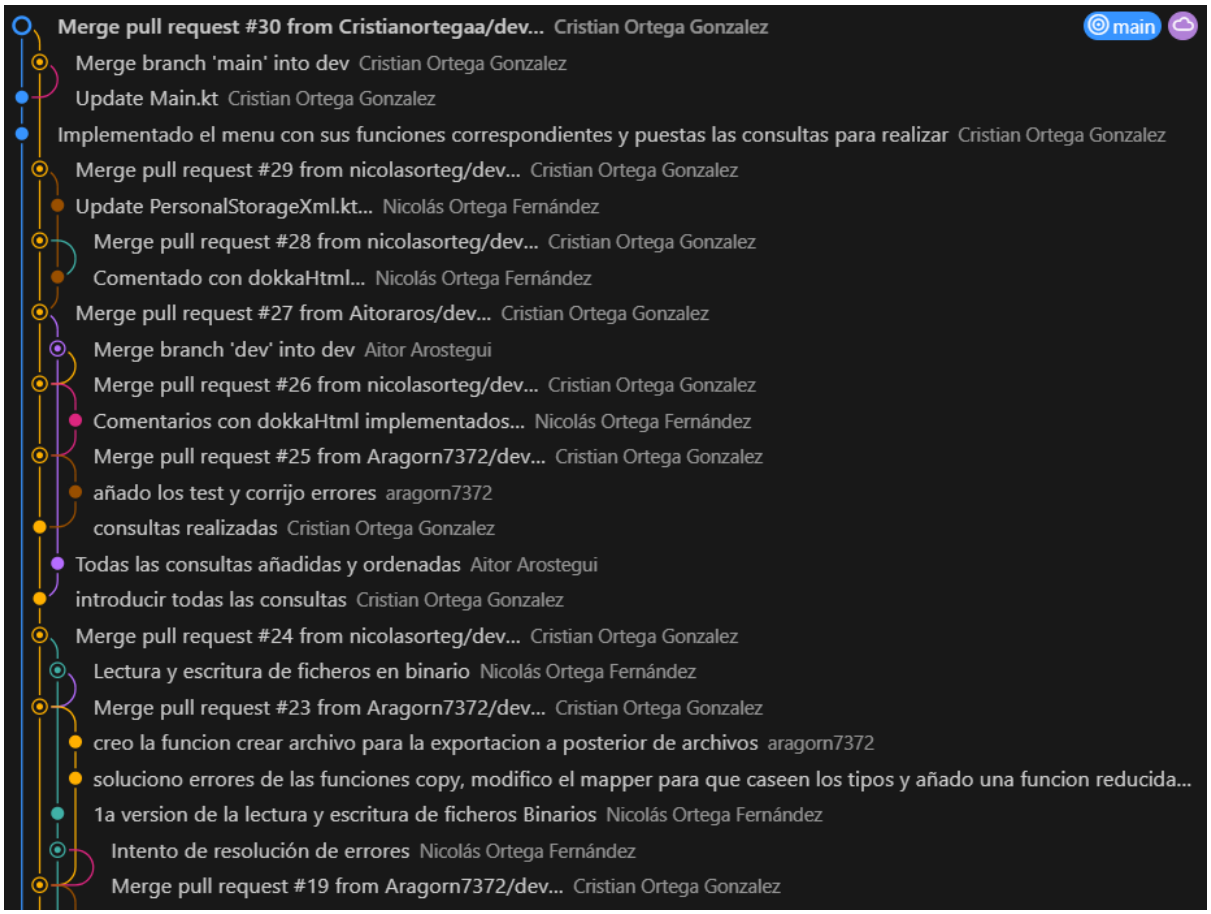
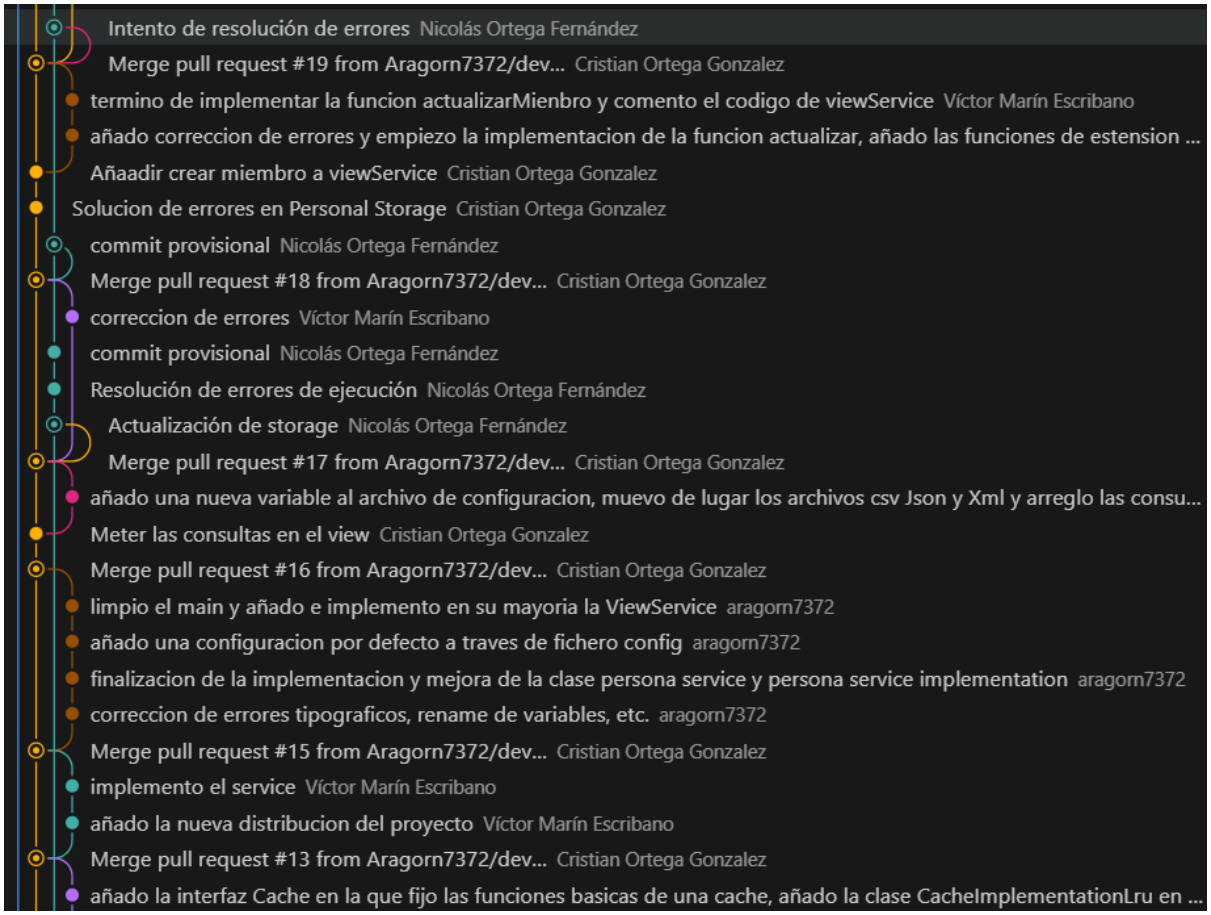
HERRAMIENTAS TECNOLÓGICAS USADAS

