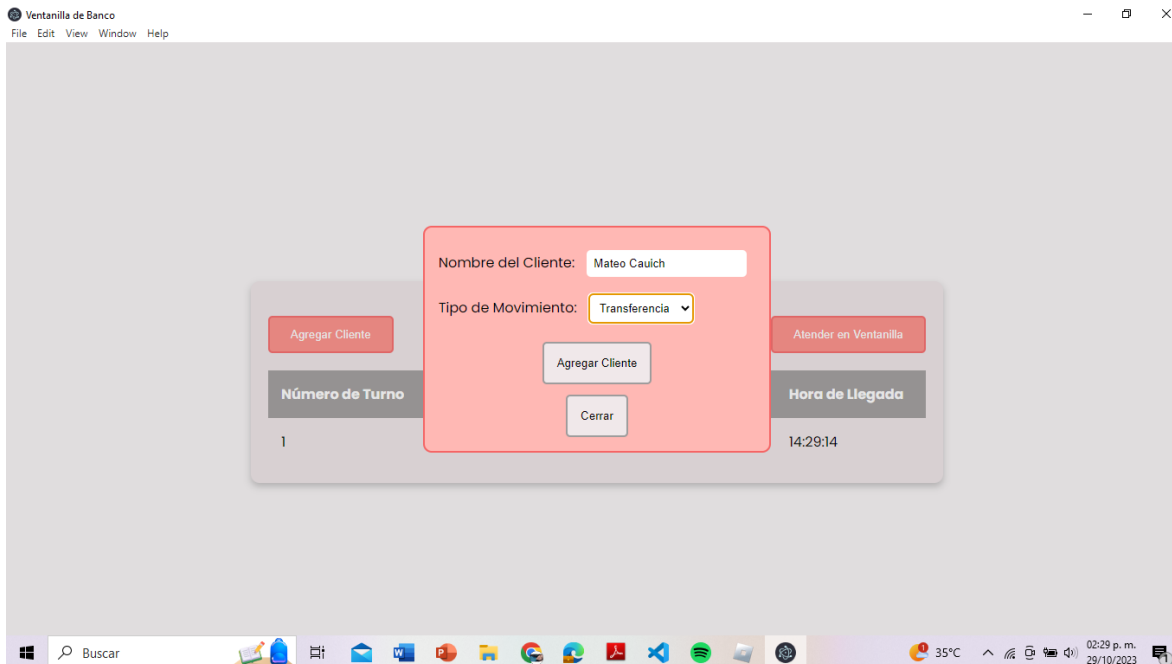


Para poder abrir la aplicación tenemos que ejecutar el siguiente comando **npm start** desde la consola de **visual studio code**.

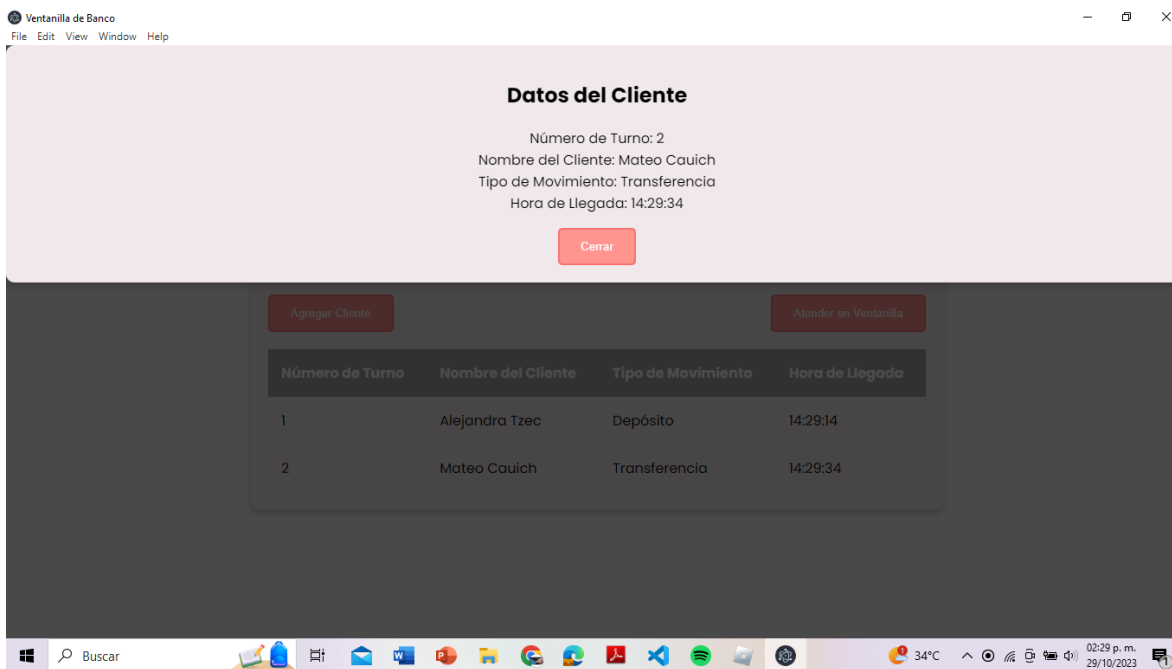
```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS
PS C:\Users\Personal\Desktop\TSU_Desarrollo_de_software\CUATRIMESTRE_4\Estructura De Datos Aplicadas\PARCIAL 2\Practica 10_12\Practica10_Electron> npm start
> index@1.0.0 start
> electron .
```

