CICLO FORMATIVO DE GRADO SUPERIOR DESARROLLO DE APLICACIONES MULTIPLATAFORMA

METALSDASHBOARD

Tutor: Enrique Prieto Gómez
Alumno: Juan Manuel Ramos Villarreal
EE. SS. Mª Auxiliadora
SEVILLA Curso 2023-2024

INDICE

APARTADO *PAGINA*

1- INTRODUCCION

El presente trabajo se centra en el desarrollo de un cuadro de mandos interactivo para la visualización y el análisis de mercado sobre las distintas tendencias en el mercado de los metales, concretamente nos basaremos en la cotización del oro, plata, paladio, platino y cobre. Este proyecto está dirigido a empresas y profesionales del sector financiero e inversores que necesiten una alternativa a la oferta existente sobre el uso de estas aplicaciones, ya que se podrá customizar a petición del cliente que soliciten estos servicios. Los servicios básicos que ofrece es la visualización de esas cotizaciones en tiempo real para rflejar las fluctuaciones de mercado en las distintas temporalidades .

La solución propuesta está diseñada para operar en un entorno web, utilizando tecnologías modernas como Spring Boot para la API REST y Studio para el front-end entre otras. La aplicación ofrece un panel sencillo e intuitivo con distintas herramientas para navegar por la graficar, tomar capturas, detalles de cada vela, etc

2- MODELADO DE LA SOLUCION

Para la implantación, gestión y mantenimiento se necesitarán los siguientes recursos:

-Necesitaremos una persona con conocimientos generales de sql, especializada en spring framework y en lenguaje R para gestionar y mantener la aplicación. Este servicio se podrá obtener como freelance sin necesidad de contratar a una persona o equipo que lo gestione y que por supuesto, tendrá su respectivo costo de mantenimiento.

-En materia de hardware, necesitaremos dos servidores. Uno para producción y otro para desarrollo. Se recomientda que el servidor de producción sea Instancia D2s v3 (2 vCPUs, 8 GB RAM) de microsoft azure o equivalente. Saldría por unos 75€ aproximadamente. Para el servidor de desarrollo, se recomienda Instancia B2s (2 vCPUs, 4 GB RAM) con un coste de 25€. Esto se debe a que en desarrollo solo se harían pruebas a menor escala.

-En materia de software, necesitaremos los siguientes componentes:

Spring Boot: Framework para desarrollar la API REST.
#Maven: Herramienta de gestión de dependencias y construcción de proyectos.
#Rstudio: Herramienta de desarrollo para el front.
#Base de datos: SQLDeveloper valdría para como sistema de gestión de bases de datos.
#Github: Control de versiones
#Servidores de aplicacion: Tomcat para desplegar aplicaciones Java.

Todas estas aplicaciones son gratuitas.

A todos estos costes, hay que añadir la suscripción a metalDEV, que es la API que proporciona la información. El precio dependerá del número de peticiones mensuales que queramos realizar y del valor de actualización del mercado, siendo el paquete básico unos 20€ al mes.

3- EJECUCION DEL PROYECTO

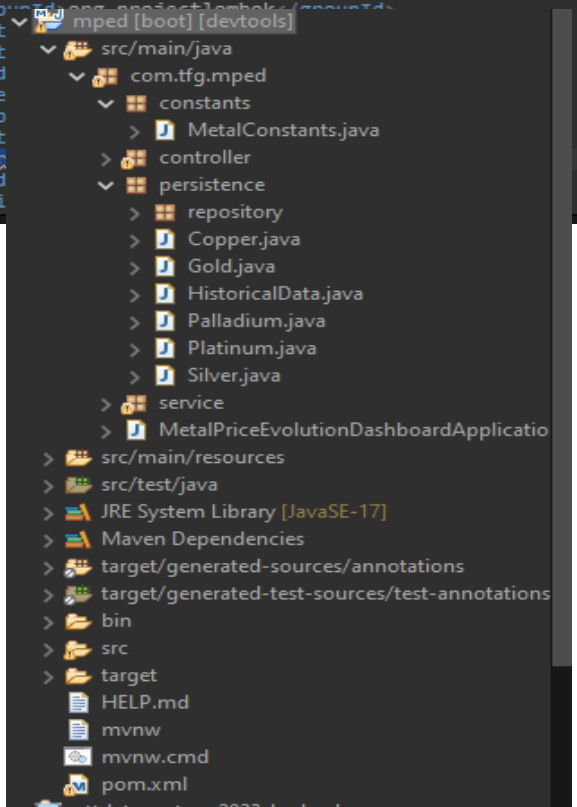
El proyecto ha sido desarrollado, fundamentalmente en dos tecnologías: Spring y Shiny. Empezaré primero describiendo todos los detalles del back.

