Veterinaria Web App

Tinajero Anaya Bryam David

Universidad Autónoma de Querétaro

Índice

Índice....2
Introducción....3
Diagrama E/R de la Base de Datos....4
Flujograma....5
Plataforma web....6
- Instalación y configuración inicial....6
- Creación de modelos, migraciones, paneles y recursos para el panel admin....7
- Desarrollo de lógica de recursos para el panel admin....7
- Anexacion de recursos al panel veterinario....9
- Creación del panel usuario....9
- Creación de un login global....9
- Restricciones a paneles usuario y veterinario....10

Application Programming Interface....11
- Creación de API en proyecto....11
- Desarrollo de lógica de acceso por peticiones y redireccionamiento....12

Aplicación Móvil....13
- Home e Inicio de Sesión....13
- Catálogo de Productos....13
- Citas....14
- Consultas....14
- Mascotas....14
- Datos de Usuario y Veterinario....14
- Notificaciones....14
- Registro y Autenticación....15
- Base de Datos Local....15

Recuperación de contraseña....15

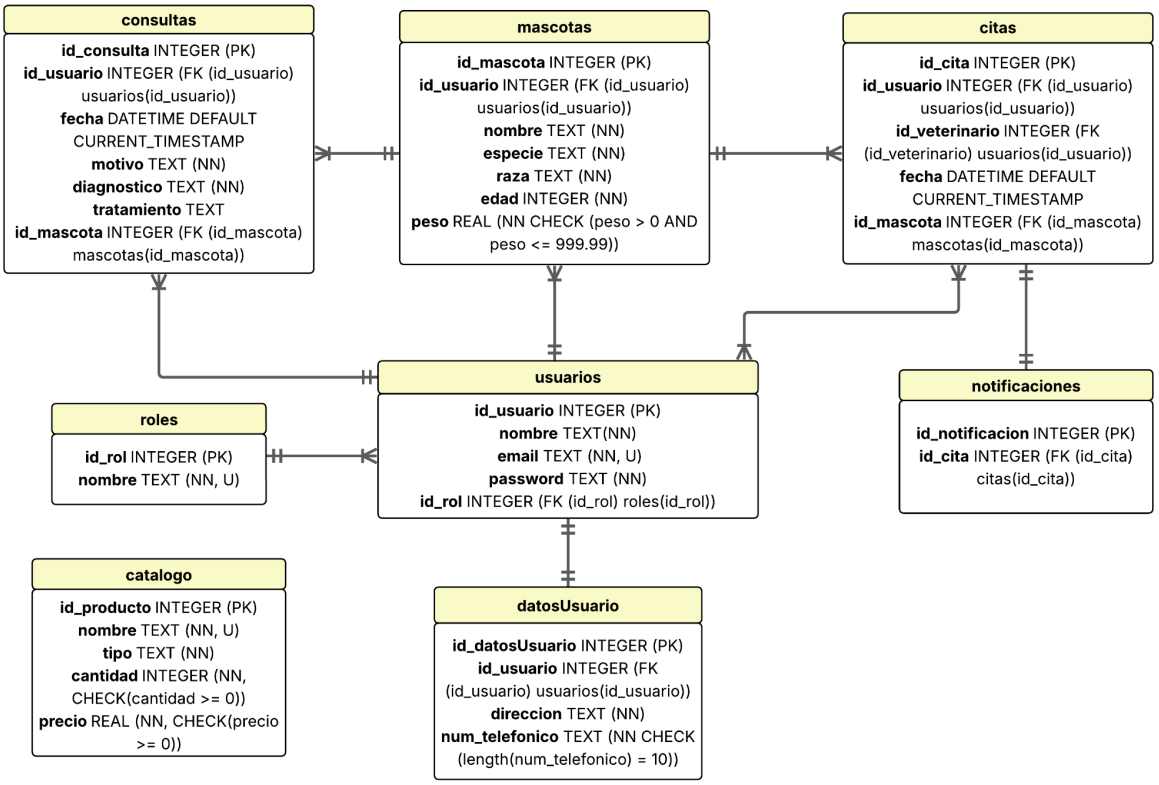
Introducción

Este proyecto surge a partir de un proyecto final con el objetivo de desarrollar un sistema completo de gestión veterinaria que abarque una plataforma web, una aplicación móvil y un API para comunicar ambas partes. Se buscó centralizar la administración de usuarios, veterinarios, mascotas, consultas y citas, todo con distintos niveles de acceso y seguridad.
