

Perrinder

Proyecto final de 2 DAM elaborado por Juan Antonio Marquez Ruiz en el instituto IES Zaidin-Vergeles para presentar el 18 de junio de 2024. Tutor del proyecto: García Expósito, Antonio

Índice

- [Definición del proyecto](#)
- [Tags](#)
- [Desarrollo](#)
- [Conclusiones](#)
- [Recursos](#)
- [Autor](#)

Definición

Es una aplicación móvil sobre un tinder de perros con estructura de cliente servidor. Tu puedes dar like o dislike a los perros que te gusten o no te gusten y si el dueño del perro también te da like se creará un match y podrás hablar con el dueño del perro a través de un chat. Puedes subir fotos de tu perro y editar su perfil. También puedes ver los perros que te han dado like y los que te han dado dislike.

Tags

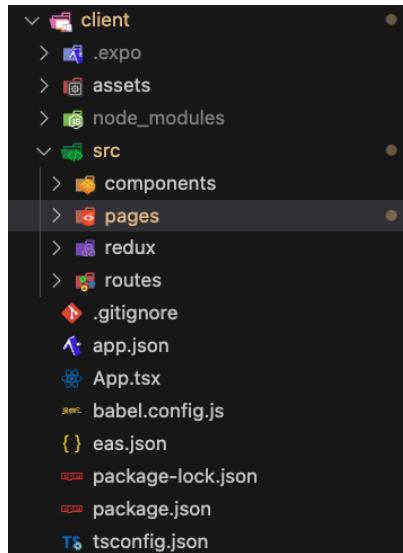
- React Native
- NestJS
- MySQL
- TypeScript
- Expo

Desarrollo

Para el desarrollo de la aplicación se ha utilizado [React Native con Expo](#) para crear una aplicación móvil y [Node con NestJS](#) para el servidor para crear una API REST. La base de datos está diseñada en [MySQL](#) y se ha utilizado [TypeScript](#) para el desarrollo, tipado y compilación de la aplicación. A continuación explico cada parte de la aplicación desde el login, inicio, chat, perfil y likes con capturas de pantalla y trozos de código de los componentes utilizados.

Cliente/App

Para explicar el desarrollo establecido en el lado de la aplicación móvil he usado esta estructura de carpetas:

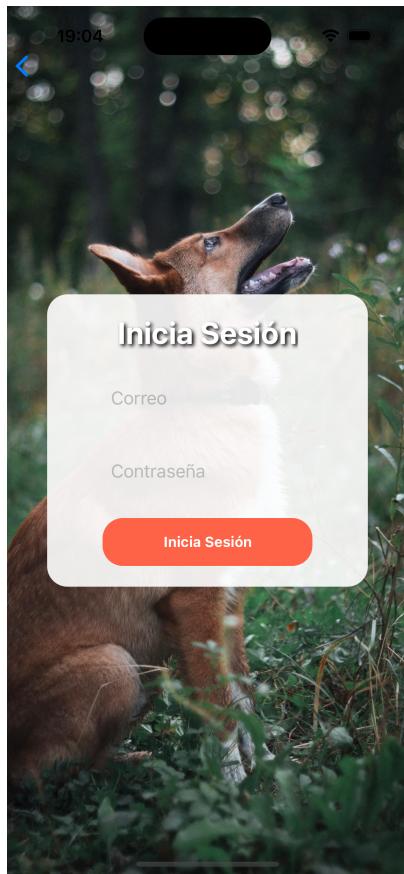


Contamos con las siguientes carpetas y archivos:

- **assets**: En esta carpeta se encuentran las imágenes que se utilizan en la aplicación.
- **components**: En esta carpeta se encuentran los componentes que se utilizan en la aplicación.
- **pages**: En esta carpeta se encuentran las páginas JSX que se utilizan en la aplicación.
- **routes**: En esta carpeta se encuentran las rutas de la aplicación.
- **redux**: En esta carpeta se encuentran los states de la aplicación.
- **App.tsx**: Archivo principal de la aplicación.

Implementación del login

Los usuarios para utilizar nuestra aplicación si o si tienen que estar logueados o registrarse. Para implementar el login de nuestra aplicación, mostramos un formulario simple con dos campos, uno para el email y otro para la contraseña.



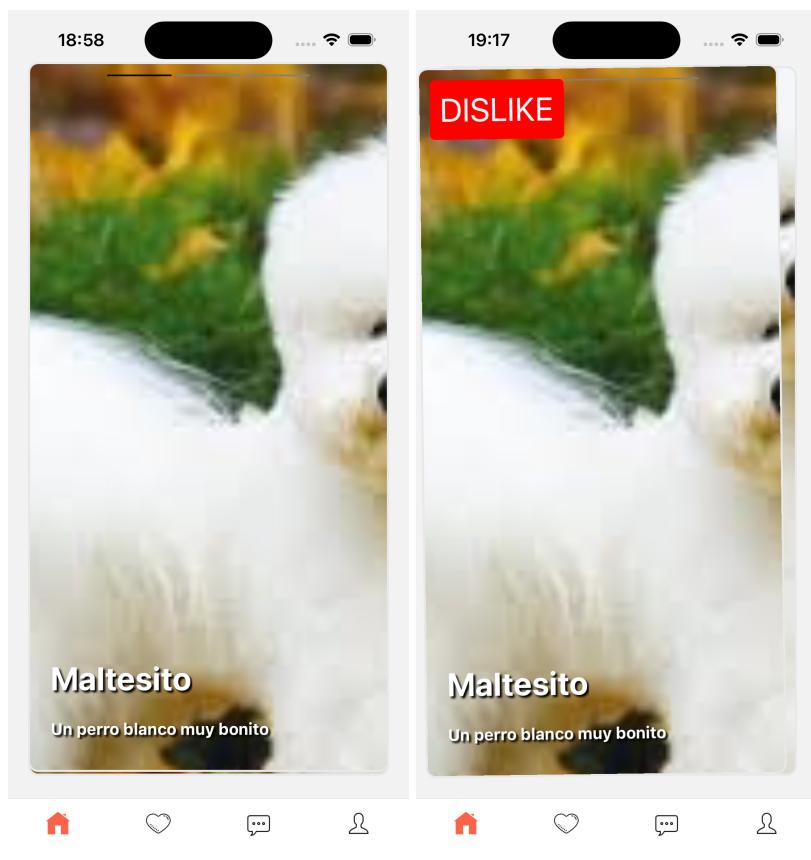
Una vez rellenados los datos realizamos una petición al servidor pasandole el email y la contraseña. Para ello utilizamos fetch y async/await. A continuación se muestra un ejemplo de cómo se realiza una petición POST con fetch utilizada en el archivo src/pages/login/components/login/login.tsx:

```
function handleLogin() {
    fetch(store.getState().url + '/auth/login', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify({
            email: mail,
            password: password
        })
    }).then((res) => {
        if (res.status === 201) {
            res.json().then((data) => {
                AsyncStorage.setItem('token', data.token);
                AsyncStorage.setItem('user', JSON.stringify(data.user));
                props.loadUserFromStorage();
            })
        } else {
            alert('Usuario o contraseña incorrectos')
        }
    })
}
```

En caso de que la respuesta del servidor API REST sea 201 osea que sea OK, se guarda el token y el usuario devuelto por el servidor para poder hacer más peticiones en el futuro. En caso de que la respuesta sea incorrecta, se muestra un mensaje de error al usuario dandole la oportunidad de volver a intentar iniciar sesión.