## Visualizamos la funcionalidad del programa

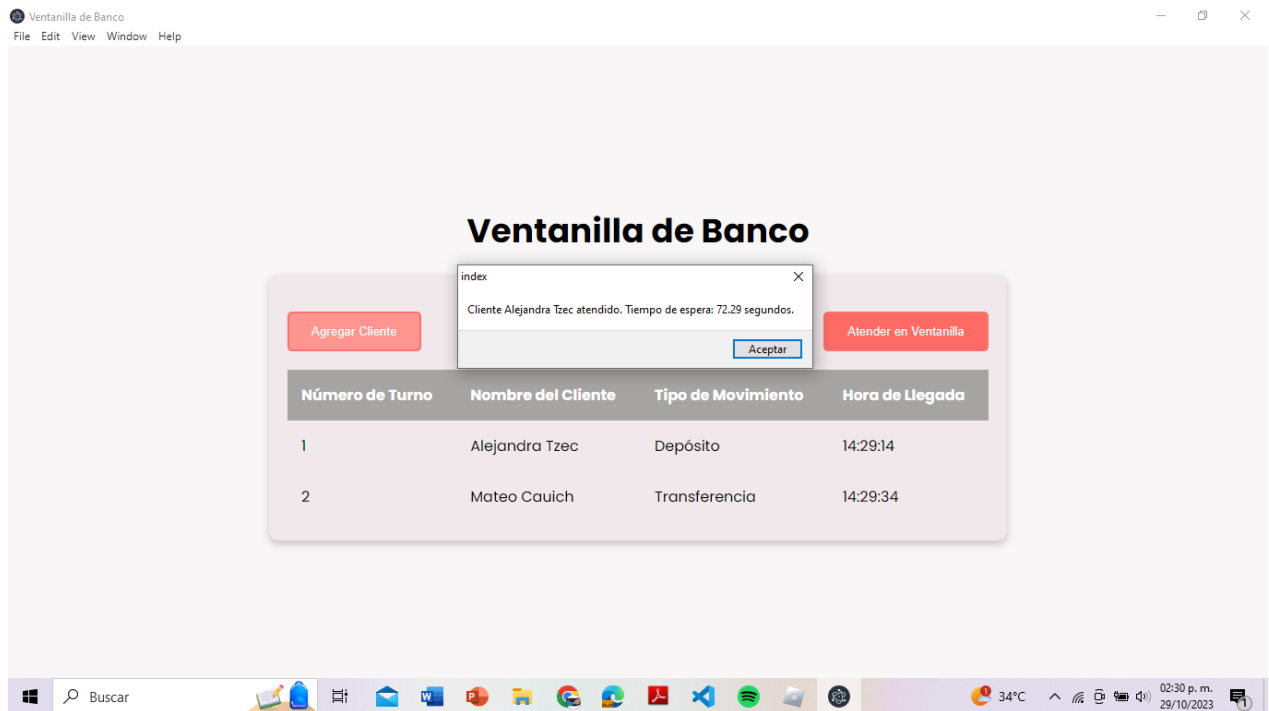
Llenamos los datos para poder agregar a un cliente a la cola.



Nos sale una ventana en donde nos proporciona los datos del cliente y nos dice que turno le toco y su hora de llegada.



Vemos que en la tabla se agregó el cliente 2 y cuando atendemos al cliente nos sale una ventana en donde nos dice cual cliente ha sido atendido.



## PRÁCTICA 11

Comenzamos con esta práctica con la estructura de mi HTML

```
practica11 > <> index.html > ...
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <meta charset="UTF-8">
5    <meta name="viewport" content="width=device-width, initial-scale=1.0">
6    <link rel="stylesheet" href="style.css">
7    <title>Juego Pintar Coches</title>
8  </head>
9  <body>
10   <div id="div-completo">
11     <div class="game-container">
12       <h1>Juego Pintar Coches</h1>
13       <h3>Carros en la fila</h3>
14       <div class="car-container">
15
16       </div>
17       <div class="color-palette">
18
19       </div>
20     </div>
21   </div>
22
23   <div class="boton-iniciar">
24     <button id="startButton" >Iniciar Juego</button>
25     <button id="resetButton" style="display: none;">Reiniciar juego</button>
26   </div>
27
28   <div class="carros-atendidos">
29     <h3>Carros atendidos</h3>
30   </div>
31   <div class="resultados">
32     <h3>Resultados</h3>
33     <p id="resultadosText"></p>
34   </div>
35   <script type="module" src="js/main.js"></script>
36 </body>
37 </html>
```

Después se hizo el primer archivo js, llamado carGame donde están cada uno de los métodos que tendrá el carro:

```
practica11 > js > JS carGame.js > CarGame > constructor
1  export class CarGame {
2    constructor() {
3      this.colors = ['red', 'blue', 'green', 'yellow', 'purple', 'orange'];
4      this.carQueue = [];
5      this.carCount = 0;
6      this.timerInterval = null;
7      this.carSpeed = 3000;
8      this.paintedCars = 0;
9    }
```

Comienza con la declaración de la clase CarGame con su constructor, en la captura son las primeras propiedades iniciales del juego, como los colores, la cola, la generación de colores random, contador de carros, velocidad de los carros iniciales a 3000 milisegundos y cuántos carros se han pintado.

```
10 this.carText = document.querySelector('.car-color-text');
11 this.carContainer = document.querySelector('.car-container');
12 this.colorPalette = document.querySelector('.color-palette');
13 this.colorButtons = document.querySelectorAll('.color-button');
14 this.startButton = document.getElementById('startButton');
15
16 this.carrosAtendidos = document.querySelector('.carros-atendidos');
17 this.resultados = document.querySelector('.resultados');
18 this.resultadosText = document.getElementById('resultadosText');
19
```

Después se declaran propiedades que llaman a ciertas partes del HTML y de lo que se vaya creando para el HTML.

```
19
20 this.record = {
21   paintedCars: 0,
22   startTime: null,
23   endTime: null,
24   totalTime: 0
25 };
26
27 this.juegoTerminado = false;
28 this.blankCarsContainer = 0;
29 this.startGame();
30
```

Por último, en el constructor, se encuentra lo que el registro del juego guarda, estará cantidad de carros pintados que inician en 0, la hora de inicio, así como hora de finalización, y también el tiempo total que se hizo, después están otras propiedades como juegoTerminado que indica si el juego ha terminado, como falso al inicio, blankCarsContainer que busca cuántos carros en blanco hay en el contenedor, que ayudará a determinar cuánto finalizar el

juego.

```
32 startGame() {
33   clearInterval(this.timerInterval);
34   this.record.startTime = new Date().getTime();
35   this.createColorPalette();
36   this.createCar();
37
38   this.blankCarsInQueue = 0;
39   this.createdCarCount = 0;
40
41   this.timerInterval = setInterval(() => {
42     this.increaseCarSpeed();
43     this.createCar();
44   }, this.carSpeed);
45 }
```

Proseguimos con el método StartGame el cual se encarga, como dice su nombre, de iniciar el juego, en este método se inicializan ciertas propiedades para cada vez que se inicie un juego, por ejemplo, el temporizador se limpia, y la cola de carros en blanco y el contador de carros se inicia en 0 siempre que comienza el juego. Por el otro lado del método, se registra el tiempo de inicio del juego, se usan los métodos de createColorPalette y createCar para crear la paleta de colores y los carros, respectivamente, los carros se crean a intervalos regulares utilizando un temporizador. La velocidad de creación de carros aumenta con el tiempo.

```
47     resetGame(){
48         clearInterval(this.timerInterval);
49         this.carQueue = [];
50         this.carCount = 0;
51         this.carSpeed = 3000;
52         this.paintedCars = 0;
53         this.carrosAtendidos.innerHTML = '';
54         this.carContainer.innerHTML = '';
55         this.resultados.textContent = '';
56         this.colorPalette.innerHTML = '';
57         this.record.paintedCars = 0;
58         this.record.startTime = null;
59         this.record.endTime = null;
60         this.record.totalTime = 0;
61
62         this.blankCarsContainer = 0;
63
64         this.createColorPalette();
65     }
66
```

En este método, como su nombre lo indica se 'resetea' el juego a su estado inicial, limpia todas las propiedades y elementos del juego, incluyendo la cola de carros, a velocidad, los respectivos conteos, y los registros declarados en el constructor, todo esto se hace, pues al momento de reiniciar el juego, se use este 'reseteo' para poder jugar como la primera vez.

