

Sistema de Control de Acceso por Reconocimiento de Matrículas

Autor: Daniel Serrano Marín

I.E.S. Francisco Romero Vargas

Administración de Sistemas Informáticos en Red

Curso: 2024/2025

Índice

1. [Resumen](#)
 - 1.1 [Introducción](#)
 - 1.2 [Finalidad](#)
 - 1.3 [Objetivos](#)
 - 1.4 [Medios utilizados](#)
 - 1.5 [Estructura del repositorio](#)
2. [Arquitectura del sistema](#)
 - 2.1 [Arquitectura MVC](#)
3. [Componentes del backend](#)
 - 3.1 [Sistema de autenticación](#)
 - 3.2 [Control de acceso basado en roles](#)
 - 3.3 [Gestión de matrículas](#)
4. [Comunicación en tiempo real](#)
 - 4.1 [Visión general](#)
 - 4.2 [Arquitectura de implementación](#)
 - 4.3 [Implementación en el servidor](#)
 - 4.4 [Implementación en el cliente](#)
 - 4.5 [Flujo de datos del evento](#)
 - 4.6 [Integración con la interfaz de historial](#)
5. [Interfaz web](#)
 - 5.1 [Para usuarios normales](#)
 - 5.2 [Para administradores](#)
6. [Componente Raspberry Pi](#)
 - 6.1 [Funcionamiento paso a paso](#)

- 6.2 [Reconocimiento de matrícula](#)
 - 6.3 [Comunicación con el servidor](#)
 - 6.4 [Repetición automática](#)
 - 6.5 [Cómo se ejecuta automáticamente](#)
 - 6.6 [Ventajas de este diseño](#)
-

1. Resumen

1.1 Introducción

Este documento ofrece una visión general completa del sistema **Control Acceso Matrículas**, una solución de control de acceso de vehículos basada en el reconocimiento automático de matrículas. El sistema permite una gestión segura del acceso a instalaciones mediante la captura de imágenes de matrículas con una Raspberry Pi, su procesamiento usando OpenALPR y la verificación de autorización en una base de datos centralizada. Esta página cubre la arquitectura general, los componentes clave, los flujos de trabajo y cómo interactúan dichos componentes.

1.2 Finalidad

El objetivo principal es mejorar la **automatización del acceso** mediante el reconocimiento de matrículas.

Beneficios del sistema:

- **Acceso automatizado**, eliminando la necesidad de tarjetas o mandos.
- **Mayor seguridad**, permitiendo solo la entrada de vehículos autorizados.
- **Gestión eficiente**, con un sistema centralizado para administrar accesos.
- **Registro detallado** de todos los accesos.

1.3 Objetivos

Desde un punto de vista técnico, el proyecto se centra en:

- **Capturar imágenes de matrículas** con una **cámara en Raspberry Pi 3B**.
- **Detectar matrículas automáticamente** con **OpenALPR**.
- **Almacenar y gestionar matrículas** en una **base de datos MySQL**.
- **Desarrollo un script en Python** que compare matrículas con la base de datos.
- Una **API** con Flask para la gestión de matrículas.

- **Una interfaz web** para que los usuarios puedan solicitar el registro de su matrícula y otra **interfaz web** para administradores.
- **Implementación un panel de administración** donde se aprueben o rechacen matrículas.

1.4 Medios Utilizados

Para llevar a cabo este proyecto, se necesitará:

Hardware:

- Raspberry Pi 3B con Ubuntu Server.
- Cámara Raspberry Pi HQ.
- MicroSD de al menos 16GB con sistema operativo instalado.
- VPS en DigitalOcean para alojar la base de datos y la API.

Software:

- Python, Flask (API), MySQL (base de datos).
- OpenALPR para reconocimiento de matrículas.
- HTML + Bootstrap + Flask para la interfaz web.
- Servidor web.

