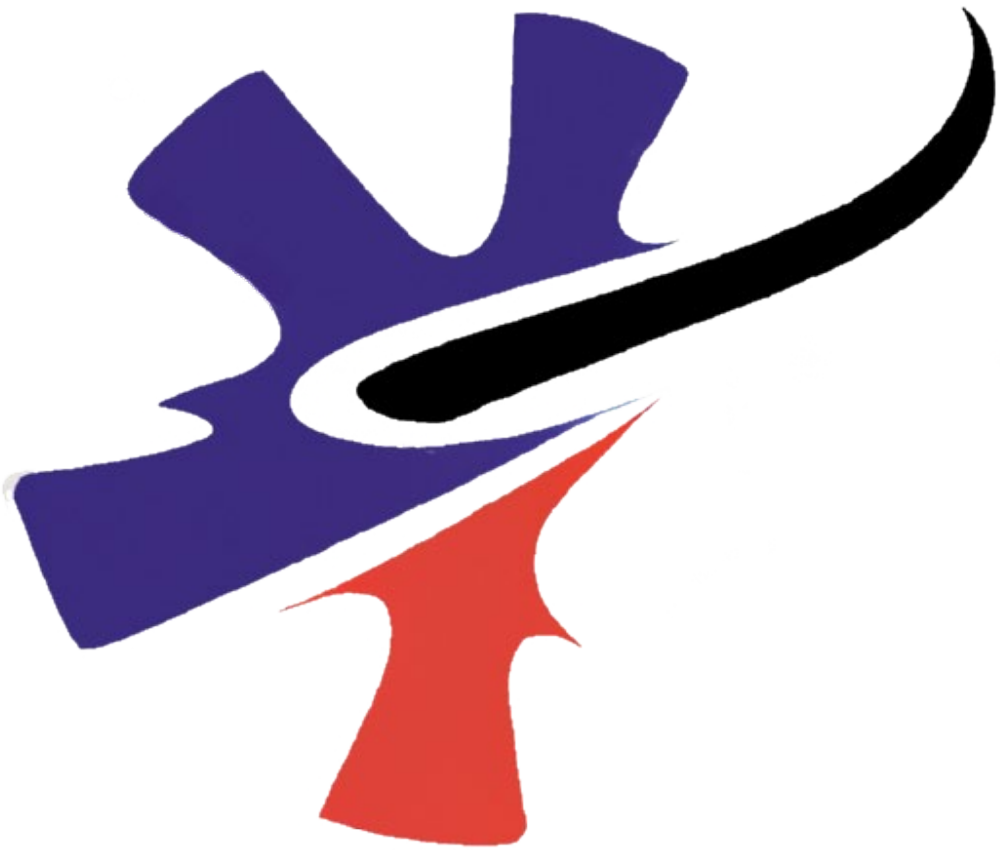
Trabajo de Fin de Grado

NeuroNavigator: Gestor de pacientes



Guillermo Rojo Santos

Contenido

Resumen

En este TFG desarrollaremos NeuroNavigator. El objetivo de esta aplicación es ofrecer a psicólogos/as una forma fácil y segura de almacenar los datos y documentos relacionados con sus respectivos pacientes haciendo uso de una base de datos y un servidor SFTP (a elección del usuario), para poder acceder a ellos de forma fácil y remota.

El hecho de tener los datos registrados digitalmente brinda a los usuarios finales una mayor facilidad de gestión a la hora de editar los registros de un paciente y acceder a los datos en cualquier momento.

Al igual que la base de datos, el servidor SFTP (Secure File Transfer Protocol) puede ser contratado o bien montado por el propio usuario; aunque su uso, si bien está implementado en la aplicación es opcional y no supone ningún problema.

Para alcanzar un mayor número de usuarios, NeuroNavigator cuenta con la capacidad de traducir su interfaz en tres idiomas distintos: Español, Francés e Inglés.

Introducción

Como ya hemos explicado, NeuroNavigator se centra en ser una herramienta para la fácil accesibilidad tanto de los pacientes como sus archivos relevantes, ya que, en el campo de la psicología, como en la mayoría de campos de la medicina, la información que se tiene de un paciente es extremadamente importante. Para ello, nuestra aplicación ofrece al usuario una interfaz simple pero ordenada, con todas las funcionalidades relevantes para una aplicación de su estilo. Ofrecemos también un modelo de paciente bastante amplio, y que, aunque esté limitado, puede ser expandido gracias a la posibilidad de adjuntar a cada paciente cuántos archivos sean necesarios.

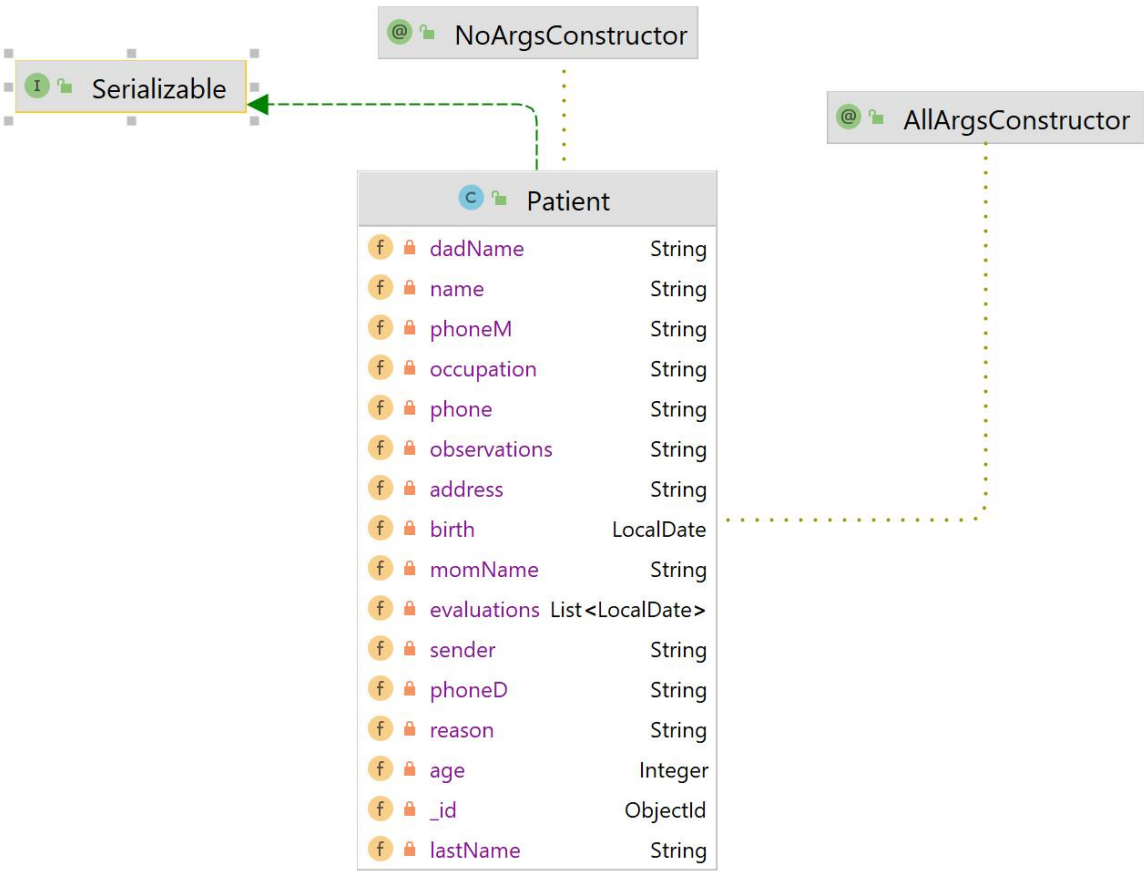


Ilustración 1Campos disponibles del paciente

Como podemos ver en la *Ilustración* 1, en un paciente podemos guardar tanto información de la persona que representa el paciente (Nombre, Apellido, Fecha de nacimiento, etc.), como información relevante para el psicólogo/a (Fechas de las evaluaciones u observaciones sobre el Paciente). Además, el servidor SFTP puede guardar cualquier tipo de archivos (Documentos de Word, Excel, PowerPoint, resultados de pruebas, etc.).

Una vez expuesta a naturaleza y propósito de NeuroNavigator, entramos en detalle de este documento. Aquí abordaremos el funcionamiento de las distintas funcionalidades de la aplicación, las tecnologías implicadas en el desarrollo de esta (tanto Hardware como Software), las distintas etapas de desarrollo por las que pasó el proyecto, las distintas pruebas que se hicieron y una conclusión sobre el producto final.

Objetivos y Características

En el campo de la psicología vemos que un paciente genera una gran cantidad de datos, ya sean observaciones del psicólogo/a, resultados de pruebas, expedientes médicos y más; es por ello que NeuroNavigator nos ofrece una solución ágil y relativamente fácil de implementar en cualquier gabinete de psicología o centro médico que posea un departamento de psicología. Para el almacenamiento de datos, NeuroNavigator usa *MongoDB*, que es una B*ase de datos NoSQL* que guarda a los pacientes en formato Json. La configuración de esta es excepcionalmente sencilla. Nos da algunas opciones como la de gestionar los distintos usuarios que pueden tener acceso a la base de datos al igual que las IPs que tienen acceso a esta, añadiendo así una capa de seguridad extra a nuestra base de datos. Además, su plan gratuito nos ofrece un almacenamiento de 512MB, y un paciente simple ocupa solo una media de 450 Bytes, con estos números podemos hacer el siguiente cálculo:

*(512 MB * 1024 KB/MB * 1024 bytes/KB) / 450 bytes*

Lo que nos da un total de 587202 pacientes “simples” (es decir, pacientes que tengan sus campos básicos y obligatorios rellenados y una cantidad razonable de fechas de evaluación) aproximadamente. Otra característica interesante de NeuroNavigator es la gestión de campos. Hemos mencionado previamente, un paciente posee campos obligatorios y opcionales.