Para llevar a cabo el proyecto hemos usado distintas tecnologías, estas son:

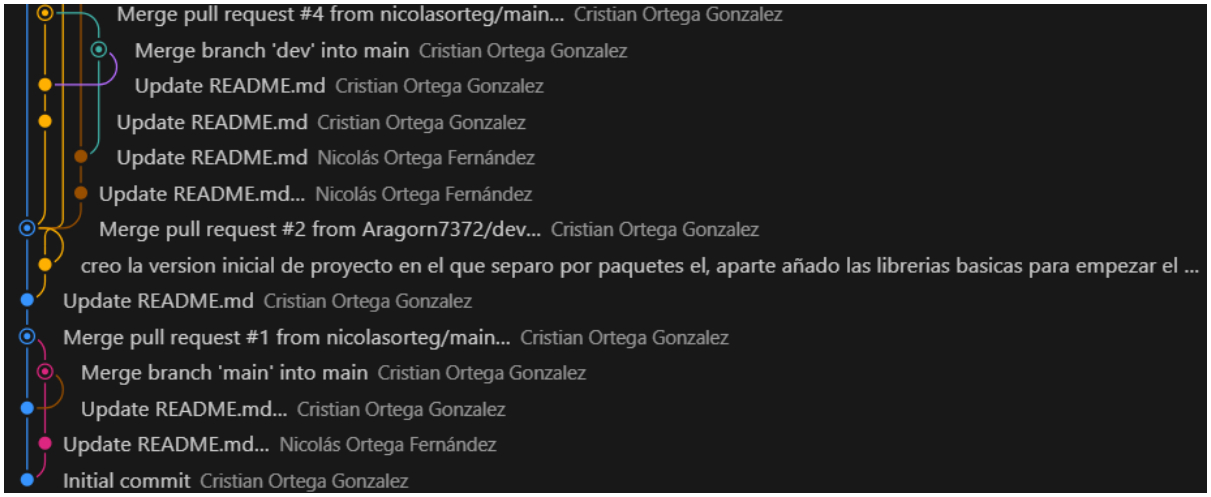
- **IntelliJ**: utilizado como entorno de desarrollo para crear y editar el código.
- Visual Studio Code: empleado para visualizar el código de los compañeros de equipo, lo que nos ha permitido una mejor colaboración.
- **GitHub**: plataforma usada para manejar el código de proyecto de manera remota.
- **GitFlow**: estrategia utilizada para gestionar el flujo de trabajo de Git, estableciendo la rama Dev para el desarrollo del proyecto.
- **Git**: control de versiones usado para gestionar los cambios en el código fuente y facilitar la colaboración.
- **Windows PowerShell**: herramienta usada para interactuar con Git mediante comandos.
- **Trello**: plataforma utilizada la distribución de tareas, lo que nos ha permitido organizarnos mejor el proyecto.