1.5 Estructura del repositorio

```

/control-acceso-matriculas
├── README.md                # Documentación del proyecto
├── .gitignore               # Archivos ignorados por Git
├── ── canvas/                # Diagrama de rutas en Obsidian Canvas
│   └── ── rutasaplicacion.canvas
├── ── docs/                  # Documentación técnica
│   ├── ── ── capturas_documentacion/
│   ├── ── ── Raspberry/
│   │   ├── ── EjecucionAutomaticaScript.md
│   │   ├── ── ExplicacionProcesarMatricula.md
│   │   └── ── ── OpenALPR/
│   │       └── ── InstalacionOpenALPR.md
│   └── ── ── VPS/
│       └── ── DespliegueAplicacionFlaskconGunicorn.md
├── ── ── backend/            # Aplicación Flask (API y frontend integrado)
│   ├── ── app.py
│   └── ── wsgi.py

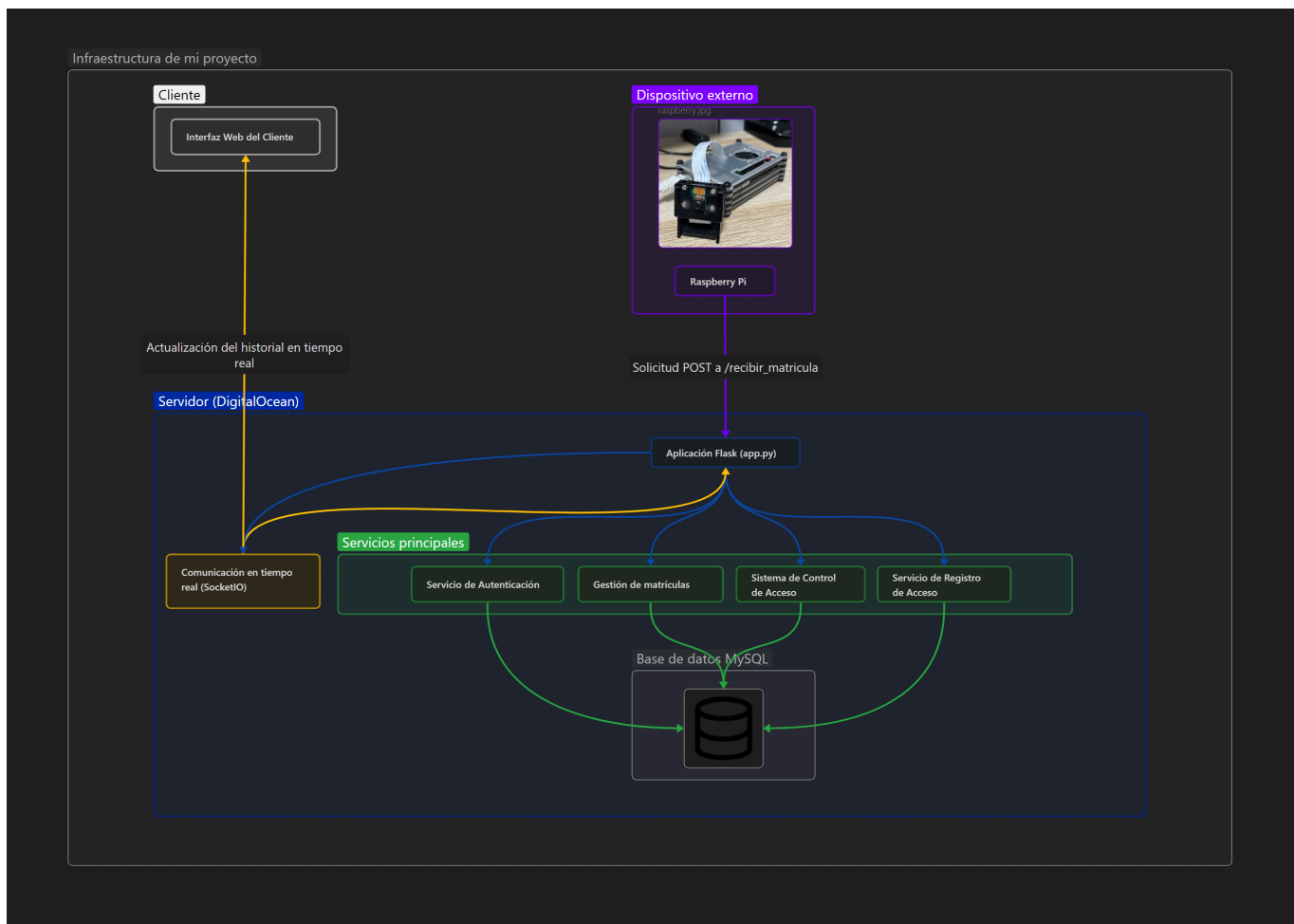
```

```
|   ├── requirements.txt
|   ├── 📁 routes/           # Blueprints Flask: auth, api, admin, main,
matriculas
|   ├── 📁 templates/       # Plantillas HTML
|   ├── 📁 static/          # CSS, iconos, imágenes
|   └── 📁 utils/           # db_utils.py y funciones auxiliares
└── 📁 raspberry-pi/        # Script de captura en Raspberry Pi
    └── procesar_matricula.py
```

2. Arquitectura del Sistema

El sistema *Control Acceso Matrículas* consta de tres componentes principales:

- **Aplicación Web:** Un servidor basado en Flask que gestiona la autenticación de usuarios, la gestión de matrículas y la lógica de control de acceso.
- **Componente Raspberry Pi:** Captura imágenes, procesa las matrículas y se comunica con el servidor.
- **Interfaz de Usuario:** Interfaces web tanto para usuarios normales como para administradores.