Para la parte web opté por usar Laravel con Filament, lo que me permitió estructurar de manera rápida los recursos, modelos y paneles para los tres tipos de usuarios por roles: usuario, veterinario y administrador. Para definir adecuadamente los roles construí toda la lógica de permisos y relaciones directamente sobre el framework, reutilizándola después en la API. También se integró un login global que redirige al panel adecuado de manera automática según el tipo de usuario.

La API fue construida sobre el mismo proyecto con Laravel Sanctum para manejar la autenticación mediante tokens requeridos en casi todas las rutas. Esto permitió una integración directa con la app móvil, que se desarrolló en Android Studio usando Java y XML. Cada pantalla de la app está vinculada a una funcionalidad específica de la API, respetando las restricciones por rol del usuario. Para el manejo de sesiones, tokens y persistencia, implementé SQLite dentro del dispositivo.

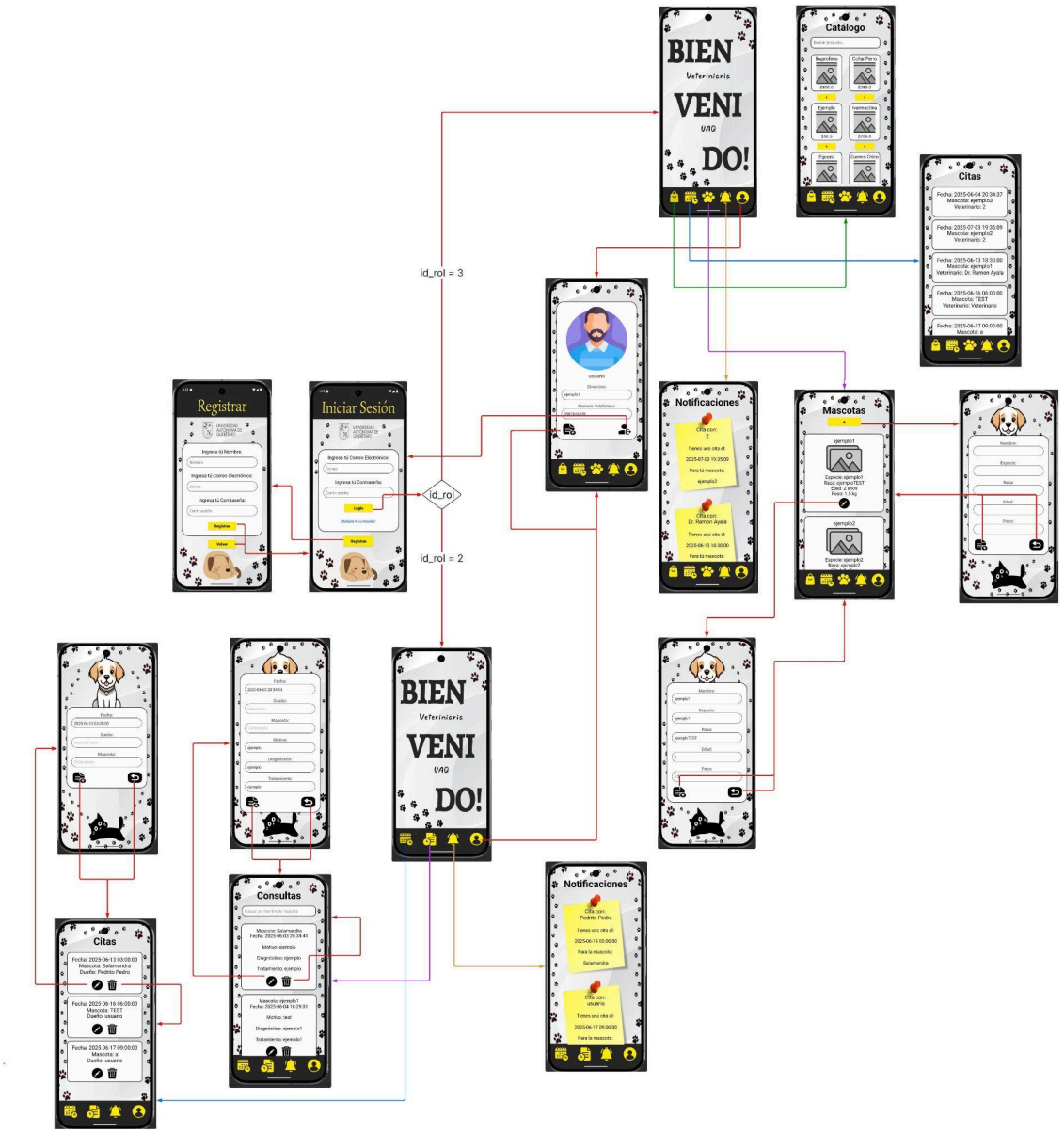
El enfoque durante todo el desarrollo fue garantizar una estructura clara, segura y modular, con validaciones tanto del lado del cliente como del servidor, y con una experiencia de usuario adaptada al tipo de usuario logueado. El resultado es una solución funcional, escalable y con buena separación entre lógica, vista y acceso a datos.