GITFLOW




Intento de resolución de errores Nicolás Ortega Fernández
Merge pull request #19 from Aragorn7372/dev... Cristian Ortega Gonzalez
termino de implementar la funcion actualizarMienbro y comento el codigo de viewService Víctor Marín Escribano
añado correccion de errores y empiezo la implementacion de la funcion actualizar, añado las funciones de estension ...
Añaadir crear miembro a viewService Cristian Ortega Gonzalez
Solucion de errores en Personal Storage Cristian Ortega Gonzalez
commit provisional Nicolás Ortega Fernández
Merge pull request #18 from Aragorn7372/dev... Cristian Ortega Gonzalez
correccion de errores Víctor Marín Escribano
commit provisional Nicolás Ortega Fernández
Resolución de errores de ejecución Nicolás Ortega Fernández
Actualización de storage Nicolás Ortega Fernández
Merge pull request #17 from Aragorn7372/dev... Cristian Ortega Gonzalez
añado una nueva variable al archivo de configuracion, muevo de lugar los archivos csv Json y Xml y arreglo las consu...
Meter las consultas en el view Cristian Ortega Gonzalez
Merge pull request #16 from Aragorn7372/dev... Cristian Ortega Gonzalez
limpio el main y añado e implemento en su mayoria la ViewService aragorn7372
añado una configuracion por defecto a traves de fichero config aragorn7372
finalizacion de la implementacion y mejora de la clase persona service y persona service implementation aragorn7372
correccion de errores tipograficos, rename de variables, etc. aragorn7372
Merge pull request #15 from Aragorn7372/dev... Cristian Ortega Gonzalez
implemento el service Víctor Marín Escribano
añado la nueva distribucion del proyecto Víctor Marín Escribano
Merge pull request #13 from Aragorn7372/dev... Cristian Ortega Gonzalez
añado la interfaz Cache en la que fijo las funciones basicas de una cache, añado la clase CacheImplementationLru en ...


Merge pull request #12 from nicolasorteg/dev... Cristian Ortega Gonzalez
Tratamiento de archivos JSON implementado y comentado Nicolás Ortega Fernández
Tratamiento de archivos CSV implementado y comentado Nicolás Ortega Fernández
Interfaz implementada y comentada correctamente Nicolás Ortega Fernández
Excepciones implementadas y comentadas Nicolás Ortega Fernández
DTOs implementados y comentados Nicolás Ortega Fernández
Librerías implementadas Nicolás Ortega Fernández
Clase sellada con todas las excepciones necesarias Nicolás Ortega Fernández
Resolución de fallos al subir a Git Nicolás Ortega Fernández
Lecturas de los archivos .csv y .json implementadas junto con sus DTOs y mapper correspondiente... Nicolás Ortega Fern...
Corrección de errores. Nicolás Ortega Fernández
Realizado las 14 primeras consultas incluyendo las listas de los jugadores y de los entrenadores Cristian Ortega Gonzalez
Implementado menú con sus funciones correspondientes y puesto las consultas para realizarlas Cristian Ortega Gonzalez
Add files via upload... Cristian Ortega Gonzalez
Merge pull request #9 from Aragorn7372/dev... Cristian Ortega Gonzalez
Merge branch 'Cristianortegaa:dev' into dev Víctor Marín Escribano
Merge pull request #8 from Aragorn7372/dev... Cristian Ortega Gonzalez
añado los logger y corrijo errores aragorn7372
añado la rama repositoy, con la interfaz que dictamina la funcionalidad de RepositoryCRUD, tambien incluyo la interf...
Merge pull request #6 from Cristianortegaa/dev... Cristian Ortega Gonzalez
Merge pull request #5 from Aragorn7372/features_inicio_proyecto... Cristian Ortega Gonzalez
Merge branch 'Cristianortegaa:main' into features_inicio_proyecto Víctor Marín Escribano
Merge pull request #3 from nicolasorteg/main... Cristian Ortega Gonzalez
añado las clases jugadores,entrenador,persona Víctor Marín Escribano
Merge pull request #4 from nicolasorteg/main... Cristian Ortega Gonzalez



LECTURA DE FICHEROS

En este apartado se detalla todo el funcionamiento del apartado encargado de la lectura y escritura de ficheros, con extensiones .csv, .json, .xml y .bin.

Interfaz PersonalStorage.

Esta interfaz define los métodos necesarios para poder leer y escribir los ficheros. Al ser una interfaz, actúa como un contrato, y todas las clases que hereden de esta interfaz se verán obligadas implementar las funciones:

- **leerDelArchivo**(file: File): List <Persona> -> el propósito de esta función es leer un archivo y extraer los datos de los jugadores y entrenadores almacenados dentro. Se da como parámetro de entrada un objeto File que contiene el archivo de donde se leerá la información. La salida será la lista del Personal obtenida en el archivo. EL funcionamiento es muy simple: abre el archivo, lo lee, procesa los datos para convertirlos en objetos del tipo Persona y devuelve la lista de todas las personas.
- **escribirAUnArchivo**(file: File, personas: List<Persona>) -> el propósito de esta función es guardar una lista de objetos Persona en un archivo. De entrada, se introduce el objeto File que contiene el archivo donde se escribirá la documentación, y una lista de Persona que contiene los datos a guardar. La salida no devuelve nada, simplemente escribe la información en el archivo. El funcionamiento es más sencillo aún, convierte la lista en el formato adecuado y escribe el contenido.

Cada tipo de almacenamiento tiene su propia implementación. Hay una implementación para estas extensiones:

CSV

Esta clase es una implementación de la interfaz PersonalStorage y se encarga de la lectura y escritura de objetos Persona en archivos de formato CSV. Esta clase usa

loggings para registrar los sucesos más importantes. Como se mencionó antes, al heredar de la interfaz PersonalStorage, implementa los métodos leerDelArchivo y escribirAUnArchivo.