2.1 Arquitectura MVC

► Archivos fuente de esta parte

El patrón de diseño utilizado en este proyecto es la arquitectura **MVC** (Modelo-Vista-Controlador). La arquitectura **MVC** es un patrón de diseño muy común en el desarrollo de aplicaciones web, incluido. Divide la lógica de una aplicación en tres componentes separados:

- **Modelos**

- ¿Qué es?

Representa **los datos** y la lógica de la base de datos de la aplicación.

- En mi proyecto:
 - Se gestiona con funciones de acceso a la [base de datos](#) en [db_utils.py](#).
 - Se encarga de:
 - Conectarse a MySQL.
 - Recuperar y guardar información sobre usuarios, matrículas, accesos.

Ejemplo:

```
def conectar_db():

    return mysql.connector.connect(
        host="localhost",
        user="flask_user",
        password="flask_user",
        database="control_acceso"
    )
```

Código extraído del archivo: [db_utils.py](#).

• Vistas

- ¿Qué es?

Es la **interfaz visual** con la que interactúa el usuario: HTML, CSS y Flask (Python).

- En mi proyecto:
 - Están en la carpeta [templates/](#).
 - Se usan con **Jinja2** para insertar dinámicamente datos en las páginas.
 - Muestran matrículas, formularios de login, tablas de usuarios, etc. **Ejemplo:**

```
<h4 class="text-white mb-0">Mis Matrículas Registradas</h4>
{% for matricula, estado in todas %}
    <tr>
        <td>{{ matricula }}</td>
        <td>{{ estado }}</td>
    </tr>
{% endfor %}
```

Código extraído del archivo: [index.html](#).

• Controladores

- ¿Qué es?

Es el **punto entre el Modelo y la Vista**. Gestiona la lógica de la aplicación: recibe peticiones del usuario, actualiza modelos y decide qué vista mostrar.

- En mi proyecto:
 - Están en [routes/](#) : [auth.py](#), [main.py](#), [admin.py](#), etc.
 - Cada archivo define rutas (`@app.route`) y qué hacer cuando se accede a ellas.

Ejemplo:

```
@main.route("/")
@login_required
def index():
    conexion = conectar_db()
    cursor = conexion.cursor()
    ...
```

Código extraído del archivo: [main.py](#).

3. Componentes del Backend

3.1 Sistema de Autenticación

► Archivos fuente de esta parte

Resumen del sistema

El sistema de autenticación gestiona la verificación de identidad de usuarios, mantiene sus sesiones y controla el acceso a las distintas secciones de la aplicación según el rol del usuario (usuario o administrador).