Diagrama E/R de la Base de Datos



Flujograma

https://lucid.app/lucidchart/17cfdffa-e153-4738-86b6-7b26f536813d/edit?viewport_loc=700%2C273%2C2782%2C1418%2C0_0&invitationId=inv_8068414f-a5d7-48e3-b3da-b4d17c788523



Plataforma web

Para cumplir con los requerimientos del proyecto es necesario crear una plataforma web donde se maneje la administración de todos los recursos, decidí emplear laravel con filament para esta parte del proyecto; en la plataforma web se definirán las tablas de la base de datos, se crearán 3 niveles de acceso distintos: admin, veterinario y usuario; con distinción en permisos y acceso a recursos.

La lógica desarrollada para las relaciones entre tablas y las restricciones en recursos también será aprovechada para el desarrollo de la API posteriormente.

Instalación y configuración inicial

- Correr Apache en XAMPP. (Verificar que la bd a utilizar este habilitada en el php.ini en este caso sqlite)
- Crear proyecto laravel
    - ***laravel new larafilament***
        - sin kit y sqlite como bd
            - instalar sqlite si no se tiene desde el cmd
    - meterse a la carpeta del proyecto creada
        - ***cd larafilament***
            - *Es importante recordar esta dirección ya que aquí es donde se ejecutará el portal siempre.*
- Instalar filament
    - https://filamentphp.com/docs/3.x/panels/installation
        - ***composer require filament/filament:"^3.3" -W***
        - ***php artisan filament:install --panels***
            - creará un panel con el nombre dado, en este proyecto serán 2 paneles admin y veterinario, al inicio se crea el panel admin.
        - Creación de la BD sqlite para poder almacenar usuarios y demás datos.
            - ***type nul > database\database.sqlite***
            - ***php artisan migrate***
        - ***php artisan make:filament-user***
            - llenar los datos y guardarlos
                - ejemplo: admin, bryamdta@gmail.com, admin
- Validar el funcionamiento
    - Correr el servidor
        - ***php artisan serve***
        - acceder a http://127.0.0.1:8000/admin
        - llenar los datos previamente ingresados y darle a login
- Para restringir el acceso a usuario al panel admin de modo en que solo el admin pueda acceder hay que realizar lo siguiente:
    - Dentro de la carpeta App\Models\ el codigo User.php se modifica según lo requerido, en este caso agregue una función que realice la validación y vincule User.php con los models/tablas Rol.php y DatosUsuario.php.
- Hay que crear el model rol para la validación

    - ***php artisan make:model Rol -m***
    - en Rol.php agregar el código correspondiente para los datos de la tabla rol

