DP1 2020-2021

Documento de Diseño del Sistema

Proyecto MyGym

https://github.com/gii-is-DP1/MyGym

Miembros:

- Manuel Ales Rogríguez
- Jose Manuel Lobato Troncoso
- Manuel Outeiriño Barneto
- Borja Vera Casal

Tutor: Irene Bedilia Estrada Torres

GRUPO G3-05

Versión 1

13/01/2021

Historial de versiones

Fecha	Versión	Descripción de los cambios	Sprint
13/01/2021	V1	• Creación del documento	3
23/01/2021	V3	Patrones de diseño	4
07/02/2021	V4	Decisión de diseño	4
09/02/2021	V4	Decisión de diseño	4

Contents

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

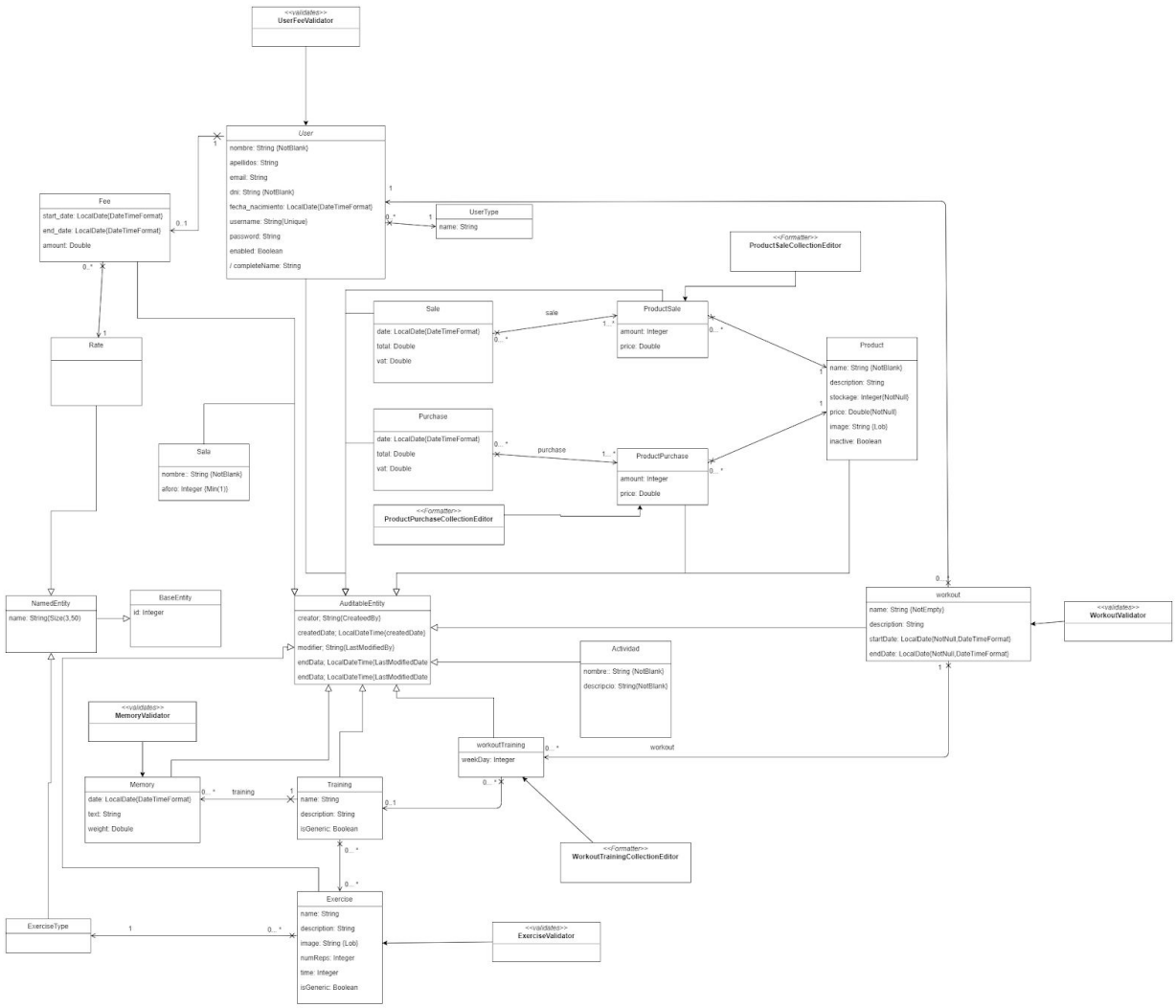