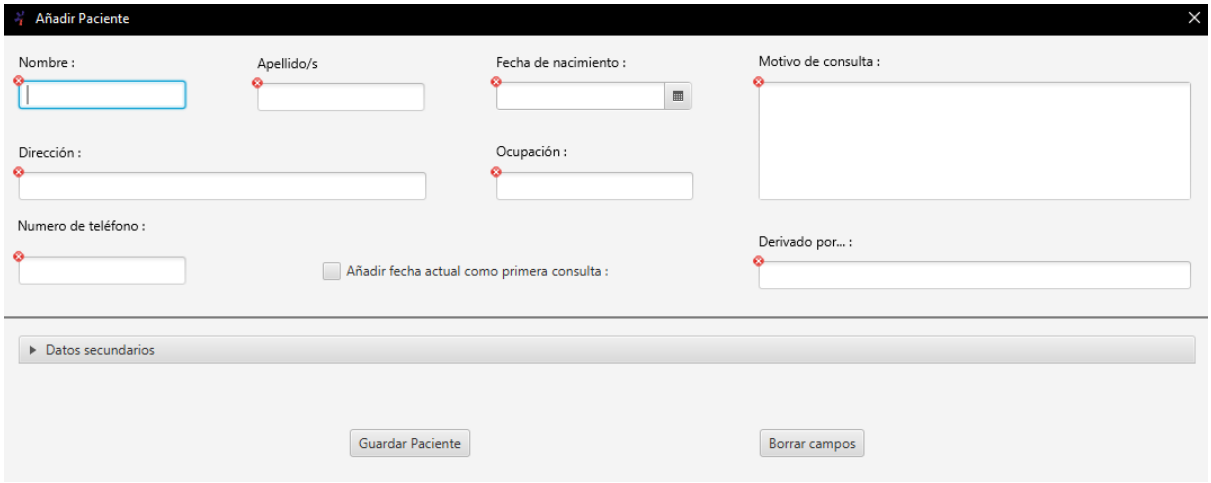


Ilustración 2 Campos obligatorios de NeuroNavigator

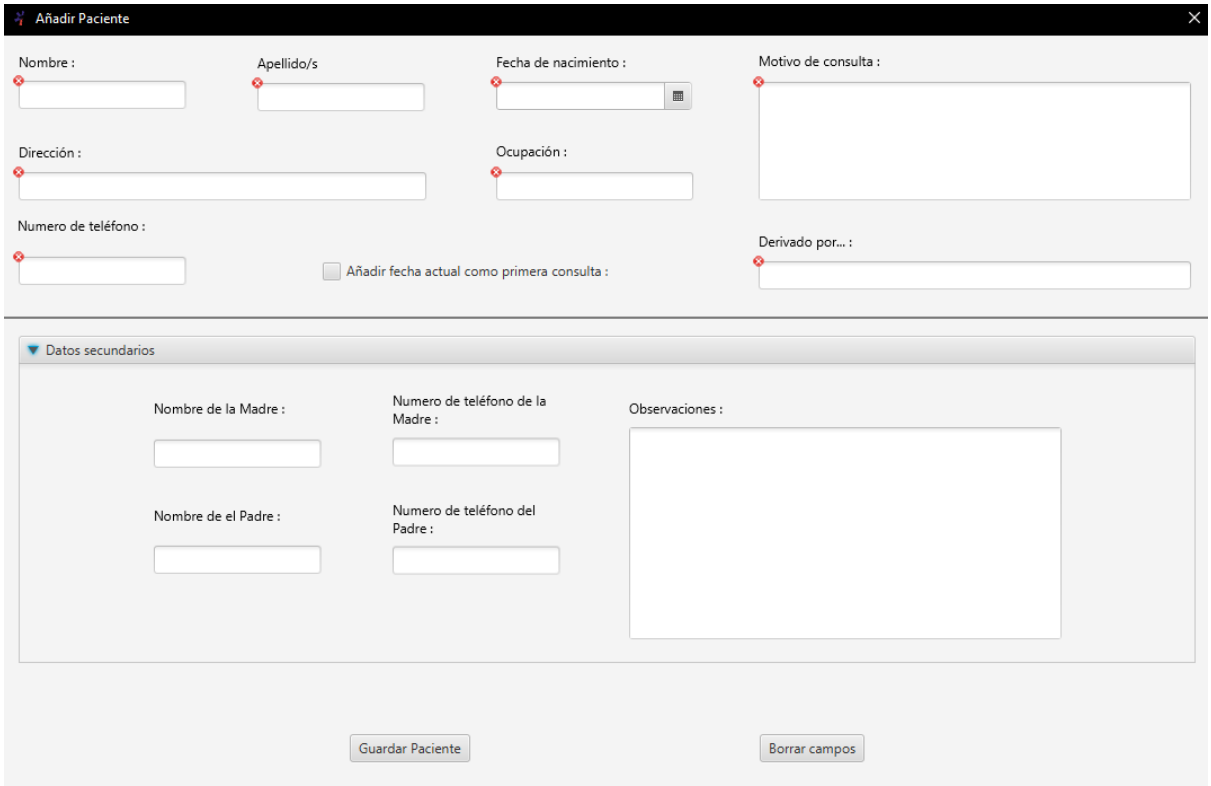


Ilustración 3 Campos opcionales de NeuroNavigator

En la *Ilustración 2* podemos observar los campos obligatorios del formulario del paciente. Al rellenar y agregar un paciente de la base de datos, NeuroNavigator se encargará de asegurarse de que todos los campos obligatorios estén rellenados y, en el caso de que tengan que llevar un formato concreto, de que ese formato sea el correcto.

En la *Ilustración 3* podemos ver el apartado de campos opcionales, que para ofrecer una interfaz más minimalista y sencilla está replegado al abrir el formulario.

Otra de las características principales de NeuroNavigator es su módulo de conexión SFTP. Gracias a él, nos podemos conectar a cualquier servidor SFTP siempre que esté bien configurado, y dado que la transferencia de archivos se encripta usando SSL, los archivos están encriptados de punta a punta (Aunque debido a la naturaleza privada de los documentos que están destinados a almacenarse en el SFTP, recomendamos que estos se encripten por parte del usuario o que se protejan por contraseña antes de subirlos al servidor).

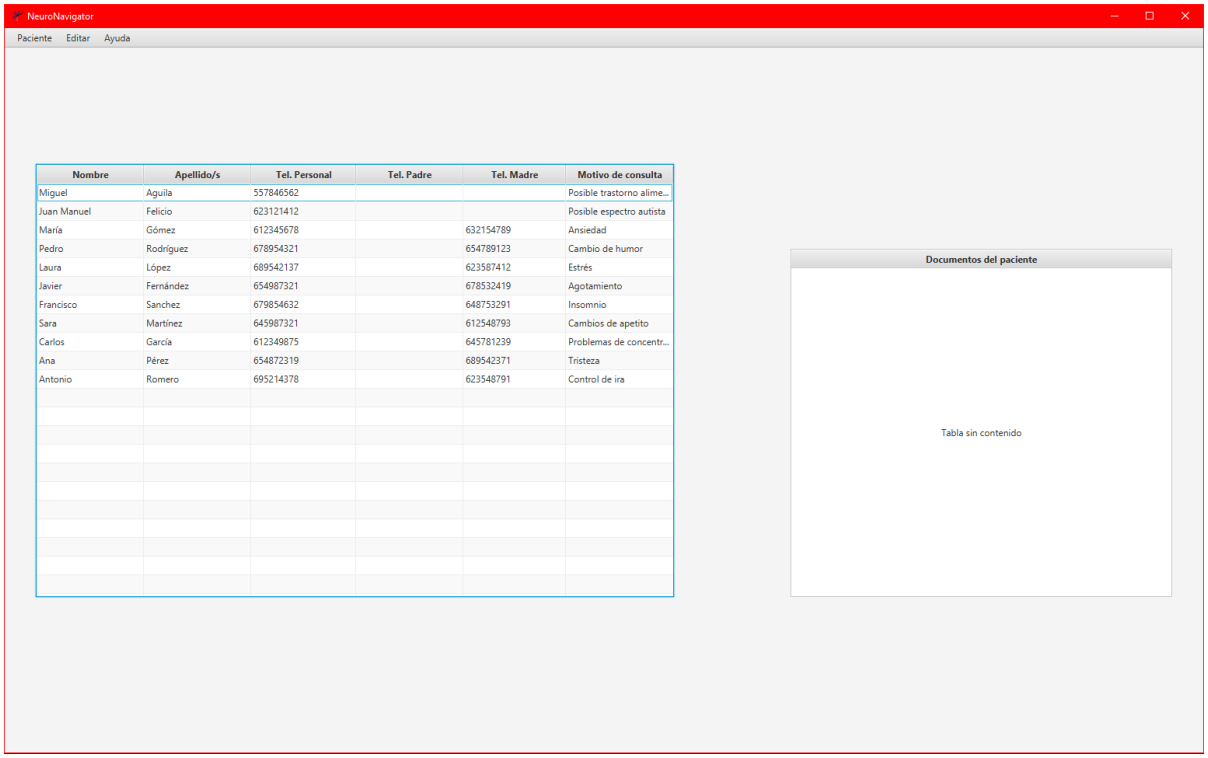


Ilustración 4 Interfaz Simple de NeuroNavigator

Otra de las características de NeuroNavigator es su interfaz sencilla. Como podemos ver en la *Ilustración 4* Con una pantalla principal que muestra directamente los datos juzgados como más relevantes, nuestra aplicación busca a adaptarse a todos sin importar el grado de manejo de informática que posea.

Finalidad

NeuroNavigator está diseñada para ser usada en gabinetes de psicología, concretamente negocios autónomos con uno o máximo dos psicólogos, ya que es una herramienta que, si bien las herramientas para almacenar los datos cuentan con medios para configurar distintos usuarios, NeuroNavigator se ideo y desarrollo para ser usado por una única persona. Pese a estas limitaciones previamente mencionadas, nuestra aplicación se puede adaptar a múltiples usuarios, siempre y cuando cada uno tenga su propio equipo informático ya que NeuroNavigator no cuenta con un gestor de usuarios e inicios de sesión.

Como ya hemos dicho, gracias a NeuroNavigator buscamos ofrecer una organización más limpia y accesible para el psicólogo/a, además de utilidades adicionales para llevar un seguimiento de los pacientes, tanto las fechas de las consultas como observaciones del paciente y la edición rápida de sus datos.

Medios usados

Ya hemos mencionado anteriormente algunas de las tecnologías utilizadas en este proyecto como lo son MongoDB o servidor SFTP, pero en este apartado entraremos en detalle de cada uno de las tecnologías utilizadas.

Maven:

Ilustración 5 Logo de Maven

Maven es una herramienta de gestión de proyectos ampliamente utilizada en el desarrollo de software. Es una herramienta de automatización y construcción que ayuda a los desarrolladores a gestionar eficientemente las dependencias, compilar el código fuente, ejecutar pruebas, empaquetar y distribuir sus aplicaciones. Además, gracias a él, podremos añadir librerías extra a Java fácilmente a través del archivo pom.xml.

Java Y JavaFX:

Ilustración 6 Logos de JavaFX y Java

El lenguaje principal en el que está desarrollado NeuroNavigator es Java. Con este lenguaje de programación orientado a objetos nos conectamos al servidor de los pacientes y al SFTP a través de distintas tecnologías que detallaremos más adelante.

