App valores

Módulo: DAW **Proyecto:** appValores **Fecha de exposición:** 11/06/2021 **Alumno:** Jesús David Gutiérrez Delgado **Tutor:** Alfonso Jiménez Vílchez

[TOC]

1. Introducción

El objetivo de este proyecto es el desarrollo de una aplicación web capaz de almacenar y mostrar información de valores bursátiles y su evolución en el tiempo. La aplicación se podrá ejecutar en los principales navegadores así como en distintos dispositivos móviles. Su funcionalidad será muy básica en un primer momento por lo que será susceptible de mejoras en un futuro. Esta funcionalidad básica es lo que hace que vaya orientada al gran público o a personas con pocos o ningún conocimiento sobre infromática y que sólo dispongan de un dispositivo móvil o tablet.

2. Objetivos

El objetivo general es tener una app capaz de mostrar y comparar información relativa a valores bursátiles y divisas. Además, hacer que dicha información sea accesible por distintos sistemas. La información será recogida y almacenada por un sistema back-end, que también desarrollaremos, y que permitirá no sólo el acceso sino también su mantenimiento (CRUD).

De lo dicho anteriormente se deduce que los objetivos específicos a conseguir serán los siguientes:

- Desarrollar un Back-End que, mediante API, permita:
    - Hacer CRUD de mercados.
    - Hacer CRUD de valores bursátiles.
    - Hacer CRUD de divisas.
    - Añadir información de precios para un valor o divisa determinado en el tiempo.
    - Consultar el historial de precios para un valor o divisa determinado.
- Desarrollar un Front-End que, haciendo uso de la API:
    - Muestre una portada con los 5 valores que más suben y los que más bajan en cada mercado.

- Permita consultar el historial de un valor, mostrando gráficas con su evolución en 1 día, 5 días, una semana, un mes, 6 meses y un año. Se podrá consultar más de un valor a la vez.
- Permita definir una cartera de inversión (añadir cuántas acciones se dispone de cada valor) y muestre gráficas e información sobre la composición de la cartera, su precio y su historial.

3. Análisis previo.

Para la consecución de nuestros objetivos deberemos contar con un software capaz de acceder a un sistema remoto de almacenamiento donde tendremos los datos. Deberemos contar también con un gestor de BB.DD (SGBD) y las herramientas necesarias para interactuar con el. La interfaz con el usuario, o front-end, será independiente del sistema de gestión de los datos, de hecho tendrán alojamiento en distintos servidores.

La facilidad de mantenimiento, la seguridad y la interoperabilidad futura con otros sistemas son los motivos que nos han llevado a plantear una solución de este tipo. Veamos cada uno de ellos con más detalle:

- Mantenimiento: al estar dividido el sistema en dos partes bien diferenciadas, back-end y front-end, su mantenimiento y solución de incidencias en mucho más simple, evitando que el fallo puntual de alguna de sus funcionalidades paralice todo el sistema.
- Seguridad: ambos sistemas estarán alojados en servidores distintos, lo que reducirá a la mitad la posibilidad de que haya una pérdida total del sistema.
- Interoperabilidad: nuestro back-end podrá ser utilizado por futuras aplicaciones, no sólo la nuestra, sin necesidad de realizar cambios en la misma.

Para la consecucución de los objetivios esbozados en las líneas anteriores se podrían tener en cuenta las siguientes soluciones:

- Backend: posibilidades de uso de frameworks tipo Symfony (PHP), Angular(typescrpt) o Springboot(JAVA). Todos son de amplio uso y funcionalidades más que avanzadas, sin embargo, en aras de una mayor rapidez en el desarrollo y una curva de aprendizaje lo más pronunciada posible, utilizaremos Springboot.

- Frontend: aquí son también amplias las posibilidades en cuanto a la implementación de una interfaz de usuario, pero optaremos por una solución convencional y a la vez muy extendida: Javascript. Para estandarizar el desarrollo todo lo posible, en su parte de diseño y estilo optaremos por Bootstrap que nos aportará facilidad y rapidez a la hora de implementar el estilo de la web.

  Por otra parte, necesitaremos algún tipo de componente que nos permita implementar de forma rápida y eficaz las distintas gráficas que necesitaremos para mostrar al usuario la evolución temporal de valores, divisas, etc. El componente Chart.js es, a priori, el que presenta una mayor facilidad de uso. La integración con JScript es total y la faciclidad a la hora de implementar distintos tipos de gráficas "alimentadas" por datos proporcionados por nuestra API hacen que sea una opción más que factible.

4. Recursos

Para llevar a cabo este proyecto, necesitaremos contar con los siguientes recursos:

- Hardware: para el trabajo de desarrollo, propiamente dicho, necesitamos contar con un equipo capaz de hacer correr a la vez todos los recursos software que veremos en el siguiente punto. Se trata de un

equipo dotado de un procesador Intel i7 a 3Ghz de velocidad y con 32 Gb de RAM.

- Software: son varias las aplicaciones y herramientas que necesitaremos. Serían las siguientes:
    - S.O: Windows 10 Pro.
    - IDE: utilizaremos Eclipse en la versión 2019-12 para backend y Visual Studio Code para frontend
    - SGBD: MySql administrado desde la versión web de phpMyAdmin en local y desde Dbvisualizer cuando la BB.DD. está ya en el servidor remoto
    - Pruebas: Postman 8.1
    - Documentación: utilizaremos Typora como editor para Markdown así como LibreOffice Writter
- Repositorios: para la gestión documental y del desarrollo nos apoyaremos en Github. Nuestro repositorio principal de Github se llamará ProyectoFinDeCurso y contará con las siguientes ramas:
    - Main: resumen visual y bibliografía
    - Backend: objetos del proyecto en Spring y JPAl. Desde esta rama se realizará el despliegue en Heroku
    - Frontend: páginas y código en JavaScript. Desde esta rama se realizará el despliegue en Vercel
    - Doc: documentación asociada al proyecto, diagramas en formato drawio, etc.
    - Img: imágenes o gráficos explicativos
- Servicios online:
    - Despliegue back-end: Heroku
    - Despliegue front-end: Vercel
    - Despliegue SGBD: Gearhost

5. Desarrollo del proyecto.

5.1 Implantación y despliegue.

La implantación deberá ser realizada en las siguientes fases para ir comprobando el correcto funcionamiento antes de pasar a la siguiente fase.

1. BB.DD: crearemos nuestra base de datos en GearHost indicando que se trata de una BB.DD de MySql. Tomaremos la cadena de conexión que nos proporcione este alojamiento, así como el usuario y la contraseña que decidamos, para incorporarlos a los parámetros de configuración de nuestra API.
2. API: crearemos en Heroku nuestra aplicación dándola de alta y, utilizando la rama de backend de github, realizaremos el despliegue
3. Pruebas: comprobaremos mediante la batería de pruebas creadas en Postman si la API responde correctamente a nuestras peiticones. Las pruebas habrá que redirigirlas hacia la nueva URL que nos proporcione Heroku en vez de hacia la ubicación local de nuestra máquina.
4. Web: desde Github realizaremos el despliegue de la aplicación web en Vercel y comprobaremos si las llamadas Ajax a la API así como la respuesta de la página es la correcta.

   El siguiente esquema condensa la forma en la que se va a realizar el despliegue:

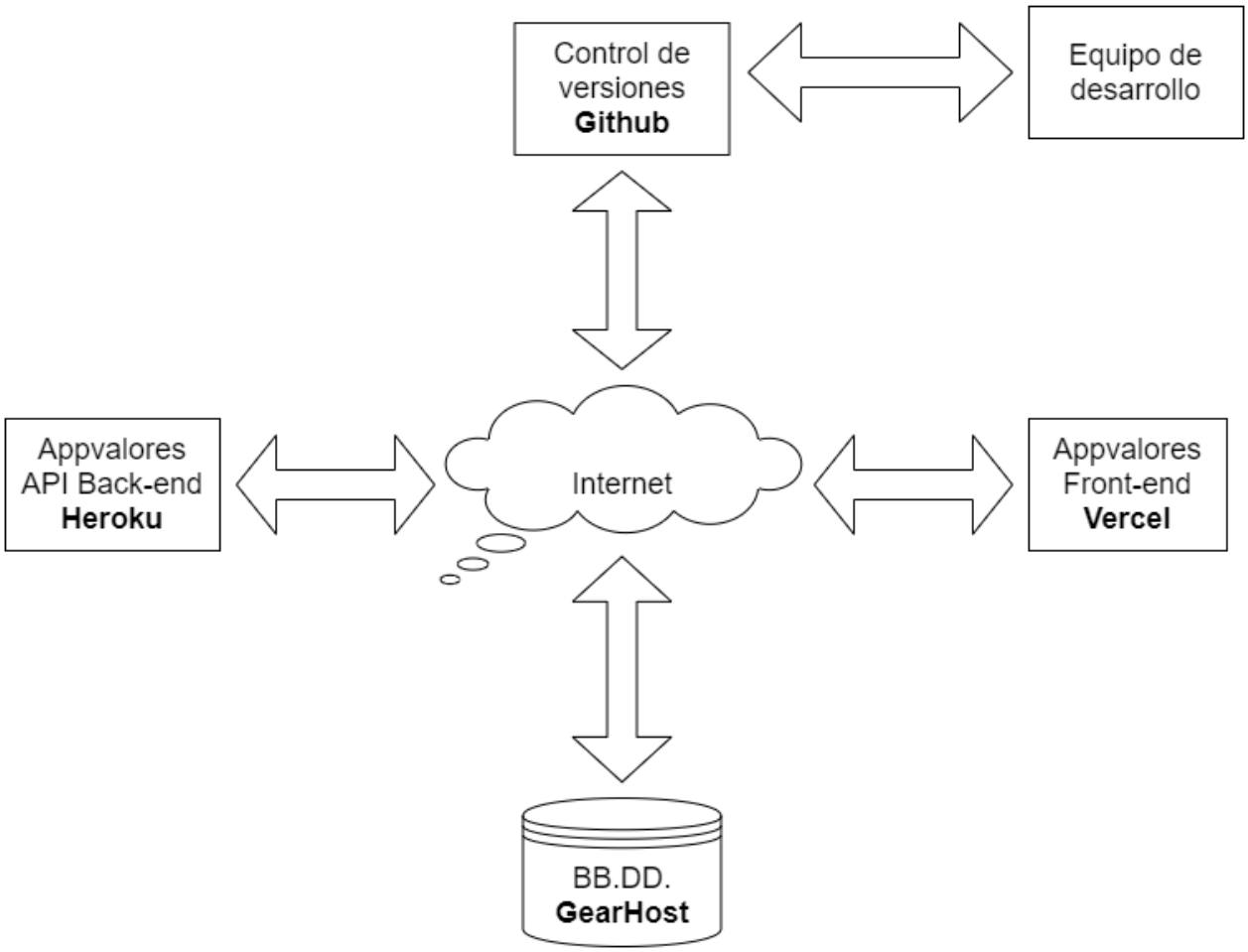
Despliegue Appvalores

Alumno: Jesús David Gutiérrez Delgado

Curso: 2020/2021

Asignatura: Proyecto fin de curso

Ejercicio: Proyecto final



5.2 Modelado de datos.

Al haber optado por JPA a la hora de gestionar y desarrollar nuestra capa de persistencia, la BB.DD. aparece reflejada como entidades que se relacionan entre sí utilizando etiquetas. De esta forma hemos evitado la creación de un script de creación propiamente dicho (create, alters, etc.) pero, por otra parte, hemos tenido que ser más cuidadosos a la hora del diseño de las entidades ya que son estas un reflejo de las tablas, como veremos más adelante. En cualquier caso, se ha diseñado un diagrama E/R como elemento previo y de ayuda para el diseño de las clases y su etiquetado.

