DP1 2020-2021

Documento de Diseño del Sistema

Proyecto Alexandria

Repositorio: https://github.com/gii-is-DP1/dp1-2020-g2-10

Miembros <en orden alfabético por apellidos>:

- Cabello Colmenares, Carlos Manuel.
- Calle Pérez, Pablo.
- Conde Marrón, Félix
- Dorado Abadías, Oscar.
- Gañán García, David.
- Martín Avecilla, Mariano.

Tutor: Müller Cejas, Carlos Guillermo.

GRUPO G2-10

Versión 1.00

16 de Diciembre de 2020

Historial de versiones

Fecha	Versión	Descripción de los cambios	Sprint
16/12/2020	1.00	● Versión inicial del documento.	3

Contents

Introducción

Alexandria es una plataforma online de lectura y escritura. En ella los creadores pueden publicar sus novelas, relatos, historias, poemas y muchos otros géneros que los demás usuarios pueden leer de forma gratuita.
El principal objetivo que pretende nuestra plataforma es que las promesas de la escritura se den a conocer y que gracias a ello puedan ser contactados por diversas empresas además de promover la lectura.
Las funcionalidades más destacadas e interesantes son las contribuciones, la labor de los moderadores de aprobar o banear historias reportadas y la contratación de autores por parte de las empresas.

Diagrama(s) UML:

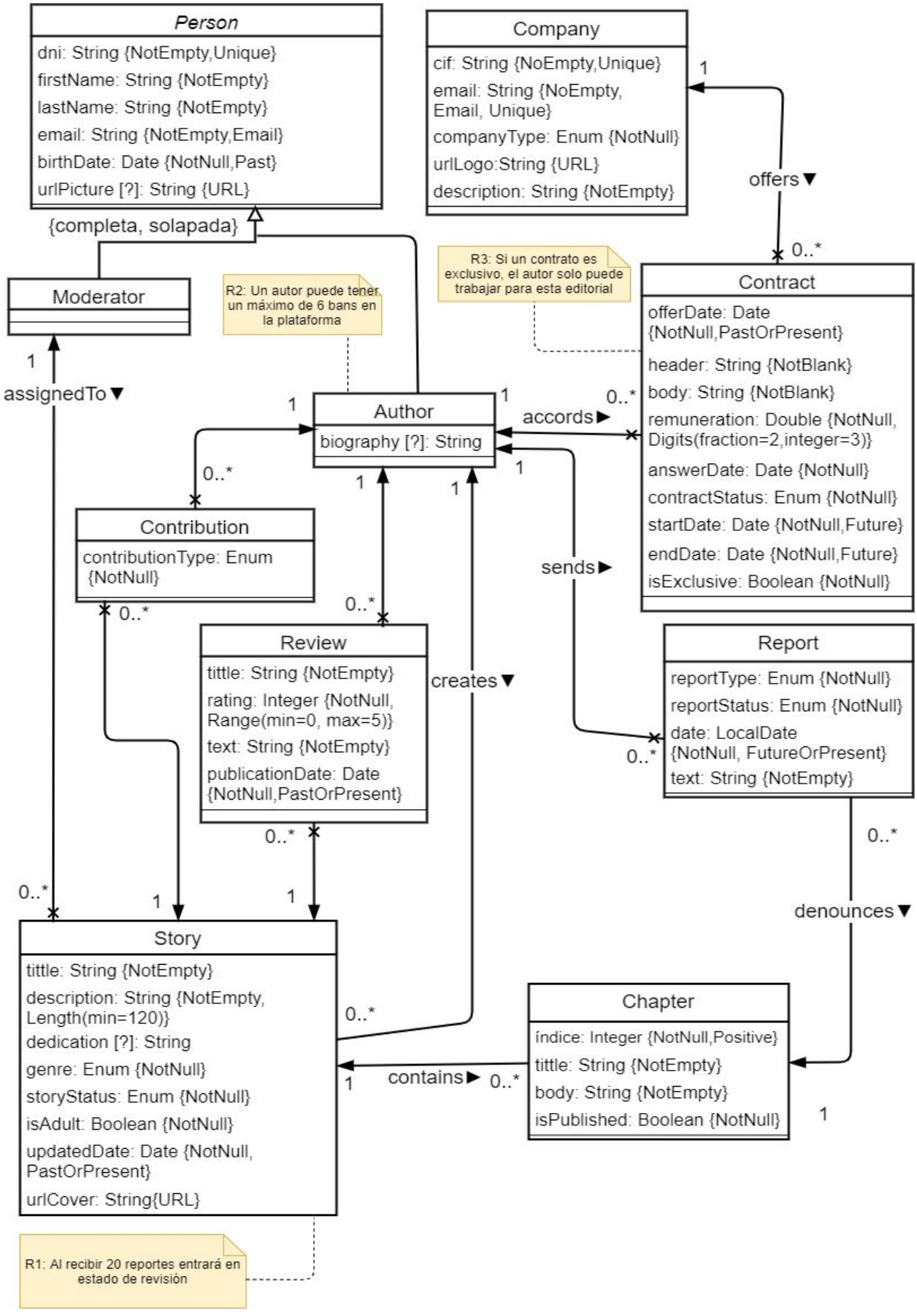
Diagrama de Dominio/Diseño

- Diagrama de dominio:

  NOTA 1: No se han añadido las clases específicas de la tecnología usada como BaseEntity y NamedEntity, puesto que rompían las distribución del diagrama, no permitiendo una lectura correcta. Las entidades que extienden BaseEntity son: Contribution, Review, Story, Contract, Report y Chapter. Y las que extienden a NamedEntity son: Company.

  NOTA 2: Las restricciones respecto a patrones del tipo: `@DateTimeFormat(pattern = "yyyy/MM/dd HH:mm")` y a los nombres de las propiedades del tipo: `@Column(name = "start_date")`, han sido obviadas por la misma razón de antes.

  NOTA 3: La notación [?] indica que el atributo es opcional.


Person
dni: String {NotEmpty,Unique}
firstName: String {NotEmpty}
lastName: String {NotEmpty}
email: String {NotEmpty,Email}
birthDate: Date {NotNull,Past}
urlPicture [?]: String {URL}
Company
cif: String {NoEmpty,Unique}
email: String {NoEmpty, Email, Unique}
companyType: Enum {NotNull}
urlLogo:String {URL}
description: String {NotEmpty}
offers
{completa, solapada}
Moderator
R2: Un autor puede tener un máximo de 6 bans en la plataforma
R3: Si un contrato es exclusivo, el autor solo puede trabajar para esta editorial
Contract
offerDate: Date {NotNull,PastOrPresent}
header: String {NotBlank}
body: String {NotBlank}
remuneration: Double {NotNull, Digits(fraction=2,integer=3)}
answerDate: Date {NotNull}
contractStatus: Enum {NotNull}
startDate: Date {NotNull,Future}
endDate: Date {NotNull,Future}
isExclusive: Boolean {NotNull}
assignedTo
Author
biography [?]: String
accords
Contribution
contributionType: Enum {NotNull}
sends
Review
tittle: String {NotEmpty}
rating: Integer {NotNull, Range(min=0, max=5)}
text: String {NotEmpty}
publicationDate: Date {NotNull,PastOrPresent}
creates
Report
reportType: Enum {NotNull}
reportStatus: Enum {NotNull}
date: LocalDate {NotNull, FutureOrPresent}
text: String {NotEmpty}
denounces
Story
tittle: String {NotEmpty}
description: String {NotEmpty, Length(min=120)}
dedication [?]: String
genre: Enum {NotNull}
storyStatus: Enum {NotNull}
isAdult: Boolean {NotNull}
updatedDate: Date {NotNull, PastOrPresent}
urlCover: String{URL}
Chapter
índice: Integer {NotNull,Positive}
tittle: String {NotEmpty}
body: String {NotEmpty}
isPublished: Boolean {NotNull}
contains
R1: Al recibir 20 reportes entrará en estado de revisión

- Enumerados:

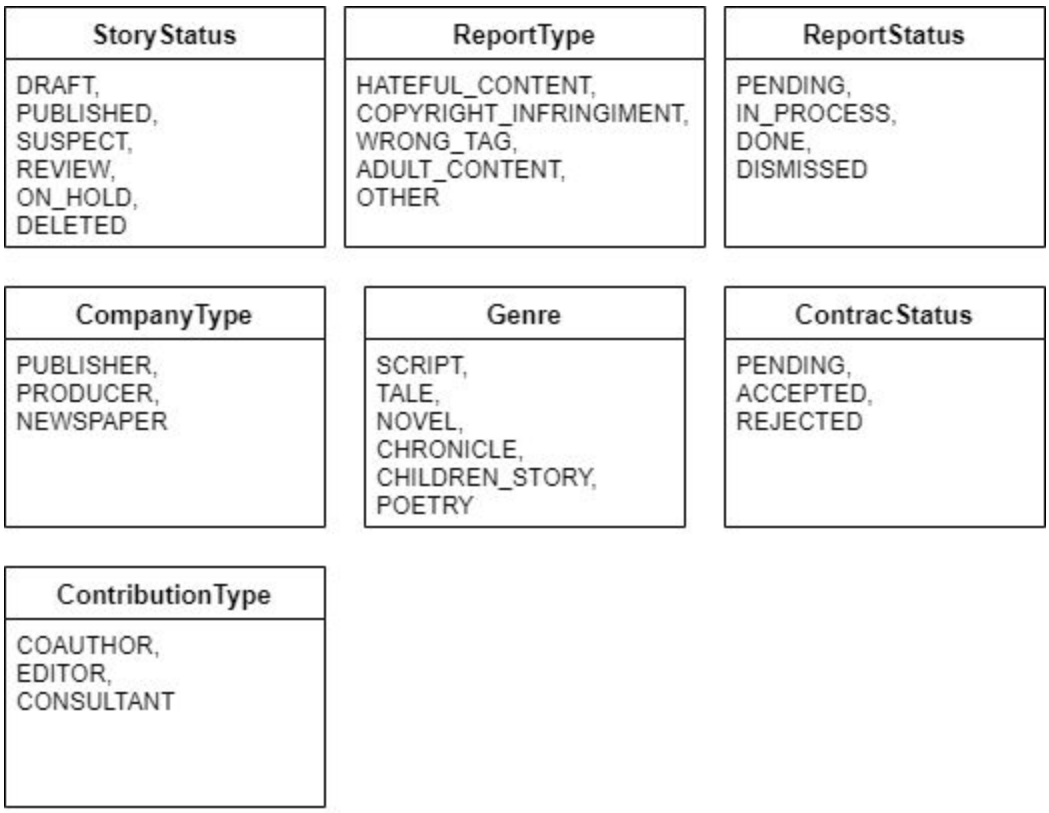
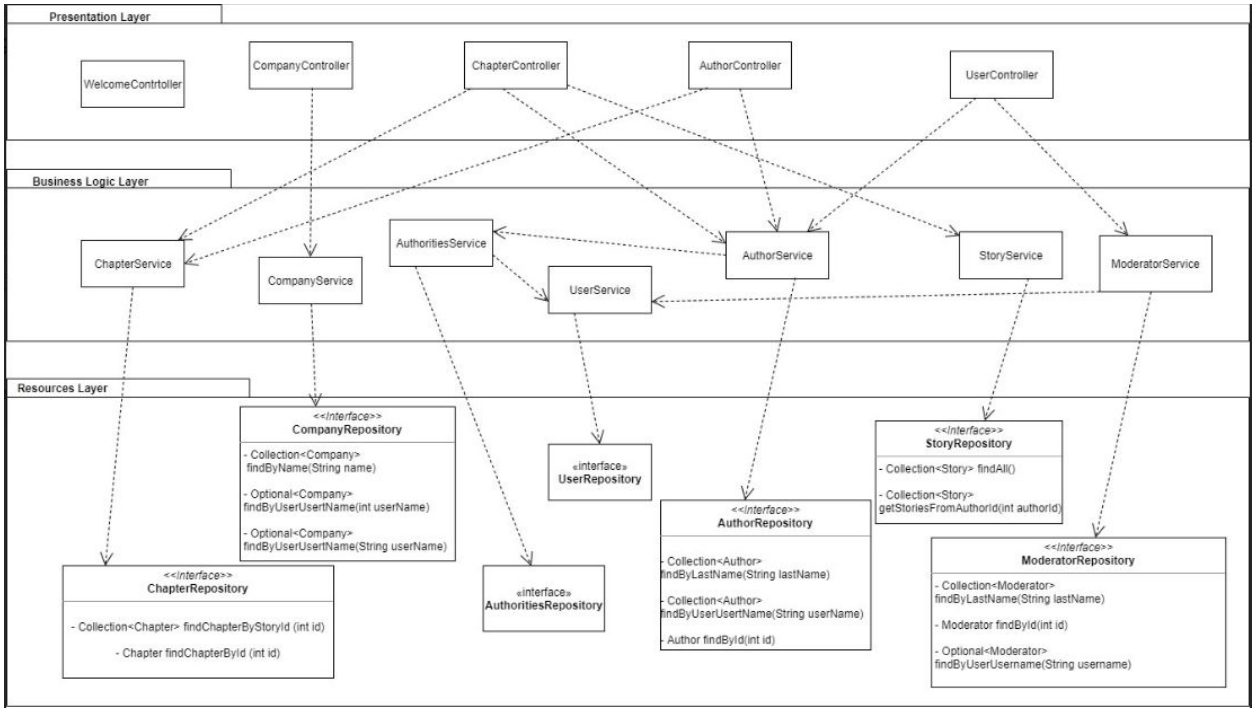
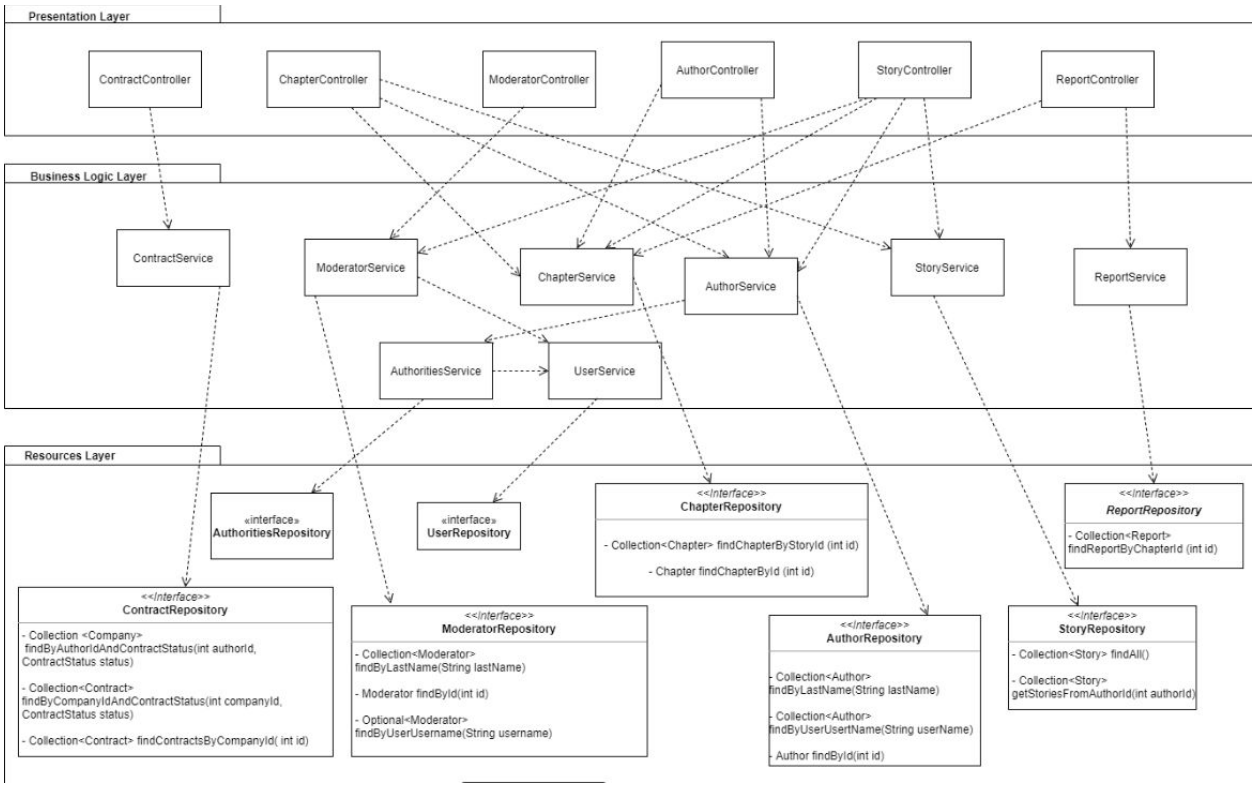

