Digitalització d’un Taller Mecànic Connectat

Mòdul: 1665 Digitalització

**Nom de l’equip**: Jordi Arabia, Cristian Moyano y Denís Ramos

**Data de lliurament**: 26/05/2025

# Índex

[Índex 2](#_Toc197945044)

[1 Introducció 3](#_Toc197945045)

[1.1 Objetivos del proyecto 3](#_Toc197945046)

[1.2 Componentes del Sistema 3](#_Toc197945047)

[2 Anàlisi Inicial 4](#_Toc197945048)

[2.1 Necesidades del cliente 4](#_Toc197945049)

[3 Model de Dades 5](#_Toc197945050)

[3.1 Diagrama ER 5](#_Toc197945051)

[3.2 Esquema relacional 6](#_Toc197945052)

[3.3 Script SQL 6](#_Toc197945053)

[4 Desenvolupament de la Solució 9](#_Toc197945054)

[4.1 API 9](#_Toc197945055)

[4.2 Web 9](#_Toc197945056)

[4.3 Scripts 9](#_Toc197945057)

[5 Infraestructura de Sistemes 10](#_Toc197945058)

[6 Problemes i Solucions 11](#_Toc197945059)

[7 Conclusions 12](#_Toc197945060)

[8 Annexos 13](#_Toc197945061)

# Introducció

El objetivo de nuestro proyecto es desarrollar un sistema digital todo en uno para la gestión de clientes y citas mecánicas en un taller. Se basará en un sistema denominado Taller Mecánico MWP, desarrollado mediante una arquitectura de microservicios que incluirá una interfaz web orientada al usuario, una API como backend, y una base de datos relacional para la persistencia de la información. La finalidad esencial es simplificar las tareas mate cuándo se trata de brindar soluciones en el nivel más básico, la mejora del proceso de citas y clientes y, finalmente, aumentar la satisfacción del cliente por medio de la facilidad en la que póngase a disposición una solicitud de servicios. La infraestructura está compuesta por contenedores Docker para garantizar un despliegue eficiente y escalado de la solución.

## Objetivos del proyecto

El proyecto tiene como objetivo principal mejorar la eficiencia y la experiencia del cliente en los talleres mecánicos. Para lograr esto, se han definido los siguientes objetivos:

* **Crear un sistema digital para la gestión de clientes y citas en un taller mecánico:** El objetivo principal de este proyecto es desarrollar un sistema que permita a los clientes gestionar sus citas y servicios de manera fácil y accesible en línea.
* **Automatizar tareas básicas relacionadas con la gestión de clientes y citas:** El sistema debe ser capaz de automatizar tareas como la gestión de citas, la notificación de los clientes y la gestión de la información de los clientes.
* **Mejorar la experiencia del cliente a través de una interfaz de usuario amigable y accesible en línea:** El sistema debe ser fácil de usar y permitir a los clientes acceder a la información de sus citas y servicios en cualquier momento y desde cualquier lugar.

## Componentes del Sistema

* Una web de gestión de clientes y citas para que los clientes puedan acceder a la información de sus citas y servicios en línea.
* Una API (Application Programming Interface) para permitir la comunicación entre la web y la base de datos.
* Una base de datos segura y organizada para registrar y gestionar la información de los clientes y sus citas.

# Anàlisi Inicial

Antes de empezar con el desarrollo del proyecto y explicar cada apartado con las cosas que hemos hecho primero vamos a explicar nuestro primer análisis, más enfocado a que cosas se podrían mejorar o hacerlas más llevaderas para los clientes y para nosotros mismos que somos los que gran parte estamos implicados en el proyecto.

## Necesidades del cliente

Parándonos a pensar que es lo que se podria mejorar lo primero que se nos ocurrió fué como se gestiona el pedir una cita en un taller y generalmente suele ser por llamada telefónica o en algunos casos ir en persona, otro punto a tener en cuenta y relacionado con el anterior era que el hecho de querer pedir una cita fuera de horario como bien se ha dicho antes en algunos casos se limitaba al punto de no poder pedir cita ya que el taller estaba cerrado por el horario.

Entonces pensando nosotros como si fueramos un cliente queriedo pedir una cita fuera de horario o dentro de la franja horaria se nos ocurrió hacer lo que se lleva haciendo toda la vida pero haciendo que sea más cómodo y eficiente para el usuario/cliente.

