DP1 2020-2021

Documento de Diseño del Sistema

Volleyball

https://github.com/gii-is-DP1/dp1-2020-g3-02

Miembros <en orden alfabético por apellidos>:

- Laura Castillo Ortíz
- Benjamín Crespo Alcaide
- Diego Manuel Gil Losa
- Javier Gutiérrez Falcón
- Gonzalo Lallena Alva
- Blanca del Rosario Mauri Robles

Tutor: José Antonio Parejo Maestre

GRUPO G3-02

Versión 1

28/12/2020

Historial de versiones

Estos son ejemplo del contenido que debería tener el historial de cambios del documento a entregar a lo largo de los sprints del proyecto

Fecha	Versión	Descripción de los cambios	Sprint
13/12/2020	V1	● Creación del documento	2
13/12/2020	V2	● Añadido diagrama de dominio/diseño ● Explicación de la aplicación del patrón caché	3

Contents

DP1 2020/21
Volleyball Grupo: G3-02

Esta es una plantilla que sirve como guía para realizar este entregable. Por favor, mantén las mismas secciones y los contenidos que se indican para poder hacer su revisión más ágil.

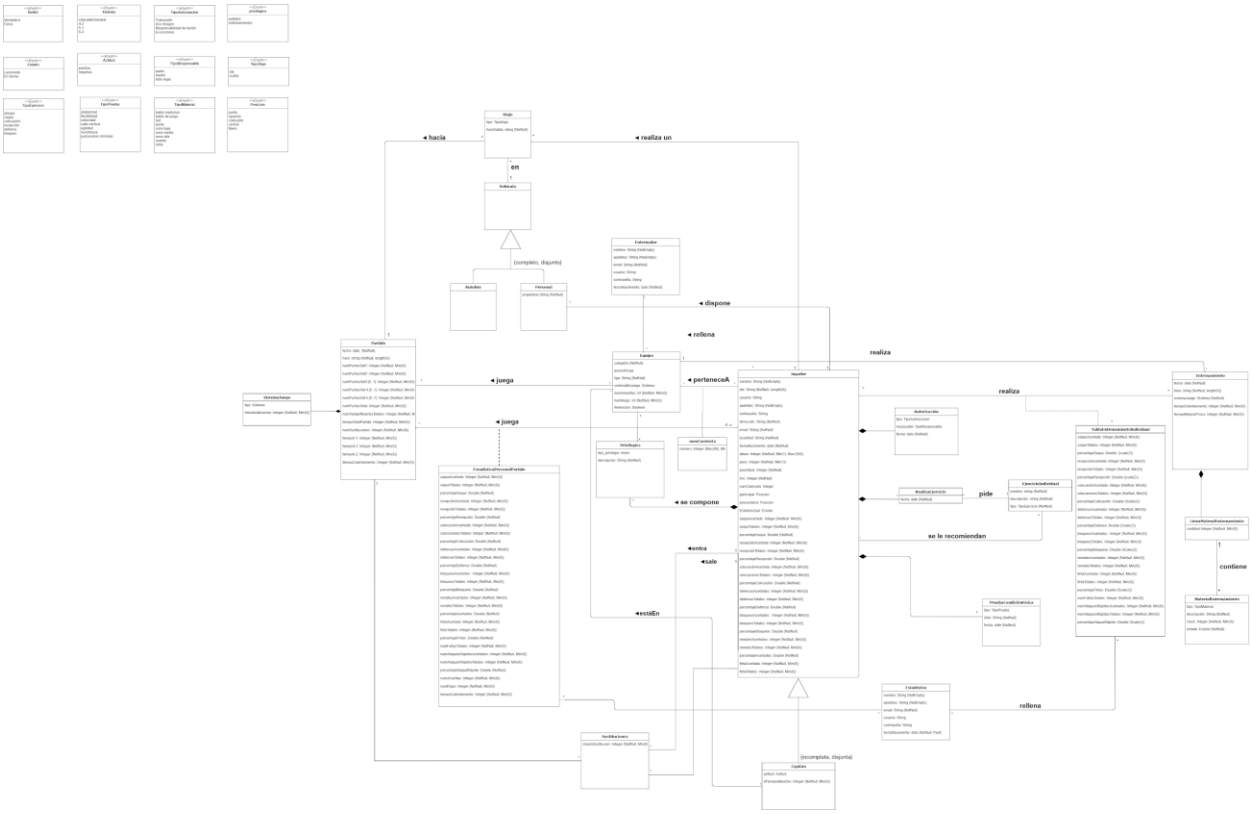
Introducción

En esta sección debes describir de manera general cual es la funcionalidad del proyecto a rasgos generales (puedes copiar el contenido del documento de análisis del sistema). Además puedes indicar las funcionalidades del sistema (a nivel de módulos o historias de usuario) que consideras más interesantes desde el punto de vista del diseño realizado.

