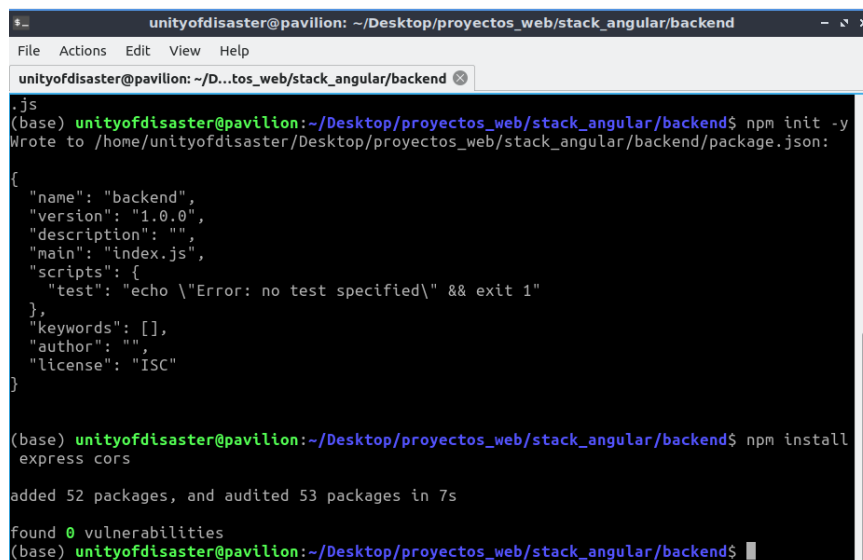


# Documentación NotesAPP Stack\_Angular

## preparacion de backend

Se crea un proyecto de node con las configuraciones básicas en el archivo package.json.



```
unityofdisaster@pavilion: ~/Desktop/proyectos_web/stack_angular/backend
(base) unityofdisaster@pavilion:~/Desktop/proyectos_web/stack_angular/backend$ npm init -y
Wrote to /home/unityofdisaster/Desktop/proyectos_web/stack_angular/backend/package.json:

{
  "name": "backend",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}

(base) unityofdisaster@pavilion:~/Desktop/proyectos_web/stack_angular/backend$ npm install express cors
added 52 packages, and audited 53 packages in 7s
found 0 vulnerabilities
(base) unityofdisaster@pavilion:~/Desktop/proyectos_web/stack_angular/backend$
```

Dentro del directorio principal del proyecto para el backend se agrega el archivo index.js en el que se agrega la configuración básica para desplegar un servidor web sencillo. Para este paso se utilizan los paquetes **express**, que funciona para crear servidores web de forma ágil en nodejs y **cors** para dar permisos sobre quien puede acceder a la API que se va a generar.

```
const express = require('express');
var cors = require('cors');

// creacion de servidor
const exprApp = express();

// uso de middlewares
// Habilitar todas las peticiones CORS
exprApp.use( cors() );

// se agregan rutas habilitadas para la API

exprApp.use('/servicio/api_notes_app/users', (req, res) => {res.send("<h1>Users</h1>");});
exprApp.use('/servicio/api_notes_app/notes', (req, res) => {res.send("<h1>Notes</h1>");});

// middleware utilizado para poder parsear formatos JSON
exprApp.use( express.json() );

// habilitar servidor para que escuche peticiones en el puerto especificado
exprApp.listen(8081, () => {
  console.log('Servidor corriendo');
})
```

Imagen 1. Código inicial del servidor web

Como se puede ver en la imagen 1, también se agregaron las rutas iniciales que se manejarán en la API, una para usuarios y otra para notas.

Para no tener toda la lógica dentro del mismo archivo se crean directorios para establecer el enrutamiento a cada endpoint. Dentro de los archivos generados en este directorio se crea la lógica del crud con los métodos que corresponden a cada operación.

```
const { Router } = require('express');

const router = Router();

// se agregan las rutas y metodos correspondientes a cada operacion crud
// que se realizara con las notas

// en todas las operaciones se hace referencia a la raiz ya que la ruta
// completa se encuentra en el index y se completa al referenciar este modulo

// ruta para obtener notas
router.get('/', () => {});

// ruta para agregar notas
router.post('/', () => {});

// ruta para modificar notas
router.put('/:id', () => {});

// ruta para eliminar una nota
router.delete('/:id', () => {});

module.exports = router;
```

Imagen 2. Estructura del archivo de rutas.

Estas rutas se deben asociar a un handler o controlador que servirá para realizar todas las operaciones necesarias con el contenido que se obtenga a través del request así como devolver una respuesta por medio del response, por lo tanto se agregan funciones para manejar cada operación del CRUD.

```
// ruta para obtener notas
router.get('/', getUsers);

// ruta para agregar notas
router.post('/', createUser);

// ruta para modificar notas
router.put('/:id', updateUser);

// ruta para eliminar una nota
router.delete('/:id', deleteUser);
```

Imagen 3. Asociación de funciones controladoras de cada operación CRUD.

ya que se cuenta con los handlers o controladores de las rutas, el siguiente paso consiste en agregar el paquete de **mongoose** para poder interactuar con la base de datos mongoDB. Una vez agregado se añade un directorio que alojará los modelos que se ligarán con las colecciones de mongo, en este caso los dos modelos a realizar corresponden a usuario y nota. Adicionalmente se agrega el paquete **dotenv** para agregar la cadena de conexión a la base de datos como una variable de entorno y así evitar que este expuesta la información de usuario y contraseña explícitamente en el código. La configuración de conexión a la base de datos mostrada en la imagen 4 se guarda en la ruta db/config.js.

```
const mongoose = require('mongoose');

// se llama al metodo config para cargar el contenido del archivo .env en las variables de entorno
require('dotenv').config();

const dbConnection = async() => {

  try {

    // se toma cadena de conexion de las variables de entorno
    await mongoose.connect(process.env.CONN_STR, {
      useNewUrlParser: true,
      useUnifiedTopology: true,
      useCreateIndex: true
    });
  } catch (error){
    throw new Error(
      'Error al intentar conectar a la base de datos');
  }

}

module.exports = {
  dbConnection
}
```

Imagen 4. Configuración para establecer conexión con mongoDB.

## modelo usuario

Para el modelo de usuario asociado se considera sólo un campo conformado por el nombre.

```
const UserSchema = Schema({
  nombre: {
    type: String,
    required: true,
    unique: true
  }
});
```

Imagen 5. Estructura del modelo de usuario.

## modelo nota

Para el modelo de la nota se consideran los campos título, información, nombre del creador de la nota y fecha.

```
const {Schema, model} = require('mongoose');

const NoteSchema = Schema({
  titulo: {
    type: String,
    required: true
  },
  informacion: {
    type: String,
    required: true
  },
  nombreCreador: {
    type: String,
    required: true
  },
  fecha: {
    type: Date,
    required: true
  },
});
```

Imagen 6. Estructura del modelo de la nota.

Una vez que se han creado los modelos, se procede a crear las operaciones CRUD en cada uno de los handlers correspondientes al usuario y nota.

## Obtener notas

Para obtener todos los registros del modelo Nota se utiliza la función find.

```
const getNotes = async(req, res) => {  
  
  // obtener todos los registros de la coleccion no  
  tas  
  const notas = await Nota.find({});  
  res.json({  
    notas  
  });  
}
```

Imagen 7. Estructura del handler getNotes.