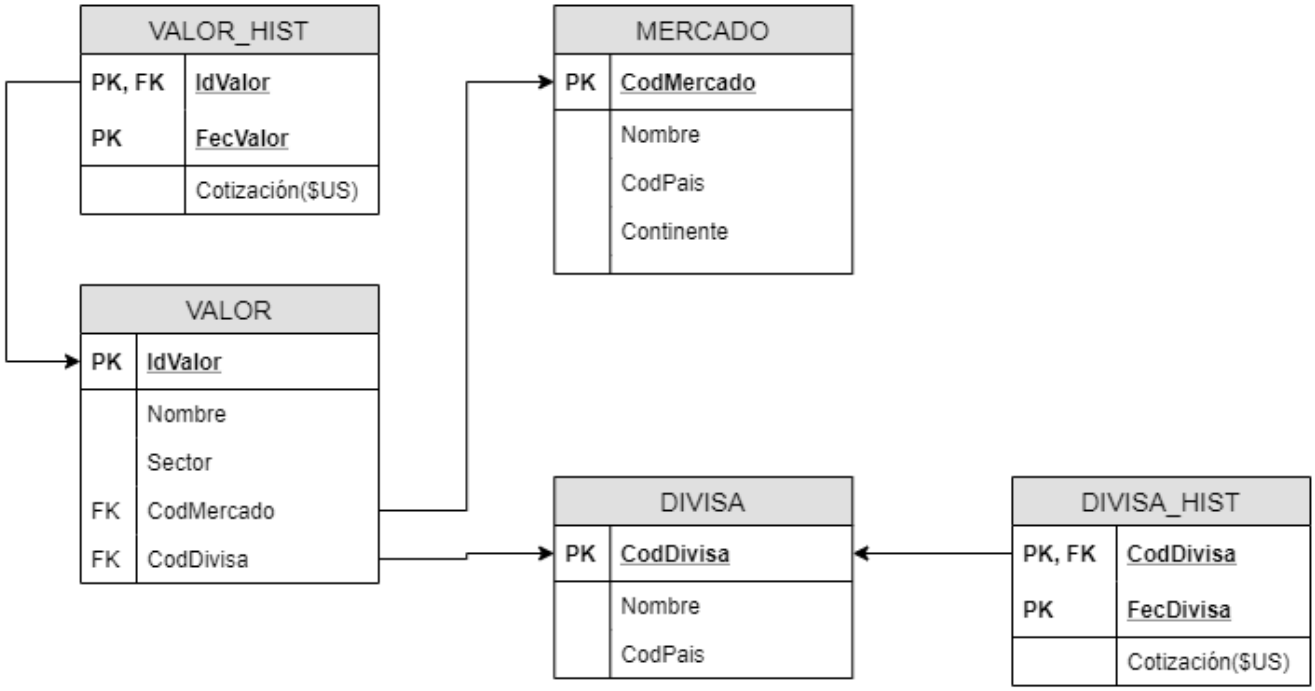
DAW-2

Alumno: Jesús David Gutiérrez Delgado

Curso: 2020/2021

Asignatura: Proyecto fin de curso

Ejercicio: Gestión de valores bursátiles

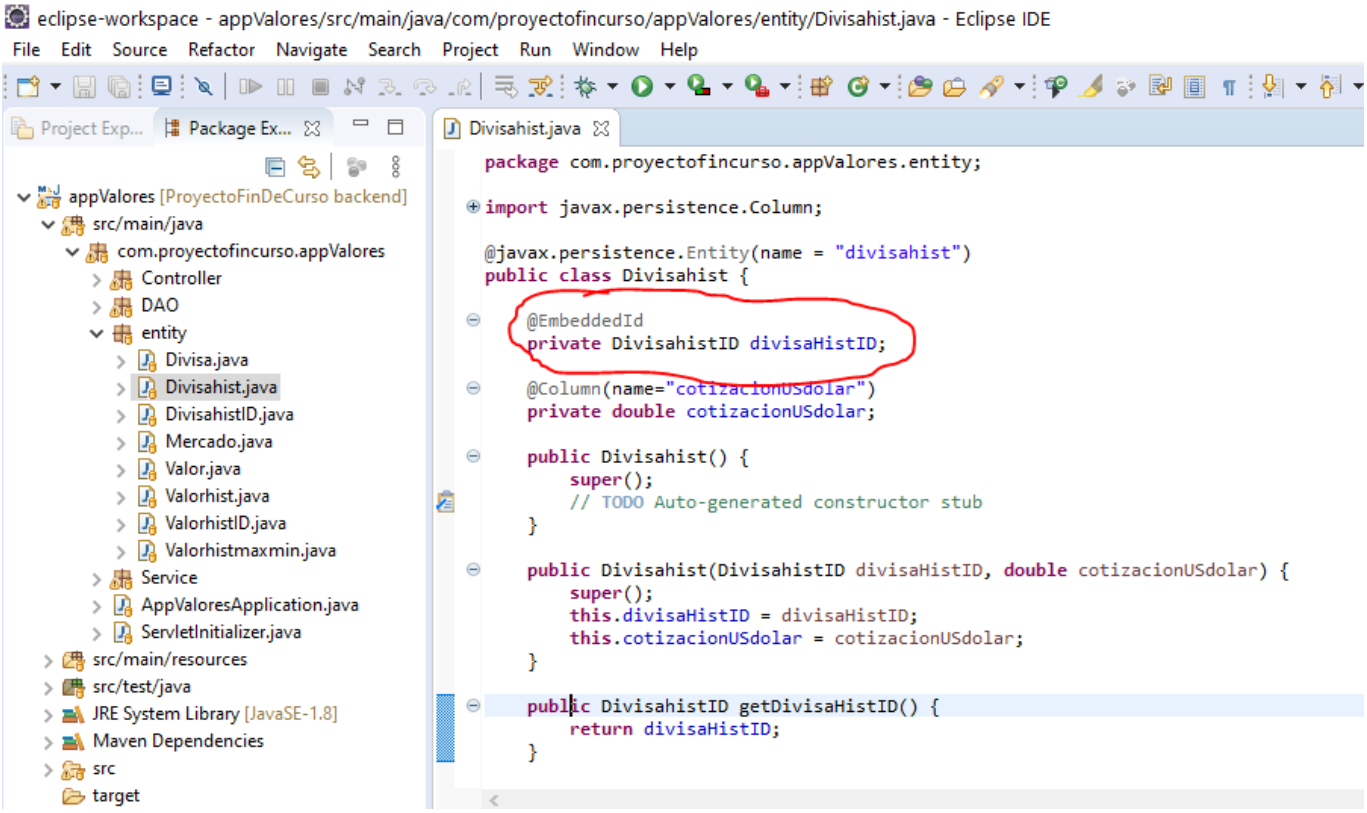


En base al proceso de modelización del punto anterior, podemos ver las siguientes entidades:

- Mercado: representa a los mercados en los que cotizan los distintos valores. Se compone de:
  - CodMercado: clave
  - Nombre.
  - CodPais: país al que pertenece.
  - Continente.
- Valor: es el valor que cotiza en un mercado. Se compone de:
  - Idvalor: clave
  - Nombre.
  - Sector: industrial, teleco, banca, etc.
  - CodMercado: mercado en el que cotiza.
  - CodDivisa: divisa en la que cotiza.
- Valor histórico: serían las distintas cotizaciones que ha sufrido el valor a lo largo del tiempo. Se compone de:
  - IdValor: clave
  - FecValor: clave, fecha de la cotización.
  - Cotización: en dolares US.
- Divisa: moneda en la que cotiza un valor. Se compone de:
  - CodDivisa: clave

    - Nombre.
    - CodPais: país al que pertenece
- Divisa histórica: distintas cotizaciones de la divisa a lo largo del tiempo. Se compone de:
    - CodDivisa: clave
    - FecDivisa: fecha de la cotización de la divisa.
    - Cotización: en dolares US.

Como ya veremos más adelante, uno de los principales problemas a la hora de trabajar con JPA es establecer relaciones a nivel de ID de clase. Esto nos ha llevado a la creación de entidades intermedias de tipo ID o clave primaria que necesitamos a la hora de establecer dichas relaciones.

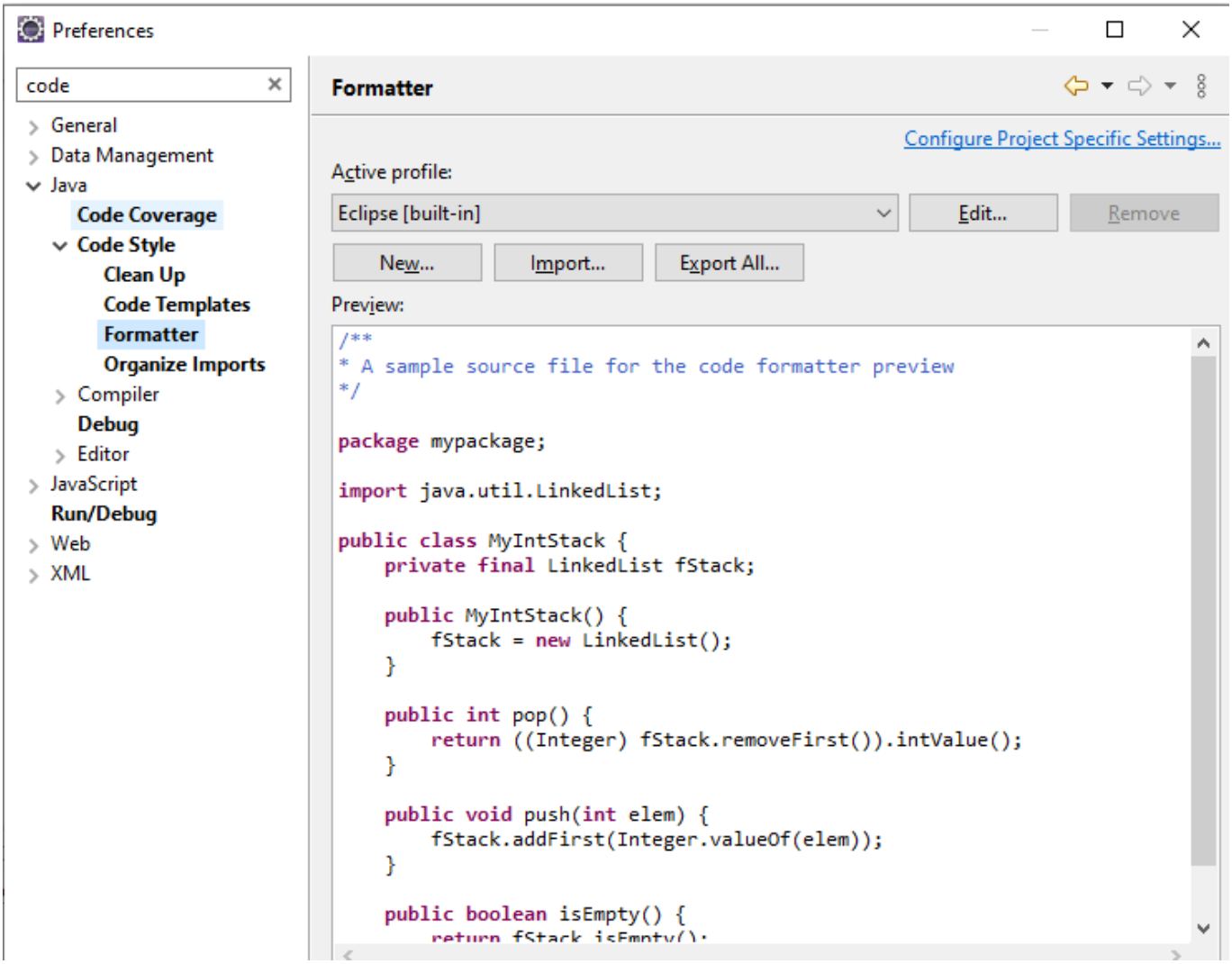


La clase ID que hemos tomado como ejemplo para ilustrar esta casuística consta básicamente con una referencia a otro objeto (Divisa) y una propiedad más para identificarla univocamente, en este caso fecha.