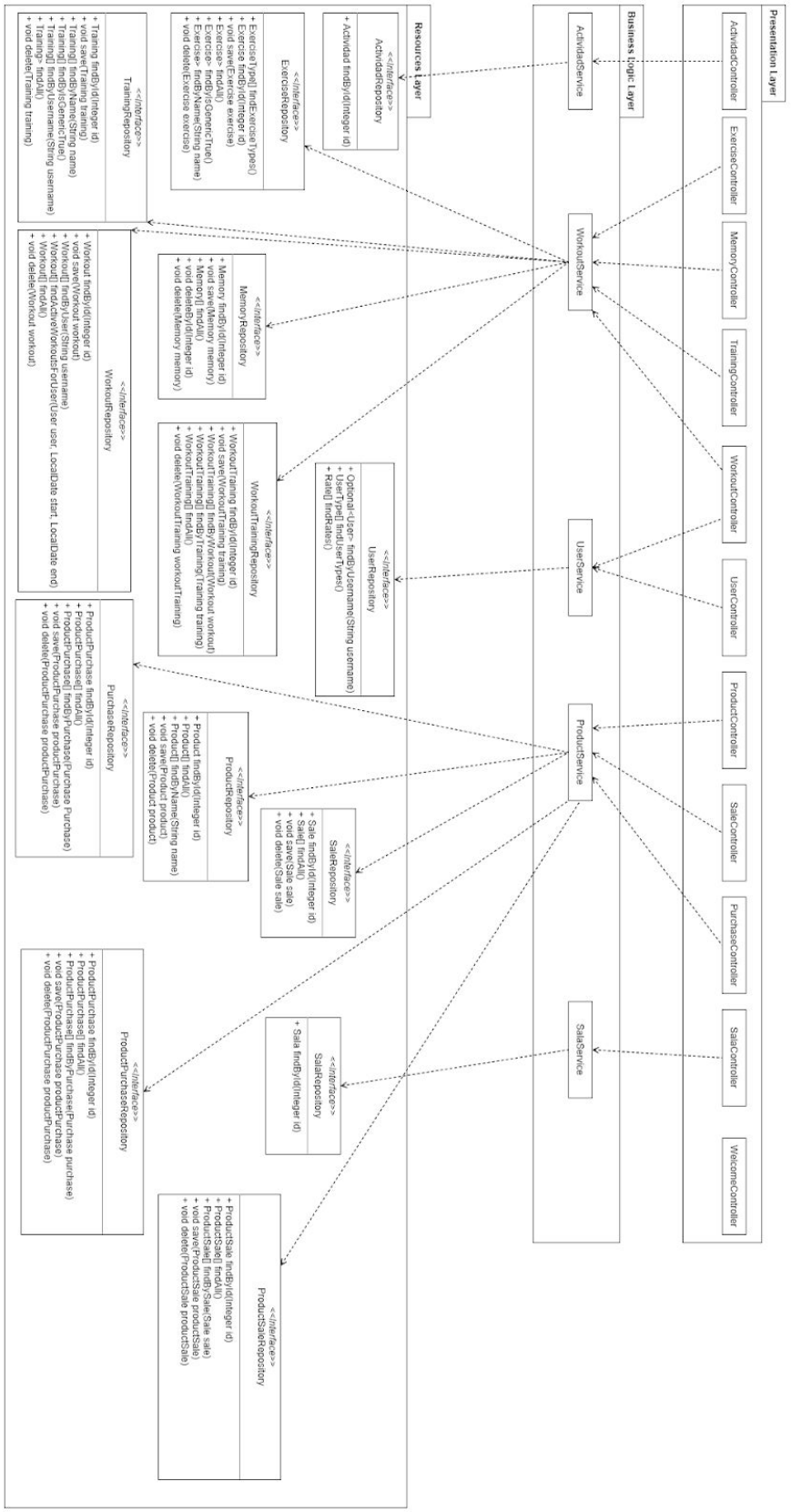


Diagrama de Capas (incluyendo Controladores, Servicios y Repositorios)



Patrones de diseño y arquitectónicos aplicados

En esta sección de especificar el conjunto de patrones de diseño y arquitectónicos aplicados durante el proyecto. Para especificar la aplicación de cada patrón puede usar la siguiente plantilla:

Patrón: MVC

Tipo: De diseño

Contexto de Aplicación

Se habilita un paquete donde se declaran todos los controladores de las distintas vistas definidas en los distintos ficheros *jsp* y que gestionan el modelo de la aplicación ("org.springframework.samples.petclinic.model").