## crear nota

Para agregar contenido a una nota primero se debe extraer la información recibida por el request. Posteriormente se debe crear una instancia del modelo Nota y agregar los parámetros al constructor. Finalmente se debe llamar al método save de la instancia creada y si todo sale bien se puede enviar la respuesta al cliente. Cabe destacar que las operaciones de escritura y las de lectura que dependan de un parámetro de búsqueda pueden generar una excepción, así que es preferible envolver el código en un bloque try catch para manejar los posibles errores.

```
const createNote = async(req, res) => {  
  const { titulo, informacion, nombreCreador, fecha } = req.body;  
  
  try{  
    // validar que usuario ya existe, pero despues  
    // asignar parametros que seran guardados en la base  
    const nota = new Nota(titulo, informacion, nombreCreador, fecha);  
  
    await nota.save();  
  
    res.json({  
      nota  
    });  
  } catch(error) {  
    res.status(500).json({  
      errorMsg: 'Error al intentar guardar registro'  
    });  
  }  
}
```

Imagen 8. Estructura del handler createNote.

## actualizar nota

Para este caso se siguen los mismos pasos que cuando se crea una nota, aunque con una validación adicional. Ya que esta operación depende de un id ligado a un registro de la base de datos, se debe verificar que este registro exista, por lo tanto primero se utiliza la función `findById` para validar la existencia del registro, en caso de haberlo encontrado se procede a guardar la nueva información en el registro, en caso contrario se retorna el estatus 400 y un mensaje de error que indica que no existe la nota.

```
const updateNote = async(req, res) => {
  const noteID = req.params.id;

  try {
    const { titulo, informacion,
    nombreCreador, fecha } = req.body;

    console.log(noteID);

    // se verifica que el id ingresado sea valido
    const nota = await Nota.findById(noteID);

    if(!nota) {
      return res.status(400).json({
        errorMsg: 'no existe nota'
      });
    }

    // se agrega el parametro new: true para que la f
    // uncion retorne el contenido
    // del nuevo registro
    const notaActualizada = await Nota.
    findByIdAndUpdate(noteID,
      { titulo, informacion, nombreCreador
      , fecha },
      { new: true}, );

    console.log(notaActualizada)

    res.json({
      notaActualizada
    });

  } catch(error) {
    res.status(500).json({
      errorMsg:
      'no ha sido posible actualizar la nota'
    });
  }
}
```

Imagen 9. Estructura del handler utilizado para actualizar notas.

## Eliminar nota

Para el caso de eliminar notas se hace la misma validación que en el caso anterior, es decir, primero se busca el registro y si existe se procede a borrarlo.

```
const deleteNote = async(req, res) => {
  const noteID = req.params.id;

  try{
    // se verifica que exista el usuario que se pretende elimina
    const nota = await Nota.findById( noteID );
    if(!nota) {
      return res.status(404).json({
        errorMsg: 'No existe nota'
      });
    }

    await Nota.findOneAndDelete( noteID);

    res.json({
      msg: `nota eliminada`
    })
  }catch(error) {
    res.status(500).json({
      errorMsg: 'No se pudo eliminar el registro'
    });
  }
}
```

Imagen 10. Estructura del handler utilizado para eliminar registros de la base de datos.

Nota: Con los handlers de operaciones CRUD con el modelo de usuario se una lógica similar.

## Probando endpoints con POSTMAN

Con lo anterior ya se cuenta con una versión lo suficientemente funcional como para poder hacer pruebas sobre la API. Para hacer estas pruebas se utiliza el programa postman.

### Probando creación de usuario

Con el backend en ejecución se accede por el momento a la ruta del localhost y se hace la primera prueba en el Endpoint de usuarios para agregar un registro por medio del método POST. Como se puede ver en la imagen 11 al ejecutar la petición se obtuvo como respuesta el estatus 200 y la estructura del registro añadido.

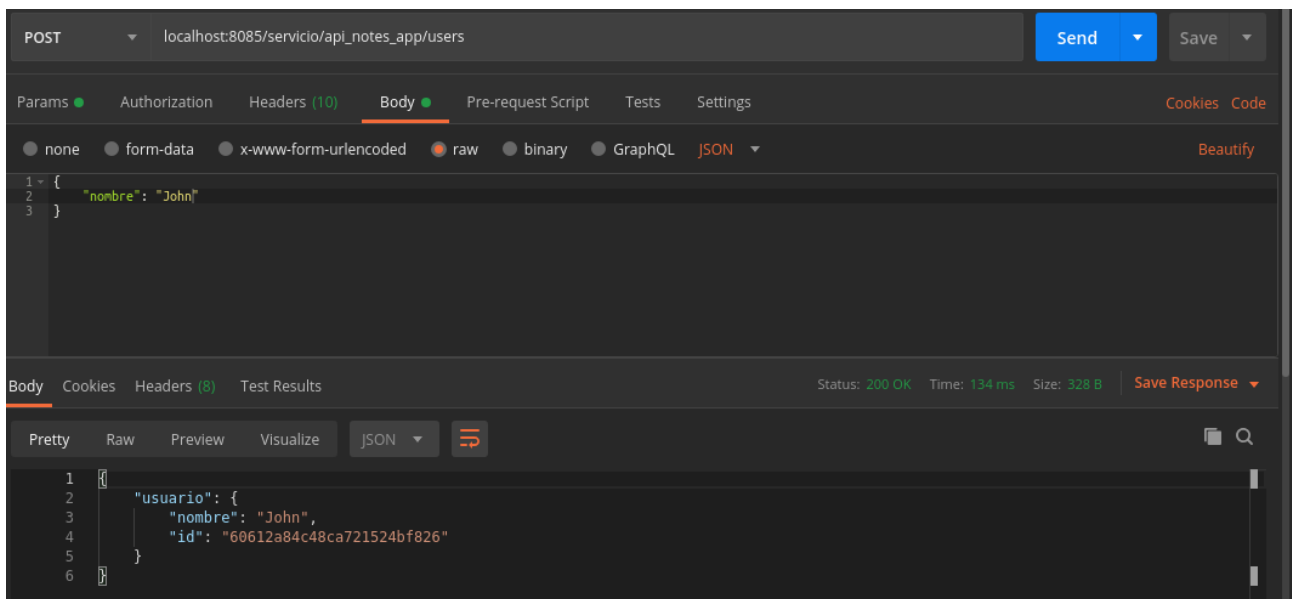


Imagen 11. Prueba de creación de usuario en Postman.



## Obtener lista de usuarios

Como segunda prueba se usa el método GET que en la API esta destinado para devolver todos los registros que se encuentren en la base de datos. En la imagen 12 se puede observar que después de hacer la petición se obtiene como respuesta un arreglo con todos los elementos de la base de datos.

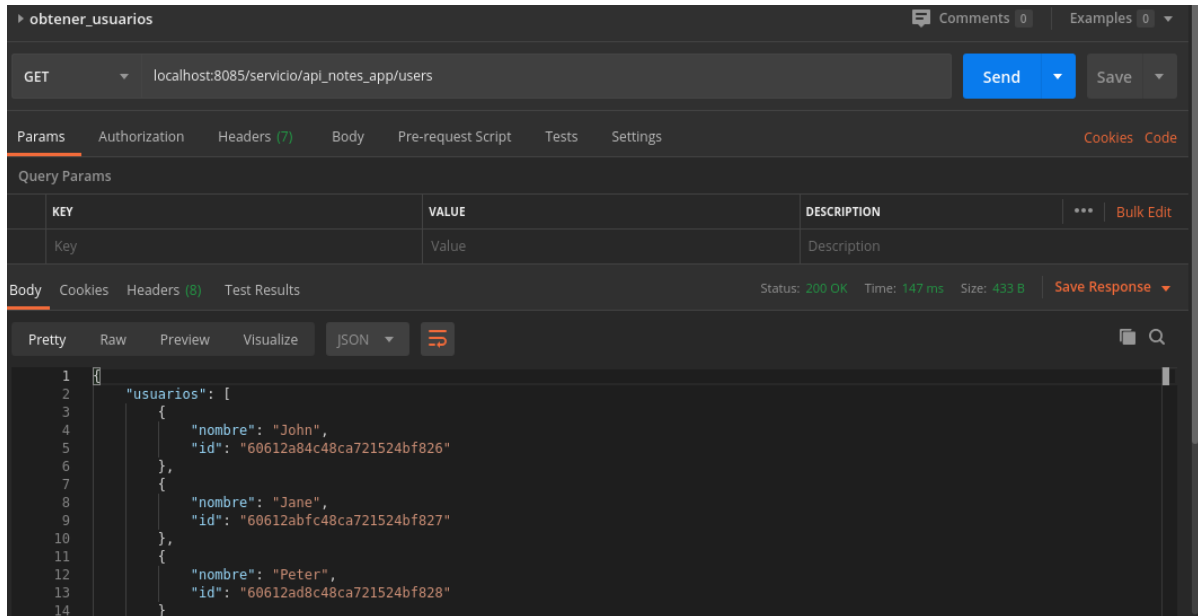


Imagen 12. Prueba de obtención de usuarios en Postman.