El método siguiente lo que hace habilitar los botones de los colores, y pone el juegoTerminado en false, lo que indica que el juego no ha acabado.

```
67     resetButtons(){
68         this.juegoTerminado = false;
69
70         this.colorButtons.forEach((button) => {
71             button.classList.remove('disabled');
72         });
73     }
74
```

```

75  createColorPalette() {
76      this.colorPalette.innerHTML = '';
77
78      this.colors.forEach((color) => {
79          const colorButton = document.createElement('div');
80          colorButton.className = 'color-button';
81          colorButton.style.backgroundColor = color;
82          if(!this.juegoTerminado){
83              colorButton.addEventListener('click', () => this.paintCar(color));
84          }else{
85              colorButton.classList.add('disabled');
86          }
87          this.colorPalette.appendChild(colorButton);
88      });
89  }
90

```

Este método createColorPalette, crea la paleta de colores que se usa en la interfaz del juego., limpia cualquier paleta de colores existente, y por cada color que exista, se crea un nuevo 'div' en el que se pinta el fondo y se convierte en un botón, cada botón es un diferente color, en el que el jugador puede hacer click. Luego, existe una condición en donde si el juego es diferente de false (true), se puede hacer click a los botones para pintar los carros (de ahí llama al método paintCar, en caso de que el juego haya terminado, es donde se deshabilitan los botones. Por último, cada botón se agrega en la fila.

```

90
91  createCar() {
92      const carContainer = document.createElement('div');
93      carContainer.className = 'unitCar-container';
94
95      const car = document.createElement('img');
96      car.className = 'car';
97      car.classList.add('unpainted');
98      const carColor = this.getRandomColor();
99      car.src="carros/carro-white.png";
100     car.dataset.color = carColor;
101
102     const carColorText = document.createElement('p');
103     carColorText.className = 'car-color-text';
104     carColorText.textContent = carColor;
105
106     carContainer.appendChild(car);
107     carContainer.appendChild(carColorText);

```

Luego, está el método createCar, el cual crea un nuevo carro y se agrega a la cola de carros (carQueue). Cada carro es creado como un contenedor (con su clase para identificarlo), por cada contenedor, existe un elemento tipo imagen y un texto donde indica



el color al que se debe pintar (con su respectiva clase, también). Cada carro se inicializa como 'no pintado' (unpainted) y se le asigna un color aleatorio. Para el elemento 'car' de tipo imagen, se necesita de una imagen en específico (un carro vacío), por lo que, agrego una ruta a esa imagen del carro en blanco. Por cada contenedor de carro se le agrega tanto su imagen como su texto.

```
108
109     this.carQueue.push(carContainer);
110     this.carContainer.appendChild(carContainer);
111     this.carCount++;
112
```

También existe en este método, por cada contenedor se agrega tanto a la cola como al contenedor en sí de los carros, así como se aumenta el contador de carros.

```
112
113     if(this.carCount === 3){
114         this.increaseCarSpeed();
115     }
116
117     if(car.classList.contains('unpainted')){
118         this.blankCarsContainer++;
119         console.log(`Carros en blanco en el contenedor: ${this.blankCarsContainer}`);
120     }
121
122     if(this.blankCarsContainer === 5){
123         this.endGame();
124     }
125
```

Por último, en este método, están ciertas condiciones:

La primera es que, si el contador de carros llega ser 3, entonces se llama al método `increaseCarSpeed` para subir la velocidad del juego.

La segunda es una ayuda para poder contar cuantos carros hay con la clase 'unpainted' o blancos en el contenedor de carros en la fila, para que se agregue al contador `blankCarsContainer`, lo cual, nos ayuda para el límite de carros en la cola, es por esto, que existe la tercera condición, donde si este contador llega a ser igual a 5, se llama a el método `endGame`, que finaliza el juego.

```

126
127     paintCar(color) {
128
129         if(!this.juegoTerminado){
130             const unpaintedCar = this.carContainer.querySelector('.unpainted');
131
132             if (unpaintedCar) {
133                 const carColor = unpaintedCar.dataset.color;
134
135                 if (carColor === color) {
136                     unpaintedCar.classList.remove('unpainted');
137                     unpaintedCar.src=`carros/carro-${color}.png`;
138
139                     const carrosAtendidos = document.querySelector('.carros-atendidos');
140                     carrosAtendidos.appendChild(unpaintedCar);
141
142                     const carText = document.querySelector('.car-color-text');
143                     carText.remove();
144
145                     this.carQueue.shift();
146
147                     this.record.paintedCars++;
148
149                     this.blankCarsContainer--;
150                 }
151             }
152         }
153     }
154

```

Luego, está el método `paintCar`, que permite al jugador a pintar cada carro si hace click en el color correcto que se indica. Aquí primero se verifica que el juego siga en curso para poder pintar. Se crea una variable `unpaintedCar` que toma a cada carro con la clase 'unpainted' del contenedor de la fila de carros, lo que significa que no ha sido pintado. Si se encuentra algún carro no pintado se compara el color del carro no pintado (`carColor`) con el color seleccionado por el jugador (`color`). Si coinciden, se ejecuta el bloque de código:

Primero, se le quita la clase llamada 'unpainted' indicando que ya ha sido pintado, y se le cambia la fuente de imagen a la imagen del carro del color que el jugador seleccionó.

Se busca en el HTML el espacio para mostrar a los carros atendidos y se guarda este `unpaintedCar` (ahora ya pintado) en este espacio. También se llama al texto del color que indicaba qué color pintar y se remueve por completo de lugar. Después, se elimina este elemento (que es el primero de la cola) de esta cola de carros, ya que este carro ha sido pintado y movido con éxito. Por último, se decrementa el contador de carros en blanco del contenedor.

```

154
155     getRandomColor() {
156         return this.colors[Math.floor(Math.random() * this.colors.length)];
157     }
158
159     increaseCarSpeed() {
160         this.carSpeed -= 2000;
161     }

```

Proseguimos con el método de obtener un color aleatorio para los botones de la paleta de colores del juego. También está el método de `increaseCarSpeed`, el cual incrementa la velocidad de creación de carros, lo que hace que el juego sea más desafiante al reducir el tiempo entre la aparición de carros.

```
162
163     mostrarRecord(){
164         if (this.record.startTime && this.record.endTime) {
165             this.record.totalTime = (this.record.endTime - this.record.startTime) / 1000;
166             this.resultados.textContent = `Record: Carros pintados - ${this.record.paintedCars},
167             Tiempo total - ${this.record.totalTime.toFixed(2)} segundos`;
168         }
169     }
170
```

Después, el método `mostrarRecord` calcula y muestra el registro de carros pintados (cuándo empezó y cuánto acabado), así como el tiempo total al final del juego, esto se muestra del espacio con la clase 'resultados'.

```
171     endGame() {
172         if (!this.juegoTerminado) {
173             this.juegoTerminado = true;
174             clearInterval(this.timerInterval);
175             this.record.endTime = new Date().getTime();
176             this.mostrarRecord();
177
178             this.colorButtons.forEach((button) => {
179                 button.classList.add('disabled');
180             });
181         }
182     }

