**Manual Del Programador Proyecto Veterinaria**

Juan Pablo Adams Parra

Isac Cortes Buitrago

Emmanuel Bolivar Marin

Juan Jose Estrada Velez

Politécnico Jaime Isaza Cadavid.

Tabla de contenido

[Introducción 6](#_Toc152612130)

[Arquitectura del sistema 7](#_Toc152612131)

[Modelo (Model) 7](#_Toc152612132)

[Vista (View): 7](#_Toc152612133)

[Controlador (Controller): 8](#_Toc152612134)

[Flujo de Trabajo: 8](#_Toc152612135)

[Ventajas de MVC: 8](#_Toc152612136)

[Configuración del Entorno de desarrollo 8](#_Toc152612137)

[Requisitos de Software: 8](#_Toc152612138)

[Configuración del Entorno: 9](#_Toc152612139)

[Estructura de las carpetas : 10](#_Toc152612140)

[Back-end 10](#_Toc152612141)

[Estructura del Código Fuente: 14](#_Toc152612142)

[Base De datos 14](#_Toc152612143)

[Estructura de la Base de Datos 14](#_Toc152612144)

[Tablas de la base de datos 14](#_Toc152612145)

[Procedimientos de almacenado 17](#_Toc152612146)

[Front-end 19](#_Toc152612147)

[Dentro del directorio Front-end se incluyen las siguientes carpetas: 19](#_Toc152612148)

[Conclusiones 24](#_Toc152612149)

[Referencias 26](#_Toc152612150)

# Introducción

En el mundo contemporáneo, la integración de la tecnología desempeña un papel esencial en la optimización y eficiencia de diversos sectores, y la atención veterinaria no es la excepción. En este contexto, se presenta un proyecto web innovador diseñado para revolucionar la gestión interna de una clínica veterinaria, utilizando tecnologías de vanguardia como Angular, Node.js y MySQL, con la aplicación del sólido patrón de arquitectura Modelo Vista Controlador (MVC). Este proyecto no solo busca modernizar los procesos administrativos de la veterinaria, sino también mejorar la calidad de atención al cliente y la eficacia en el cuidado de las mascotas.

El enfoque principal del sistema se basa en la gestión integral de roles, adaptando la plataforma a las necesidades específicas de distintos actores dentro de la clínica. Desde veterinarios que requieren acceso a historias clínicas y datos médicos, hasta vendedores encargados de gestionar productos y facturas, el sistema asigna responsabilidades de manera eficiente. La inclusión de roles como administrador y dueño permite una supervisión completa del sistema y una gestión estratégica de la información.

Además de la asignación de roles, el proyecto aborda la administración de datos cruciales para el funcionamiento de la veterinaria. Desde la gestión de personas, incluyendo clientes y personal interno, hasta el seguimiento detallado de mascotas y sus historias clínicas, el sistema proporciona una plataforma completa para el almacenamiento y acceso rápido a información relevante. La gestión integrada de facturas y productos contribuye aún más a la eficiencia operativa, permitiendo un control preciso sobre las transacciones comerciales y el inventario de la clínica. En resumen, este proyecto web representa una solución integral que fusiona la tecnología avanzada con la atención veterinaria, marcando un hito en la evolución de la gestión clínica para el bienestar de las mascotas y la comodidad de los propietarios.

# Arquitectura del sistema

La arquitectura utilizada es La arquitectura Modelo-Vista-Controlador (MVC) es un patrón de diseño ampliamente utilizado en el desarrollo de software que separa la lógica de la aplicación en tres componentes principales: Modelo, Vista y Controlador. Esta separación facilita la modularidad, el mantenimiento y la escalabilidad de las aplicaciones. Aquí se describe cada componente de la arquitectura MVC:

## Modelo (Model)

* **Responsabilidad**

El Modelo representa la lógica de la aplicación y la manipulación de datos. Es responsable de acceder a la base de datos, procesar la información y realizar las operaciones de lógica empresarial

* Caracteristicas
* Contiene la representación de los datos y las reglas de negocio.
* Puede enviar notificaciones de cambios a las Vistas.
* No tiene conocimiento de las Vistas o Controladores.

## Vista (View):

* Responsabilidad: La Vista es responsable de la presentación de la información al usuario y de la interfaz de usuario. Muestra los datos provenientes del Modelo y reacciona a las acciones del usuario, enviando eventos al Controlador.
* Características:
  + Presenta la información de manera visual.
  + Recibe datos del Modelo y los muestra al usuario.
  + Puede enviar eventos al Controlador.