Implementación del inicio

El inicio de la aplicación muestra una serie de cards estilo tinder que muestran los perros que están en el sistema pero que aún no has interactuado con ellos (que no le has dado ni like ni dislike) y están programadas para que se puedan arrastrar y mover hacia la derecha si quieres dar un like o hacia la izquierda si quieres dar un dislike.



Para implementar las cards de nuestra aplicación he utilizado una librería llamada PanResponder que es nativa de React Native y la he implementado a mi gusto en el archivo pages/home/home.tsx:

```
import { PanResponder } from 'react-native';
function createPanResponder() {
    return PanResponder.create({
        onStartShouldSetPanResponder: (event, gesture) => {
            // Guarda el tiempo de inicio y la posición x de inicio cuando alguien empieza a mover una carta
            this.startTime = Date.now(); // Registra el tiempo de inicio
            this.startX = gesture.x0; // Registra la posición x de inicio
            return true;
        },
        onPanResponderMove: (event, gesture) => {
```

```
// Cambia la posición de la tarjeta cuando se mueve
position.setValue({ x: gesture.dx, y: gesture.dy });
setLikeTextPosition({ x: gesture.dx, y: gesture.dy });
},
onPanResponderRelease: (event, gesture) => {
    // Registra el tiempo de finalización y la posición x de
    cuando suelta la carta
    this.endTime = Date.now(); // Registra el tiempo de
finalización
    this.endX = gesture.x0; // Registra la posición x de
finalización

    const duration = this.endTime - this.startTime; // Calcula la
duración

    // Soltar la carta
    if (gesture.dx > 120) {
        // Soltar la carta a la derecha y dar like
        Animated.spring(position, {
            toValue: { x: SCREEN_WIDTH + 100, y: gesture.dy },
            useNativeDriver: false,
        }).start(async () => {
            // Cuando se ha terminado la animación, se da like al
perro y se pasa a la siguiente carta
            await likePet(pets[0]);
            nextCard();
        });
    } else if (gesture.dx < -120) {
        // Soltar la carta a la izquierda y dar dislike
        Animated.spring(position, {
            toValue: { x: -SCREEN_WIDTH - 100, y: gesture.dy },
            useNativeDriver: false,
        }).start(async () => {
            // Cuando se ha terminado la animación, se da dislike
al perro y se pasa a la siguiente carta
            await dislikePet(pets[0].user.id);
            nextCard();
        });
    } else {
        // Si no se ha movido la carta lo suficiente, se vuelve a
la posición inicial
        setLikeTextPosition({ x: 0, y: 0 });
        Animated.spring(position, {
            toValue: { x: 0, y: 0 },
            useNativeDriver: false,
        }).start();
    }
    // Clicks
    if (this.endX > 280 && duration < 200) {
        // Click derecho de la imagen
        // Muestra la siguiente imagen del perro
        setIndicePet(prevIndicePet => {
            if (prevIndicePet < imagePets[indice].images.length -
1) {

```

```

        return prevIndicePet + 1;
    } else {
        return prevIndicePet;
    }
});

} else if (this.endX < 120 && duration < 200) {
    // Click izquierdo de la imagen
    // Muestra la imagen anterior del perro
    setIndicePet(prevIndicePet => {
        if (prevIndicePet > 0) {
            return prevIndicePet - 1;
        } else {
            return prevIndicePet;
        }
    });
}

},
})
}

```

Es muy largo y complejo de ver de primeras pero es muy sencillo de entender. Con este trozo de código controlo el comportamiento de las cards para que el usuario pueda moverlas y dar like, dislike o cambiar la imagen haciendo clicks tanto para adelante como para atrás. Así puede interactuar con las cards de la aplicación de manera sencilla y rápida. Cuando ha realizado una acción de like o dislike la aplicación se encarga de mostrar la siguiente card y de guardar en la base de datos la acción realizada.

Para mostrarlo en la pantalla es tan fácil como mostrarlo en el return del componente. Este código que pongo a continuación es la vista que va a devolver nuestro componente home para mostrar cada una de las cards:

```

return (
<Animated.View
    // Cargamos el panResponder para que se pueda mover la carta
    {...(panResponder && panResponder.panHandlers)}
    key={pet.id}
    style={[position.getLayout(), styles.card, { transform: [{ rotate: rotateCard }] }]}

>
{
    // Texto de like oculto o visible
    likeTextPosition.x > 0 ? <View style={[styles.likeTextView]}>
        <Text style={styles.likeText}>LIKE</Text>
    </View> : null
}
{
    // Texto de dislike oculto o visible
    likeTextPosition.x < 0 ? <View style=
    {[styles.dislikeTextView]}>

        <Text style={styles.dislikeText}>DISLIKE</Text>
    </View> : null
}

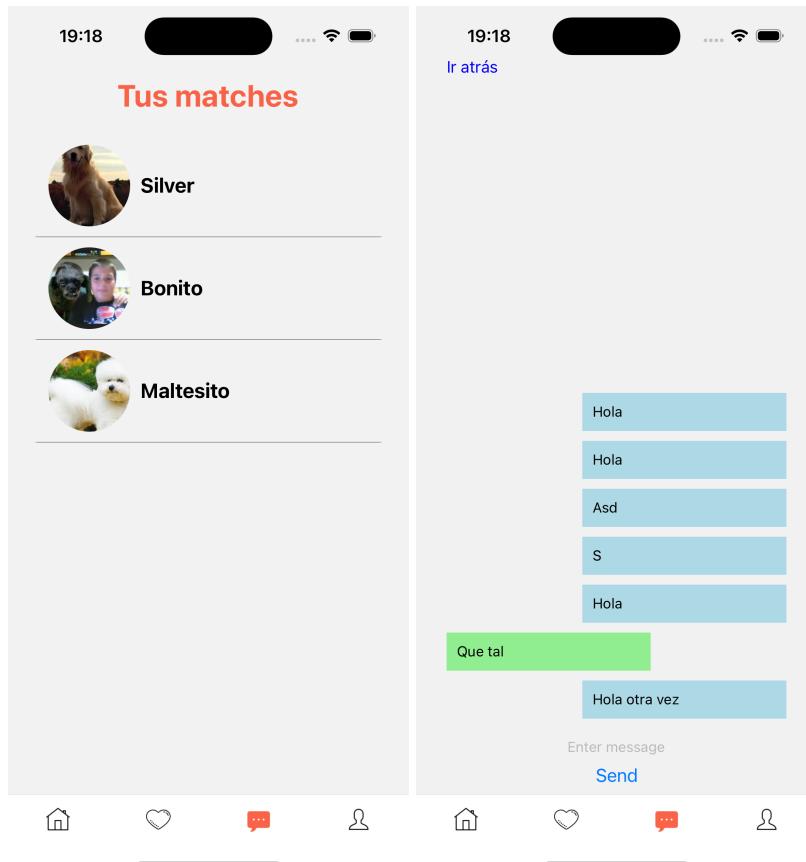
```

```
        }
        {
            // Barra superior que muestra cuantas imagenes tiene el perro
            imagePets[index].images.length > 1 ? <View style=
{styles.photoIndex}>
                {
                    imagePets[index].images.map((image, index) => {
                        return <Text key={index} style=
{[styles.photoIndexText, { color: indicePet == index ? 'black' : 'grey'}]}>_____</Text>
                    })
                }
            </View> : null
        }
        {
            // Mostramos aquí la imagen del perro
            image ?
                <Image style={styles.image} source={{ uri: image }} /> :
                <Image style={styles.image} source={{ uri:
imagePets[index].images[indicePet] }} />
            }
            // Nombre y descripción del perro
            <Text style={styles.name}>{pet.name}</Text>
            <Text style={styles.description}>{pet.description}</Text>
        </Animated.View>
    );
}
```