Para la concepción de la interfaz se usó JavaFX, una plataforma open source para crear nuestras propias interfaces de usuario. A esta plataforma le añadimos ValidatorFX; una librería inspirada en ControlFX que intenta mejorar sus deficiencias. Con Esta librería, validamos los campos de los formularios en la aplicación de forma fácil y rápida, ya que para validar los campos solo hay que crear un nuevo Validator() y asignarle, dentro del controlador de la ventana correspondiente, el nodo de JavaFX que se quiere validar. A partir de ahí, podemos especificarle al Validator que tiene que comprobar en ese campo y cuando. A continuación, tenemos un ejemplo de un Validator en uso:

```
Validator validator = new Validator();

//Nodo de JavaFX
@FXML
private TextField txtName;
```

```
/*
**
resto del código del controlador
**
*/

//Añadir al validator un check
validator.createCheck().dependsOn("name", txtName.textProperty()).withMethod(c -> {
        String userName = c.get("name");
        if (userName.length() < 2) {
            c.error(MainApplication.resourceBundle.getString("add_errors_name1"));
        }
    }).decorates(txtName).immediate();
```

Aquí podemos ver lo fácil que es añadir validadores a los campos de la aplicación, lo cual facilita mucho el trabajo.

Otra ventaja de ValidatorFX es que, como podemos ver en las últimas líneas de código, podemos añadirle la opción de que añada una especie de Tooltip al nodo de JavaFX con un texto determinado para ayudar al usuario a saber cómo rellenar correctamente el campo. Queremos subrayar también que un mismo nodo puede tener varios validators, y un validator puede tener varias condiciones siempre y cuando esas condiciones dependan de la misma propiedad del nodo (Ej. Comprobar la longitud del campo, ver que contenga por lo menos una mayúscula, y comprobar que haya por lo menos un carácter especial; todo esto depende del #.textProperty() del input, por lo que se podría poner en el mismo validador).

MongoDB y MongoDB Atlas:

Ilustración 6 Ltogo de MongoDB Atlas

MongoDB ofrece Atlas, una base de datos NoSQL. Esto significa, literalmente, Not Only SQL o No solamente SQL, lo que implica que Atlas no se limita únicamente al uso del lenguaje de consulta SQL tradicional utilizado en las bases de datos relacionales.

En lugar de eso, Atlas adopta un enfoque diferente al almacenamiento y acceso a los datos, utilizando un modelo de datos flexible y escalable conocido como "documentos". En Atlas, los datos se organizan en documentos Bson (Binary Json), que son estructuras de datos similares a Json, pero con soporte para tipos de datos binarios y otras características adicionales. Por otra parte, las bases de datos de Atlas están organizadas de la siguiente manera:

Clúster -> Base de datos -> Colección -> Documentos

Un Clúster contiene varias bases de datos, una base de datos contiene múltiples colecciones, una colección contiene múltiples documentos, y un documento representa un objeto; en este caso, un Paciente.

Para poder conectarnos a través de Java a nuestro servidor de Atlas, tendremos que usar el driver de MongoDB. Para poder usarlo, lo añadiremos al proyecto a través del pom.xml de Maven:

```
<dependency>
   <groupId>org.mongodb</groupId>
   <artifactId>mongodb-driver-sync</artifactId>
   <version>4.9.1</version>
</dependency>
```

El funcionamiento del Driver es muy simple. Para establecer una conexión necesitaremos lo siguiente:

- Cadena de Conexión: Cadena proporcionada por MongoDB en la cual se declara el usuario y contraseña, además de otros parámetros como el tipo de verificación o la utilización de los datos a través del protocolo SSL.
- Base de datos: El nombre de la base de datos que hayamos creado desde la página en nuestro Clúster.
- Colección: Nombre de la colección a la que queramos acceder dentro de nuestra base de datos previamente especificada.
- Codecs: Los Codecs son una parte muy importante del driver. Como ya hemos dicho, los datos en MongoDB se almacenan en formato Bson. Es decir, al verlos en la base de datos son Json normales, pero a la hora de hacer una Query a la base de datos, hay que encargarse de traducir los datos recogidos al tipo de dato del atributo del paciente correspondiente. De esto se encargan precisamente los Codecs. A la hora de cargar los Codecs, tendremos que cargar dos:

- Codecs de tipo POJO: Son unos Codecs proporcionados por el Driver de MongoDB, específicamente pensados para convertir las propiedades de un POJO (Plain Old Java Object) a campos de un Bson.
- Default Codecs + Codecs POJO: Realmente solo cargamos unos Codecs, los de POJOs. Pero hay que especificarle al driver de MongoDB que Codecs tiene que utilizar, es por eso que hacemos una primera declaración cargando los Codecs POJO y luego cargamos los predeterminados y los juntamos de la siguiente forma:

```
this.pojoCodecRegistry = fromProviders(PojoCodecProvider.builder().automatic(true).build());

this.codecRegistry = fromProviders(MongoClientSettings.getDefaultCodecRegistry(),
pojoCodecRegistry);
```

- Configuración del cliente: La configuración es el último paso antes de la conexión. Aquí juntamos todos los datos que hemos mencionados previamente y se los asignamos al constructor de un nuevo cliente de mongo. En NeuroNavigator tenemos un solo objeto (Ilustración 7) que nos proporciona los métodos necesarios para las operaciones de nuestra aplicación especificadas en una interfaz. Dentro de este objeto, tenemos un constructor al cual añadimos los datos para luego usar los métodos preestablecidos por una interfaz para hacer las operaciones necesarias para nuestra aplicación.

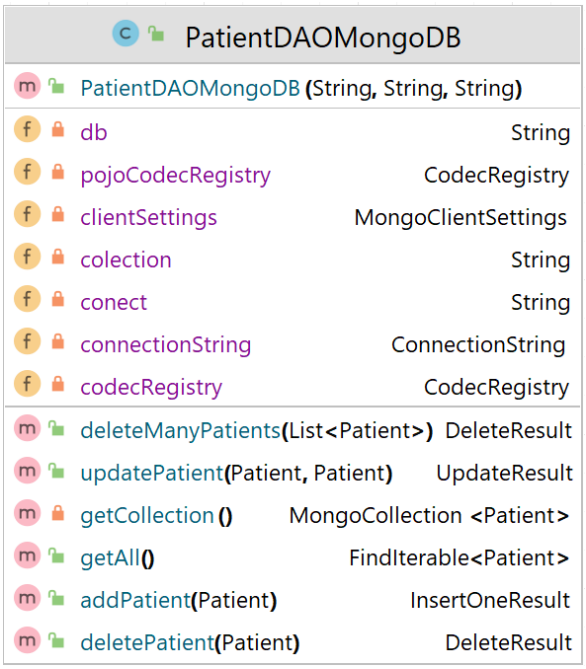


Ilustración 7 Diagrama de la clase PatientDAOMongoDB

SSHJ

Es una librería de Java especializada en la conexión y trasferencia de datos a través de canales SFTP (Secure File Transfer Protocol). El SFTP es un protocolo seguro diseñado para la transferencia de archivos sobre una conexión SSH (Secure Shell). A diferencia de otros protocolos de transferencia de archivos, como FTP (File Transfer Protocol), SFTP utiliza una conexión cifrada y autenticada para garantizar la confidencialidad y la integridad de los datos transferidos.

Cuando se establece una conexión SFTP, se inicia una sesión SSH para autenticar al cliente y al servidor. Una vez establecida la conexión, se utiliza el protocolo SFTP para realizar operaciones de transferencia de archivos y administración de archivos en el servidor remoto.

SFTP se encarga de transferir archivos en bloques de datos, lo que permite una transferencia eficiente y confiable, especialmente cuando se trata de archivos de gran tamaño. Además, SFTP ofrece la posibilidad de comprimir y encriptar los datos durante la transferencia, lo cual añade una capa adicional de seguridad para proteger la información.

Pasando al aspecto más técnico de la librería. SSHJ nos permite usar tanto usuario y contraseña como la llave pública del servidor para la autenticación de este. Para esta última opción tenemos dos formas de hacerlo:

1. Cuando nos conectamos a un servidor a través de SSH, el servidor al que nos conectemos nos enviará su llave pública para que podamos verificar que es él con el que nos estamos comunicando. Una vez recibida esa clave, nuestro equipo la guarda en un archivo llamado KnownHosts, que es un registro de todos los hosts remotos a los que nos hemos conectado correctamente haciendo uso de SSH.

   Para verificar que el servidor SFTP al que se conecta NeuroNavigator es correcto, al establecer una conexión, nuestra aplicación abre ese archivo y compara la clave recibida por el servidor con las claves disponibles del archivo KnownHosts. En caso de que haya alguna coincidencia, SSHJ permitirá la conexión y nos dejará acceder a los archivos de forma remota

2. La segunda forma de hacerlo es obviando esta capa de seguridad. A la hora de establecer la conexión, le podremos pasar al constructor del cliente SSH el objeto *PromiscuousVerifier.* Al hacer esto, SSHJ no comprobará la llave pública del servidor remoto y aceptará cualquier conexión que se le pida (siempre y cuando el host exista, independientemente de la seguridad que posea o de su veracidad).

Obviamente no se recomienda en ningún momento establecer la conexión a través de este último método ya que corremos el riesgo de sufrir un ataque conocido como “man-in-the-middle” u “hombre en el medio”. En este tipo de ataques, una entidad se hace pasar por nuestro servidor enviando una clave pública falsa, por lo cual la información que se le envía puede ser interceptada y leída por el sin que nosotros nos demos cuenta.

Para esclarecer cómo se establece esta conexión en nuestra aplicación, a continuación, mostramos el fragmento de código que se encarga de listar todos los archivos de un solo paciente:

```
@Override
public List<String> listFiles(String number) throws IOException {
    SSHClient client = new SSHClient();
    client.addHostKeyVerifier(new OpenSSHKnownHosts(new
File(String.format("%s/.ssh/known_hosts", System.getProperty("user.home")))));
    client.connect(FTP_HOST, Integer.parseInt(FTP_PORT));
    client.authPassword(FTP_USER, FTP_PASS);
    SFTPClient sftpClient = client.newSFTPClient();
    List<RemoteResourceInfo> files = sftpClient.ls("home/NeuroNavigator/" + number
+ "/");
    sftpClient.close();
    client.disconnect();
    List<String> result = new ArrayList<>();
    for (RemoteResourceInfo f : files) {
        result.add(f.getName());
    }
    return result;
}
```

Tanto *FTP_HOST, FTP_PORT, FTP_USER y FTP_PASS* son variables que se asignan en el constructor de la clase. Estos valores se sacan a su vez del archivo properties de NeuroNavigator, el cual explicaremos a continuación.

Properties Files

Los Properties Files o Archivos de Propiedades en español son un tipo de archivos de java con extensión *.properties* el cual nos permite almacenar valores de tipo *clave-valor* como un hashmap. Lo interesante de estos archivos es que, además de ser un tipo de archivo, estos también son una clase de Java; concretamente la clase Properties. A través de esta clase podremos disponer de varios métodos para poder acceder a las propiedades que estén dentro del documento.

