

# Programacion

## Hito2

### 3trimestre



Alejandro Pozuelo Perez



# Contenido

Programacion.....	1
Hito2 3trimestre .....	1
Objetivo .....	3
Justificaciones teóricas.....	3
JavaFX .....	3
Conexión con fuente de datos en la nube (Mongo Atlas).....	4
Plataforma y entorno de desarrollo .....	5
Tecnología con versiones .....	5
Explicación del código más relevante .....	6
Operaciones crud .....	7
Demo de las funcionalidades .....	12
Capa de seguridad aplicada (cifrado, login, roles de usuarios...) .....	13
Look and Feel .....	14
Depuración .....	17
Versionado .....	18
Testing .....	18

# Objetivo

El **objetivo** principal de este proyecto es desarrollar una **aplicación de escritorio** utilizando **JavaFX** y **MongoDB** que permita gestionar **operaciones CRUD** (Crear, Leer, Actualizar, Eliminar) en una **base de datos de clientes**. La aplicación busca brindar una solución eficiente y amigable para administrar la información de clientes en diferentes contextos, como **tiendas online**.

## Justificaciones teóricas

### JavaFX

**JavaFX** es un framework de desarrollo de aplicaciones de escritorio moderno y rico en características, que ofrece una mejor experiencia de usuario en comparación con Swing, su predecesor. JavaFX proporciona una amplia gama de **componentes visuales, animaciones y efectos**, lo que permite crear **interfaces de usuario atractivas** y altamente interactivas. Además, JavaFX se integra de manera nativa con el lenguaje Java, lo que facilita su adopción por parte de los desarrolladores familiarizados con este lenguaje. **Scene Builder** es una herramienta visual que simplifica el **diseño de interfaces de usuario** en JavaFX. Con Scene Builder, los desarrolladores pueden **arrastrar y colocar componentes visuales**, configurar sus propiedades y establecer relaciones entre ellos de manera visual, lo que agiliza el proceso de desarrollo y facilita la iteración en el diseño de la interfaz.

### Utilización de Statement, Prepared Statement, Callable Statement

1. **Statement:** Es una interfaz de la API JDBC que se utiliza para ejecutar consultas SQL simples en una base de datos. Cuando utilizas un **Statement**, cada vez que ejecutas una consulta, esta se envía directamente a la base de datos para su ejecución. Sin embargo, esto puede ser ineficiente ya que la consulta se compila cada vez que se ejecuta, lo que puede afectar al rendimiento, especialmente en aplicaciones con muchas consultas repetitivas.
2. **PreparedStatement:** Esta es una extensión de **Statement** que proporciona una forma de precompilar una consulta SQL y reutilizarla varias veces con diferentes parámetros. Cuando creas un **PreparedStatement**, la consulta se compila una sola vez y se almacena en la base de datos. Luego, puedes ejecutar la consulta múltiples veces con diferentes valores de parámetros sin tener que recompilarla cada vez. Esto mejora significativamente el rendimiento y la seguridad de la aplicación, ya que ayuda a prevenir ataques de inyección SQL.
3. **CallableStatement:** Similar a un **PreparedStatement**, un **CallableStatement** también se utiliza para ejecutar consultas precompiladas en una base de datos. Sin embargo, a diferencia de un **PreparedStatement** que se utiliza para consultas SQL estándar, un **CallableStatement** se utiliza específicamente para llamar a procedimientos almacenados en la base de datos. Los procedimientos almacenados son bloques de código SQL que se almacenan en la base de datos y se pueden invocar desde una aplicación. Un **CallableStatement** permite ejecutar estos procedimientos almacenados y procesar sus resultados si es necesario.

En resumen, **Statement** se utiliza para consultas SQL simples pero puede ser menos eficiente, **PreparedStatement** se utiliza para consultas precompiladas con parámetros variables para mejorar el rendimiento y la seguridad, y **CallableStatement** se utiliza específicamente para llamar a procedimientos almacenados en la base de datos.

Aunque en lugar de **Statement**, **PreparedStatement** o **CallableStatement**, al interactuar con MongoDB, generalmente utilizas las clases proporcionadas por el controlador de MongoDB para Java, que incluyen **MongoClient**, **MongoDatabase**, **MongoCollection**, y otras. Estas clases permiten interactuar con la base de datos MongoDB utilizando operaciones específicas de MongoDB, como **insertOne**, **find**, **updateOne**, **deleteOne**, etc.