Diagrama(s) UML:

Diagrama de Dominio/Diseño

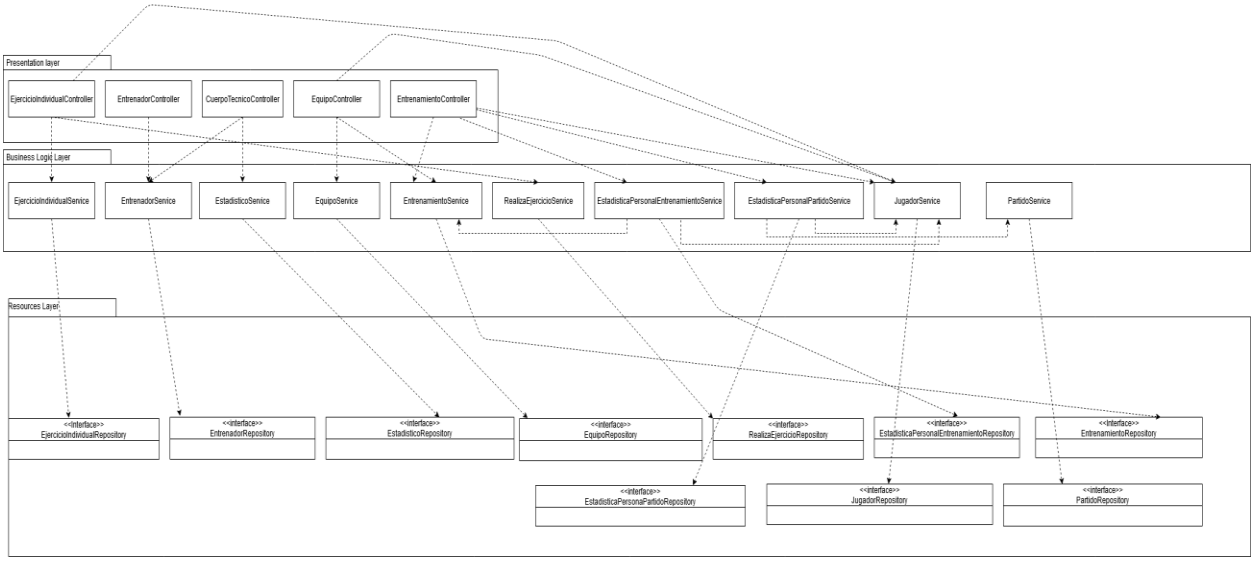
Hemos omitido las generalizaciones hacia BaseEntity para simplificar el diagrama:



https://viewer.diagrams.net/?highlight=0000ff&edit=_blank&layers=1&nav=1&title=Modelado%20Conceptual%20Voleibol%20entrega(sin%20RN)%20(1)%20(1)%20(3)%20(2).xml#Uhttps%3A%2F%2Fdrive.google.com%2Fuc%3Fid%3D155i6xnmi8DeoiZCMoKDgyQN_gRQ6MZAL%26export%3Ddownload

Diagrama de Capas (incluyendo Controladores, Servicios y Repositorios)

En esta sección debe proporcionar un diagrama UML de clases que describa el conjunto de controladores, servicios, y repositorios implementados, incluya la división en capas del sistema como paquetes horizontales tal y como se muestra en el siguiente ejemplo:

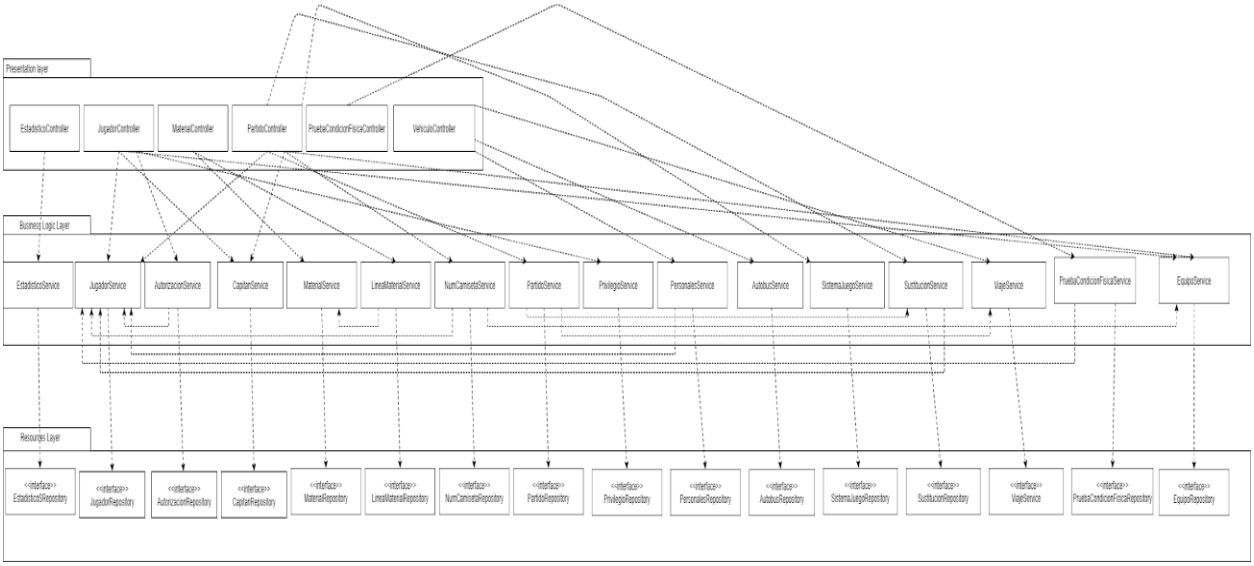


Nombre del Repositorios	Funciones añadidas al repositorio
EjercicioIndividualRepository	❖ Optional<EjercicioIndividual> findByNombre(String nombre); ❖ List<EjercicioIndividual> findByNombreContaining(String nombre); ❖ List<EjercicioIndividual> findByTipoEjercicio(TipoEjercicio tipo_ejercicio);
EstadisticoRepository	❖ List<Estadistico> findByFirstName(String name); ❖ Estadistico findByEmail(String email); ❖ List<Estadistico> findByFechaNacimientoBetween OrderByFechaNacimiento(LocalDate firstDate, LocalDate secondDate); ❖ List<Estadistico> findByFechaNacimientoAfterOrderByFechaNacimiento(LocalDate date); ❖ Estadistico findByUser(User user);
EntrenadorRepository	❖ List<Entrenador> findByFirstName(String name);

	❖ Entrenador findByEmail(String email); ❖ Entrenador findByUser(User user); ❖ List<Entrenador> findByFechaNacimientoBetweenOrderByFechaNacimiento(LocalDate firstDate, LocalDate secondDate); ❖ List<Entrenador> findByFechaNacimientoAfterOrderByFechaNacimiento(LocalDate date);
EntrenamientoRepository	❖ List<Entrenamiento> findByEquipoOrderByFecha(Equipo team); ❖ List<Entrenamiento> findByFechaOrderByHora(LocalDate date); ❖ List<Entrenamiento> findByEquipo(Equipo equipo);
EquipoRepository	❖ Equipo findByCategoria(String category); ❖ List<Equipo> findByCategoriaStartingWith(String category); ❖ List<Equipo> findByLiga(String league); ❖ Equipo findByEntrenadorAndCategoria(Entrenador entrenador, String categoria); ❖ List<Equipo> findByEntrenador(Entrenador entrenador); ❖ List<Equipo> findByCapitan(Capitan capitan); ❖ List<Equipo> findByJugador(int idJugador); ❖ List<String> findCategoria();
RealizaEjercicioRepository	❖ List<RealizaEjercicio> findByFecha(LocalDate fecha); ❖ List<RealizaEjercicio> findByEjercicioIndividual(EjercicioIndividual ejercicio); ❖ List<RealizaEjercicio> findByJugador(int jugador_id);
EstadisticaPersonalPartidoRepository	❖ List<EstadisticaPersonalPartido>

	findByPartido(Partido partido); ❖ List<EstadisticaPersonalPartido> findByJugador(Jugador jugador); ❖ EstadisticaPersonalPartido findByJugadorAndPartido(Jugador jugador, Partido partido);
EstadisticaPersonalEntrenamientoRepository	❖ List<EstadisticaPersonalEntrenamiento> findByEntrenamiento(Entrenamiento entrenamiento); ❖ List<EstadisticaPersonalEntrenamiento> findByJugador(Jugador jugador); ❖ EstadisticaPersonalEntrenamiento findByJugadorAndEntrenamiento(Jugador jugador, Entrenamiento entrenamiento);
PartidoRepository	❖ List<Partido> findByFechaOrderByHora(LocalDate date); ❖ List<Partido> findByFechaAfter(LocalDate date); ❖ List<Partido> findByEquipoAndFechaAndHoraBetween(Equipo equipo, LocalDate fecha, String hora1, String hora2); ❖ List<Partido> findByEquipo(Equipo equipo);
JugadorRepository	❖ List<Jugador> findByFirstName(String name); ❖ Jugador findByUser(User username); ❖ Jugador findByEmail(String email); ❖ List<Jugador> findByPosicionPrincipal(Posicion position); ❖ List<Jugador> findByFechaNacimientoBetweenOrderByFechaNacimiento(LocalDate firstDate, LocalDate secondDate); ❖ List<Jugador> findByFechaNacimientoAfterOrderByFechaNacimiento(LocalDate date); ❖ List<Jugador> findByAlturaGreaterThanEqual(int height);