Cada controlador contiene un mapeo con las distintas direcciones desde las que se puede gestionar su modelo asociado, indicando que vista debe de utilizarse según el resultado de cada operación.

Clases o paquetes creados

La distribución se las distintas clases se dispone de la siguiente manera:

- Paquete de controladores: org.springframework.samples.petclinic.web
  - Ejemplo: UserController, WorkoutController, ActivityController, etc.
- Paquete de modelos: org.springframework.samples.petclinic.model
  - Ejemplo: User, Workout, Activity, etc.
- Directorio de vistas: src/main/webapp/WEB-INF/jsp
  - Ejemplo: vistas del modelo User: users/listadoUsuarios.jsp, users/detalleUsuario.jsp, etc.

Ventajas alcanzadas al aplicar el patrón

Se trata de un patrón que favorece una cohesión alta (cada vista puede centrarse en cumplir un único objetivo) y un bajo acoplamiento (cada vista puede ser completamente independiente del resto) lo que permite dar soporte a múltiples vistas con un mismo controlador y modelo.

Patrón: Front Controller

Tipo: De diseño

Contexto de Aplicación

Patrón implementado por el uso de la librería Spring.

Clases o paquetes creados

Spring utiliza la clase DispatcherServlet para distribuir las distintas peticiones.

Ventajas alcanzadas al aplicar el patrón

El uso de este patrón permite dividir y agrupar los mappings de la aplicación en cada controlador, mejorando la cohesión y el acoplamiento.

Patrón: Dependency Injection

Tipo: De diseño

Contexto de Aplicación

Patrón implementado por el uso de la librería Spring.

Clases o paquetes creados

Todos los paquetes de la aplicación contenidos en “src/main/java” (org.springframework.samples.petclinic.*).

Ventajas alcanzadas al aplicar el patrón

Simplicidad a la hora de desarrollar y mantener la aplicación además de necesitar menos codificación para conseguir la misma funcionalidad.

Patrón: Proxy

Tipo: De diseño

Contexto de Aplicación

Patrón implementado por el uso de la librería Spring.

Clases o paquetes creados

Para el uso de este patrón no ha sido necesario desarrollar ni crear ningún paquete.

Ventajas alcanzadas al aplicar el patrón

El manejo de las entidades de dominio se simplifica ya que los resultados de las consultas son proxies e Hibernate gestiona automáticamente el guardado de esas entidades.

Patrón: Template View

Tipo: Arquitectónico

Contexto de Aplicación

Patrón soportado por la librería Spring e implementado en la aplicación desde la primera versión de Petclinic.

Clases o paquetes creados

El contenido de las distintas partes de la aplicación se encuentran en el directorio *src/main/webapp/WEB-INF/jsp/**/** **y** *src/main/webapp/WEB-INF/tags /**/*.*

Ventajas alcanzadas al aplicar el patrón

Simplicidad a la hora de crear las distintas vistas además de poder componer cada una con partes reutilizables.

Patrón: Transaction Script

Tipo: Arquitectónico

Contexto de Aplicación

Se define toda la lógica de la aplicación en procedimientos vinculados a cada petición desde los controladores hasta los repositorios pasando por las distintas capas.

Clases o paquetes creados

Paquetes afectados:

- org.springframework.samples.petclinic.repository
- org.springframework.samples.petclinic.service
- org.springframework.samples.petclinic.web

Un controller define una acción asociada a un url path haciendo uso del servicio y, en consecuencia, de uno o varios repositorios involucrados en la acción de dicho servicio.

Ventajas alcanzadas al aplicar el patrón

Agrupa y separa responsabilidades. Favorece la reutilización de código.

Patrón: Domain Model

Tipo: De diseño

Contexto de Aplicación

Patrón implementado por el uso de la librería Javax Persistence (incluído en Spring).