Ese sería el código equivalente a la vista de la card de la aplicación. En ella se muestra la imagen del perro, el nombre y la descripción del perro y se controla el comportamiento de la card con el panResponder que se ha creado anteriormente. En caso de que la card se mueva hacia la izquierda o derecha se mostrará un texto de LIKE o DISLIKE respectivamente. Además, si el usuario hace click en la parte derecha o izquierda de la imagen, se mostrará la siguiente o la anterior imagen del perro.

Implementación del chat

Cuando tu das like a un perro que ya te ha dado like a ti podéis mantener una conversación a través del chat. Para implementar el chat por la parte del cliente he tenido que obtener con una petición GET simple como la que se muestra en el login y mostrar por pantalla todas las conversaciones activas que tiene el usuario logueado. Una vez que se que conversaciones tiene el usuario las muestro por pantalla en forma de lista y si hace click en ellas se abrirá un chat con esa conversación y un mensaje arriba para poder ir atrás.



Una vez que te metes dentro de un chat puedes ver todos los mensajes que has enviado y recibido y puedes enviar mensajes nuevos. Para ello he utilizado WebSockets que es una tecnología que permite la comunicación bidireccional entre un cliente y un servidor. A continuación se muestra un ejemplo de cómo se envían y reciben mensajes con WebSockets usando la librería [socket.io-client](#)

```
useEffect(() => {
  // si no hay sala no se puede enviar mensajes
  if (room.id == null) return;
  socket.emit('joinRoom', room.id );

  // Cada vez que se recibe un mensaje se añade al array de mensajes
  // y se muestra en la pantalla
  socket.on('message', (msg) => {
    setMessages((prevMessages) => [...prevMessages, msg]);
  });

  // Cada vez que se envía un mensaje se limpia el input
  // y se envía el mensaje al servidor para que lo envíe a los demás
  // usuarios
  const sendMessage = () => {
    socket.emit('message', { roomId: room.id, message, senderId: user.id });
    setMessage('');
  };

  return () => {
    socket.off('message');
  };
});
```

```
    };
}, [room]);

/**
 * Función para obtener los mensajes de una sala
 */
const fetchMessages = async () => {
    const token = await AsyncStorage.getItem('token');
    const response = await
fetch(`$store.getState().url}/chat/room/${room.id}/messages`, {
    method: 'GET',
    headers: {
        'Content-Type': 'application/json',
        'Authorization': `Bearer ${token}`
    }
});
// Si todo va bien guardamos los mensajes en el estado
if (response.status === 200) {
    const data = await response.json();
    setMessages(data);
} else {
    // Si hay un error mostramos un mensaje de error
    alert('Ha ocurrido un error, por favor, vuelve a iniciar sesión');
}
```

Para mostrar los mensajes en la aplicación he utilizado un `FlatList` que es un componente de React Native que permite mostrar una lista de elementos de manera eficiente. A continuación se muestra un ejemplo de cómo se muestra la lista de mensajes en la aplicación:

```
// Lista de mensajes
<FlatList
    style={styles.flatList}
    data={messages}
    renderItem={({ item }) =>
        <View style={styles.message}>
            // Tenemos que diferenciar los mensajes nuestros
            // de los mensajes de los demás usuarios para mostrarlos de
            // manera diferente
            {
                item.sender.id === user.id ?
                    <Text style={styles.myMessage}>{item.content}</Text>:
                    <Text style={styles.theirMessage}>{item.content}</Text>
            }
        </View>
    }
    keyExtractor={(item, index) => index.toString()}
/>
// Campo de texto para enviar mensajes
<TextInput
    value={message}
    onChangeText={setMessage}
```

```

    placeholder="Enter message"
/>
// Botón para enviar mensajes
<Button title="Send" onPress={sendMessage} />

```

Implementación del perfil

Cada usuario tiene su perfil que está asociado a su perro. En tu perfil puedes tanto editar el nombre de tu mascota como su descripción que es el contenido que aparecerá en el inicio a otros usuarios. Para editar tu perfil y poder subir fotos de tu perro he implementado un componente situado en `src/pages/profile/components/createPet/createPet.tsx` que te permite subir fotos de tu perro y editar su nombre y descripción. Para ello he utilizado la librería [React Native Image Picker](#) que permite seleccionar imágenes de la galería o hacer fotos con la cámara del dispositivo.



A continuación se muestra un ejemplo de cómo se seleccionan imágenes con React Native Image Picker:

```

import * as ImagePicker from 'expo-image-picker';

async function selectImage() {
  // Seleccionar imagen de la galería
  let result = await ImagePicker.launchImageLibraryAsync({
    mediaTypes: ImagePicker.MediaTypeOptions.All,
    allowsEditing: true,
    aspect: [4, 3],
    quality: 1,
  }