Esto nos llevó a entender la necesidad de un sistema que no solo digitalizara el proceso de solicitud de citas, sino que también ofreciera flexibilidad horaria y una interfaz intuitiva. Así, los clientes podrían gestionar sus citas de servicio en cualquier momento y desde cualquier lugar, mejorando su experiencia.

* **Gestión eficiente de clientes y vehículos:** El cliente necesita un sistema que le permita mantener una base de datos actualizada de sus clientes y vehículos, incluyendo información como contactos, historiales de mantenimiento y reparaciones.
* **Solicitud de cita en línea:** El cliente necesita la capacidad de permitir a sus clientes solicitar cita en línea, lo que les permitirá planificar sus visitas de manera más eficiente.
* **Copias de seguridad de datos:** El cliente necesita un sistema que le permita tomar copias de seguridad de sus datos, lo que les permitirá proteger sus datos en caso de una falla o pérdida de información.

# Model de Dades

En la base de datos nos hemos centrado en que principalmente haga su función junto con las demás partes del proyecto. Por ejemplo, es fundamental que desde la web el usuario pueda pedir su cita de una manera sencilla, introduzca sus datos y nosotros podamos almacenar esa información también de una manera sencilla, teniendo un registro de nuestros clientes y sus vehículos con sus fechas seleccionades y problemas que tienen con sus vehículos.

El diseño de nuestra base de datos està hecho en MySQL, se nos ha hecho más cómodo hacerlo con este gestor de base de datos relacional porque ya hemos adquirido experiencia previo y no es muy difícil de usar, antes de tener la base final o versión final hemos realizado antes un diagrama, después un esquema y finalmente el script siendo la versión final de la base de datos.

## Diagrama ER

Diagrama

El contenido generado por IA puede ser incorrecto.

## Esquema relacional

Diagrama

El contenido generado por IA puede ser incorrecto.

## Script SQL

Para que todo este diseño se convierta en una base de datos real y funcional, utilizamos el script SQL que en nuestro proyecto el dichero se llama: **db\_schema.sql**. Lo bueno es que este script se ejecuta automáticamente cuando el contenedor de la base de datos se pone en marcha, gracias a cómo lo hemos configurado en el **docker-compose.yml**.

Este script se encarga de todo: crea nuestra base de datos dbmwp, luego las tablas Clients, Cita y Vehicles, define sus campos, establece las claves primarias y, lo más importante, configura las relaciones entre las tablas usando las claves foráneas. Además, nos aseguramos de que no haya duplicados en campos como el DNI o el email de los clientes, y en la matrícula de los vehículos.

Los comentarios que hay entre medias del código eran para saber en su momento que campos tenían clave primaria, clave foranea con el tema de las relaciones entre las tablas, sobretodo está así para a la hora de hacerlo hacerlo bien o no dejarse nada

DROP DATABASE IF EXISTS `dbmwp`;

CREATE DATABASE `dbmwp`;

USE `dbmwp`;

-- CREATE TABLES

-- Creación de la tabla 'clients'

CREATE TABLE `Clients` (

    `IdClient`  *INT*, -- Clave primaria

    `nom`       *VARCHAR*(64) NOT NULL,

    `DNI`       *CHAR*(9) NOT NULL,

    `telefon`   *VARCHAR*(20) NOT NULL,

    `email`     *VARCHAR*(64) NOT NULL

);

-- Creación de la tabla 'cita'

CREATE TABLE `Cita` (

    `IdCita`    *INT*, -- Clave primaria

    `IdClient`  *INT* NOT NULL, -- Clave foránea hacia 'clients'

    `data`      *DATE* NOT NULL,

    `servei`    *VARCHAR*(64) NOT NULL,

    `IdVehicle` *INT* -- Clave foránea hacia 'Vehicles'

);

-- Creación de la tabla 'Vehicles'

CREATE TABLE `Vehicles` (

    `IdVehicle` *INT*, -- Clave primaria

    `IdClient`  *INT*, -- Clave foránea opcional hacia 'clients'

    `matricula` *VARCHAR*(20) NOT NULL,

    `model`     *VARCHAR*(64) NOT NULL,

    `any`       *INT* NOT NULL

);