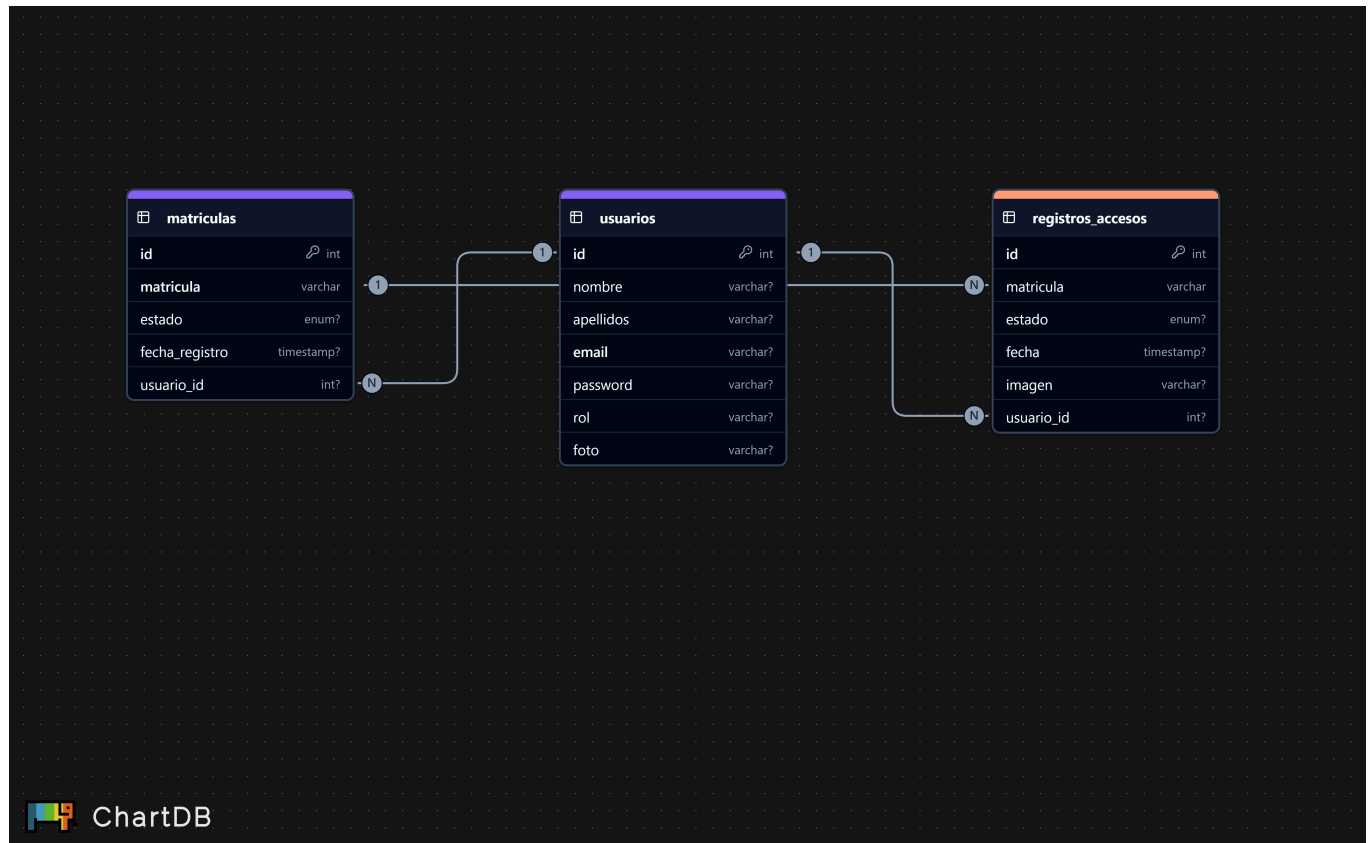
Modelo de usuario y almacenamiento de datos

Se utiliza una clase personalizada [User](#) que implementa UserMixin de Flask-Login para representar a los usuarios autenticados. Los datos se almacenan en la tabla [usuarios](#) de la base de datos MySQL.

Atributos del modelo [User](#) :

- [id](#) : identificador único
- [nombre](#) : nombre del usuario
- [email](#) : dirección de correo (para login)
- [password](#) : contraseña (hash)
- [matrícula](#) : matrícula asociada
- [rol](#) : admin o usuario
- [foto](#) : foto del usuario

Las contraseñas se almacenan con hash seguro usando `generate_password_hash` , y se verifican con `check_password_hash` .



Flujo de autenticación

Inicio de sesión:

1. El usuario envía email y contraseña al endpoint `/login` .
2. El sistema consulta el usuario por email.
3. Se compara el hash de la contraseña.
4. Si coincide:
 - Se inicia sesión con `login_user()` .
 - Se guarda el rol en la sesión.
 - Se redirige según el rol: dashboard o panel admin.

Registro:

1. El usuario completa el formulario.
2. Se valida:
 - Coincidencia de contraseñas.

- Unicidad del email.

3. Se guarda el usuario con rol `usuario` y se redirige al login.

Gestión de sesiones

Usa Flask-Login para:

- Verificar si el usuario está autenticado.
 - Proteger rutas con `@login_required`.
 - Cerrar sesión correctamente (`logout_user()`).
 - Guardar el rol en la sesión para controlar el acceso.
-

3.2 Control de acceso basado en roles

Se definen dos roles:

- `usuario` : permisos limitados.
- `admin` : acceso completo.