```

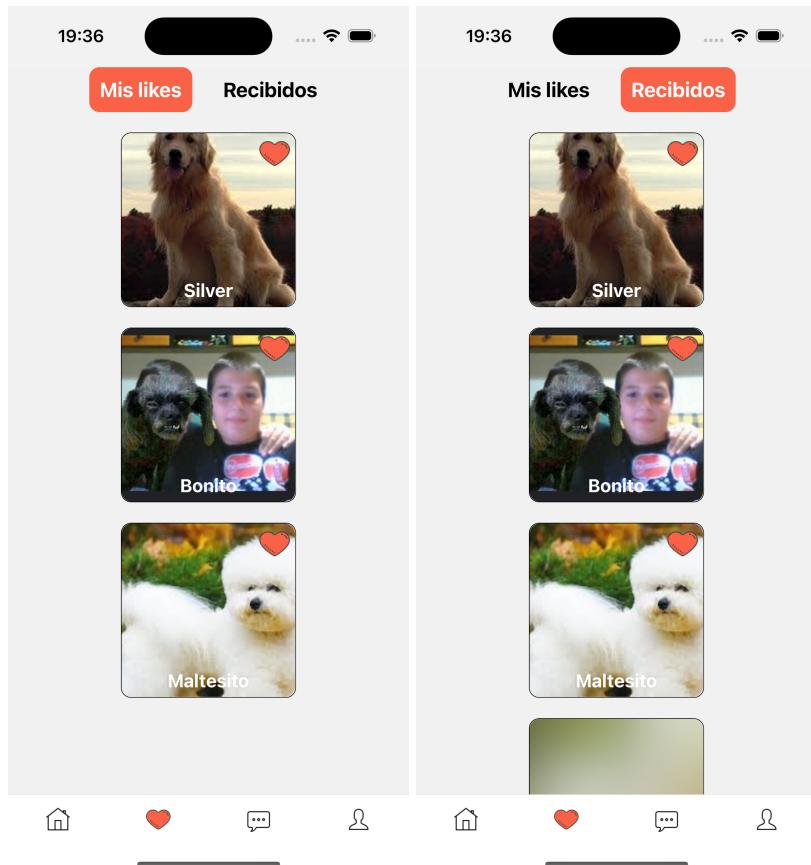
```
});  
  
    // Si no se ha cancelado la selección de la imagen  
    // se convierte la url de la imagen en un blob y se añade al array de  
    imágenes  
    if (!result.canceled) {  
        const imageBlob = await urlToBlob(result.assets[0].uri);  
        setImages([...images, result.assets[0].uri]);  
    }  
};
```

Con este método podemos obtener imágenes de la galería de un usuario y mostrarlas en pantalla para posteriormente subirlas a FireStore y guardar los nuevos datos en nuestra base de datos con una petición POST a mi servidor API REST una vez que se pulse el botón de Guardar Perro pasándole el nombre de la mascota, la descripción, el usuario que registra esa mascota y las fotos que ha subido a la API REST mencionada anteriormente.

```
async function createPet(token: string, userId: number, array: string[]) {  
    return await fetch(store.getState().url + '/pets/' + userId, {  
        method: 'POST',  
        headers: {  
            'Content-Type': 'application/json',  
            'Authorization': 'Bearer ' + token  
        },  
        body: JSON.stringify({  
            name: name,  
            description: description,  
            user: userId,  
            photos: array  
        })  
    })  
}
```

Implementación de likes recibidos

Otra pantalla de la aplicación es sobre los likes. Puedes ver los likes que tu has recibido y los que tu has dado a otra gente. Sabiendo también al momento si ellos te han dado like a ti. Para implementar un screen de la aplicación para mostrar los likes que hemos dado y que hemos recibido he implementado otro componente situado en src/pages/match/match.tsx que te permite ver los likes que has dado y los likes que has recibido.



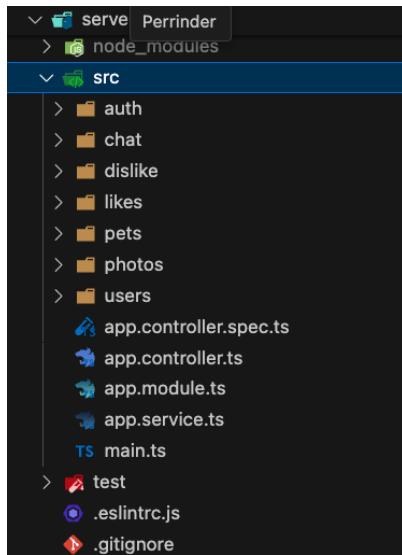
Como podéis ver en las imágenes, cuando hemos hecho un match sale un corazón arriba del perro que nosotros anteriormente hemos dado like, pero si vas a recibidos y hay algún perro que te ha dado like pero tu a él no te saldrá de manera difuminada y sin el corazón arriba

Conclusión del cliente

Estas han sido unas explicaciones sobre como he ido implementando todas las partes de la aplicación móvil. Como resumen rápido el login sirve para autenticar los usuarios con la aplicación, el inicio de la app nada más iniciar sesión muestra las cards de los perros que aun no has interactuado con ellos, el chat sirve para comunicarte con los dueños de los perros que te han dado like y tu a ellos, el perfil sirve para editar tu perfil y subir fotos de tu perro y los likes recibidos sirven para ver los likes que has dado y los que has recibido.

Servidor

El servidor se ha desarrollado con Node y NestJS. La estructura de carpetas es la siguiente:



La documentación generada de la API REST para su uso es la siguiente: [Documentación](#). De todas maneras voy a documentar el proceso utilizado para la creación de cada carpeta de nuestra API REST. Contamos con una carpeta para cada tabla de nuestra base de datos y dentro de ella contamos con entity, createDto, updateDto, controller, module y service. Controlando tanto los datos de la entidad y el gestión de base de datos como los END POINTS y la lógica de negocio. Para la seguridad de la aplicación backend he implementado [JWT](#). Es un sistema de autenticación que se basa en la generación de un token que envía el cliente y este lo envía en cada petición para comprobar que está autenticado y así evitar que cualquier persona pueda acceder a los datos de la aplicación. Para permitir que el cliente pueda acceder a los datos de la aplicación he implementado [CORS](#). Es un sistema que permite que un cliente pueda acceder a los datos de un servidor que no está en el mismo dominio y a parte se utiliza la tecnología de [WebSockets](#) para el chat de la aplicación. Para el almacenaje de las fotos he utilizado [Firebase Storage](#) que es un storage en línea que ofrece Google y que permite subir y descargar fotos de manera sencilla. A continuación se muestra un ejemplo de cómo subir y descargar fotos con Firebase Storage:

```
import { getStorage, ref, uploadBytes, getDownloadURL } from
"firebase/storage";

// Descargar fotos
const promises = data.map(async (pet: any) => {
  const photoUrls = await Promise.all(pet.photos.map(async (photo: any)
=> {
    const storage = getStorage();
    const storageRef = ref(storage, photo.path);
    return getDownloadURL(storageRef);
  }));
  return {
    id: pet.id,
    name: pet.name,
    images: photoUrls
  };
});

// Subir fotos
```

```

const uploadTasks = images.map(async (image) => {
  const imageBlob = await urlToBlob(image);
  return uploadToFirebase(imageBlob);
});

async function uploadToFirebase(blob) {
  const storage = getStorage();
  const storageRef = ref(storage, 'images/' + Date.now());
  try {
    const snapshot = await uploadBytes(storageRef, blob);
    return snapshot.metadata.fullPath;
  } catch (error) {
    console.error('Error uploading file:', error);
  }
}