	❖ List<Jugador> findByAlturaLessThanEqual(int height); ❖ List<Jugador> findByPesoGreaterThanEqual(int weight); ❖ List<Jugador> findByPesoLessThanEqual(int weight); ❖ List<Jugador> findAuto(TipoAutorizacion autorizacion); ❖ List<Jugador> findPrivilegio(TipoPrivilegio privilegio); ❖ List<Jugador> findByEquipo(int equipo_id);



Nombre del Repositorio	Funciones del Repositorio
AutorizacionRepository	❖ List<Autorizacion> findByFecha(LocalDate date); ❖ List<Autorizacion> findByTipoAutorizacion(TipoAutorizacion tipo); ❖ List<Autorizacion> findByJugador(Jugador jugador); ❖ Autorizacion findByJugadorAndTipoAutorizacion(Jugador jugador, TipoAutorizacion tipo); ❖ List<Jugador> findJugadorByTipoAutorizacion(TipoAutorizacion tipoautorizacion);
CapitanRepository	❖ List<Capitan> findByActitud(Actitud actitud); ❖ List<Capitan> findByNtiemposmuertos(Integer ntiemposmuertos); ❖ public Capitan findByJugador(Jugador jugador); ❖ List<Capitan> findByEquipo(@Param("equipo_id") int equipo_id);

MaterialRepository	❖ List<Material> findByTipo(TipoMaterial tipo); ❖ List<Material> findByDescripcion(String descripcion); ❖ List<Material> findByStock(Integer stock);
LineaMaterialRepository	❖ List<LineaMaterial> findByMaterial(Material material); ❖ List<LineaMaterial> findByEntrenamiento(Entrenamiento entrenamiento);
NumCamisetaRepository	❖ List<NumCamiseta> findByNumero(Integer numero); ❖ List<NumCamiseta> findByJugador(Jugador jugador); ❖ List<NumCamiseta> findByEquipo(Equipo equipo); ❖ NumCamiseta findByEquipoAndJugador(Equipo equipo, Jugador jugador);
PrivilegioRepository	❖ List<Privilegio> findByJugador(Jugador jugador); ❖ List<Privilegio> findByEquipo(Equipo equipo);
PersonalesRepository	❖ List<Personales> findByPropietario(String propietario); ❖ List<Personales> findByJugador(Jugador jugador);
AutobusRepository	
SistemaJuegoRepository	❖ List<SistemaJuego> findByminutoAplicacion(int minuto_aplicacion); ❖ List<SistemaJuego> findBySistema(Sistema sistema); ❖ List<SistemaJuego> findByPartido(int partido_id);
SustitucionRepository	❖ List<Sustitucion> findByJugadorEntra(Jugador jugador); ❖ List<Sustitucion>

	findByJugadorSale(Jugador jugador); ❖ List<Sustitucion> findByPartido(Partido partido);
ViajeRepository	❖ List<Viaje> findByJugador(Jugador jugador); ❖ List<Viaje> findByPartido(Partido partido); ❖ List<Viaje> findByTipoViaje(TipoViaje tipoViaje); ❖ List<Viaje> findByAutobus(Autobus autobus); ❖ List<Viaje> findByPersonal(Personales personal); ❖ Viaje findByJugadorAndPartidoAndTipoViaje(Jugador jugador, Partido partido, TipoViaje tipoViaje); ❖ List<Viaje> findByJugadorAndTipoViaje(Jugador jugador, TipoViaje tipoViaje); ❖ List<Viaje> findByPartidoAndTipoViaje(Partido partido, TipoViaje tipoViaje); ❖ List<Viaje> findByJugadorAndPersonal(Jugador jugador, Personales personal);
PruebaCondicionFisicaRepository	❖ List<PruebaCondicionFisica> findByTipoPrueba(TipoPrueba tipo_prueba); ❖ List<PruebaCondicionFisica> findByDatoLessThanEqual(double dato); ❖ List<PruebaCondicionFisica> findByJugadorAndTipoPrueba(Jugador jugador, TipoPrueba tipo_prueba); ❖ List<PruebaCondicionFisica> findByDatoAndTipoPrueba(double dato, TipoPrueba tipo_prueba); ❖ List<PruebaCondicionFisica> findByTipoPruebaAndDatoLessThanEqual(TipoPrueba tipo_prueba, double dato); ❖ List<PruebaCondicionFisica> findByJugador(Jugador jugador);