Se usa un decorador `@solo_admin` para:

1. Verificar si el rol en sesión es `admin`.
2. Redirigir con error si no lo es.
3. Permitir acceso si lo es.

Rutas protegidas para admin:

- `/matriculas_admin`
 - `/admin/editar_matricula`
 - `/admin/eliminar_matricula/<int:matricula_id>`
 - `/admin/eliminar_matricula/<id>`
-

Integración en la interfaz

Navegación condicional

La barra de navegación muestra enlaces distintos según el rol y estado de autenticación.

Formularios

- **Login:** solicita email y contraseña.
 - **Registro:** incluye nombre, email, contraseña y confirmación.
-

Funciones de administración

Crear usuarios

El administrador puede crear nuevos usuarios desde el panel.

Editar usuarios

Puede cambiar nombre, apellidos y email, verificando que no esté duplicado.

Seguridad

1. Contraseñas:

- Hash seguro (Werkzeug).
- Validación en login y registro.

2. Validaciones:

- Emails únicos.
- Confirmación de contraseña.

3. Sesiones:

- Se borra todo en logout.
 - Decoradores protegen rutas sensibles.
-

3.3. Gestión de Matrículas

► Archivos fuente de esta parte

El sistema de gestión de matrículas se encarga del ciclo de vida completo de las matrículas de vehículos dentro de la solución de control de accesos. Este módulo administra cómo se

solicitan, registran, modifican y autorizan las matrículas, siendo una parte crítica de la infraestructura de acceso de vehículos.

Las matrículas siguen un formato estándar español (4 números y 3 letras, por ejemplo, 1234ABC) y pueden estar en tres estados:

- **pendiente**
- **autorizada**
- **denegada**

Este documento describe el modelo de datos, el flujo de estados, las operaciones disponibles para el usuario, las funciones administrativas y cómo este subsistema se integra con el resto del sistema.

Operaciones de usuario

Los usuarios normales pueden:

1. **Solicitar una nueva matrícula:**
 - Se valida el formato (regex y validación en HTML).
 - Se comprueba si ya está registrada.
 - Se guarda con estado `pendiente` .
2. **Visualizar sus matrículas:**
 - Número de matrícula
 - Estado actual
 - Acciones disponibles (según estado)
3. **Cancelar solicitudes pendientes**
4. **Eliminar matrículas denegadas o pendientes**

Regex del formato aceptado: `\d{4}[A-Z]{3}` (ejemplo: 1234ABC)

Operaciones administrativas

Los administradores tienen funciones ampliadas:

- Ver y gestionar **todas** las matrículas del sistema
- Filtrar por estado o usuario
- Aprobar o denegar solicitudes

- Editar matrículas autorizadas
- Eliminar matrículas obsoletas
- Añadir matrículas directamente (ya autorizadas)

Integración con el sistema de control de accesos

Cuando una matrícula es detectada:

1. La Raspberry Pi la envía al endpoint `/recibir_matricula`
2. El sistema consulta su estado
3. Solo si es `autorizada`, se concede el acceso
4. Se registra el intento
5. Se emite un evento WebSocket en tiempo real

Seguridad y validaciones

- Todas las operaciones requieren usuario autenticado
 - Las validaciones se aplican en:
 - Cliente
 - Servidor
 - Detección de duplicados
 - Comprobación de roles
-

4. Comunicación en Tiempo Real

► Archivos fuente de esta parte

Este documento describe el sistema de comunicación en tiempo real usado en la aplicación Control Acceso Matrículas. Detalla cómo se implementa la tecnología WebSocket para proporcionar actualizaciones instantáneas sobre eventos de acceso de matrículas a los clientes conectados, sin necesidad de refrescar la página.

4.1 Visión General

El sistema usa WebSockets (mediante Socket.IO) para enviar en tiempo real los eventos de acceso por matrícula a los clientes web. Esto permite a los administradores y usuarios

monitorizar intentos de acceso en el momento en que ocurren, sin tener que recargar la página de historial.

4.2 Arquitectura de Implementación

El sistema de comunicación en tiempo real se compone de:

1. **Servidor:** Aplicación Flask con integración de Socket.IO para emitir eventos.
 2. **Cliente:** Cliente JavaScript de Socket.IO que se conecta al servidor y actualiza la interfaz según los eventos.
-

4.3 Implementación en el Servidor

Inicialización de Socket.IO

```
app = Flask(__name__)
app.secret_key = "clave_segura"
socketio = SocketIO(app)
```

Código extraído del archivo: [app.py](#).

Se utiliza el objeto `socketio` para emitir eventos y correr la aplicación Flask con soporte WebSocket.