3.1- ARQUITECTURA BACK-END

En lo referente a Spring, ha sido utilizado para el desarrollo del back-end de la aplicación. Además, incluye Hibernate y JPA para la gestión de de la explotación de datos y la comunicación con la base de datos. También se ha incorporado maven como gestor de proyectos y dependencias junto con las siguientes librerías para la configuración del proyecto:

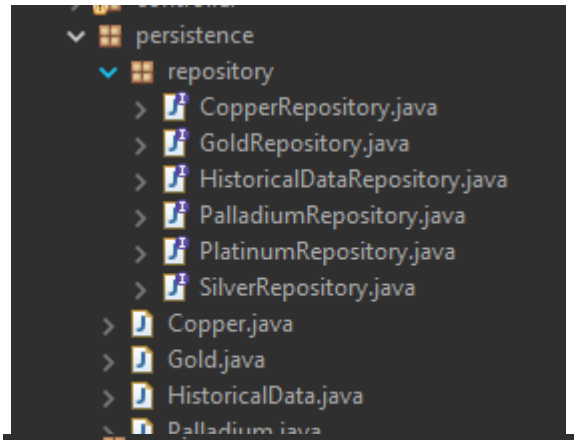
```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <scope>runtime</scope>
        <optional>true</optional>
    </dependency>
    <dependency>
        <groupId>com.mysql</groupId>
        <artifactId>mysql-connector-j</artifactId>
        <scope>compile</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
```

El código está basado en el modelo vista controlador con la siguiente estructura:

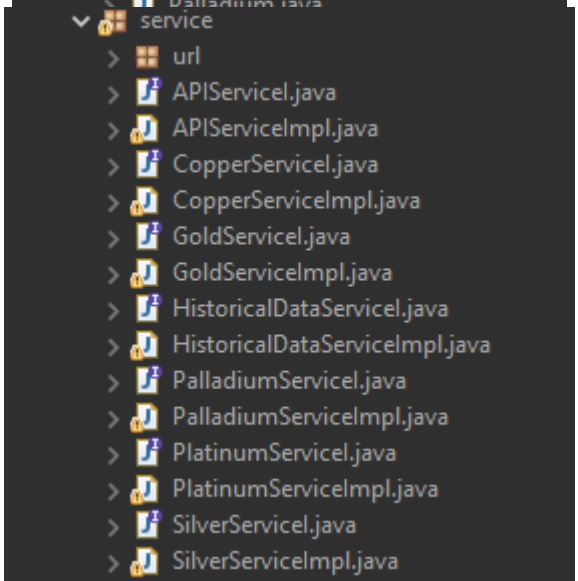


Los componentes del sistema son:

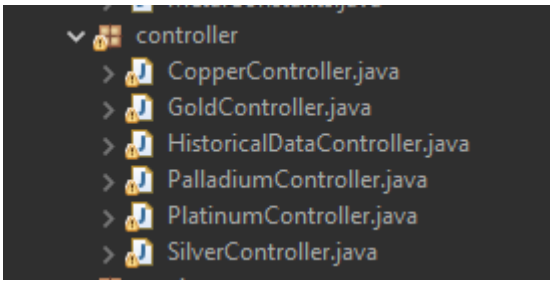
Persistence: Ahí se crea todas las entidades que van a estar en la base de datos y los repositorios, gestionados por JPA para la gestión de la base de datos.



Service: Aquí está toda la lógica de negocio de la aplicación, métodos de comprobación, etc



Controller: En esta carpeta de sedarrollará todos los endpoints necesarios para la comunicación entre el front y el back de la aplicacion.



3.2- ARQUITECTURA FRONT-END

En lo referente a shiny, ha sido utilizado para el desarrollo del front-end de la aplicación. Se ha empleado una única clase con la intención de hacerlo lo más simple posible ya que solo debe pintar el resultado de la petición en la gráfica.