EquipoRepository	❖ Equipo findByCategoria(String category); ❖ List<Equipo> findByCategoriaStartingWith(String category); ❖ List<Equipo> findByLiga(String league); ❖ Equipo findByEntrenadorAndCategoria(Entrenador entrenador, String categoria); ❖ List<Equipo> findByEntrenador(Entrenador entrenador); ❖ List<Equipo> findByCapitan(Capitan capitan); ❖ List<Equipo> findByJugador(int idJugador); ❖ List<String> findCategoria();

El diagrama debe especificar además las relaciones de uso entre controladores y servicios, entre servicios y servicios, y entre servicios y repositorios.

Tal y como se muestra en el diagrama de ejemplo, para el caso de los repositorios se deben especificar las consultas personalizadas creadas (usando la signatura de su método asociado).

Patrones de diseño y arquitectónicos aplicados

Patrón: Composite

Tipo: Diseño

Contexto de Aplicación

Este patrón de diseño se ha utilizado sobretodo para abstraer código en la parte de los servicios. Se puede ver cláramente en los paquetes dentro de nuestra aplicación: service.base y la implementación de estas interfaces en sus clases abstractas dentro del paquete service.base.impl.

Clases o paquetes creados

Los paquetes serán los mencionados arriba, y las clases creadas serán:

- service.base:
  - BaseService(interfaz)
  - BaseEstadisticasService(interfaz, extiende de BaseService)
- service.base.impl:
  - AbstractService(implementación de la interfaz BaseService, que será la fábrica genérica de todos los servicios y entidades que estos traen)
  - AbstractEstadisticasService(implementación de la interfaz BaseEstadisticasService, que será la fábrica genérica de todos los servicios y entidades que además poseen estadísticas)

Ventajas alcanzadas al aplicar el patrón

Con este patrón ahorramos muchísimo código repetido, y nos permite reutilizar muchos métodos idénticos para todos los servicios. Nos encontrábamos con el problema de tener que acceder a las funciones básicas de los repositorios, ya sean findById, save, delete… Por no hablar del acceso a los datos de las entidades que poseían estadísticas, lo que abultaba muchísimo. Con estas factorías de métodos, hemos reutilizado muchísimo código, hemos hecho los servicios mucho más legibles y orientados únicamente a la función específica que les correspondía, y hemos aislado mucho código que funciona y no debe ser reescrito en clases aisladas, lo que facilita el entendimiento y mantenimiento de los servicios. No hace falta decir que si una clase requiere sobrescribir un método de la clase padre, no hay más que reescribir este método en el servicio concreto.

Patrón: Fachada/DTO

Tipo: Diseño

Contexto de Aplicación

El patrón de fachada se ha usado sobre todo en los modelos, creando los paquetes auxiliares, ediciones y estadísticas y sus convertidores correspondientes en el paquete converter.

Clases o paquetes creados

Los paquetes son los vistos arriba, y las clases todas las incluidas en estos paquetes:

- auxiliares:
    - EjercicioIndividualDTO
    - EquipoTablaEquipos
    - JugadorAut
    - JugadoresInEquipo
    - JugadorWithEquipo
    - MaterialEstados
    - PartidoConAsistencia
    - PruebasSinJugador
    - RealizarEjerciciosDTO
- ediciones:
    - EquipoEdit
    - JugadorEdit
    - MaterialDTO
    - PartidoEdit
    - PrivilegioEdit
- estadisticas:
    - EntrenamientoStats
    - EquipoStats
    - EstadisticasDeUnJugadorStats
    - EstadisticasPersonalesStats
    - JugadorPartidoStats

- JugadorStats
- PartidoStats

Ventajas alcanzadas al aplicar el patrón

Este patrón nos permite llevar siempre a la vista únicamente los datos que necesitamos para cada ocasión. Nuestras entidades tienen muchas relaciones y dependencias con otras ya que nuestro diagrama de clases es extenso y complejo, esto hacía que siempre se tuvieran que llevar a las vistas una cantidad de datos excesiva. Para ello, hemos ido generando clases auxiliares intermedias y los convertidores correspondientes, para así reducir el peso de los datos que van hacia las vistas. Al haber dividido el trabajo, se han creado algunas clases de más ya que no hemos tenido tiempo de hacer alguna clase padre para muchas de estas clases fachada, si queda tiempo, se plantea reducir estas clases realizando esa solución.