## actualizar usuario

Para probar la actualización de un registro se requiere enviar un id como parámetro así que para este caso se copia y pega de forma manual uno de los ya existentes y se agrega un nuevo nombre. En la imagen 13 se puede observar que después de hacer la petición se obtuvo como respuesta el estatus 200 y la estructura del nuevo registro que ha sido agregado.

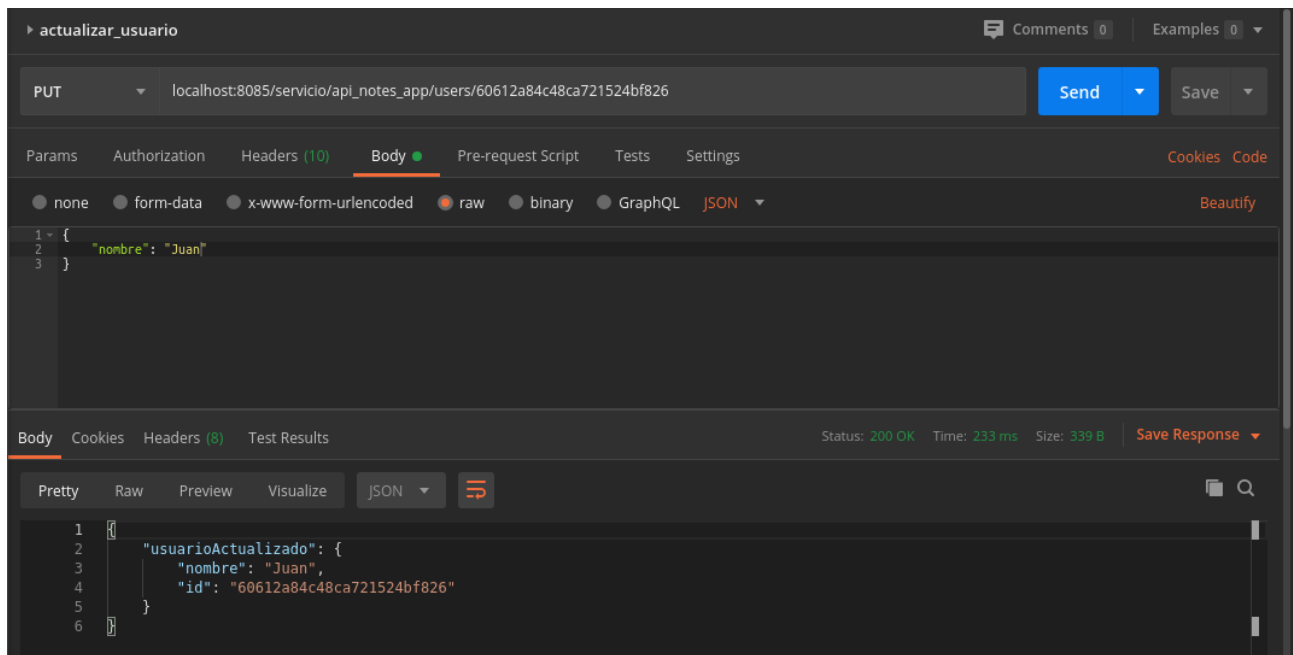


Imagen 13. Prueba de actualización de usuario en Postman.

## eliminar usuario

Similar al caso anterior, se requiere como parámetro un ID. Este se agrega de forma manual de los ya existentes y se envía la petición. Como respuesta se obtiene el estatus 200 y un mensaje de confirmación.

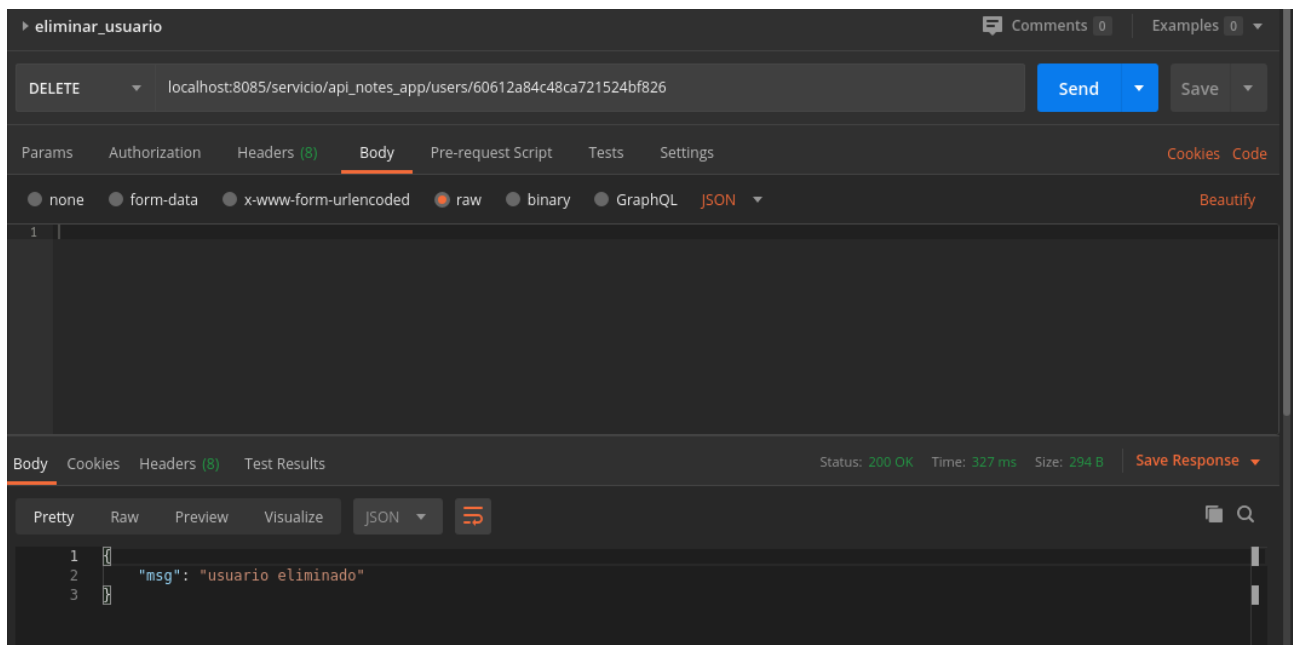


Imagen 14. Prueba de eliminación de usuario en postman.

## Preparacion del frontend

En la parte correspondiente al frontend se decide utilizar angular. Como primer paso se crea un proyecto sencillo con enrutamiento por defecto y para los estilos se agrega bootstrap como dependencia para manejar los estilos de forma más ágil.

Conforme a los requerimientos o las vistas solicitadas se consideran los siguientes componentes:

- Navbar que será compartido por todas las vistas.
- Componente para manejar la lista de notas.
- Componente para manejar el formulario y lista de usuarios existentes.
- Componente para manejar el formulario de creación y edición de notas.