-- PRIMARY KEYS

ALTER TABLE `Clients`

    MODIFY `IdClient` *INT* AUTO\_INCREMENT,

    ADD CONSTRAINT PK\_IdClient PRIMARY KEY (`IdClient`);

ALTER TABLE `Cita`

    MODIFY `IdCita` *INT* AUTO\_INCREMENT,

    ADD CONSTRAINT PK\_IdCita PRIMARY KEY (`IdCita`);

ALTER TABLE `Vehicles`

    MODIFY `IdVehicle` *INT* AUTO\_INCREMENT,

    ADD CONSTRAINT PK\_IdVehicle PRIMARY KEY (`IdVehicle`);

-- FOREIGN KEYS

ALTER TABLE `Cita`

    ADD CONSTRAINT FK\_Cita\_Cliente FOREIGN KEY (`IdClient`)

        REFERENCES Clients(`IdClient`),

    ADD CONSTRAINT FK\_Cita\_Vehicle FOREIGN KEY (`IdVehicle`)

        REFERENCES Vehicles(`IdVehicle`);

ALTER TABLE `Vehicles`

    ADD CONSTRAINT FK\_Vehicle\_Client FOREIGN KEY (`IdClient`)

        REFERENCES Clients(`IdClient`);

-- UNIQUE

ALTER TABLE `Clients`

    ADD CONSTRAINT UQ\_Client\_DNI UNIQUE (`DNI`),

    ADD CONSTRAINT UQ\_Client\_Email UNIQUE (`email`);

ALTER TABLE `Vehicles`

    ADD CONSTRAINT UQ\_Vehicle\_Matricula UNIQUE (`matricula`);

# Desenvolupament de la Solució

Llegados a esta parte se podría decir que estamos en el corazón de nuestro proyecto, donde hemos puesto en práctica todo lo que habíamos diseñado. Aquí vamos a contaros cómo hemos construido cada una de las partes clave de nuestro sistema: la API es la que se encarga de ser el puente entre todo, la parte web que ve el usuario y los scripts que nos ayudan con el mantenimiento. Hemos trabajado para que cada componente cumpla su función de forma robusta y, sobre todo, que todo funcione en conjunto como unos engranajes.

## API

**Lenguaje y Framework utilitzado:** Python y Flask

Decidimos usar Python junto con el framework Flask porque esta combinación nos ha resultado más cómoda porque ya teníamos algo de experiencia previa con Python, y Flask es una opción ligera y eficiente para construir una API.

Nuestra API se centra principalmente en la gestión de las citas y la información asociada a clientes y vehículos. La ruta principal con la que trabaja es /api/cita, a la que se realizan peticiones de tipo POST. Cuando un usuario envía el formulario desde la web, la API recibe todos esos datos (nombre, email, teléfono, detalles del coche y del servicio). Lo que hace la API es comprobar si el cliente o el vehículo ya existen en nuestra base de datos, para no tener datos duplicados. Si son nuevos, los registra; si ya están, usa su información existente. Una vez que tenemos al cliente y al vehículo gestionados, la API procede a insertar la nueva cita en la tabla Cita de la base de datos, enlazándola correctamente con el cliente y el vehículo. Resumiendo todo esto, la API se encarga de que toda la información que entra por la web se guarde de forma segura y organitzada en la base de datos.

## Web

La parte web es la cara visible de nuestro proyecto, lo que el usuario final ve y con lo que interactúa. La hemos hecho utilizando: HTML para la estructura, CSS para el diseño y el aspecto visual, y JavaScript para añadir interactividad y gestionar el envío de datos a nuestra API.

Hemos diseñado una interfaz sencilla y limpia, centrada en un formulario de servicio que facilita al usuario pedir su cita. Desde la página principal (**index.html**), el cliente puede introducir su información personal, los datos de su vehículo (marca, modelo, matrícula) y seleccionar el tipo de servicio que necesita, como por ejemplo, un cambio de aceito o revisiones, y también junto con la fecha preferida. El objetivo es que sea lo más directo posible para que cualquier persona pueda solicitar su cita sin complicaciones. El archivo main.js es el que se encarga de recoger toda esta información del formulario y enviarla a nuestra API (**http://localhost:5000/api/cita**) mediante una petición POST.

**Versión escritorio**

Interfaz de usuario gráfica, Aplicación

El contenido generado por IA puede ser incorrecto.