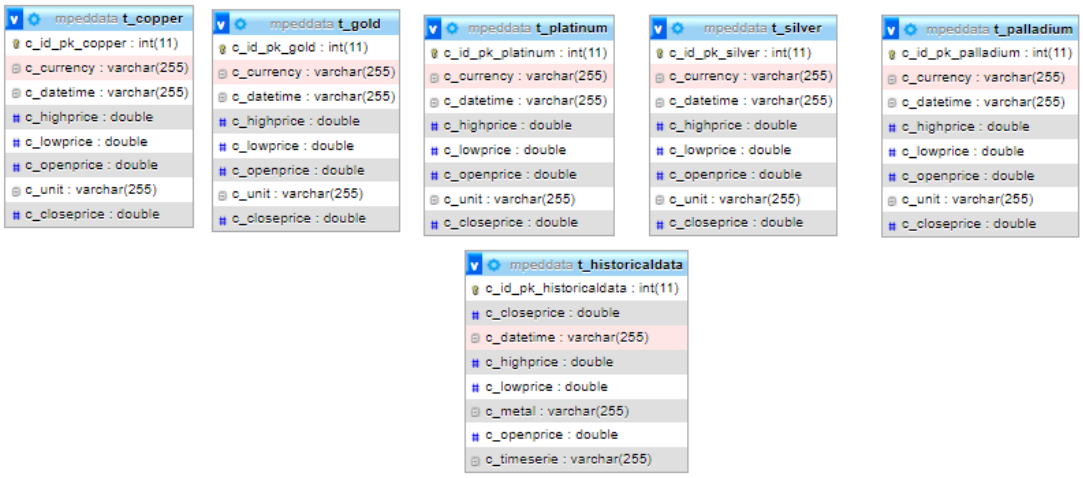
```
  # Application title
  titlePanel("Metal Dashboard"),

  # Sidebar layout
  sidebarLayout(
    sidebarPanel(
      selectInput("metal_select",
                  "Select Metal:",
                  choices = c("Copper", "Gold", "Palladium", "Platinum", "Silver"),
                  selected = "Copper"),

      # Botonera
      fluidRow(
        column(12,
               actionButton("H1_button", "H1"),
               actionButton("H4_button", "H4"),
               actionButton("D_button", "D"),
               actionButton("S_button", "S")
        )
      )
    ),

    # Main panel
    mainPanel(
      plotlyOutput("distPlot")
    )
  )
)

# Define server logic required to draw the plot
server <- function(input, output, session) {...}

# Run the application
shinyApp(ui = ui, server = server)


# Run the application
shinyApp(ui = ui, server = server)
```

Las librerias utilizadas en R han sido las siguientes:

```
# Cargar las librerías
library(shiny)
library(plotly)
library(jsonlite)
library(httr)
library(ggplot2)
library(dplyr)
```

3.3- ARQUITECTURA DE LA BASE DE DATOS

Para la base de datos se ha utilizado PHPMYADMIN a través de XAMPP. La base de datos tiene el siguiente modelo entidad relación:



Cada metal tiene su propia tabla para guardad los registros correspondientes. Aparte de ello, hay una tabla más “t_historicaldata” donde almacena todas las cotizaciones históricas de los cinco metales en varias temporalidades.

El esquema de la base de datos es creado a través de hibernate:

```java
@Entity
@Table(name = "T_GOLD")
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class Gold implements Serializable {

    /** Serial Version */
    private static final long serialVersionUID = 1L;

    /** Identificador (PK) */
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "C_ID_PK_GOLD")
    private int id;

    @Column(name = "C_DATETIME", nullable = false)
    private String datetime;

    /** Precio de apertura */
    @Column(name = "C_OPENPRICE", nullable = false)
    private Double openPrice;

    /** Precio de cierre */
    @Column(name = "C_CLOSEPRICE", nullable = false)
    private Double closePrice;

    /** Precio máximo */
    @Column(name = "C_HIGHPRICE", nullable = false)
    private Double highPrice;

    /** Precio mínimo */
    @Column(name = "C_LOWPRICE", nullable = false)
    private Double lowPrice;

    /** Divisa */
    @Column(name = "C_CURRENCY", nullable = false)
    private String currency;

    /** Unidad */
    @Column(name = "C_UNIT", nullable = false)
    private String unit;
}
```

Ejemplo de la entidad oro

```java
@Entity
@Table(name = "T_HISTORICALDATA")
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class HistoricalData implements Serializable{


    /** Serial Version */
    private static final long serialVersionUID = 1L;

    /** Identificador (PK) */
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "C_ID_PK_HISTORICALDATA")
    private int id;

    @Column(name = "C_METAL", nullable = false)
    private String metal;

    @Column(name = "C_DATETIME", nullable = false)
    private String datetime;

    /** Precio de apertura */
    @Column(name = "C_OPENPRICE", nullable = false)
    private Double openPrice;

    /** Precio de cierre */
    @Column(name = "C_CLOSEPRICE", nullable = false)
    private Double closePrice;

    /** Precio máximo */
    @Column(name = "C_HIGHPRICE", nullable = false)
    private Double highPrice;

    /** Precio mínimo */
    @Column(name = "C_LOWPRICE", nullable = false)
    private Double lowPrice;

    /** Temporalidad */
    @Column(name = "C_TIMESERIE", nullable = false)
    private String timeSerie;
}
```