```

Por último, de este archivo, se encuentra el método `endGame` que finaliza el juego, en donde si el juego está activo, lo detiene con `juegoTerminado = true`, detiene el temporizador, registra el tiempo de finalización y muestra el registro del juego. También deshabilita los botones de colores para que el jugador no pueda interactuar más con el juego.

```

practica11 > js > JS main.js > ...
1  import { CarGame } from "../carGame.js";
2
3  let game = null;
4  const startButton = document.getElementById('startButton');
5  const resetButton = document.getElementById('resetButton');
6
7  startButton.addEventListener('click', () => {
8      startButton.style.display = 'none';
9      resetButton.style.display = 'inline';
10
11      if (game) {
12          game.startGame();
13      } else {
14          game = new CarGame();
15      }
16  });
17
18  resetButton.addEventListener('click', () => {
19      game.resetGame();
20      startButton.style.display = 'inline';
21      resetButton.style.display = 'none';
22      game.resetButtons();
23  });
24

```

Por el otro lado, para terminar la codificación del juego, está la creación de la parte que interactúa con el juego. Primero se importa la clase CarGame del otro archivo. Se declara una variable 'game' y se inicializa como null. Esta variable se utilizará para almacenar una instancia de la clase 'CarGame'. Se llaman del HTML los botones de startButton y resetButton, y para startButton se crea un evento para iniciar con el juego, en este evento se oculta el botón de Iniciar Juego, y sale el botón Reiniciar Juego. Se verifica si la variable 'game' ya tiene una instancia de la clase CarGame. Si es así, se llama al método startGame de esa instancia para reanudar el juego. Si no existe una instancia de juego, se crea una nueva instancia de la clase CarGame y se almacena en la variable game.

Para el otro botón resetButton, también se le agrega un evento a este, en donde se llama al método de resetGame para reiniciar los valores del juego, se muestra de nuevo el botón de Iniciar Juego y desaparece el otro de Reiniciar Juego.

Le agregamos su respectivo diseño con CSS:

```

1  body {
2      font-family: Arial, sans-serif;
3      background-color: #f0f0f0;
4      margin: 0;
5  }
6
7  .game-container {
8      background-color: #fff;
9      border: 2px solid #ccc;
10     border-radius: 10px;
11     padding: 20px;
12     text-align: center;
13     align-items: center;
14 }
15
16 h1 {
17     font-size: 24px;
18 }
19
20 .car-container {
21     display: flex;
22     flex-wrap: wrap;
23     justify-content: center;
24     margin-top: 20px;
25 }
26
27 .car {
28     width: 100px;
29     height: 50px;
30     border: 2px solid #000;
31     margin: 5px;
32     text-align: center;
33     line-height: 50px;
34     font-weight: bold;
35 }
36
37 .color-palette {
38     display: flex;
39     justify-content: center;
40     margin-top: 20px;
41 }
42
43 .color-button {
44     width: 50px;
45     height: 50px;
46     margin: 0 10px;
47     cursor: pointer;
48     border: 2px solid #000;
49 }
50

```

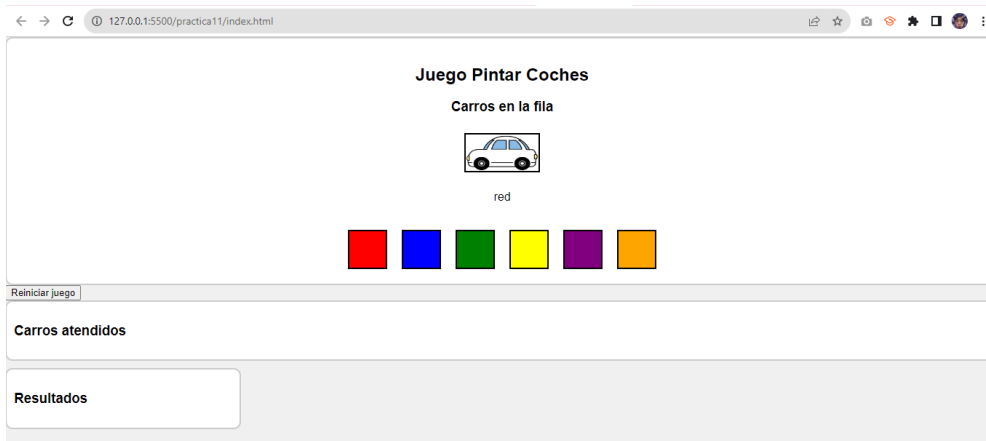
```

57
58  .resultados{
59      display: block;
60      margin-top: 10px;
61      font-size: 16px;
62      background-color: #fff;
63      border: 2px solid #ccc;
64      border-radius: 10px;
65      padding: 10px;
66      width: 300px;
67  }
68

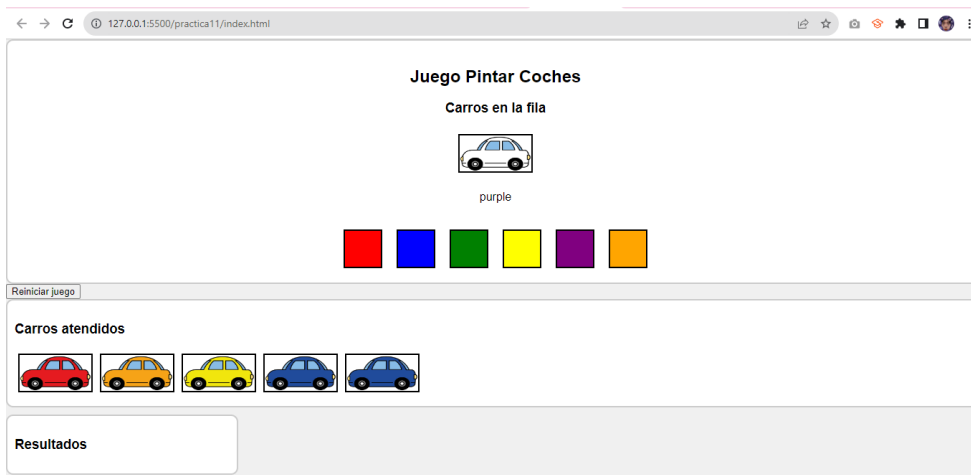
```

Con ayuda de Electron, se manda para aplicación:

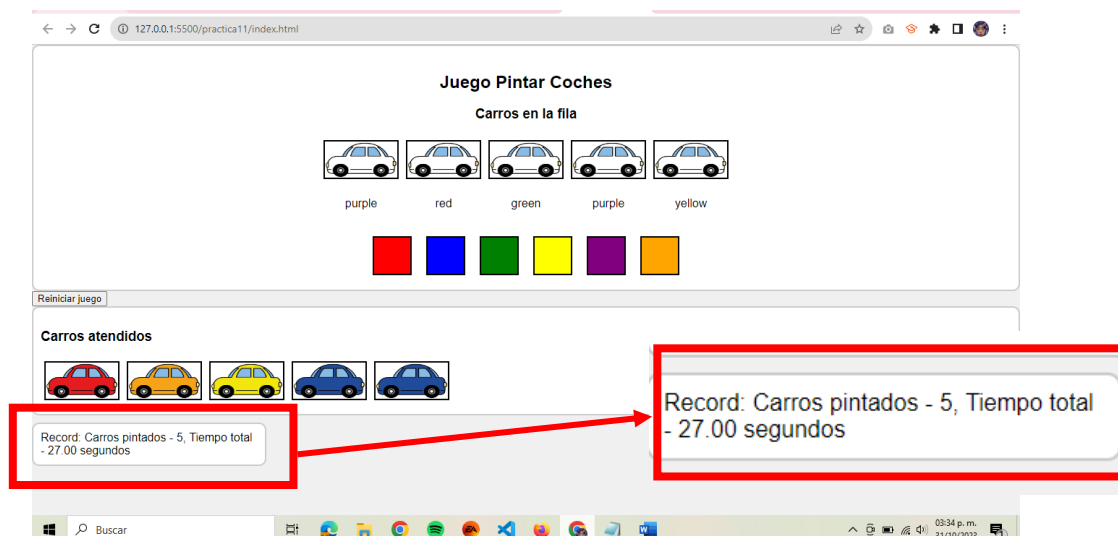
### Visualizamos la funcionalidad del programa:



El juego ya avanzado con varios carros pintados:



Cuando llega al límite de 5 carros en blanco en la fila:



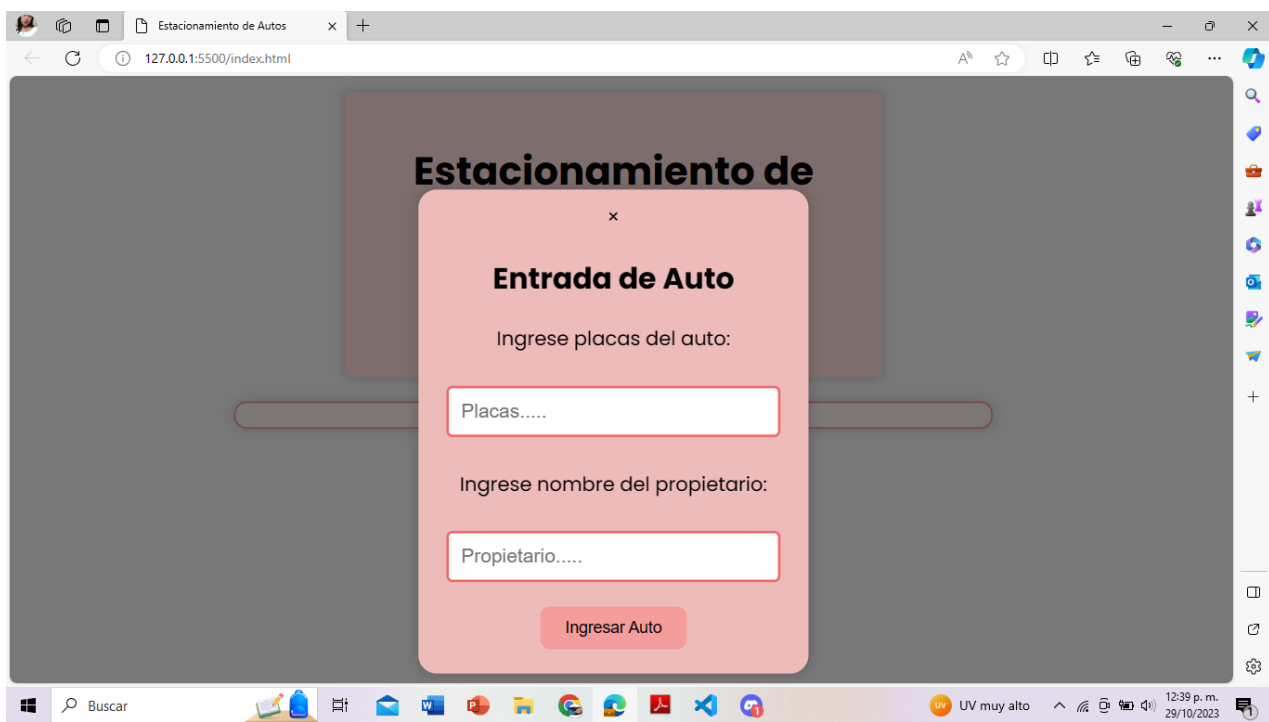
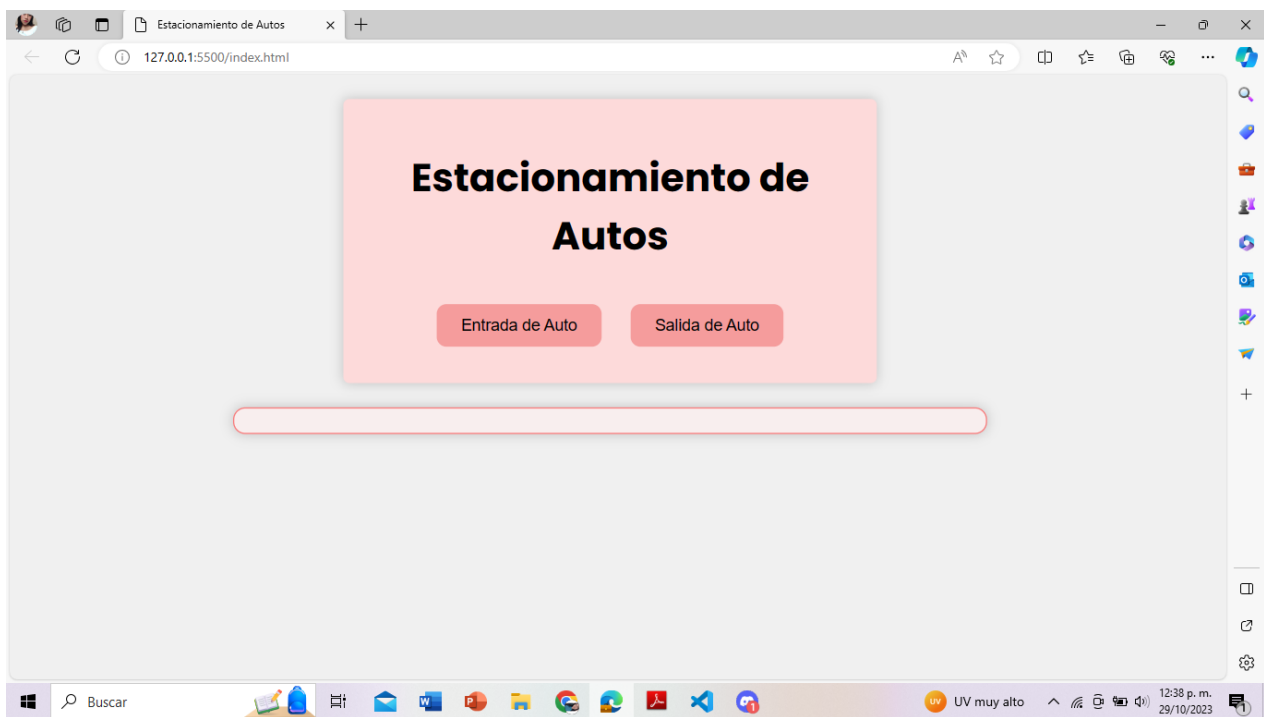
## PRACTICA 12

