ACCESO A DATOS

Proyecto final de Acceso a Datos
2º DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Utilidad financiera para PYMES con JavaFX, Spring Boot y ThymeLeaf

Agustín Rodríguez Márquez *IES Rafael Alberti*

INDICE

1. Introducción……3

2. Tecnologías usadas……3

3. Arquitectura del proyecto……6

4. Documentación del proyecto……7

5. Vistas del programa y manejo del mismo……37

6. Conclusiones……42

1. INTRODUCCIÓN

Este es mi proyecto final de la asignatura de Acceso a Datos para el ciclo de Técnico Superior de Desarrollo de Aplicaciones Multiplataformas del IES Rafael Alberti, en el curso 2023, 2024.

Durante este texto desarrollaré la idea sobre la que se basa mi proyecto, las tecnologías usadas, desgranaré y documentaré el código paso a paso, para continuar con un repaso simple al funcionamiento básico del programa, y terminar con mis conclusiones sobre el mismo.

2. TECNOLOGÍAS USADAS

Mi proyecto está basado en algunas de las diferentes tecnologías usadas a lo largo del curso 2023/2024 en la asignatura de Acceso a Datos.

La aplicación se basa en una utilidad para pequeñas empresas que desean guardar sus registros de gastos e ingresos en una base de datos de forma simple y organizada.

Para el desarrollo de esta aplicación he usado las siguientes tecnologías:

IntelliJ: IntelliJ es el entorno de desarrollo que he elegido para desarrollar este proyecto.


Kotlin: Es el lenguaje de programación en el que está escrito el código del proyecto.

JavaFX: JavaFX es la plataforma que he usado para implementar una interfaz a mi aplicación.


Scene-Builder: La aplicación con la que he podido construir las vistas o UI de mi aplicación en JavaFX de forma sencilla.


Hibernate: Un framework de mapeo ORM para Java. Hibernate ofrece un lenguaje de consultas HQL, similar a SQL, pero usando objetos Java, en lugar de tablas y columnas de base de datos.

Spring Boot: Spring Boot es una extensión de Spring para plataformas Java, y por extensión Kotlin, que permite con una configuración mínima, crear aplicaciones autónomas de forma sencilla y rápida, eliminando por el camino gran parte de la configuración manual que sería necesario. En mi aplicación se usa junto a Hibernate como el proveedor de JPA.

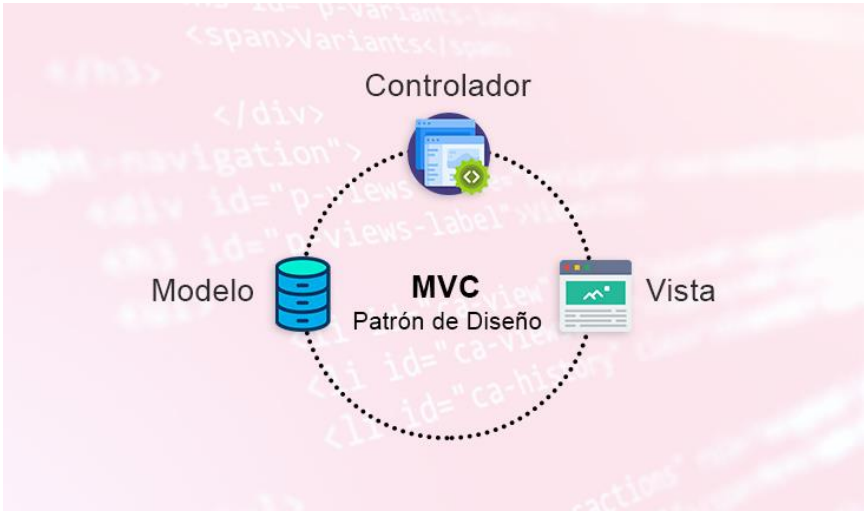
MySQL: El sistema de gestión de base de datos relacionales que he decidido usar para este proyecto. MySQL es realmente confiable y fácil de usar, especialmente combinado con Spring Boot.

ThymeLeaf: Un motor de plantillas para Java que se utiliza en el desarrollo web para desarrollar vistas HTML. Es usado en mi proyecto para generar webs HTML con los registros financieros de las empresas.

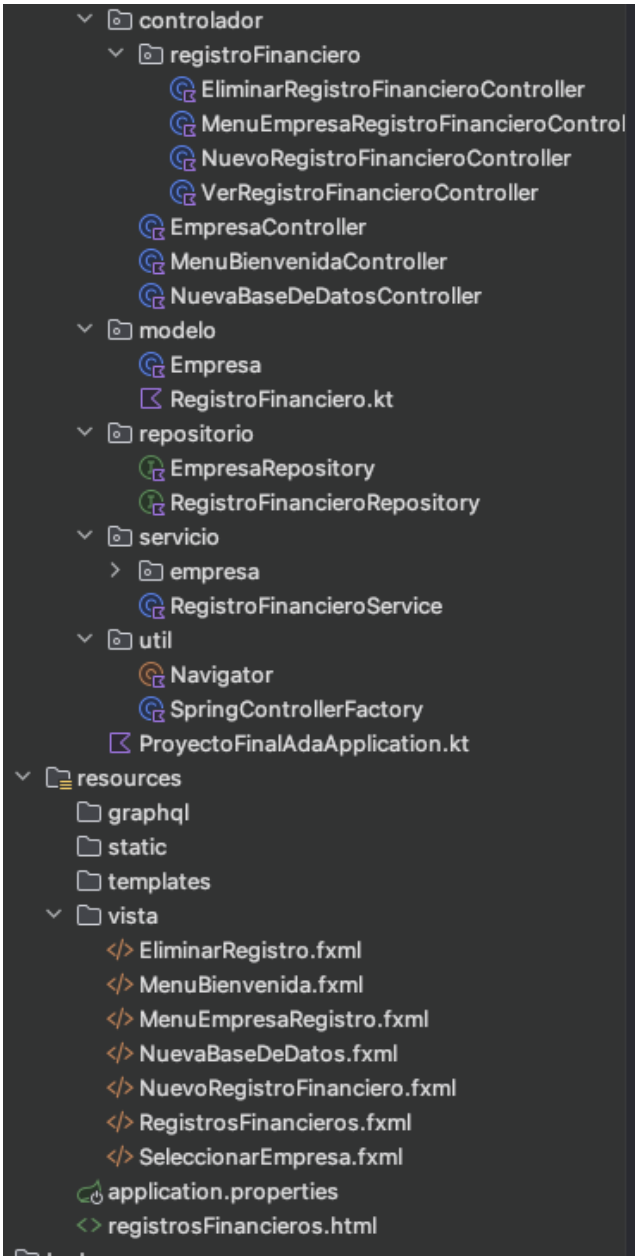
Git: El sistema de control de versiones que he usado en el proyecto y que sin duda me ha sido vital para llevar el trabajo a buen puerto, puesto que en más de alguna ocasión he tenido que rescatar código de alguna versión anterior.

3. ARQUITECTURA DEL PROGRAMA

La arquitectura de mi proyecto sigue el principio de la programación orientada a objetos y una arquitectura limpia basada en el **Modelo-Vista-Controlador.** Una arquitectura trabajada durante el curso, que ofrece ventajas claras como la separación de responsabilidades, facilidad de testeo, o mejora en la legibilidad del código.



Patrón de arquitectura MVC.



Jerarquía de mi programa con arquitectura MVC (Modelo-Vista-Controlador).

4. DOCUMENTACIÓN DEL PROYECTO

Habiendo resumido la arquitectura y jerarquía del proyecto, así como habiendo visto las diferentes tecnologías usadas para el desarrollo del programa, paso a exponer la documentación reglamentaria del código. He de añadir que todo el código se encuentra disponible con **documentación kdoc**, para un mayor entendimiento y seguimiento del mismo aun careciendo de este PDF.

Dicho esto, paso a desgranar las funciones de las clases de mi programa:

 **Paquete controlador:**

En el paquete controlador se encuentran las clases de tipo controlador de la aplicación. Las clases que controlan las vistas relacionadas con registros financieros están en un subpaquete por tema de organización y claridad en el código.

subpaquete registroFinanciero:

Dentro del paquete registroFinanciero se encuentran las clases de tipo controlador que gestionan los registros financieros, el hecho de que se usen varias clases para gestionar distintos tipos de acciones relacionadas con los registros financieros responde a la **naturaleza diversa de la aplicación** que usa Spring Boot y JavaFX. Al usar distintas pantallas de JavaFX y ser los ciclos de vida de Spring Boot y JavaFX distintos, sería contraproducente usar un solo controlador para manejar las diversas vistas, puesto que peligrarían la integridad y persistencia de los registros financieros.

Dicho esto, veamos los controladores del paquete.