Cuando se recibe una matrícula desde la Raspberry Pi, se emite un evento `nuevo_acceso` con los datos:

```
socketio.emit(f"nuevo_acceso_{usuario_id}", {
    "matricula": matricula,
    "estado": estado,
    "fecha": fecha_actual,
    "imagen": nombre_imagen,
    "usuario_id": usuario_id
})
```

Código extraído del archivo: [api.py](#).

4.4 Implementación en el Cliente

Conexión WebSocket (cliente JS)

```
const socket = io({
  path: "/socket.io",
  transports: ["websocket"]
});
```

Código extraído del archivo: [historial.html](#).

Se configura para usar solo WebSocket.

Indicador de Conexión

```
<h2 class="mb-4 text-white">
  Historial de Accesos
  <span id="estado-ws" class="badge connection-badge bg-secondary">
    Conectando...
  </span>
</h2>
```

Código extraído del archivo: [historial.html](#).

Actualizado por JavaScript:

```
socket.on("connect", () => {
  estadoWS.textContent = "● Conectado";
  estadoWS.className = "badge connection-badge bg-success";
});

socket.on("disconnect", () => {
  estadoWS.textContent = "● Desconectado";
  estadoWS.className = "badge connection-badge bg-danger";
});
```

Código extraído del archivo: [historial.html](#).

Escucha de eventos:

```
socket.on(canal, (acceso) => {
```

```
const fila = document.createElement("tr");  
...
```

Código extraído del archivo: [historial.html](#).

4.5 Flujo de Datos del Evento

1. La Raspberry Pi detecta una matrícula y envía un POST a `/recibir_matricula`
 2. El servidor valida la matrícula y registra el acceso
 3. Se emite el evento `nuevo_acceso` con los datos del intento
 4. Los clientes conectados reciben el evento y actualizan la interfaz
-

4.6 Integración con la Interfaz de Historial

- **Indicador de conexión WebSocket:** Muestra si está conectado
 - **Actualización dinámica de la tabla:** Nuevos accesos se agregan al principio sin recargar
 - **Estilos visuales según estado:** Se colorea y etiqueta según esté autorizado, pendiente o denegado
-

5. Interfaz web

- Archivos fuente de esta parte

La interfaz de usuario está desarrollada con HTML, CSS (combinándolo con Bootstrap también), y el motor de plantillas Jinja2 integrado en Flask. Su diseño adapta dinámicamente los elementos mostrados según el rol del usuario: `admin` o `usuario`.

5.1 Para usuarios normales

- Página principal (`/`) que muestra un resumen de sus matrículas registradas, divididas por estado (`autorizadas` , `pendientes` , `denegadas`).

MATRÍCULA	ESTADO
3456CXT	Autorizada
1234ABC	Autorizada
2341SAZ	Autorizada
3435PSD	Denegada
4356MSN	Autorizada
5464DXA	Autorizada
2344ASX	Denegada
5352DSS	Pendiente

- Un gráfico con sus accesos diarios, generado con **Chart.js**.



- Código del gráfico:

```
<script>
const ctxEntradas =
document.getElementById('graficoEntradas').getContext('2d');

new Chart(ctxEntradas, {

  type: 'line',
  data: {
    labels: {{ fechas|tojson }},

    datasets: [{
      label: 'Entradas por Día',
      data: {{ cantidades|tojson }},
      fill: true,
    ]
  }
});
```



```

        backgroundColor: 'rgba(20, 179, 242, 0.15)',
        borderColor: '#14b3f2',
        tension: 0.4
    }
},

options: {
    responsive: true,

    plugins: {
        legend: { labels: { color: 'white' } },
        title: { display: false }
    },

    scales: {
        x: { ticks: { color: '#fff' } },

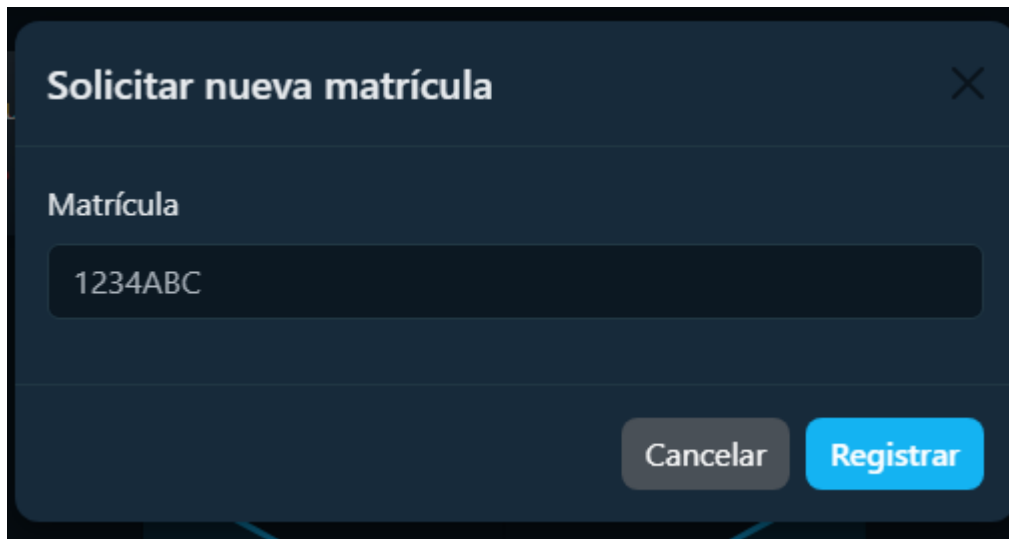
        y: {
            beginAtZero: true,

            ticks: {
                color: '#fff',
                stepSize: 1,
                callback: function(value) {
                    return Number.isInteger(value) ? value : null;
                }
            }
        }
    }
});
</script>