- **leerDelArchivo**: su objetivo es leer el CSV y convertir su contenido en una lista de objetos Persona. Para este tipo de archivos lo que hace es verificar si el archivo es válido y accesible mediante un condicional if else. Si no es válido o no se accesible, lanza una excepción creada en la carpeta exception. La excepción es PersonasStorageException, que indica un problema con el almacenamiento de personas. A continuación, para llevar a cabo la lectura del fichero comienza omitiendo la cabecera haciendo uso de un drop(1). El 1 corresponde al número de filas que va a saltar. Después divide las líneas en columnas usando split(‘,’), elimina los espacios en blanco con el trim y mapea el archivo. Al haber dos tipos de persona, se ha tenido que hacer uso del condicional when. Así, puede dividir entre entrenadores, when rol sea Entrenador, y jugadores when rol sea Jugador. En caso de que el rol sea desconocido/incorrecto lanza una excepción. Lo mismo pasa si el archivo tiene un formato incorrecto.
- **escribirAUnArchivo**: escribe una lista de objetos Persona en un archivo CSV. Primero verifica que el CSV sea válido y que el directorio de destino exista. Luego escribe la cabecera en el archivo mediante el writeText. Recorre la lista de persona y las convierte en líneas del CSV, diferenciando por rol (entrenador y jugador). Por último, agrega estas líneas en el archivo y registra en logs la operación.

Como se ha mencionado, para la gestión de errores se usa PersonasException.PersonasStorageException para poder manejar los problemas como archivos que no existen, formatos/extensiones incorrectas o roles no válidos.

Si posteriormente se van a añadir nuevos tipos de Persona, es importante saber que habrá que modificar la lógica de la lectura y escritura para poder introducirlos.

JSON

Esta clase es una implementación de la interfaz PersonalStorage y se encarga de la lectura y escritura de objetos Persona en archivos de formato JSON. Esta clase usa loggings para registrar los sucesos más importantes. Como se mencionó antes, al heredar de la interfaz PersonalStorage, implementa los métodos leerDelArchivo y escribirAUnArchivo.

- **leerDelArchivo**: en esta implementación empieza verificando si el archivo es válido, existe, se puede leer y tiene la extensión .json mediante el condicional if else. En caso de que algo de esto no se cumpla, lanza la excepción

PersonasStorageException, que proviene de PersonasException y ha sido mencionada anteriormente en la trata de archivos de los CSV. Posteriormente lee el contenido y lo convierte a una lista. Dependiendo del 'rol', crea un EntrenadorDto o uno JugadorDto. Usa la PersonaMapper para poder convertir los DTO en los modelos base Persona. Por último, devuelve la lista de objetos creados y lanza una excepción si el archivo no tiene el formato correcto.

- **escribirAUnArchivo**: empieza verificando que el archivo sea válido y que el directorio exista de nuevo mediante el condicional if y lanzando la excepción si algo no se cumple. Convierte cada objeto Persona en un JSON, diferenciando entre Entrenador y Jugador, en base a la columna rol. Genera la estructura JSON válida poniendo de prefijos y sufijos '[]'. Por último, escribe el contenido en el archivo mediante el writeText y listo.

Como se ha mencionado, para la gestión de errores se usa PersonasException.PersonasStorageException para poder manejar los problemas como archivos que no existen, formatos/extensiones incorrectas o roles no válidos.

Si posteriormente se van a añadir nuevos tipos de Persona, es importante saber que habrá que modificar la lógica de la lectura y escritura para poder introducirlos.

XML

Esta clase es una implementación de la interfaz PersonalStorage y se encarga de la lectura y escritura de objetos Persona en archivos de formato XML, usando la biblioteca kotlinx.serialization. Esta clase usa loggings para registrar los sucesos más importantes. En este caso, además de insertar las dependencias necesarias como se ha hecho en los anteriores formatos, hay que inyectar además las dependencias 'decodeFromString' y 'encodeToString'. Como se mencionó antes, al heredar de la interfaz PersonalStorage, implementa los métodos leerDelArchivo y escribirAUnArchivo.

- **leerDelArchivo**: comienza verificando que el archivo exista mediante un condicional. Para que la función empiece verifica que exista, que se pueda leer, que no esté vacío y que sea de la extensión xml. Posteriormente lee el contenido del archivo y lo deserializa en un objeto llamado EquipoDtoXml. Sigue conviertiendo cada persona del XML en su respectivo modelo en base al rol, Jugador o Entrenador, haciendo uso del condicional when. Por último, como salida devuelve la lista de personas ya convertidas.
- **escribirAUnArchivo**: primero verifica que el archivo sea válido. Mediante el condicional if, verifica que exista, que esté en el directorio y que tenga la extesión correcta. Después lo que hace es convertir las personas a los DTO correspondiente, dependiendo y haciendo uso del casting de si son Jugadores o

Entrenadores. Finalmente, serializa la lista de personas en el formato de XML y los escribe en el archivo.

Como se ha mencionado, para la gestión de errores se usa PersonasException.PersonasStorageException para poder manejar los problemas como archivos que no existen, formatos/extensiones incorrectas o roles no válidos.