Clases o paquetes creados

Todas las clases de dominio incluídas en el paquete *org.springframework.samples.petclinic.model.*

Ventajas alcanzadas al aplicar el patrón

Todas las tablas de la base de datos quedan representadas en clases Java. No requiere scripts de creación de tablas ni de restricciones. Fácil compresión y uso incluso para modificar propiedades de cada tabla.

Patrón: Service Layer

Tipo: De diseño

Contexto de Aplicación

Capa aplicada sobre el modelo de dominio para gestionar las distintas operaciones que se pueden llevar a cabo para gestionarlos.

Clases o paquetes creados

Todos los servicios definidos en el paquete *org.springframework.samples.petclinic.service.*

Ventajas alcanzadas al aplicar el patrón

Las distintas entidades quedan agrupadas y relacionadas entre sí. Facilita la gestión de relaciones complejas.

Patrón: (Meta) Data Mapper

Tipo: De diseño

Contexto de Aplicación

Patrón implementado por el uso de la librería Spring a través de la clase EntityManager.

Clases o paquetes creados

Para el uso de este patrón no ha sido necesario desarrollar ni crear ningún paquete.

Ventajas alcanzadas al aplicar el patrón

La clase EntityManager simplifica las operaciones genéricas que se realizan contra la base de datos.

Patrón: Identity Field

Tipo: Arquitectónico

Contexto de Aplicación

Patrón implementado por el uso de la librería Javax Persistence (incluído en Spring).

Clases o paquetes creados

Se definen identificadores numéricos autogenerados para los modelos que hereden de BaseEntity.

Ventajas alcanzadas al aplicar el patrón

Evita la generación de SQL incluyendo las restricciones y secuencias de autoincremento automáticamente.

Patrón: Layer Supertype

Tipo: De diseño

Contexto de Aplicación

Existe un grupo de entidades cuyos atributos son id y name. Dado que la mayoría de las entidades necesitan un campo de identificación y algunas (entidades correspondientes con tablas maestras) un atributo nombre se opta por definir una entidad base (BaseEntity) que se identifique por un campo ID de tipo entero y otra para las entidades con atributos id y nombre, NamedEntity de las que heredarán el resto de modelos de dominio.

Clases o paquetes creados

Se implementa clase BaseEntity para identificacion por ID y NamedEntity (que también extiende de BaseEntity) que incluye un atributo “name” por defecto.

Ventajas alcanzadas al aplicar el patrón

Disminuye los desarrollos al no tener que mapear los mismos atributos comunes en todas las clases.

Patrón: Repository Pattern

Tipo: De diseño

Contexto de Aplicación

Para cubrir la necesidad de ofrecer un CRUD básico para cada uno de los modelos de dominio y poder extenderlo según la necesidad se plantea una capa repository que defina el conjunto de operaciones disponibles para cada entidad.

Clases o paquetes creados

Todas las clases incluídas en el paquete *org.springframework.samples.petclinic.repository.*

Ventajas alcanzadas al aplicar el patrón

Se agrupan las funcionalidades de cada entidad dividiendo responsabilidades para cada tipo (un repositorio para la entidad Training no se hace responsable de la gestión de la entidad Exercise contenida en uno de los atributos de Training).

Patrón: Eager/Lazy loading

Tipo: De diseño

Contexto de Aplicación

A la hora de cargar todos los atributos de una entidad estos pueden representar a otras entidades y en consecuencia necesitar subconsultas para cargarlos. Se hace uso del patrón Eager/Lazy loading ofrecido en la librería javax.persistence para definir una estrategia de carga para esos atributos a la hora de solicitar a un repositorio información sobre una entidad.

Clases o paquetes creados

Este patrón afecta a todos los modelos de dominio (org.springframework.samples.petclinic.model) que estén relacionados con otra entidad y necesiten representarse mediante un listado.

Ventajas alcanzadas al aplicar el patrón

Reduce una enorme cantidad de desarrollo al automatizar el cuándo y cómo quieres que se carguen los listados.

Decisiones de diseño

En esta sección describiremos las decisiones de diseño que se han tomado a lo largo del desarrollo de la aplicación que vayan más allá de la mera aplicación de patrones de diseño o arquitectónicos.