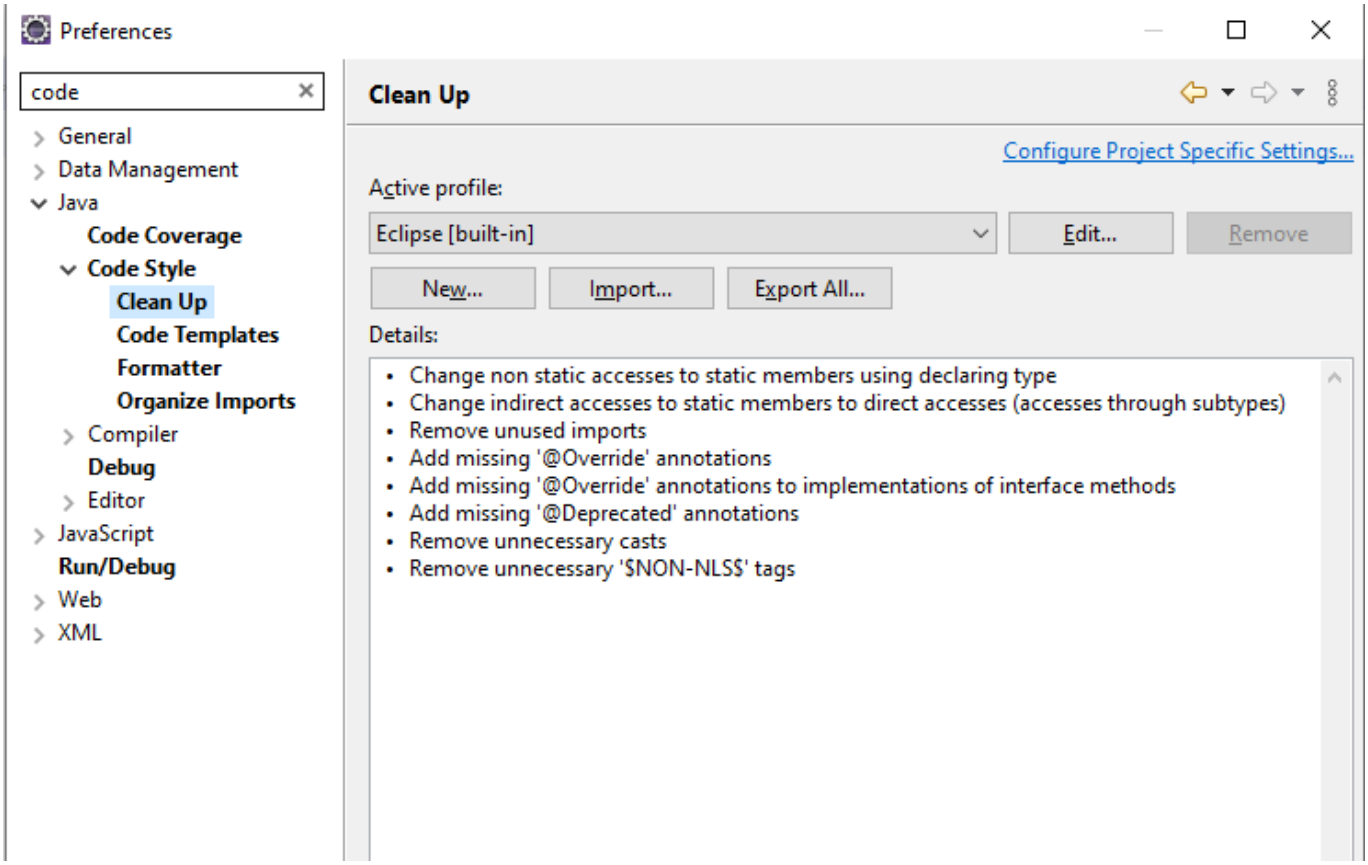
5.3 Programación.

5.3.1 Guía de estilo

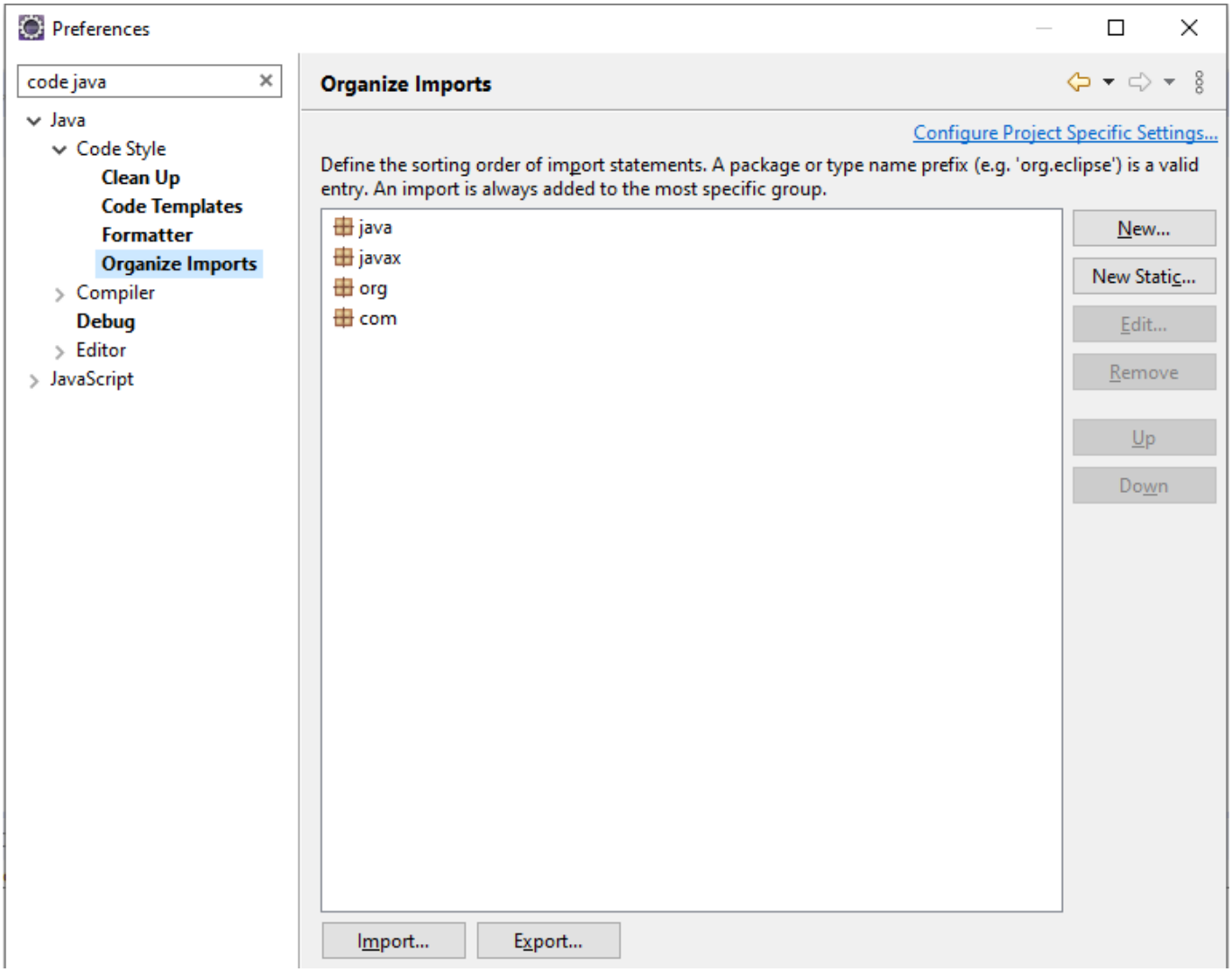
Eclipse nos proporciona opciones para personalizar, si lo deseamos, casi cualquier aspecto del proceso de codificación. En nuestro caso, hemos optado por seguir el estilo marcado por la herramienta, ya de por sí, es un estandar. Tabulaciones, apertura y cierre de llaves, organización y orden de librerías son sólo algunos de los aspectos que podemos parametrizar.



La limpieza de código es otro aspecto que podemos controlar automatizando determinadas tareas rutinarias.



Por último, y a modo de ejemplo, vemos que también es posible la organizació de los paquetes del desarrollo siguiendo una determinada pauta.



Por otra parte, señalamos a continuación las reglas de estilo y buenas prácticas que se han puesto en práctica durante el desarrollo y algunos ejemplos de los mismos en el código:

5.3.1.1 Declaraciones

Para la declaración de las variables se utiliza una declaración de cada vez y no se permiten dejar variables locales sin inicializar salvo en el caso de que sean propiedades de un objeto bean.

```
 @PostMapping("/loadFecdesdeFechasta/{fecDesde}/{fecHasta}")
 public Map<String, String> loadFecdesdeFechasta(@PathVariable String fecDesde,
@PathVariable String fecHasta)  throws ParseException {

     double start = 1.5;
     double end = 0.5;
     double random, result;

     DateFormat sourceFormat = new SimpleDateFormat("dd/MM/yyyy");

     DecimalFormatSymbols simbolos = new DecimalFormatSymbols();
```

```
        simbolos.setDecimalSeparator('.');
        DecimalFormat df = new DecimalFormat("#.00",simbolos);
```

5.3.1.2 Constantes

Como norma general todas las constantes numéricas no deberían codificarse directamente, salvo la excepción de -1, 0 y 1.

5.3.1.3 Propiedades

El acceso/modificación de las propiedades de una clase (no constantes) siempre mediante métodos de acceso get/set.

```
public Valor getValor() {
    return valor;
}

public void setValor(Valor valor) {
    this.valor = valor;
}

public Date getFecValor() {
    return fecValor;
}
```

La asignación de variables / propiedades no podrá ser consecutiva.

```
      inicio = sourceFormat.parse(fechaDesde);
      fin = sourceFormat.parse(fechaHasta);

      Date actual = inicio;

      // Recuperamos todos los valores
      List<Valor> listaValores;
      listaValores = valorService.findAll();
```

5.3.1.4 Métodos

Los métodos deberán ser verbos (en infinitivo), en mayúsculas y minúsculas con la primera letra del nombre en minúsculas, y con la primera letra de cada palabra interna en mayúsculas (lowerCamelCase).

No se permiten caracteres especiales.

