PROGRAMACIÓN ORIENTADA A OBJETOS CON ACCESO A BBDD

IMPLEMENTACIÓN DE LA APLICACIÓN DE ESCRITORIO CON INTERFAZ GRÁFICA

MIEMBROS

Katiane Coutinho Rosa
Nadia Igual Bravo
Xavier Melo Pardo
Daniel Boj

Grupo - Cirice

Producto 5. Implementación de la aplicación de escritorio con interfaz gráfica

Enlace a GitHub

Para realizar el seguimiento de las tareas que vamos realizando, tanto a nivel individual como grupal, compartimos el enlace a nuestro repositorio de GitHub:
https://github.com/DanielBoj/UOC-POO-con-acceso-a-BBDD

Creación de la BD con docker-compose

Para facilitar la prueba de la aplicación en local, hemos creado un archivo *docker-compose* relacionado con un *Dockerfile* adjuntados a nuestra solución. De este modo, en el directorio raíz vienen incluidos los archivos necesarios para montar rápidamente la BD en un contenedor Docker y el ejecutable punto JAR ya viene listo para conectarse a esta.

Para evitar colisiones de puertos, el contenedor muestra el puerto 3307 en lugar del 3306. Por lo tanto, las variables de entorno del JAR escuchan a este puerto. Si se prefiere generar una base de datos en un servidor local, hay que cambiar el puerto en el archivo *.env*. Hay dos archivos a modificar, uno en el directorio raíz y otro en el directorio *target*.

Comando Docker – Hay que ejecutarlo en el directorio raíz de la solución, donde están situados los dos archivos de referencia:
$ docker-compose up –build -d

Refactorizando la aplicación para usar JavaFX

Para poder trasladar toda nuestra lógica de aplicación a una app de tipo *JavaFX*, hemos de realizar algunos cambios importantes en nuestro programa.

Creación de la clase module-info.java

Conforme Java fue evolucionando, se vio clara la necesidad de mejorar la implementación de la modularidad en el desarrollo. Para poder manejar de forma más controlada las dependencias entre módulos, con Java 9 se creó el archivo *module-info.java* que sirve para estructurar la configuración de los módulos en Java. Es muy importante que modularicemos todos nuestros *packages* y los pongamos a disposición unos de otros para garantizar que pueden acceder e intercambiar información entre ellos.

- **Exports**: Exporta un módulo y lo hace visible al resto.
- **Requires**: Gestiona la dependencia de módulos externos.

- **Opens**: Abre un módulo para que pueda accederse a sus clases internas.

```
module CiriceFP.OnlineStore {
    requires javafx.controls;
    requires javafx.fxml;
    requires javafx.base;
    requires javafx.web;
    requires spring.context;
    requires org.jetbrains.annotations;
    requires org.hibernate.orm.core;
    requires org.controlsfx.controls;
    requires net.synedra.validatorfx;
    requires jakarta.persistence;
    requires java.dotenv;
    requires commons.dbcp2;

    opens ciricefp.controlador to javafx.fxml;
    exports ciricefp.controlador;
    exports ciricefp.modelo;
    opens ciricefp.modelo to org.hibernate.orm.core;
    exports ciricefp.modelo.utils;
    opens ciricefp.modelo.utils to org.hibernate.orm.core;
    exports ciricefp.vista;
}
```

Implementando la nueva clase OnlineStoreApplication

Hasta ahora, teníamos toda nuestra lógica *main* en la clase *OnlineStore* desde donde ejecutábamos nuestro programa. Es necesario que traslademos la lógica de modelo MVC y de persistencia de datos a la nueva clase *OnlineStoreApplication***.** Como hemos explicado, esta clase hereda de *Application* e implementa los métodos y la lógica para crear y lanzar nuestra aplicación *JavaFX*. Además, **hemos añadido un método de cierre** para asegurarnos de que se cierra correctamente el acceso a la capa de persistencia y que la aplicación detiene todos sus procesos. Como realizábamos con nuestro *main* original, llevamos un control de posibles errores con bloques *try-catch* y control de variable de finalización.

Veremos que esta clase únicamente crea el *stage* de nuestra aplicación, pero no va a crear ninguna vista, todo esto debe manejarse independientemente en el módulo *Vista*, manteniendo así el paradigma MVC.

Para mantener la interconexión entre los módulos, es muy importante que manejemos la creación de las distintas instancias y el paso de estas a cada uno de los módulos.

Creando una clase *launcher* independiente

Para que no tengamos errores con las librerías de *JavaFX* al crear nuestro ejecutable de la aplicación, no podemos usar directamente nuestra reimplementación de la clase *main*, sino que

hemos de crear una nueva clase que no herede de *application* y que se encargue de iniciar la ejecución de nuestra aplicación llamando al método de la clase *application*. El código de esta va a ser realmente sencillo, pero es muy importante.

```
package ciricefp.controlador;

public class Launcher {
    public static void main(String[] args) {
        OnlineStoreApplication.main(args);
    }
}
```

Cambiando la clase main en el archivo *pom* y añadiendo las dependencias y plugins necesarios para crear el jar

Como hemos cambiado la clase que ejecutará nuestro programa, es muy importante que cambiemos también la referencia a la nueva clase *main* en el archivo de configuración de Maven. Así mismo, hemos de cambiar totalmente la estrategia de configuración de nuestro archivo para poder ser capaces de generar de forma correcta nuestro ejecutable.

Lo cierto es que este ha sido, seguramente, el paso más complicado de todos en este producto, ya que ha exigido bucear durante horas por la documentación oficial de Maven y sus plugin para conseguir configurar correctamente la ejecución del instalador. Para tener un punto de partida y no andar a ciegas, **generamos un proyecto nuevo mediante el CLI de Maven usando una plantilla (número 43) para crear una aplicación externa con *JavaFX***. A partir de aquí, añadimos toda nuestra lógica de la aplicación y fuimos cambiando los archivos *pom* y *module-info* para adaptarlos a nuestras necesidades.