```

Implementación de la autenticación

Para autenticar a cada usuario como he dicho he utilizado JWT y para ello he creado un servicio de autenticación en el archivo src/auth/auth.service.ts que se encarga de autenticar a los usuarios y de generar el token que se envía al cliente. A continuación se muestra un ejemplo de cómo se autentica a un usuario y se genera el token:

```

async login(createAuthDto: CreateAuthDto) {
  // Obtenemos el email y contraseña del usuario
  const { email, password } = createAuthDto;
  // buscamos al usuario en la base de datos
  let userFound = await this.authRepository.findOne({
    where: { email },
  });

  // Si nunca se ha logeado antes, se crea una autenticación
  if (!userFound) {
    const plainToHash = await hash(password, 10);
    const newUser = this.authRepository.create({
      email,
      password: plainToHash,
    });
    await this.authRepository.save(newUser);
    // Se busca el usuario creado para hacer el login
    userFound = await this.authRepository.findOne({
      where: { email },
    });
  }

  // Aquí se comprueba si la contraseña hasheada es igual a la
  // contraseña del usuario
  const checkPassword = await compare(password, userFound.password);
  if (!checkPassword) {
    // Si no es igual se lanza un error
    throw new HttpException('Invalid credentials',
    HttpStatus.UNAUTHORIZED);
  }
}

```

```

    }
    // Si todo va bien se genera el token
    // Utilizando el id del usuario y la librería JWT
    const payload = { id: userFound.id };
    const token = this.jwtService.sign(payload);
    const user = await this.userRepository.findOne({
        where: { email },
    });
    if (!user) {
        // Si no se encuentra el usuario se lanza un error
        throw new HttpException(
            'Internal Error',
            HttpStatus.INTERNAL_SERVER_ERROR,
        );
    }
    // Si todo va correcto se devuelve el usuario y el token al
    controlador
    const data = {
        user,
        token,
    };
    return data;
}

```

Este es el servicio encargado de autenticar a los usuarios el controlador que se encarga de llamar a este servicio es el siguiente:

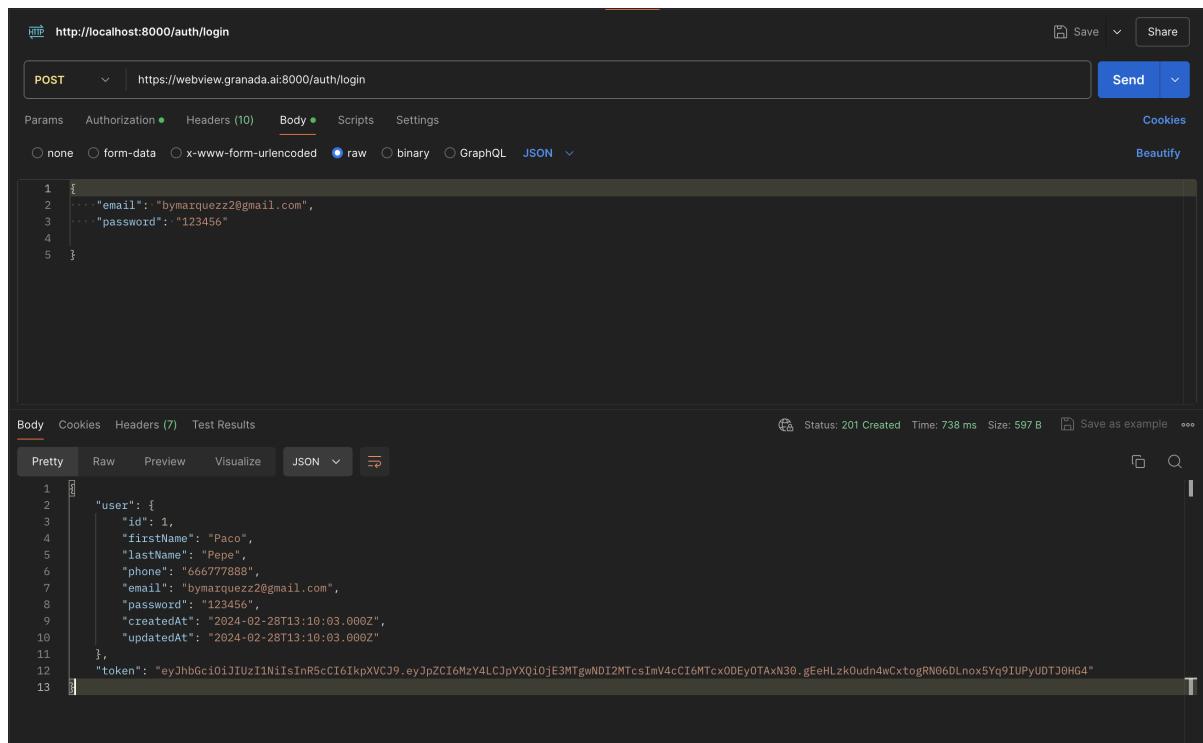
```

@Controller('auth')
@ApiTags('auth')
export class AuthController {
    constructor(private readonly authService: AuthService) {}

    @Post('login')
    @ApiBody({ type: CreateAuthDto })
    @ApiResponse({
        status: 201,
        description: 'The user has been successfully logged in.',
    })
    login(@Body() createAuthDto: CreateAuthDto) {
        return this.authService.login(createAuthDto);
    }
}

```

Cuando se envía una petición POST a la ruta /auth/login con el email y la contraseña del usuario, el controlador llama al servicio de autenticación y si todo va bien se devuelve el usuario y el token al cliente.



Luego el cliente se guarda el usuario y el token como hemos visto anteriormente y lo utiliza en cada petición que haga al servidor para comprobar que está autenticado.

Implementación de la lógica de negocio

Para la lógica de negocio de la aplicación he creado un servicio para cada entidad de la base de datos. Por ejemplo, para la entidad de usuarios he creado un servicio en el archivo src/users/users.service.ts que se encarga de gestionar los usuarios de la aplicación. A continuación se muestra un ejemplo de cómo se obtienen todos los usuarios de la aplicación:

```
import { HttpException, Injectable, HttpStatus } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { User } from './entities/user.entity';

@Injectable()
export class UsersService {
  constructor(
    @InjectRepository(User) private userRepository: Repository<User>,
  ) {}

  getAllUsers() {
    return this.userRepository.find({ relations: ['pets', 'pets.photos'] });
  }
}
```

Cada tabla de la base de datos tiene su propia entidad siendo a su vez un ORM para poder interactuar con la base de datos de manera simple.

```
import {  
    Entity,  
    Column,  
    PrimaryGeneratedColumn,  
    OneToMany,  
} from 'typeorm';  
import { Pet } from '../../../../../pets/entities/pet.entity';  
  
@Entity({ name: 'users' })  
export class User {  
    @PrimaryGeneratedColumn()  
    id: number;  
  
    @Column()  
    firstName: string;  
  
    @Column()  
    lastName: string;  
  
    @Column()  
    phone: string;  
  
    @Column({ unique: true })  
    email: string;  
  
    @Column()  
    password: string;  
  
    @OneToMany(() => Pet, (pet) => pet.user)  
    pets: Pet[];  
  
    @Column({ type: 'timestamp', default: () => 'CURRENT_TIMESTAMP' })  
    createdAt: Date;  
  
    @Column({  
        nullable: true,  
        type: 'timestamp',  
        default: () => 'CURRENT_TIMESTAMP',  
    })  
    updatedAt: Date;  
}
```