Ejemplo de la entidad datos históricos

3.4- CONFIGURACION DE LA BASE DE DATOS

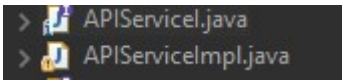
```
spring.datasource.url=jdbc:mysql://localhost:3306/mpeddata
spring.datasource.username=root
spring.datasource.password=

spring.jpa.database-platform=org.hibernate.dialect.MySQL57Dialect

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true

server.port=8090
```

4- Flujos de la aplicacion

Se puede dividir en dos: primero el flujo para extraer los datos de la API hasta el guardado en la base de datos, y después la solicitud desde el front hasta que pinta la grafica.

4.1- FLUJO API-BASE DE DATOS

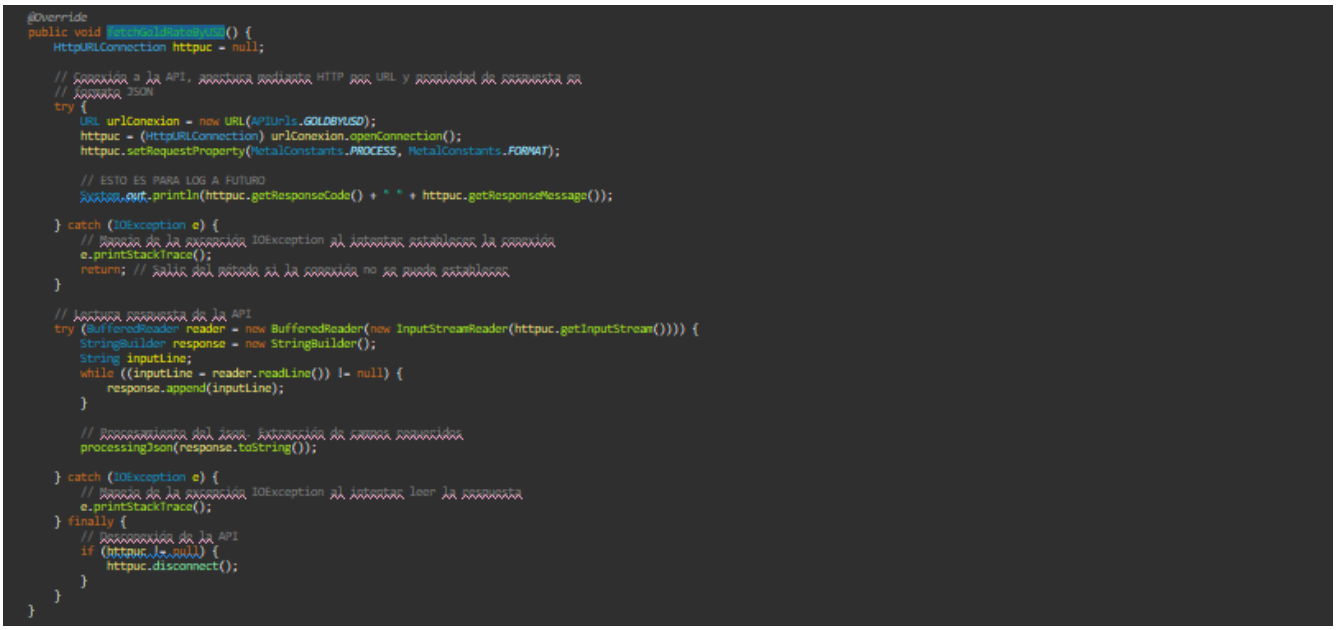
Todo ocurre en estas clases:



Cada diez minutos se ejecuta automáticamente el siguiente método:

```
/**
 * Método que arranca el proceso de automatización de conexiones para cotización
 * de metales.
 */
@Scheduled(cron = "0 0/10 * * 1-5 *")
public void fetchMetalRates() {

    fetchGoldRateByUSD();
    fetchCopperRateByUSD();
    fetchPalladiumRateByUSD();
    fetchPlatinumRateByUSD();
    fetchSilverRateByUSD();

}
```