El nombre ha de ser lo suficientemente descriptivo, no importando a priori la longitud del mismo.

```
 @DeleteMapping("valores/{valorHistId}/{fecha}")
 public String deleteValorHist(@PathVariable int valorHistId, @PathVariable String
fecha) {

        Valorhist valorHist = valorhistService.findById(valorHistId, fecha);

        if(valorHist == null) {
            throw new RuntimeException("Valor histórico desconocido
id:"+valorHistId);
        }

        valorhistService.deleteById(valorHistId,fecha);

        return "Valor histórico borrado con id - "+valorHistId + " y fecha
"+fecha;
  }
```

5.3.2 Organización del código

5.3.2.1 Backend.

El código del proyecto estará estructurado en cuatro paquetes básicos que nos darán una idea bastante clara de la jerarquía de los objetos que contienen. Son los siguientes:

- Entidades: representarán a la tupla de la BBDD y contendrán los constructores y métodos básicos de acceso a sus propiedades. En algunos casos también contendrá a las clases "clave" que nos servirán para identificar al objeto univocamente utilizando otro objeto contenido en el.
- Acceso a datos (DAO): estos objetos serán los responsables de interactuar con JPA y, utilizando los métodos necesarios, interactuar con la BBDD.
- Servicios: los servicios serán la herramienta o capa visible que utilizará el desarrollador para acceder a los datos e interactuar con ellos.
- Controladores: serán los que reciban las peticiones http y en función de las mismas, realicen la acción que se les solicite (GET, POST, PUT y DELETE). Serán la capa visible de cara al frontend.

  Además de estos paquetes básicos, tendremos también otros como el de recursos donde almacenaremos la parametrización de la conexión a BBDD.

5.3.2.2 Frontend.

Nuestra web constará de los siguientes elementos a nivel de desarrollo:

- Código JavaScript:
  - dashboard.js: contendrá la funcionalidad principla de acceso a la API, llamadas AJAX, así como el tratamiento de los datos recibidos.
  - objetos.js: contiene las clases creadas para el tratamiento de la información.

- Ventanas: una ventana principal, index, será el inicio de la navegación y donde se nos ofrecerán las distintas funcionalidades.

5.3.3 Arquitectura de clases

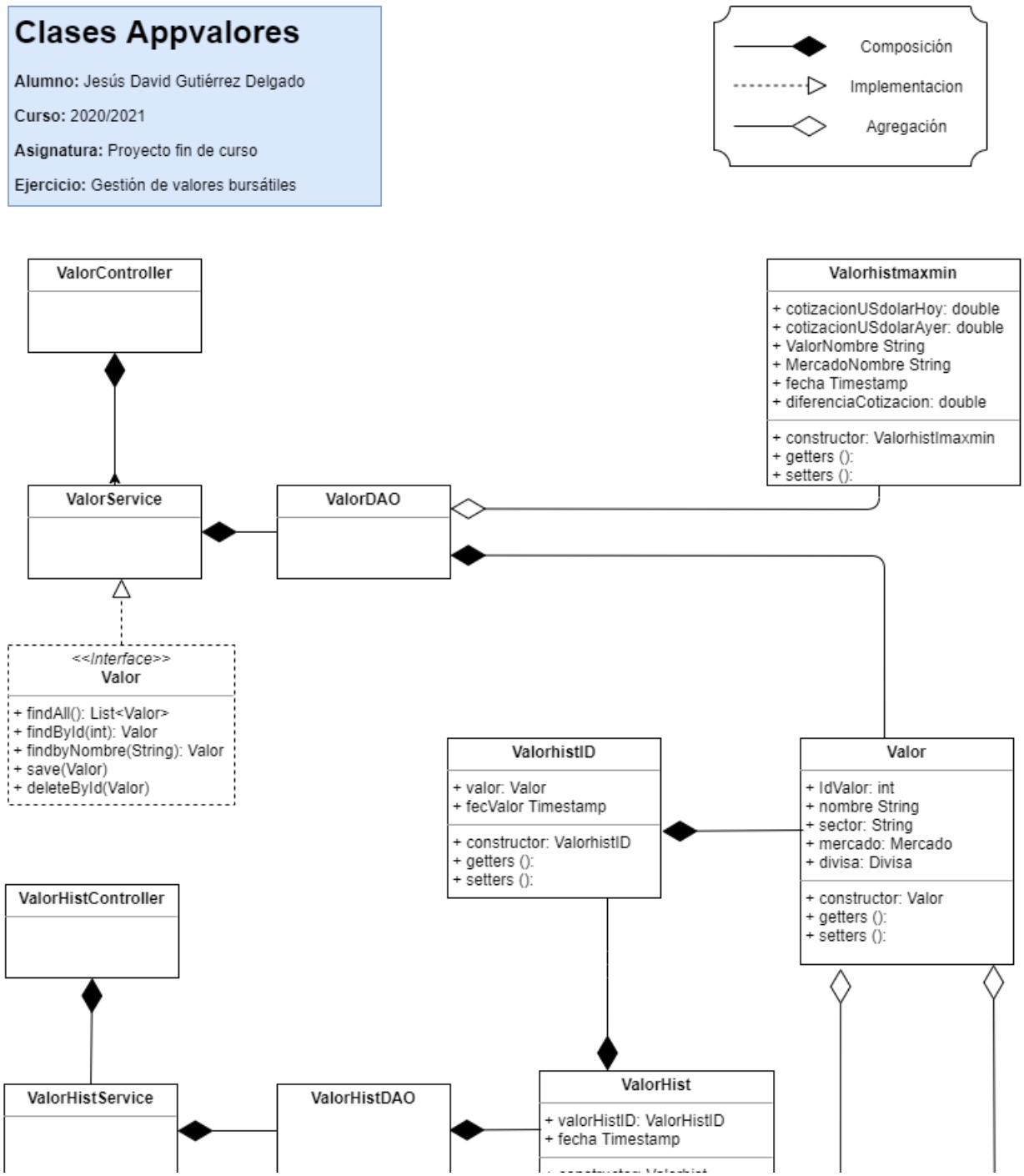
Nuestra arquitectura de clases se base en cuatro elementos básicos sobre los que se ha construido todo el sistema. En el diagrama de clases podemos ver más gráficamente cómo se estructuran las diferentes clases y cuales son sus propiedades y métodos. Algunas entidades son sólo de uso interno y como apoyo de otras, tal es el caso de los ID necesarios en algunos casos y cuyo tratamiento es un poco más complejo y específico. Los objetos serían los siguientes:

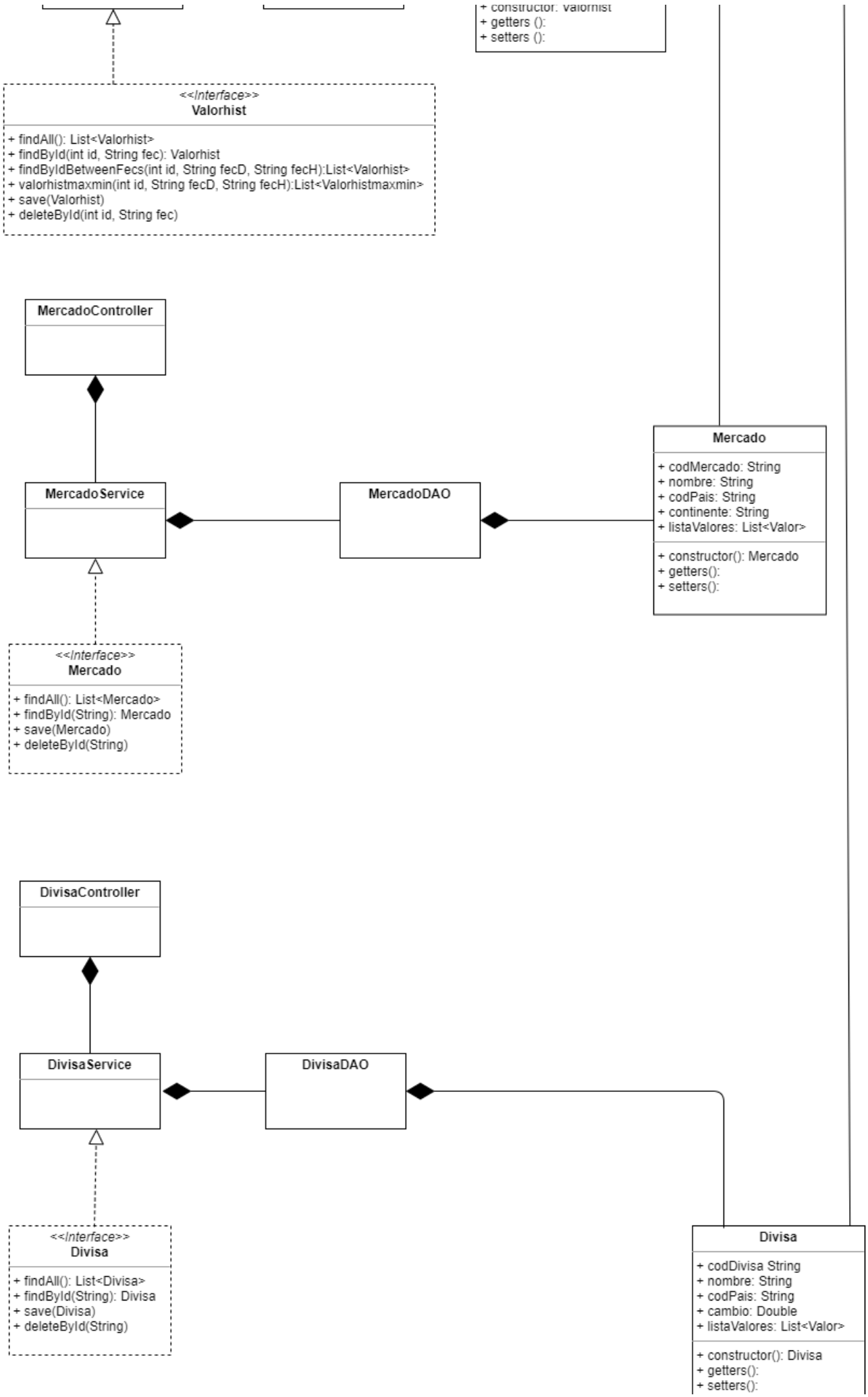
- Entidad: será nuestra vía para poder interaccionar con la tabla correspondiente en BB.DD. Sus propiedades o atributos serán los campos de la tabla. Contaremos con las siguientes entidades:
    - Divisa: este objeto será el encargado de almacenar la información relacionada con las divisas (moneda de un país) con las que puede operar el sistema.
    - Divisahist: representa los distintos valores, en dolares, que ha tenido la divisa en cuestión a lo largo del tiempo.
    - DivisahistID: clase creada como propiedad de la anterior y que actua como clave de la misma. Se utiliza a nivel interno por la arquitectura.
    - Mercado: sería cada una de las bolsas donde se gestiona un valor.
    - Valor: es la accion propiamante dicha que cotiza en un mercado a un precio en dólares.
    - Valorhist: al igual que en la divisa, este objeto representa las distintas cotizaciones que ha tenido un determinado valor a lo largo del tiempo.
    - ValorhistID: clase creada como propiedad de la anterior y que actua como clave de la misma. Se utiliza a nivel interno por la arquitectura.
    - Valorhistmaxmin: objeto creado a nivel de arquitectura como necesidad de mapear una consulta muy concreta. Estos objetos son necesarios debido a que las consultas gestionadas por Springboot sólo devuelven tipos "objeto" que hay que devolver a través del controlador.
- Controladores: son la puerta de entrada a la API. Reciben las peticiones y las enrutan según su tipo (GET, PUT.,etc) y nombre. Su función principal es recibir la petición HTTP y llamar al servicio correspondientes.
    - DivisaController: gestiona las llamadas a la API en relación con las divisas.
    - DivisaHistController: gestiona las llamadas a la API en relación con el histórico de divisas.
    - MercadoController: gestiona las llamadas a la API en relación con los mercados.
    - ValorController: gestiona las llamadas a la API en relación con los valores.
    - ValorhistController: gestiona las llamadas a la API en relación con el histórico de valores.
- Servicios: son el nexo entre el objeto de acceso a datos (DAO) y el controlador. Su misión es indipendizar ambas capas para permitir un mantenimiento más simple. Se ha generado una interfaz para cada uno de ellos en previsión de posibles cambios futuros, como así ha sido. Esto evita que haya que propagar el cambio realizado en la interfaz, debido a una funcionalidad específica del objeto, en todos los objetos afectados.
    - DivisaServiceImpl: gestiona las peticiones realizadas en relación a las divisas y las redirecciona a su objeto DAO correspondiente.