En esta entidad se definen los campos de la tabla de la base de datos y las relaciones con otras tablas. En este caso, la tabla de usuarios tiene una relación de uno a muchos con la tabla de mascotas. Luego también hay que implementar un controlador por cada tabla. En el caso de usuarios este es el controlador:

```
import {  
    Body,  
    Controller,  
    Delete,  
}
```

```
Get,
Post,
Param,
Patch,
UseGuards,
} from '@nestjs/common';
import { UsersService } from './users.service';
import { CreateUserDto } from './dto/create-user.dto';
import { UpdateUserDto } from './dto/update-user.dto';
import { JwtAuthGuard } from 'src/auth/jwt-auth.guard';
import {
  ApiBearerAuth,
  ApiBody,
  ApiParam,
  ApiResponse,
  ApiTags,
} from '@nestjs/swagger';

@ApiTags('users')
@Controller('users')
export class UsersController {
  constructor(private readonly userService: UsersService) { }

  @Post()
  @ApiBody({ type: CreateUserDto })
  @ApiResponse({
    status: 201,
    description: 'The user has been successfully created.',
  })
  create(@Body() createUserDto: CreateUserDto) {
    return this.userService.createUser(createUserDto);
  }

  @Get()
  @ApiBearerAuth()
  @UseGuards(JwtAuthGuard)
  @ApiResponse({ status: 200, description: 'Returns all users.' })
  findAll() {
    return this.userService.getAllUsers();
  }

  @Get(':id')
  @ApiBearerAuth()
  @UseGuards(JwtAuthGuard)
  @ApiParam({ name: 'id', description: 'User ID' })
  @ApiResponse({
    status: 200,
    description: 'Returns the user with the specified ID.',
  })
  findOne(@Param('id') id: string) {
    return this.userService.getUserById(+id);
  }

  @Patch(':id')
```

```

@ApiBearerAuth()
@UseGuards(JwtAuthGuard)
@ApiParam({ name: 'id', description: 'User ID' })
@ApiBody({ type: UpdateUserDto })
@ApiResponse({
  status: 200,
  description: 'The user has been successfully updated.',
})
update(@Param('id') id: string, @Body() updateUserDto: UpdateUserDto) {
  return this.userService.updateUser(+id, updateUserDto);
}

@Delete(':id')
@ApiBearerAuth()
@UseGuards(JwtAuthGuard)
@ApiParam({ name: 'id', description: 'User ID' })
@ApiResponse({
  status: 200,
  description: 'The user has been successfully deleted.',
})
remove(@Param('id') id: string) {
  return this.userService.deleteUser(+id);
}
}

```

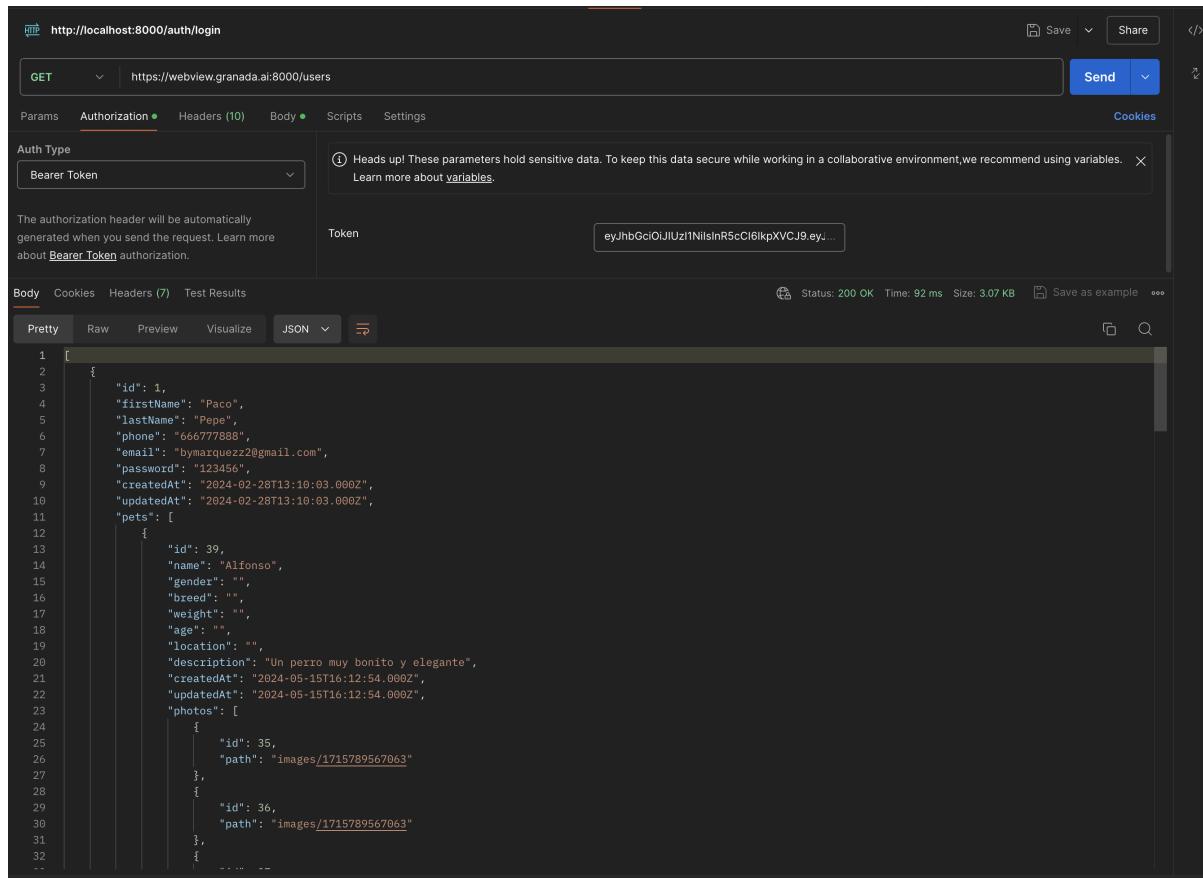
Para cada tabla tenemos una petición POST que es para crear nuevos usuarios, una petición GET simple que es para obtener todos los usuarios que tiene la aplicación, una petición GET con un parámetro que es para obtener un usuario en concreto, una petición PATCH que es para actualizar un usuario y una petición DELETE que es para borrar un usuario. Cada petición tiene su propio DTO que es un objeto que se utiliza para transferir datos entre el cliente y el servidor. A continuación se muestra un ejemplo de cómo se crea un DTO:

```

export class CreateUserDto {
  firstName: string;
  lastName: string;
  phone: string;
  password: string;
  email: string;
}