Adicionalmente se agregan dos servicios que serán utilizados para poder hacer peticiones http e interactuar con la API desarrollada previamente.

### Prototipo inicial

Antes de iniciar con la programación de los componentes se genera un diseño estático inicial para cada una de las vistas. En la figura 15 se puede ver el diseño de la lista de notas que consiste en grupo de cards con campos para alojar cada uno de atributos del modelo de la nota. Adicionalmente se agregan dos botones para editar y eliminar alguna nota en específico.

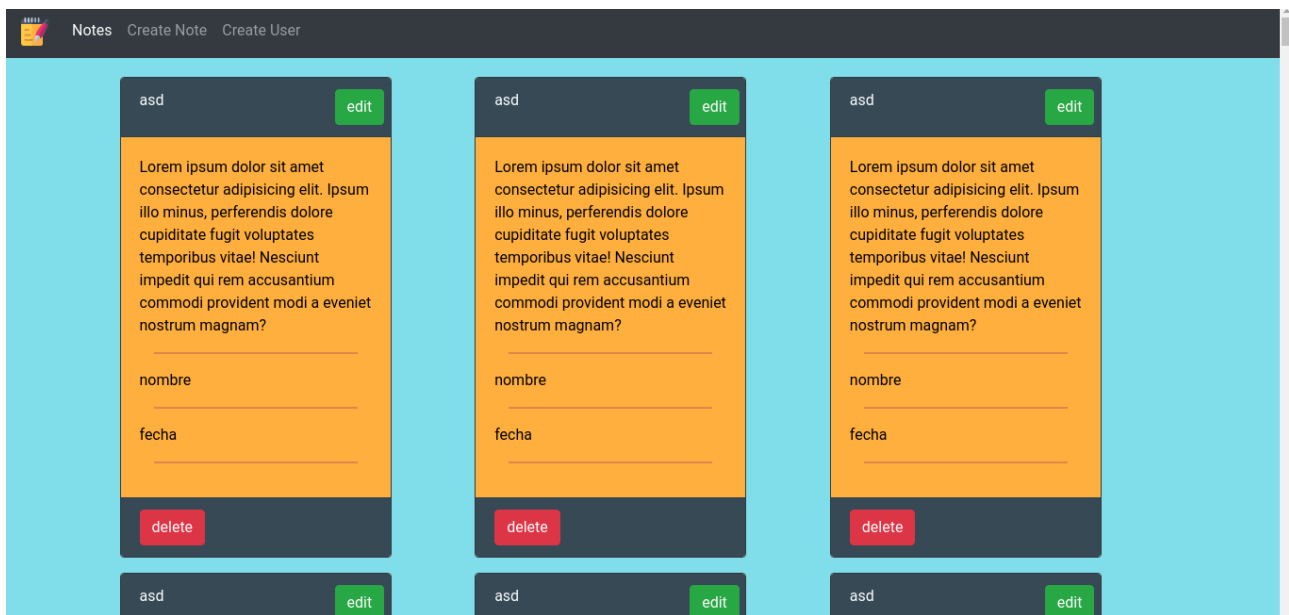
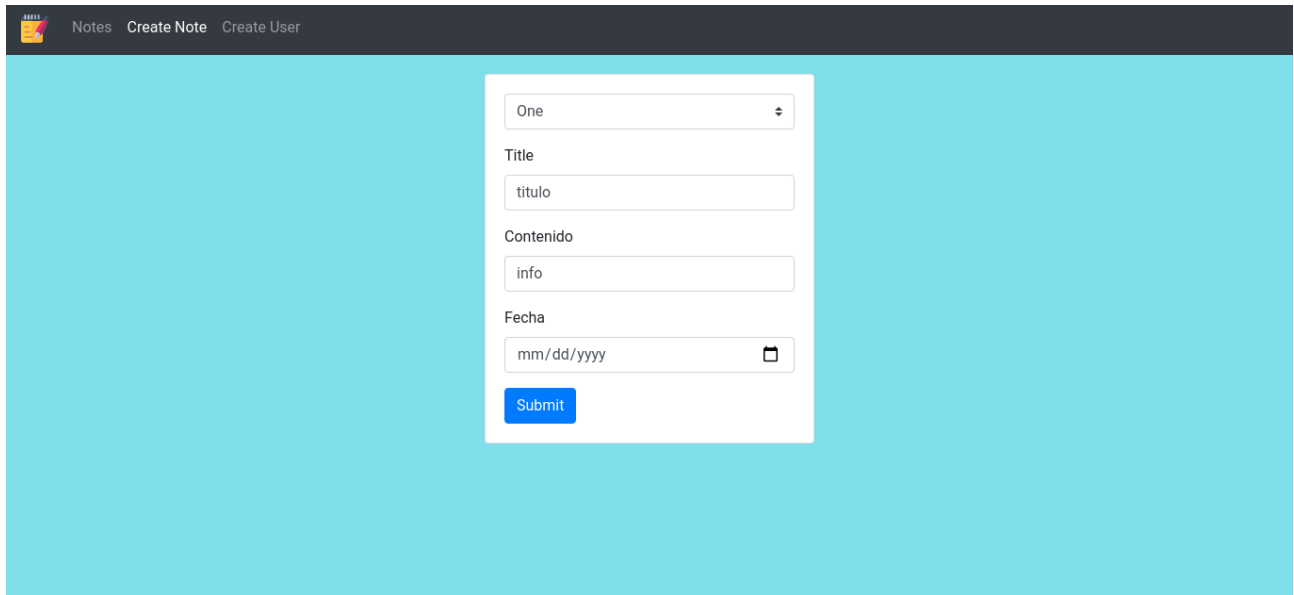


Imagen 15. Diseño de lista de notas.

La segunda vista se compone por el formulario donde se podrán agregar información a una nota y mandar la petición para que esta pueda ser guardada en la base de datos. Este formulario también será utilizado para actualizar información de un registro cuando sea requerido.



Notes Create Note Create User

One

Title

titulo

Contenido

info

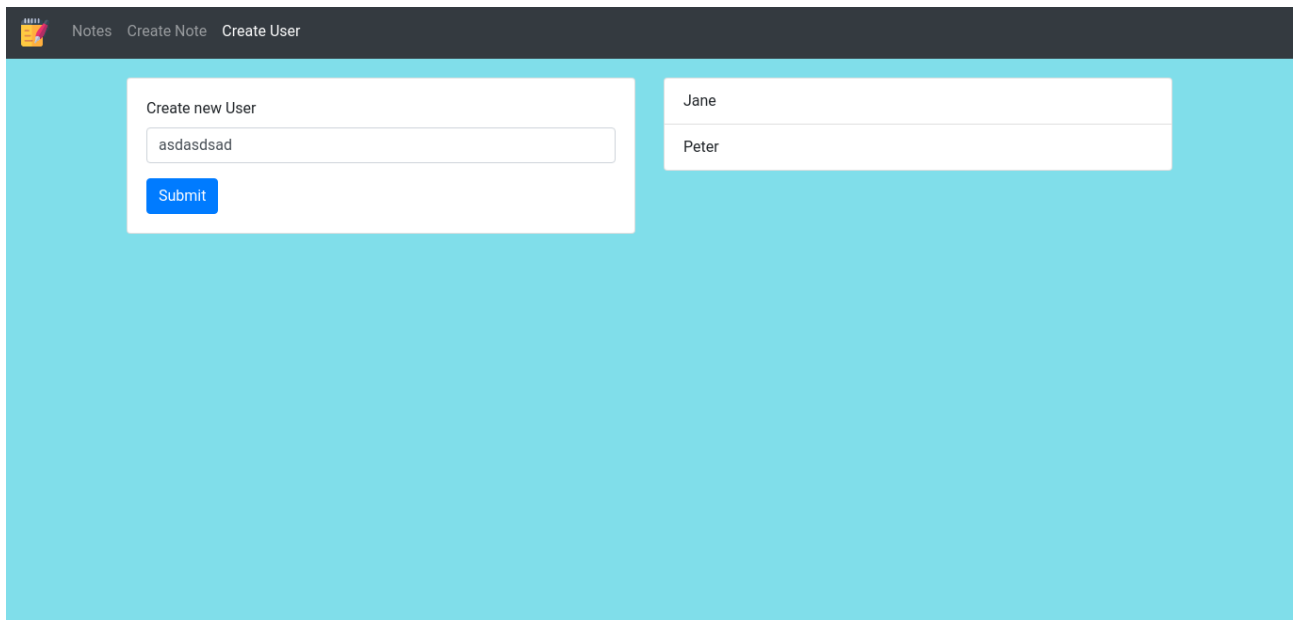
Fecha

mm/dd/yyyy

Submit

Imagen 16. Formulario que será utilizado para agregar información y guardar nota.