Creamos la estructura de la página web en **HTML**. Incluye un título, un encabezado, botones para mostrar un modal y realizar acciones relacionadas con estacionamiento de autos, y un área para mostrar información. La funcionalidad de los botones y la interacción con el usuario se controla mediante JavaScript en el archivo "script.js".

```
index.html X
index.html > html > body > div.container > div#options.options > button#mostrarModalBtn

3 <head>
4   <link rel="stylesheet" type="text/css" href="style.css">
5   <title>Estacionamiento de Autos</title>
6 </head>
7 <body>
8   <div class="container">
9     <h1>Estacionamiento de Autos</h1>
10    <div id="options" class="options">
11      <button id="mostrarModalBtn" onclick="mostrarModal()">Entrada de Auto</button>
12      <button id="salidaAutoBtn" onclick="salidaAuto()">Salida de Auto</button>
13    </div>
14  </div>
15
16  <div id="modal" class="modal">
17    <div class="modal-content">
18      <span id="close" onclick="cerrarModal()">&times;</span>
19
20      <h2>Entrada de Auto</h2>
21
22      <p>Ingrese placas del auto: </p>
23      <input id="placas" type="text" placeholder="Placas.....">
24
25      <p>Ingrese nombre del propietario: </p>
26      <input id="propietario" type="text" placeholder="Propietario.....">
27
28      <button id="entradaAutoBtn" onclick="entradaAuto()">Ingresar Auto</button>
29    </div>
30  </div>
31
32  <div id="output" class="output"></div>
33
34  <script type="module" src="script.js"></script>
```

Este es la forma en la que se ve el maquetado con estilo de la página.





Después está el primer archivo llamado **Auto.js**. En este archivo define una clase llamada **Auto** que representa un objeto de automóvil en el contexto de un sistema de estacionamiento. Esta clase tiene un constructor que toma información sobre un automóvil, como sus placas, propietario y hora de entrada al estacionamiento, y un método "**calcularCosto**" que determina el costo del estacionamiento en función del tiempo transcurrido desde su entrada. En este cálculo, se asume un costo de **\$2** por segundo.

```
JS Auto.js  X
JS Auto.js > ...
1  export default class Auto {
2      constructor(placas, propietario, horaEntrada) {
3          this.placas = placas;
4          this.propietario = propietario;
5          this.horaEntrada = horaEntrada;
6      }
7
8      calcularCosto() {
9          const horaSalida = new Date();
10         const tiempoEstacionado = (horaSalida - this.horaEntrada) / 1000; // en segundos
11         const costo = tiempoEstacionado * 2; // $2 por segundo
12         return costo;
13     }
14 }
15
```

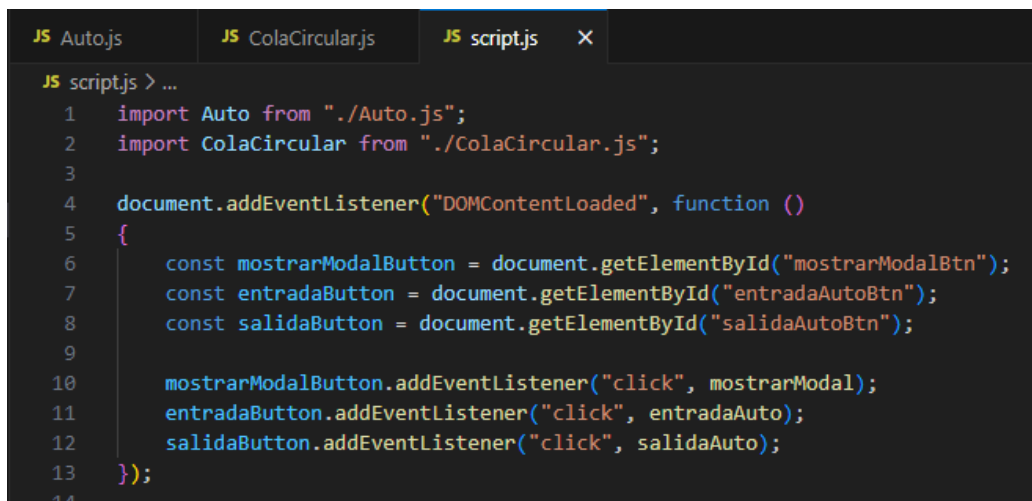
Luego cree el segundo archivo llamado **ColaCircular.js**. Este archivo tiene clase llamada **ColaCircular**, que representa una cola (o fila) de elementos con una capacidad fija. La cola circular tiene la particularidad de que, cuando se llega al final de su capacidad, los elementos se agregan nuevamente al principio, creando un ciclo. En su constructor, se establece la capacidad máxima, se crea un arreglo para almacenar elementos y se inicializan las posiciones de inicio y fin. Dos métodos, **estaVacía** y **estaLlena**, verifican el estado de la cola en función de sus propiedades.

```
JS Auto.js JS ColaCircular.js X
JS ColaCircular.js > ColaCircular > desencolar
1 export default class ColaCircular
2 {
3   constructor()
4   {
5     this.capacidad = 10; // Cambia la capacidad según tus necesidades
6     this.autos = new Array(this.capacidad);
7     this.inicio = 0;
8     this.fin = 0;
9   }
10
11   estaVacia()
12   {
13     return this.inicio === this.fin && this.autos[this.inicio] === undefined;
14   }
15
16   estaLlena()
17   {
18     return this.inicio === this.fin && this.autos[this.inicio] !== undefined;
19   }
20 }
```

La función **encolar** agrega elementos al final de la cola, teniendo en cuenta la capacidad y aplicando una lógica de cola circular, y **desencolar** elimina y devuelve el primer elemento de la cola, también utilizando una lógica de cola circular si la cola no está vacía. Esta clase es útil para gestionar elementos en una estructura de cola con una capacidad predefinida y garantiza un uso eficiente del espacio.

```
21   encolar(auto)
22   {
23     if (!this.estaLlena())
24     {
25       this.autos[this.fin] = auto;
26       this.fin = (this.fin + 1) % this.capacidad;
27     }
28   }
29
30   desencolar()
31   {
32     if (!this.estaVacia())
33     {
34       const auto = this.autos[this.inicio];
35       this.autos[this.inicio] = undefined;
36       this.inicio = (this.inicio + 1) % this.capacidad;
37       return auto;
38     }
39     return null;
40   }
41 }
```