Decisión 1

Descripción del problema:

La versión inicial de petclinic plantea el sistema de autenticación a través de una clase User donde un usuario se identifica por un username y lleva asociado un listado de permisos llamado authorities. Dado que el modelo conceptual de la aplicación no requiere de permisos para identificar el tipo de usuario que se está autenticando se plantean distintas alternativas.

Alternativas de solución evaluadas:

Alternativa 1.a: Utilizar la entidad “Authorities” ya incluída en la aplicación para determinar los distintos roles/permisos que puede tener un usuario en una sesión.

Ventajas:

- Al ser un listado de permisos permite diversificar las distintas acciones que se pueden realizar en la aplicación.

Inconvenientes:

- Requiere un mapeo más exhaustivo.
- La gestión de los permisos a la hora de crear/editar un usuario es más compleja.
- Hay que agrupar los permisos por rol/tipo de usuario.

Alternativa 1.b: Utilizar una entidad “TipoUsuario” que determine el tipo de usuario que se está autenticando contra el sistema y le asigne un único rol (el propio tipo).

Ventajas:

- Simplifica la gestión de los permisos de un usuario.
- Al ser un único permiso

Inconvenientes:

- Las acciones que se pueden realizar no pueden estar muy diversificadas.

Justificación de la solución adoptada

Según el modelo de la aplicación los usuarios están relacionados con un tipo de usuario. Por simplicidad en diseño y desarrollo y porque los requisitos de la aplicación no requieren un control exhaustivo de los permisos de cada acción se decide optar por la alternativa 1.b.

Decisión 2

Descripción del problema:

Un usuario tiene asignado una rutina, y esta consta de un entrenamiento por cada día de la semana. El entrenamiento está formado por una lista de ejercicios.
Se tiene la necesidad de personalizar los entrenamientos y ejercicios para cada usuario.

Alternativas de solución evaluadas:

Alternativa 2.a: Los monitores crearán entrenamientos y ejercicios que serán utilizados como plantillas para después ser asignados a los usuarios. Diremos que estos entrenamientos y ejercicios son genéricos. Al asignar una rutina a un usuario asignaremos una copia del entrenamiento genérico (junto a sus ejercicios, también copiados) para que puedan ser personalizados según las necesidades del usuario.

Ventajas:

- Al crear la rutina del usuario, partimos de un entrenamiento genérico con ejercicios asignados que podemos personalizar para el caso concreto de cada usuario.
- Este grado de personalización permite separar los entrenamientos genéricos de los personalizados.

Inconvenientes:

- La posibilidad de tener registros de rutinas con entrenamientos y ejercicios idénticos (registros duplicados).
- El número de registros en las tablas.
- Baja cohesión de la información: no se podrá editar de forma común un entrenamiento o ejercicio.

Alternativa 2.b: Crear rutina desde cero. Al asignar entrenamientos a la rutina, tendríamos que asignar un entrenamiento vacío y añadir los ejercicios al mismo, sin tener la posibilidad de seleccionar un entrenamiento prediseñado.

Ventajas:

- No tener la necesidad de crear entrenamientos y ejercicios genéricos.

Inconvenientes:

- Tener que crear rutinas y ejercicios partiendo de cero, sin tener la posibilidad de usar uno genérico que nos ahorre el tener que seleccionar los días de entrenamientos y sus respectivos ejercicios uno a uno.

Justificación de la solución adoptada:

Creemos que para la usabilidad de la aplicación y para facilitar el trabajo de los monitores la alternativa 2.a es la más conveniente para nuestra aplicación. De esta forma, un monitor puede asignar las rutinas de un usuario según sus necesidades de forma más rápida y eficaz.

Decisión 3

Descripción del problema:

La versión de Bootstrap que incorpora la plantilla del proyecto está anticuada y su versión actual incluye features que pueden facilitar el desarrollo del proyecto.

Alternativas de solución evaluadas:

Alternativa 1.a: Utilizar MDBootstrap (una implementación del patrón Material Design con Bootstrap.

Ventajas:

- Diseño más innovador y usable.
- Se unifican los componentes UI con los de Bootstrap en una misma librería por lo que se puede prescindir de Bootstrap-UI.
- Se aprovechan todas las features de la versión más reciente de Bootstrap.