EliminarRegistroFinancieroController()

```
/**
 * Controlador para eliminar un registro financiero.
 *
 * Este controlador maneja la acción del usuario de eliminar un registro financiero.
 */
@Controller
class
EliminarRegistroFinancieroController {

    @Autowired
    private lateinit var registroFinancieroService: RegistroFinancieroService

    @Autowired
    private lateinit var empresaActualService: EmpresaActualService

    @Autowired
    private lateinit var context: ApplicationContext
```

```
    @FXML
    private lateinit var registrosAEliminarList: ListView<RegistroFinanciero>

    @FXML
    private lateinit var volverAlMenuButton: Button

    /**
     * Inicializa el controlador.
     *
     * Este método se llama después de que se ha cargado el archivo FXML. Aquí es donde puedes realizar cualquier
     * inicialización necesaria para tu controlador.
     */
    @FXML
    fun initialize() {
        volverAlMenuButton.setOnAction { handleVolverAlMenuButtonAction() }

        // Carga los registros financieros de la empresa seleccionada
        val empresa = empresaActualService.getEmpresa()
        if (empresa != null) {
            val registrosFinancieros = registroFinancieroService.encontrarPorEmpresa(empresa)
            registrosAEliminarList.items = FXCollections.observableArrayList(registrosFinancieros)
        }
    }

    /**
     * Maneja la acción del botón para volver al menú.
     *
     * Cuando el usuario hace clic en el botón para volver al menú, este método carga la escena del menú de bienvenida.
     */
    fun handleVolverAlMenuButtonAction() {
        try {
            Navigator.loadScene("/vista/MenuBienvenida.fxml", context)
        } catch (e: Exception) {
            e.printStackTrace()
        }

    }

    /**
     * Maneja la acción del botón para eliminar un registro financiero.
     *
     * Cuando el usuario hace clic en el botón para eliminar un registro financiero, este método elimina el registro
     * financiero seleccionado.
     *
     * @param event El evento de acción.
     */
    @FXML
    fun handleEliminarRegistroButtonAction(event: ActionEvent) {
        val registroSeleccionado = registrosAEliminarList.selectionModel.selectedItem

        if (registroSeleccionado != null) {
            try {
                // Elimina el registro financiero
```

```
            registroFinancieroService.eliminar(registroSeleccionado)
        } catch (e: Exception) {
            e.printStackTrace()
        }
    }
  }
}
```

 # EliminarRegistroFinancieroController

La clase EliminarRegistroFinancieroController es un controlador que se utiliza para manejar las operaciones de eliminación de registros financieros en la aplicación. Esta clase utiliza RegistroFinancieroService para manejar las operaciones de los registros financieros y EmpresaActualService para obtener la empresa actual.

Propiedades

- registroFinancieroService: Un servicio para manejar las operaciones de los registros financieros.
- empresaActualService: Un servicio para obtener la empresa actual.
- context: Un contexto de la aplicación.
- registrosAEliminarList: Una lista de registros financieros a eliminar.
- volverAlMenuButton: Un botón para volver al menú.

Métodos

- initialize(): Inicializa el controlador. Este método se llama después de que se ha cargado el archivo FXML. Aquí es donde se puede realizar cualquier inicialización necesaria para el controlador.
- handleVolverAlMenuButtonAction(): Maneja la acción del botón para volver al menú. Cuando el usuario hace clic en el botón para volver al menú, este método carga la escena del menú de bienvenida.
- handleEliminarRegistroButtonAction(event: ActionEvent): Maneja la acción del botón para eliminar un registro financiero. Cuando el usuario hace clic en el botón para eliminar un registro financiero, este método elimina el registro financiero seleccionado.

MenuEmpresaRegistroFinancieroController

```
/**
 * Controlador para el menú de registros financieros de una empresa.
 *
 * Este controlador maneja las acciones del usuario en el menú de registros financieros,
 * como ver, añadir y eliminar registros financieros.
 */
@Suppress("SpellCheckingInspection")
@Controller
class MenuEmpresaRegistroFinancieroController {

    @Autowired
```

```
    private lateinit var context: ApplicationContext

    @FXML
    private lateinit var verRegistrosFinancierosButton: Button

    @FXML
    private lateinit var nuevoRegistrosFinancierosButton: Button

    @FXML
    private lateinit var eliminarRegistrosFinancierosButton: Button

    @FXML
    private lateinit var volverAlMenuButton: Button


    /**
     * Maneja la acción del botón para volver al menú.
     *
     * Cuando el usuario hace clic en el botón para volver al menú, este método carga la escena del menú
de bienvenida.
     */
    fun handleVolverAlMenuButtonAction() {
        try {
            Navigator.loadScene("/vista/MenuBienvenida.fxml", context)
        } catch (e: Exception) {
            e.printStackTrace()
        }
    }

    /**
     * Inicializa el controlador.
     *
     * Este método se llama después de que se ha cargado el archivo FXML. Aquí es donde se realiza
cualquier
     * inicialización necesaria para el controlador.
     */
    @FXML
    fun initialize() {
        volverAlMenuButton.setOnAction { handleVolverAlMenuButtonAction() }
        verRegistrosFinancierosButton.setOnAction {
Navigator.loadScene("/vista/RegistrosFinancieros.fxml", context) }
        nuevoRegistrosFinancierosButton.setOnAction {
Navigator.loadScene("/vista/NuevoRegistroFinanciero.fxml", context) }
        eliminarRegistrosFinancierosButton.setOnAction {
Navigator.loadScene("/vista/EliminarRegistro.fxml", context) }
    }
}
```

 # MenuEmpresaRegistroFinancieroController

La clase MenuEmpresaRegistroFinancieroController es un controlador que se utiliza para manejar las acciones del usuario en el menú de registros financieros, como ver, añadir y eliminar registros financieros.

Propiedades

- context: Un contexto de la aplicación.
- verRegistrosFinancierosButton: Un botón para ver los registros financieros.
- nuevoRegistrosFinancierosButton: Un botón para añadir nuevos registros financieros.
- eliminarRegistrosFinancierosButton: Un botón para eliminar registros financieros.
- volverAlMenuButton: Un botón para volver al menú.

Métodos

- handleVolverAlMenuButtonAction(): Maneja la acción del botón para volver al menú. Cuando el usuario hace clic en el botón para volver al menú, este método carga la escena del menú de bienvenida.
- initialize(): Inicializa el controlador. Este método se llama después de que se ha cargado el archivo FXML. Aquí es donde se realiza cualquier inicialización necesaria para el controlador.

NuevoRegistroFinancieroController:

```
/**
 * Controlador para la creación de un nuevo registro financiero.
 *
 * Este controlador maneja las acciones del usuario en la pantalla de creación de un nuevo registro financiero,
 * como seleccionar el tipo de registro, introducir el concepto y la cantidad, y guardar el nuevo registro.
 */
/**
 * Controlador para la creación de un nuevo registro financiero.
 *
 * Este controlador maneja las acciones del usuario en la pantalla de creación de un nuevo registro financiero,
 * como seleccionar el tipo de registro, introducir el concepto y la cantidad, y guardar el nuevo registro.
 */
@Controller
class NuevoRegistroFinancieroController {

    @Autowired
    private lateinit var registroFinancieroService: RegistroFinancieroService

    @Autowired
    private lateinit var empresaActualService: EmpresaActualService

    @FXML
    private lateinit var tipoRegistroComboBox: ComboBox<String>

    @FXML
    private lateinit var conceptoField: TextField

    @FXML
    private lateinit var cantidadField: TextField
```

```
    @Autowired
    private lateinit var context: ApplicationContext

    @FXML
    private lateinit var volverAlMenuButton: Button

    /**
     * Inicializa el controlador.
     *
     * Este método se llama después de que se ha cargado el archivo FXML. Aquí es donde puedes realizar cualquier
     * inicialización necesaria para tu controlador.
     */
    fun initialize() {
        volverAlMenuButton.setOnAction { handleVolverAlMenuButtonAction() }
        tipoRegistroComboBox.items = FXCollections.observableArrayList(
            "Inmovilizado intangible",
            "Inmovilizado material",
            "Inmovilizado financiero",
            "Amortizaciones",
            "Existencias",
            "Realizable",
            "Disponible o efectivo",
            "Capital",
            "Reservas",
            "Resultado del ejercicio",
            "Subvenciones",
            "Pasivo No Corriente",
            "Pasivo Corriente"
        )
    }

    /**
     * Maneja la acción del botón para volver al menú.
     *
     * Cuando el usuario hace clic en el botón para volver al menú, este método carga la escena del menú de bienvenida.
     */
    fun handleVolverAlMenuButtonAction() {
        try {
            Navigator.loadScene("/vista/MenuBienvenida.fxml", context)
        } catch (e: Exception) {
            e.printStackTrace()
        }
    }

    /**
     * Maneja la acción del botón para guardar un nuevo registro financiero.
     *
     * Cuando el usuario hace clic en el botón para guardar un nuevo registro financiero, este método crea un nuevo
     * registro financiero basado en el tipo seleccionado y los datos introducidos por el usuario, y luego guarda el
     * nuevo registro.
     */
    @FXML
    fun handleGuardarButtonAction() {
```

```
        val empresa = empresaActualService.getEmpresa()
        if (empresa != null) {
            val tipoRegistro = tipoRegistroComboBox.selectionModel.selectedItem
            val concepto = conceptoField.text
            val cantidad = BigDecimal(cantidadField.text)

            // Crea un nuevo registro financiero basado en el tipo seleccionado
            val registroFinanciero = when (tipoRegistro) {
                "Inmovilizado intangible" -> InmovilizadoIntangible(empresa, concepto, cantidad)
                "Inmovilizado material" -> InmovilizadoMaterial(empresa, concepto, cantidad)
                "Inmovilizado financiero" -> InmovilizadoFinanciero(empresa, concepto, cantidad)
                "Amortizaciones" -> Amortizaciones(empresa, concepto, cantidad)
                "Existencias" -> Existencias(empresa, concepto, cantidad)
                "Realizable" -> Realizable(empresa, concepto, cantidad)
                "Disponible o efectivo" -> Disponible(empresa, concepto, cantidad)
                "Capital" -> Capital(empresa, concepto, cantidad)
                "Reservas" -> Reservas(empresa, concepto, cantidad)
                "Resultado del ejercicio" -> ResultadoEjercicio(empresa, concepto, cantidad)
                "Subvenciones" -> Subvenciones(empresa, concepto, cantidad)
                "Pasivo No Corriente" -> PasivoNoCorriente(empresa, concepto, cantidad)
                "Pasivo Corriente" -> PasivoCorriente(empresa, concepto, cantidad)

                else -> null
            }

            // Guarda el registro financiero
            if (registroFinanciero != null) {
                try {
                    registroFinancieroService.guardar(registroFinanciero)
                } catch (e: Exception) {
                    e.printStackTrace()
                }
            }
        }
    }
}
```