Por último, está el archivo llamado **script.js**. Dentro de ese archivo se comienza por importar las clases **Auto** y **ColaCircular** desde los archivos correspondientes, lo que permite utilizar estas clases en la aplicación. Luego, se establece un evento que se dispara cuando la página web ha cargado completamente, garantizando que todos los elementos del documento HTML estén disponibles para interactuar con JavaScript. Dentro de este evento, se obtienen referencias a los botones de la página mediante sus **IDs**. Se añaden oyentes de eventos a estos botones de modo que cuando se hagan clic en ellos, se ejecuten funciones específicas, como "**mostrarModal**", "**entradaAuto**" y "**salidaAuto**," que probablemente gestionarán acciones relacionadas con el estacionamiento de autos en la aplicación.



```
JS Auto.js JS ColaCircular.js JS script.js X
JS script.js > ...
1 import Auto from "../Auto.js";
2 import ColaCircular from "../ColaCircular.js";
3
4 document.addEventListener("DOMContentLoaded", function ()
5 {
6     const mostrarModalButton = document.getElementById("mostrarModalBtn");
7     const entradaButton = document.getElementById("entradaAutoBtn");
8     const salidaButton = document.getElementById("salidaAutoBtn");
9
10    mostrarModalButton.addEventListener("click", mostrarModal);
11    entradaButton.addEventListener("click", entradaAuto);
12    salidaButton.addEventListener("click", salidaAuto);
13 });
14
```

Creamos una cola para administrar elementos de estacionamiento, y también define funciones para mostrar y ocultar un modal en respuesta a acciones del usuario, como hacer clic en botones. El modal se muestra al hacer clic en "Entrada de Auto" y se oculta al hacer clic en el botón de cierre. Estas funciones son parte de la interacción del usuario en la página y se relacionan con la lógica de estacionamiento de autos.

```

14
15   const estacionamiento = new ColaCircular();
16
17   function mostrarModal()
18   {
19       const modal = document.getElementById("modal");
20       modal.style.display = "block";
21   }
22
23   function cerrarModal()
24   {
25       const modal = document.getElementById("modal");
26       modal.style.display = "none";
27   }
28

```

La función **entradaAuto** registra la entrada de un automóvil en el estacionamiento. Verifica si se proporcionaron las placas y el nombre del propietario, y si es así, agrega el automóvil a la cola de estacionamiento si no está llena. Luego, muestra información del automóvil, cierra un modal y borra los campos de entrada. Si la cola está llena o si faltan datos, muestra un mensaje de alerta correspondiente.

```

29   function entradaAuto()
30   {
31       const placasInput = document.getElementById("placas");
32       const propietarioInput = document.getElementById("propietario");
33
34       const placas = placasInput.value;
35       const propietario = propietarioInput.value;
36
37       if (placas && propietario)
38       {
39           const auto = new Auto(placas, propietario, new Date());
40
41           if (!estacionamiento.estaLlena())
42           {
43               estacionamiento.encolar(auto);
44               const output = document.getElementById("output");
45               output.innerHTML = `<p>AUTO INGRESADO:<br><br> Placas: ${auto.placas}<br> Propietario: ${auto.propietario}<br>
46               cerrarModal();
47
48               // Restablecer los campos de entrada
49               placasInput.value = "";
50               propietarioInput.value = "";
51           }
52
53           else
54           {
55               alert("El estacionamiento está lleno.");
56           }
57       }
58   }

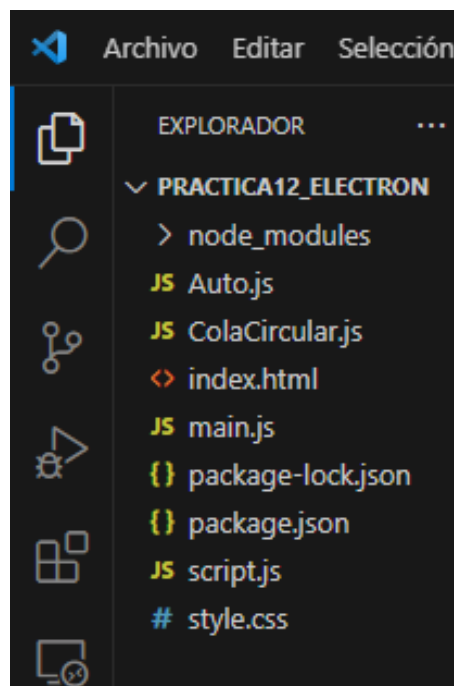
```

La función **salidaAuto** se encarga de procesar la salida de un automóvil del estacionamiento. Extrae el primer automóvil de la cola y calcula el costo del estacionamiento. Luego, muestra información sobre la salida del automóvil, incluyendo placas, propietario, horas de entrada y salida, y el costo. Si el estacionamiento está vacío, se muestra una alerta.

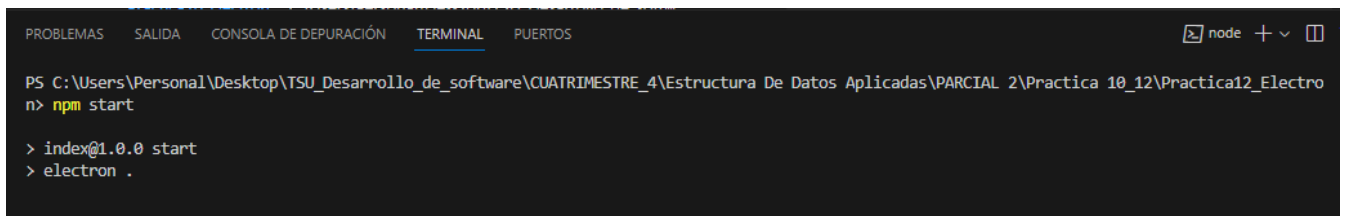
```
57
58     else
59     {
60         alert("Por favor, complete todos los campos.");
61     }
62 }
63
64 function salidaAuto()
65 {
66     const auto = estacionamiento.desencolar();
67     if (auto)
68     {
69         const costo = auto.calcularCosto();
70         const output = document.getElementById("output");
71         output.innerHTML = `<p>Auto SALIENDO:<br><br> Placas: ${auto.placas}<br> Propietario: ${auto.propietario}<br> Hora
72     }
73
74     else
75     {
76         alert("El estacionamiento está vacío.");
77     }
78 }
```

Esta página web lo convertimos a una aplicación de escritorio, con un framework llamado **electrón**.

Desde mi cd de sistemas realicé todos los pasos para poder convertirlo al igual que el visual tuvo que crear archivos.



Para poder abrir la aplicación tenemos que ejecutar el siguiente comando **npm start** desde la consola de **visual studio code**.