## Controlador (Controller):

* Responsabilidad: El Controlador actúa como intermediario entre el Modelo y la Vista. Recibe eventos de entrada del usuario desde la Vista, procesa la lógica de la aplicación y actualiza el Modelo en consecuencia. También actualiza la Vista para reflejar los cambios en el Modelo.
* Características:
  + Maneja eventos de entrada del usuario desde la Vista.
  + Actualiza el Modelo en respuesta a acciones del usuario.
  + Actualiza la Vista para reflejar cambios en el Modelo.

## Flujo de Trabajo:

1. El usuario interactúa con la Vista, generando eventos (clics, entradas, etc.).
2. La Vista envía estos eventos al Controlador.
3. El Controlador procesa los eventos y actualiza el Modelo según sea necesario.
4. El Modelo, al ser actualizado, notifica a la Vista.
5. La Vista recupera datos actualizados del Modelo y se actualiza visualmente.
6. El flujo de interacción continua según las acciones del usuario.

## Ventajas de MVC:

* Separación de Responsabilidades: La separación clara de las responsabilidades facilita el mantenimiento y la escalabilidad del código.
* Reutilización de Código: Los componentes son independientes y pueden ser reutilizados en diferentes partes de la aplicación.
* Facilita el Desarrollo en Equipo: Diferentes desarrolladores pueden trabajar en paralelo en Modelos, Vistas y Controladores sin interferencias significativas.

# Configuración del Entorno de desarrollo

## Requisitos de Software:

* HTML
* CSS
* Javascript
* Typescript
* Node.js
* Express
* Angular
* MySQL

## Configuración del Entorno:

* **En consola, ubicado en la carpeta veterinariaFrontend:**

Ejecutar "npm install" Para instalar las dependencias necesarias para el proyecto Angular.

Ejecuar "ng serve --o" o "ng serve" para ejecutar en modo desarrollador el proyecto (Es necesario tener intalado @angular/cli de manera global).

En caso de no tener @angular/cli instalado, Ejecuar "npm run start" o "npm start" para ejecutar en modo desarrollador el proyecto

* **En consola, ubicado en la carpeta veterinariaBackendjs:**

Ejecutar "npm install" Para instalar las dependencias necesarias para el proyecto Express y MySql.

Ejecutar "npm run dev" para ejecutar el proyecto en modo desarrollador mediante la herramienta nodemon o "npm start" para inicar el proyecto.

Para La Base de Datos, en la carptera veterinariaBackendjs/sql se encuentra en script de la creacion de la base de datos.

Por defecto, El proyecto está utilizando el puerto localhoost:3306 para la ejecución de la base de datos, puerto iniciado por defecto al instalar MySql con la configuración base.

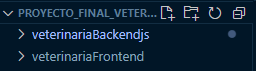
Para los llamados a la base de datos, se está utiliando por defecto el usuario: "root" con una password: "1234".

Para la Ejeucion del Proyecto, es necesario tener los 3 servicios mencionados anteriormente activos simultáneamente.

## Estructura de las carpetas :

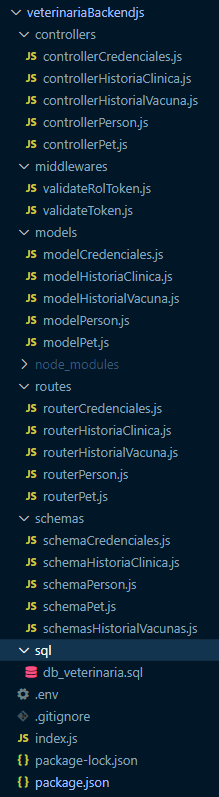
El proyecto se divide en 2 carpetas principales, como se muestra en la imagen siguiente:

**Figura 1**  
Foto carpetas de Backend y Frontend del proyecto veterinaria

s

## Back-end

**Figura 2**  
Foto carpetas que componen el Backend



**Dentro del directorio Back-end se incluyen las siguientes carpetas:**

* **Controllers:** contiene los controladores para cada modelo de la aplicación en el siguiente código se tendrá un ejemplo base de lo que se hace con los controladores
* *import* { validateCred, validateParcialCred } from '../schemas/schemaCredenciales.js'
* export *class* CredController {
* *constructor* ({ *CredModel* }) {
* this.CredModel = CredModel
* }
* getAll = async (*req*, *res*) *=>* {
* *const* creds = await this.CredModel.getAll()
* res.json(creds)
* }
* getByUser = async (*req*, *res*) *=>* {
* *const* { user } = req.params
* *const* cred = await this.CredModel.getByUser({ user })
* if (cred.err) {
* res.status(404).json(cred)
* } else {
* res.json(cred)
* }
* }
* create = async (*req*, *res*) *=>* {
* *const* result = validateCred(req.body)
* if (result.error) {
* res.status(400).json({ err: JSON.parse(result.error.message) })
* } else {
* *const* cred = await this.CredModel.create({ data: result.data })
* if (cred.err) {
* res.status(400).json(cred)
* } else {
* res.json(cred)
* }
* }
* }
* delete = async (*req*, *res*) *=>* {
* *const* { user } = req.params
* *const* cred = await this.CredModel.delete({ user })
* if (cred.err) {
* res.status(400).json(cred)
* } else {
* res.json(cred)
* }
* }
* update = async (*req*, *res*) *=>* {
* *const* result = validateParcialCred(req.body)
* if (result.error) {
* res.status(400).json({ err: JSON.parse(result.error.message) })
* } else {
* *const* { oldUser } = req.params
* *const* updatedcred = await this.CredModel.update({ oldUser, data: result.data })
* if (updatedcred.err) {
* res.status(400).json(updatedcred)
* } else {
* res.json(updatedcred)
* }
* }
* }
* login = async (*req*, *res*) *=>* {
* *const* result = validateParcialCred(req.body)
* if (result.error) {
* res.status(400).json({ err: JSON.parse(result.error.message) })
* } else {
* *const* loginUser = await this.CredModel.login({ data: result.data })
* if (loginUser.err) {
* res.status(400).json(loginUser)
* } else {
* res.json(loginUser)
* }
* }
* }
* }
* **middlewares:** contiene archivos de creación y validación de credenciales
* **models:** contiene los archivos CRUD para cada tabla de la base de datos.
* **node\_modules:** contiene los paquetes instalados de node.
* **routes:** configura un enrutador de Express con rutas relacionadas con la gestión de credenciales. La seguridad está implementada mediante la validación de tokens y roles antes de permitir el acceso a ciertas rutas. Además, hay una ruta específica para manejar la autenticación , aquí podemos ver un ejemplo base
* *import* { Router } from 'express'
* *import* { CredController } from '../controllers/controllerCredenciales.js'
* *import* { validateToken } from '../middlewares/validateToken.js'
* *import* { validateRolToken } from '../middlewares/validateRolToken.js'
* export *const* createCredRouter = ({ *CredModel* }) *=>* {
* *const* credRouter = Router()
* *const* credController = new CredController({ CredModel })
* credRouter.get('/', validateToken, validateRolToken([1]), credController.getAll)
* credRouter.get('/:user', validateToken, validateRolToken([1]), credController.getByUser)
* credRouter.post('/', validateToken, validateRolToken([1]), credController.create)
* credRouter.delete('/:user', validateToken, validateRolToken([1]), credController.delete)
* credRouter.patch('/:oldUser', validateToken, validateRolToken([1]), credController.update)
* credRouter.post('/login', credController.login)
* return credRouter
* }
* **schemas:** contiene las validaciones pertinentes de los datos que son llevados al modelo.
* sql: contiene el script creación de la base de datos

## Estructura del Código Fuente:

## Base De datos

### Estructura de la Base de Datos

#### Tablas de la base de datos

* **Persona:** Almacena información sobre personas, incluyendo veterinarios, dueños de mascotas, y personal administrativo.

CREATE TABLE Persona (

    cedula *INT* NOT NULL UNIQUE primary key,

    Primer\_nombre *VARCHAR*(30) NOT NULL,

    Segundo\_nombre *VARCHAR*(30) NULL,

    Primer\_Apellido *VARCHAR*(30) NOT NULL,

    Segundo\_Apellido *VARCHAR*(30) NOT NULL,

    edad *INT* NOT NULL,

    IdRol *INT* NOT NULL

);

* **Rol**: Define los roles asignados a las personas (Ejemplo: Administrador, Veterinario, Vendedor).

CREATE TABLE Rol (

    IdRol *INT* NOT NULL AUTO\_INCREMENT,

    NombreRol *VARCHAR*(20) NOT NULL,

    PRIMARY KEY (IdRol)

);

* **Credenciales:** Contiene información de inicio de sesión para cada persona .