```

Código extraído del archivo: [index.html](#).

- Formulario para solicitar nuevas matrículas.



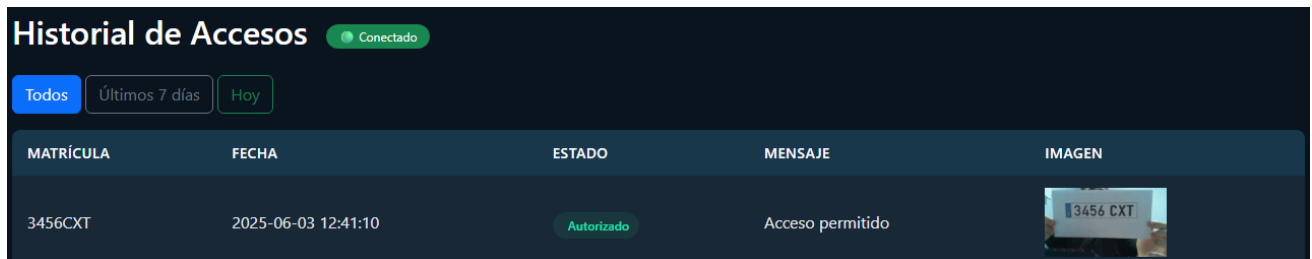
A dark-themed modal window titled "Solicitar nueva matrícula" with a close button (X) in the top right corner. Below the title is a label "Matrícula" followed by a text input field containing the value "1234ABC". At the bottom right of the modal are two buttons: "Cancelar" (grey) and "Registrar" (blue).

- Código del backend para solicitar matricula:


```
@matriculas.route('/solicitar_matricula', methods=['GET', 'POST'])
@login_required
def solicitar_matricula():
    ...
```

Código extraído del archivo: [matriculas.py](#).

- [Página de historial](#) con filtros de fechas y visualización de imágenes asociadas a cada acceso.



The "Historial de Accesos" interface features a header with the title and a "Conectado" status indicator. Below the header are three filter buttons: "Todos" (active), "Últimos 7 días", and "Hoy". The main content is a table with five columns: "MATRÍCULA", "FECHA", "ESTADO", "MENSAJE", and "IMAGEN".

MATRÍCULA	FECHA	ESTADO	MENSAJE	IMAGEN
3456CXT	2025-06-03 12:41:10	Autorizado	Acceso permitido	

5.2 Para administradores

- Acceso a `/admin` con un panel que muestra todos los usuarios registrados y todas las matrículas del sistema.