Creación de modelos, migraciones, paneles y recursos para el panel admin

- Se crea el model DatosUsuario.php con los datos de la BD.
    - ***php artisan make:model datosUsuariol -m***
- Y así sucesivamente con todas las tablas
- Migrar todos los models que aparecerán en la carpeta migrations dentro de la carpeta database
    - **php artisan make:migration User**
- Ahora migrados los modelos, acceder a la carpeta migrations y en cada migración realizar las creaciones de cada tabla en la function up y una función de drop en la function down.
- Finalizadas todas las definiciones de tablas en las migraciones hay que ejecutar las migraciones.
    - ***php artisan migrate***
- Para poder acceder a los paneles debido a las restricciones y a que la tabla usuarios está vacía se agrega un usuario admin para la visualización mediante tinker.
    - ***php artisan tinker***
    - Se crea el rol de admin
        - ***use App\Models\Roles;***
          Roles::create(['nombre' => 'Administrador']);
    - Se crea el usuario admin
        - ***use App\Models\User;***
          User::create([
          'nombre' => 'Admin',
          'email' => 'admin@admin.com',
          'password' => bcrypt('admin123'),
          'id_rol' => 1,
          ]);
- Hay que crear las tablas sessions y cache para que funcione el login.
    - ***php artisan cache:table***
    - ***php artisan session:table***
    - ***php artisan migrate***
- Además es necesario modificar en config/auth.php en ‘providers’ agregar el siguiente codigo para que el login busque los datos en la tabla usuarios.
    - ***'table' => 'usuarios',***
- Ahora sigue crear los recursos con filament anexados al panel de admin, como el admin tiene acceso a todo se le permite el CRUD en todos los recursos.
    - ***php artisan make:filament-resource Mascotas --panel=admin***

Desarrollo de lógica de recursos para el panel admin

- Una vez creados todos los recursos agregar los espacios de form de los datos a manipular con relaciones y constraints dentro del return $form, dependiendo el tipo

de dato puede ser un componente TextInput o un Select, es importante importar los elementos a usar.

    - ***use Filament\Forms\Components\TextInput;***
      use Filament\Forms\Components\Select;

- Ya marcados todos los campos rellenables en el $form ahora agregar los datos visualizables al return $table, nuevamente es necesario importar los tipos de elementos usados.
    - ***use Filament\Tables\Columns\TextColumn;***
- Se realizan inserts desde el panel para comprobar el adecuado funcionamiento y poder visualizar datos.
- Agregué lógica a los siguientes resources para solo mostrar datos relevantes, ejemplo: si se selecciona en citas un usuario dueño solo se muestran las mascotas vinculadas a ese usuario o solo permite seleccionar como dueño a un usuario con el rol de usuario o a un veterinario solo sí tiene el rol de veterinario.
    - Citas
        - Usuario seleccionable se muestra solo si cumple con el rol de usuario.
        - Veterinario seleccionable se muestra solo si cumple con el rol de veterinario.
        - Mascota seleccionable tras seleccionar un usuario y se muestra sólo si pertenece al usuario seleccionado.
    - Consultas
        - Usuario seleccionado se muestra solo si cumple con el rol de usuario.
        - Mascota seleccionable tras seleccionar un usuario y se muestra sólo si pertenece al usuario seleccionado.
    - Datos Usuario
        - Usuario seleccionable se muestra solo si cumple con el rol de usuario.
    - Mascotas
        - Usuario seleccionable se muestra solo si cumple con el rol de usuario.
    - En las relaciones con usuario y veterinario es fácil recuperar los datos que cumplen usando el parámetro
        - ***modifyQueryUsing: fn ($query) =>$query->where('id_tabla_relacionada',valor_id)***

      dentro de la función relationship dentro del componente Select del dato en el $form de la tabla.
    - Para poder recuperar los nombres de mascotas tras seleccionar un usuario es más complejo, para empezar se requiere agregar la función *reactive()* al Select del usuario y en el Select de mascota agregar el método options involucrando la función **function (callable $get)** donde se recupera el valor seleccionado en ‘id_usuario’ y se asigna a la variable $usuarioId, posteriormente se agrega un condición donde si la variable $usuarioId es nula no se retorna nada, es decir no se muestran mascotas, pero en caso que sí, recuperando el modelo Mascotas se realiza una consulta *where()* donde se checa si el valor recuperado de la bd en ‘id_usuario’ es igual al recuperado mediante la función $get, $usuarioId, se genera la lista de opciones en el options mediante pluck() mostrando la columna nombre.
        - ***Select::make('id_mascota')->label('Mascota')->options(function (callable $get) {***

```
$usuarioId = $get('id_usuario');
if (!$usuarioId) {
return [];
}
return \App\Models\Mascotas::where('id_usuario', $usuarioId)
->pluck('nombre', 'id_mascota');})->required()
```