## Conexión con fuente de datos en la nube (Mongo Atlas)

MongoDB es una **base de datos NoSQL** flexible y escalable, que utiliza un **modelo de datos basado en documentos JSON-like**. Esta estructura de datos dinámica y sin esquema rígido es especialmente adecuada para aplicaciones modernas que manejan datos semiestructurados o no estructurados. Además, MongoDB ofrece **escalabilidad horizontal**, **alta disponibilidad** y **replicación de datos**, lo que la convierte en una opción atractiva para aplicaciones empresariales.

Para conseguir la conexión con mongo atlas he seguido los siguientes pasos

### 1. Configuración de la URL de conexión:

- La URL de conexión que se utiliza para conectar la aplicación con MongoDB Atlas se define en la variable **url**.
- Esta URL contiene las credenciales y la dirección del cluster de MongoDB Atlas.

```
1 usage
private final String url = "mongodb+srv://alejandro:123@alejandropozuelobdd.eikqlx.mongodb.net/";
1 usage
```

### 2. Creación del cliente de MongoDB:

- Se utiliza **MongoClients.create(url)** para crear una instancia de **MongoClient** utilizando la URL de conexión.
- **MongoClient** es la clase principal para interactuar con el servidor MongoDB.

```
1 usage
private final MongoClient mongoClient = MongoClient.create(url);
```

### 3. Obtención de la base de datos:

- A través del cliente de MongoDB, obtienes una referencia a la base de datos que deseas utilizar.

- En este caso, la base de datos se llama "**hito2**".

```
1 usage
private final MongoDBDatabase database = mongoClient.getDatabase( s: "hito2");
5 usages
```

#### 4. Obtención de la colección:

- Dentro de la base de datos, se accede a la colección específica en la que trabajaremos. En este ejemplo, la colección se llama "**clientes**".

```
5 usages
private final MongoCollection<Document> collection = database.getCollection( s: "clientes");
```

Además para que nos podamos registrar accedemos a la colección "usuarios" donde se guardaran todos los usuarios que se registren para que posteriormente inicien sesión para acceder a la aplicación.

```
2 usages
private final MongoCollection<Document> usuarios = database.getCollection( s: "usuarios");
```

En este ejemplo vemos el código que se utiliza para registrar a los usuarios

```
private void registrar() {
    String usuario = tf_usuario.getText();
    String contrasena = pf_contrasena.getText();
    Document userDoc = usuarios.find(new Document("usuario", usuario)).first();
    if (userDoc == null) {
        usuarios.insertOne(new Document("usuario", usuario).append("contrasena", contrasena));
        lbl_mensaje.setText("Usuario registrado exitosamente");
    } else {
        lbl_mensaje.setText("El usuario ya existe");
    }
}
```

## Plataforma y entorno de desarrollo

El proyecto se desarrolló en un **sistema operativo Windows 11**, utilizando el **IDE IntelliJ IDEA 2023.2.6** como entorno de desarrollo integrado. IntelliJ IDEA es una herramienta potente y completa que ofrece características avanzadas para el desarrollo de aplicaciones Java, incluido el soporte para JavaFX y la integración con herramientas de **control de versiones** como **Git**.

## Tecnología con versiones

### ○ Java SE 21.02

Es la última versión del Kit de Desarrollo de Java Standard Edition (SE). Probablemente has optado por utilizar la última versión disponible para aprovechar las características más recientes, mejoras de rendimiento y correcciones de errores que ofrece esta versión.

- **JavaFX 21.0.2**

JavaFX es una plataforma de software utilizada para crear y entregar aplicaciones de escritorio ricas en gráficos y altamente interactivas. La versión 21.0.2 es una versión estable y reciente que proporciona compatibilidad con las últimas características y correcciones de errores de JavaFX.

- **MongoDB Atlas** (versión en la nube)

MongoDB Atlas es el servicio de base de datos en la nube global de MongoDB. Lo has seleccionado porque te permite implementar, escalar y administrar fácilmente una base de datos MongoDB en la nube sin la necesidad de configurar y mantener infraestructuras de hardware.

- **MongoDB Java Driver 3.12.12**

Este es el controlador oficial de MongoDB para Java. La versión 3.12.12 es una versión estable que proporciona una interfaz de programación para conectar tu aplicación Java con MongoDB, lo que te permite realizar operaciones de base de datos como inserciones, actualizaciones, consultas, etc.