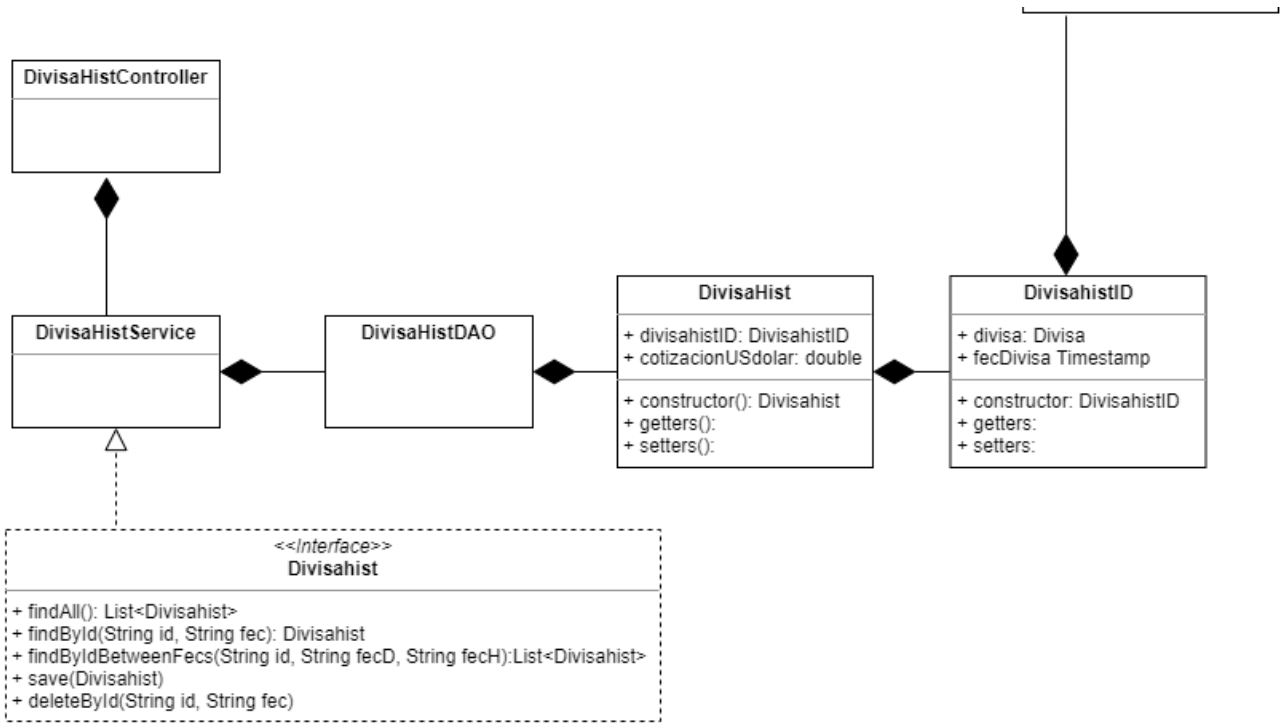
- DivisahistServiceImpl: gestiona las peticiones realizadas en relación al histórico de divisas y las redirecciona a su objeto DAO correspondiente.
- MercadoServiceImpl: gestiona las peticiones realizadas en relación a los mercados y las redirecciona a su objeto DAO correspondiente.
- ValorServiceImpl: gestiona las peticiones realizadas en relación a los valores y las redirecciona a su objeto DAO correspondiente.
- ValorhistServiceImpl: gestiona las peticiones realizadas en relación al histórico de valores y las redirecciona a su objeto DAO correspondiente.

- Objeto de acceso a datos (DAO): estos serán los responsables realmente de acceder a la BB.DD. utilizando las entidades creadas al efecto. Gestionará las llamadas al servicio mediante los correspondientes métodos y utiliza también interfaces para su construcción.

  El siguiente esquema recoge las distintas clases y sus relaciones así como los métodos más específicos.




+ constructor: Valorhist
+ getters ():
+ setters ():
<<Interface>>
Valorhist
+ findAll(): List<Valorhist>
+ findById(int id, String fec): Valorhist
+ findByIdBetweenFecs(int id, String fecD, String fecH):List<Valorhist>
+ valorhistmaxmin(int id, String fecD, String fecH):List<Valorhistmaxmin>
+ save(Valorhist)
+ deleteById(int id, String fec)
MercadoController
MercadoService
MercadoDAO
Mercado
+ codMercado: String
+ nombre: String
+ codPais: String
+ continente: String
+ listaValores: List<Valor>
+ constructor(): Mercado
+ getters():
+ setters():
<<Interface>>
Mercado
+ findAll(): List<Mercado>
+ findById(String): Mercado
+ save(Mercado)
+ deleteById(String)
DivisaController
DivisaService
DivisaDAO
<<Interface>>
Divisa
+ findAll(): List<Divisa>
+ findById(String): Divisa
+ save(Divisa)
+ deleteById(String)
Divisa
+ codDivisa String
+ nombre: String
+ codPais: String
+ cambio: Double
+ listaValores: List<Valor>
+ constructor(): Divisa
+ getters():
+ setters():

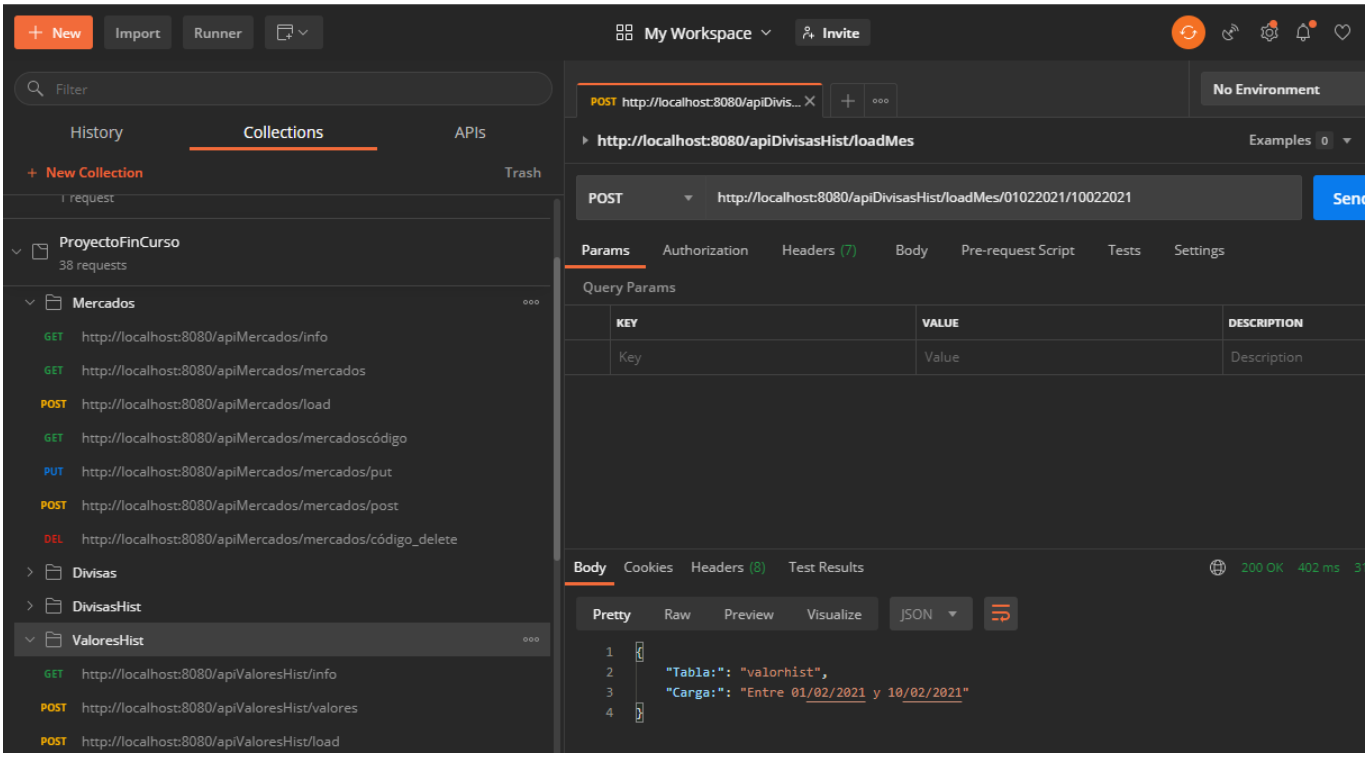


5.3.4 Pruebas

5.3.4.1 Backend

Las pruebas en backend estarán gestionadas por Postman. Para cada una de los objetos/entidades se ha creado una batería de pruebas que engloba todo el ciclo de vida del mismo (CRUD). Además, se incluyen llamadas a métodos de carga masiva, aleatoria y parametrizable para cada una de dichas entidades y así facilitar la casuística de las pruebas. Estas baterías de pruebas, agrupadas por entidad en carpetas, son ejecutables masivamente utilizando la herramienta "Runner" de Postman. No se descarta realizar pruebas unitarias más específicas desde el IDE basándonos en Junit si se dispone en un futuro de tiempo para ello.

Para algunos puntos de entrada algo más específicos, como el de localización de valores que más suben o bajan entre dos fechas, se ha realizado una prueba más específica para detectar posibles errores en los valores devueltos.

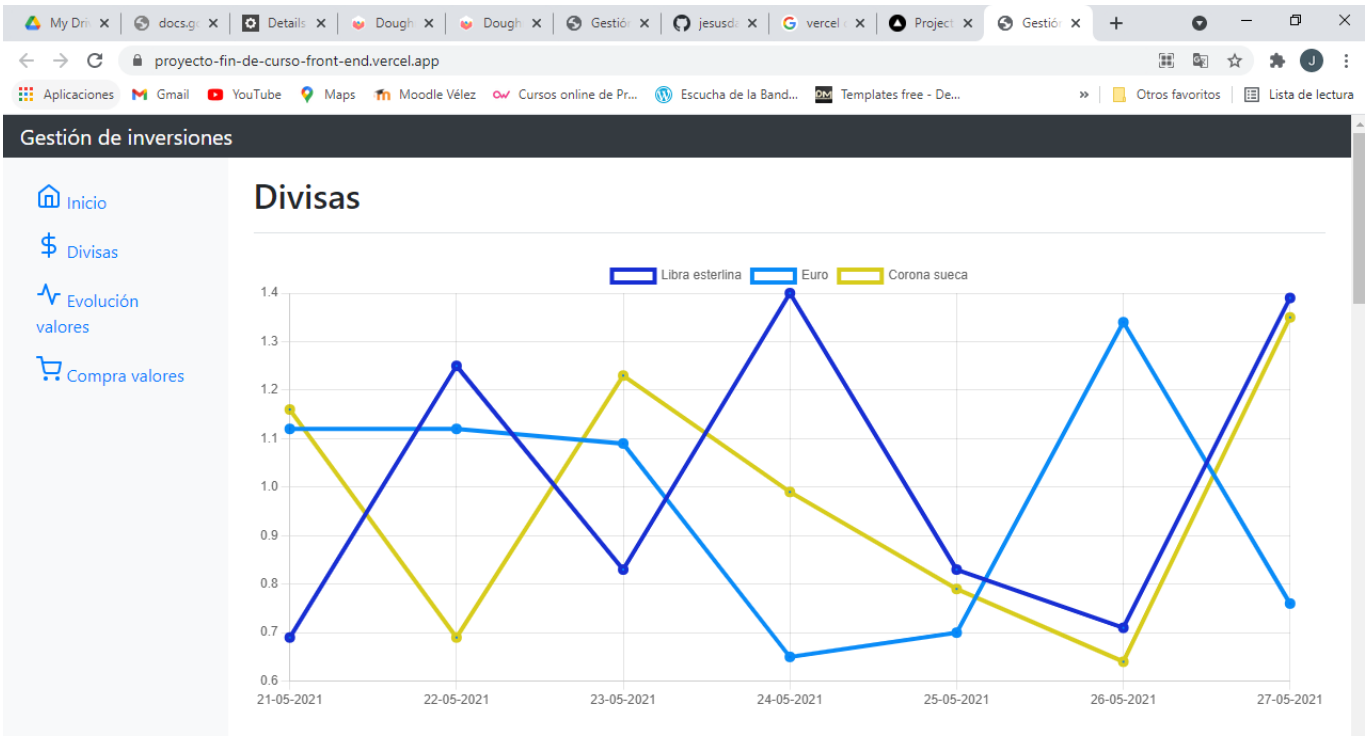


Las pruebas que vemos en la imagen de más arriba están direccionadas a una máquina local (localhost:8080). Añadiremos otro juego de pruebas, una vez realizado el despliegue tanto de la API como de la BB.DD., que apunte a la ubicacion remota de la API para poder confirmar que funciona correctamente