 # NuevoRegistroFinancieroController

La clase NuevoRegistroFinancieroController es un controlador que se utiliza para manejar las operaciones de creación de nuevos registros financieros en la aplicación. Esta clase utiliza `RegistroFinancieroService` para manejar las operaciones de los registros financieros y `EmpresaActualService` para obtener la empresa actual.

Propiedades

- registroFinancieroService: Un servicio para manejar las operaciones de los registros financieros.
- empresaActualService: Un servicio para obtener la empresa actual.
- context: Un contexto de la aplicación.
- tipoRegistroComboBox: Un ComboBox para seleccionar el tipo de registro financiero.
- conceptoField: Un TextField para introducir el concepto del registro financiero.

- cantidadField: Un TextField para introducir la cantidad del registro financiero.
- volverAlMenuButton: Un botón para volver al menú.

Métodos

- initialize(): Inicializa el controlador. Este método se llama después de que se ha cargado el archivo FXML. Aquí es donde se realiza cualquier inicialización necesaria para el controlador.
- handleVolverAlMenuButtonAction(): Maneja la acción del botón para volver al menú. Cuando el usuario hace clic en el botón para volver al menú, este método carga la escena del menú de bienvenida.
- handleGuardarButtonAction(): Maneja la acción del botón para guardar un registro financiero. Cuando el usuario hace clic en el botón para guardar un registro financiero, este método crea y guarda el registro financiero.

VerRegistroFinancieroController:

```
/**
 * Controlador para la visualización de registros financieros.
 *
 * Este controlador maneja las acciones del usuario en la pantalla de visualización de registros financieros,
 * como cargar los registros financieros de la empresa seleccionada y generar una vista HTML de los registros.
 */
@Controller
class VerRegistroFinancieroController {

    @Autowired
    private lateinit var registroFinancieroService: RegistroFinancieroService

    @Autowired
    private lateinit var empresaActualService: EmpresaActualService

    @FXML
    private lateinit var registrosFinancierosList: ListView<RegistroFinanciero>

    @FXML
    private lateinit var volverAlMenuButton: Button

    @Autowired
    private lateinit var context: ApplicationContext


    /**
     * Inicializa el controlador.
     *
     * Este método se llama después de que se ha cargado el archivo FXML. Aquí es donde se realiza cualquier
     * inicialización necesaria para el controlador.
     */
    @FXML
```

```
    fun initialize() {
        volverAlMenuButton.setOnAction { handleVolverAlMenuButtonAction() }

        // Carga los registros financieros de la empresa seleccionada
        val empresa = empresaActualService.getEmpresa()
        if (empresa != null) {
            try {
                val registrosFinancieros = registroFinancieroService.encontrarPorEmpresa(empresa)
                registrosFinancierosList.items = FXCollections.observableArrayList(registrosFinancieros)
            } catch (e: Exception) {
                e.printStackTrace()
            }
        }
    }


    /**
     * Maneja la acción del botón para volver al menú.
     *
     * Cuando el usuario hace clic en el botón para volver al menú, este método carga la escena del menú
de bienvenida.
     */
    fun handleVolverAlMenuButtonAction() {
        try {
            Navigator.loadScene("/vista/MenuBienvenida.fxml", context)
        } catch (e: Exception) {
            e.printStackTrace()
        }
    }


    /**
     * Genera una vista HTML de los registros financieros.
     *
     * Cuando el usuario hace clic en el botón para generar la vista HTML, este método crea una nueva
vista HTML de los
     * registros financieros de la empresa seleccionada y la abre en el navegador predeterminado, para
ello se usa
     * un procedimiento multihilo para compaginar JavaFX y ThymeLeaf en sus distintos ciclos de vida.
     */
    @FXML
    fun generateHtmlView() {
        Thread {
            val templateResolver = ClassLoaderTemplateResolver()
            templateResolver.suffix = ".html"
            templateResolver.templateMode = TemplateMode.HTML

            val templateEngine = TemplateEngine()
            templateEngine.setTemplateResolver(templateResolver)

            val context = Context()
            val empresa = empresaActualService.getEmpresa()
            if (empresa != null) {
                try {
                    val registrosFinancieros = registroFinancieroService.encontrarPorEmpresa(empresa)
                    context.setVariable("registrosFinancieros", registrosFinancieros)
                } catch (e: Exception) {
```

```
                e.printStackTrace()
            }
        }

        val html = templateEngine.process("registrosFinancieros", context)

        // Guarda la cadena HTML en un archivo con un nombre único
        val timestamp = SimpleDateFormat("yyyyMMddHHmmss").format(Date())
        val filename = "registrosFinancieros$timestamp.html"
        val file = File(filename)
        PrintWriter(file).use { out -> out.println(html) }

        // Abre el archivo en el navegador predeterminado
        Platform.runLater {
            val url = file.toURI().toString()

            if (Desktop.isDesktopSupported() &&
Desktop.getDesktop().isSupported(Desktop.Action.BROWSE)) {
                Desktop.getDesktop().browse(URI(url))
            } else {
                val runtime = Runtime.getRuntime()
                try {
                    val os = System.getProperty("os.name").lowercase(Locale.getDefault())
                    if (os.contains("win")) {
                        runtime.exec("cmd /c start $url")
                    } else if (os.contains("mac")) {
                        runtime.exec("open $url")
                    } else if (os.contains("nix") || os.contains("nux")) {
                        runtime.exec("xdg-open $url")
                    }
                } catch (e: IOException) {
                    e.printStackTrace()
                }
            }
        }
    }.start()
  }
}
```

 # VerRegistroFinancieroController

La clase VerRegistroFinancieroController es un controlador que se utiliza para manejar las acciones del usuario en la pantalla de visualización de registros financieros, como cargar los registros financieros de la empresa seleccionada y generar una vista HTML de los registros usando **ThymeLeaf**.

Propiedades

- registroFinancieroService: Un servicio para manejar las operaciones de los registros financieros.
- empresaActualService: Un servicio para obtener la empresa actual.
- context: Un contexto de la aplicación.

- registrosFinancierosList: Una lista de registros financieros a visualizar.
- volverAlMenuButton: Un botón para volver al menú.

Métodos

- initialize(): Inicializa el controlador. Este método se llama después de que se ha cargado el archivo FXML. Aquí es donde se realiza cualquier inicialización necesaria para el controlador.
- handleVolverAlMenuButtonAction(): Maneja la acción del botón para volver al menú. Cuando el usuario hace clic en el botón para volver al menú, este método carga la escena del menú de bienvenida.
- generateHtmlView(: Genera una vista HTML de los registros financieros. Cuando el usuario hace clic en el botón para generar la vista HTML, este método crea una nueva vista HTML de los registros financieros de la empresa seleccionada y la abre en el navegador predeterminado usando ThymeLeaf.