Algunos de estos métodos son:

- Load(): Cargamos un archivo Properties a través de un archivo.
- GetProperty(): Leemos un valor del archivo Properties a través de su clave.
- SetProperty(): Añadimos una nueva propiedad o editamos una ya existente.
- Store() y StoreToXML(): Con Store() guardamos el archivo una vez editado, mientras que con StoreToXML() lo guardaremos en formato xml. A ambos métodos les tendremos que pasar un OutputStream o FileOutputStream para poder escribir el archivo en un lugar concreto de la máquina del usuario.

En nuestro caso, NeuroNavigator carga varios archivos Properties al iniciar. El primero de todos es el archivo Properties donde se encuentra la configuración de nuestra aplicación. Si en un principio no encuentra el archivo en la ruta donde debe estar, NeuroNavigator creará un nuevo archivo con las claves que se esperan vacías, y avisará al usuario mediante un cuadro de alera (Ilustración 8) de que la configuración no está establecida y que deberá cargarla por lo tanto la aplicación se abrirá, pero no cargará ningún paciente:

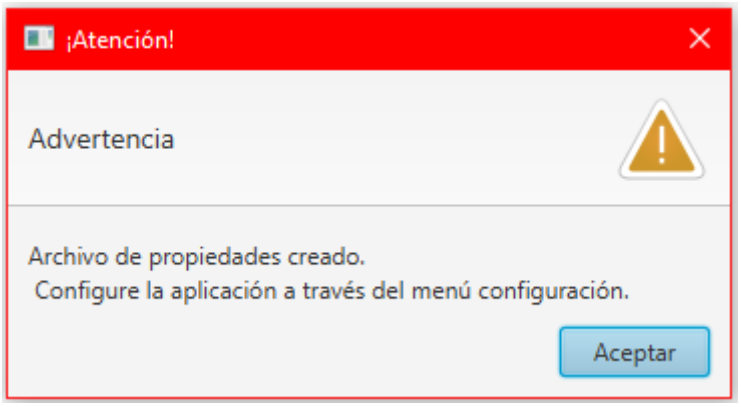


Ilustración 8 Cuadro de advertencia en caso de que no haya configuración

Además de cargar la configuración de la aplicación, utilizamos los archivos Properties para otro aspecto muy importante de NeuroNavigator: *La Internacionalización* .

Internacionalización:

Ya hemos mencionado anteriormente que uno de los aspectos más característicos de NeuroNavigator es la capacidad de ser usado en varios idiomas: español, francés e inglés.

Para lograr esto, volvemos a usar los archivos de tipo Properties de java. Esta vez, sin embargo, no se cargarán como un objeto Property, sino como un objeto llamado *ResourceBundle*.

Un *ResourceBundle* no es más que un conjunto de archivos Properties con un nombre de tipo: *NombreDelResourceBundle_IdiomaDelArchivo.* Por ejemplo, en NeuroNavigator tenemos el siguiente conjunto:

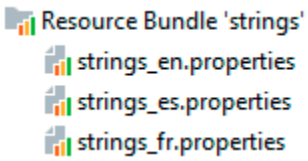


Ilustración 9 ResourceBundle de NeuroNavigator

Cada archivo que podemos ver en la Ilustración 9, está compuesto por las mismas claves y cambiando los valores de cada una en cada archivo por su equivalente en el idioma que corresponda (Ilustración 10).

```
mainTable_name:Nombre
mainTable_lastName:Apellido/s
mainTable_pNumber:Tel. Personal
mainTable_fNumber:Tel. Padre
mainTable_mNumber:Tel. Madre
mainTable_motive:Motivo de consulta
fileTable_name:Documentos del paciente
menu_file:Archivo
menu_edit:Editar
menu_help:Ayuda
menu_I_close:Cerrar
menu_I_delete:Eliminar
menu_I_config=Configuración
langs_es=Español
langs_fr=Francés
langs_en=Inglés
menu_I_add=Añadir Paciente
add_lastName=Apellido/s
add_name=Nombre :
add_dateBirth=Fecha de nacimiento :
add_adress=Dirección :
add_ocupation=Ocupación :
add_phone=Numero de teléfono :
add_derivedBy=Derivado por... :
add_nameMother=Nombre de la Madre :
add_nameFather=Nombre de el Padre :
add_phoneFather=Numero de teléfono del Padre :
add_observations=Observaciones :
add_SetActualDate=Añadir fecha actual como primera consulta :
```

Ilustración 10 Contenido de un archivo dentro del ResourceBundle

Para saber qué idioma tenemos que cargar usamos la *Configuración regional* o *Locale* en inglés. En sí, esta configuración en los dispositivos abarca muchos más que el idioma de la interfaz ya que aquí también se especifican otros aspectos como son el formato de fecha y hora, unidades de medición o incluso la moneda utilizada. Pero en el caso de nuestra aplicación usamos esta configuración para saber qué idioma tendremos que escoger.

Por defecto, NeuroNavigator carga la aplicación en inglés ya que el Locale que viene en la plantilla del archivo de configuración de NeuroNavigator es el inglés. Para cambiarlo nos tendríamos que dirigir al menú de configuración y en el apartado de configuración general, escoger uno de los tres idiomas disponibles. Hay que subrayar que al cambiar el idioma es

necesario reiniciar la aplicación, al igual que cuando cambiamos cualquier apartado de la configuración.

Rebex Buru SFTP Server

Ilustración 11 Logo de Rebex

Rebex ofrece muchos productos, pero en nuestro caso nos interesó su servidor gratuito SFTP (para su uso no comercial). Esta herramienta nos quitó muchas horas de configuración y puesta en marcha de un servidor SFTP. Con Buru podemos configurar fácilmente varios aspectos del servidor, como son la lista Blanca y Negra del servidor (que direcciones IP se pueden conectar a él y cuales no), el rango de puertos disponibles, y la configuración fácil de los usuarios. Otra de las razones por las que utilizamos Buru en nuestro proyecto es que él mismo se encarga de la generación de las claves públicas y de los certificados del servidor, lo cual resulta ser una de las partes más tediosas de la configuración de un servidor SFTP.

Planificación y Fases del Proyecto

Para el desarrollo del proyecto, dividimos el ciclo de vida en dos puntos generales con varias subcategorías en su interior:

1. Desarrollo del FrontEnd:
    a. Desarrollo de las ventanas de la aplicación.
    b. Adaptación de la Interfaz para la Internacionalización.
2. Desarrollo del BackEnd:
    a. Desarrollo de los controladores de las vistas de la aplicación.
    b. Desarrollo de la conexión de la conexión con el servidor SFTP.
    c. Desarrollo de la conexión con la base de datos principal de la aplicación.

Desarrollo del FrontEnd

Desarrollo de las ventanas de la aplicación

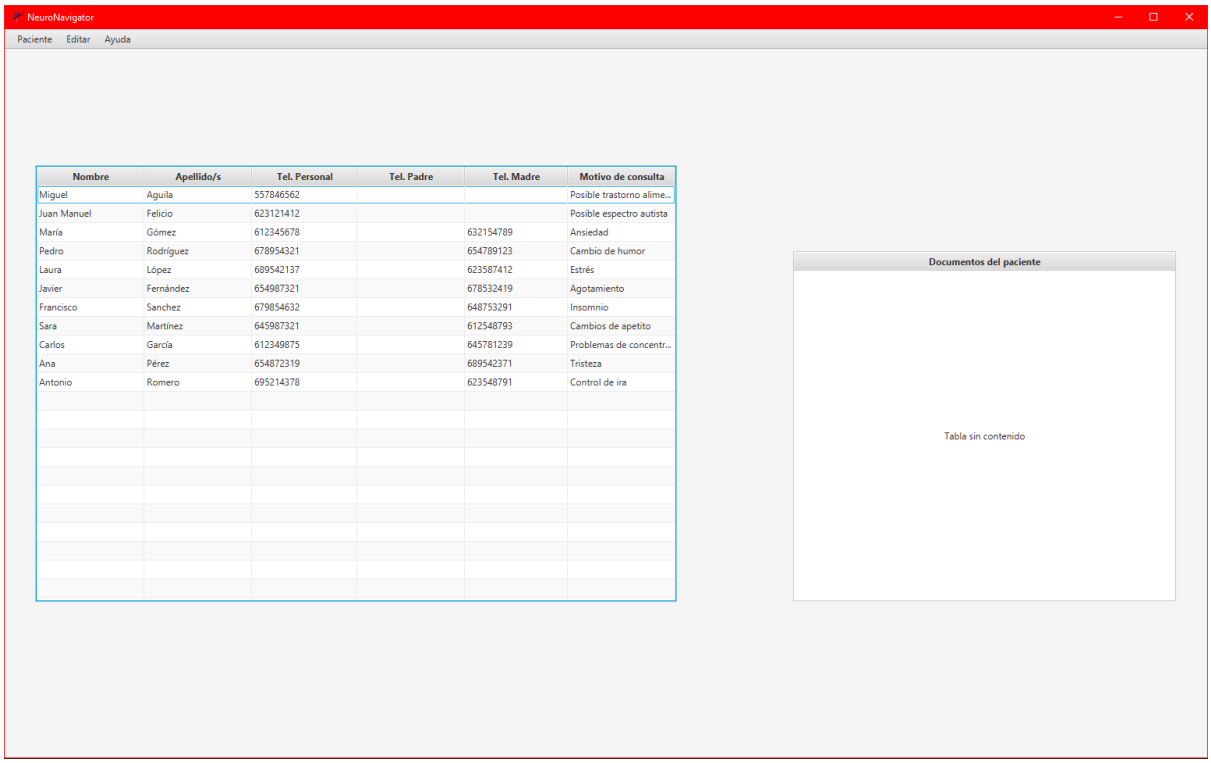


Ilustración 12 Ventana principal de NeuroNavigator

La primera etapa del desarrollo de NeuroNavigator fue el diseño de la interfaz. Para hacer una interfaz en JavaFX hay dos maneras:

- Creando la interfaz puramente con código
- Crear una interfaz usando un archivo FXML

La primera es más tediosa y puede ser más difícil de hacer, mientras que la segunda, si usamos un programa como Scene Builder, nos reduce en gran medida la carga de trabajo. Además de facilitar el diseño de la ventana, facilita otros aspectos como la implementación de la Internacionalización, al igual que el desarrollo de las funciones de los nodos de la ventana gracias al uso de los controladores que son unas clases que pose cada vista de la aplicación en

la que se encuentra programada todas las funciones de la ventana; estos dos puntos los veremos más adelante.

Desde un principio teníamos claro que la pantalla principal debía de mostrar suficiente información para poder ver e identificar a los pacientes rápidamente. Es por eso que nos decidimos por una interfaz simple compuesta de dos tablas (Ilustración 12): una para los pacientes y la otra para los archivos de los pacientes.

La tabla de los pacientes solo se actualiza si algún paciente ha sido actualizado, añadido o eliminado; mientras que la tabla de los archivos cambiará su contenido dependiendo del paciente que tengamos seleccionado.