## Explicación del código más relevante

- **HelloApplication.java**: Clase principal que extiende **Application** y contiene el método **start** para iniciar la aplicación JavaFX. Esta clase maneja la carga de las **vistas FXML** y la **navegación entre ellas** mediante el método **setRoot**. Además, configura la **escena inicial** y establece el **título de la ventana**.
- **HelloController.java**: **Controlador principal** que gestiona las **operaciones CRUD** en la base de datos MongoDB. Utiliza el **patrón MVC (Modelo-Vista-Controlador)** para separar la **lógica de negocio** de la **interfaz de usuario**. Este controlador maneja la **conexión a MongoDB**, la **inicialización de la tabla de clientes**, la **gestión de eventos** (agregar, editar, eliminar y buscar clientes), y la **actualización de la vista** con los datos de la base de datos.
- **LoginController.java** y **RegistroController.java**: Controladores para las vistas de **inicio de sesión** y **registro de usuarios**, respectivamente. Interactúan con la **colección de usuarios** en MongoDB para autenticar y registrar nuevos usuarios. Utilizan **diálogos modales** y **mensajes informativos** para guiar al usuario durante el proceso.
- **Conexión a MongoDB**: Se establece una **conexión** a la base de datos **MongoDB Atlas en la nube** utilizando el **driver de Java para MongoDB**. Se utiliza una **cadena de conexión segura** y **credenciales de acceso** para garantizar la autenticación y el acceso a los datos.

- **Uso de CRUD:** Se implementan **operaciones CRUD** utilizando las funciones proporcionadas por el driver de Java para MongoDB. Se utilizan **consultas, inserciones, actualizaciones y eliminaciones** para interactuar con la base de datos de clientes.
- **Validación de datos:** Se realiza una **validación de los datos** ingresados por el usuario antes de realizar cualquier operación en la base de datos. Esto incluye verificar que los **campos obligatorios** no estén vacíos y que los **valores numéricos** (como la facturación) sean válidos.
- **Gestión de errores:** Se implementan mecanismos de **manejo de errores** para informar al usuario sobre posibles problemas, como una **conexión fallida** a la base de datos, **errores de validación de datos** o **excepciones inesperadas**. Se utilizan **cuadros de diálogo** y **mensajes descriptivos** para comunicar los errores de manera clara.

## Operaciones crud

Se implementan **operaciones CRUD** utilizando las funciones proporcionadas por el driver de Java para MongoDB. Se utilizan **consultas, inserciones, actualizaciones y eliminaciones** para interactuar con la base de datos de clientes. Estas operaciones crud en concreto editar y eliminar para poder acceder a estas he implementado que si con el **click derecho** pulsas en algún cliente te da la opción de eliminarlas o editarlas

```
ContextMenu contextMenu = new ContextMenu();
MenuItem deleteItem = new MenuItem(s: "Eliminar");
MenuItem editItem = new MenuItem(s: "Editar");

deleteItem.setOnAction(event -> eliminarSeleccionado());
editItem.setOnAction(event -> editarSeleccionado());

contextMenu.getItems().addAll(deleteItem, editItem);

tv_datos.setContextMenu(contextMenu);
```

## 1. Create (Crear)

La operación de crear se realiza cuando agregamos un nuevo documento a la colección. En tu código, esto se hace mediante el método **agregar()**.

```
private void agregar() {
    String nombre = tf_nombre.getText();
    String ciudad = tf_ciudad.getText();
    double facturacion;
    try {
        facturacion = Double.parseDouble(tf_facturacion.getText());
    } catch (NumberFormatException e) {
        welcomeText.setText("Facturación debe ser un número.");
        return;
    }
}
```

```
if (!nombre.isEmpty() && !ciudad.isEmpty()) {
    Document doc = new Document("nombre", nombre)
        .append("ciudad", ciudad)
        .append("facturacion", facturacion);
    collection.insertOne(doc);
    mostrar(); // Actualizar la lista después de agregar
    tf_nombre.clear();
    tf_ciudad.clear();
    tf_facturacion.clear();
} else {
    welcomeText.setText("Nombre y Ciudad no pueden estar vacíos.");
}
}
```

Este método:

- Obtiene los valores ingresados en los **TextFields**.

- Valida que el campo de facturación sea un número y que los campos de nombre y ciudad no estén vacíos.