EmpresaController:

```
/**
    * Maneja la acción del botón para volver al menú.
    *
    * Cuando el usuario hace clic en el botón para volver al menú, este método carga la escena del menú
de bienvenida.
    */
   @FXML
   fun handleAtrasButtonAction() {
      try {
         Navigator.loadScene("/vista/MenuBienvenida.fxml", context)
      } catch (e: Exception) {
         e.printStackTrace()
      }
   }

   /**
    * Inicializa el controlador.
    *
    * Este método se llama después de que se ha cargado el archivo FXML. Aquí es donde puedes realizar
cualquier
    * inicialización necesaria para tu controlador.
    */
   @FXML
   fun initialize() {
      // Carga las empresas y las añade al ComboBox
      val empresas = empresaService.encontrarTodo()
      val nombresEmpresas = empresas.map { it.nombre }
      empresaComboBox.items = FXCollections.observableArrayList(nombresEmpresas)

      // Añade un listener para manejar los cambios de selección en el ComboBox
      empresaComboBox.valueProperty().addListener { _, _, selectedEmpresa ->
         val empresa = empresas.find { it.nombre == selectedEmpresa }
         if (empresa != null) {
            empresaActualService.setEmpresa(empresa)
```

```
        }
    }

    // Añade un listener al botón de cargar para cargar la escena de registros financieros de la empresa
seleccionada
    cargarButton.setOnAction {
        try {
            Navigator.loadScene("/vista/MenuEmpresaRegistro.fxml", context)
        } catch (e: Exception) {
            e.printStackTrace()
        }
    }
  }
}
```

#EmpresaController

Este controlador maneja las acciones del usuario en la pantalla de visualización de registros financieros, como cargar los registros financieros de la empresa seleccionada y generar una vista HTML de los registros.

Propiedades

- context: Un contexto de la aplicación.
- empresaComboBox: Un ComboBox para seleccionar la empresa.
- volverAlMenuButton: Un botón para volver al menú.
- cargarButton: Un botón para cargar la escena de registros financieros de la empresa seleccionada.

Métodos

- handleAtrasButtonAction(): Maneja la acción del botón para volver al menú. Cuando el usuario hace clic en el botón para volver al menú, este método carga la escena del menú de bienvenida.
- initialize(): Inicializa el controlador. Este método se llama después de que se ha cargado el archivo FXML. Aquí es donde se realiza cualquier inicialización necesaria para el controlador. Carga las empresas y las añade al ComboBox, añade un listener para manejar los cambios de selección en el ComboBox, y añade un listener al botón de cargar para cargar la escena de registros financieros de la empresa seleccionada.

MenuBienveidaController:

```
/**
 * Controlador para el menú de bienvenida.
 *
 * Este controlador maneja las acciones del usuario en la pantalla de bienvenida,
 * como crear una nueva base de datos, usar una base de datos existente y ajustar la base de datos.
 *
 * @property context El contexto de la aplicación.
 */
```

```kotlin
@Controller
class MenuBienvenidaController(private val context: ApplicationContext) {


    /**
     * Maneja la acción del botón para crear una nueva base de datos.
     *
     * Cuando el usuario hace clic en el botón para crear una nueva base de datos, este método carga la escena de creación de una nueva base de datos.
     *
     * @param event El evento de acción.
     */
    @FXML
    fun handleCrearNuevaBaseDeDatosButtonAction(event: ActionEvent) {
        try {
            Navigator.loadScene("/vista/NuevaBaseDeDatos.fxml", context)
        } catch (e: Exception) {
            e.printStackTrace()
        }
    }

    /**
     * Maneja la acción del botón para usar una base de datos existente.
     *
     * Cuando el usuario hace clic en el botón para usar una base de datos existente, este método carga la escena de selección de empresa.
     *
     * @param event El evento de acción.
     */
    @FXML
    fun handleUsarBaseDeDatosExistenteButtonAction(event: ActionEvent) {
        try {
            Navigator.loadScene("/vista/SeleccionarEmpresa.fxml", context)
        } catch (e: Exception) {
            e.printStackTrace()
        }
    }
}
```

 # MenuBienvenidaController

La clase MenuBienvenidaController es un controlador que se utiliza para manejar las acciones del usuario en la pantalla de bienvenida, como crear una nueva base de datos y usar una base de datos existente.

Propiedades

- context: Un contexto de la aplicación.

Métodos

- handleCrearNuevaBaseDeDatosButtonAction(event: ActionEvent): Maneja la acción del botón para crear una nueva base de datos. Cuando el usuario hace clic en el botón

para crear una nueva base de datos, este método carga la escena de creación de una nueva base de datos.
- handleUsarBaseDeDatosExistenteButtonAction(event: ActionEvent): Maneja la acción del botón para usar una base de datos existente. Cuando el usuario hace clic en el botón para usar una base de datos existente, este método carga la escena de selección de empresa.

NuevaBaseDeDatosController:

```
/**
 * Controlador para la creación de una nueva base de datos.
 *
 * Este controlador maneja las acciones del usuario en la pantalla de creación de una nueva base de datos,
 * como introducir el nombre de la nueva base de datos y guardar la nueva base de datos.
 *
 * @property empresaService El servicio para interactuar con las empresas.
 */
@Controller
class NuevaBaseDeDatosController(private val empresaService: EmpresaService) {

    @Autowired
    private lateinit var context: ApplicationContext

    @FXML
    private lateinit var nombreField: TextField

    /**
     * Maneja la acción del botón para volver al menú.
     *
     * Cuando el usuario hace clic en el botón para volver al menú, este método carga la escena del menú de bienvenida.
     */
    fun handleAtrasButtonAction() {
        try {
            Navigator.loadScene("/vista/MenuBienvenida.fxml", context)
        } catch (e: Exception) {
            e.printStackTrace()
        }
    }

    /**
     * Maneja la acción del botón para guardar una nueva base de datos.
     *
     * Cuando el usuario hace clic en el botón para guardar una nueva base de datos, este método crea una nueva base de datos
     * con el nombre introducido por el usuario y luego guarda la nueva base de datos.
     */
    @FXML
    fun handleGuardarButtonAction() {
        try {
            val nombre = nombreField.text
            val nuevaEmpresa = Empresa(nombre = nombre)
            val savedEmpresa = empresaService.guardar(nuevaEmpresa)
            println("Empresa guardada: $savedEmpresa")
```

```
    } catch (e: Exception) {
        e.printStackTrace()
    }
  }
}
```

 # NuevaBaseDeDatosController

La clase NuevaBaseDeDatosController es un controlador que se utiliza para manejar las acciones del usuario en la pantalla de creación de una nueva base de datos, como introducir el nombre de la nueva base de datos y guardar la nueva base de datos.

Propiedades

- empresaService: Un servicio para interactuar con las empresas.
- context: Un contexto de la aplicación.
- nombreField: Un TextField para introducir el nombre de la nueva base de datos.

Métodos

- handleAtrasButtonAction(): Maneja la acción del botón para volver al menú. Cuando el usuario hace clic en el botón para volver al menú, este método carga la escena del menú de bienvenida.
- handleGuardarButtonAction(): Maneja la acción del botón para guardar una nueva base de datos. Cuando el usuario hace clic en el botón para guardar una nueva base de datos, este método crea una nueva base de datos con el nombre introducido por el usuario y luego guarda la nueva base de datos.

Paquete Modelo:

Dentro del modelo de la aplicación se encuentran las clases que encierran la lógica para el funcionamiento del programa. El modelo de mi aplicación encierra clases ligeras. Gestionan también la lógica interna y el funcionamiento del software los paquetes **repositorio, servicio y util**, los cuales desgranaré aquí. El motivo por el que estas clases se encuentran fuera del paquete modelo es debido a la **convención general** de programación en **Kotlin**, puesto que el estándar requiere que los repositorios se encuentren en su propio paquete, al igual que las clases que componen el servicio u otras utilidades como la navegación. Dicho esto, paso a desgranar el modelo.

El modelo está basado fundamentalmente en todas las clases que recogen los distintos tipos de registros financieros existentes en el programa, cada uno con su clase, más la clase que gestiona los objetos de tipo empresa. Estas son pequeñas clases que siguen el principio de **herencia** de la programación orientada a **objetos**, encontrándonos con subclases dentro de subclases, asegurando así que tratamos con entidades **lo**

suficientemente complejas, acorde a lo estudiado a lo largo del curso. En total mi modelo gestiona **más de 20 clases.**

Empresa

```
@Entity
data class Empresa(
    @jakarta.persistence.Id @Id @GeneratedValue(strategy = GenerationType.AUTO)
    val id: Long = 0,

    val nombre: String,

    @OneToMany(mappedBy = "empresa", cascade = [CascadeType.ALL], orphanRemoval = true)
    val registros: List<RegistroFinanciero> = ArrayList()
)
```

La clase `Empresa` representa una entidad de empresa en la base de datos. Cada empresa tiene un `id`, un `nombre` y una lista de `registros` financieros.