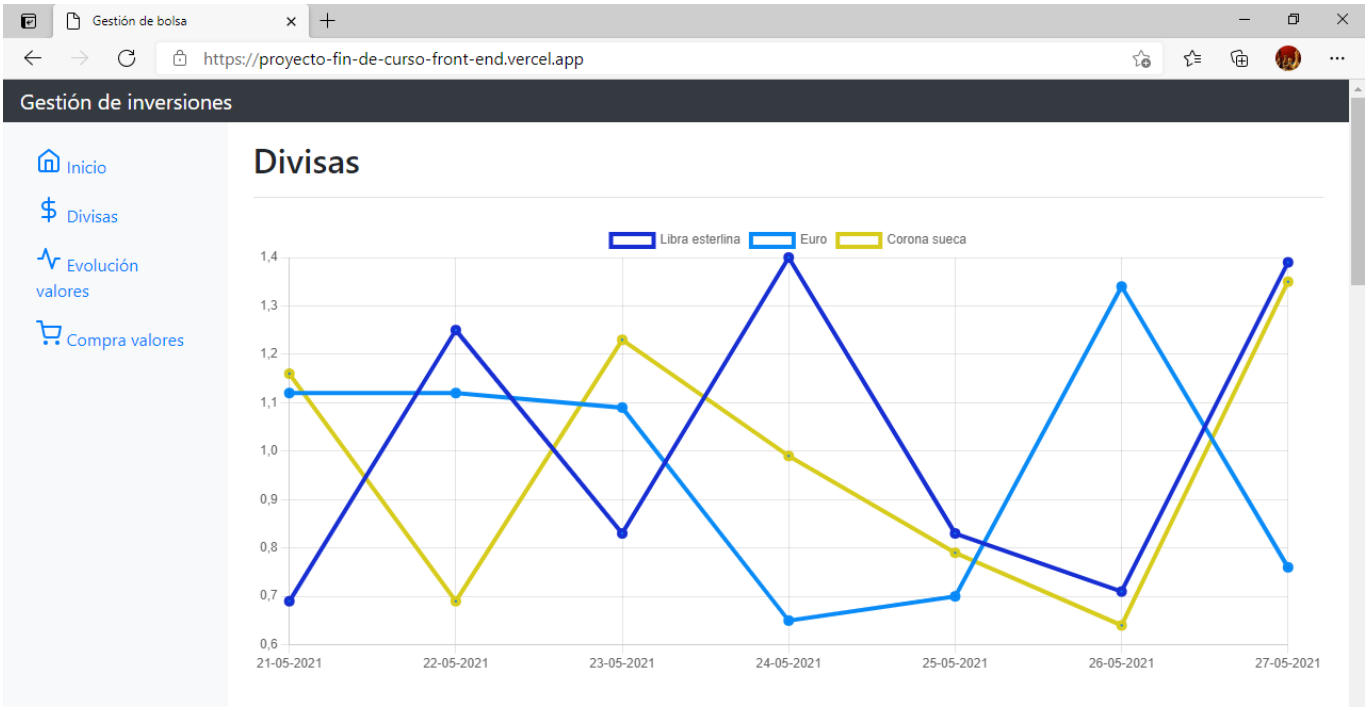
5.3.4.2 Frontend

Las pruebas de la web se han realizado con varios navegadores (Chrome y Edge) a fin de confirmar que su aspecto y respuesta es la correcta. Gracias a estas pruebas se detectaron algunas incidencias importantes en el rendimiento de los accesos a la API. Por su importancia, han quedado reflejadas en el apartado "Conclusiones".

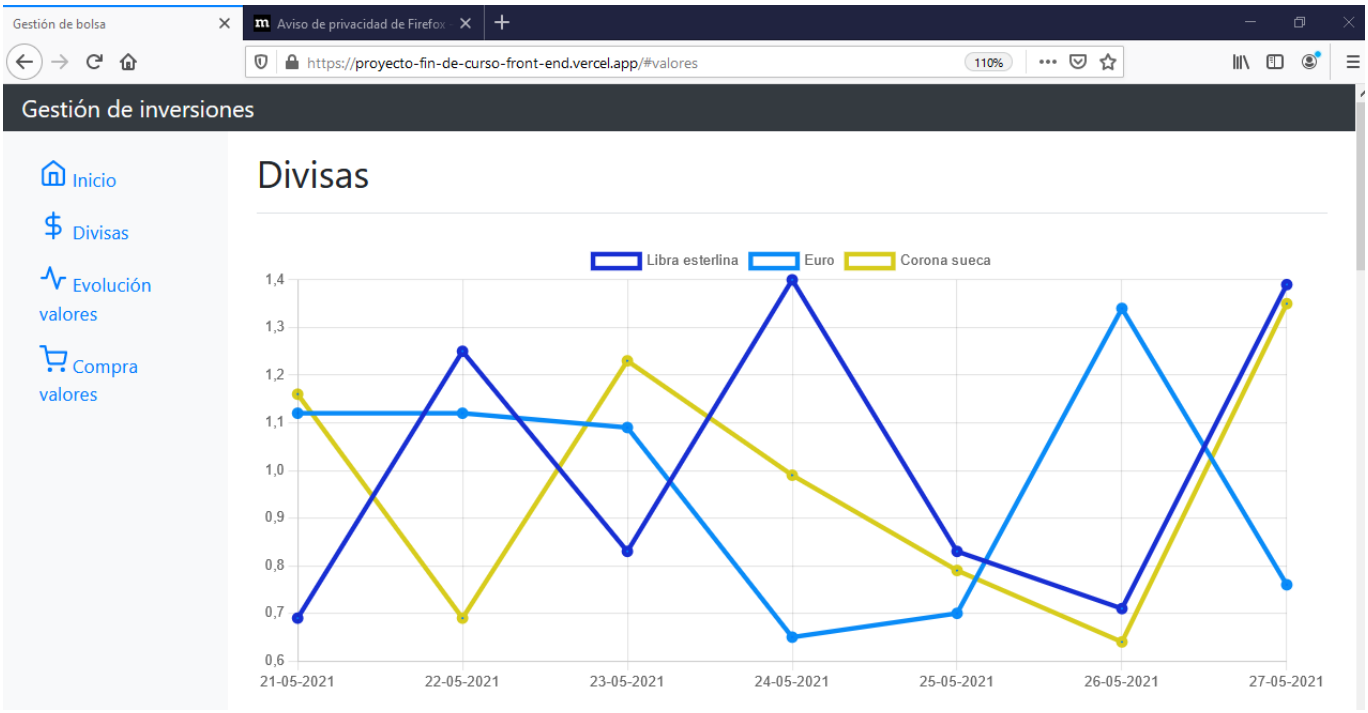
Chrome.



Edge.



Firefox.



5.4 Desarrollo web

5.4.1 Estructura del sitio web

- Backend

  Dependiendo de si estamos realizando las llamadas a la API en local (pruebas) o de si ya está desplegada en Heroku, utilizaremos distintas url las cuales estan gestionadas por la clase *montaUrl*:

  - Local: http://localhost:8080/
  - Desplegada: https://proyectofincurso.herokuapp.com/

  La API, a través de los objetos controladores, pone a nuestra disposición los siguientes puntos de entrada:

  - Divisas
    - GET
      - /apiDivisas/info: devuelve información genérica del controlador divisa (nombre, versión, descripción, etc)
      - /apiDivisas/divisas: devuelve una lista de todas las divisas.
      - /apiDivisas/divisas/(divisaID): devuelve la divisa cuyo id coincida con el que le pasamos.
    - POST
      - /apiDivisas/divisas: inserta el objeto divisa que le pasamos en el cuerpo de la peticón html
      - /apiDivisas/load: carga inicial de divisas de ejemplo.
    - PUT
      - /apiDivisas/divisas: updatea el objeto divisa que le pasamos en el cuerpo de la peticón html
    - DELETE
      - /apiDivisas/divisas/(divisaID)): borra la divisa con id que le llega como parámetro.

- Histórico de divisas
  - GET
    - /apiDivisasHist/info: devuelve información genérica del controlador histórico de divisas (nombre, versión, descripción, etc)
    - /apiDivisasHist/divisashist: devuelve una lista de todo el histórico de divisas.
    - /apiDivisasHist/divisashist/{divisahistId}/{fecha}: devuelve un objeto histórico de divisa en base al id de la divisa y la fecha del histórico. Es decir, la cotización de la divisa a una fecha determinada.
    - /apiDivisasHist/divisashistBetweenFecs/{divisaHistId}/{fechaD}/{fechaH}: devuelve todas las cotizaciones de una determinada divisa entre dos fechas.
  - POST
    - /apiDivisasHist/divisashist: da de alta un registro histórico para la divisa que le llegue en el cuerpo de la petición.
    - /apiDivisasHist/loadFecdesdeFechasta/{fecDesde}/{fecHasta}: genera aleatoriamente valores de cotización entre dos fechas dadas para todas las divisas del sistema.
    - /apiDivisasHist/load: genera un pequeño grupo de datos históricos de prueba para el euro.
  - PUT
    - /apiDivisasHist/divisashist: actualiza los datos históricos para una determinada divisa que le llegue en el cuerpo de la petición.
  - DELETE
    - /apiDivisasHist/divisashist/{divisahistId, fec}: borra los datos históricos de una divisa a una fecha determinada.
- Mercado
  - GET
    - /apiMercados/info: devuelve información genérica del controlador mercado (nombre, versión, descripción, etc)
    - /apiMercados/mercados: devuelve una lista de todas los mercados.
    - /apiMercados/mercados/(mercadoID): devuelve el mercado cuyo id coincida con el que le pasamos.
  - POST
    - /apiMercados/mercados: inserta el objeto mercado que le pasamos en el cuerpo de la peticón html
    - /apiMercados/load: carga inicial de mercados de ejemplo.
  - PUT
    - /apiMercados/mercados: updatea el objeto mercado que le pasamos en el cuerpo de la peticón html.
  - DELETE
    - /apiMercados/mercados/(mercadoID)): borra el mercado con id que le llega como parámetro.
- Valores
  - GET
    - /apiValores/info: devuelve información genérica del controlador valor (nombre, versión, descripción, etc)
    - /apiValores/valores: devuelve una lista de todas los valores.