CREATE TABLE Credenciales (

    Usuario *varchar*(30) not null unique primary key,

    Contrasenia *varchar*(255),

    idPersona *INT* NOT NULL

);

* **Mascota**: Almacena datos específicos de las mascotas, como nombre, edad, especie, etc.

CREATE TABLE Mascota (

    IdMascota binary(16) NOT NULL UNIQUE,

    Nombre *VARCHAR*(20) NOT NULL,

    Edad *INT* NOT NULL,

    Especie *VARCHAR*(20) NOT NULL,

    Raza *VARCHAR*(20) NOT NULL,

    Color *VARCHAR*(20) NOT NULL,

    Tamanio *VARCHAR*(10) NOT NULL,

    Peso *VARCHAR*(10) NOT NULL,

    IdDuenio *INT* NOT NULL,

    PRIMARY KEY (IdMascota)

);

* **Historia\_Clinica:** Registra la historia clínica de cada mascota, incluyendo detalles sobre motivos de consulta, diagnósticos, tratamientos, etc

CREATE TABLE Historia\_Clinica (

    IdHistoria\_Clinica *INT* NOT NULL UNIQUE AUTO\_INCREMENT,

    Fecha *DATE* NOT NULL,

    Motivo *VARCHAR*(200) NOT NULL,

    Sintomatologia TINYTEXT NOT NULL,

    Diagnostico *TEXT* NOT NULL,

    Procedimiento *TEXT* NULL,

    MedicamentosAlergia *VARCHAR*(100) NOT NULL,

    IdMascota binary(16) NOT NULL,

    IdOrden *INT* NULL,

    IdVeterinario *INT* NOT NULL,

    PRIMARY KEY (IdHistoria\_Clinica)

);

.

* **Historial\_Vacunas**: Registra el historial de vacunación de las mascotas.

create table Historial\_Vacunas (

    IdHistorialVacunas *INT* NOT NULL UNIQUE AUTO\_INCREMENT PRIMARY KEY,

    Fecha *DATE* NOT NULL,

    IdVacuna *INT* NOT NULL,

    IdMascota binary(16) NOT NULL

);

* **Vacuna:** Contiene información sobre las vacunas disponibles.

create table Vacuna (

    IdVacuna *INT* NOT NULL UNIQUE AUTO\_INCREMENT PRIMARY KEY,

    nombre *varchar*(50) not null

);

* **Orden:** Gestiona las órdenes asociadas a las mascotas.

create table Orden (

    IdOrden *INT* NOT NULL UNIQUE AUTO\_INCREMENT,

    IdMascota binary(16) NOT NULL,

    Anulada *boolean* not null default false,

    primary key (IdOrden)

);

* **Orden\_Medicamento:** Relaciona órdenes con medicamentos y registra las dosis.

create table orden\_Medicamento (

    IdOrden *INT* NOT NULL,

    IdMedicamento *INT* NOT NULL,

    Dosis *varchar*(15) not null,

    primary key(IdOrden, IdMedicamento)

);

* **Medicamento:** Almacena información sobre medicamentos disponibles.

create table Medicamento (

    IdMedicamento *INT* NOT NULL AUTO\_INCREMENT PRIMARY KEY,

    nombre *varchar*(50) not null

);

* **Factura:** Registra información sobre las facturas generadas.

CREATE TABLE Factura (

    IdFactura *INT* NOT NULL UNIQUE AUTO\_INCREMENT,

    valorTotal FLOAT NOT NULL,

    fecha *DATE* NOT NULL,

    IdOrden *INT* NULL,

    IdDuenio *INT* NOT NULL,

    PRIMARY KEY (IdFactura)

);

* **Factura\_Producto:** Asocia facturas con productos y registra cantidades.

create table Factura\_Producto (

    IdFactura *INT* NOT NULL,

    IdProducto *INT* NOT NULL,

    cantidad *int* not null default 0,

    primary key(IdFactura, IdProducto)

);

* **Producto**: Contiene información sobre los productos disponibles.

create table Producto (

    IdProducto *INT* NOT NULL AUTO\_INCREMENT PRIMARY KEY,

    nombre *varchar*(50) not null,

    valor float not null

);

## Procedimientos de almacenado

En el siguiente código podemos ver la base de como son los procedure de cada tabla

*/\*CRUD tabla Persona\*/*

Drop procedure if exists Create\_Persona;

DELIMITER &&

CREATE PROCEDURE Create\_Persona (in cedula *INT*,

    in Primer\_nombre *VARCHAR*(30) ,

    in Segundo\_nombre *VARCHAR*(30),

    in Primer\_Apellido *VARCHAR*(30),

    in Segundo\_Apellido *VARCHAR*(30),

    in edad *INT*,

    in IdRol *INT*)