- *Con esto terminado ya quedó toda la lógica del panel de administrador y a continuación sigue crear el de veterinario.*

Anexacion de recursos al panel veterinario

- De acuerdo a los requerimientos dados en clase, el veterinario puede acceder a la plataforma web sin embargo sólo tendrá acceso a visualizar las tablas citas y consultas.
- Para agregar los recursos mencionados se agrega este método en app/filament/VeterinarioPanelProvider, esto le permite acceder.
    - ***->resources([***
      \App\Filament\Resources\CitasResource::class,
      \App\Filament\Resources\ConsultasResource::class,
      \App\Filament\Resources\DatosUsuarioResource::class,
      ])

Creación del panel usuario

- Debido a lo especificado por el maestro, también he agregado un panel para el usuario.
    - ***php artisan make:filament-panel UsuarioPanelProvider***
- Se agrega el mismo código que en el VeterinarioPanelProvider dándole acceso a los recursos necesarios que en este caso sería DatosUsuario, Mascotas y Citas.

Creación de un login global

- Es necesario crear un login global para que al acceder al login dependiendo del id_rol que contenga el usuario accediendo este sea redirigido al panel adecuado.
    - Se agrega a resources/views/auth/ el archivo login.blade.php con el HTML para mostrar el login.
    - Posteriormente hay que agregar en routes/web.php la redirección usando Route:: para que mande al login global.
    - Además es necesario agregar el método post al login para recuperar el id_rol del usuario accediendo y dependiendo del valor de este redirigir al panel que corresponde.
    - Finalmente debido a la forma en que funciona filament, al hacer logout de algún panel este nos dirige al login base que no sirve adecuadamente para el proyecto por lo que tras casi 2 horas de buscar en todos lados e intentar varias cosas, lo único necesario para corregir esto es borrar el método login() del AdminPanelProvider.

Restricciones a paneles usuario y veterinario

- Para desarrollar un sistema completo y seguro es necesario crear limitaciones en los recursos que serán compartidos con los paneles de usuario y veterinario para que estos solo puedan acceder y manipular los datos que son relevantes.
    - Para el panel usuario se requiere de acceso a los recursos DatosUsuario, Mascotas y Citas; esto bajo las siguientes restricciones.
        - DatosUsuario
            - No se muestra su ID, sólo puede ver y modificar los datos que pertenecen al usuario logeado en el panel.
        - Mascotas
            - No se muestra su ID, sólo puede ver y modificar las mascotas que pertenecen al usuario logeado en el panel.
        - Citas
            - No se muestra su ID, sólo puede ver las citas a las que está relacionado el usuario logeado en el panel.
    - Para el panel veterinario se requiere de acceso a los recursos DatosUsuario, CItas y Consultas.
        - DatosUsuario
            - No se muestra su ID, sólo puede ver y modificar los datos que pertenecen al usuario logeado en el panel.
        - Citas
            - No se muestra su ID, sólo puede ver y modificar las citas a las que está relacionado el usuario logeado en el panel.
        - Consultas
            - No se muestra su ID, sólo puede ver y modificar las consultas a las que está relacionado el usuario logeado en el panel.
    - Para lograr implementar estas restricciones es necesario realizar las siguientes modificaciones a los recursos.
        - Agregar variables que recuperen la información del panel actual y del usuario logeado.
            - ***$panel = Filament::getCurrentPanel()?->getId();***
            - ***$idUsuario = Filament::auth()->user()?->id_usuario;***
        - Se agrega una validación dentro de la función que genera el form donde si el panel actual es igual a admin retorna un form completo con todos los datos.
            - En caso que el panel actual no sea igual a admin entonces retorna el form con el valor del id con el componente Hidden en lugar del Select o TextInput, donde se declara el valor default como el id almacenado en la variable idUsuario y se deshidrata para que este pueda ser enviado y almacenado en la bd.
        - En la función table que genera los datos mostrados se hace exactamente el mismo proceso con la diferencia de que en lugar de usar el componente Hidden simplemente se elimina el TextColumn que contiene el id; y se agrega después del ->columns() una función ->query() que recupera los datos del usuario logeado mediante un where con la variable del idUsuario.