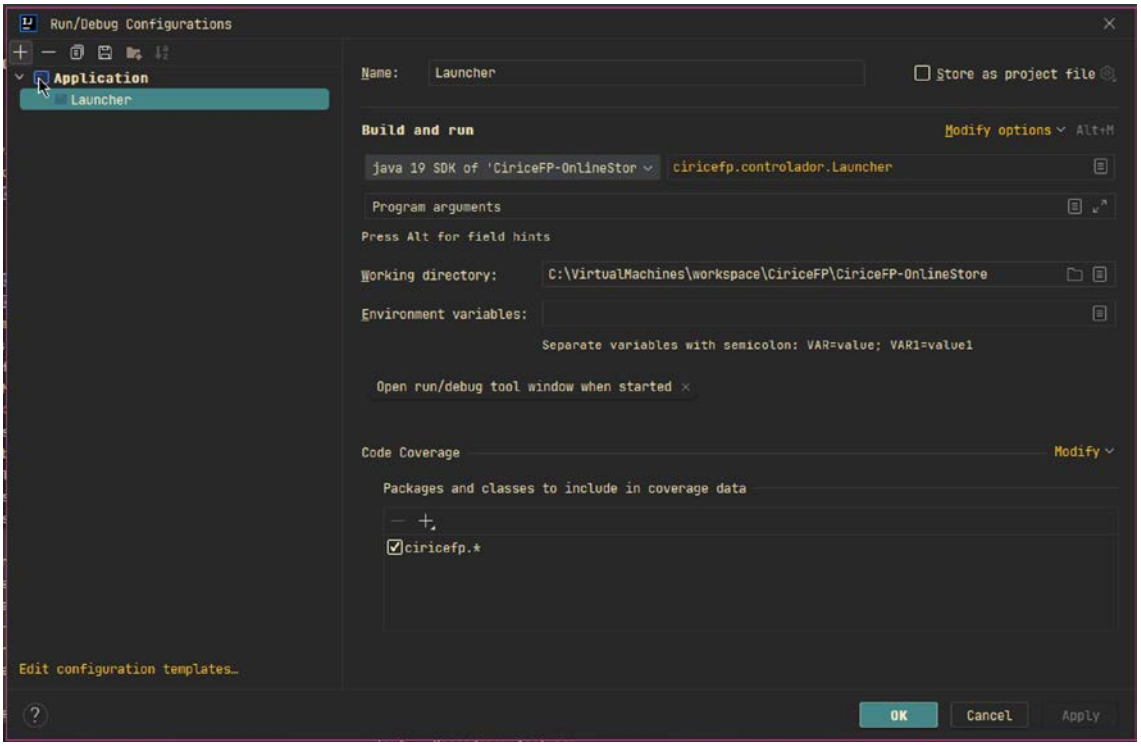
Además, cambiamos de aproximación, y en lugar de usar un *jlink*, creamos un *jar* con las dependencias necesarias, incluyendo maven-shade-plugin y *javafx-plugin*.

Target modularizado

Una de las ventajas añadidas de la nueva configuración de Maven es que a partir de ahora tendremos un programa modularizado en nuestra salida. Como vemos a lo largo de la configuración *pom*, vamos indicando a Maven que genere nuestro ejecutable basándose en los módulos de nuestra aplicación. Así, obtendremos una salida en nuestro directorio *target* muy distinta de los anteriores productos, donde todo queda mucho más claro y ordenado.

Reconfigurando IntelliJ

De igual modo, hemos de cambiar la configuración de IntelliJ para que ejecute el *run* del nuevo *main*.



Refactorizando los controladores

Eliminamos los métodos Show()

A partir de ahora, manejaremos la creación de las ventanas con nuestro controlador en el módulo *views***,** así que ya no necesitamos los métodos show() del controlador principal.

Eliminamos la implementación por consola

De la misma forma, hemos de cambiar los métodos de los controladores de vistas para que ya no muestren información por consola. Como hemos explicado, hemos dividido cada vista en un controlador y un renderizador de la vista. De este modo, la parte visual se manejará en las clases *View* y la lógica de negocio en las clases *Controller*.

Además, es muy importante que en la instanciación de los controladores de cada clase recibamos una instancia del controlador principal del módulo *Views*, *MenuPrincipalController*, y que esté haya recibido una instancia del controlador principal de la aplicación.

Adaptamos los métodos al uso de JavaFX

Hemos de adaptar el retorno e implementación de los métodos para que los podamos usar desde nuestras clases *view*. A partir de ahora, seguimos una cadena estricta de operaciones que será igual para todas las vistas. Vamos a trabajar con un modelo se *servicios*, nuestra vista solicita un recurso a su controlador, que actúa como servicio, este se pone en contacto con el controlador principal del módulo que conecta con el controlador de la App que actúa de API.

El controlador principal mantiene su configuración ya que en este producto no vamos a cambiar nada de la capa de datos.

Como hicimos en el producto anterior, usaremos *Optional* para manejar los retornos de las respuestas y llevamos los bloques *try-catch* a la clase *servicio* que serán los controladores internos de cada vista.

Igualmente, para seguir las buenas prácticas, modelaremos cada *servicio* basándolo en una interfaz.

Refactorizando las clases *view*

Por último, hemos de realizar toda la reimplementación de nuestras clases view. Como ya hemos preparado nuestros *controladores*, borramos directamente las clases y las creamos de nuevo.

Cómo estamos usando el modelo MVC, hay que implementar el módulo siguiendo el mismo patrón:

- Nuestra nueva clase principal *OnlineStoreApplication* no genera ninguna vista directamente, solo realizará las llamadas al controlador interno del módulo *vista*.
- El módulo *vista* continúa trabajando con un controlador interno que centraliza el manejo de todas las vistas de nuestra aplicación, *MenuPrincipal*.
- Cada vista individual se manejará en su propia clase encargada de *renderizarla*, estas clases siguen la nomenclatura *TipoView*.
- Cada clase *view* tiene un controlador-servicio relacionado que manejará toda la lógica de negocio de esta.
- Las clases *view* devuelven un *pane* principal que manejará nuestro controlador interno.

Además, queremos puntualizar que es muy importante que todas las clases *view* reciban una instancia del controlador interno del módulo al crearse o no funcionarán correctamente. De igual modo, nuestro controlador interno ha de recibir una instancia del controlador principal de la App para poder establecer la comunicación entre los módulos de forma correcta.

JavaFX

Introducción

JavaFX es un ***framework*** que nace como sustituto a Java Swing para desarrollar aplicaciones con GUI.

AWT fue la primera librería que se creó en Java para construir GUI, pero tenía muchos problemas y solo podía desarrollarse de forma nativa.

JavaFx incorpora tecnologías modernas para poder desarrollar aplicaciones tanto de escritorio como web, incluso pudiendo desarrollar aplicaciones híbridas. Las apps *JavaFX*, también pueden ejecutarse desde el browser e incluyen contenido multimedia.

Estructura del programa

Trabajaremos sobre un programa estructurado con Maven.