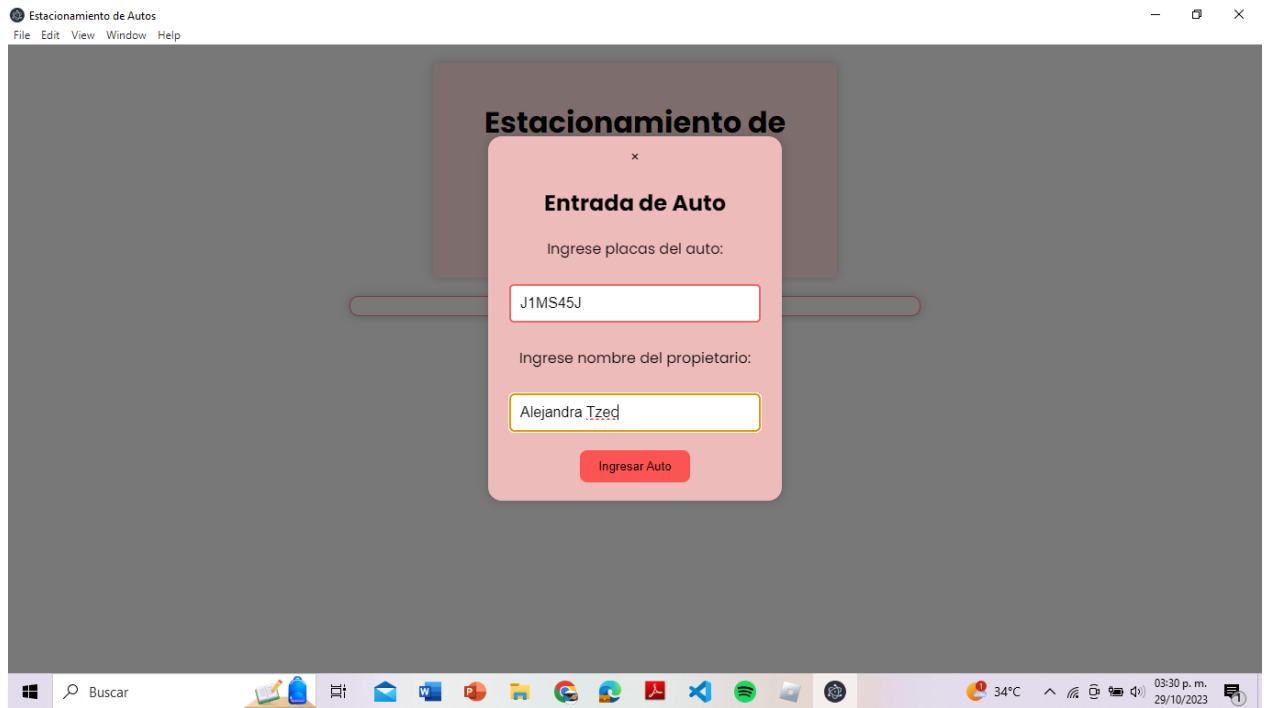
A screenshot of the Visual Studio Code terminal window. The terminal has a dark background with light-colored text. At the top, there is a header bar with tabs for 'PROBLEMAS', 'SALIDA', 'CONSOLA DE DEPURACIÓN', 'TERMINAL' (which is active and underlined), and 'PUERTOS'. To the right of the tabs, there is a search icon, the text 'node', and a plus-minus icon. The terminal content shows the following: the current directory path 'PS C:\Users\Personal\Desktop\TSU\_Desarrollo\_de\_software\CUATRIMESTRE\_4\Estructura De Datos Aplicadas\PARCIAL 2\Practica 10\_12\Practica12\_Electro', followed by the command 'n> npm start' where 'npm' is highlighted in yellow. Below this, there are two lines of output: '> index@1.0.0 start' and '> electron .' where the dot is on a new line.

```
PS C:\Users\Personal\Desktop\TSU_Desarrollo_de_software\CUATRIMESTRE_4\Estructura De Datos Aplicadas\PARCIAL 2\Practica 10_12\Practica12_Electro
n> npm start

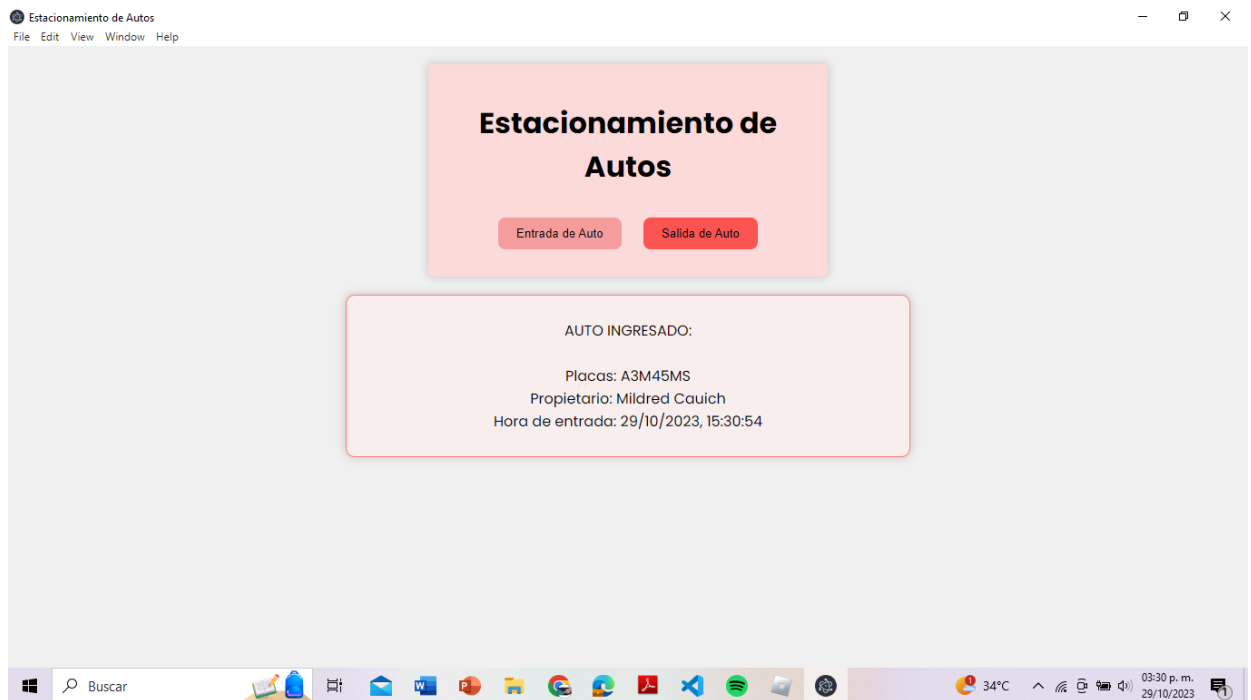
> index@1.0.0 start
> electron .
```

## Visualizamos la funcionalidad del programa

Al entrar un auto tiene que ingresar sus datos.



Al ingresar los datos del auto se agrega a la cola con numero de entrada.



Al darle clic al botón de salida de auto se sale el primer auto que ingreso con sus datos y el costo.