La última vista generada, corresponde a la que aloja tanto el formulario para dar de alta o crear un usuario como de la lista de usuarios que existen hasta el momento.



Notes Create Note Create User

Create new User

asdasdsad

Submit

Jane

Peter

Imagen 17. Diseño de formulario para usuario y lista de usuarios existentes.

## Preparación de servicios

Dado que todos los componentes requieren algún elemento de la base de datos, se comienza con el diseño de funcionalidades para las operaciones CRUD sobre la API por medio de los servicios.

El servicio de usuario solo requiere dos operaciones ya que en las vistas solo se requiere la operación de obtener usuarios para llenar la lista de usuarios, y la operación de crear usuario, ambas muestra la imagen 17. Para realizar estas operaciones se utiliza el objeto HttpClient. En el caso del método getUsers se aplica la operación map para hacer una manipulación previa sobre la respuesta y retornar solo la parte esencial de la respuesta que son los usuarios.

En la parte de crear usuarios se envía la información recibida como parámetro como cuerpo de la petición y se aplica un mapeo para verificar el estatus de la respuesta y se envía una bandera.

```
getUsers() {  
  return this.http.get(`${apiURL}/users`).pipe(  
    map((response: any) => {  
      console.log(response);  
  
      // se extrae arreglo de usuarios de la respuesta  
      return response.usuarios;  
    } ),  
  )  
}  
  
createUser(user: UserInterface) {  
  return this.http.post(`${apiURL}/users`, user  
    , { observe: 'response' }).pipe(  
    map((response) => {  
      if(response.status === 200) {  
        return true;  
      } else {  
        return false;  
      }  
    } )  
  )  
}
```

Imagen 18. Código del servicio utilizado para conectar con el endpoint de usuarios.

En el servicio de notas se sigue una estructura similar, teniendo como extra los métodos para eliminar y actualizar una nota. En el caso de la eliminación de nota se puede notar que solo recibe como argumento el id correspondiente al registro y este se agrega como parámetro en la URL. Para la actualización se recibe como argumento un objeto tipado por una interfaz que representa todos los campos que modelan a una nota.

```
deleteNoteDB(id: string) {  
  return this.http.delete(`${apiURL}/notes/${id}`  
    , {observe: 'response'}).pipe(  
    map((response: any) => {  
      if(response.status == 200) {  
        return true;  
      } else {  
        return false;  
      }  
    })  
  );  
}  
  
updateNote(nota: NotaInterface) {  
  return this.http.put(`${apiURL}/notes/${nota.id}`  
    , nota, {observe: 'response'}).pipe(  
    map((response: any) => {  
      if(response.status == 200) {  
        return true;  
      } else {  
        return false;  
      }  
    })  
  );  
}
```

Imagen 19. Operaciones de eliminación y actualización en el servicio de notas.

Una vez terminado el código de los servicios se procede a modelar el dinamismo en cada uno de los componentes generados, empezando por el etiquetado como notes-component. Este componente esta pensado para mostrar la lista de notas que se encuentran en la base de datos. Para hacer esto primero se genera un fragmento dentro del ngOnInit en el que se guardaran todos los objetos en un atributo del componente (Imagen 20).

```
this.noteService.getNotes().subscribe((notas) => {  
  this.noteArray = notas;  
});
```

Imagen 20. llamado a servicio de notas para obtener registros y guardarlos en atributo del notes-component.

Teniendo como base el arreglo de notas que contiene los registros de la base de datos, se hace un ciclo con ngFor para iterar sobre ellos y poblar el contenido de cada una de las tarjetas que forman parte de la vista (Imagen 22). Adicionalmente se asocian dos métodos con el evento click del botón de eliminar y con el evento click del boton utilizado para editar.

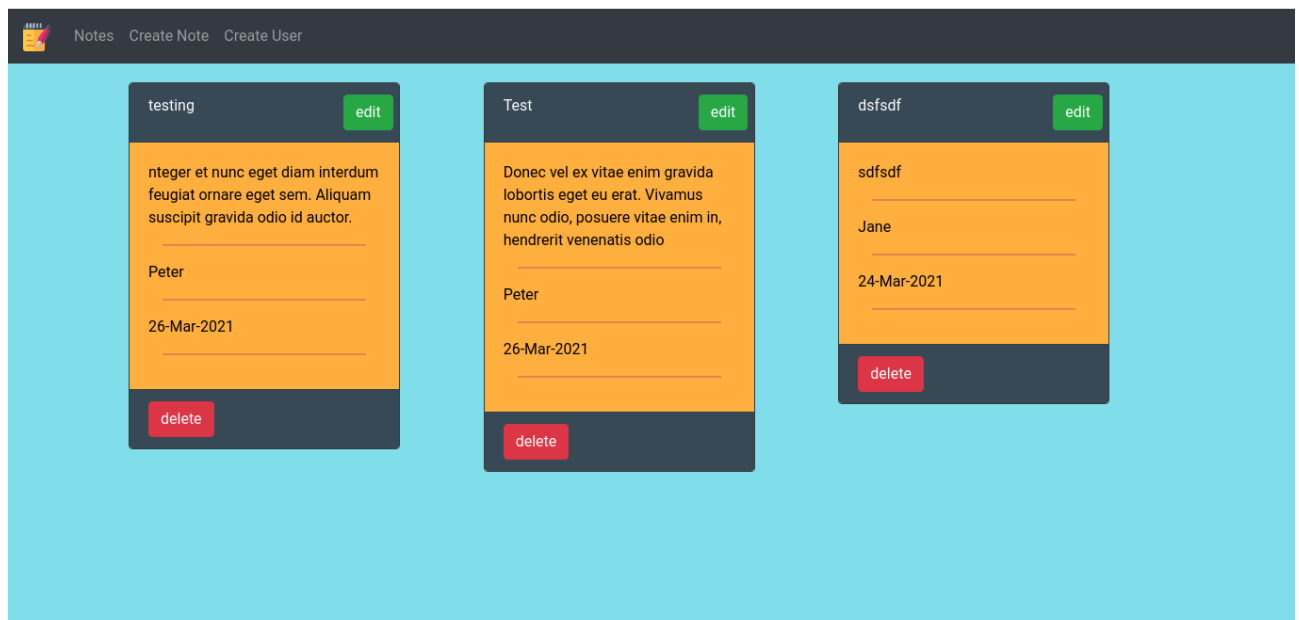


Imagen 21. Tarjetas del componente notes-component pobladas con el contenido de la base de datos.