- **Clase *Application*** → Usaremos una clase especial para lanzar la aplicación *JavaFX*.
- **Dependencias Maven** → Debemos modificar nuestro archivo ***pom.xml*** para agregar los distintos módulos y plugins que necesitaremos.
- **Modularización** → Crearemos un archivo *module-info* para manejar la modularización de nuestra aplicación.
- **Refactorizaremos nuestro *package* de *views*** → Ahora vamos a trabajar generando las distintas vistas con JavaFX, cada vista tendrá un archivo principal para implementar todos los elementos que deberá renderizar y una clase controlador con los métodos para trabajar con la capa de datos.
- ***MenuPrincipal*** → Será la vista principal desde la que controlaremos el resto y el controlador interno de las vistas.

Para seguir nuestra estructura modularizada, para cada vista crearemos dos archivos, un archivo ***controlador*** en el que generaremos los métodos de la vista y funcionará como un servicio, y otro donde generaremos el ***layout*** y el renderizado. Además, para estructurar correctamente el modelo, prototiparemos nuestros controladores mediante *interfaces*.

Creación del programa

JavaFX se ejecuta de un modo un tanto distinto, normalmente no vamos a requerir de un ***main,*** este solo se usa si vamos a ejecutar el programa en un entorno de desarrollo sin soporte o con

soporte limitado de JavaFX. Es decir, este archivo se encargará de **lanzar** nuestra aplicación, por eso, normalmente, llamaremos a esta clase *Launcher*.

En nuestro caso, necesitaremos esta clase ya que vamos a generar un ejecutable Jar.

Dentro del método *main* solo tenemos que ejecutar:

```
Application.launch(args);
```

Importación de las clases de JavaFX

Es más recomendable usar una importación manual de todas las clases que vamos a usar ya que las importaciones con caracteres comodín pueden dar problemas.

Las clases principales son:

1. **javafx.application.Application**
2. **javafx.scene.Scene**
3. **javafx.stage.Stage**

Creación de la clase principal para JavaFX

Para iniciar nuestra aplicación *JavaFX* necesitamos una clase principal que se encargue de lanzar el proceso. Para ello, crearemos una clase que hereda de la clase ***Application***. Comúnmente usamos la palabra ***Application*** para el nombre de esta clase, por ejemplo, ***OnlineStoreApplication*****,** que será nuestra nueva clase *main*. En esta clase, la clase ***Aplicattion*** tiene que ser la superclase para cualquier aplicación *JavaFX* que desarrollemos.

Dentro de esta clase, crearemos el método para iniciar nuestra aplicación y el método ***main*** para ejecutarla. El método para iniciar la aplicación es un contrato con nuestra clase padre y debe llamarse ***start***, como en todos los casos que trabajamos con métodos heredados, debemos anotarlo como @***override***.

```
import javafx.application.Application;
import.javafx.scene.Scene;
import.javafx.stage.Stage;
//import.jafx.fxml.FXMLLoader;

public class OnlineStoreApplication extends Application {
        @Override
        public void start(Stage stage) throwm IOException { }
        public static void main(String[] args) { launch(args); }
}
```

El método ***start*** va a indicar a nuestro programa a qué recursos debe acceder para ejecutar la aplicación. Este toma como argumento un objeto del tipo ***Stage*** que representa la ventana principal de nuestra aplicación. Dentro del código, instanciaremos un objeto *Scene* que recibirá

los componentes que queremos mostrar. Además, tendremos que configurar el ***stage***, setear la ***scene*** y usar el método .show() para mostrar la pantalla.

Argumentos

- **PrimaryStage** → Es muy recomendable pasar ***primaryStage*** como argumento del método ***start*** para que cree automáticamente el área principal de trabajo de nuestra app.

Stage

Podemos entender el ***stage*** como el escenario donde vamos a ejecutar nuestro GUI, es decir, el área de trabajo de nuestra aplicación, cada ***stage*** es una ventana.

Scene

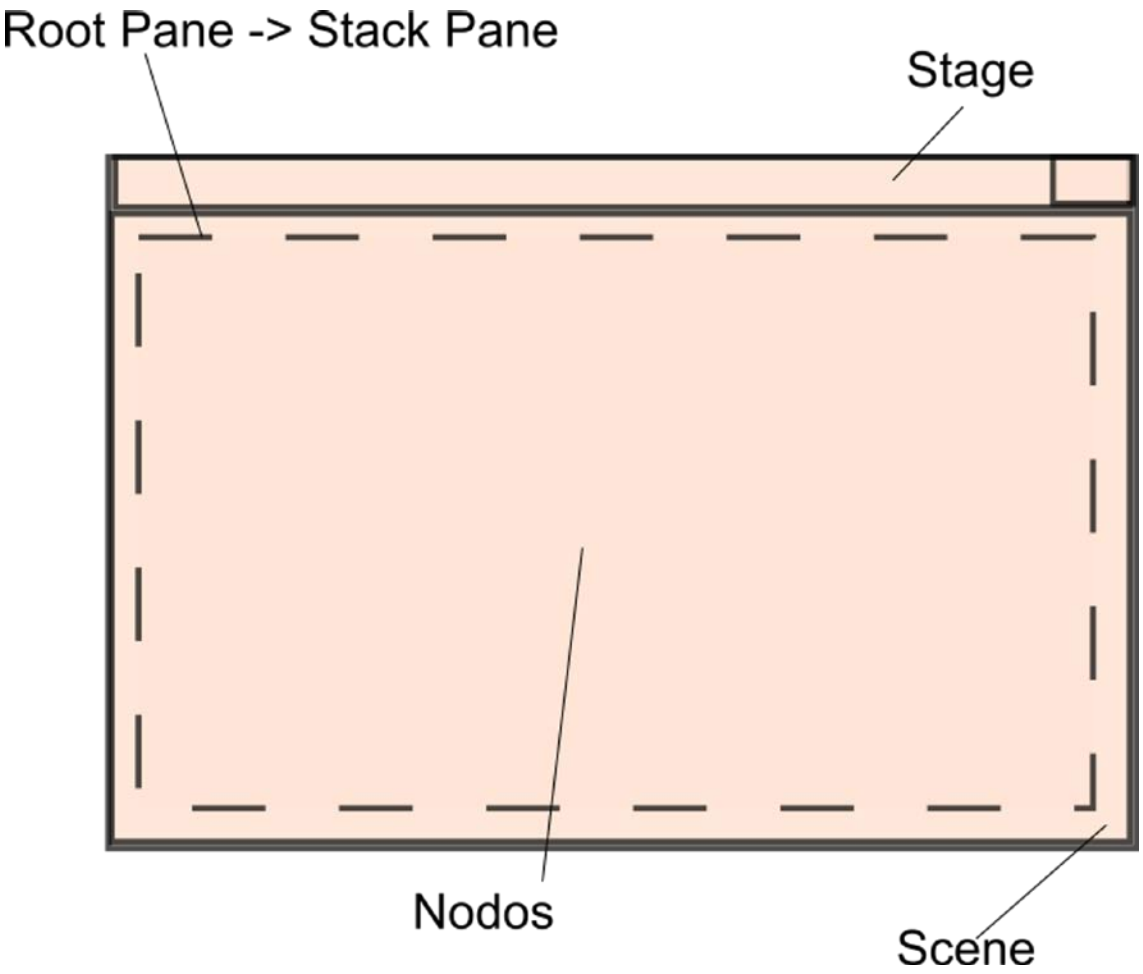
La ***scene*** manejará los elementos que queremos mostrar a nuestro usuario. A la ***scene*** le pasaremos los distintos elementos que queremos mostrar.