Panel de Administración

Usuarios registrados

ID	NOMBRE COMPLETO	EMAIL	ROL	ACCIONES
7	prueba prueba	prueba@prueba.com	Usuario	
8	Admin Principal	admin@admin.com	Admin	
17	Gigante Mental	gigante@gmail.com	Usuario	

Matrículas pendientes

MATRÍCULA	NOMBRE USUARIO	EMAIL	ACEPTAR RECHAZAR
-----------	----------------	-------	--------------------

Matrículas

Filtrar por estado

Todos

Buscar por usuario

Nombre o correo

ID	MATRÍCULA	NOMBRE USUARIO	EMAIL	ESTADO	ACCIONES
51	2344ASX	prueba prueba	prueba@prueba.com	Denegada	
50	5464DXA	prueba prueba	prueba@prueba.com	Autorizada	
44	4356MSN	prueba prueba	prueba@prueba.com	Autorizada	
43	3435PSD	prueba prueba	prueba@prueba.com	Denegada	
38	2341SAZ	prueba prueba	prueba@prueba.com	Autorizada	
30	3547NXB	Admin Principal	admin@admin.com	Autorizada	
5	1234ABC	prueba prueba	prueba@prueba.com	Autorizada	
1	3456CXT	prueba prueba	prueba@prueba.com	Autorizada	

- Tabla de matrículas pendientes con botones para aprobar o rechazar solicitudes.

Matrículas pendientes

MATRÍCULA	NOMBRE USUARIO	EMAIL	ACEPTAR RECHAZAR
4536UES	prueba prueba	prueba@prueba.com	

- Vistas filtradas y editables de matrículas existentes.
- Modales para crear [nuevos usuarios](#) y [editar usuarios existentes](#).
- Botones de acción rápida para [gestionar roles](#), [limpiar historial](#), o [eliminar registros](#).

6. Componente Raspberry Pi

► Archivos fuente de esta parte

Es el **sensor inteligente del sistema**. Se encarga de capturar la matrícula de un vehículo en tiempo real y comunicarse con el servidor para validar el acceso.

6.1 Funcionamiento paso a paso

- La Raspberry Pi utiliza una [cámara](#) conectada físicamente.
- El script [procesar_matricula.py](#) ejecuta continuamente este comando:

```
fswebcam -r 1280x720 --no-banner {CAPTURA}
```

Código extraído del archivo: [procesar_matricula.py](#).

Y con ese comando, se guarda una imagen de la matrícula que está frente a la cámara.

6.2 Reconocimiento de matrícula

- Se analiza la imagen usando **OpenALPR**, un sistema de reconocimiento automático de matrículas.

```
resultado = subprocess.run(["alpr", "-c", "eu", imagen], capture_output=True, text=True)
```

Código extraído del archivo: [procesar_matricula.py](#).

OpenALPR detecta si hay una matrícula en la imagen y extrae el texto, por ejemplo 1234ABC .

6.3 Comunicación con el servidor

- Si se detecta una matrícula válida, la Raspberry Pi **envía la matrícula y la imagen** al servidor web (Flask) mediante una petición **HTTP POST**:

```
SERVIDOR="https://matriculas.dsermar0808.tech/recibir_matricula"
...
respuesta = requests.post(SERVIDOR, files=archivos, data=datos, timeout=5)
```

Código extraído del archivo: [procesar_matricula.py](#).

El servidor se encarga de comprobar si esa matrícula está autorizada o no.

6.4 Repetición automática

- Este proceso se ejecuta [cada segundo](#) en un bucle infinito.
- También se evita repetir matrículas si son consecutivas.

```
if matricula_detectada:

    print(f"🚗 Matrícula detectada: {matricula_detectada}")

    if matricula_detectada != ultima_matricula:
        enviar_matricula(matricula_detectada, imagen)
        ultima_matricula = matricula_detectada
    else:

        print("▶ Matrícula repetida, no se envía de nuevo.")

else:
    print("⚠ No se detectó ninguna matrícula.")
    ultima_matricula = None
```

Código extraído del archivo: [procesar_matricula.py](#).

6.5 ¿Cómo se ejecuta automáticamente?

Se configura como **servicio** `systemd`, es decir, se inicia solo cuando se enciende la Raspberry.

Este es el archivo de configuración [matricula.service](#).

```
[Unit]
Description=Script de detección de matrículas
After=network.target

[Service]
ExecStart=/usr/bin/python3 /home/dsermar/control-acceso-matriculas/raspberry-pi/procesar_matricula.py
WorkingDirectory=/home/dsermar/control-acceso-matriculas/raspberry-pi
StandardOutput=append:/var/log/matricula.log
StandardError=append:/var/log/matricula.log
Restart=always
User=dsermar
```

```
[Install]
```

```
WantedBy=multi-user.target
```

6.6 Ventajas de este diseño

- **Descentralizado:** la Raspberry Pi toma decisiones rápidamente sin depender de cámaras IP complejas.
- **Flexible:** puedes cambiar la lógica del servidor sin tocar el script.
- **Escalable:** puedes añadir más Raspberrys en otras entradas fácilmente.