Para el caso del evento click encargado de eliminar, simplemente se liga con un método que recibe como parámetro el índice del registro que se usa para extraer la nota específica de la lista de notas. Una vez obtenida la lista y principalmente el id, se llama al método para eliminación del servicio de notas y se pasa como argumento el id del elemento seleccionado. Antes de borrar de manera definitiva se despliega una ventana en la que se pedirá la confirmación, en dicho caso se elimina el registro y se actualiza la lista para no tener que recargar la página.

```
editarNota(index: number) {
  this.noteService.storeNote(this.noteArray[index]);
  this.router.navigate(['note']);
}

eliminarNota(index: number) {
  Swal.fire({
    title: 'Esta seguro de borrar esta nota?',
    showDenyButton: true,
    icon: 'warning',
    showConfirmButton: false,
    showCancelButton: true,
    denyButtonText: `Borrar definitivamente`,
  }).then((result) => {
    // se hace la operacion con el boton deny solo por estetica
    if(result.isDenied) {
      // llamado a la base de datos para eliminar registro
      const id = this.noteArray[index].id;
      this.noteService.deleteNoteDB(id).subscribe((res) => {
        if(res){
          Swal.fire('se ha eliminado la nota!', '', 'success');
        } else {
          Swal.fire('algo salio mal', '', 'error');
        }
      });
    }
    // se elimina nota de la lista de elementos interna y tambien
    // en el html
    this.noteArray.splice(index,1);
  });
}
```

Imagen 22. Métodos asociados al evento click de eliminar y editar en cada nota.



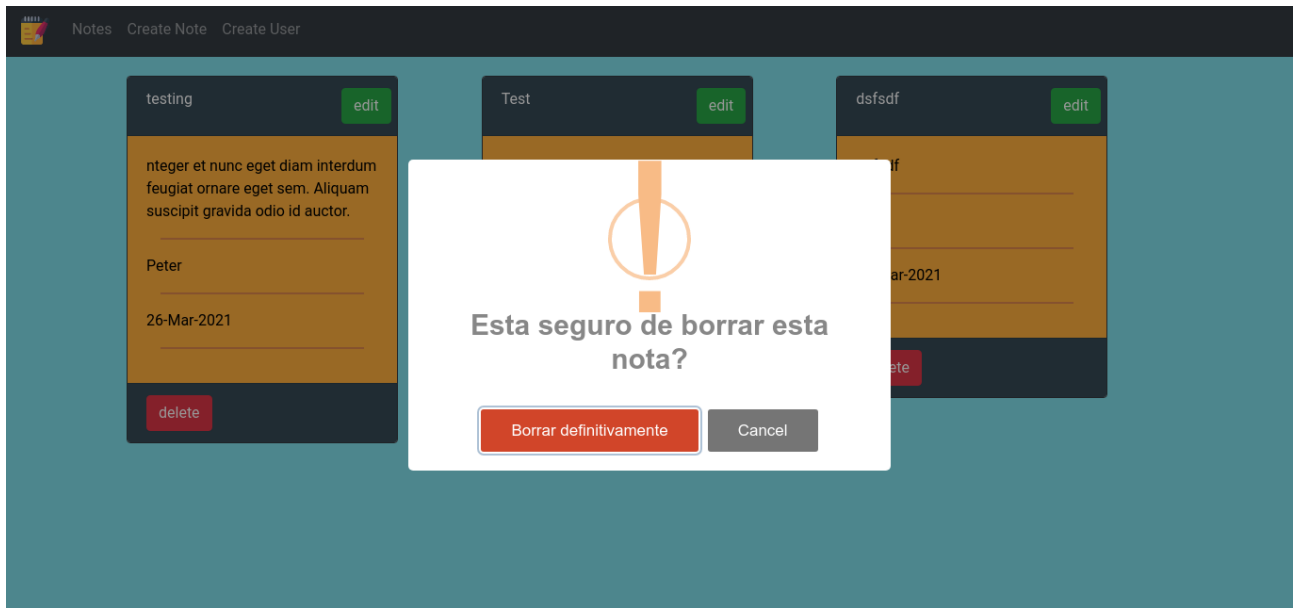


Imagen 23. Ventana de diálogo para confirmar la eliminación de un registro.

En el caso del evento para editar una nota, solo cuenta con dos líneas. En este fragmento se utiliza un servicio como intermediario para alojar el registro que se desea editar y se hace un cambio de ruta hacia el componente que contiene el formulario de notas. En este componente se modela el comportamiento para que sea posible detectar cuando se desea crear una nueva nota o cuando solo se quiere editar una como se puede observar en las imágenes 24 y 25.

A screenshot of a web application interface. At the top, there is a dark navigation bar with a logo and links for 'Notes', 'Create Note', and 'Create User'. The main area has a light blue background. In the center, there is a white form with a dropdown menu at the top. Below the dropdown are three text input fields labeled 'Title', 'Contenido', and 'Fecha'. The 'Fecha' field has a placeholder 'mm/dd/yyyy' and a calendar icon. At the bottom of the form is a blue button labeled 'Guardar'.

Imagen 23. Aspecto del formulario cuando se quiere crear una nota.

The image shows a web application interface for editing a note. The top navigation bar is dark grey with links for 'Notes', 'Create Note', and 'Create User'. The main content area has a light blue background. A white form is centered, containing the following fields: a dropdown menu for user selection (currently showing 'Peter'), a text input for 'Title' (containing 'testing'), a text input for 'Contenido' (containing 'nteger et nunc eget diam interdum feugi'), and a date input for 'Fecha' (containing '03/26/2021'). A blue 'Editar' button is at the bottom of the form.

Imagen 24. Aspecto del formulario cuando se quiere editar una nota.

Para manipular el formulario del componente etiquetado como `note-form.component` se hace uso del objeto `FormBuilder` de Angular, este objeto permite tener más control sobre su contenido de manera programática. En el `FormBuilder` se agrega un objeto que se compone por el nombre del campo que se va a asociar al formulario del html, su valor y de manera opcional un arreglo de validaciones. De forma adicional, como este formulario requiere un campo select que contenga todos los usuarios se integra el servicio de usuarios, se llama al método encargado de leer todos los registros y se hace el llenado correspondiente.

```
this.noteToEdit = this.noteService.retrieveNote
();
// en caso de que se haya solicitado la edicion d
e una nota se ligam los valores
// de la nota al formulario, en caso contrario se
dejan vacios
if(this.noteToEdit) {
  this.noteForm = fb.group({
    creador: [this.noteToEdit.nombreCreador,
Validators.required],
    titulo: [this.noteToEdit.titulo, Validators.
required],
    informacion: [this.noteToEdit.informacion,
Validators.required],
    fecha: [ this.dateFormat(this.noteToEdit.
fecha), Validators.required],
  });
} else {
  this.noteForm = fb.group({
    creador: ['', Validators.required],
    titulo: ['', Validators.required],
    informacion: ['', Validators.required],
    fecha: ['', Validators.required],
  });
}

// se obtiene lista de usuarios para mostrarlos e
n el contenedor correspondiente
this.userService.getUsers().subscribe(users => {
  this.listaNombres = users;
});
```

Imagen 25. Código correspondiente al FormBuilder del formulario de notas.

Como se puede notar en la imagen 25, antes de llamar al FormBuilder se busca en el servicio de notas si hay algún objeto que se haya solicitado editar, en caso de encontrarlo los valores de cada campo del FormBuilder se llenan con la información que viene por default del registro que se va a editar, en caso contrario se dejan los campos en blanco. También se puede notar que cada campo tiene un validador. Este validador se utiliza principalmente para verificar que los campos no se hayan dejado vacios. En caso de ser así se muestra una ventana al usuario indicando que debe llenar todos los campos y también se utiliza para dibujar de color rojo los campos que no cumplan con la condición como se muestra en la imagen 26.