**Versión para móvil:**

Interfaz de usuario gráfica, Aplicación

El contenido generado por IA puede ser incorrecto.

### Operativa básica para pedir cita:

Para que un cliente pueda solicitar una cita, los pasos que seguiría son los siguientes:

1. **Acceso a la web:** El cliente accede a la página principal de "Taller Mecánico MWP" a través de un navegador web.
2. **Relleno del formulario:** En el formulario de servicio, el cliente introduce sus datos personales (Nombre Completo, Correo Electrónico, Teléfono), los detalles de su vehículo (Marca del Coche, Modelo del Coche, Matrícula del Coche), selecciona el tipo de servicio deseado de una lista predefinida (Cambio de Aceite, Servicio de Frenos, Servicio de Neumáticos, Inspección General) y elige la fecha preferida para la cita.
3. **Envío de la solicitud:** Una vez que todos los campos obligatorios están completados, el cliente pulsa el botón "Pedir Cita".
4. **Procesamiento y confirmación:** El JavaScript (**main.js**) de la página recoge todos los datos introducidos y los envía a la API (**app.py**). La API procesa la información, la guarda en la base de datos (creando o actualizando cliente y vehículo, e insertando la cita), y devuelve una respuesta. El cliente recibe una alerta en la web que le confirma si la cita se creó correctamente o si hubo algún error.

## Scripts

Para asegurar que nuestro sistema funcione correctamente y que la información esté segura, hemos desarrollado un par de scripts en Bash. Estos scripts nos ayudan a automatizar tareas.

### Script de còpia de seguretat de la base de dades

Este script lo que hace es generar un archivo .sql con toda la información de la base de datos y lo guarda en una carpeta que hemos llamado **./backups**. Además, le añade una marca de tiempo al nombre (por ejemplo, db\_backup\_2025-05-25\_17-30-00.sql), así podemos saber cuándo se hizo cada copia.

Texto

El contenido generado por IA puede ser incorrecto.

### Script de comprovació d'estat dels contenidors

Este otro script nos permite verificar rápidamente si todos los servicios de nuestro proyecto que están corriendo en Docker (la base de datos, la API y la web) están activos y funcionando como deberían. Esto nos ayuda a detectar y resolver posibles problemas a tiempo.

Texto

El contenido generado por IA puede ser incorrecto.

# Infraestructura de Sistemes

Para que todas las partes de nuestro proyecto funcionaran de forma coordinada y para que el despliegue fuera lo más sencillo posible, hemos optado por montar toda la infraestructura utilizando Docker y Docker Compose. Esto nos ha permitido "juntar" cada componente del sistema en su propio contenedor

## Explicación docker-compose

Hemos definido tres servicios principales en nuestro archivo **docker-compose.yml**, cada uno correspondiente a una pieza fundamental de nuestro sistema.

### Servicio Base de Datos (DB)

**Imagen:** Hemos usado la imagen oficial **mysql:8.0**, que es una versión estable y robusta de MySQL.

**Variables de entorno:** Aquí configuramos las credenciales de acceso a la base de datos (usuario **mwp**, contraseña y la base de datos **dbmwp**), lo cual es importante para la seguridad y para que la API pueda conectarse.

**Red:** Lo hemos configurado con una dirección IP estática (172.28.0.2) dentro de nuestra red interna mwp, facilitando así la comunicación con el resto de servicios.

**Persistencia de datos:** Hemos creado un volumen (**db\_data**) para asegurar que los datos de la base de datos se guarden de forma persistente. Esto significa que, si el contenedor de la base de datos se reinicia o se elimina, los datos no se pierden.

### Servicio API (api)

**Imagen:** En lugar de usar una imagen preexistente, nuestra API se construye a partir de un Dockerfile que está en la carpeta **./api**. Esto nos permite personalizar el entorno, instalar las dependencias de Python (requirements.txt) y copiar nuestro código de la API.

**Variables de entorno:** Aquí también pasamos las credenciales de la base de datos a la API (**DB\_HOST, DB\_USER, DB\_PASSWORD, DB\_NAME**), pero es importante destacar que el **DB\_HOST** lo hemos puesto como db. Esto es importante, ya que Docker Compose se encarga de que db se resuelva a la dirección IP del contenedor de la base de datos dentro de la red **mwp**.