Otro punto que suma a la facilidad de manejo de NeuroNavigator es que las funciones de la aplicación están todas en la barra superior de la ventana:

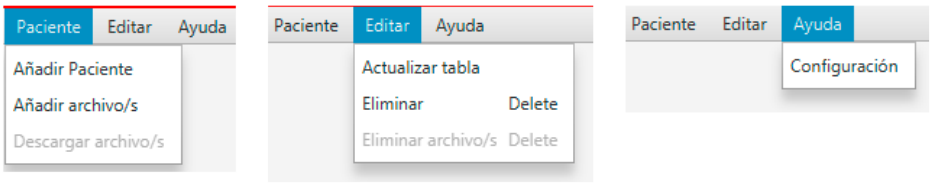


Ilustración 13 Opciones del menú superior de NeuroNavigator

Todo lo relacionado con las operaciones de añadir tanto pacientes como archivos se encuentran en el apartado de tabla. En el apartado de editar nos encontramos un botón para actualizar de manera forzada la tabla y otros dos más para eliminar pacientes o archivos. Finalmente tenemos el botón de ayuda cuya única opción es la de configuración.

Las funciones que faltan respecto a los pacientes serían las de ver el paciente en detalle, editarlo e imprimirlo. Para ello volveremos sobre la tabla de pacientes.

Como ya hemos dicho, una única pulsación sobre un único paciente listará los archivos disponibles de este; pero una doble pulsación nos abrirá la ventana de detalles del paciente (Ilustración 14)

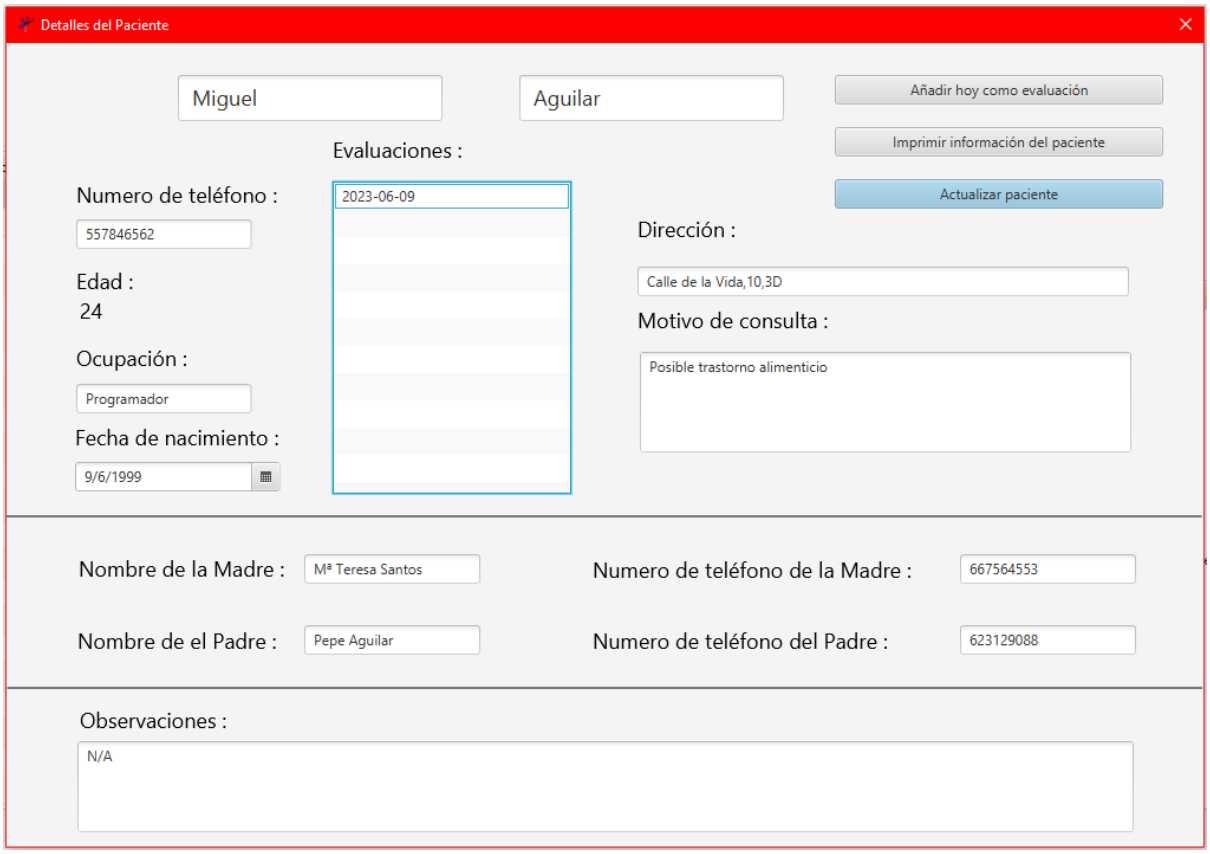


Ilustración 14 Ventana de detalle de Paciente

En esta ventana tenemos varios campos los cuales podemos editarlos todos con completa libertad. Además, los campos que sean obligatorios a la hora de rellenar el paciente, tienen un validador añadido, y no podrán estar vacíos a la hora de actualizar el paciente.

Cuando hay un error en algún formulario de la aplicación, nos saldrá un aviso en rojo el cual nos detallará cual es el problema actual con el campo:

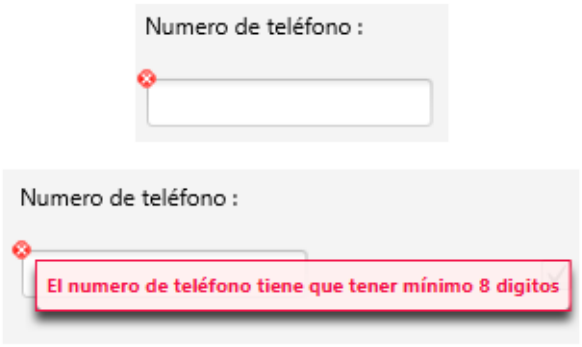


Ilustración 15 Ejemplo de campo con Validador

En cualquier formulario, la aplicación no dejará avanzar al usuario hasta que todos los campos estén *correctamente* rellenados.

Adaptación de la Interfaz para la Internacionalización

Para implementar la internacionalización en la aplicación tuvimos un problema principal; la diferencia de longitud de las frases en distintos idiomas.

La interfaz de NeuroNavigator intenta ser siempre lo más reducida posible para no causar mucha confusión; es por eso que muchas veces nos encontramos con apartados cuyos elementos están bastante recogidos entre sí (un buen ejemplo es la Ilustración 14 Ventana de detalle de Paciente, en la que podemos ver que no hay mucho espacio libre). Esto durante el desarrollo de la interfaz no supuso problema, hasta que llegó el momento de añadir otros idiomas, especialmente el francés, que comparado con el español o inglés solía ser más largo que los demás idiomas. Es por ello que tuvimos que editar casi todas las vistas de la interfaz para asegurarnos de que se veían bien en los tres idiomas disponibles.

Otra parte importante de este proceso fue asegurarse de que no hubiese fallos en la traducción. Si bien un archivo del ResourceBundle solo contiene 89 entradas (es decir 89 frases y palabras), pero multiplicándolo por 3 nos da un total de 267 cadenas de texto. Si bien usamos el traductor en alguna de ellas para acelerar el proceso, muchas de ellas fueron traducidas a mano; lo que conllevo tener que corregir pequeños errores a lo largo del desarrollo de la aplicación.

Sin embargo, a parte de las dificultades previamente mencionadas, la implementación de la internacionalización a nivel de código dentro de la aplicación fue extremadamente sencilla. Al cargar una ventana en JavaFX usando un archivo FXML, tendremos que usar un FXMLLoader, el cual nos permitirá cargar el archivo y añadírselo a una *escena* de JavaFX (una escena es el lugar que ocupa el diseño de una ventana concreta; es decir; la ventana de la aplicación en sí). Al cargar el archivo también podremos pasarle un ResourceBundle junto con un objeto Locale para indicarle el idioma que deberá utilizar. Solo faltaría añadir la clave del texto esperado en el archivo FXML para que, al cargar la pantalla, el FXMLLoader se encargue de rellenar la interfaz con los valores que corresponden.

```
locale = new Locale(properties.getProperty("lang"), properties.getProperty("country"));
resourceBundle = ResourceBundle.getBundle( baseName: "strings", locale);
FXMLLoader fxmlLoader = new FXMLLoader(MainApplication.class.getResource( name: "main-view.fxml"), resourceBundle);
```

Ilustración 16 Ejemplo de carga de un ResourceBundle con FXMLloader

Desarrollo del BackEnd

Desarrollo de los controladores de las vistas de la aplicación

Una vez terminada la fase de desarrollo de la interfaz, vino la etapa de añadirle toda la funcionalidad a cada botón, campo, y menú de la aplicación.

Lo primero fue darle un fx:id. Como ya hemos dicho, nosotros usamos los FXML junto con los Controladores, y el fx:id es un identificador que se les da a los nodos que nos interesen para poder manipularlos posteriormente en el código del controlador. Dentro del controlador estos fx:id se representan de la siguiente manera:

```
@FXML
   private MenuItem addFile;
```

El nombre del fx:id que se escribe en el FXML deberá ser el mismo que el nombre de la variable que representa el nodo dentro del controlador, añadiéndole además la anotación @FXML para que el FXMLLoader sepa que esa variable se trata correctamente de un nodo dentro de la pantalla. Gracias a esto podemos activar y desactivar nodos de la pantalla si queremos que el usuario pueda hacer clic en ellos o no. Por ejemplo, el botón de descargar un archivo no está activo si no hay conexión al SFTP o si el listado de los archivos de un paciente no devolvió ningún archivo.

También podemos usar los controladores para asignarle una acción a un botón. Y fue en este punto del desarrollo donde nos encontramos con otro problema; a lo largo de la explicación del desarrollo de la interfaz hemos mencionado varias veces la palabra *nodo*; un *nodo* representa un elemento de la interfaz. Cada elemento de la interfaz está asociado a otro elemento que se

podría considerar su *elemento padre* a través del cual podremos obtener información de la ventana en general. Esto es bastante importante a la hora de abrir una nueva ventana desde un nodo de la aplicación ya que, para que la nueva ventana forme parte de la aplicación, hace falta asignarle un dueño. El dueño será la ventana desde la cual se está mostrando la aplicación. Acceder a ella a través de un nodo no supone problema, pero al usar un ítem de un menú no se hace exactamente de la misma manera. Esto se debe a que el ítem de un menú no cuenta exactamente como un nodo dentro de la ventana; cómo podemos ver en la Ilustración 13, los ítems de los menús aparecen en una ventana desplegable, la cual es en verdad un "popup" cuya posición es relativa al botón del menú que hay que pulsar para abrirlo. Al haber encontrado este error por primera vez no sabíamos cómo proceder hasta que, tras un poco de investigación, para obtener la ventana padre del menú solo teníamos que cambiar "a quien le pedíamos la ventana". En el caso de un ítem, tendremos que preguntarle al popup quien es su padre (en este caso el botón del menú) y a este último pedirle que nos devuelva a su vez la ventana de su padre, ya que el botón del menú está encapsulado en otro componente llamado MenuBar.