En cada uno de los métodos fetch, se realiza una conexión a la API:



```
@Override
public void fetchGoldRateByUSD() {
    HttpURLConnection httpuc = null;

    // Conexión a la API, apertura mediante HTTP con URL y propiedad de respuesta en
    // formato JSON
    try {
        URL urlConexion = new URL(APIUrls.GOLDBYUSD);
        httpuc = (HttpURLConnection) urlConexion.openConnection();
        httpuc.setRequestProperty(MetalConstants.PROCESS, MetalConstants.FORMAT);

        // ESTO ES PARA LOG A FUTURO
        System.out.println(httpuc.getResponseCode() + " " + httpuc.getResponseMessage());

    } catch (IOException e) {
        // Manejo de la excepción IOException si ocurrió mientras se establecía la conexión
        e.printStackTrace();
        return; // Salir del método si la conexión no se pudo establecer
    }

    // Lectura secuencial de la API
    try (BufferedReader reader = new BufferedReader(new InputStreamReader(httpuc.getInputStream()))) {
        StringBuilder response = new StringBuilder();
        String inputLine;
        while ((inputLine = reader.readLine()) != null) {
            response.append(inputLine);
        }

        // Procesamiento del json. Extracción de campos requeridos
        processingJson(response.toString());

    } catch (IOException e) {
        // Manejo de la excepción IOException si ocurrió leer la respuesta
        e.printStackTrace();
    } finally {
        // Desconexión de la API
        if (httpuc != null) {
            httpuc.disconnect();
        }
    }
}
```

Y un procesamiento de datos en el método “processingJson”. En este mismo método, se añade la información a la base de datos:

```java
*/
public void processingJson(final String json) {

    // Parseo de la información
    Gson gson = new Gson();
    JsonObject jsonObject = gson.fromJson(json, JsonObject.class);

    // Extraer los campos relevantes del JsonObject.
    String datetime = jsonObject.has("timestamp") ? jsonObject.get("timestamp").getAsString() : null;
    String currency = jsonObject.has("currency") ? jsonObject.get("currency").getAsString() : null;
    String unit = jsonObject.has("unit") ? jsonObject.get("unit").getAsString() : null;
    String metal = jsonObject.has("metal") ? jsonObject.get("metal").getAsString() : null;
    Double openPrice = jsonObject.getAsJsonObject("rate").has("price")
            ? jsonObject.getAsJsonObject("rate").get("price").getAsDouble()
            : null;
    Double highPrice = jsonObject.getAsJsonObject("rate").has("high")
            ? jsonObject.getAsJsonObject("rate").get("high").getAsDouble()
            : null;
    Double lowPrice = jsonObject.getAsJsonObject("rate").has("low")
            ? jsonObject.getAsJsonObject("rate").get("low").getAsDouble()
            : null;
    Double closePrice= 0.0;

    // LOG PARA LAS VARIABLES
    System.out.println("Timestamp: " + datetime + " Currency: " + currency + " Unit: " + unit + " Metal: " + metal
            + " Openprice: " + openPrice + " highPrice: " + highPrice + " lowPrice: " + lowPrice + "closePrice: " + closePrice);

    // Verificación de campos
    if (datetime != null && currency != null && unit != null && metal != null && openPrice != null
            && highPrice != null && lowPrice != null) {
        addPriceQuote(datetime, currency, unit, metal, openPrice, highPrice, lowPrice, closePrice);
    } else {

        // LOG DE ERROR. ALGUN CAMPO ES NULO
        System.out.println("Alguno de los campos es nulo. Realizar alguna otra acción.");
    }
}
```

Este es un ejemplo de flujo. Para el resto de los metales es exactamente igual.

4.2- FLUJO PETICION- VISUALIZACION

Desde el front, se realiza siempre cada 10 minutos una llamada a la API mped. Este es el método de ejecución que lo activa cuando se inicia:

```r
# Render the plot
output$distPlot <- renderPlotly({
  metal <- input$metal_select
  interval <- values$interval
  df_seg <- fetchData(metal, interval)
  req(df_seg)
  createPlot(df_seg, metal, interval)
})
```

La funcion fetchData se encarga de realizar dos peticiones. Una para extraer los datos históricos y otra para extraer los datos más actuales. El metal se le pasa a través de la selección de un desplegable y el intervalo a través de botones:

```
  url_hist <- paste0("http://localhost:8090/mped/metalHD/", tolower(metal), "/",interval)
  response_hist <- GET(url_hist)
  if (!http_error(response_hist)) {
    data_hist <- fromJSON(content(response_hist, "text", encoding = "UTF-8"))
    df_hist <- data.frame(
      datetime = as.POSIXct(data_hist$datetime,
                            format = ifelse(interval %in% c("h1","h4","D", "S"), "%d.%m.%Y", "%Y-%m-%dT%H:%M:%OSZ"),
                            tz = "UTC"),
      openPrice = data_hist$openPrice,
      closePrice = data_hist$closePrice,
      highPrice = data_hist$highPrice,
      lowPrice = data_hist$lowPrice
    )
  } else {
    showNotification("Error al obtener los datos históricos", type = "error")
    return(NULL)
  }
} else {
  df_hist <- NULL
}
```