- ***->query(DatosUsuario::query()->where('id_usuario', $idUsuario))***

- Para finalmente terminar con la lógica, se restringe al usuario las acciones de crear un registro en CitasResource modificando el archivo ListCitas del recurso CitasResource usando una comparación dentro de la función getHeaderActions().
  - ***$panel = \Filament\Facades\Filament::getCurrentPanel()?->getId();***
    if ($panel === 'usuario') {
    return [];
    }
    return [
    Actions\CreateAction::make(),
    ];
- Se hace lo mismo con el ListDatosUsuarios del UsuariosResource pero usando una comparación en diferencia a 'admin' para afectar al panel veterinario y usuario.
- Y por último para restringir la modificación en el recurso Citas solo al panel usuario agregue una comparación en la función table en que si el panel es de un usuario se asigna a la variable $actions un array vacío y en caso distinto se declara la variable con la dirección de acción base que proporciona larafilament.
  - ***if($panel === 'usuario'){***
    $actions = [];
    }else{
    $actions = [\Filament\Tables\Actions\EditAction::make(),];
    };

- Con estos cambios ya se completo la lógica necesaria para poder manejar bien logins, logouts, accesos a recursos, etc de todos los paneles para los tipos de usuarios que están involucrados en el sistema.

Application Programming Interface

De acuerdo a los requerimientos del proyecto es necesario desarrollar una API que permita la comunicación entre la plataforma web y la aplicación móvil que desarrollare al final, la API será empleada para la recuperación de datos de la base de datos para ser mostrados en la app.

La API formará parte del proyecto larafilament con el fin de aprovechar la lógica ya creada en el desarrollo de la plataforma web.

Creación de API en proyecto

- Dentro de la carpeta routes del proyecto es necesario crear la interfaz de comunicación, en este archivo se definirán todas las rutas y el método a ejecutar en el controlador creado para cada modelo del proyecto
  - ***php artisan install:api***

- Para el correcto funcionamiento del login es necesario instalar e implementar sanctum para que este maneje la autenticación, esto mediante la generación de tokens personales que validan y vinculan los usuarios para las solicitudes.
    - ***composer require laravel/sanctum***
      php artisan vendor:publish --tag=sanctum-config

Desarrollo de lógica de acceso por peticiones y redireccionamiento

- Dentro de Http\Controllers se deben crear todos los controladores necesarios para las solicitudes que se estarán realizando desde la app móvil, en estos se define la lógica mediante funciones que retornan distintas ejecuciones según la ruta accedida, como lo podría ser el mostrar datos, eliminarlos, actualizarlos, crear nuevos; los controladores necesarios son los siguientes.
    - ***php artisan make:controller CatalogoController --resource***
    - CatalogoController
        - Los usuarios al ingresar al apartado de tienda realizan una solicitud GET para mostrar los distintos productos dentro del catálogo.
            - ***public function index()***
              {
              return Catalogo::all();
              }
        - En api.php es necesario agregar la ruta y redirección del controller con el método a emplear, en este caso GET.
            - ***Route::middleware('auth:sanctum')->get('/catalogo',[CatalogoController::class, 'index']);***
    - En el resto de Controllers de la misma forma se agrega la lógica necesaria para las funciones index (GET), store (POST), update (PUT) y destroy (DELETE) según las limitaciones que se ocupe por controlador y posteriormente hay que agregar la redirección en el archivo api.php.
    - CitasController
        - Los veterinarios al acceder al apartado de citas realizan una solicitud GET de los datos en que su id está relacionado, pueden realizar un POST para crear una nueva cita, PUT para actualizar datos y DELETE para eliminar una cita.
        - Los usuarios al acceder a citas realizan una solicitud GET para mostrar las citas donde su id está relacionado.
    - ConsultasController
        - Los veterinarios al acceder al apartado de consultas realizan una solicitud GET de los datos en que su id está relacionado, pueden realizar un POST para crear una nueva consulta, PUT para actualizar datos y DELETE para eliminar una consulta.
    - DatosUsuarioController
        - Los veterinarios y los usuarios realizan una solicitud GET al acceder al apartado de datos donde se muestran lo recuperado donde su id esté relacionado, también pueden realizar una consulta POST para subir sus datos y una consulta PUT para actualizarlos.
    - MascotasController