- Crea un nuevo documento **Document** con los valores proporcionados.

- Inserta el documento en la colección **clientes**.

- Actualiza la vista llamando a **mostrar()** para reflejar los cambios.



## 2. Read (Leer)

La operación de leer se realiza cuando mostramos todos los documentos en la colección o cuando buscamos documentos específicos. En tu código, esto se realiza mediante los métodos **mostrar()** y **buscar()**.

```
@FXML
protected void mostrar() {
    ObservableList<Map<String, Object>> datos = FXCollections.observableArrayList();
    for (Document doc : collection.find()) {
        datos.add(doc);
    }
    tv_datos.setItems(datos);
}
```

El método **mostrar()**:

Recupera todos los documentos de la colección **clientes**.

Añade cada documento a una lista observable **ObservableList**.

Configura el **TableView** para mostrar esta lista.

```
@FXML
private void buscar() {
    String textoBusqueda = tf_buscar.getText();
    ObservableList<Map<String, Object>> datos = FXCollections.observableArrayList();
    for (Document doc : collection.find(Filters.or(
        Filters.regex(fieldName: "nombre", textoBusqueda, options: "i"),
        Filters.regex(fieldName: "ciudad", textoBusqueda, options: "i"),
        Filters.regex(fieldName: "facturacion", textoBusqueda, options: "i")
    ))) {
        datos.add(doc);
    }
    tv_datos.setItems(datos);
}
```

El método **buscar()**:

Obtiene el texto de búsqueda del **TextField**.

Filtra los documentos en la colección utilizando el texto de búsqueda en los campos **nombre**, **ciudad** y **facturacion**.

Añade los documentos filtrados a una lista observable.

Configura el **TableView** para mostrar esta lista.

### 3. Update (Actualizar)

La operación de actualizar se realiza cuando editamos un documento existente. En tu código, esto se realiza mediante los métodos **editarSeleccionado()** y **actualizarDocumento()**.

```
private void editarSeleccionado() {
    Map<String, Object> seleccionado = tv_datos.getSelectionModel().getSelectedItem();
    if (seleccionado != null) {
        ObjectId id = (ObjectId) seleccionado.get("_id");

        Dialog<Map<String, Object>> dialog = new Dialog<>();
        dialog.setTitle("Editar Cliente");

        ButtonType guardarButtonType = new ButtonType(s: "Guardar", ButtonBar.ButtonData.OK_DONE);
        dialog.getDialogPane().getButtonTypes().addAll(guardarButtonType, ButtonType.CANCEL);

        GridPane grid = new GridPane();
        grid.setHgap(10);
        grid.setVgap(10);

        TextField tfEditNombre = new TextField((String) seleccionado.get("nombre"));
        TextField tfEditCiudad = new TextField((String) seleccionado.get("ciudad"));
        TextField tfEditFacturacion = new TextField(String.valueOf(seleccionado.get("facturacion")));
```

```
        dialog.setResultConverter(dialogButton -> {
            if (dialogButton == guardarButtonType) {
                try {
                    double facturacion = Double.parseDouble(tfEditFacturacion.getText());
                    return Map.of(
                        k1: "_id", id,
                        k2: "nombre", tfEditNombre.getText(),
                        k3: "ciudad", tfEditCiudad.getText(),
                        k4: "facturacion", facturacion
                    );
                } catch (NumberFormatException e) {
                    Alert alert = new Alert(Alert.AlertType.ERROR, s: "Facturación debe ser un número.", ButtonType.OK);
                    alert.showAndWait();
                }
            }
            return null;
        });

        Optional<Map<String, Object>> result = dialog.showAndWait();
        result.ifPresent(this::actualizarDocumento);
    }
}
```

El método **editarSeleccionado()**:

- Obtiene el documento seleccionado en el **TableView**.
- Abre un diálogo para editar los campos del documento.
- Si el usuario guarda los cambios, llama a **actualizarDocumento()** con los nuevos valores.

```

1 usage new *
private void actualizarDocumento(Map<String, Object> updatedRow) {
    ObjectId id = (ObjectId) updatedRow.get("_id");
    Document updatedDoc = new Document("nombre", updatedRow.get("nombre"))
        .append("ciudad", updatedRow.get("ciudad"))
        .append("facturacion", updatedRow.get("facturacion"));
    collection.updateOne(Filters.eq(fieldName: "_id", id), new Document("$set", updatedDoc));
    mostrar(); // Actualizar la lista después de actualizar
}

```

El método **actualizarDocumento()**:

- Obtiene el **\_id** del documento a actualizar.
- Crea un nuevo documento con los valores actualizados.
- Utiliza **updateOne()** para actualizar el documento en la colección.
- Llama a **mostrar()** para actualizar la vista.