Inconvenientes:

- Hay que adaptar los layouts y componentes que vienen con la aplicación a la nueva tecnología.

Alternativa 1.b: Actualizar la versión de Bootstrap y Bootstrap-UI a su versión compatible más reciente..

Ventajas:

- Se aprovechan todas las features de la versión más reciente de Bootstrap.

Inconvenientes:

- Revisar los breaking changes y adaptar las vistas a las nuevas versiones.

Alternativa 1.c: Seguir usando las versiones de Bootstrap y Bootstrap-UI incluídas en la plantilla del proyecto.

Ventajas:

- No hay que crear un nuevo diseño.
- Se pueden reutilizar todas las plantillas incluidas en los ejemplos.

Inconvenientes:

- Debido a que la versión está anticuada, cuando nos surjan problemas y busquemos en foros, no encontraremos soluciones para dicha versión.

Justificación de la solución adoptada

Elegimos la alternativa 1.a para así aprovechar las features de la última versión y trabajar con un diseño más innovador que aumente la motivación del equipo.

Decisión 4

Descripción del problema:

Existe la necesidad de establecer una política de logging para organizar la trazabilidad de la aplicación.

Alternativas de solución evaluadas:

Alternativa 1.a: La política de logging quedaría definida por las siguientes características:

- Estrategia de nombrado: dentro del directorio logs, correspondería un directorio para cada día y uno o varios ficheros para ese día (`logs/<YYYY-MM-DD>/server.<index>.log`).
- Se establece un límite de 100MB por fichero.
- Se conservarán copias de hasta 1 mes (30 días) de antigüedad.

Ventajas:

- Un directorio por día nos permite ser concisos a la hora de buscar los logs de un único día
- Se conservan logs de tiempo suficiente en caso de error

Inconvenientes:

- Búsquedas dentro de los logs más lentas (por ser un fichero de 100MB de texto plano abierto en memoria)

Alternativa 1.b: La política de logging quedaría definida por las siguientes características:

- Estrategia de nombrado: dentro del directorio logs, correspondería un directorio para cada día acompañado de un índice que indique el número de log generado durante ese día: (`logs/<YYYY-MM-DD>-<index>.log`).
- Se establece un límite de 50MB por fichero.
- Se conservarán copias de hasta 10 días de antigüedad.

Ventajas:

- Búsquedas más rápidas al incrementar el rendimiento de búsqueda por reducir el tamaño de cada fichero.
- Menor necesidad de almacenamiento.

Inconvenientes:

- Mayor número de ficheros para cada día.
- Mayor probabilidad de perder información al conservarse ficheros de hasta 10 días.

Justificación de la solución adoptada

Elegimos la alternativa 1.a puesto que consideramos que cuanto más información se almacene mejor respuesta podremos dar a la hora de dar una solución a un problema.

Decisión 5

Descripción del problema:

El flujo de trabajo más adecuado para reducir los errores y facilitar el desarrollo en paralelo.

Alternativas de solución evaluadas:

Alternativa 1.a: Flujo de trabajo de rama de función. Se generan ramas directamente sobre la rama master y al hacer la subida de tu rama tienes que generar una solicitud de incorporación.

Ventajas:

- Flujo de trabajo más simple. Las ramas se crean directamente sobre master y los commit se realizan sobre ella.
- Antes de mergear la rama, un compañero debe revisar el código y dar el visto bueno.

Inconvenientes:

- Al crear directamente la rama sobre master, es más fácil que se pasen errores.
- El flujo de trabajo es más lento, ya que para añadir una funcionalidad, un compañero tiene que validarla.

Alternativa 1.b: Flujo de trabajo de gitflow. De la rama master sacamos una rama llamada develop, sobre la que se crearán las ramas para llevar a cabo las tareas.Estas ramas se sitúan en la carpeta "feature". Cuando se acerca la fecha de una demo, se genera una rama en la carpeta "release" desde la rama develop y añadimos todas las ramas de las tareas terminadas. Los errores generados detectados se corrigen directamente sobre la rama creada en la carpeta "release". Si todo funciona correctamente, añadimos la rama creada en release en master y en develop.

Ventajas:

- La rama master no sufre cambios hasta que se han validado tanto en la rama de la tarea como en la demo el funcionamiento de ella.
- En el caso de encontrar un error en master, se genera una rama de hotfix, para solucionar el problema. Esta rama se mergea tanto en master como en develop.