Petición datos históricos

```
# Fetch current data
url <- paste0("http://localhost:8090/mped/", tolower(metal), interval, "/USD")
response <- GET(url)
if (http_error(response)) {
  showNotification("Error al obtener los datos", type = "error")
  return(NULL)
} else {
  data <- fromJSON(content(response, "text", encoding = "UTF-8"))
  if (length(data) == 0) {

    if (!(interval %in% c("h1","h4","D", "S"))){
      # Eliminating duplicates
      showNotification("No hay datos disponibles", type = "warning")
    }
```

Petición datos actuales

Vamos a suponer que se ha realizado para el oro en H1. Una vez que llega esa petición al controlador, llama al servicio correspondiente y en función de la temporalidad:

```java
@GetMapping("/goldh1/{currency}")
public ResponseEntity<List<Gold>> goldTimeSerieH1(@PathVariable String currency) {

    List<Gold> dataList = gServ.loadTimeSerieH1(currency);

    return new ResponseEntity<>(dataList, HttpStatus.OK);
}

/**
 * Metodo para devolver la cotizacion del oro en vela de 4 horas segun divisa
 *
 * @param currency
 * @return ResponseEntity
 */
@GetMapping("/goldh4/{currency}")
public ResponseEntity<List<Gold>> goldTimeSerieH4(@PathVariable String currency) {

    List<Gold> dataList = gServ.loadTimeSerieH4(currency);

    return new ResponseEntity<>(dataList, HttpStatus.OK);
}

/**
 * Metodo para devolver la cotizacion del oro en vela diaria segun divisa
 *
 * @param currency
 * @return ResponseEntity
 */
@GetMapping("/goldD/{currency}")
public ResponseEntity<List<Gold>> goldTimeSerieD(@PathVariable String currency) {

    List<Gold> dataList = gServ.loadTimeSerieD(currency);

    return new ResponseEntity<>(dataList, HttpStatus.OK);
}
```

En el método “loadTimeSerieH1” extrae todos los datos correspondiente a la tabla “T_GOLD” y los agrupa en grupos de seis, ya que todos los registros son cada diez minutos. Se va guardando en la lista “timeSerieH1”:

```java
@Override
public List<Gold> loadTimeSerieH1(String currency) {

    // Lista donde se almacena la informacion y variables
    List<Gold> timeSerieH1 = new ArrayList<>();
    Gold goldDataArray;
    int id = 0;
    Integer index = -1;

    // Resultados de la base de datos
    List<Gold> goldList = new ArrayList<>(gRepo.findByCurrencyOrderByDatetimeAsc(currency));

    // Recorre todos los resultados de la base de datos hasta encontrar el primero
    // que empieza en hora y obtener su indice.
    for (int i = 0; i < goldList.size(); i++) {
        Gold goldDataArrayInic = goldList.get(i);
        if (goldDataArrayInic.getDatetime().substring(14, 16).equals("00")) {
            index = i;
            break;
        }
    }

    // Si no se encuentra el registro, se manda una lista vacia
    if (index == -1) {

        return Collections.emptyList();
    }
```

```
// Recorrido de los resultados de la base de datos
while (index < goldList.size()) {

    // control de las 6 iteraciones
    boolean isFirstIteration = true;
    boolean isLastIteration = false;
    Gold goldh1 = new Gold();

    // Recorrer los próximos 6 registros después del objeto con el indice deseado
    for (int in = index; in < index + 7 && in < goldList.size(); in++) {

        // obtención del objeto
        goldDataArray = goldList.get(in);

        // Realizar acciones específicas solo durante la primera iteración
        if (isFirstIteration) {

            // adjudicación de hora y precio apertura
            goldh1.setDatetime(goldDataArray.getDatetime());
            goldh1.setOpenPrice(goldDataArray.getOpenPrice());
            goldh1.setCurrency(goldDataArray.getCurrency());
            goldh1.setUnit(goldDataArray.getUnit());
            goldh1.setClosePrice(goldDataArray.getOpenPrice());

            if (goldh1.getHighPrice() == null) {
                goldh1.setHighPrice(goldDataArray.getHighPrice());
            }
            if (goldh1.getLowPrice() == null) {
                goldh1.setLowPrice(goldDataArray.getLowPrice());
            }

            if (goldDataArray.getHighPrice() > goldh1.getHighPrice()) {
                goldh1.setHighPrice(goldDataArray.getHighPrice());
            }
```

```
                // Cambiar el valor de isFirstIteration a falso después de la primera iteración
                isFirstIteration = false;
            }

            goldh1.setClosePrice(goldDataArray.getOpenPrice());

            if (goldDataArray.getHighPrice() > goldh1.getHighPrice()) {
                goldh1.setHighPrice(goldDataArray.getHighPrice());
            }

            if (goldDataArray.getLowPrice() < goldh1.getLowPrice()) {
                goldh1.setLowPrice(goldDataArray.getLowPrice());
            }

            // Realizar acciones específicas para la última iteración
            if (in == index + 6 || in == goldList.size() - 1) {

                // asignación precio de cierre
                goldh1.setClosePrice(goldDataArray.getOpenPrice());

                if (goldDataArray.getHighPrice() > goldh1.getHighPrice()) {
                    goldh1.setHighPrice(goldDataArray.getHighPrice());
                }

                if (goldDataArray.getLowPrice() < goldh1.getLowPrice()) {
                    goldh1.setLowPrice(goldDataArray.getLowPrice());
                }

                isLastIteration = true;
            }
```

```
                isLastIteration = true;
            }
            if (isLastIteration) {
                goldh1.setId(id);
                timeSerieH1.add(goldh1);
                isLastIteration = false;
                System.out.println(goldh1);
            }
        }
        index += 6;
        id++;
    }

    // Resultado del agrupamiento de datos en cada hora
    return timeSerieH1;
}
```