Nodos

Son todos los elementos que vamos a usar en nuestra GUI.

Ejemplo

```
// Método start
@Overrride
public void start(Stage primaryStage) {
        Scene scene =
stage.setTitle("Título");
stage.setScene(scene);
stage.show();
}
```



Nodos

Elementos principales

Panes

Son las clases contenedores para nuestros elementos, en sí, son un nodo que contiene otros nodos. Permiten configurar fácilmente su tamaño y posición y colocar los elementos que queramos agrupándolos donde más nos interese en nuestra ventana.

Shapes

Son los nodos que representan textos, líneas, círculos, elípses o polígonos.

UI Controllers

Son los nodos que representan botones, *checkboxes*, *radiobuttons*, *sliders*, campos de texto y áreas de texto.

Image Views

Son imágenes que queremos mostrar en nuestra ventana.

Relaciones

Las relaciones entre los diferentes nodos en *JavaFX* pueden ser algo complicadas, por ello es importante exponer cómo funciona su jerarquía:

- **Stage** → Es la ventana, el elemento ***abuelo*** en nuestro GUI.
- **Scene** → Colocamos las escenas en nuestro ***stage***, cada ***scene*** representa un ***view*** de nuestra aplicación donde mostraremos una serie de elementos.
- **Nodos** → Elementos que vamos a mostrar en nuestra escena.
  - **Pane** → Los ***pane*** son contenedores de elementos que nos sirven para agrupar elementos y configurar cómo vamos a mostrarlos en pantalla.
  - **Shapes** → Textos, líneas y formas.
  - **Controllers** → Botones, campos de texto y elementos de formulario.
  - **Image Views** → Imágenes.

Jerarquía de trabajo mediante contenedores

Para trabajar de forma ordenada y facilitar las tareas de configuración de nuestros GUI, comenzaremos por crear los ***pane*** que contendrán nuestros elementos. Normalmente, trabajamos con un contenedor raíz del que llamaremos ***root***.

```
Pane root = new StackPane();
```

En esencia, los ***pane*** van a permitirnos crear y administrar el ***layout*** de nuestra ventana.

A continuación, podemos creará los distintos nodos que queremos asignar a nuestro contenedor y, una vez los tengamos, los añadimos a este mediante el método *pane.getChildren().add(nodo)*, este método parece un poco enrevesado, pero enseguida entendemos cómo funciona.

Primero obtenemos un objeto del tipo lista de observables que contiene los nodos hijo del ***pane*****,** son hijos ya que el mismo ***pane*** es un nodo, y le añadimos los nodos que vaya a contener. Podemos ver un ***pane*** como una colección de nodos. Si queremos añadir varios elementos a nuestro ***pane***, usaremos el método ***pane.addAll()***.

Evidentemente, a nuestro ***pane*** raíz le podremos pasar otros nodos del tipo ***pane***. De esta forma, tendremos elementos totalmente modularizados. Es un poco como crear componentes de un ***frontend*** estilo React o Angular.

```
Pane root = new StackPane();
... Creamos nodos ...
root.getChildren().add(nodo);
Scene scene = new Scene(root, xsize, ysize);
```

Property Binding

Una característica importante de *JavaFX* es la capacidad de relacionar dos objetos entre sí, con ello, conseguimos que, si el objeto original cambia, también lo haga el objeto relacionado. Esto se consigue mediante la asociación de propiedades o ***property binding***, un elemento importante en ***framewroks*** de diseño de UI y ***frontend*****.**

El objeto original se conoce como ***objeto observable*** o ***vinculable***, y el objeto objetivo como ***objeto vinculante***. Mediante esta funcionalidad, podemos relacionar nuestros objetos entre sí, permitiéndonos, por ejemplo, diseñar **entornos *responsive***.

Por ejemplo, podemos mantener un nodo centrado en un ***pane*** cuando el usuario cambie el tamaño de este. Para capturar las propiedades de nuestros nodos, usamos un método que recoge el nombre de la propiedad y la palabra clave ***property*****,** a continuación tenemos que usar el método **bind()** para vincularlo al elemento observable y capturamos la propiedad del elemento que queremos usando la palabra clave ***property*** en el método. Por ejemplo, para central un círculo en un ***pane*****:**

```
circle.centerXProperty().bind(elementopadre).widthProperty().divide(2));
```

En efecto, podemos usar el ***property binding*** para vincular cualquier propiedad de nuestros objetos entre sí. Todas las propiedades vinculadas tienen, implícitamente, un método get y un método set.

Vinculación bidireccional

Por defecto, cuando vinculamos propiedades estamos realizando un vínculo unidireccional, el objeto vinculado se ve afectado por los cambios del vinculante, pero no al revés. Para que los cambios sean bidireccionales, hemos de tener en cuenta que hemos de ser muy específicos si realizamos este tipo de vinculación ya que podemos generar bucles infinitos con facilidad.

Para implementarlo, deberemos vincular explícitamente nuestros objetos con la relación observable/vinculado contraria.

Propiedades y métodos de los nodos

Propiedades y Métodos comunes

Java proporciona una serie de propiedades y métodos comunes a todos los objetos que heredan de la clase abstracta ***Node***.