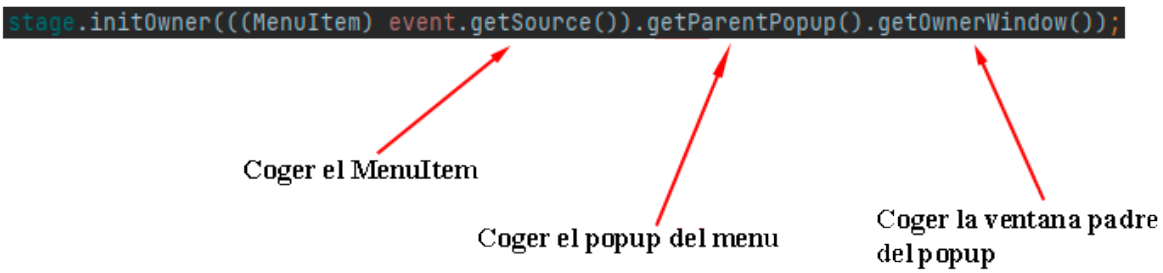


Ilustración 17 Crear una ventana a partir de un menuItem

No fue un error muy grande pero sí que fue un problema que nos detuvo por un par de horas.

Una vez superado este inconveniente no tuvimos problemas para hacer los controladores.

Respecto a la arquitectura de los controladores de la aplicación, cada uno funciona por separado salvo en dos casos:

- El controlador principal (MainController)
- El controlador para añadir un paciente (AddController)

Por un lado, el MainController comparte sus instancias tanto de PatientDAOMongoDB (la clase encargada de las operaciones con la base de datos de MongoDB) como de FTPUtils (la clase encargada de las operaciones de con el servidor SFTP); esto para no tener varias instancias de la misma clase en otros controladores y evitar tener que coger la configuración del archivo Properties para luego rellenar los constructores de la clase.

Por otro lado, el AddController tiene implementada la interfaz PacientesListener. El objetivo de esta interfaz es solucionar un problema que comentaremos más adelante, pero permite a las clases que la implementen actualizar la tabla de la vista principal.

Desarrollo de la conexión de la conexión con el servidor SFTP

Al iniciar esta etapa partimos de dos tecnologías: Apache Commons Net como herramienta para la conexión al servidor FTPS y FileZilla Server como servidor FTPS.

Lo primero fue la configuración del servidor lo cual nos dio bastantes problemas. El servidor que usamos en este trabajo no es un servidor completamente preparado para este trabajo ya que para hacer un servidor SFTP o FTPS se recomienda tener una dirección fija para poder acceder a él. Sin embargo, en este trabajo, para acceder al servidor, hay que introducir la dirección IP pública del router donde se encuentra. Esto es una mala práctica a la hora de montar el servidor por varias razones:

- Cambios en la IP pública: La mayoría de los proveedores de servicios de Internet (ISP) asignan direcciones IP públicas de forma dinámica, lo que significa que estas direcciones pueden cambiar periódicamente. Si se accede al servidor SFTP utilizando la IP pública del router, cada vez que cambie la dirección IP, será necesario actualizar todas las configuraciones y referencias a esa dirección, lo cual puede resultar incómodo y propenso a errores.
- Dependencia del router: Al utilizar la IP pública del router, se está creando una dependencia directa de dicho router. Si el router experimenta problemas de conexión, se reinicia o se reemplaza, la conexión al servidor SFTP se verá afectada. Esto puede generar interrupciones en el acceso al servidor y dificultades para la resolución de problemas.
- Limitaciones de seguridad: Al exponer directamente la IP pública del router, se está aumentando la superficie de ataque potencial para los posibles intrusos. Algunos routers pueden tener vulnerabilidades de seguridad o configuraciones incorrectas, lo que podría comprometer la seguridad del servidor SFTP.

Es por eso que al montar el servidor SFTP o FTPS recomendamos o bien contratarlo a una compañía o usar comprar una dirección IP estática para nuestro router a través de nuestro proveedor de red; aunque lo ideal sería usar un dominio de internet y asignar el DNS a nuestro servidor o usar un servicio como el que presta No-IP (DNS Dinámico) el cual reasignará a nuestro DNS nuestra IP pública en cuanto cambie.

Cabe notar que hemos hablado de SFTP y FTPS; ¿pero NeuroNavigator solo aceptaba conexiones en SFTP no? En efecto; pero durante las primeras etapas de desarrollo del servidor para el almacenamiento de los datos de los pacientes, se pensó utilizar FTPS (File Transfer Protocol over SSL/TLS) que, si bien sigue siendo un método de encriptación válido, no es el

estándar ni el más seguro comparado al SFTP. Así que, debido al carácter sensible de los archivos que se guardan en dicho servidor, nos decidimos por el protocolo SFTP.

Continuando con el tópico del desarrollo del servidor; al estar tras la IP de un router, tuvimos varios problemas con el modo pasivo del servidor.

El modo pasivo en un servidor FTPS es un modo en el cual, al conectarse un usuario al servidor, este le responde en primer lugar con otro puerto del servidor a través del cual establecer una conexión de datos. Esto es un problema ya que al estar detrás de un router hay que abrir tanto los puertos de la máquina del servidor como la del router; por lo que cuantas más direcciones queramos establecer, más puertos tendremos que abrir.

Todos estos problemas fueron los que nos dio el servidor como tal (aparte de que la documentación para montar el servidor usando FileZilla es casi nula y muy mal explicada o anticuada), pero del lado del código también tuvimos nuestros problemas. Apache Commons Net es una librería de Apache que tiene varias implementaciones de distintos protocolos como FTP/FTPS o POP3, pero tiene un gran problema a la hora de aplicarlo a lo que nosotros necesitábamos. Al conectarse con un usuario en FileZilla, este guarda la información de la sesión TLS (Transport Layer Security) para que, al volver a conectarse, use la misma. Sin embargo, la librería de Apache no tenía soporte de esta función; y pese a que tiene solución, esto implica alterar la clase del cliente FTPS de la librería.

Así que finalmente, tras múltiples problemas con el desarrollo del servidor FTPS, FileZilla y Apache Commons Net; cambiamos completamente de enfoque y nos decidimos a usar tecnologías completamente distintas:

SFTP, SSHJ y Rebex Buru SFTP Server.

Ya al empezar con estos cambios, el desarrollo del módulo SFTP en nuestra aplicación fue mucho más fluido.

En primer lugar, como comentamos en el apartado de Medios usados, el protocolo SFTP es mucho más seguro que FTPS. Lo único que parecía difícil era la generación de los archivos necesarios para el servidor (certificados, además de llaves públicas y privadas) pero al empezar a usar Rebex estos problemas desaparecieron.

Esto se debe a que Rebex fue muy fácil de instalar, además de que toda la configuración es automática y que genera los certificados y claves necesarias automáticamente. Además, la interfaz que tiene para acceder a la configuración del servidor es muy cómoda y manejada a través del localhost.

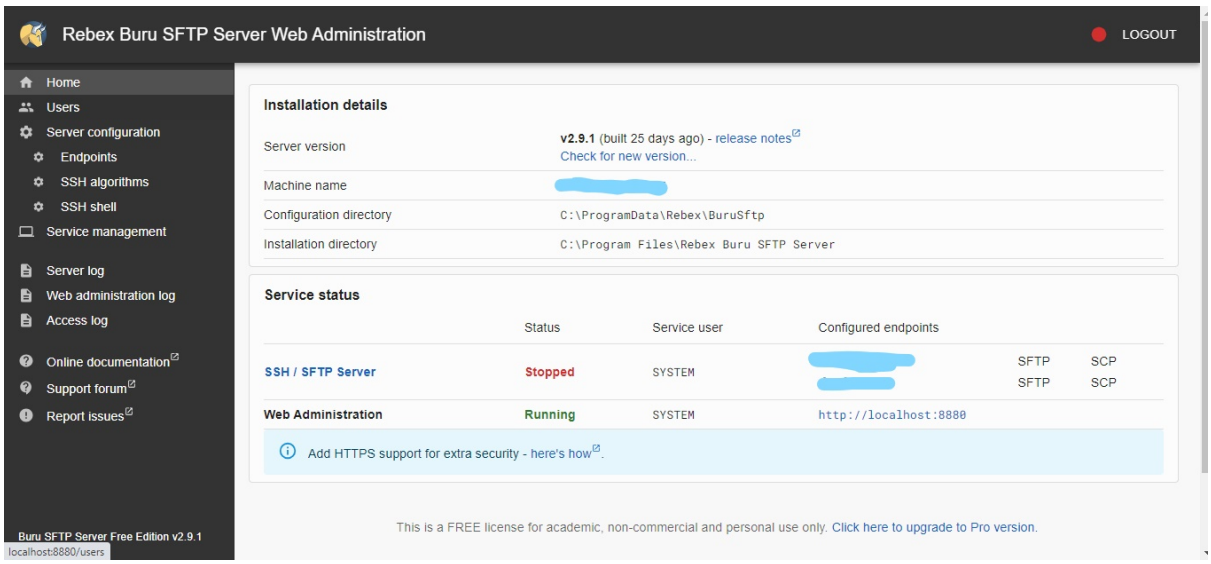


Ilustración 18 Interfaz de Rebex Buru SFTP Server

Finalmente tenemos el uso de la librería SSHJ que como comentábamos en el apartado SSHJ de Medios usados, es una librería ampliamente utilizada para la conexión desde Java con servidores SFTP. La conexión es muy simple, al igual que los métodos para operar con el servidor.

El único problema que tuvimos con el módulo SFTP en la aplicación fue los tiempos de espera para operar con el servidor. En un programa, las líneas de código se ejecutan de forma secuencial, es decir, hasta que la línea 68 no termine de ejecutarse, el programa no pasará a la línea 69; y en nuestra aplicación, como ya hemos comentado, al pulsar en un paciente en la tabla principal, se accede al servidor SFTP y se cargan los archivos de dicho paciente. Y al tener que esperar a que el listado de archivos acabase, esto bloqueaba la aplicación e incluso había casos en los que la aplicación dejaba de funcionar y se cerraba. Para solucionar esto empleamos la programación concurrente con dos tecnologías distintas:

- Tasks (proporcionadas por JavaFX)
- Threads (proporcionadas por Java)

Una Task en JavaFX es una clase que ejecuta una tarea en segundo plano independientemente de la interfaz de la aplicación. Gracias a las Tasks podemos actualizar cosas como tablas o elementos de la interfaz cuyo contenido requiera un procesamiento de datos de algún tipo, en nuestro caso, rellenar una tabla con el contenido de una carpeta en el servidor SFTP.