        - Los usuarios al acceder al apartado mascotas realizan una solicitud GET para mostrar los datos donde su id está relacionado, pueden realizar un POST para crear una nuevo registro de mascota y PUT para actualizar datos de la mascota.
    - NotificacionesController
        - Los usuarios y veterinarios realizan una solicitud GET donde se recuperan los registros donde su id esté relacionado.
    - UserController
        - Los usuarios y veterinarios realizan una solicitud POST para hacer login y otra para hacer logout usando sanctum.
- Para corroborar el adecuado funcionamiento de cada ruta de la API se testeo de postman todas las rutas, para poder testearlo es necesario realizar una solicitud POST a /login con credenciales válidas para recuperar el token requerido para las solicitudes y poder identificar al usuario que las realiza, ya con el token generado desde un usuario válido se checo cada ruta con su respectivas solicitudes.

Aplicación Móvil

- Para la aplicación móvil desarrolle en total 17 pantallas con xml y 16 archivos de lógica con java, además también hice 4 diseños xml para bordes en distintos apartados y 6 diseños de fondo para distintas pantallas diseñado en Canva, por último utilice múltiples iconos de flaticon para los distintos botones de las pantallas.
- Existen 2 tipos de pantallas a mostrar según el tipo de usuario logueado, correspondiendo a veterinario y usuario o dueño de la mascota, además según su tipo de usuario existen restricciones en cuanto a las acciones dentro de pantallas.

Home e Inicio de Sesión (HomeUsuarioActivity.java, HomeVeterinarioActivity.java, MainActivity.java, home_usuario_activity.xml, home_veterinario_activity.xml, main_activity.xml)

- El acceso a la app se gestiona desde MainActivity.java, donde se autentica al usuario vía API haciendo un login con token.
- Según su rol, es redirigido a la pantalla de inicio correspondiente (usuario o veterinario). Se utiliza SQLite mediante un código auxiliar DBHelper.java para guardar datos persistentes como lo son el token y rol del usuario.

Catálogo de Productos (CatalogoActivity.java, catalogo_activity.xml)

- Se utilizó un GridLayout para la disposición visual de productos, y una conexión a la API mediante Volley para obtener los datos.
- Se integró un campo de búsqueda que usa un TextWatcher para filtrar los productos en tiempo real.
- Los productos se cargan mediante una solicitud GET autenticada con token Bearer, y se visualizan con nombre, imagen y precio, junto con una animación fade_in incluida en un archivo XML separado.

Citas (Veterinario y Usuario) (CitaActivity.java, CitaEditarActivity.java, cita_veterinario_activity.xml, cita_usuario_activity.xml, cita_editar_activity.xml)

- Las pantallas de citas permiten tanto a veterinarios como usuarios visualizar y en el caso de los veterinarios, editar sus citas médicas. Se implementa una lógica condicional para mostrar campos según el tipo de usuario autenticado, usando validaciones desde la base de datos SQLite y el token del usuario.
- La edición se realiza llenando formularios con datos existentes, los cuales son enviados a través de peticiones PUT. Las vistas están diseñadas con XML y adaptadas según el tipo de usuario.

Consultas (ConsultaActivity.java, ConsultaEditarActivity.java, consulta_activity.xml, consulta_editar_activity.xml)

- Las consultas son accesibles únicamente por los veterinarios desde esta actividad.
- Se emplea lógica similar para la obtención de datos mediante GET y la edición de datos ya existentes con PUT.
- Se hace uso de estructuras LinearLayout y ScrollView para una correcta visualización de los registros médicos.
- Los datos son filtrados y adaptados al veterinario autenticado mediante su id_usuario.