Patrón: Modelo-Vista-Controlador

Tipo: Arquitectónico

Contexto de Aplicación

El patrón arquitectónico ha sido aplicado en toda la aplicación en general, ya que spring propone este tipo de arquitectura, y se ha desarrollado toda la aplicación con este framework.

Clases o paquetes creados

Los paquetes creados para la utilización de este patrón han sido: model, service, controller y toda la carpeta de resources con la parte de las vistas. Omito aquí las clases ya que son muchas, pero están todas dentro de los paquetes enumerados anteriormente.

Ventajas alcanzadas al aplicar el patrón

Este patrón nos permite aislar muy bien toda la lógica de la aplicación de la parte de la interfaz y control. Tiene como ventajas también que nos permite hacer pruebas de forma más aislada, y poder detectar más fácilmente los errores a través de pruebas unitarias o debug.

Decisiones de diseño

Decisión 1: Importación de datos reales para demostración

Descripción del problema:

Como grupo nos gustaría poder hacer pruebas con un conjunto de datos reales suficientes, porque resulta más motivador. El problema es que al incluir todos esos datos como parte del script de inicialización de la base de datos, el arranque del sistema para desarrollo y pruebas resulta muy tedioso.

Alternativas de solución evaluadas:

Alternativa 1.a: Incluir los datos en el propio script de inicialización de la BD (data.sql).

Ventajas:

- Simple, no requiere nada más que escribir el SQL que genere los datos.

Inconvenientes:

- Ralentiza todo el trabajo con el sistema para el desarrollo.
- Tenemos que buscar nosotros los datos reales

Alternativa 1.b: Crear un script con los datos adicionales a incluir (extra-data.sql) y un controlador que se encargue de leerlo y lanzar las consultas a petición cuando queramos tener más datos para mostrar.

Ventajas:

- Podemos reutilizar parte de los datos que ya tenemos especificados en (data.sql).
- No afecta al trabajo diario de desarrollo y pruebas de la aplicación

Inconvenientes:

- Puede suponer saltarnos hasta cierto punto la división en capas si no creamos un servicio de carga de datos.
- Tenemos que buscar nosotros los datos reales adicionales

Alternativa 1.c: Crear un controlador que llame a un servicio de importación de datos, que a su vez invoca a un cliente REST de la API de datos oficiales de XXXX para traerse los datos, procesarlos y poder grabarlos desde el servicio de importación.

Ventajas:

- No necesitamos inventarnos ni buscar nosotros lo datos.
- Cumple 100% con la división en capas de la aplicación.
- No afecta al trabajo diario de desarrollo y pruebas de la aplicación

Inconvenientes:

- Supone mucho más trabajo.
- Añade cierta complejidad al proyecto

Justificación de la solución adoptada

Dado que no existe un conjunto de datos reales en ningún lugar, podemos pedir los datos a papel de las estadísticas recogidas de algunos partidos reales dentro del club, por lo que para crear una aplicación, nos valdría con introducir esos datos en el data.sql hasta la aprobación del cliente, donde ya al ver la aplicación, creemos la base de datos persistente, por lo que tomaremos la alternativa 1.a.

Decisión 2: Validación de datos al crear cualquier nuevo objeto

Descripción del problema:

Prácticamente todas las entidades de nuestra aplicación requieren validaciones complejas, por lo que debemos tomar una decisión respecto a cómo podemos de forma exhaustiva, simple y al mismo tiempo clara para el usuario.

Alternativas de solución evaluadas:

No incluiré las ventajas e inconvenientes ya que son alternativas dadas en clase, se conocen las ventajas e inconvenientes.

Alternativa 2.a: Validar los campos a través de excepciones, algunas personalizadas y otras genéricas.

Alternativa 2.b: Crear un validador personalizado, con cada una de las reglas de negocio controladas dentro del mismo y lanzando un mensaje de error para cada incumplimiento de las mismas.

Justificación de la solución adoptada

Debido a que es más sencillo seguir unos pasos repetitivos y constantes para la creación de una rama desde la entidad hasta la vista, se tomó la decisión de en el momento antes de crear el controlador, acceder a las reglas de negocio de esa entidad concreta y a veces de sus relaciones si era necesario, y por orden ir escribiendo las condiciones necesarias para cada una de ellas en un validator único que sólo tocaría la persona que crea la clase, por esto hemos optado en hacer las validaciones en servidor(ya que no es una aplicación con excesivos cambios y una recarga con ajax haría lo suficientemente rápida esta validación) utilizando la alternativa 2.b.