```

Con toda esta implementación ya podemos obtener usuarios, borrarlos, actualizarlos y crear nuevos usuarios en nuestra base de datos. Para cada tabla de la base de datos se ha seguido el mismo proceso. Ahora muestro un ejemplo en postman de como se obtienen todos los usuarios de la aplicación:

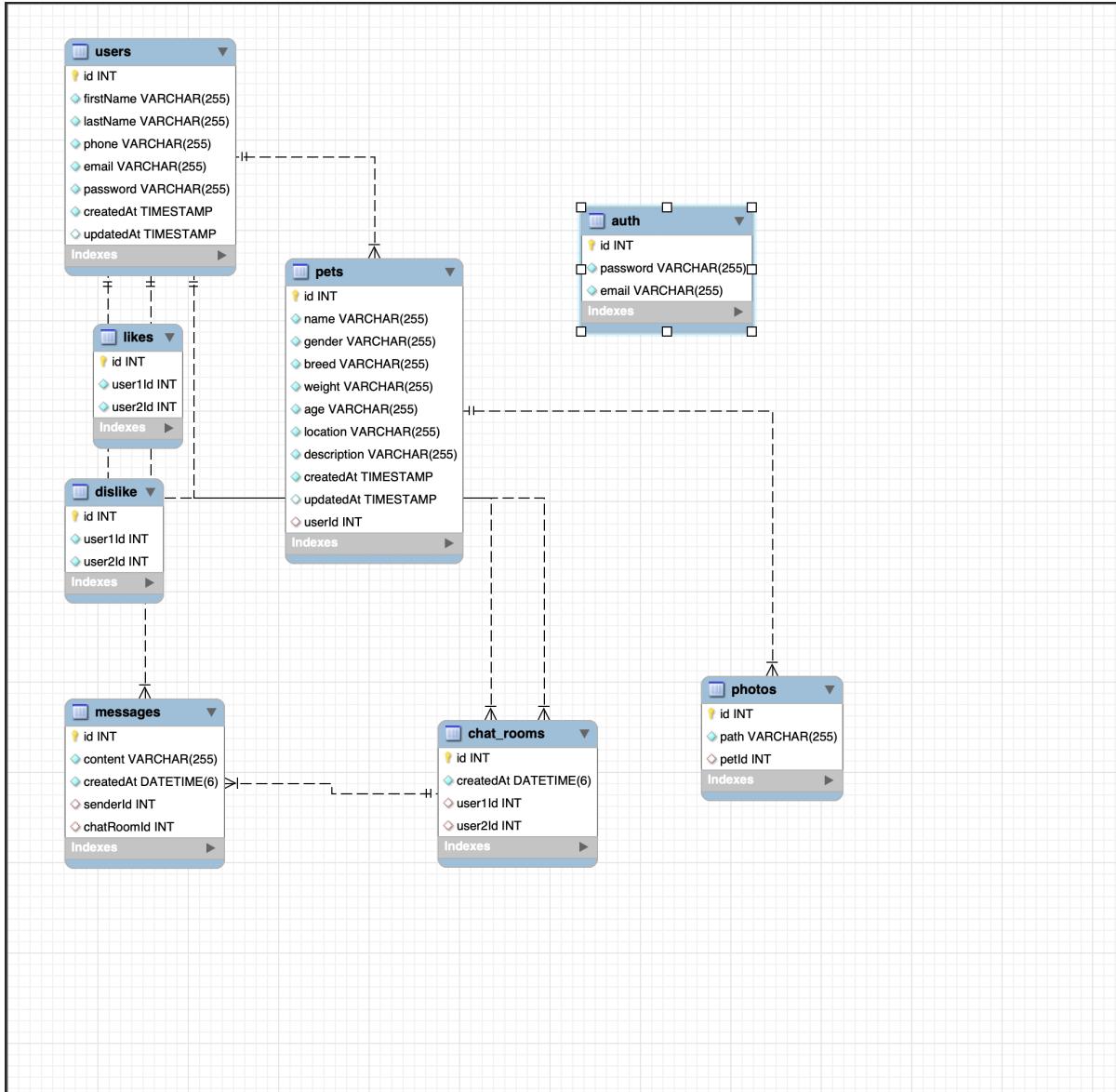


The screenshot shows a browser-based API testing tool. At the top, it displays the URL <http://localhost:8000/auth/login>. Below this, there's a search bar with the URL <https://webview.granada.ai:8000/users>. The main interface shows a "GET" request selected. Under "Params", there is an "Auth Type" dropdown set to "Bearer Token". A warning message states: "Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables." Below this, a "Token" field contains a long JWT token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ...". The "Headers" tab shows 10 headers, and the "Body" tab shows a JSON response. The response is a single user object:

```
1 [  
2 {  
3   "id": 1,  
4   "firstName": "Paco",  
5   "lastName": "Pepe",  
6   "phone": "+666777888",  
7   "email": "bymazquez2@gmail.com",  
8   "password": "123456",  
9   "createdAt": "2024-02-28T13:10:03.000Z",  
10  "updatedAt": "2024-02-28T13:10:03.000Z",  
11  "pets": [  
12    {  
13      "id": 39,  
14      "name": "Alfonso",  
15      "gender": "",  
16      "breed": "",  
17      "weight": "",  
18      "age": "",  
19      "location": "",  
20      "description": "Un perro muy bonito y elegante",  
21      "createdAt": "2024-05-15T16:12:54.000Z",  
22      "updatedAt": "2024-05-15T16:12:54.000Z",  
23      "photos": [  
24        {  
25          "id": 35,  
26          "path": "images/1715789567063"  
27        },  
28        {  
29          "id": 36,  
30          "path": "images/1715789567063"  
31        },  
32        ...  
33      ]  
34    }  
35  ]
```

Base de datos

A continuación se muestra el diagrama de la base de datos.



Es una base de datos hecha en MySQL siendo las entidades clave users y pets y teniendo como entidades más secundarias en la lógica de negocio de la aplicación, como like, message y photos. En ella se busca la integridad de los datos y la eficiencia en las consultas, para evitar problemas de rendimiento o de seguridad. La relación entre las tablas está bien cuidado para evitar problemas de integridad referencial y para que las consultas sean lo más eficientes posibles. Tenemos el script para la base de datos dentro del proyecto en la carpeta server/output.sql que se puede ejecutar en un servidor MySQL para crear la base de datos y las tablas necesarias.