#### 4. Delete (Eliminar)

La operación de eliminar se realiza cuando eliminamos un documento seleccionado en el **TableView**. En tu código, esto se hace mediante el método **eliminarSeleccionado()**.

```

1 usage new *
private void eliminarSeleccionado() {
    Map<String, Object> seleccionado = tv_datos.getSelectionModel().getSelectedItem();
    if (seleccionado != null) {
        ObjectId id = (ObjectId) seleccionado.get("_id");
        collection.deleteOne(Filters.eq(fieldName: "_id", id));
        mostrar(); // Actualizar la lista después de eliminar
    }
}

```

Este método:

- Obtiene el documento seleccionado en el **TableView**.
- Si hay un documento seleccionado, obtiene su **\_id**.
- Utiliza **deleteOne()** para eliminar el documento de la colección.
- Llama a **mostrar()** para actualizar la vista.

# Demo de las funcionalidades

La aplicación cuenta con las siguientes **funcionalidades principales**:

- **Inicio de sesión y registro de usuarios:** Los usuarios pueden **iniciar sesión** con sus credenciales o **registrarse** si aún no tienen una cuenta. La aplicación verifica las credenciales del usuario contra la **base de datos de usuarios** en MongoDB.
- **Visualización de la lista de clientes:** La aplicación muestra una **tabla** con la **lista de clientes** almacenados en la base de datos MongoDB. La tabla muestra información clave como el **nombre**, la **ciudad** y la **facturación** de cada cliente.
- **Crear un nuevo cliente:** Los usuarios pueden **agregar un nuevo cliente** a la base de datos ingresando los datos correspondientes (**nombre**, **ciudad** y **facturación**) en los **campos de texto** proporcionados.
- **Editar un cliente existente:** Al hacer **clic derecho** sobre un cliente en la tabla, se muestra un **menú contextual** con la opción "Editar". Al seleccionar esta opción, se abre un **diálogo modal** que permite **modificar los datos** del cliente seleccionado.
- **Eliminar un cliente existente:** Mediante el **menú contextual**, los usuarios también pueden **eliminar un cliente existente** de la base de datos.
- **Búsqueda de clientes:** La aplicación cuenta con un **campo de búsqueda** que permite **filtrar** los clientes mostrados en la tabla según su **nombre**, **ciudad** o **facturación**.



# Capa de seguridad aplicada (cifrado, login, roles de usuarios...)

Utilizaron

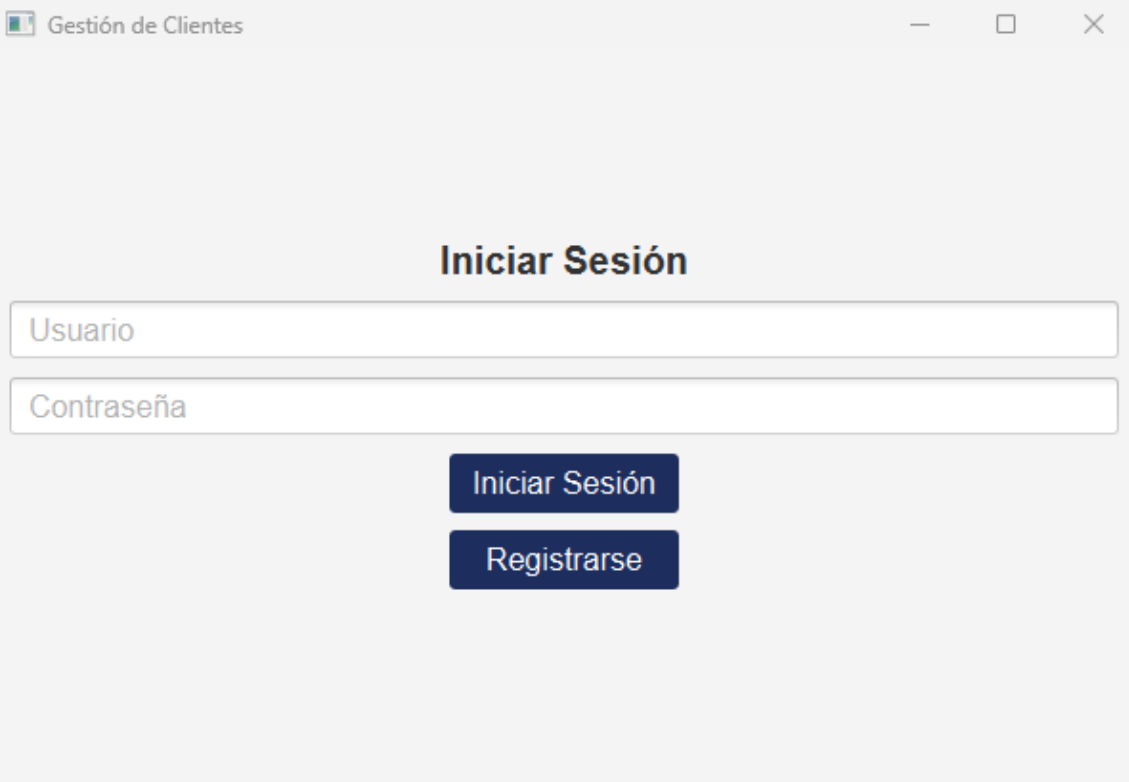
Vamos a analizar las medidas de seguridad aplicadas en este proyecto:

1. **Autenticación en MongoDB Atlas:** En el código, se utiliza una cadena de conexión que incluye el nombre de usuario (**alejandro**) y la contraseña (**123**) para autenticarse en la base de datos MongoDB Atlas. Esto asegura que solo los usuarios autorizados puedan acceder a la base de datos.
2. **Control de acceso a la base de datos:** Una vez autenticado, el proyecto interactúa con la base de datos a través de objetos **MongoClient**, **MongoDatabase** y **MongoCollection**. Estos objetos permiten ejecutar operaciones de base de datos como **insertOne**, **find**, **deleteOne**, etc. El código muestra cómo realizar operaciones CRUD (crear, leer, actualizar, eliminar) en la colección "usuarios" y "clientes". Al interactuar con la base de datos, el proyecto se asegura de que solo se realicen operaciones permitidas y autorizadas.
3. **Cifrado de contraseñas:** En la parte de registro (**RegistroController**), se recibe una contraseña y se almacena en la base de datos. Sin embargo, el código no muestra ningún proceso de cifrado de contraseñas. En aplicaciones seguras, las contraseñas deberían cifrarse antes de almacenarse en la base de datos para proteger la información sensible en caso de una brecha de seguridad. Se recomienda utilizar algoritmos de cifrado seguros como bcrypt o PBKDF2.
4. **Validación de inicio de sesión:** En la parte de inicio de sesión (**LoginController**), el código busca un documento en la colección de usuarios que coincida con el nombre de usuario y la contraseña proporcionados. Si se encuentra un documento, se considera que el inicio de sesión es exitoso. Sin embargo, el código no muestra cómo se verifica la contraseña. Es importante realizar una comparación segura de la contraseña almacenada cifrada con la contraseña proporcionada por el usuario para evitar ataques de fuerza bruta y de diccionario.