Por otra parte, un Thread o hilo en español, es una clase de Java que permite la ejecución de procesos en un hilo distinto al del programa desde el cual se ejecute.

Estas dos clases nos permitieron poder ejecutar las operaciones del servidor SFTP en segundo plano y así prevenir el bloqueo de la aplicación. Por un lado, hicimos uso de la Task para actualizar la tabla de archivos ya que al ser una clase específica de la librería del framework de JavaFX, este nos permitía controlar mejor el nodo de la tabla dentro de la interfaz al igual de indicar que hacer en caso de que la operación fuera exitosa o no.

En el caso de los Threads, estos se usaron en casos en los que la interfaz principal no sufría cambios, por lo que crear un nuevo objeto Thread y ejecutarlo era lo más fácil.

Desarrollo de la conexión con la base de datos principal de la aplicación

Dentro del ciclo de vida del desarrollo de NeuroNavigator, esta esta es la etapa más larga y la que seguramente más cambios sufrió durante su desarrollo.

NeuroNavigator empezó siendo pensada para poder usar bases de datos tanto de MongoDB Atlas como DynamoDB de AWS (una base de datos NoSQL al igual que Atlas). Sin embargo, rápidamente cambiamos a poder escoger entre DynamoDB y una base de datos relacional SQL haciendo uso de JPA e Hibernate para gestionar las operaciones. Esto cambió rápidamente por dos motivos:

- DynamoDB: Es verdad que era un servicio muy interesante y barato tanto para el usuario final como para el desarrollador. El único problema de DynamoDB era su configuración. Como ya hemos dicho el objetivo de NeuroNavigator es ser fácil de montar y de utilizar, y tener que montar la base de datos en DynamoDB no era tarea fácil. Cuenta con muchas capas de seguridad lo cual es muy bueno, pero esto resulta en una cantidad de tiempo innecesaria para añadir un usuario a la base de datos. Además, configurar la base de datos en sí tampoco era muy fácil ya que cuenta con funciones adicionales y un amplio apartado de configuración adicional para que el usuario final de la base de datos pueda usar una instancia de esta lo más ajustada a sus necesidades. Esta configuración sería fácil para un especialista en el entorno de AWS o para alguien que se lea detenidamente la documentación de AWS ya que, si no se tiene experiencia o conocimientos, se puede acabar con una base de datos que, si bien será funcional, no estará seguramente adaptada a las necesidades del usuario, por lo que podremos encontrar problemas como pagar demás por su servicio (esto debido a que, por lo general, AWS da la opción de pagar una cantidad fija de dinero al mes por un servicio

o bien la modalidad *pay-as-you-go* que te permite pagar exclusivamente por lo que estés utilizando que, aunque parezca que no, puede llevar a precios altos si no se configuran bien aspectos como la capacidad de procesamiento o permisos de conexión de la base de datos).

- Varios tipos de BBDD, ¿por qué?:

  Al principio nos pareció buena idea de que el usuario pueda almacenar los datos siguiendo distintas estructuras de BBDD. Pero según íbamos desarrollando esta idea nos dimos cuenta de que muy probablemente, el usuario lo que quiera será guardar sus datos rápidamente *desde la aplicación*, por lo que el método usado para almacenar los pacientes es irrelevante.

  Otras de las razones en las que se pensó para añadir el posible uso de bases de datos SQL es que serían más fáciles de montar servidores con herramientas como MySQL, MariaDB o PostgreSQL; pero en verdad daría igual ya que con MongoDB tenemos la posibilidad de configurar y montar nuestra propia base de datos ya sea local o en nuestro propio servidor, o contratar una en la nube.

Así que, después de planificación y comparación de los pros y contras de las distintas opciones nos acabamos decidiendo por MongoDB ya que, de las tres opciones, era la que mejor cubría las necesidades tanto para el desarrollo como para su uso; además de que tras leer la introducción al driver de MongoDB en java, nos percatamos que el trabajo con esta base de datos era sumamente fácil.

Al empezar a hacer pruebas con MongoDB vimos rápidamente algunas de sus muchas ventajas, como son el poder obviar las anotaciones que eran necesarias en Hibernate y DynamoDB, al igual que la facilidad para hacer las operaciones con la base de datos.

Lo bueno de hacer consultas a través del driver de MongoDB es que hay dos formas de hacerlo: con métodos preestablecidos o con consultas en formato Bson. Algunas de esas funciones son:

- find(): Si no se le pasa ningún parámetro de búsqueda, devolverá todas las entradas de la base de datos
- deleteMany() y deleteOne(): Son dos métodos encargados de eliminar entradas de la base de datos. Para saber que paciente o pacientes borrar, tendremos que pasarles un Bson o un filtro Bson como eq (igual a), gt (mayor qué), lt (menos qué) y más.
- insertOne(): A este método le tendremos que pasar como parámetro un Bson con los campos y valores de la nueva entrada que se quiera añadir a la base de datos o bien, si tenemos configurado los Codecs en la configuración del cliente de MongoDB, directamente el objeto que queramos guardar; en este caso un paciente.
- updateOne(): Para este método tendremos que pasarle un filtro para identificar el paciente que se quiere actualizar y el nuevo paciente en un documento Bson ya que, incluso con los codecs, este método espera un Bson y no un POJO. Afortunadamente, para añadir rápidamente un POJO a un Bson podemos hacerlo de la siguiente manera:

  ```
  Bson updateOperationDocument = new Document("$set",
  newPatient);
  ```

  Con el operador *$set* podremos añadir directamente el POJO al Bson sin ninguna dificultad; y ya solo faltaría ejecutar el método con el filtro y el nuevo paciente.

Una vez explicados los métodos principales para consultas dentro del driver de MongoDB, podemos explicar cómo procedimos a organizar el acceso a la base de datos en la aplicación. Como ya hemos dicho, NeuroNavigator tiene una clase llamada

PatientDAOMongoDB (Ilustración 19), que es la encargada de hacer las operaciones con la base de datos.

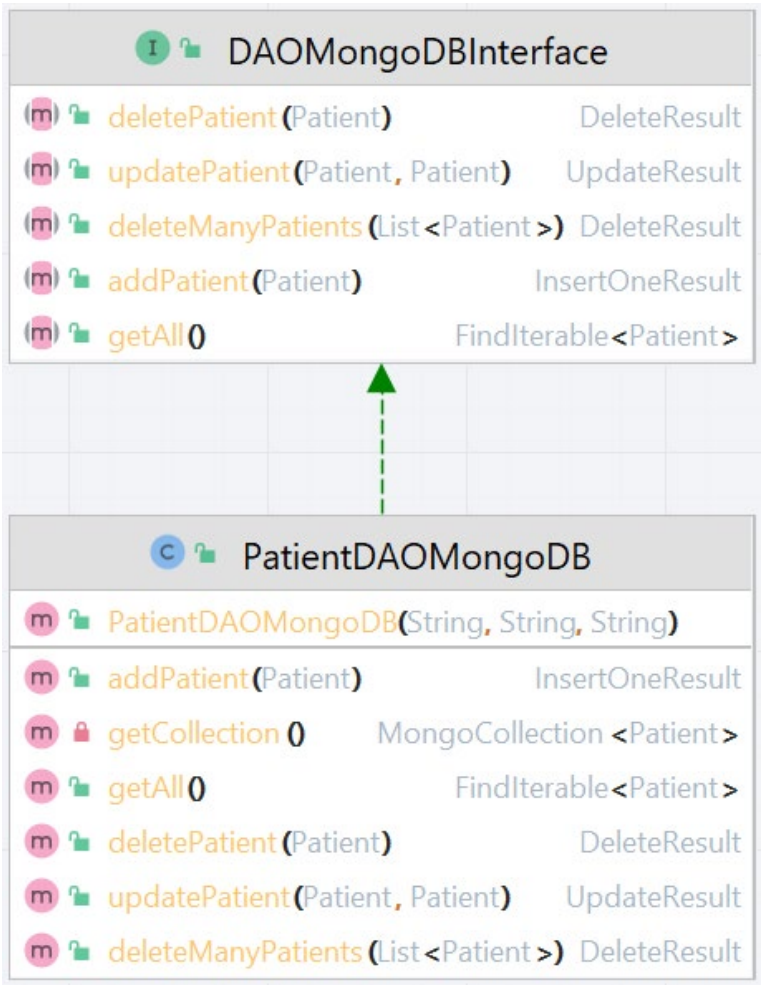


Ilustración 19 Métodos y Constructor de PatientDAOMongoDB

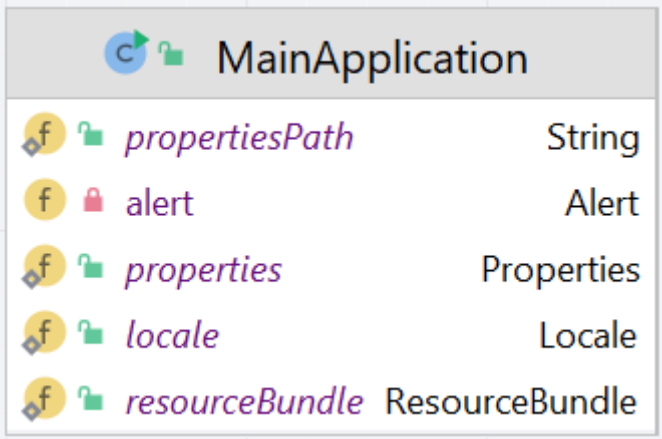


Ilustración 20 Campos públicos de MainApplication

Solo hay una instancia de esta clase en todo el programa, y esa instancia se encuentra en la clase MainController, que es la clase controlador de la ventana principal del programa la cual se usa para especificar como debe funcionar esta. Se podría haber implementado un bloque estático dentro de la clase PatientDAOMongoDB para que devolviese una instancia de la clase ya configurada. Sin embargo, nos decantamos por compartir una sola instancia proporcionada por MainController ya que esta clase obtiene el archivo de configuración de la aplicación (que es donde se encuentran todos los datos necesarios para la configuración de la conexión con la base de datos) desde MainApplication. De esta forma nos ahorramos tener que cargar múltiples veces el archivo Properties para rellenar una instancia de PatientDAOMongoDB; que, aunque sigamos compartiendo el archivo Properties como campo público de MainApplication, como podemos ver en la Ilustración 20, seguiríamos teniendo que cargar los valores del archivo Properties necesarios para la conexión con la base de datos múltiples veces.