Si posteriormente se van a añadir nuevos tipos de Persona, es importante saber que habrá que modificar la lógica de la lectura y escritura para poder introducirlos.

BINARIO

Esta clase es una implementación de la interfaz PersonalStorage y se encarga de la lectura y escritura de objetos Persona en archivos de formato BIN. Esta clase usa loggings para registrar los sucesos más importantes. Como se mencionó antes, al heredar de la interfaz PersonalStorage, implementa los métodos leerDelArchivo y escribirAUnArchivo.

- **leerDelArchivo**: inicia creando el logger y llamando al mapper. Después verifica mediante un condicional if si el fichero existe, se puede leer, no esta vacío y es de la extensión.bin. Si se cumple todo esto, declara la lista donde se almacenarán las personas y comienza con la lectura, haciendo uso del RandomAccesFile (raf). Esto sirve para convertir los datos en Jugadores y Entrenadores. Por último, acaba devolviendo la listade objetos ya convertidos.
- **escribirAUnArchivo**: como siempre, arranca verificando si el directorio existe y si es de la extensión adecuada mediante el condicional if, y lanzando las excepcion PersonasStorageException si no cumple algo de lo antes mencionado. Para empezar la escritura mediante el RandomAccesFile de nuevo empieza limpiando el archivo antes de proceder a la escritura con el setLength(0), asegurándose así de que esté vacío. Finaliza guardando la info. De cada persona en el archivo

Como se ha mencionado, para la gestión de errores se usa PersonasException.PersonasStorageException para poder manejar los problemas como archivos que no existen, formatos/extensiones incorrectas o roles no válidos.

Si posteriormente se van a añadir nuevos tipos de Persona, es importante saber que habrá que modificar la lógica de la lectura y escritura para poder introducirlos.

PRINCIPIOS SOLID

1 -> **Principio de responsabilidad única**: cada clase debe tener una única responsabilidad como por ejemplo la clase jugadores que solo es exclusivamente para los jugadores o también como la clase entrenadores que como indica el nombre es para los entrenadores, es decir, que cada clase cumple su función.

2 -> **Principio de Abierto/Cerrado**: el principio dice que una clase debe estar abierta para añadir nuevas funcionalidades, sin modificar su código. En este caso, por ejemplo, en personaServiceImplementation tenemos el ejemplo de a la hora de meter un nuevo formato de archivo, estamos extendiendo, pero no modificando.

3 -> **Principio de sustitución de Liskov**: tenemos subclases para usarse en lugar se su clase, como por ejemplo tenemos la clase persona y de ellas las subclases que heredan de ella que son entrenador y jugador

4 -> **Principio de Segregación de Interfaces**: es mejor tener interfaces específicas que no una gran interfaz como en este caso por ejemplo la interfaz del CrudPersonas que es exclusivamente el crud para Persona o la interfaz RepositoryCrud, que es el crud del repositorio.

5 -> **Principio de Inversión de Dependencias**: nuestro código depende de interfaces y de abstract class, como abstract class persona o las interfaces RepositorioCrud.

PATRONES UTILIZADOS

Los patrones que hemos utilizado son:

Patrón CRUD -> en nuestro caso es un sistema de almacenamiento que tiene Crear, Leer, Actualizar, y Borrar, es decir, create, read, update y delete.

- Crear Miembro
- Actualizar Miembro / Actualizar Entrenador / Actualizar Persona /Actualizar Jugador
- Eliminar Miembro
- Cargar datos a fichero

Patrón LRU -> es una cache en la que se elimina el elemento que más tarda en usarse para sustituirlo con elementos más recientes.

Tenemos la parte de CacheImplementationLru en la cual empezamos con una data class llamada cache entrada que encapsula el valor almacenado y el timestamp del ultimo acceso. Dentro de CacheImplementationLru, implementa una cache con capacidad máxima, usa un ConcurrentHasMap para el almacenamiento concurrente y

gestiona automáticamente la eliminación del elemento menos usado cuando se alcanza la capacidad máxima.

EXCEPCIONES

Clase sellada que contiene las excepciones usadas para ayudarnos con la gestión de errores de personas. Los tipos de excepciones creadas son:

- **PersonasStorageException**: indica el problema con el alamcenamiento.

- **PersonaNotFoundException**: esta se lanza cuando no puede encontrar a la persona.

- **PersonasInvalidoException**: indica que los datos de la persona no son válidos.

MAPPER

Es la clase que se encarga de convertir los modelso Jugadores y Entrenadores en sus correspondientes DTO. Usa los siguientes métodos:

- **toDto**(persona: Persona): PersonaDto: convierte una persona en su DTO correspondiente.

- **toModel**(jugadorDto: JugadorDto): Jugadores: convierte un JugadorDto en un objeto Jugadores.

- **toModel**(entrenadorDto: EntrenadorDto): Entrenadores: convierte un EntrenadorDto en un objeto Entrenadores.