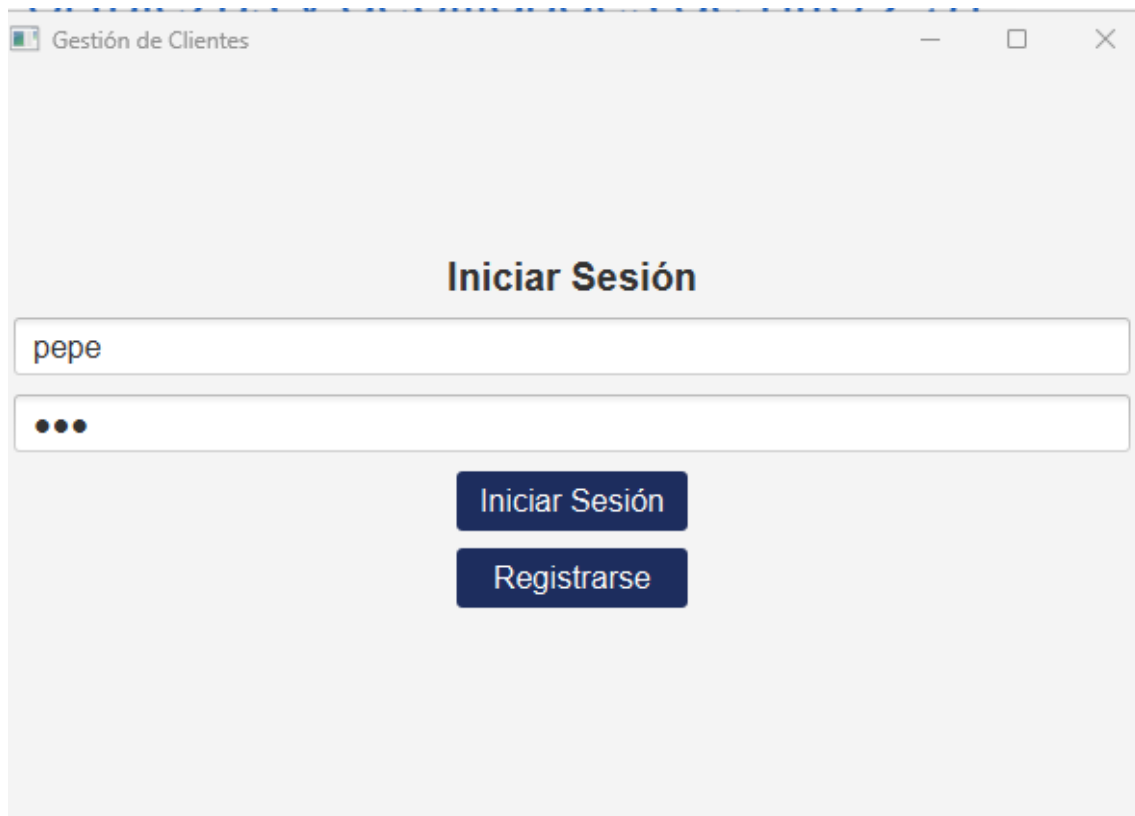
## Look and Feel

La aplicación sigue un **diseño minimalista y limpio**, con una **interfaz de usuario intuitiva y fácil de usar**. Se utilizaron **componentes JavaFX estándar**, como **TableView**, **TextField**, **Button** y **PasswordField**, para crear una experiencia de usuario coherente y familiar. Se aplicaron principios de **diseño visual**, como el uso de **espacios en blanco**, la **alineación adecuada de elementos** y la **elección cuidadosa de colores y tipografías**, para lograr una apariencia atractiva y profesional.

Primero la aplicación abre pidiéndote que inicies sesión aunque si no tienes usuario te puedes registrar

The image shows a screenshot of a JavaFX application window titled "Gestión de Clientes". The window has a light gray background and standard window controls (minimize, maximize, close) in the top right corner. In the center of the window, the text "Iniciar Sesión" is displayed in a bold, black font. Below this text, there are two text input fields. The first field is labeled "Usuario" and the second field is labeled "Contraseña". Both labels are in a light gray font and are positioned to the left of their respective input fields. Below the input fields, there are two dark blue buttons with white text. The top button is labeled "Iniciar Sesión" and the bottom button is labeled "Registrarse". The buttons are centered horizontally and have a slight shadow effect.

Al iniciar sesión no te muestra la contraseña por razones de de seguridad.



Gestión de Clientes

## Iniciar Sesión

pepe

...

Iniciar Sesión

Registrarse

Una vez entras a la aplicación se ve la tabla donde estan los clientes ademas de un buscador y la opcion de añadir mas clientes



Gestión de Clientes

Nombre	Ciudad	Facturación
alejandro	malaga	1500.0
carlos	getafe	2298.0
maria	madrid	4089.0

Nombre:

Ciudad:

Facturación:

Agregar

Buscar cliente

Buscar

Para poder eliminar o editar click derecho

The screenshot shows a web application titled "Gestión de Clientes". It features a table with three columns: "Nombre", "Ciudad", and "Facturación". The table contains three rows of data. A context menu is open over the second row, showing "Eliminar" and "Editar" options. Below the table, there are input fields for "Nombre:", "Ciudad:", and "Facturación:", followed by an "Agregar" button. At the bottom, there is a search bar labeled "Buscar cliente" and a "Buscar" button.