Decisión 3: Política de Logs para detectar errores durante el uso de la aplicación.

Descripción del problema:

La aplicación se usará por usuarios para pruebas además de que unos desarrolladores probarán la parte de otros, es bueno tener una buena política de logs para poder encontrar el error rápidamente y así poder arreglarlo lo antes posible, lo que facilitará el desarrollo y mantenimiento de la aplicación.

Alternativas de solución evaluadas:

Alternativa 3.a: Activar los logs de DEBUG propios de Spring.

Alternativa 3.b: Crear una clase de auditoría, que nos permite conocer quién ha creado y actualizado la clase en cada momento, lo que controla el uso indebido de la aplicación. Esto generaría logs cada vez que se hiciera una creación o edición, aunque debido a que no usamos borrado lógico, no controlaríamos los borrados.

Alternativa 3.c: Generar logs siempre que se haga una operación de modificación de datos, tanto creación, edición o borrado, ya que tenemos muchas relaciones y son momentos delicados que debemos tener controlados. Se incluirían logs de WARN en caso de que falle alguna validación, y un ERROR en caso de que falle la modificación y obtenga una excepción inesperada.

Justificación de la solución adoptada

Descartamos directamente la alternativa de auditorías, debido a que la aplicación en principio es de un club, no va a usarla una gran cantidad de personas y es fácil controlar los permisos de los que pueden realizar cambios en los datos, que serán los entrenadores y algún jugador suelto con ciertos permisos especiales, por lo que no es necesario saber quien toca cada objeto de la aplicación. Por otra parte, el controlar cualquier cambio en la persistencia de la base de datos nos parece muy importante ya que es una parte muy delicada, esto exige más trabajo a la hora de escribir los logs en los puntos concretos, pero nos permite un gran control de las zonas más vulnerables. Además, hemos decidido optar por la alternativa 3.a junto con la 3.c, ya que aunque sea algo redundante, los logs de DEBUG de Spring nos detalla exactamente donde entramos y qué devuelve en cada momento un controlador. En el momento de pasar la aplicación a producción, la opción de los logs de Spring de DEBUG quedarán deshabilitadas, para que en el momento en que un usuario use la aplicación y nos envíe los logs en caso de algún problema, sea más fácilmente entendible para los desarrolladores que deban corregir el problema, ya que todos los logs donde se modifican datos estarán controlados con la alternativa 3.c.

Decisión 4: Elección del motor de plantillas

Descripción del problema:

La selección de un motor de plantillas es imprescindible, ya que necesitamos gestionar los datos llevados desde el back-end en la vista, tenemos una gran cantidad de entidades y diferentes tipos de datos que requerimos gestionar a veces en las vistas, por ello esta decisión es importante.

Alternativas de solución evaluadas:

Alternativa 4.a: Uso de JSP para el tratado de datos en las vistas.

Ventajas:

- Tecnología que tiene mucho recorrido en el mercado.
- El procesamiento de las plantillas es muy rápido, en aplicaciones donde la velocidad sea crítica sería planteable siempre usar JSP.

Inconvenientes:

- No es html, por lo que las plantillas se vuelven mucho más difíciles de leer.
- Debemos tener cargado siempre el servidor para poder ver el aspecto de la vista, por lo que dificulta el maquetado y prototipado.

Alternativa 4.b: Uso de Thymeleaf para el tratado de datos en las vistas.

Ventajas:

- Se añaden expresiones a un html, por lo que es muy legible sin añadir ninguna etiqueta extra.
- Permite hacer cambios rápidos en la plantilla sin necesidad de cargar el servidor, por lo que facilita muchísimo el maquetado y el prototipado de las vistas.
- Usado con Spring tiene una serie de funcionalidades que son muy ricas para el desarrollo

Inconvenientes:

- No es tan rápido como otros motores.

Justificación de la solución adoptada

Nosotros tratamos los datos en las plantillas con ajax, esto nos permite no preocuparnos mucho por la velocidad, ya que está compensada por esta parte, esto nos deja un claro ganador en ventajas, que es thymeleaf. Es cierto que es una tecnología con la que no hemos trabajado antes, pero al ser más legible hace más rápido el aprendizaje de la tecnología.

Decisión 5: Visualización de los datos en las vistas

Descripción del problema:

Como he mencionado varias veces en el documento, tenemos muchos datos de diferentes tipos, por lo que hay que decidir de forma adecuada cómo mostrarlos a los usuarios. Puesto que habrá datos de todo tipo es de lógica que se usen tablas, pero debemos controlar paginación, resoluciones, búsqueda de datos concretos… Para esto tenemos las siguientes alternativas.