Luego, la API lo que devuelve es esa lista ficticia. Para H4, D Y S siguen la misma lógica.

Una vez que la petición llega a front, se convierte en un dataframe:

```r
df_hist <- data.frame(
  datetime = as.POSIXct(data_hist$datetime,
                        format = ifelse(interval %in% c("h1","h4","D", "S"), "%d.%m.%Y", "%Y-%m-%dT%H:%M:%OSZ"),
                        tz = "UTC"),
  openPrice = data_hist$openPrice,
  closePrice = data_hist$closePrice,
  highPrice = data_hist$highPrice,
  lowPrice = data_hist$lowPrice
)
```

Dataframe para la petición de datos historicos

```r
    df <- data.frame(
      datetime = as.POSIXct(character()),
      openPrice = numeric(),
      closePrice = numeric(),
      highPrice = numeric(),
      lowPrice = numeric()
    )
  } else {
    df <- data.frame(
      datetime = as.POSIXct(data$datetime, format="%Y-%m-%dT%H:%M:%OSZ", tz="UTC"),
      openPrice = data$openPrice,
      closePrice = data$closePrice,
      highPrice = data$highPrice,
      lowPrice = data$lowPrice
    )
  }
}
```

Dataframe para datos actuales

Puntualizar que si la API no develve ningún json porque no encuentre valores, se seguirá creando el dataframe pero vacio para que no muestre error y solo cargue los datos históricos en la gráfica.

Luego se hace una comparación entre los dos dataframes para que no haya repetición de datos y que en ese caso, se eliminen los datos actuales. Posteriormente, se combinan en un nuevo dataframe:

```r
if (interval %in% c("h1","h4","D", "S")){
  # Eliminating duplicates
  df_hist <- distinct(df_hist, datetime, .keep_all = TRUE)
}

# Combine historical and current data
if (!is.null(df_hist)) {
  df <- rbind(df_hist, df)
}
```

Y en este observe se pinta la gráfica con los datos finales:

```
# Observer to update the plot data when the interval or metal selection changes
observe({
  metal <- input$metal_select
  interval <- values$interval
  df_seg <- fetchData(metal, interval)
  req(df_seg)
  plotlyProxy("distPlot", session) %>% updatePlotData(df_seg)
})
```

Eso es el flujo para todos los metales. Además el front tiene un método para repetir todo este procedimiento cada 10 minutos y así actualizar los datos de las gráficas