Propiedades de estilo

Java sigue un patrón de estilado de elementos muy parecido al CSS que se usa en diseño web, se le conoce como **Java CSS**. Las propiedades de estilo en *JavaFX* se usan mediante la llamada al método *setStyle(),* dentro del cual, como argumentos, usaremos los estilos que queramos definir de forma parecida a cómo lo haríamos en CSS. Todas las propiedades CSS se enmarcarán en un string:

```
circle.setStyle(
        "-fx-stroke: black;" +
        "-fx-fill: red;"
)
```

A continuación, enlazamos una guía con todas las referencias de estilo para JavaFX CSS: JavaFX CSS Reference Guide

Por otro lado, JavaFX tiene métodos setter para cada tipo de Nodo que permiten definir de forma individual una propiedad del objeto, aunque, normalmente, resulta más cómodo estilar el objeto mediante ***setStyle()***:

```
circle.setStroke(Color.Black);
circle.setFill(Color.Red);
```

Además, una de las ventajas del estilado con CSS es que. si nos equivocamos en una propiedad, el programa compilará igualmente y, sencillamente, ignorará la propiedad errónea, igual que en HTML+CSS.

Cargar una hoja de estilos externa

Del mismo modo, podemos crear una hoja de estilos CSS en un archivo externo y cargarla en nuestra ***scene***:

```
scene.getStylesheets().add("org/ciricefp/ciricefponlinestoredesktop/css/style
s.css");
```

Una vez nos hayamos familiarizado con el uso de JavaFX CSS, esta es la forma más cómoda de estilar todos los nodos de nuestra app.

Clase Color

Podemos manejar el color de nuestros *nodes* de dos maneras, una de ellas es referenciando sus clases en nuestra hoja CSS; pero también podes setear el estilo mediante métodos.

- Estilar el color mediante el método ***setStyle()*.**
- Estilar el color en una hoja CSS enlazada.
- Usar el método ***set(Color.NOMBRE).***

En la implementación con método setter, se nos presenta un escollo, ya que, de salida, solo nos permiten usar los colores CSS asociados a un nombre, trabajamos con enumeradores.

Si queremos personalizar los colores, tenemos que usar la clase ***Color***. Esta clase permite utilizar notación RGB y notación hexadecimal para definir nuestros colores.

Extra → Diccionario

Si queremos facilitar una aproximación algo más funcional a nuestra asignación de colores, podemos crear un diccionario con los nombres de los colores que vayamos a usar y sus valores en una clase a parte e importarlo cuanto necesitemos.

Clase Font

Podemos asignar las fuentes para nuestros elementos de diversas manejar:

- A través del método ***Font.GetFamiles()*** obtenemos una lista con las familias de fuentes.
- A través de ***Font.getFontNames("familia")*** obtenemos los nombres de las fuentes disponibles de una familia.
- Crear un objeto ***Font***

```
// Método sencillo
Font font = new Font("Arial", 12);
// Usamos los métodos para configurarla
Font    font    =    Font.font("Times    New    Roman",    FontWeight.BOLD(),
FontPosture.ITALIC(), 12);
```

Extra → Diccionario

Del mismo que con los colores, podemos manejar las fuentes de nuestros nodos usando la clase ***Font***. Como en el caso anterior, vamos a predefinir las fuentes usando una clase estática para crear un diccionario de fuentes que podremos usar fácilmente en la configuración de nuestros elementos.

Panes

La manera más fácil de entender los ***panes*** es asimilarlos a los *divs* de HTML. Cada ***pane*** es un contenedor para nuestros nodos que, además, según su tipo, **autoconfigura el *display layout* de los elementos que va a contener**. Es decir, utilizamos los ***panes*** para diseñar el ***layout*** de nuestra escena.

Tenemos distintos tipos de ***panes*** para facilitar el proceso de diseño:

- **Stack Pane** → Suele usarse como raíz, es un panel en el que los distintos elementos se apilan uno sobre el otro, se puede usar para conseguir el efecto de capas.
- **Pane** → Es el contenedor básico, podemos colocar los nodos de forma absoluta indicando las coordenadas x e y.
- **Flow Pane** → Es un contenedor que podemos configurar para decidir cómo gestionará el *display* de los elementos contenidos, en una fila o en una columna. Es útil para crear contenidos flexibles.
- **Border Pane** → Estable una matriz de diseño dividida en cinco secciones, las cuatro direcciones y el centro. Cada sección puede contener un nodo. Tiene ajuste ***responsive***. Es muy útil para construir interfaces de usuario con una estructura de barra de herramientas + contenido central.
- **Grid Pane** → Organiza los nodos en una matriz de celdas distribuida en filas y columnas. Puede especificarse la posición de cada nodo en la cuadrícula usando las coordenadas de las celdas que queremos ocupar.
- **HBox y VBox** → Son contenedores que organizan los nodos en bloques horizontales y verticales respectivamente. Son muy útiles en diseños sencillos y lineales.
- **Tile Pane** → Organiza los nodos en una matriz con un número fijo de columnas y fijas, los nodos se irán ajustando de forma automática para adaptarse a las celdas libres.

Declaración

Como es habitual en las recomendaciones al trabajar con herencia, al instanciar un objeto, comenzamos por la clase más general y luego especificamos. Por ejemplo, para declarar una VBox, aunque en algunos casos nos interesa trabajar con la subclase desde el principio como con *BorderPane* o *GridPane*.

```
Pane pane1 = new VBox();
```

Número máximo de nodos

Cuando creamos un **pane** podemos especificar en el constructor el número máximo de nodos que podrá contener

```
Pane pane1 = new VBox(5);
```

Border Pane

Para colocar los elementos, usamos un método setter:

```
BorderPane root = new BorderPane();
// Creamos los elementos
root.setTop(elemento1);
root.setCenter(elemento2);
```

VBox y HBox

En la declaración, podemos pasar fácilmente el número de pixeles de padding que queremos establecer entre los botones.

```
Pane box = new VBox(5);
```

Grid Pane

Ofrece un gran control sobre los elementos que queremos mostrar ya que nos permite posicionarlos siguiendo una matriz de celdas. Seguimos un esquema de m, n donde m es la columna y n es la fila.

```
pane.add(elemento, m, n);
```

Podemos flexibilizar el posicionamiento ocupando varias columnas y filas. El primer par de cifras es la posición y el segundo par el **span.**

```
pane.add(elemento, 0, 2, 2, 1);
```

También podemos definir los aliniemientos de forma independiente.

```
GridPane.setHalignment(elemento, HPos.RIGHT);
```

Tenemos que definir el padding entre elementos a mano.

```
pane.setHgap(valor);
pane.setVgap(valor);
pane.setPadding(new Insets(valor));
```

Label

Los ***label*** permiten crear etiquetas para otros nodos.