Despliegue y puesta en producción

Para realizar el despliegue contamos de dos partes principales de la aplicación, el cliente (aplicación móvil hecha en React Native) que la compilado a apk ya que la otra opción es subirlo a las tiendas como Google Play y App Store y tiene un coste monetario por lo que lo descarté para un proyecto final, y luego tengo la segunda parte que es servidor (API REST hecha en NestJS y NodeJS) que para desplegarla he utilizado un servidor VPS personal que tengo presencialmente en mi casa y que me conecto a él a través de ssh y ftp. Para su despliegue he compilado el proyecto y desplegado en local con [pm2](#) luego he abierto los puertos en red que en este caso he elegido el puerto 8000 y asignado un subdominio de mi dominio personal, generando también los archivos ssl para tener cifrado https con [certbot](#). Todo ello gestionado desde el

hosting propio que tengo en mi casa con Ubuntu y Windows Server. Para compilar el proyecto de la aplicación móvil tuve que crearme una cuenta en [Expo.dev](#) y loguearme desde la terminal en EAS para después compilarla utilizando la serie de comandos:

```
eas login
npx expo prebuild --clean
eas build -p android --profile preview --local
```

```
~/Proycts/Perrinder/client git:(master)±35 (2m 7.55s)
eas build -p android --profile preview --local

[RUN_GRADLEW] > Task :expo-font:lintVitalAnalyzeRelease
[RUN_GRADLEW] > Task :app:packageRelease
[RUN_GRADLEW] > Task :app:createReleaseApkListingFileRedirect
[RUN_GRADLEW] > Task :expo:lintVitalAnalyzeRelease
[RUN_GRADLEW] > Task :expo-image-loader:lintVitalAnalyzeRelease
[RUN_GRADLEW] > Task :expo-file-system:lintVitalAnalyzeRelease
[RUN_GRADLEW] > Task :expo-keep-awake:lintVitalAnalyzeRelease
[RUN_GRADLEW] > Task :expo-linear-gradient:lintVitalAnalyzeRelease
[RUN_GRADLEW] > Task :expo-image:lintVitalAnalyzeRelease
[RUN_GRADLEW] > Task :expo-image-picker:lintVitalAnalyzeRelease
[RUN_GRADLEW] > Task :expo-system-ui:lintVitalAnalyzeRelease
[RUN_GRADLEW] > Task :expo-splash-screen:lintVitalAnalyzeRelease
[RUN_GRADLEW] > Task :react-native-async-storage_async-storage:lintVitalAnalyzeRelea
se
[RUN_GRADLEW] > Task :react-native-image-picker:lintVitalAnalyzeRelease
[RUN_GRADLEW] > Task :react-native-permissions:lintVitalAnalyzeRelease
[RUN_GRADLEW] > Task :react-native-gesture-handler:lintVitalAnalyzeRelease
[RUN_GRADLEW] > Task :react-native-webrtc:lintVitalAnalyzeRelease
[RUN_GRADLEW] > Task :expo-modules-core:lintVitalAnalyzeRelease
[RUN_GRADLEW] > Task :react-native-safe-area-context:lintVitalAnalyzeRelease
[RUN_GRADLEW] > Task :app:lintVitalReportRelease
[RUN_GRADLEW] > Task :app:lintVitalRelease
[RUN_GRADLEW] > Task :app:assembleRelease
[RUN_GRADLEW] BUILD SUCCESSFUL in 1m 42s
[RUN_GRADLEW] 742 actionable tasks: 742 executed
[UPLOAD_APPLICATION_ARCHIVE] Application archives: /var/folders/yw/94z25ntd64bcj4chy
l215nvc0000gn/T/eas-build-local-nodejs/8c8cc962-81e1-48bf-8066-95ab5ad07b51/build/cl
ient/android/app/build/outputs/apk/release/app-release.apk
[UPLOAD_APPLICATION_ARCHIVE] Uploading application archive...
[PREPARE_ARTIFACTS] Preparing artifacts
[PREPARE_ARTIFACTS] Writing artifacts to /Users/bymarquezz/Proycts/Perrinder/client
/build-1718043381498.apk

Build successful
You can find the build artifacts in /Users/bymarquezz/Proycts/Perrinder/client/buil
d-1718043381498.apk
npm notice
npm notice New minor version of npm available! 10.7.0 → 10.8.1
npm notice Changelog: https://github.com/npm/cli/releases/tag/v10.8.1
npm notice To update run: npm install -g npm@10.8.1
npm notice
```

Una vez ejecutada esa serie de comandos se generará un apk que se podrá instalar en cualquier dispositivo Android y que se podrá utilizar para probar la aplicación. Para el despliegue de la aplicación móvil en la tienda de Google Play o App Store se necesita una cuenta de desarrollador y un coste monetario. Para la compilación del servidor he utilizado el comando:

```
npm install
npm run build
```

```
~/Projects/Perrinder git:(master)±36 (0.031s)
cd server/


---


~/Projects/Perrinder/server git:(master)±36 (4.47s)
npm run build

> perrinderapi@1.0.0 build
> nest build
```

Generandome este una carpeta dist con toda la aplicación de NodeJs compilada y lista para ser desplegada en un servidor. Para desplegarla en un servidor VPS he utilizado el comando:

```
pm2 start dist/main.js
```

```
Last login: Mon Jun 10 15:48:46 2024 from 83.60.9.71
root@ubuntu:~# pm2 list
+----+-----+-----+-----+-----+-----+-----+
| id | name        | mode   | ✨ | status    | cpu    | memory   |
+----+-----+-----+-----+-----+-----+-----+
| 8  | 1.0.8.7778  | fork   | 36 | online   | 0%    | 69.2mb   |
| 5  | api_express_bot | fork   | 1  | online   | 0%    | 53.7mb   |
| 10 | api_perrinder | fork   | 0  | online   | 0%    | 81.9mb   |
| 11 | chatbot     | fork   | 0  | stopped  | 0%    | 0b       |
+----+-----+-----+-----+-----+-----+-----+
root@ubuntu:~#
```

Conclusiones

En conclusión, el proyecto consta de una aplicación móvil hecha en React Native y un servidor API REST hecho en NodeJS y NestJS. La aplicación móvil es un tinder de perros en el que puedes dar like o dislike a los perros que te gusten o no te gusten y si el dueño del perro también te da like se creará un match y podrás hablar con el dueño del perro a través de un chat. Puedes subir fotos de tu perro y editar su perfil. También puedes ver los perros que te han dado like y los que te han dado dislike. La aplicación móvil se ha desarrollado con React Native y Expo y el servidor API REST se ha desarrollado con NodeJS y NestJS. La base de datos se ha diseñado en MySQL y se ha utilizado TypeScript para el desarrollo, tipado y compilación de la aplicación. Para el despliegue de la aplicación móvil se ha utilizado EAS y para el despliegue del servidor API REST se ha utilizado pm2. En resumen, el proyecto ha sido un éxito y se ha conseguido desarrollar una aplicación móvil con una estructura de cliente servidor que permite a los usuarios interactuar con los perros de una manera sencilla y rápida. Con todo esto se ha conseguido un proyecto final de ciclo formativo de grado superior de desarrollo de aplicaciones multiplataforma.

Recursos

- React Native
 - NestJS
 - MySQL
 - TypeScript
 - Expo
 - JWT
 - CORS

Autores

- Nombre: Juan Antonio Marquez Ruiz
- Github: [GitHub](#)
- Correo: jmarri889s@ieszaidinvergeles.org
- Linkedin: [Linkedin](#)