        - /apiValores/valores/(valorID): devuelve el valor cuyo id coincida con el que le pasamos.
        - /apiValores/valoresNombre/(nombre): devuelve el valor cuyo nombre coincida con el que le pasamos.-
      - POST
        - /apiValores/valores: inserta el objeto valor que le pasamos en el cuerpo de la peticón html
        - /apiValores/load: carga inicial de valores de ejemplo.
      - PUT
        - /apiValores/valores: updatea el objeto valor que le pasamos en el cuerpo de la peticón html
      - DELETE
        - /apiValores/valores/(valorID)): borra el valor con id que le llega como parámetro.
    - Histórico de valores
      - GET
        - /apiValoresHist/info: devuelve información genérica del controlador histórico de valores (nombre, versión, descripción, etc)
        - /apiValoresHist/valoreshist: devuelve una lista de todo el histórico de valores.
        - /apiValoresHist/valoreshist/{valoreshistId}/{fecha}: devuelve un objeto histórico de valor en base al id del valor y la fecha del histórico. Es decir, la cotización del valor a una fecha determinada.
        - /apiValoresHist/valoreshistBetweenFecs/{divisaHistId}/{fechaD}/{fechaH}: devuelve todas las cotizaciones de un determinado valor entre dos fechas.
      - POST
        - /apiValoresHist/valoreshist: da de alta un registro histórico para valor que le llegue en el cuerpo de la petición.
        - /apiValoresHist/loadFecdesdeFechasta/{fecDesde}/{fecHasta}: genera aleatoriamente valores históricos de cotización entre dos fechas dadas para todos los valores del sistema.
        - /apiValoresHist/load: genera un pequeño grupo de datos históricos de prueba para el valor con id=1.
      - PUT
        - /apiValoresHist/valoreshist: actualiza los datos históricos para un determinado valor que le llegue en el cuerpo de la petición.
      - DELETE
        - /apiValoresHist/valoreshist/{valorhistId, fec}: borra los datos históricos de un valor a una fecha determinada.

- FrontEnd

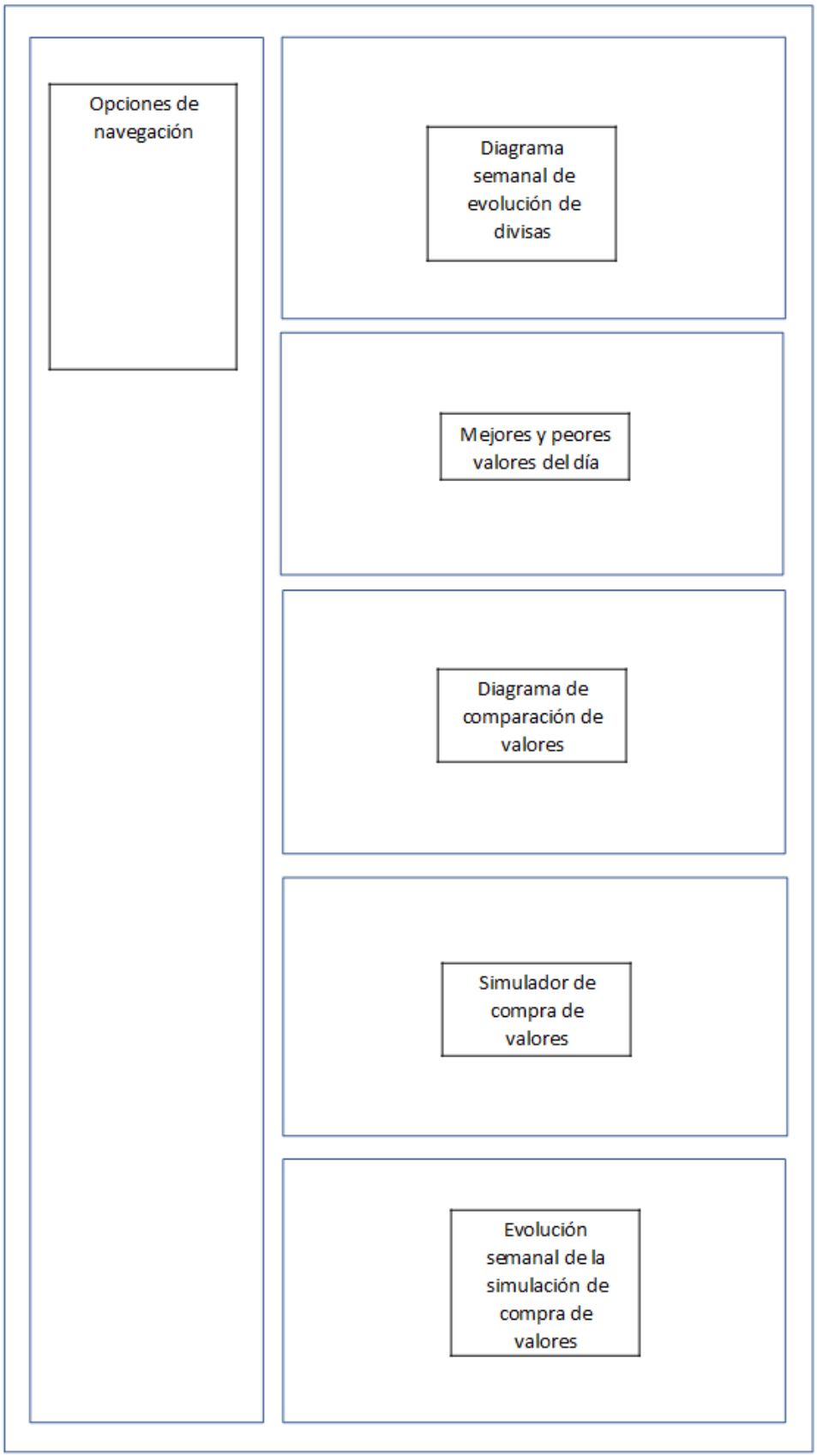
  Para acceder a nuestra aplicación, una vez desplegada en Vercel, utilizaremos la siguiente URL:

  https://proyecto-fin-de-curso-front-end.vercel.app/

5.4.2 Maquetación

Nuestro sitio web constará de una página principal desde la que podremos acceder a las distintas opciones de mantenimiento, cuando estén disponibles. Tal y como podemos ver en el documento de maquetación,

contaremos con dos gráficas principales, divisas y valores, entre las que se insetará una tabla con las subidas y bajadas más señaladas del día.



5.4.3 Manual de estilo.

Al utilizar bootstrap, version 4, se fijan una serie de estilos por defecto que pasamos a ver a continuación:

- El tamaño de fuentes por defecto es **font-size de 16px** y separación de líneas **line-height de 1.5 rem**.
- Dispone de una **font-family** predeterminada de tipo "**Helvetica Neue", Helvetica, Arial, sans-serif.**
- Pas etiquetas

  tienen **margin-top: 0 y margin-bottom: 1rem** (16px por defecto).

- Los encabezados

  —

  ---

  disponen de estilos predefinidos:

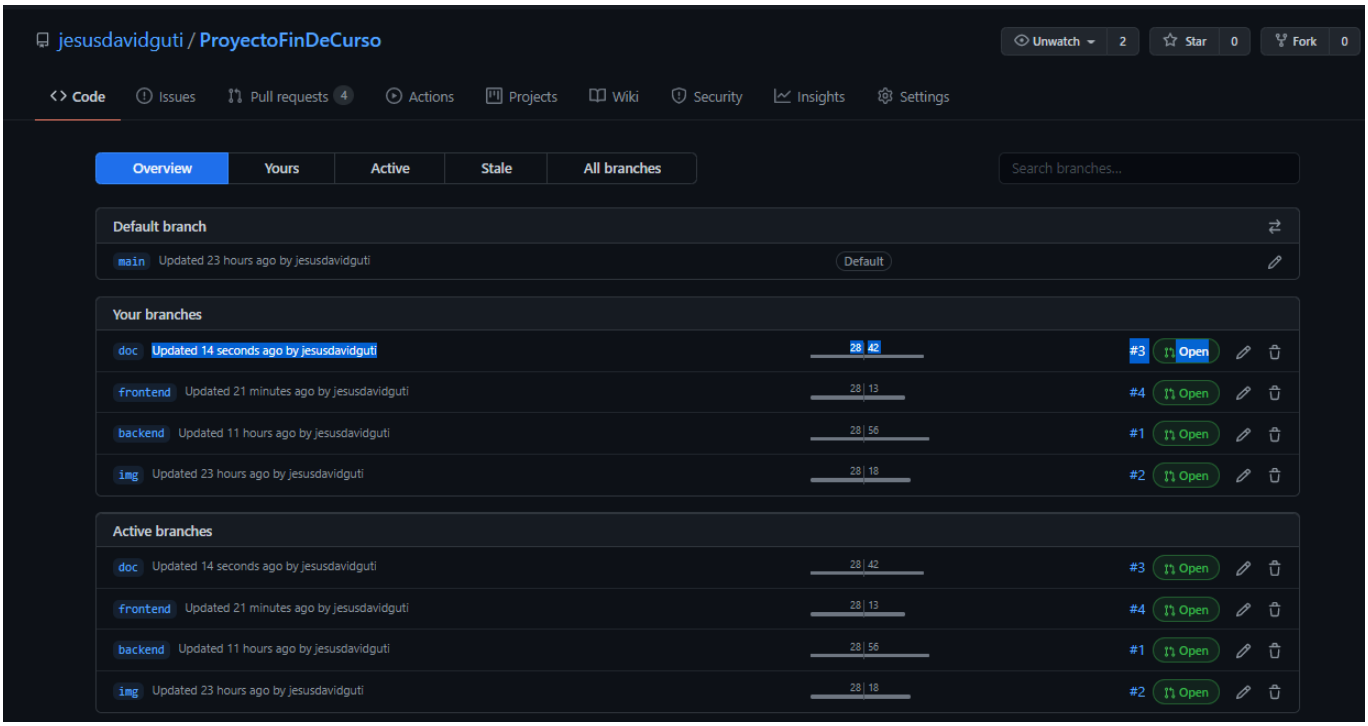
  - **h1 Bootstrap (2.5rem = 40px)**
  - **h2 Bootstrap (2rem = 32px)**
  - **h3 Bootstrap (1.75rem = 28px)**
  - **h4 Bootstrap (1.5rem = 24px)**
  - **h5 Bootstrap (1.25rem = 20px)**
  - **h6 Bootstrap (1rem = 16px)**

- Las clases para los colores del texto son: **.text-primary, .text-secondary, .text-muted, .text-success, .text-danger, .text-warning, .text-info, .text-white, .text-dark y .text-light.**
- Las clases de colores de fondo son: **.bg-primary , .bg-success , .bg-info , .bg-warning , .bg-danger , .bg-secondary , .bg-dark y .bg-light .** Cuando utilizamos los colores de fondo es muy usual combinarlo junto con las clases de los colores de texto para configurar un buen contraste entre ellos.