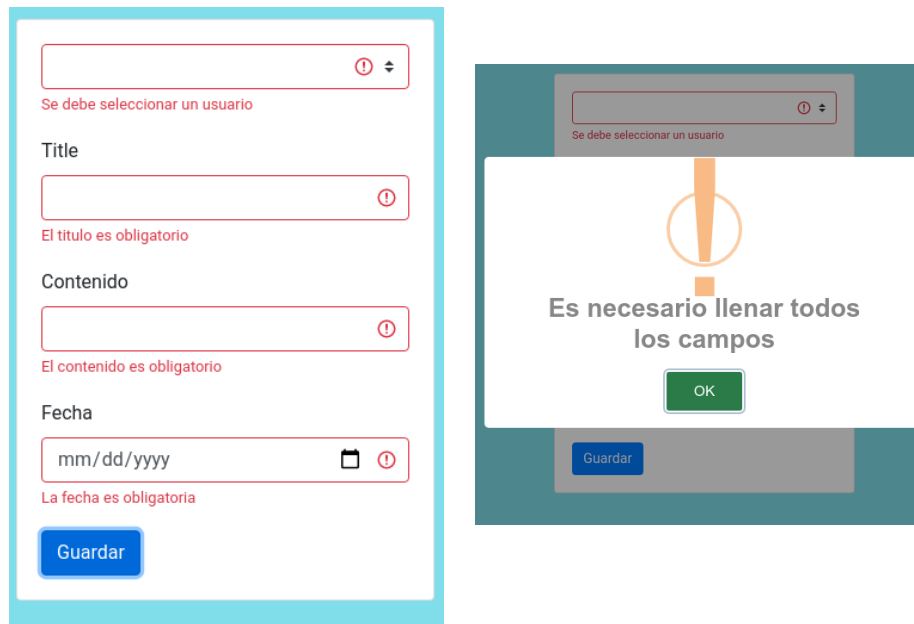


Imagen 26. Errores mostrados al usuario si se dejan vacios los campos.

Después de manejar los posibles errores se procede a modelar el comportamiento del formulario cuando todos los valores son correctos. En este caso se extraen los valores de cada campo a través del FormBuilder y se guardan en un objeto. Si la operación es de guardado se usa como apoyo una función donde se llama a la petición de creación del servicio de notas. En caso de ser una modificación se constuye de igual forma un objeto a partir de los valores ingresados al FormBuilder y de manera adicional se toma el id correspondiente. Finalmente se usa como apoyo una función que se encarga de llamar a la petición de actualización por medio del servicio de notas.

```
guardarNota() {
  if(!this.noteForm.valid) {
    for (const control of Object.values(this.noteForm.controls)) {
      control.markAsTouched();
    }
    Swal.fire('Es necesario llenar todos los campos', '',
      'warning');
  } else {
    // se guarda contenido de formulario en un objeto que sera enviado
    a la base de datos

    let nuevaNota: NotaInterface;
    nuevaNota = {
      titulo: this.noteForm.get('titulo').value,
      fecha: this.noteForm.get('fecha').value,
      informacion: this.noteForm.get('informacion').value,
      nombreCreador: this.noteForm.get('creador').value,
    }

    // si se solicito editar una nota se llama a la peticion para actu
    alizar
    if(this.noteToEdit) {
      nuevaNota.id = this.noteToEdit.id;
      this.peticionActualizar(nuevaNota);
    } else {
      // se guarda el contenido de la nueva nota
      this.peticionGuardar(nuevaNota);
    }
  }
}
```

Imagen 27. Lógica utilizada para crear o actualizar una nota.

Como nota adicional, cuando se guarda o edita una nota y se hace de manera exitosa se muestra un cuadro de diálogo y se hace un redireccionamiento a la página donde se encuentra la lista de notas (Imagen 28).

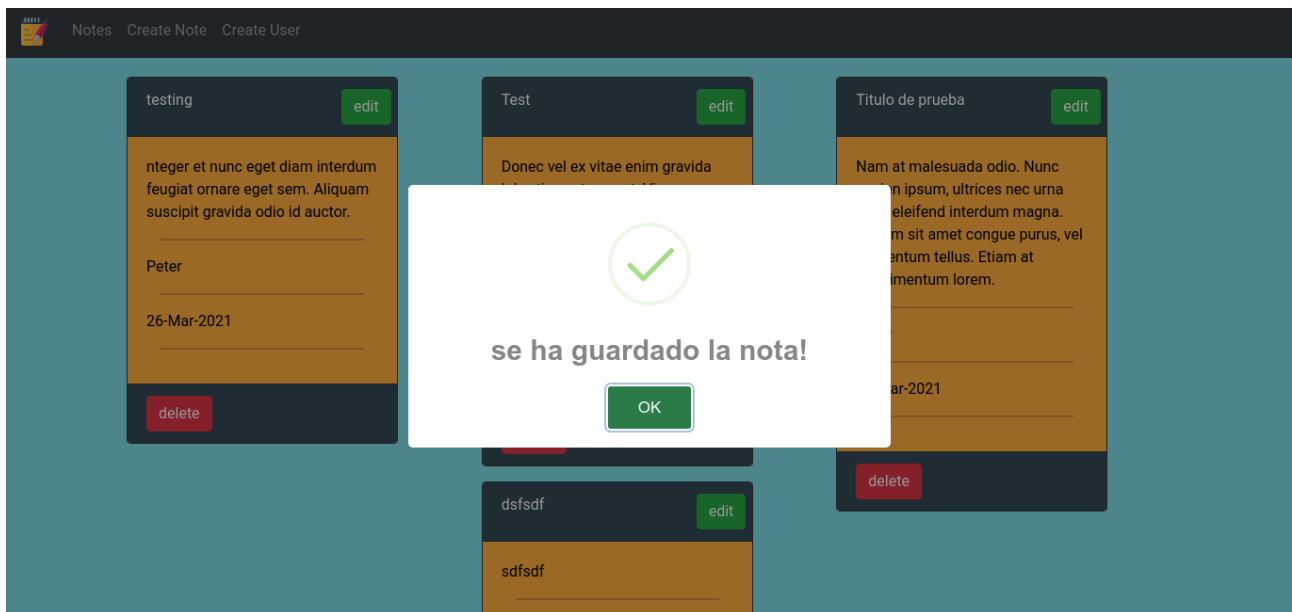


Imagen 28. Operación de guardado de nota exitosa.

Para el caso de creación de usuario se utiliza una lógica similar dejando del lado la validación adicional que se hizo en el de notas para el caso donde se tenía que realizar una operación de actualización. En este caso se utilizó un formulario más simple que se compone por el campo de nombre. Para la parte del listado de usuarios se hace un llamado al método de obtención del servicio de usuarios y se hace el poblado del html usando la directiva ngFor.