**Red:** También se le asigna una IP estática (**172.28.0.3**) dentro de la red mwp para que exsita una comunicación.

**Exposición Puerto:** Hemos mapeado el puerto **5000** del contenedor al puerto **5000** de la máquina host (**ports: "5000:5000"**). Así, nuestra API es accesible desde fuera del entorno Docker, por ejemplo, para que la web pueda comunicarse con ella.

**Dependencia:** Hemos usado **depends\_on: - db** para indicarle a Docker Compose que la API no debe iniciarse hasta que el servicio de la base de datos (db) esté en marcha.

## Redes Personalizadas

Para facilitar la comunicación entre nuestros servicios y darles un orden, hemos definido una red personalizada llamada **mwp** en nuestro **docker-compose.yml**. Esta red (**172.28.0.0/24**) nos permite asignar direcciones IP estáticas a cada uno de nuestros contenedores (**db** en **172.28.0.2**, **api** en **172.28.0.3**, y **web** en **172.28.0.4**). Esto nos da un control más preciso sobre cómo se comunican y nos ayuda a asegurar que los servicios siempre se encuentren entre sí, incluso si se reinician.

Texto

El contenido generado por IA puede ser incorrecto.

Texto

El contenido generado por IA puede ser incorrecto.

Interfaz de usuario gráfica, Texto

El contenido generado por IA puede ser incorrecto.

# Problemes i Solucions

**Problema: Adaptación del modelo de datos y la conexión de la API de T-SQL a MySQL**

Al principio de nuestro proyecto, quien se encargó de hacer la parte de la base de datos fué con la idea de trabajar con T-SQL (Transact-SQL), un lenguaje SQL utilizado por sistemas como Microsoft SQL Server. Pero al final por motivos de comodidad/estar acostumbrados más a MySQL optamos por cambiar. Entonces a la hora de cambiarlo todo para que funicone con MySQL nos costó un poco porque nuestro conocimiento sobre API/Flask es casi nulo y nos costó un poco reeformarlo todo de nuevo para que fucnionase.

El cambio implicó que nuestro diseño de base de datos, y sobre todo la forma en que la API interactuaba con ella, necesitaba una adaptación importante. Las sintaxis SQL, aunque parecidas, tienen sus diferencias entre T-SQL y MySQL. Esto nos causó algunos problemas de conexión y ejecución de consultas en la API, ya que estaba intentando "hablar" en un dialecto SQL que MySQL no entendía del todo bien.

**Solución:**

La solución pasó por una revisión de nuestro script de base de datos **(db\_schema.sql**) para asegurarnos de que toda la sintaxis estuviera en el dialecto correcto de MySQL. Se tuvo que ajustar algunos comandos y datos. Además, la conexión desde la API (desarrollada en Python con Flask) tuvo que ser modificada. Inicialmente, las librerías o la forma de conectar podían estar pensadas para T-SQL, así que tuvimos que cambiarlas también para que estas funcionasen con MySQL.

# Conclusions

Durante este proyecto sí que hemos aprendido más o menos a cómo trabajar, en este caso, con Flask y Python para el tema de la API. Esta se encarga un poco de juntar gran parte de las piezas que componen este proyecto. Nuestro aprendizaje no ha sido tan grande en algunas áreas, pero sí que podemos decir que hemos aprendido a cómo enfrentarnos a algo que desconocemos más o menos. Gracias a otras partes que se incluyen en el proyecto, como la base de datos o la web, sí las hemos aprendido y afianzado durante el curso cosa que nos ha hecho que el camino se haga un poco más a meno.

El reto difícil lo centraríamos en Docker y en hacer que todo funcione en conjunto. Hemos aprendido a saber desarrollar un poco de "control" sobre lo que se tiene que hacer o cómo hacerlo, especialmente en la orquestación de servicios. Si tuviéramos más tiempo, nos gustaría mejorar la validación de los datos en el servidor (en la API) para hacerla más robusta, y quizás explorar la posibilidad de añadir más funcionalidades para el taller, como una interfaz de administración para gestionar citas y clientes directamente. En general, este proyecto nos ha servido para entender un poco más cómo se monta una aplicación completa desde cero y cómo se interconectan sus diferentes partes.

# Annexos