BEGIN

    insert into Persona values (cedula,Primer\_nombre,Segundo\_nombre,Primer\_Apellido,Segundo\_Apellido,edad,IdRol);

END &&

DELIMITER ;

Drop procedure if exists Actualizar\_Persona;

DELIMITER &&

CREATE PROCEDURE Actualizar\_Persona (in id *INT*,

    in P\_nombre *VARCHAR*(30) ,

    in S\_nombre *VARCHAR*(30),

    in P\_Apellido *VARCHAR*(30),

    in S\_Apellido *VARCHAR*(30),

    in P\_edad *INT*,

    in Rol *INT*)

BEGIN

     update Persona set Primer\_nombre = P\_nombre , Segundo\_nombre = S\_nombre,

     Primer\_Apellido = P\_Apellido, Segundo\_Apellido = S\_Apellido,edad = P\_edad, IdRol = Rol

     where cedula=id;

END &&

DELIMITER ;

Drop procedure if exists Consultar\_Persona;

DELIMITER &&

CREATE PROCEDURE Consultar\_Persona (in id *INT*)

BEGIN

     select \* from Persona where cedula=id;

END &&

DELIMITER ;

Drop procedure if exists Eliminar\_Persona;

DELIMITER &&

CREATE PROCEDURE Eliminar\_Persona (in id *INT*)

BEGIN

     delete from Persona where cedula=id;

END &&

DELIMITER ;

## Front-end

### Dentro del directorio Front-end se incluyen las siguientes carpetas:

* **Angular:** contiene los procesos para
* **Vscode:** contiene archivos de creación y validación de credenciales
* **components:** como dice el titulo es la carpeta que contiene los componentes básicos para el diseño web de la aplicación como el footer , el carrusel y los formularios
* **Navnar:** contiene los archivos para
* **Guards:** estos guards son funciones que se usan en Angular para proteger ciertas rutas o componentes en la aplicación. Se ejecutarán antes de permitir el acceso a una ruta y verificarán las condiciones de autenticación y roles antes de permitir o denegar el acceso
* **Interfaces:** Estas son interfaces en TypeScript que definen la estructura o forma esperada de los objetos:
* **Pages:** contiene los archivos para el funcionamiento correcto de las ventanas de la aplicación como su diseños y archivos de confirmación
* **Services:** services permite la conexión con el back mediante peticiones http
* **Utils:** contiene las validaciones pertinentes de los datos que son llevados al modelo.
* **Assets:** contiene las imágenes que se usan en la aplicación

# Conclusiones

En el transcurso del desarrollo de nuestra aplicación, la elección estratégica de Node.js, TypeScript y Angular ha demostrado ser esencial para alcanzar nuestros objetivos de manera eficaz. Node.js ha desempeñado un papel crucial al brindar un entorno de ejecución eficiente y escalable, permitiéndonos crear una aplicación robusta y de alto rendimiento de manera ágil.

La introducción de TypeScript ha mejorado significativamente la calidad de nuestro código al incorporar tipos estáticos. Este enfoque no solo ha facilitado la detección temprana de errores, sino que también ha mejorado la legibilidad y la mantenibilidad del código, garantizando un desarrollo confiable y eficiente a lo largo del tiempo.

Angular, como marco de desarrollo front-end, ha sido esencial para la construcción de interfaces de usuario interactivas y dinámicas en nuestra aplicación. Su arquitectura basada en componentes ha simplificado la gestión de la interfaz de usuario, facilitando tanto la fase inicial como la evolución continua de la aplicación.

En el ámbito de la gestión de datos, la elección de MySQL como sistema de gestión de bases de datos ha sido acertada. Su rendimiento confiable y su capacidad de escalabilidad nos han permitido almacenar y recuperar datos de manera eficiente, contribuyendo a la solidez y eficacia de nuestra aplicación.

En resumen, la combinación de Node.js, TypeScript, Angular y MySQL ha sido fundamental para el éxito de nuestra aplicación. Este conjunto de tecnologías nos ha proporcionado un entorno coherente y poderoso que maximiza nuestra productividad y eficiencia, estableciendo una base sólida para el éxito continuo de nuestra aplicación.

# Referencias

Angular. (2023). Guía del desarrollador. Recuperado de https://angular.io/guide

Node.js. (2023). Documentación. Recuperado de https://nodejs.org/en/docs/