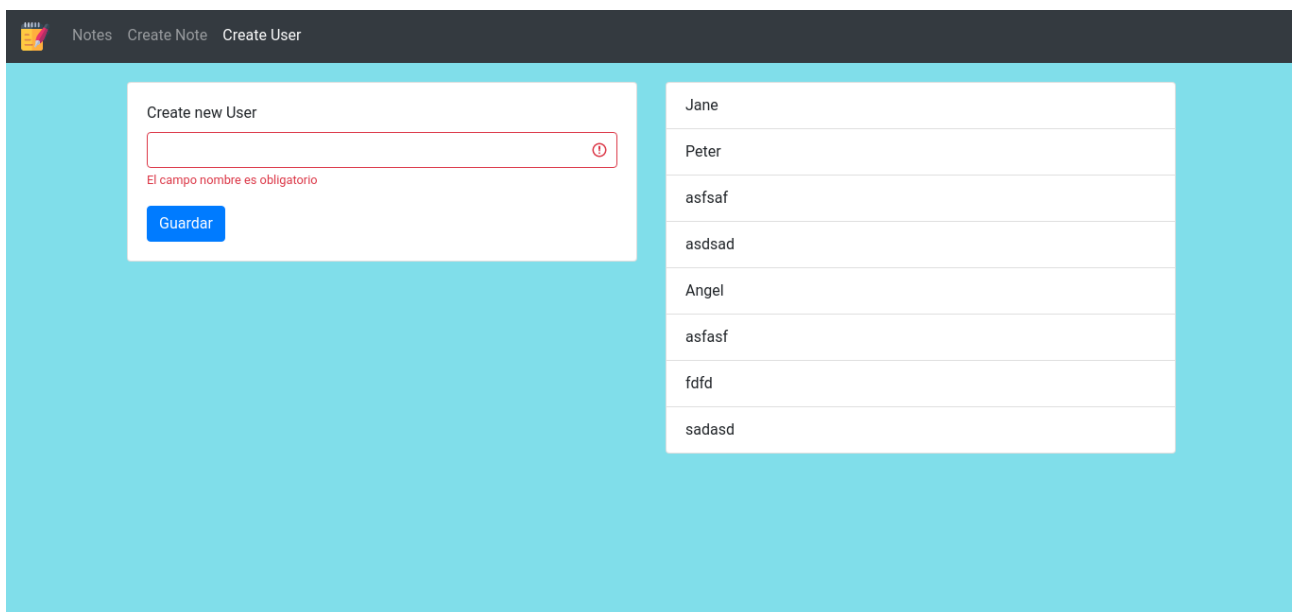


Imagen 29. Vista del formulario de usuario y de la lista de usuarios existentes en la base de datos.

Como se muestra en la imagen 29 también se integraron las validaciones pertinentes para que notificar que el campo no debe dejarse vacío. Además se agrega otra validación para verificar que no se agregue un registro repetido a la lista de usuarios, en caso de intentarlo se mostrará una ventana al usuario indicándole el problema (Imagen 30).

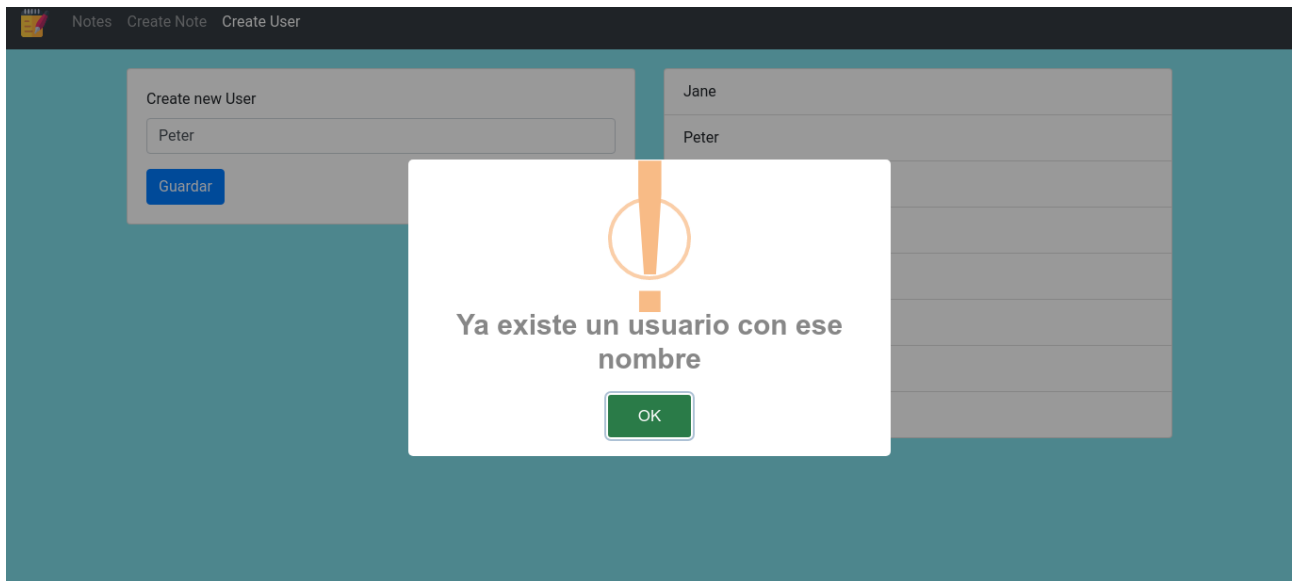


Imagen 30. Tratando de agregar registro repetido a la lista de usuarios.

Finalmente, en el caso de que si se cumplan todas las condiciones para la creación de un registro, se le notifica al usuario por medio de una ventana que la operación se ejecutó de manera correcta y se renderiza el nuevo registro en la lista de usuarios (Imagen 31).

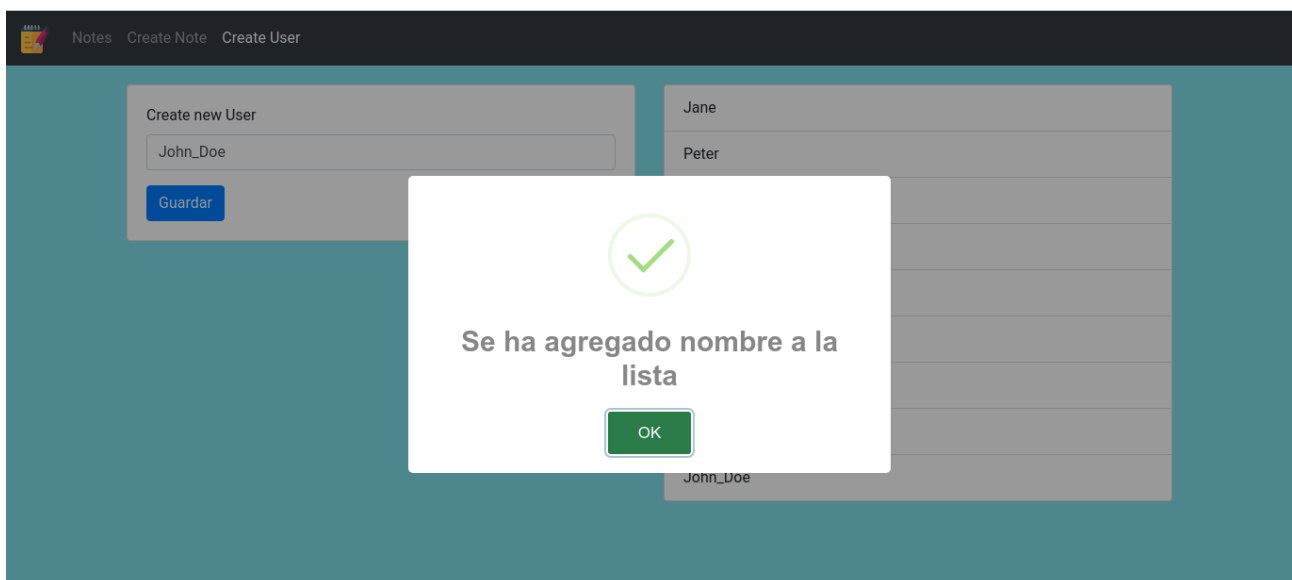


Imagen 31. Vista mostrada al crear un usuario de forma correcta.

Anexo A. Link de aplicación en producción.

<https://immense-wildwood-39793.herokuapp.com/#/>

Anexo B. Repositorios generados.

## Backend

<https://github.com/unityofdisaster2/backend-notes-app-sps>

## FrontEnd

[https://github.com/unityofdisaster2/prueba\\_sps](https://github.com/unityofdisaster2/prueba_sps)

Anexo C. Aspectos a mejorar:

- Agregar una animación o icono que indique cuando están cargando las notas al frontend.
- El diseño de los formularios y de la lista de usuarios.
- Agregar un componente para manejar de forma individual las tarjetas que representan las notas en lugar de manejar todo en el componente notes.

Créditos por el icono de notas utilizado.

creditos      <div>Icons      made      by      <a      href="https://www.freepik.com"  
title="Freepik">Freepik</a>      from      <a      href="https://www.flaticon.com/"  
title="Flaticon">[www.flaticon.com](https://www.flaticon.com/)</a></div>