Inconvenientes:

- Es un flujo más complejo.

Justificación de la solución adoptada

Elegimos la alternativa 1.b ya que los componentes del grupo conocemos este flujo de trabajo y por adaptarse mejor a nuestra necesidades organizativas.

Decisión 6

Descripción del problema:

En la gestión de los productos, al eliminar un producto de la aplicación nos encontramos con el problema de dejar inconsistente las compras y ventas.

Alternativas de solución evaluadas:

Alternativa 6.a: Al eliminar el producto, hacemos un borrado lógico. Le añadimos el campo "inactive" y gestionamos el estado del producto.

Ventajas:

- Las compras y ventas mantienen sus datos consistentes.

Inconvenientes:

- La cantidad de datos que pueden acumularse en la tabla con el tiempo.

Alternativa 6.b: Al eliminar el producto, eliminar el registro del producto y actualizar las compras o ventas en las que ese producto aparezca.

Ventajas:

- No dejamos datos innecesarios en la tabla de productos

Inconvenientes:

- Se perderían datos necesarios para la gestión de compra y venta.
- La eliminación de un producto podría llegar a ser una operación costosa.

Justificación de la solución adoptada

Elegimos la alternativa 6.a ya que consideramos que para la gestión de las compras y ventas es necesario mantener todos los datos de los productos, aunque estos ya no estén activos.

Decisión 7

Descripción del problema:

Para poder gestionar la auditoría de la aplicación. ya que varios usuarios pueden tener acceso a la gestión de tablas sensibles y así tener un registro de quién ha ido realizando los cambios.

Alternativas de solución evaluadas:

Alternativa 7.a: implementar la auditoría con spring

Ventajas:

- Delegar en una aplicación que está muy probada y que ofrece una gran seguridad.
- Más simple y rápido de implementar.

Inconvenientes:

- Investigar como implementarlo.

Alternativa 7.b: Controlar la auditoría en cada servicio.

Ventajas:

- Control sobre los campos que se auditan.

Inconvenientes:

- Código menos legible.
- Complejidad.
- Rendimiento.

Justificación de la solución adoptada

Hemos optado por la opción 7.a, ya que delegamos la auditoria en una herramienta muy probada y fácil de usar.

Decisión 8

Descripción del problema:

Gestionar y mantener los logs de la aplicación para poder buscar futuros errores y tener un registro de ellos.

Alternativas de solución evaluadas:

Alternativa 8.a: Meter logs para todos los métodos.

Ventajas:

- Rastro de todo lo que sucede en la aplicación

Inconvenientes:

- Rendimiento.
- Muchos datos innecesarios.

Alternativa 8.b: No meter ningún log.

Ventajas:

- Rendimiento

Inconvenientes:

- Dificultad para encontrar errores.

Alternativa 8.c: Meter logs en los controladores críticos

Ventajas:

- Rendimiento
- Control de errores y operaciones críticas.

Inconvenientes:

- Quizás, información insuficiente.

Justificación de la solución adoptada

Hemos optado por la opción 8.c ya que la consideramos la más equilibrada para mantener un buen rendimiento y teniendo los datos suficientes para controlar los posibles errores o operaciones críticas.

Decisión 9

Descripción del problema:

En las siguientes relaciones nxm, nos hemos encontrado con la necesidad de añadir un atributo a la tabla intermedia de la relación. Hibernate no ofrece ninguna configuración que nos permita mapear esos campos sin implementar clases nuevas. Las relaciones afectadas son:

-Entrenamiento con rutina

-Producto con compra

-Producto con venta

Alternativas de solución evaluadas:

Alternativa 9.a: Crear tablas intermedias con los atributos necesarios

Ventajas:

- Fácil mapeo.
- gestión de operaciones.

Inconvenientes:

-

Alternativa 9.b: Meter la propiedad en una de las tablas.

Ventajas:

- Menos tablas que gestionar

Inconvenientes:

- Tabla con propiedades que no corresponden a la entidad.
- Inserción de valores nulos al crear entidades que no estén contempladas en la relación

Justificación de la solución adoptada

Hemos optado por la opción 9.a por considerarla más limpia y conceptualmente más correcta.