Creación

El constructor por defecto toma un string con el contenido de nuestra etiqueta.

```
Label lbl = new Label("Esto es una etiqueta!");
```

Configuración

Podemos configurar nuestro objeto con diferentes características, la más importante suele ser el color y la fuente mediante los métodos **setPropiedad()**.

Images

Añade el nodo de una imagen como elemento de un ***pane***.

Creación

Image

Nos permite importar una imagen tal cuál sin modificarla, puede tratarse de un recurso interno o de una URL y crear un objeto del tipo Imagen.

```
Image img = new Image("URI");
```

ImageView

Es la clase que representa el controlador gráfico de nuestra imagen, extiende igualmente la clase *node* y es el recurso que vamos a pasar a nuestro ***pane.*** Tiene la capacidad de cargar las imágenes y modificarlas o transformarlas.

Text

El nodo *Text* permite crear bloques de texto para mostrar en pantalla. Es uno de los bloques básicos.

Creación

Usamos el constructor del nodo para definir el texto de nuestro elemento.

```
Text texto = new Text("Esto es un texo");
```

Configuración

Podemos configurar las propiedades de nuestro texto mediante setters o a través de nuestro CSS.

Eventos

Debemos enfocar el diseño de nuestro GUI como un paradigma de programación orientada a eventos. Nuestro GUI debe prever la reacción a todas las acciones posibles del usuario y, para ello, vamos a usar la captura de eventos mediante ***listeners***.

Nuestra UI va a producir distintos tipos de eventos:

- **Bajo nivel:** Por ejemplo, presionar teclas o mover el ratón.
- **Eventos semánticos:** Por ejemplo, presionar un botón o mover un ***slider***.
- **Eventos de actualización de propiedades**.

Listeners

Para poder reaccionar a los eventos, necesitamos ser conscientes de sus cambios, para ello, usamos los ***listeners***. Estos se crean mediante la interfaz genérica ***EventHandler<tipo>*** que especifica el tipo de evento que queremos escuchar. Es una interfaz de tipo funcional que permite el uso de **funciones lambda**.

Ejemplo:

```
Button button = new Button("Click me!");
button.setOnAction(event -> System.out.println("I was clicked!"));
```

Cuando queremos que un evento no se propague, por ejemplo, en la confirmación de una acción donde nuestro usuario cancela su acción previa, debemos consumir el evento:

```
event -> event.consume();
```

De este modo, cada vez que necesitemos manejar un evento, usaremos el método ***setOn*** () donde, entre los [], hay que especificar el tipo de evento y, entre (), creamos el manejador.

Para añadir un ***listener*** trabajamos pasando tres variables a la función lambda, ***observable, oldValue*** y ***newValue.*** Estos tres parámetros automatizarán el manejo de nuestro *listener*, normalmente, nos va a interesar trabajar con ***newValue***. En el siguiente ejemplo vemos cómo usamos el parámetro y cómo capturamos el valor de un ***radio button***.

```
tipoCliente.selectedToggleProperty().addListener((observable,        oldValue,
newValue) -> {
  // Comprobamos que el nuevo valor no sea nulo
  if (newValue != null) {
      // Limpiamos el contenido
```

```
        content.getChildren().clear();
        // Vaciamos la lista
        clientes.clear();
        // Obtenemos el valor del radio button seleccionado
        String tipo = ((RadioButton) newValue).getText();
        // Devolvemos los clientes filtrados

clientes.addAll(controller.filtrarClientes(tipo).orElseGet(ArrayList::new));
          ......
}
```

Propiedades

También podemos crear manejadores de eventos relacionados a propiedades de nuestros objetos.

Por ejemplo:

```
slider.valueProperty().addListener(property -> message.setFont(new
Font(family, slider.getValue())));
```

Eventos

Ratón

- mouse pressed
- mouse released
- mouse clicked
- getClickCount() → Distingue entre el número de clicks del ratón.

Ratón movimienton

- On Mouse Entered
- On Moue Exit
- On Mouse Dragged

Teclado

- is***Tecla***Down
- Solicitar ***focus*** → resquestFocus()
- General → setonKeyTyped() || setOnKeyPressed()

Diseño del layout

Opciones

En JavaFX podemos escoger tres vías para diseñár nuestros interfaces:

1. De forma programática.
2. Con programación declarativa, como en HTML.
3. Mediante un gestor visual → *JavaFX Scene Builder* que trabaja generando archivos ***fxml*** que luego importamos en nuestro programa.

Creemos que para profundizar más activamente en los conocimientos de *JavaFX*, lo mejor es no usar de salida el método visual porque necesitamos entender los conceptos antes de automatizarlos.

Diseño programático

Nos basamos en el uso de los ***panes*** como contenedores de nuestros elementos, cada uno de los distintos ***pane*** tiene unas reglas concretas para manejar los nodos que contiene. En nuestro caso hemos optado por realizar un diseño programático ya que la opción mediante *Scene Builder* es la recomendada para diseñadores de UI y no para programadores.

Inclusión de CSS

Podemos crear un archivo de estilos JavaFX CSS en el directorio ***resources*** y cargarlo en nuestras escenas mediante el siguiente código:

```
scene.getStylesheets().add("file:src/main/resources/styles.css");
```

Personalización de clases

Igual que en HTML, podemos crear selectores de clase para nuestros elementos:

```
elemento.getStyleClass().add("nombre-clase");
```

Selectores genéricos

También tenemos selectores genéricos como:

- root
- label
- button
- text-field

Propiedades programáticas con diccionarios

Hay casos, como para los colores de algunos elementos y las fuentes, en los que, por comodidad y para evitarnos repetir código o posibles errores de configuración, hemos usado los valores de los diccionarios que hemos creado con las fuentes y los colores y hemos cargado las propiedades del elemento mediante configuración programática.

Formularios

Podemos construir formularios usando distintos tipos de **inputs**.

TextInputControls

Tenemos tres subclases básicas:

- **TextField** → Campo de texto simple.
- **TextArea** → Área de texto a la que podemos predefinir el tamaño.
- **PasswordField** → Oculta los caracteres que introduce el usuario.

Métodos

- **getText** → Captura el texto introducido.
- **setText** → Envía un texto.
- **appendText** → Añade texto al campo.
- **setEditable** → Indica si el control es editable.
- **setPrefColumnCount** → Ancho aproximado del campo.
- **setpromptText(”texto”)** → Muestra un texto por defecto.

Tamaño del área de texto

- **setPrefRowCount(valor)** → Alto del área de texto.