CONSULTAS

En este apartado se explican cómo se han realizado y el porqué de todas las consultas:

1 -> Listados de personal agrupados por entrenadores y jugadores:

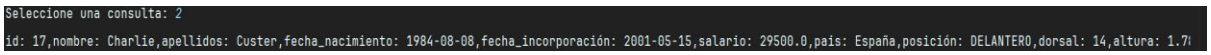
println(lista)

- Imprimes la lista, ya que son los entrenadores y los jugadores

2 -> El delantero más alto:

println(jugadores.filter { it.posicion == Posicion.DELANTERO }.maxByOrNull { it.altura })

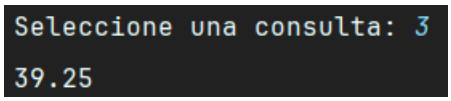
- Filtra los jugadores que su posicion es delantero y encuentra el jugador mas alto con el maxByOrNull



3 -> Media de goles de los delanteros:

println(jugadores.filter{ it.posicion == Posicion.DELANTERO }.map { it.goles }.average())

- Filtra los jugadores por delantero, devuelve una coleccion con map que devuelve una media de goles con el average



4 -> Defensa con más partidos jugados:

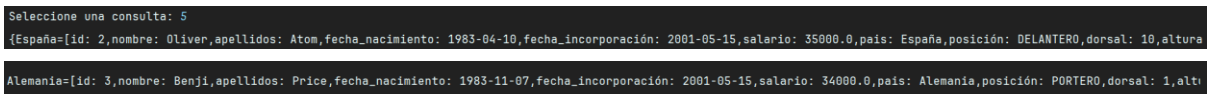
println(jugadores.filter{it.posicion==Posicion.DEFENSA}.maxByOrNull{it.partidosJugados})

- Filtra por la posicion defensa del jugador y encuentra el jugador con mas partidos jugados con el maxByOrNull

5 -> Jugadores agrupados por su país de origen:

println(jugadores.groupBy { it.pais })

- Agrupa los jugadores por pais con el groupBy



6 -> Entrenador con el mayor salario:

println(entrenadores.maxByOrNull { it.salario })

- Se usa el maxByOrNull para encontrar al entrenador con el mayor salario

7 -> Promedio de altura de los jugadores agrupados por posición:

```
jugadores.groupBy { it.posicion }.forEach { (posicion, lista) ->
  val promedioAltura = lista.map { it.altura }.average()
  println("$posicion: $promedioAltura")
```

- Se agrupan los jugadores con el groupBy por posición, luego, se recorre cada grupo de jugadores con un forEach, donde tenemos la posicion y la lista de los jugadores que tienen esa posicion, luego de la lista hacemos el map que nos devuelve la colección con las medias de las alturas, y hacemos el println para imprimirlo

```
Seleccione una consulta: 7
DELANTERO: 1.765
PORTERO: 1.8450000000000002
CENTROCAMPISTA: 1.7349999999999999
DEFENSA: 1.795
```

8 -> Listado de todos los jugadores que han anotado más de 10 goles:

```
println(jugadores.filter { it.goles > 10 })
```

- Filtramos por los jugadores que hayan metido más de 10 goles con el filter.

9 -> Jugadores con un salario mayor al promedio del equipo:

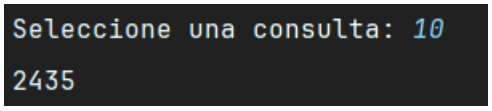
```
val salarioPromedio = jugadores.map { it.salario }.average()
println(jugadores.filter { it.salario > salarioPromedio })
```

- Creamos la variable salarioPromedio en la que hacemos el map del salario y le hacemos la media con el average. Luego a la hora de imprimir, filtramos si el salario es mayor al salarioPromedio creado anteriormente.

10-> Número total de partidos jugados por todos los jugadores:

println(jugadores.sumOf { it.partidosJugados })

- Con el sumOf obtenemmos la suma total de todos los partidos jugados de todos los jugadores que lo indicamos que son jugadores.



11-> Jugadores agrupados por el año de su incorporación al club:

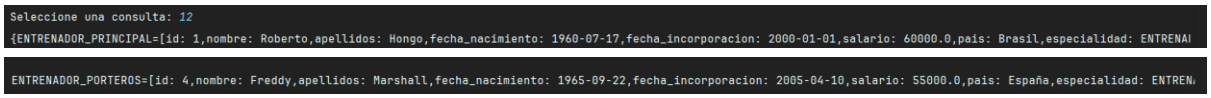
println(jugadores.groupBy { it.fechaIncorporacion.year })

- Agrupamos los jugadores con el groupBy por la fecha de incorporación, y especificamos que sea por el año.

12-> Entrenadores agrupados por su especialidad:

println(entrenadores.groupBy { it.especialidad })

- Agrupamos los entrenadores con el groupBy ya que en este caso especificamos que son entrenadores y los agrupamos por su especialidad.



13-> Jugador más joven en el equipo:

println(jugadores.minByOrNull { it.fechaNacimiento })

- Obtenemos la fecha de nacimiento del jugador más reciente con el minByOrNull y así obtenemos el jugador más joven ya que la fecha de nacimiento más reciente significa que es el jugador más joven.