RegistroFinanciero

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "tipo_registro")
sealed class RegistroFinanciero(
    @jakarta.persistence.Id @Id @GeneratedValue(strategy = GenerationType.AUTO)
    val id: Long = 0,

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "empresa_id")
    val empresa: Empresa,

    val concepto: String,

    val cantidad: BigDecimal
)

// ACTIVO
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "tipo_activo")
open class Activo(
    empresa: Empresa,
    concepto: String,
    cantidad: BigDecimal
) : RegistroFinanciero(empresa = empresa, concepto = concepto, cantidad = cantidad) {
    override fun toString(): String {
        return "$concepto $cantidad euros"
    }
}

// ACTIVO NO CORRIENTE
@Entity
```

```
@DiscriminatorValue("activo_no_corriente")
open class ActivoNoCorriente(
    empresa: Empresa,
    concepto: String,
    cantidad: BigDecimal
) : Activo(empresa = empresa, concepto = concepto, cantidad = cantidad) {
    override fun toString(): String {
        return "$concepto $cantidad euros"
    }
}

@Entity
@DiscriminatorValue("inmovilizado_intangible")
class InmovilizadoIntangible(
    empresa: Empresa,
    concepto: String,
    cantidad: BigDecimal) : ActivoNoCorriente(empresa = empresa, concepto = concepto, cantidad =
cantidad) {
    override fun toString(): String {
        return "$concepto $cantidad euros"
    }
    }

@Entity
@DiscriminatorValue("inmovilizado_material")
class InmovilizadoMaterial(
    empresa: Empresa,
    concepto: String,
    cantidad: BigDecimal) : ActivoNoCorriente(empresa = empresa, concepto = concepto, cantidad =
cantidad) {
    override fun toString(): String {
        return "$concepto $cantidad euros"
    }
    }

@Entity
@DiscriminatorValue("inmovilizado_financiero")
class InmovilizadoFinanciero(
    empresa: Empresa,
    concepto: String,
    cantidad : BigDecimal) : ActivoNoCorriente(empresa = empresa, concepto = concepto, cantidad =
cantidad) {
    override fun toString(): String {
        return "$concepto $cantidad euros"
    }
    }

@Entity
@DiscriminatorValue("amortizaciones")
class Amortizaciones(
    empresa: Empresa,
    concepto: String,
    cantidad : BigDecimal) : ActivoNoCorriente(empresa = empresa, concepto = concepto, cantidad =
cantidad) {
    override fun toString(): String {
        return "$concepto $cantidad euros"
    }
```

```
    }

// ACTIVO CORRIENTE
@Entity
@DiscriminatorValue("activo_corriente")
open class ActivoCorriente(
    empresa: Empresa,
    concepto: String,
    cantidad: BigDecimal
) : Activo(empresa = empresa, concepto = concepto, cantidad = cantidad) {
    override fun toString(): String {
        return "$concepto $cantidad euros"
    }
}

@Entity
@DiscriminatorValue("existencias")
class Existencias(
    empresa: Empresa,
    concepto: String,
    cantidad: BigDecimal) : ActivoCorriente(empresa = empresa, concepto = concepto, cantidad =
cantidad) {
    override fun toString(): String {
        return "$concepto $cantidad euros"
    }
    }

@Entity
@DiscriminatorValue("realizable")
class Realizable(
    empresa: Empresa,
    concepto: String,
    cantidad: BigDecimal) : ActivoCorriente(empresa = empresa, concepto = concepto, cantidad =
cantidad) {
    override fun toString(): String {
        return "$concepto $cantidad euros"
    }
    }

@Entity
@DiscriminatorValue("disponible")
class Disponible(
    empresa: Empresa,
    concepto: String,
    cantidad: BigDecimal) : ActivoCorriente(empresa = empresa, concepto = concepto, cantidad =
cantidad) {
    override fun toString(): String {
        return "$concepto $cantidad euros"
    }
    }

// PASIVO
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "tipo_pasivo")
open class Pasivo(
    empresa: Empresa,
```

```
    concepto: String,
    cantidad: BigDecimal
) : RegistroFinanciero(empresa = empresa, concepto = concepto, cantidad = cantidad) {
    override fun toString(): String {
        return "$concepto $cantidad euros"
    }
}

// NETO
@Entity
@DiscriminatorValue("neto")
open class Neto(
    empresa: Empresa,
    concepto: String,
    cantidad: BigDecimal
) : Pasivo(empresa = empresa, concepto = concepto, cantidad = cantidad) {
    override fun toString(): String {
        return "$concepto $cantidad euros"
    }
}

@Entity
@DiscriminatorValue("capital")
class Capital(
    empresa: Empresa,
    concepto: String,
    cantidad: BigDecimal) : Neto(empresa = empresa, concepto = concepto, cantidad = cantidad) {
    override fun toString(): String {
        return "$concepto $cantidad euros"
    }
    }

@Entity
@DiscriminatorValue("reservas")
class Reservas(
    empresa: Empresa,
    concepto: String,
    cantidad: BigDecimal) : Neto(empresa = empresa, concepto = concepto, cantidad = cantidad) {
    override fun toString(): String {
        return "$concepto $cantidad euros"
    }
    }

@Entity
@DiscriminatorValue("resultado_ejercicio")
class ResultadoEjercicio(
    empresa: Empresa,
    concepto: String,
    cantidad: BigDecimal) : Neto(empresa = empresa, concepto = concepto, cantidad = cantidad) {
    override fun toString(): String {
        return "$concepto $cantidad euros"
    }
    }

@Entity
@DiscriminatorValue("subvenciones")
class Subvenciones(
```

```
    empresa: Empresa,
    concepto: String,
    cantidad: BigDecimal) : Neto(empresa = empresa, concepto = concepto, cantidad = cantidad) {
    override fun toString(): String {
        return "$concepto $cantidad euros"
    }
    }

// PASIVO NO CORRIENTE
@Entity
@DiscriminatorValue("pasivo_no_corriente")
open class PasivoNoCorriente(
    empresa: Empresa,
    concepto: String,
    cantidad: BigDecimal
) : Pasivo(empresa = empresa, concepto = concepto, cantidad = cantidad) {
    override fun toString(): String {
        return "$concepto $cantidad euros"
    }
}

@Entity
@DiscriminatorValue("deudas_largo_plazo")
class DeudasLargoPlazo(empresa: Empresa,
                       concepto: String,
                       cantidad: BigDecimal
) : PasivoNoCorriente(empresa = empresa, concepto = concepto, cantidad = cantidad) {
    override fun toString(): String {
        return "$concepto $cantidad euros"
    }
}

// PASIVO CORRIENTE
@Entity
@DiscriminatorValue("pasivo_corriente")
open class PasivoCorriente(
    empresa: Empresa,
    concepto: String,
    cantidad: BigDecimal
) : Pasivo(empresa = empresa, concepto = concepto, cantidad = cantidad) {
    override fun toString(): String {
        return "$concepto $cantidad euros"
    }
}

@Entity
@DiscriminatorValue("deudas_corto_plazo")
class DeudasCortoPlazo(
    empresa: Empresa,
    concepto: String,
    cantidad: BigDecimal) : PasivoCorriente(empresa = empresa, concepto = concepto, cantidad =
cantidad) {
    override fun toString(): String {
        return "$concepto $cantidad euros"
    }
}
```

 # RegistroFinanciero

La clase RegistroFinanciero es una clase sellada que representa un registro financiero genérico en la base de datos. Cada registro financiero tiene un id, una empresa asociada, un concepto y una cantidad.

 # Activo

La clase `Activo` es una subclase de `RegistroFinanciero` que representa un activo en la base de datos.

 # ActivoNoCorriente

La clase ActivoNoCorriente es una subclase de Activo que representa un activo no corriente en la base de datos.

InmovilizadoIntangible

La clase InmovilizadoIntangible es una subclase de ActivoNoCorriente que representa un inmovilizado intangible en la base de datos.

InmovilizadoMaterial

La clase InmovilizadoMaterial es una subclase de ActivoNoCorriente que representa un inmovilizado material en la base de datos.

InmovilizadoFinanciero

La clase InmovilizadoFinanciero es una subclase de ActivoNoCorriente que representa un inmovilizado financiero en la base de datos.

Amortizaciones

La clase Amortizaciones es una subclase de ActivoNoCorriente que representa las amortizaciones en la base de datos.

 # ActivoCorriente

La clase ActivoCorriente es una subclase de Activo que representa un activo corriente en la base de datos.

Existencias

La clase Existencias es una subclase de ActivoCorriente que representa las existencias en la base de datos.

Realizable

La clase Realizable es una subclase de ActivoCorriente que representa el realizable en la base de datos.

Disponible

La clase Disponible es una subclase de ActivoCorriente que representa el disponible en la base de datos.

 # Pasivo

La clase Pasivo es una clase abierta que representa un pasivo genérico en la base de datos. Cada pasivo tiene un id, una empresa asociada, un concepto y una cantidad.

 # Neto

La clase Neto es una subclase de Pasivo que representa un neto en la base de datos.

Capital

La clase Capital es una subclase de Neto que representa el capital en la base de datos.

Reservas

La clase Reservas es una subclase de Neto que representa las reservas en la base de datos.

ResultadoEjercicio

La clase ResultadoEjercicio es una subclase de Neto que representa el resultado del ejercicio en la base de datos.

Subvenciones

La clase Subvenciones es una subclase de Neto que representa las subvenciones en la base de datos.

 # PasivoNoCorriente

La clase PasivoNoCorriente es una subclase de Pasivo que representa un pasivo no corriente en la base de datos.

DeudasLargoPlazo

La clase DeudasLargoPlazo es una subclase de PasivoNoCorriente que representa las deudas a largo plazo en la base de datos.

 # PasivoCorriente

La clase PasivoCorriente es una subclase de Pasivo que representa un pasivo corriente en la base de datos.

DeudasCortoPlazo

La clase DeudasCortoPlazo es una subclase de PasivoCorriente que representa las deudas a corto plazo en la base de datos.

Paquetes repositorio, servicio y util.

Como cité anteriormente, las clases que se encuentran en estos paquetes encierran lógica necesaria para el funcionamiento interno de las librerías que usa el programa, como Spring Boot, controlar los ciclos de vida de las entidades o establecer la navegación, es por eso que son de vital importancia, sin embargo, la convención recomendada exige encerrarlas en sus respectivos paquetes.