Nombre	Ciudad	Facturación
alejandro	malaga	1500.0
carlos	getafe	2298.0
maria	madrid	1089.0

Nombre:

Ciudad:

Facturación:

**Agregar**

Buscar cliente

**Buscar**

Para editar se abre un menu para que lo puedas modificar

The screenshot shows the same "Gestión de Clientes" application, but with an "Editar Cliente" modal dialog open. The dialog has input fields for "Nombre:", "Ciudad:", and "Facturación:", and "Guardar" and "Cancelar" buttons. The background table shows the first four rows of data. The search bar and "Buscar" button are also visible at the bottom.

Nombre	Ciudad	Facturación
alejandro	malaga	1500.0
jorge	getafe	2298.0
maria	madrid	1089.0
mario	jaen	1500.0

Nombre:

Ciudad:

Facturación:

**Agregar**

Buscar cliente

**Buscar**



```
_id: ObjectId('664dae61283be868689234b2')
nombre : "carlos"
ciudad : "getafe"
facturacion : 2298
```

---

La base de datos se actualiza automáticamente.

## Depuración

Durante el desarrollo, se utilizaron las **herramientas de depuración** integradas en **IntelliJ IDEA**, como **puntos de interrupción**, **seguimiento de variables** y **evaluación de expresiones**, para **identificar y solucionar errores**. Además, se implementaron **registros (logs)** para **rastrear el flujo de ejecución** y facilitar la **depuración**. Se realizaron **pruebas exhaustivas** en diferentes **escenarios** y **casos de uso**, verificando el **funcionamiento correcto** de las **operaciones CRUD**, la **validación de datos**, la **gestión de errores** y la **experiencia de usuario**.

**Despliegue** La aplicación se **despliega localmente** en el **entorno de desarrollo**. No se requiere un **despliegue adicional**, ya que se **conecta directamente** a la **base de datos MongoDB Atlas en la nube**. Sin embargo, si se desea **desplegar** la aplicación en un **entorno de producción**, se deberían seguir los siguientes pasos:

1. **Empaquetar** la aplicación como un **ejecutable JAR** o un **instalador**.
2. **Configurar** la **cadena de conexión** a MongoDB Atlas con las **credenciales** y **permisos** adecuados.
3. **Asegurar** la **compatibilidad** de las **versiones** de Java y las **dependencias** en el **entorno de producción**.
4. **Implementar medidas de seguridad**, como la **encriptación** de las **contraseñas** y la **protección contra ataques CSRF y XSS**.

## Versionado

Se utilizó **Git** como sistema de **control de versiones**. El proyecto se aloja en un **repositorio privado de GitHub**, y se realizaron **múltiples commits** para rastrear los cambios y mantener un **historial de desarrollo**. Se siguió un **flujo de trabajo** basado en **ramas**, donde la **rama principal** (main o master) contenía el **código estable**, mientras que las **ramas de características** se utilizaron para **desarrollar nuevas funcionalidades** o **correcciones de errores**. Una vez que una **rama de características** estaba lista, se **fusionaba (merge)** con la **rama principal** después de realizar **pruebas exhaustivas**. Cada **commit** incluía un **mensaje descriptivo** que explicaba los **cambios realizados**, lo que facilitaba el **seguimiento** del **progreso** y la **colaboración** en equipo.

## Testing

Se realizaron **pruebas unitarias** utilizando **JUnit** para verificar el **correcto funcionamiento** de los **métodos clave** de la aplicación, como la **conexión a MongoDB** y las **operaciones CRUD**. Estas pruebas unitarias ayudaron a **detectar errores** de forma temprana y a **garantizar la calidad del código**. Además, se llevaron a cabo **pruebas manuales exhaustivas** de la **interfaz de usuario** y de las **funcionalidades principales**. Esto incluyó **pruebas de aceptación**, donde se **simularon diferentes escenarios de uso** para **validar** que la aplicación cumplía con los **requisitos** establecidos. Se implementaron **pruebas de integración** para **verificar la interacción** correcta entre los **diferentes componentes** de la aplicación, como la **capa de presentación** (JavaFX), la **capa de lógica de negocio** y la **capa de acceso a datos** (MongoDB). También se realizaron **pruebas de rendimiento** y **pruebas de estrés** para **evaluar el comportamiento** de la aplicación bajo **cargas elevadas** y **condiciones extremas**, con el fin de **identificar y corregir** posibles **cuellos de botella** o **problemas de escalabilidad**.