14-> Promedio de peso de los jugadores por posición:

```
jugadores.groupBy { it.posicion }.forEach { (posicion, lista) ->
   val promedioPeso = lista.map { it.peso }.average()
   println("$posicion: $promedioPeso")
}
```

- Agrupa los jugadores por poscioion con el groupBy y hace que, por cada uno con el forEach, te de la lista de la posición y se crea la variable promedioPeso para calcular la media del peso de los jugadores con el average pero antes hacemos el map del peso, es decir obtenemos una lista con las medias de los pesos y luego imprimimos para que por cada posición nos de la media del peso.

```
Seleccione una consulta: 14
DELANTERO: 67.25
PORTERO: 79.75
CENTROCAMPISTA: 66.0
DEFENSA: 71.5
```

15-> Listado de todos los jugadores que tienen un dorsal par:

```
println(jugadores.filter { it.dorsal.toInt() % 2 == 0 }.map { it.nombre + " " + it.apellidos })
```

Se filtran los jugadores cuyo dorsal sea par, y después se mapea para que la consulta devuelva el nombre y apellidos separados de un espacio.

16 -> Jugadores que han jugado menos de 5 partidos:

```
println(jugadores.filter { it.partidosJugados <= 5})
```

Se filtran los jugadores que tengan una cantidad menor que 5 de partidos jugados.

17 -> Media de goles por partido de cada jugador:

```
jugadores.filter { it.partidosJugados > 0 }.map { it.nombre to it.goles.toDouble() /
it.partidosJugados }.forEach { (nombre, media) ->
println("$nombre: $media goles por partido")}
```

- Se filtran los jugadores que hayan jugado al menos un partido, y después se mapea para que la consulta nos devuelva una lista con el nombre seguido de la media de goles por partido como resultado de esa división. Por último, con el forEach se obtiene por consola cada jugador con su media.

18 -> Listado de jugadores que tienen una altura superior a la media del equipo:

```
fun listadoJugadoresAlturaSuperiorMedia(jugadores:List<Jugadores>): List<Jugadores>
{
val alturaMediaJugadores = jugadores.map { it.altura.toDouble() }.average()
   return jugadores.filter { it.altura > alturaMediaJugadores }
```

- Se crea una variable para calcular la altura media de jugadores, en la que se mapea pasando la altura a Double y aplicando “average” para la media. Después, la consulta devolverá una lista en la que se filtran los jugadores que su altura sea superior a la alturaMedia recientemente calculada.

19 -> Entrenadores que se incorporaron al club en los últimos 5 años:

```
println(entrenadores.filter { (LocalDate.now().year - it.fechaIncorporacion!!.year) <=5 })
// revisar
```

- Se filtran los entrenadores, y se calcula el año actual y se le resta la fechaIncoporación (solo cogiendo el año). Esa resta debe ser menor o igual a 5.

20 -> Jugadores que han anotado más goles que el promedio de su posición:

```
fun jugadoresSobrePromedio(jugadores:List<Jugadores>): List<Jugadores> {
   val promedioGolesPorPosicion = jugadores.filter { it.goles >= 0 }.groupBy { it.posicion }
      .mapValues { (_, jugadores) -> jugadores.map { it.goles }.average() }
   return jugadores.filter { it.goles > (promedioGolesPorPosicion[it.posicion] ?: 0.0)} }
```

- Se crea una variable para calcular el promedio de goles por posición, en el que se filtran los jugadores cuyos goles no sean nulos y se agrupan por posición. Después se mapean los goles para cada grupo y se hace la media y la consulta devuelve una lista de jugadores filtrada de los jugadores cuyos goles son más que el promedio por posición.

21 -> Por posición, máximo de goles, mínimo de goles y media:

```
fun golesPorPosicion(jugadores:List<Jugadores>): Map<Posicion, Triple<Int?, Int?,
Double>> {
   return jugadores.filter { it.goles >= 0 }
      .groupBy { it.posicion }
      .mapValues { (_, jugadores) ->
```

```
        val goles =
            jugadores.map { it.goles.toInt() }
Triple(goles.maxOrNull(), goles.minOrNull(), goles.average())  }}
```

- Se filtran los jugadores cuyos goles no sean nulos, se agrupa por posición, se extrae la cantidad de goles por jugador en cada posición y se transforma a Int, y por último se calcula el máximo, mínimo y la media de goles.

22 -> Estimación del coste total de la plantilla

```
Println(lista.sumOf { it.salario.toDouble() } )
```

- Se calcula la suma del salario de toda la plantilla incluyendo jugadores y entrenadores.

23 -> Total del salario pagado, agrupados por año de incorporación:

```
println(lista.filter { (it.fechaIncorporacion != null) && (it.salario != null) }.groupBy { it.fechaIncorporacion }
   .mapValues { (_, plantilla) -> plantilla.sumOf { it.salario }.toDouble() })
```

- Se filtra la lista de toda la plantilla en la que su fechaIncorporacion no sea nula ni su salario sea nulo, se agrupa por fechaIncoporacion y por cada grupo se suma su salario.