```
/**
 * El interface de Empresa
 */
@Repository
interface EmpresaRepository : JpaRepository<Empresa, Long>
```

 # EmpresaRepository

La interfaz EmpresaRepository extiende JpaRepository y proporciona métodos para interactuar con la entidad `Empresa` en la base de datos.

```
/**
 * El interface de RegistroFinanciero
 */
interface RegistroFinancieroRepository : JpaRepository<RegistroFinanciero, Long> {

    fun findByEmpresa(empresa: Empresa): List<RegistroFinanciero>
}
```

 # RegistroFinancieroRepository

La interfaz RegistroFinancieroRepository extiende JpaRepository y proporciona métodos para interactuar con la entidad RegistroFinanciero en la base de datos. Incluye un método personalizado findByEmpresa para encontrar todos los registros financieros asociados a una empresa específica.

Servicio

Dentro del paquete servicio se encuentran tres clases. Dos dedicadas a empresas y otro al servicio propio de los registros financieros.

```kotlin
/**
 * Servicio para interactuar con las empresas en la base de datos.
 *
 * Este servicio proporciona métodos para encontrar todas las empresas y guardar una nueva empresa.
 *
 * @property empresaRepository El repositorio para interactuar con las empresas en la base de datos.
 */
@Service
class EmpresaService(private val empresaRepository: EmpresaRepository) {

    /**
     * Encuentra todas las empresas en la base de datos.
     *
     * @return Una lista de todas las empresas.
     */
    @Transactional()
    fun encontrarTodo(): List<Empresa> {
        return try {
            empresaRepository.findAll()
        } catch (e: Exception) {
            e.printStackTrace()
            emptyList()
        }
    }

    /**
     * Guarda una nueva empresa en la base de datos.
     *
     * @param empresa La empresa a guardar.
     * @return La empresa guardada.
     */
    @Transactional
    fun guardar(empresa: Empresa): Empresa {
        return try {
            empresaRepository.save(empresa)
        } catch (e: Exception) {
            e.printStackTrace()
            empresa
        }
    }
}
```

 # EmpresaService

La clase EmpresaService es un servicio que proporciona métodos para interactuar con las empresas en la base de datos. Proporciona métodos para encontrar todas las empresas (encontrarTodo) y guardar una nueva empresa (guardar).

```
/**
 * El servicio de empresa que se usa para mantener
 * la empresa seleccionada a lo largo de la vida
 * util de la aplicación.
 */
@Service
class EmpresaActualService {
    private var empresa: Empresa? = null

    /**
     * Obtiene la empresa actualmente seleccionada.
     *
     * @return La empresa actualmente seleccionada, o null si no hay ninguna empresa seleccionada.
     */
    fun getEmpresa(): Empresa? {
        return empresa
    }

    /**
     * Establece la empresa actualmente seleccionada.
     *
     * @param empresa La empresa a seleccionar.
     */
    fun setEmpresa(empresa: Empresa) {
        this.empresa = empresa
    }
}
```

 # EmpresaActualService

La clase EmpresaActualService es un servicio que se utiliza para mantener la empresa seleccionada a lo largo de la vida útil de la aplicación. Proporciona métodos para obtener (getEmpresa) y establecer (setEmpresa) la empresa actualmente seleccionada.

```
/**
 * Servicio para interactuar con los registros financieros en la base de datos.
 *
 * Este servicio proporciona métodos para encontrar todos los registros financieros, guardar un nuevo registro financiero,
 * eliminar un registro financiero y encontrar registros financieros por empresa.
 *
 * @property registroFinancieroRepository El repositorio para interactuar con los registros financieros en la base de datos.
 * @property empresaRepository El repositorio para interactuar con las empresas en la base de datos.
 */
@Service
class RegistroFinancieroService(
    private val registroFinancieroRepository: RegistroFinancieroRepository,
    private val empresaRepository: EmpresaRepository
) {

    /**
     * Guarda un nuevo registro financiero en la base de datos.
     *
     * @param registroFinanciero El registro financiero a guardar.
```

```
     * @return El registro financiero guardado.
     */
    @Transactional
    fun guardar(registroFinanciero: RegistroFinanciero): RegistroFinanciero {
        return try {
            registroFinancieroRepository.save(registroFinanciero)
        } catch (e: Exception) {
            e.printStackTrace()
            registroFinanciero
        }
    }

    /**
     * Elimina un registro financiero de la base de datos.
     *
     * @param registroFinanciero El registro financiero a eliminar.
     */
    @Transactional
    fun eliminar(registroFinanciero: RegistroFinanciero) {
        try {
            registroFinancieroRepository.delete(registroFinanciero)
        } catch (e: Exception) {
            e.printStackTrace()
        }
    }

    /**
     * Encuentra los registros financieros de una empresa específica.
     *
     * @param empresa La empresa cuyos registros financieros se van a encontrar.
     * @return Una lista de los registros financieros de la empresa.
     */
    fun encontrarPorEmpresa(empresa: Empresa): List<RegistroFinanciero> {
        return try {
            registroFinancieroRepository.findByEmpresa(empresa)
        } catch (e: Exception) {
            e.printStackTrace()
            emptyList()
        }
    }
}
```

 # RegistroFinancieroService

La clase RegistroFinancieroService es un servicio que proporciona métodos para interactuar con los registros financieros en la base de datos. Proporciona métodos para guardar un nuevo registro financiero (guardar), eliminar un registro financiero (eliminar) y encontrar los registros financieros de una empresa específica (encontrarPorEmpresa).

El paquete util

El paquete util encierra la lógica necesaria para navegar entre pantallas de JavaFX y el SpringControllerFactory para crear instancias de los controladores, lo que permite a Spring inyectar dependencias en ellos.

```kotlin
/**
 * Objeto para navegar entre las diferentes escenas de la aplicación.
 *
 * Este objeto proporciona métodos para establecer el escenario principal y cargar una nueva escena.
 */
object Navigator {
    private var primaryStage: Stage? = null

    /**
     * Establece el escenario principal de la aplicación.
     *
     * @param stage El escenario principal.
     */
    fun setPrimaryStage(stage: Stage?) {
        primaryStage = stage
    }

    /**
     * Carga una nueva escena en el escenario principal.
     *
     * Este método carga un archivo FXML y lo establece como la escena del escenario principal.
     *
     * @param fxmlPath La ruta al archivo FXML de la escena a cargar.
     * @param context El contexto de la aplicación.
     */
    fun loadScene(fxmlPath: String?, context: ApplicationContext?) {
        try {
            val fxmlLoader = FXMLLoader()
            fxmlLoader.classLoader = this.javaClass.classLoader
            fxmlLoader.location = Navigator::class.java.getResource(fxmlPath)
            fxmlLoader.setControllerFactory { clazz -> context!!.getBean(clazz) }
            val root: Parent = fxmlLoader.load()
            primaryStage!!.scene = Scene(root)
            primaryStage!!.show()
        } catch (e: IOException) {
            e.printStackTrace()
        }
    }
}
```

 # Navigator

La clase Navigator es un objeto que se utiliza para navegar entre las diferentes escenas de la aplicación. Proporciona métodos para establecer el escenario principal (setPrimaryStage) y cargar una nueva escena (loadScene) en el escenario principal.

Propiedades

- primaryStage: El escenario principal de la aplicación.

Métodos

- setPrimaryStage(stage: Stage?): Establece el escenario principal de la aplicación.

- loadScene(fxmlPath: String?, context: ApplicationContext?): Carga una nueva escena en el escenario principal. Este método carga un archivo FXML y lo establece como la escena del escenario principal.