Label

Para mostrar el nombre del campo, usamos nodos del tipo ***label***.

CheckBox, RadioButton y ComboBox

Sirven para dar opciones de selección a los usuarios.

- **CheckBoxes** → Permiten seleccionar más de una opción.
- **RadioButtons** → Permiten seleccionar una única opción.
- **ComboBox** → Es una caja para seleccionar un elemento.

Los selectores de tipo *check* y *radiobutton* usando los métodos ***isSelected*** y ***setSelected***.

Para *setear* correctamente el funcionamiento de los ***radiobuttons*** los agrupamos en un elemento ***ToggleGroup***.

ComboBox

Es una clase de tipo genérico a la que le tenemos que pasar el tipo de dato que vamos a cargar.

Además de mostrar una serie de opciones, podemos convertirla en editable para que el usuario pueda escribir su propia opción.

Usa los métodos ***getValue*** y ***setValue***.

Dialogs

En JavaFX podemos crear ventanas de diálogo para interactuar con los usuarios y manderlas mensajes reactivos.

Creación

```
Alert alerta = new Alert(AlertType.***TIPO***, "Mensaje");
// Podemos setear el mensaje tras la creación
alerta.setContentText("Mensaje");
// Lanzamos el mensaje a la espera de la interacción del usuario
alerta.showAndWait();
```

Tipos

- **INFORMATION** → Ventana de información con un botón para aceptar.
- **WARNING** → Ventana con un mensaje de alerta y un botón para aceptar.
- **ERROR** → Ventana con un mensaje de error y un botón para aceptar.
- **CONFIRMATION** → Ventana que muestra dos botones y devuelve un
- **OPTIONAL** → Tenemos que manejar el retorno.
    - Podemos añadir botones extra usando el método *ButtonType.NEXT* | *PREVIOUS* | *FINISH.*

Validación de campos

Uso de observables

Para validar los campos de los formularios, podemos usar *listeners* que comprobarán los valores que vayamos introduciendo en tiempo real a través de un observables.

```
txtNombre.textProperty().addListener((observable, oldValue, newValue) ->{});
```

Para las **validaciones de patrones** usaremos estas aproximaciones:

Patrones *regex*

Para casos como el NIF o el nombre de un cliente, podemos usar la validación de expresiones regulares mediante la clase ***Matches***:

```
txtNombre.textProperty().addListener((observable, oldValue, newValue) -> {
    // Permitimos letras mayúsculas y minúsculas y espacios
    if (!newValue.matches("[A-Za-z ]*")) {
        txtNombre.setStyle("-fx-border-color: red");
    } else {
        txtNombre.setStyle("-fx-border-color: green");
        // Tenemos que cambiar el valor de validators, no del elemento aislado
para que el listener funcione
        validators.setValue(new   Boolean[]{validators.getValue()[0],   true,
validators.getValue()[2], validators.getValue()[3]});
    }
});
```

Validación de campos numéricos

Como los inputs de *JavaFX* se realizan mediante strings, deberemos intentar *parsear* el input para comprobar que sea un valor numérico. Otros lenguajes ofrecen la posibilidad de usar el NAN.

```
txtGastosEnvio.textProperty().addListener((observable, oldValue, newValue) ->
{
    try {
        Double.parseDouble(newValue);
        lblGastosEnvio.setStyle("-fx-border-color: green;");
        // Tenemos que cambiar el valor de validators, no del elemento aislado
para que el listener funcione
        validators.setValue(new   Boolean[]{validators.getValue()[0],   true,
validators.getValue()[2], validators.getValue()[3]});
    } catch (NumberFormatException e) {
        lblGastosEnvio.setStyle("-fx-border-color: red;");
    }
});
```

Validación de Email

La validación de emails mediante ***regex*** no es la aproximación más recomendable; por suerte, contamos con librerías externas como *Apache Commons Validator* que tienen herramientas de validación avanzadas. Para poderla usar, como tenemos un proyecto modularizado y configurado con Maven, hemos de importar la dependencia en Maven y también añadir la librería a nuestro archivo ***module-info:***

```
<dependency>
  <groupId>commons-validator</groupId>
  <artifactId>commons-validator</artifactId>
  <version>1.7</version>
</dependency>
// Validación de email
import org.apache.commons.validator.routines.EmailValidator;
...
txtEmail.textProperty().addListener((observable, oldValue, newValue) -> {
    // Usamos Apache Commons para validar el email
    if (!EmailValidator.getInstance().isValid(newValue)) {
        txtEmail.setStyle("-fx-border-color: red");
    } else {
        txtEmail.setStyle("-fx-border-color: green");
        // Tenemos que cambiar el valor de validators, no del elemento aislado
para que el listener funcione
        validators.setValue(new   Boolean[]{validators.getValue()[0],   true,
validators.getValue()[2], validators.getValue()[3]});
    }
});
```

Validación del formulario

Para aplicar la validación de campos, hemos controlado los distintos validadores independientes dentro de un Array de booleanos, además, hemos convertido el array en un observable, mediante la clase *wrapper* ***Property***. Manejamos cada posición del array dentro de la lógica de los validadores y, finalmente, creamos un *listener* asociado con el método ***setDisable*** del botón para activar la acción del formulario.

```
Property<Boolean[]>    validators    =    new    SimpleObjectProperty<>(new
Boolean[]{false, false, false, false});
...
// Por defecto, el botón de añadir estará desactivado
btnAnadir.setDisable(true);
//System.out.println(Arrays.stream(validators.getValue()).findAny().map(value
-> !value).orElse(false));
// Si las validaciones son correctas, el botón de añadir se activará
validators.addListener(
        (observable,            oldValue,            newValue)            ->
btnAnadir.setDisable(!Arrays.stream(validators.getValue())
                            .allMatch(value -> value))
);
```

Análisis del producto

Elementos de mayor dificultad o que hayan exigido mayor dedicación

Curiosamente, la parte más complicada de este producto ha sido adaptar el archivo *pom* para generar correctamente nuestro ejecutable y poder presentar una aplicación de escritorio real. Como comentamos en el documento, partimos de una plantilla Maven y, a partir de ahí, pasamos

muchas horas investigando la documentación oficial para entender cómo crear correctamente nuestro Jar.
Por otro lado, en cuanto a la implementación del código, hemos podido ir integrando los nuevos conocimientos sobre JavaFX con los distintos modelos y paradigmas de diseño que habíamos aprendido en los productos anteriores, quizás la parte más complicada ha tenido que ver más con esto que con el manejo de JavaFX en sí, ya que la mayoría de cursos y guías disponibles se quedan únicamente en mostrar cómo realizar todo el manejo desde la clase *main*, pero para nosotros era muy importante mantener correctamente el diseño modular de nuestra aplicación y hacerlo de la forma más adecuada posible, es decir, con controladores e interfaces, tal y cómo hemos aprendido en los productos anteriores.