Una vez explicado esto podemos pasar a el funcionamiento de las operaciones. Ya dijimos que se pueden hacer con Bson o con los métodos que nos ofrece el driver de MongoDB. Escogimos la segunda opción ya que con los Codecs cargado en la configuración del cliente, el driver ya

estaría transformando las ordenes que le damos a través de los métodos y POJOs a Bson. Es por esto que los métodos dentro de PatientDAOMongoDB son bastante sencillos:

```
@Override
   public FindIterable<Patient> getAll() {
      MongoCollection<Patient> collection = getCollection();
      FindIterable<Patient> result = collection.find();
      return result;
   }
@Override
   public DeleteResult deletePatient(Patient patient) {
      MongoCollection<Patient> collection = getCollection();
      return collection.deleteOne(eq("phone", patient.getPhone()));
   }
```

Además de proporcionar métodos para las consultas a la base de datos, el driver de MongoDB también nos da unos objetos e interfaces concretos para manejar los resultados de estas operaciones. El primer ejemplo de esto es el resultado que nos da el método getAll(), que en vez de ser una lista (List) de pacientes nos devuelve una interfaz; FindIterable. Esta interfaz representa un conjunto de resultados de una consulta de búsqueda en una colección de MongoDB; además, nos proporciona algunos métodos para iterar sobre ella, filtrar los resultados de la búsqueda o limitar el número de entradas que devuelve. En el caso de nuestra aplicación solo recorremos el FindIterable con un loop foreach para añadir los pacientes a la tabla principal.

Otro objeto que nos encontramos es el método deletePatient(Patient patient) es un DeleteResult; este objeto representa el resultado de una operación de eliminación de documentos en una colección de MongoDB. Gracias a él podemos ver información como cuantos campos han sido eliminados o si la base de datos ha reconocido y efectuado la operación. En este caso, sí que hacemos uso de los métodos de este objeto para que al eliminar un paciente podamos avisar al usuario de si se ha realizado correctamente la operación:

```
if (patientDAOMongoDB.deletePatient(patientHolder).getDeletedCount() == 1){
```

```
alert.setAlertType(Alert.AlertType.CONFIRMATION);
alert.setContentText(MainApplication.resourceBundle.getString("mainTable_operations_delete_success"));
alert.setTitle(MainApplication.resourceBundle.getString("mainTable_operations_tittle"));
alert.showAndWait();
}
```

En el momento de eliminar un paciente, a través del método getDeletedCount, podremos avisar al usuario de que el paciente se ha eliminado correctamente.

Hay otros objetos del mismo tipo como lo son UpdateResult o InsertOneResult, los cuales ofrecen los mismos métodos que DeleteResult, pero se usan cuando actualizamos o añadimos un paciente respectivamente.

Otra cosa a destacar del funcionamiento de PatientDAOMongoDB es como obtiene cada método la conexión a la base de datos. Ya hemos visto que para establecer una conexión con la base de datos de MongoDB; tendremos que configurar un cliente con toda la configuración necesaria, y será ese mismo cliente el que nos dará una colección concreta en una base de datos concreta según los parámetros que hayamos puesto en el constructor de PatientDAOMongoDB.

Dentro de este objeto tenemos el siguiente método privado:

```
private MongoCollection<Patient> getCollection() {
    MongoClient client = MongoClients.create(clientSettings);
    MongoDatabase mongodb = client.getDatabase(db);
    return mongodb.getCollection(colection, Patient.class);
}
```

Este método es llamado por todos los métodos públicos que hagan operaciones con la base de datos para obtener acceso a la colección. Algo que está muy bien pensado de MongoCollection es que, al ser una interfaz, le podemos especificar qué tipo de Objeto se espera que haya dentro de la colección por lo que esto además de los Codecs hace el trabajo de procesar los pacientes a la hora de operar con ellos mucho más fácil.

Finalmente, respecto a la configuración de la conexión, MongoDB solo requiere una cosa: una cadena de conexión. Esta cadena de conexión es una cadena de texto que nos da o bien Atlas a través de su página web para la gestión de nuestras bases de datos, o bien la podemos hacer nosotros si nuestra base de datos está en un servidor físico y no en la nube ya que independientemente de si la tenemos contratada como servicio o montada por nuestra cuenta, esta cadena sigue la misma estructura:

mongodb://<username>:<password>@<host>:<port>/<database>

Como podemos ver solo tenemos que reemplazar los campos de usuario, contraseña, host (la ip del servidor), el puerto, y la base de datos dentro de la cadena y podremos conectarnos directamente a la base de datos. Es verdad que no es tan fácil como ingresar usuario y contraseña en un formulario, pero por suerte, al intentar conectarnos y hay algún error en la cadena, gracias al driver de MongoDB estos errores pueden ser manejados con gran facilidad para intentar orientar al usuario hacía la solución.

En la ventana de configuración de MongoDB dentro de NeuroNavigator tenemos un botón para probar la conexión antes de guardar la configuración. Esta función puede distinguir, gracias al manejo de errores con bloques try/catch, hasta 5 errores distintos:

- La colección especificada por el usuario no se encuentra
- La base de datos especificada por el usuario no se encuentra
- Los parámetros de inicio de sesión proporcionadas por el usuario no son correctos
- El tiempo de espera de la conexión se ha superado
- Error general (problemas con la aplicación o MongoDB)

Cada error viene con un mensaje personalizado junto con que debería repasar el usuario para intentar corregirlo como podemos ver en la Ilustración 21.

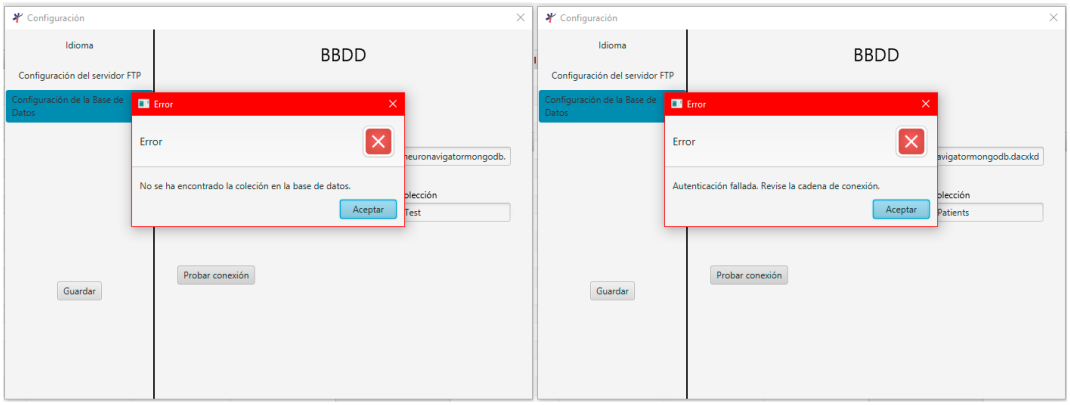


Ilustración 21 Manejo de errores en la prueba de conexión de MongoDB

Estas alertas se seguirán mostrando hasta que la conexión sea correcta, en cuyo caso el usuario podrá pulsar el botón de guardar para comenzar a utilizar su base de datos en NeuroNavigator.

Conclusión y posibles mejoras

NeuroNavigator acaba siendo una herramienta completamente funcional y lista para su uso en un entorno adecuado. Puede ser usada instalando el jdk19 o superior de Java y usar su JAR ejecutable para iniciar la aplicación.

Sin embargo, sí que se podrían añadir algunas mejoras como lo son:

- Instalador o ejecutable binario (.exe) para la aplicación: Este binario contendría tanto el JAR de la aplicación, así como un instalador de la versión de java mínima para funcionar en caso de que el usuario no la tenga instalada, ya que actualmente esa descarga hay que hacerla por separado.
- Filtro de búsqueda de pacientes: Esto sería útil para un gran volumen de pacientes ya que de por sí, NeuroNavigator nos permite ordenar las filas de la tabla de pacientes con cada columna que existe en la tabla, organizando los resultados de mayor a menor. Esto para un volumen reducido de pacientes (100-200) puede estar bien ya que se podría buscar por nombre alfabéticamente; pero cuando hablamos de volumen de pacientes más grandes, este tipo de filtrado puede ser insuficiente.
- Más idiomas: Uno de los objetivos de esta aplicación es que pueda ser usado por un usuario independientemente de su nacionalidad o localización geográfica, por lo que aumentar la lista de idiomas disponibles sería una buena idea para aumentar el número de casos de uso.
- Modelos de pacientes más extendidos: NeuroNavigator ya nos permite guardar un volumen de datos importante para nuestros pacientes, pero un paciente puede contener muchos más datos a parte de los que ya hay disponibles.

- Seguridad más avanzada: Si bien NeuroNavigator ya cuenta con algunas capas de seguridad, añadir más no sería perjudicial. Sobre todo, en la base de datos principal ya que si bien el driver soporta encriptación a través de SSL/TLS, no se pudo implementar por problemas con la generación y validación de los certificados de la base de datos, aunque implementar esta capa de seguridad en el driver no sería difícil. Otros métodos de seguridad que se podrían añadir es el uso de librerías de tercero de encriptación o nuestro propio algoritmo de encriptación, para encriptar los datos de los pacientes antes de subirlos a la base de datos por lo que estarían cifrados por nuestra parte y encriptado por SSL/TLS.

Agradecimientos y Dedicaciones

Tanto este trabajo como la aplicación van dedicadas a mi Madre, Mª Carmen Santos Marcos, fallecida el 4 de junio de 2023. Era psicóloga y pedagoga, y pese a que para el desarrollo de esta aplicación estuve investigando la organización de otros psicólogos, fue en la organización de su gabinete en la que más me inspiré y ajusté la aplicación. Así mismo, esta aplicación fue diseñada para que ella pudiese, incluso enferma, trabajar desde casa, ya que era una persona dedicada a su trabajo tanto como psicóloga como de madre hasta en sus últimos momentos. Fue ella también una de las muchas personas que me apoyó hasta el final con mis decisiones respecto a mi futuro académico, y le debo este grado, este trabajo, y esta aplicación a ella por todos los recursos que puso de mi mano y el duro trabajo que tuvo que hacer para obtener dichos recursos, para que yo lograra esto y mucho más.

Gracias Mama.

Referencias Bibliográficas

Documentación de JavaFX (2018) https://openjfx.io/javadoc/11/

Documentación de driver MongoDB para java (2014)

https://www.mongodb.com/docs/drivers/java/sync/current/#mongodb-java-driver

MongoDB (2022) Java - Mapping POJOs https://www.mongodb.com/developer/languages/java/java-mapping-pojos/

Baeldung (2023). Transferring a File Through SFTP in Java https://www.baeldung.com/java-file-sftp

JavatPoint (2012) Internationalization and Localization in Java https://www.javatpoint.com/internationalization-in-java

GitHub, jewelsea (2012) JavaFX Task example https://gist.github.com/jewelsea/2774481

GitHub, hierynomus (2023) SSHJ https://github.com/hierynomus/sshj.git

GitHub, effad (2019) ValidatorFX https://github.com/effad/ValidatorFX.git

Rebex (2020) Buru SFTP Server documentation https://www.rebex.net/doc/buru-sftp-server/

Javiergarciaescobedo (2014) Archivo de propiedades (Properties) https://javiergarciaescobedo.es/programacion-en-java/15-ficheros/358-archivo-de-propiedades-properties