Alternativas de solución evaluadas:

Alternativa 5.a: Mostrar los datos en una tabla normal, con scroll lateral en caso de que sea más larga que la resolución de la pantalla donde se visualice, controlando la paginación en el back-end, y realizando búsquedas con datos que pasemos al controlador en un conjunto, donde buscaremos datos concretos que traeremos a la vista recargando la página.

Ventajas:

- Conocemos los pasos y es mucho más simple de controlar ya que todo se controla en el back-end.
- Menos cantidad de datos traídos por llamada a la vista.
- Búsqueda exhaustiva de los datos.

Inconvenientes:

- Muy lento, se deben hacer llamadas continuamente.
- El scroll lateral es funcional pero puede resultar incómodo en ciertas situaciones.
- Mayor complejidad en caso de que queramos facilitar la selección del número de registros que se van a mostrar cada vez.

Alternativa 5.b: Usar DataTables, una tecnología muy usada para la gestión de datos en vistas usando tablas.

Ventajas:

- Evita que usemos el scroll lateral, permitiendo el uso de una de sus propiedades para las resoluciones responsive que es muy cómoda para el usuario.
- Permite el paginado directamente en la vista, lo que da velocidad.
- Búsqueda genérica o exhaustiva por JavaScript, según decidamos.
- Una gran cantidad de funcionalidades y fácil gestión de las mismas mediante Ajax y jQuery.
- Combinación con otras tecnologías, como por ejemplo HighCharts.
- Una documentación muy rica, y con una comunidad considerable, por lo que la gestión de errores es fácil de tratar.

Inconvenientes:

- Genera scripts grandes debido a la cantidad de atributos que se deben controlar.
- En un principio no conocemos la tecnología.

Justificación de la solución adoptada

Al conocer la existencia de DataTable nos pusimos a investigar sobre la tecnología, ya que ofrecía una excelente funcionalidad y nos facilitaba el uso de Ajax para dar una gran fluidez a la página, y descubrimos que lo que ofrecía hacía el tratamiento de datos mucho más sencillo en las vistas para el usuario, por lo que decidimos, aunque requiriese algo de más esfuerzo, optar por la alternativa 5.b.

Decisión 6: Visualización de las estadísticas

Descripción del problema:

Las estadísticas son la parte más importante de toda la aplicación, pero también son las más complicadas de visualizar de una forma cómoda y fácilmente legible para el usuario.

Alternativas de solución evaluadas:

Alternativa 6.a: Mostrar las estadísticas con tablas, intentando no ocupar mucho espacio para que al mismo tiempo sea fácil de leer y de comparar unas con otras, creando las funciones en javaScript desde cero para generar las gráficas totalmente a la carta.

Ventajas:

- Gran libertad de elección de estilos y total libertad en la funcionalidad ya que está hecho todo de cero.
- No habría que pagar ninguna licencia ya que sería de nuestra propiedad.

Inconvenientes:

- Coste de formación excesivo, requiere de una búsqueda exhaustiva de información y de pruebas en cliente.
- Hay demasiados campos en estadísticas, por lo que sería difícil gestionar el espacio para que se vieran todas las estadísticas a la vez en la pantalla con las tablas.
- Es una tecnología creada por nosotros en poco tiempo, muy vulnerable a fallos ya que aprenderemos a usarla al tiempo que la desarrollamos.
- Requiere de muchísimo tiempo para el desarrollo.

Alternativa 6.b: Usar Highcharts, una tecnología orientada a la generación de gráficos de diferentes tipos que para estadísticas son muy visuales.

Ventajas:

- DataTable combina muy bien con esta tecnología.
- Hay muchos tipos de gráficas y da mucha libertad para elegir el que más convenga en cada momento.
- Una muy rica funcionalidad, que combina muy bien con Ajax y con jQuery, lo que también da velocidad a la gestión de datos.
- La visualización de estadísticas con tablas es impecable desde un punto de vista de usuario.

Inconvenientes:

- Genera scripts grandes debido a la cantidad de atributos que se deben controlar.
- Menos libertad en el diseño.
- En un principio no conocemos la tecnología.

Justificación de la solución adoptada

Investigando DataTables descubrimos HighCharts, con una documentación bastante buena, nos venía perfecto para tratar las estadísticas, ya que la alternativa 6.a era la única hasta conocer la existencia de esta tecnología, por lo que aunque nos llevase algo de tiempo de entender, era menos tiempo que la primera alternativa, investigamos y vimos viable optar por la alternativa 6.b.