Puntos del enunciado no cumplidos

Por suerte, hemos podido cumplir con todos los puntos de esta actividad e implementar todos los requisitos esperados. El haber preparado un flujo de trabajo controlado, accediendo a unos recursos muy buenos y siguiendo un orden de implementación para la producción establecido para facilitar el proceso, nos ha permitido poder acabar el producto a tiempo y realizar una entrega de todos los apartados y funcionalidades. También hemos podido revisar y prepara las funcionalidades opcionales y revisar la deuda técnica heredada de nuestros productos anteriores, así, por ejemplo, hemos implementado ya la validación de campos y hemos creado un *target* modularizado.

Mejoras y refactorizaciones

Los aspectos en los que nos gustaría trabajar y que quedan pendientes:

- Normalización y mejora del archivo de estilado: Pensamos que sería interesante intentar trasladar parte de la lógica de estilado que aún se realiza programáticamente a nuestro archivo JavaFX CSS.
- Uso de herencia múltiple de *interfaces*: Una de las prácticas recomendadas para asegurar que nuestro código es lo más limpio, reusable y mantenible posible es no amalgamar todas las acciones en una única interfaz si no usar la herencia múltiple de interfaces. Así, creemos que un elemento a mejorar sería modelar más correctamente nuestra interfaz *Repositorio*. Para ello, llevaríamos las acciones CRUD a una interfaz independiente, las comparaciones para el filtrado a otra y las acciones de conteo a otra. Podemos, entonces, refactorizar nuestra interfaz *Repositorio* con un *extends* en el que añadimos las diferentes interfaces que implementa y así tenemos una *interface* con herencia múltiple.
- Implementación de un *pool* de conexiones mediante un servicio externo.

Autoevaluación

Para la autoevaluación de este producto nos gustaría tener en cuenta algunos elementos de valor en nuestro proyecto:

- Hemos mantenido los patrones de distintos patrones de diseño que hemos ido trabajando a lo largo de todos los productos: Singleton, Factory, MVC, modularización…

- Hemos añadido el manejo avanzado de la modularización en Java y lo hemos externalizado en la creación de nuestro *target*.
- Hemos incluido el manejo de servicios e interfaces en la implementación del módulo *vista*.
- Hemos manejado estructuras de datos avanzadas con *HashMaps*.
- Hemos usado funciones lambda y programación funcional siempre que nos ha sido posible.
- Hemos realizado una implementación programática de la renderización de nuestras vistas en lugar de optar por usar software de generación de archivos fxml para diseñadores de UI.
- Hemos usado la mayor parte de los nodos disponibles en JavaFX y todas sus funcionalidades básicas.
- Hemos manejado el acceso a recursos como imágenes y archivos css tanto en nuestro IDE como en la aplicación de escritorio.
- Hemos manejado un documento general de estilado JavaFX CSS.
- Se ha realizado el manejo de excepciones en el nuevo módulo *vista*.
- Se ha reimplementado la clase *main* manteniendo todas las funciones extra que habíamos creado y el seguimiento del valor de salida de nuestro programa.
- Uso de variables de entorno para no revelar datos sensibles.
- Uso de *Optional* para manejar el retorno de nuestros servicios.
- Modularización del código atendiendo a la reusabilidad, legibilidad y escalabilidad.
- Finalmente, hemos conseguido configurar Maven para generar una aplicación de escritorio funcional con todas las dependencias integradas, modularizada y capaz de conectarse a la base de datos.

Teniendo en cuenta todos estos puntos y que hemos intentado presentar un código optimizado y diseñado teniendo en mente un escenario real; pero también que nos hubiera gustado acabar de trabajar en el archivo CSS e implementar el pool de conexiones, nuestra autoevaluación sería de un 95.

Programa ejecutable

Dentro del directorio raíz del proyecto, temenos que acceder a la carpeta *target* y podremos encontrar el ejecutable del programa:
CiriceFP-OnlineStore-jar-with-dependencies.jar

Recursos

JavaFX CSS Reference Guide. (s. f.). https://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html

Tutorial gratuito sobre JavaFX - Crash Course Into JavaFX: The Best Way to make GUI Apps. (n.d.). Udemy. https://www.udemy.com/course/crash-course-into-javafx-the-best-way-to-make-gui-apps/

Máster Completo en Java de cero a experto 2023 (+127 hrs). (2023, May 24). Udemy. https://www.udemy.com/course/master-completo-java-de-cero-a-experto/

Team, A. C. D. (s. f.). *Apache Commons – Apache Commons*. https://commons.apache.org/

Maven Repository: org.apache.maven.plugins. (n.d.). https://mvnrepository.com/artifact/org.apache.maven.plugins

Not loading css file in executable jar javafx8. (s. f.). Stack Overflow. https://stackoverflow.com/questions/26833296/not-loading-css-file-in-executable-jar-javafx8

How to fix «Error: JavaFX runtime components are missing, and are required to run this application»? (s. f.). Stack Overflow. https://stackoverflow.com/questions/56894627/how-to-fix-error-javafx-runtime-components-are-missing-and-are-required-to-ru

Openjfx. (s. f.). *GitHub - openjfx/javafx-maven-plugin: Maven plugin to run JavaFX 11+ applications*. GitHub. https://github.com/openjfx/javafx-maven-plugin

Talevi, M. (n.d.). *Apache Maven Shade Plugin – Selecting Contents for Uber JAR*. https://maven.apache.org/plugins/maven-shade-plugin/examples/includes-excludes.html

Talevi, M. (n.d.). *Apache Maven Shade Plugin – Executable JAR*. https://maven.apache.org/plugins/maven-shade-plugin/examples/executable-jar.html

Moditect. (2023, 3 mayo). *moditect/README.md at main · moditect/moditect*. GitHub. https://github.com/moditect/moditect/blob/main/README.md

Maven Repository: org.springframework » spring-context » 6.0.9. (n.d.). https://mvnrepository.com/artifact/org.springframework/spring-context/6.0.9

Maven plugins can not be found in IntelliJ. (s. f.). Stack Overflow. https://stackoverflow.com/questions/20496239/maven-plugins-can-not-be-found-in-intellij