```
/**
 * Fábrica de controladores para Spring.
 *
 * Esta clase es un componente de Spring que se utiliza para crear instancias de controladores.
 * Implementa la interfaz `Callback` de JavaFX, lo que permite a Spring inyectar dependencias en los controladores.
 *
 * @property context El contexto de la aplicación de Spring.
 */
@Component
class SpringControllerFactory(private val context: ApplicationContext) : Callback<Class<*>, Any> {

    /**
     * Crea una instancia de un controlador.
     *
     * Este método se llama cuando se necesita una nueva instancia de un controlador.
     * Utiliza el contexto de la aplicación de Spring para crear la instancia, lo que permite la inyección de dependencias.
     *
     * @param param La clase del controlador.
     * @return Una nueva instancia del controlador.
     */
    override fun call(param: Class<*>): Any {
        return context.getBean(param)
    }
}
```

 # SpringControllerFactory

La clase SpringControllerFactory es un componente de Spring que se utiliza para crear instancias de controladores. Implementa la interfaz `Callback` de JavaFX, lo que permite a Spring inyectar dependencias en los controladores.

Propiedades

- context: El contexto de la aplicación de Spring.

Métodos

- call(param: Class<*>): Crea una instancia de un controlador. Este método se llama cuando se necesita una nueva instancia de un controlador. Utiliza el contexto de la aplicación de Spring para crear la instancia, lo que permite la inyección de dependencias.

Application

```
/**
 * Clase principal de la aplicación.
 *
 * Esta clase es la entrada a la aplicación y se encarga de iniciar la aplicación Spring, cargar la escena inicial y cerrar el contexto de la aplicación cuando se detiene.
 */
@SpringBootApplication
class ProyectoFinalAdaApplication : Application() {

    // El contexto de la aplicación Spring.
    private lateinit var context: ConfigurableApplicationContext

    /**
     * Inicializa la aplicación.
     *
     * Este método se llama antes de que se inicie la aplicación JavaFX. Aquí es donde se inicia la aplicación Spring.
     */
    override fun init() {
        context = SpringApplication.run(ProyectoFinalAdaApplication::class.java)
    }

    /**
     * Inicia la aplicación.
     *
     * Este método se llama después de que se ha inicializado la aplicación JavaFX. Aquí es donde se carga la escena inicial en el escenario principal.
     *
     * @param primaryStage El escenario principal de la aplicación.
     */
    override fun start(primaryStage: Stage) {
        Navigator.setPrimaryStage(primaryStage)
        Navigator.loadScene("/vista/MenuBienvenida.fxml", context)
    }

    /**
     * Detiene la aplicación.
     *
     * Este método se llama cuando se detiene la aplicación JavaFX. Aquí es donde se cierra el contexto de la aplicación Spring.
     */
    override fun stop() {
        context.close()
    }
}

/**
 * Método principal de la aplicación.
 *
 * Este método lanza la aplicación JavaFX.
 *
 * @param args Los argumentos de la línea de comandos.
 */
fun main(args: Array<String>) {
```

```
    Application.launch(ProyectoFinalAdaApplication::class.java, *args)
}
```

 # ProyectoFinalAdaApplication

La clase ProyectoFinalAdaApplication es la clase principal de la aplicación y se encarga de iniciar la aplicación Spring, cargar la escena inicial y cerrar el contexto de la aplicación cuando se detiene.

Propiedades

- context: El contexto de la aplicación Spring.

Métodos

- init(): Inicializa la aplicación. Este método se llama antes de que se inicie la aplicación JavaFX. Aquí es donde se inicia la aplicación Spring.
- start(primaryStage: Stage): Inicia la aplicación. Este método se llama después de que se ha inicializado la aplicación JavaFX. Aquí es donde se carga la escena inicial en el escenario principal.
- stop(): Detiene la aplicación. Este método se llama cuando se detiene la aplicación JavaFX. Aquí es donde se cierra el contexto de la aplicación Spring.

 # main

El método `main es el método principal de la aplicación. Este método lanza la aplicación JavaFX.

5. LAS VISTAS Y EL FUNCIONAMIENTO DEL PROGRAMA.

En este apartado enseñaré el acabado final de los fxml del programa creados con Scene-Builder, al mismo tiempo que voy enseñando el funcionamiento del programa de forma sencilla.

Como dije al principio del texto, el programa se dedica a gestionar recursos financieros de empresas, especialmente centrado en un año financiero.

MenuBienvenida.fxml

La primera pantalla de la aplicación. Como se puede ver es sencilla y directa. Le da la opción al usuario de crear una nueva empresa. Para continuar con la explicación del programa escogeré esa opción.

NuevaBaseDeDatos.fxml

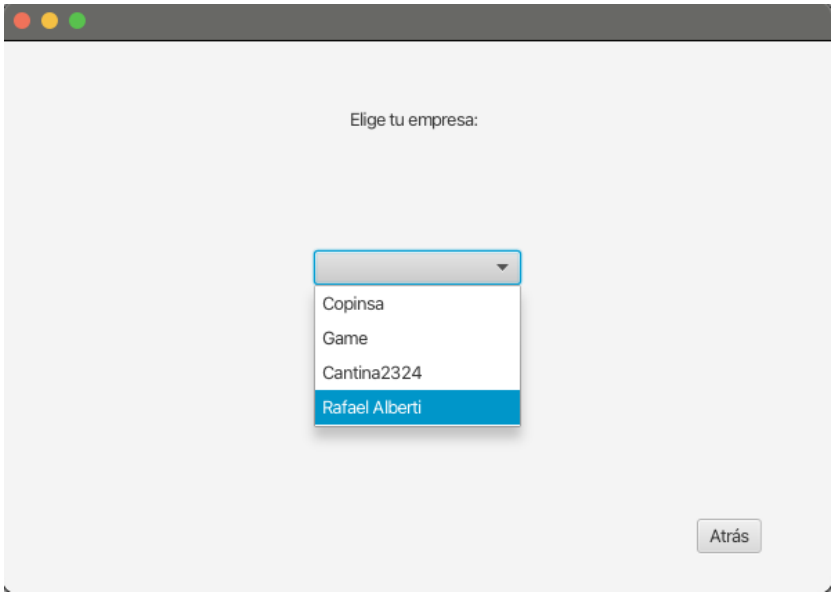
Esta segunda pantalla de la aplicación comparte en líneas generales el estilo de la primera. Es sencilla y permite al usuario darle un nombre a la empresa que va a registrar. Para este caso escogeremos el nombre de nuestra institución académica. IES Rafael Alberti, pulsando sobre el botón “Guardar”.

Desde la consola del programa, podemos ver como Hibernate ha guardado la empresa correctamente.