24 -> Jugadores agrupados por país y, dentro de cada grupo, el jugador con más partidos jugados:

```
println(jugadores.groupBy { it.pais }.mapValues { (_, jugadores) -> jugadores.maxByOrNull { it.partidosJugados } })
```

- Se agrupan los jugadores por su país de origen y por cada grupo se extrae el jugador que haya jugado más partidos con el “maxByOrNull”

25 -> Promedio de goles por posición, y dentro de cada posicion, el jugador con mayor numero de goles:

```
fun promedioGolesMaxGoleador(jugadores:List<Jugadores>) {
   val resultados = jugadores.groupBy { it.posicion }
      .mapValues { (_, jugadores) ->
         val promedioGoles = jugadores.map { it.goles }.average()
```

```
        val maxGoleador = jugadores.maxByOrNull { it.goles }
        Pair(promedioGoles, maxGoleador)
      }
    resultados.forEach { (posicion, datos) ->
      val (promedio, maxGoleador) = datos
      println("$posicion -> Promedio de goles: $promedio, Jugador con más goles:
${maxGoleador?.nombre} (${maxGoleador?.goles})")
    }
}
```

26-> Entrenadores agrupados por especialidad, y dentro de cada especialidad, el entrenador con el salario más alto:

```
println(entrenadores.groupBy { it.especialidad }
      .mapValues { (_, entrenadores) -> entrenadores.maxByOrNull { it.salario.toDouble()
} })
```

- Se agrupan los entrenadores por su especialiadad, y despues por cada grupo se saca el entrenador con el salario más alto con el “maxByOrNull”.

27 -> Jugadores agrupados por década de nacimiento, y dentro de cada grupo, el promedio de partidos jugados:

```
fun jugadoresPorDecada(jugadores:List<Jugadores>): Map<String, Double> {
    return jugadores.groupBy {
      val year = it.fechaNacimiento?.year ?: 0
      "${(year / 10) * 10}s"
    }.mapValues { (_, jugadores) -> jugadores.map { it.partidosJugados.toInt() }.average() }
}
```

- Se agrupan los jugadores por su fecha de nacimiento, la cual la pasamos a 0 si fuese nula, dividimos el año entre 10 para posteriormente multiplicar por 10 y así quitariamos el ultimo digito. Así se transforma a década y añadimos “s” para leerlo mejor. Después, mapeamos los partidos jugados de cada jugador y calculamos la media de cada grupo.

28 -> Salario promedio de los jugadores agrupados por su país de origen, y dentro de cada grupo, el jugador con el salario más bajo y alto:

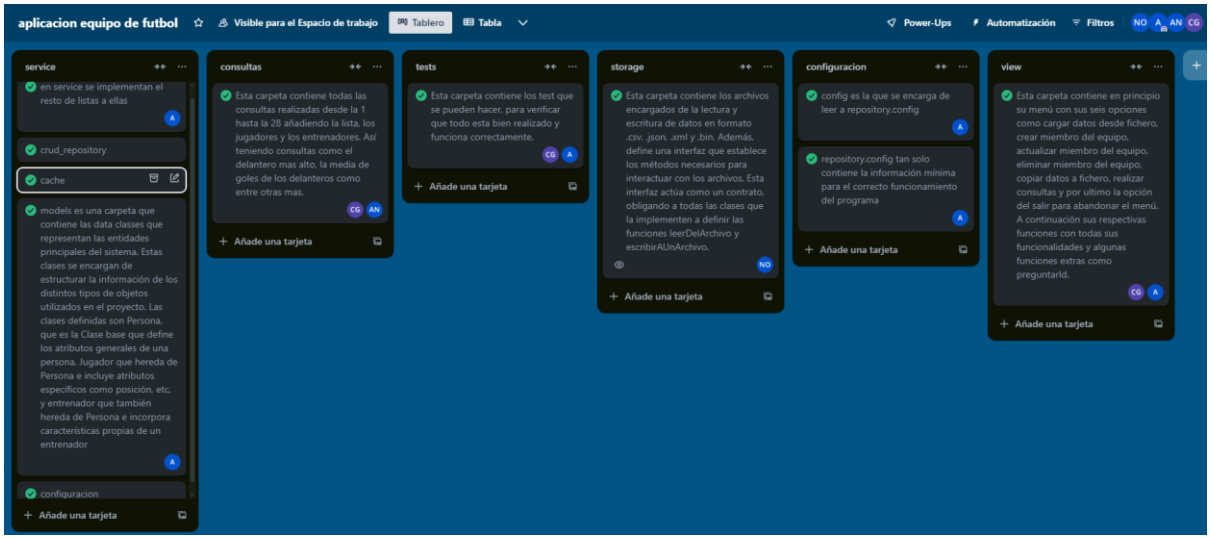
```
fun salarioPromedioPaisEstadisticas(jugadores:List<Jugadores>): Map<String,
Triple<Double, Double?, Double?>> {
   return jugadores.groupBy { it.pais }
      .mapValues { (_, jugadores) ->
         val salarios = jugadores.map { it.salario.toDouble() }
         Triple(salarios.average(), salarios.minOrNull(), salarios.maxOrNull()) }
}
```

- Se agrupan los jugadores por su país y se mapean los salarios de los jugadores. También se calculan el promedio de cada grupo y el salario más alto y más bajo de cada uno.

ESTIMACION ECONOMICA

La estimación económica lo hemos hecho con la suma de las horas realizadas por todos los integrantes del grupo y suma una cantidad de 600€ en total.

TRELLO



RIESGOS

Los riesgos que existen a la hora de desarrollar este tipo de proyectos son principalmente mentales, estrés, fatiga por sobre trabajo, etc. Sobre todo son riesgos de que el programador se ponga de los nervios debido a la alta exigencia. En nuestros

casos ha habido momentos de estrés donde ha sido complicado gestionar las emociones, pero al final se ha conseguido sacar todo. La fatiga también nos ha afectado principalmente los últimos días.