Mascotas (MascotaActivity.java, MascotaAgregarActivity.java, MascotaEditarActivity.java, mascota_activity.xml, mascota_agregar_activity.xml, mascota_editar_activity.xml)

- Estas pantallas permiten la visualización, edición y registro de mascotas asociadas al usuario logueado.
- La creación y edición de mascotas se realiza mediante formularios y solicitudes POST/PUT respectivamente.

Datos de Usuario y Veterinario (DatosActivity.java, DatosSubirActivity.java, datos_usuario_activity.xml, datos_veterinario_activity.xml)

- Para la gestión de datos personales, los usuarios y veterinarios tienen pantallas específicas, distintas únicamente por los botones de navegación disponibles.
- Se recuperan los datos mediante una petición GET a la API, y el usuario puede modificarlos con una solicitud PUT.
- Las validaciones se realizan tanto en el formulario como en el backend, asegurando que cada usuario sólo pueda modificar su propio registro.

Notificaciones (NotificacionActivity.java, notificacion_usuario_activity.xml, notificacion_veterinario_activity.xml)

- Permite a los usuarios y veterinarios visualizar notificaciones recuperadas mediante peticiones GET autenticadas.

- La visualización se adapta al tipo de usuario, y los datos son filtrados desde la base según su id_usuario.

Registro y Autenticación (RegistrarActivity.java, registrar_activity.xml)

- El proceso de registro se realiza mediante un formulario y validaciones de campos en el mismo.
- Al finalizar, se envían los datos por POST al endpoint correspondiente del API, con validación y creación del usuario en el backend Laravel.

Base de Datos Local (DBHelper.java)

- Se emplea una base SQLite para almacenar credenciales del usuario, su token y rol, esto permite mantener la sesión activa y validar permisos de acceso dentro de la app.
- El helper proporciona métodos para insertar, actualizar, y recuperar estos datos.

Recuperación de contraseña

- La recuperación de contraseñas se hará mediante el uso de SendGrid, esto para el envío de un token que al ser validado en el backend laravel permitirá al usuario restablecer su contraseña.
- Para esto es necesario primero crear una cuenta en SendGrid y configurar lo necesario, posteriormente en el proyecto laravel hay que agregar los datos necesarios al .env y crear el controlador para la petición de la API desde la app móvil.
    - ***php artisan make:controller RecuperarController***
- Posteriormente es necesario agregar la ruta en routes/api.php
    - ***Route::post('/recuperar', [RecuperarController::class, 'enviarCorreo']);***
- Después hay que crear y migrar la tabla para almacenar los tokens de contraseñas restablecidas.
    - ***php artisan make:migration create_password_reset_tokens_table --create=password_reset_tokens***
- Es necesario además crear el blade que contenga el texto en el correo a mandar esto en la ruta views/emails.
    - ***Restablecer contraseña***
- Hay que crear el archivo Recuperar.php dentro de app/Mail.
    - ***public function __construct($token, $email)***
      {
      $this->token = $token;
      $this->email = $email;
      }
      public function build()
      {
      return $this->from(config('mail.from.address'), config('mail.from.name'))
      ->subject('Recuperación de contraseña')

```
            ->view('emails.recuperar');
    }
```

- Con esto laravel maneja el envío de un correo de recuperación basado en token y email, para poder hacer el envío de la información hay que redirigir al usuario a la web de recuperación cuando ingrese por la ruta correspondiente mandada por el correo desde routes/web.php y tambien el post para el formulario.
- ***Route::get('/password/reset/{token}', function ($token) {***
  return view('auth.reset', ['token' => $token]);
  })->name('password.reset');
  Route::post('/password/reset', function (Request $request) {
- Finalmente es necesario agregar el formulario para que el usuario agregue sus datos, en este caso use un blade con el nombre reset que es redirigido desde web.php ubicado en views/auth.
- Desde la aplicación móvil en el apartado de login existe el botón “Olvidaste tú contraseña?” el cual manda la solicitud POST para enviar el correo de recuperación.