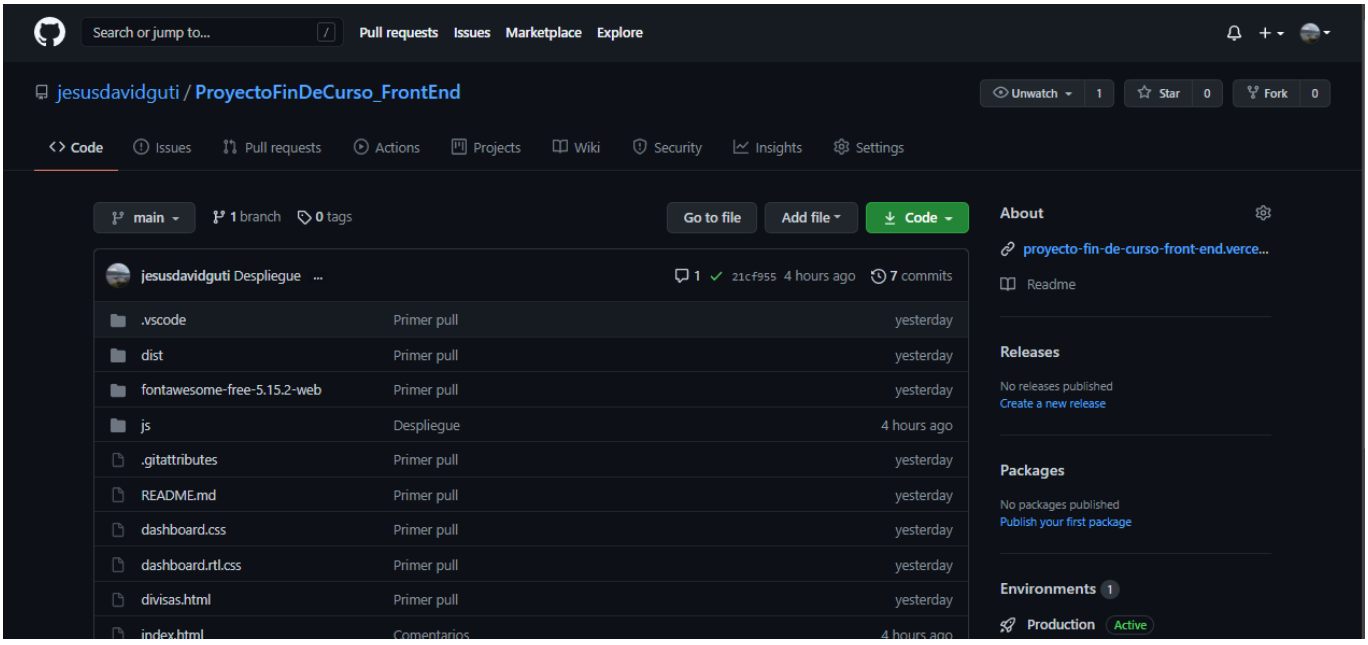
5.5 Control de versiones.

Github será nuestra herraminta de versionado y de mantenimiento de las distintas ramas del desarrollo. Estas ramas nos ayudan a manterner ordenado y estructurado todo el trabajo realizado durante el desarrollo y, en última instancia, nos ayudarán a realizar el desplieque. Las ramas, tal y como se puede ver en la imagen de más abajo, son cinco:

- Main: sería la presentación del proyecto y para información general del mismo
- Frontend: contendrá la parte web ejecutada por el navegador(html, js, etc).
- Backend: la API así como los elementos anexos a ella (ficheros de recursos, conexiones, etc) irán en esta rama.
- Doc: toda la documentación que se genere durante el proyecto irá en esta rama, bien en formato .md bien en pdf
- Img: las imágenes que utilizamos para ilustrar el proyecto, así como cualquier otra, irá ubicada aquí.



Debido a la imposibilidad de desplegar en Vercel desde una rama (sólo es posible desde main), se ha tenido que generar un nuevo proyecto Github llamado ProyectoFinDeCurso_FrontEnd que nos pertimitrá automatizar el despliegue del frontend.



6. Dificultades encontradas

- Despliegue.
  - Vercel no permite el despliegue desde una rama, siempre lo hace desde main, por lo que ha sido necesario clonar la rama de frontend para poder hacer el despliegue.
- FrontEnd
  - A la hora de usar Ajax para recuperar y mostrar datos hay que tener en cuenta la "rapidez" con que se ejecutan las sentencias. Así, para mostrar datos o actualizar un objeto con datos

procedentes de Ajax es necesario hacer que los métodos que pintan los datos no se ejecuten antes de que Ajax haya finalizado su ejecución y devuelto los datos.
    - El objeto para dibujar gráficas, llamado "Chart", es muy útil pero presenta ciertas características en su manejo que debemos respetar y aprender para su correcto uso. Podemos trabajar con el como si fuese un objeto "plano", definiendo sus propiedades una a una., o bien acceder a las mismas con la notación punto. Es esta última opción por la que nos hemos decantado a pesar de que sus propiedades están muy agrupadas y que el cambio de una sola puede afectar al resto.
    - Las versiones de Bootstrap hacen que determinados componentes funcionen o no correctamente cuando iteraccionan con otros.
    - Vercel no permite el despliegue desde otra rama que no sea main. Es por ello que ha sido necesario clonar la rama de frontend en un nuevo repositorio de Github llamado ProyectoFinDeCurso_FrontEnd. Sin ser una solución muy elegante, sí ha permitido el automatizar le despliegue.
- BackEnd
    - Spring requiere del uso de transacciones para que las acciones de inserción y actualización tengan efecto. Curiosamente no alerta de la no ejecución de acciones, simplemente no las realiza.
    - El uso de un identificador que requiera una entidad hace que se tengan que tener en cuenta una serie de aspectos a la hora de utilizarlo. El más importante es el hecho de que para poder realizar acciones de comparación es necesario añadir métodos específicos para comparación de objetos como si fuesen un ID.
    - El envío de fechas a la API se ha relizado en formato ddMMyyyy. Se ha optado por la simplicidad del dato inicial de entrada y su posterior tratamiento en la API.
    - La carga inicial de datos en la BB.DD. remota ha sido problematica ya que había que insertar datos para, al menos, un año de cotización. Las peticiones a la API para este tipo de acciones (load) se ralentizaban mucho y no se acababan de completar ya que la llamda http se cancelaba por tiempo. Para solventar este problema, hubo que lanzar el proceso en local pero atacando a la BB.DD. remota y aún así requirió varias horas en el proceso.

7. Conclusiones

La ejecución de este proyecto así como su puesta en marcha ha hecho que determinadas conclusiones, aun siendo obvias, hayan sido más evidentes aún. Veamos algunas.

- Generales
    - Una vez más se cumple el dicho de que la *primera solución que pensamos no es la mejor casi nunca*.
    - Las espedificaciones funcionales, por muy generales que sean, deben ser atendidas al pie de la letra.
- FrontEnd
    - Las llamadas a la Api deben hacerse escalonadamente si la callback es la encargada de pasar la información a otro objeto. Así, a la hora de informar el chart, los datasets con la información se deben pasar uno a uno. Es por ello que las llamadas se hacen anidadas para que se ejecuten de una en una y en orden. De otra forma, se puede dar el caso de que algunas de las llamadas a la API no hayan finalizado cuando ya estamos informando el objeto con el siguiente grupo de valores.

    - A la hora de utilizar cualquier tipo de componente web hay que tener muy en cuenta su versión y compatibilidad con otros componentes.
- BackEnd
    - Las consultas que lancemos con JPA desde backend deben devolver siempre un objeto. No importa la simpleza del dato a devolver (entero, decimal, cadena, etc.), si este no está integrado en un objeto Spring no lo gestionará.
    - Las interfaces de los objetos DAO pueden ser exactamente iguales, por lo que sólo habría que hacer una. Sin embargo, las particularidades de cada objeto hacen que sea necesario introducir variaciones que poco o nada tienen que ver con la funcionalidad del resto de objetos. Es por ello que es preferible hacer una interfaz para cada objeto DAO.
    - La BB.DD. no genera claves externas al uso una vez la creamos. Son JPA y Spring los que, mediante anotaciones, se encargan de generar dichas relaciones cuando activamos el servicio por vez primera. De hecho, las tablas son creadas en la BB.DD. como ficheros simples sin relación alguna y que podemos borrar desde el gestor de BB.DD. sin tener avisos de fallos de integridad.

8. Posibles mejoras

Las posibles mejoras podrían ser infinitas una vez visto el potencial que la API nos da. Algunas podrían ser las siguientes:

- FrontEnd
    - Area de mantenimiento: donde se podrán mantener las entidades padre (divisa, valor, mercado, etc.)
    - Moneda de uso: utilizamos el dolar americano por defecto para almacenar, pero podríamos mostrar la información en cualquier otra moneda simplemente convirtiendo los valores antes de ser mostrados.
    - El objeto Chart.js permite su tratamiento a nivel de objeto, como ya hemos visto. Aunque se han creado funciones genéricas para la gestión de este tipo de objeto, se podría haber mejorado aún más su tratamiento una vez se conoce el funcionamiento de sus métodos.
    - Hacer completamente "responsive" la aplicación.
- BackEnd
    - Incluir batería de pruebas con JUnit para un testeo más exhaustivo de los métodos, tanto públicos como privados.
    - Carga de datos más parametrizada: para una sola entidad, limitando los importes máximo y mínimo, etc.
    - Distintas consultas que nos darán información más detallada de los valores en sus ciclos de subida y bajada.
    - Enfocar algunos de los puntos de entrada a la alimentación del objeto chart.js, es decir, devolver arrays que contengan sólo los datos y las etiquetas a usar para descargar de dichos procesos a la parte cliente.

9. Fuentes de información

En líneas generales, la información a consultar ha sido muy amplia y precisa. En la rama main del Proyecto en Github aparece de forma más detallada cada una de dichas fuentes. La consulta de webs de programadores

tales como StrackOverflow ha sido de gran ayuda y, a la vez, un fuente de dudas ya que el enfoque a un mismo problema ya resuelto puede ser muy variado.

Estas son algunas de las fuentes de información consultadas principalmente:

- Funcionalidad
    - Bolsa de Madrid: https://www.bolsamadrid.es/esp/aspx/Portada/Portada.aspx
- Sprin boot
    - Mi tutorial sobre Spring: https://github.com/jesusdavidguti/TutorialSpringJPA
    - Anotaciones JPA: https://www.objectdb.com/api/java/jpa/annotations/relationship
    - Api REST: https://www.nigmacode.com/java/crear-api-rest-con-spring/
    - Query en Spring: https://spring.io/blog/2011/02/10/getting-started-with-spring-data-jpa/
- JPA
    - Persistencia con JPA: https://www.infoworld.com/article/3387643/java-persistence-with-jpa-and-hibernate-part-2-many-to-many-relationships.html
    - JPA @ OneToMany: https://www.arquitecturajava.com/jpa-onetomany/
    - JPA @ ManyToOne: https://www.arquitecturajava.com/jpa-manytoone/
    - @id automático: https://stackoverflow.com/questions/20603638/what-is-the-use-of-annotations-id-and-generatedvaluestrategy-generationtype
    - Anotar fecha y hora en JPA: https://www.it-swarm-es.com/es/java/como-almacenar-la-fechahora-y-las-marcas-de-tiempo-en-la-zona-horaria-utc-con-jpa-e-hibernate/958259387/
- Modelización
    - Asociación, agregación y composición: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-aggregation-vs-composition/
- Github
    - Sintaxis Github: https://docs.github.com/es/github/writing-on-github/basic-writing-and-formatting-syntax
    - Typora: https://support.typora.io/Links/#hyperlink
- Interfaz
    - Iconos: https://feathericons.com/
    - Diagramas estadísticos
        - Lineal: https://www.chartjs.org/docs/latest/charts/line.html
        - Donut: https://www.chartjs.org/docs/latest/charts/doughnut.html
    - Bootstrap
        - Dropdowns: https://getbootstrap.com/docs/4.0/components/dropdowns/
        - Bootstrap 4.0: https://www.eniun.com/texto-tipografia-colores-bootstrap/