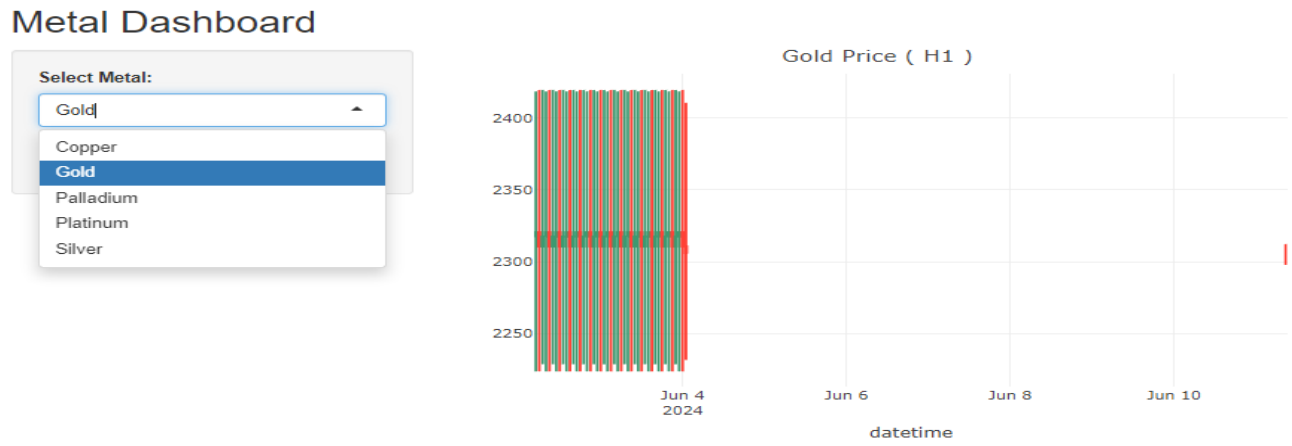
```
# Función para calcular el tiempo restante hasta el próximo intervalo de 10 minutos
calculateTimeLeft <- function() {
  current_time <- as.POSIXlt(Sys.time())
  minutes_left <- (10 - current_time$min %% 10) %% 10 # minutos restantes hasta el próximo intervalo de 10 minutos
  seconds_left <- (60 - current_time$sec) %% 60 # segundos restantes hasta el próximo minuto
  time_left <- minutes_left * 60 + seconds_left # tiempo total restante en segundos
  return(time_left)
}

# Temporizador reactivo para actualizar cada 10 minutos en los minutos 00, 10, 20, 30, 40, 50
autoUpdate <- reactiveTimer(calculateTimeLeft() * 1000)
```

```
  # Observer to update the plot data periodically
  observe({
    autoUpdate()
    metal <- isolate(input$metal_select)
    interval <- isolate(values$interval)
    df_seg <- fetchData(metal, interval)
    req(df_seg)
    plotlyProxy("distPlot", session) %>% updatePlotData(df_seg)
  })
}
```

5- MANUAL DE USUARIO

Interfaz de usuario

Esto es lo que se visualiza cuando se ejecuta la aplicación. Los precios de la gráfica se actualiza automáticamente cada diez minutos. Para cambiar de metal, se selecciona en el combobox el metal que se quiere, por ejemplo la plata:



En este caso, como no hay datos actuales aparece ese mensaje de advertencia y la gráfica vacia. Pero si nos vamos a la temporalidad D:

La aplicación tiene por defecto abrir el cobre en H1. Sobre las funcionalidades de la gráfica tenemos la siguiente barra:

Con opción de sacar una captura, ampliar o desampliar, reajustar la gráfica, hacer una selección de una región… entre otras.

Además, al situarse sobre una vela nos da toda la información necesaria:

6- CONCLUSIONES FINALES

Se ha finalizado el sistema tal y como se pretendió y se ha cumplido con todos los objetivos marcados salvo la incorporación de la serie temporal mensual.

Debido a que esto es un prototipo, tengo una serie de mejoras para incluir en el sistema a futuro donde lo primero que habría que realizar es una refactorización del back para un mejor rendimiento. Después vendrían las siguientes mejoras.

-Incorporación de time serie mensual, y 15 minutos.
-Posibilidad de ambiar la gráfica a gráfico de líneas.
-Agregar distintas divisas además del dolar
- Agregar sonido cada vez que aparece una nueva vela
- Agregar herramientas de dibujo para hacer análisis técnico

7- BIBLIOGRAFIA

Documentación API MetalsDEV: https://metals.dev/docs

Orientación sobre la visualización: https://es.investing.com/commodities

Documentación sobre gráficas en RStudio: https://plotly.com/r/candlestick-charts

Documentación sobre aplicación web para front en Rstudio: https://shiny.posit.co/r/gallery

Documentación sobre anotaciones spring: https://programandoenjava.com/scheduled-en-spring-boot

Servidores microsoft azure: https://azure.microsoft.com/es-es/pricing/details/virtual-machines/series