```
2024-02-28T21:45:40.749+01:00 DEBUG 31493 --- [lication Thread] org.hibernate.SQL                        : update empresa_seq set next_val= ? where next_val=?
Hibernate: update empresa_seq set next_val= ? where next_val=?
2024-02-28T21:45:40.794+01:00 DEBUG 31493 --- [lication Thread] org.hibernate.SQL                        : insert into empresa (nombre,id) values (?,?)
Hibernate: insert into empresa (nombre,id) values (?,?)
Empresa guardada: Empresa(id=102, nombre=Rafael Alberti , registros=[])
```

Una vez el usuario ha creado su empresa, puede acceder a SeleccionarEmpresa.fxml desde MenuBienvenida.fxml.

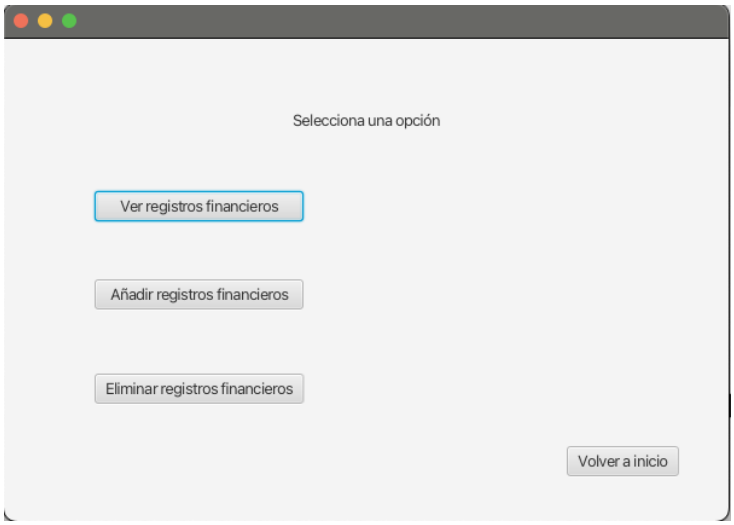
SeleccionarEmpresa.fxml



Siguiendo con la línea sencilla pero intuitiva del programa, SeleccionarEmpresa es una pantalla con un ComboBox que muestra las empresas registradas en la base de datos.

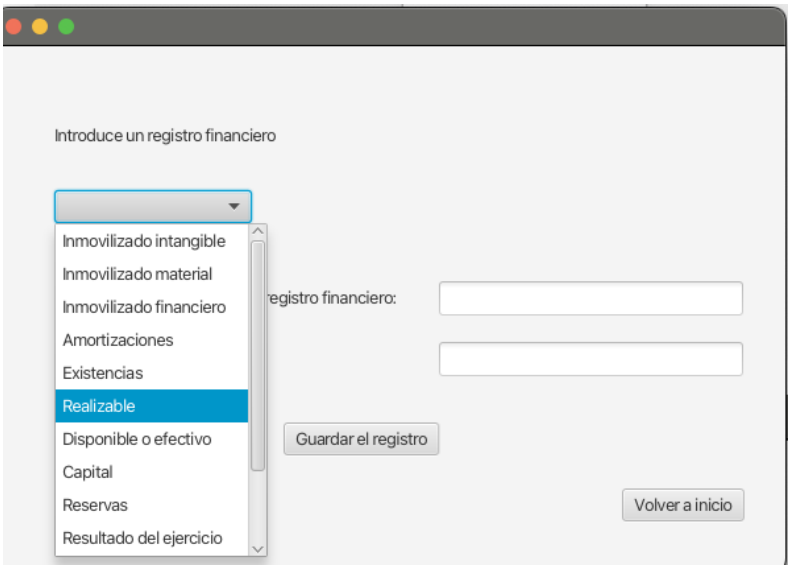
Cargada la empresa, accedemos el menú principal de gestión de registros del programa.

MenuEmpresaRegistro.fxml



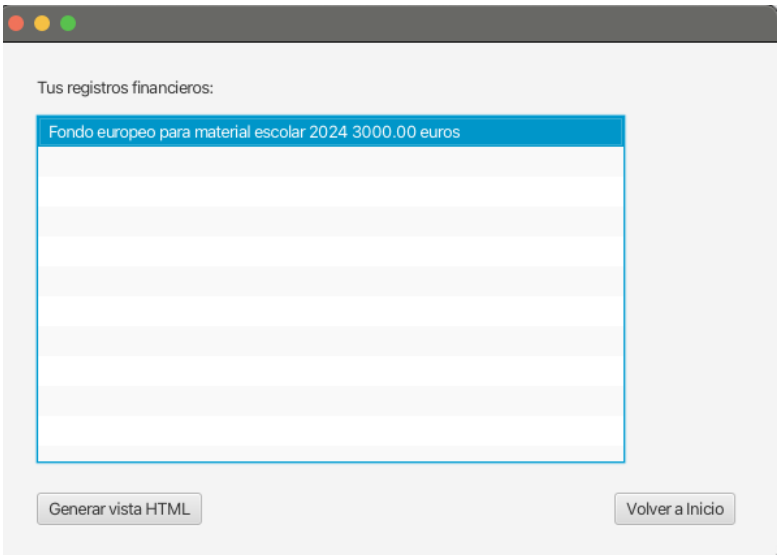
MenuEmpresaRegistro es una ventana algo más compleja, desde aquí el usuario puede escoger entre ver los registros asociados a su empresa, añadir un registro o eliminar un registro. Sin más dilación, voy a mostrar como crear un registro nuevo en nuestra empresa, para lo que pulsamos sobre el botón "Añadir registros financieros".

NuevoRegistroFinanciero.fxml



En esta pantalla aparece un ComboBox que permite elegir al usuario el tipo de registro al que irá asociado el registro financiero que desea crear, estando este ligado a alguna de las **más de 20** subclases creadas en el **modelo.** Una vez hecho esto, se añade un concepto y una cantidad económica, tras lo cual se pulsa en "Guardar el registro" para guardar el registro en la base de datos.

RegistrosFinancieros.fxml

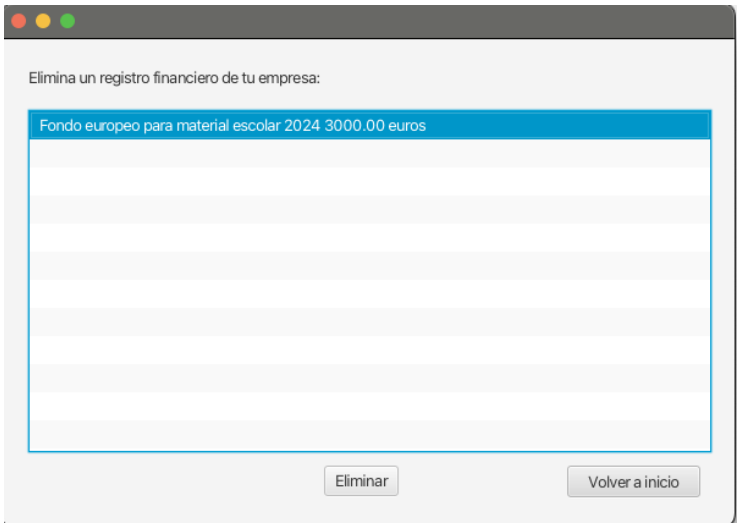


Tras haber creado el registro financiero en la pantalla anterior, se puede ir a RegistrosFinancieros pulsando sobre el botón "Ver registros financieros" en MenuEmpresaRegistro.fxml. RegistrosFinancieros.fxml es una vista simple, que muestra un ListView con los registros creados, junto al importe económico del registro.

Como cité anteriormente, este programa combina JavaFX y utilidades básicas de **ThymeLeaf** a través del método **generateHtmlView,** por lo cual, tras visualizar el ListView con los registros financieros en JavaFX, he añadido la opción de pulsar sobre el botón "Generar vista HTML", con esto el usuario genera un documento HTML con los registros financieros creados de forma sencilla y rápida.

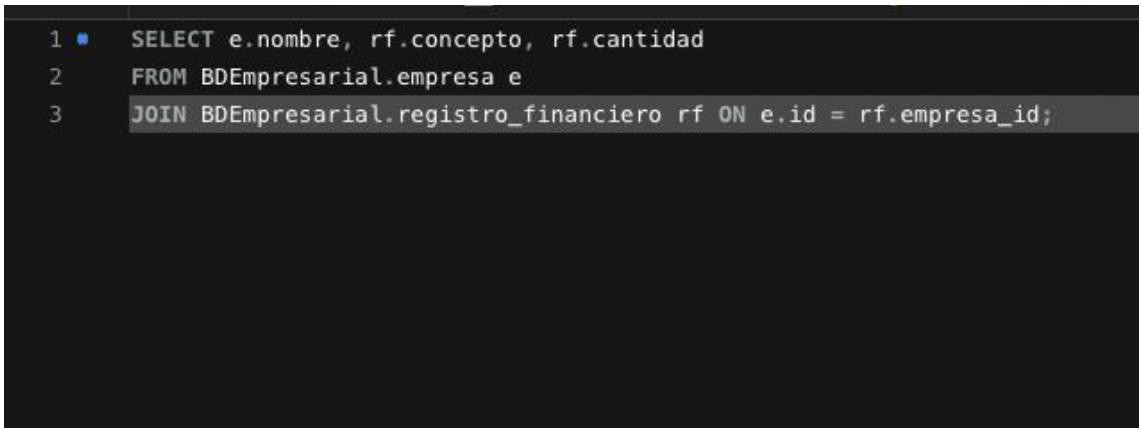
EliminarRegistro.fxml

Para eliminar un registro, podemos volver de nuevo a MenuEmpresaRegistro.fxml y desde ahí pulsar sobre el botón “Eliminar un registro financiero”, tras lo cual aparecerá un ListView que muestra los registros existentes acompañados de un botón eliminar que manda la orden a la base de datos de eliminar el registro seleccionado en el ListView. Para nuestro caso no eliminaremos el registro, sino que iré a MySQL Workbench para enseñar las bases de datos creadas con sus correspondientes registros financieros.



Acceso a la base de datos desde MySQL Workbench

Una vez en MySQL Workbench, podemos introducir una sentencia sencilla con un **JOIN** entre empresa y registro para mostrar las empresas creadas y sus correspondientes registros, como puede verse a continuación en las siguientes dos imágenes.



Copinsa	Deuda de un cliente para con la empresa	500.00
Copinsa	Ayudas de la Diputación de Cádiz	650.00
Game	Final Fantasy VII Rebirth	79.95
Cantina2324	50 litros de Coca Cola	40.00
Rafael Alberti	Fondo europeo para material escolar 2024	3000.00

5. CONCLUSIONES DEL PROYECTO Y ASPECTOS CONSEGUIDOS. POSIBLE EXPANSIÓN DEL PROYECTO.

Durante el desarrollo del proyecto he querido poner a prueba diversas tecnologías que hemos estado trabajando a lo largo del curso en la asignatura. Creo que el trabajo, aunque no recoge todas las tecnologías vistas, sí que es un esbozo de lo que se puede hacer con muchas de estas. El acceso a datos desde una aplicación que usa la máquina virtual de Java, con la ventaja multiplataforma que supone, está más que conseguido. He disfrutado usando **tecnologías nuevas**, y desafiándome a ir más allá en lo que al acceso a datos se refiere, ya que **combinar** Spring, con JavaFX, con ThymeLeaf, cada una con sus **ciclos de vida diferentes**, supone un **gran trabajo de estructura** y arquitectura limpia.

De cara a un futuro, el trabajo siempre puede crecer a niveles de crear una interfaz algo más compleja y visual que permita al usuario tener más opciones, o incluso crear tablas con los diferentes ejercicios y desarrollar balances de forma automática, a través de cálculos manejados por el propio programa.