StoryStatus
DRAFT,
PUBLISHED,
SUSPECT,
REVIEW,
ON_HOLD,
DELETED
ReportType
HATEFUL_CONTENT,
COPYRIGHT_INFRINGIMENT,
WRONG_TAG,
ADULT_CONTENT,
OTHER
ReportStatus
PENDING,
IN_PROCESS,
DONE,
DISMISSED
CompanyType
PUBLISHER,
PRODUCER,
NEWSPAPER
Genre
SCRIPT,
TALE,
NOVEL,
CHRONICLE,
CHILDREN_STORY,
POETRY
ContracStatus
PENDING,
ACCEPTED,
REJECTED
ContributionType
COAUTHOR,
EDITOR,
CONSULTANT

Diagrama de Capas (incluyendo Controladores, Servicios y Repositorios)





Patrones de diseño y arquitectónicos aplicados

En esta sección se especifica 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: < Front controller pattern >

Tipo: Diseño

Contexto de Aplicación

Este patrón se ha aplicado mediante el Dispatcher servlet de Spring Boot. Este patrón se encarga de gestionar las llamadas a la aplicación, invocando a los componentes necesarios. Es el único punto por el que pasa todo.

Clases o paquetes creados

La clase PetclinicInitializer.java.

Ventajas alcanzadas al aplicar el patrón

El uso de este patrón venía exigido un poco por la asignatura. Sin embargo, el hecho de que permita gestionar de una forma bastante simple las llamadas a la app me parece un motivo de bastante peso para usarlo.

Patrón: < Dependency Injection >

Tipo: Diseño

Contexto de Aplicación

El patrón se aplica en los controladores, servicios y los tests de estos mismos.

Clases o paquetes creados

Indicar las clases o paquetes creados como resultado de la aplicación del patrón.

Ventajas alcanzadas al aplicar el patrón

Reduce el acoplamiento de los componentes, facilita el testing al no tener que usar instancias reales y poder usar test doubles y deriva la responsabilidad de manejar los componentes al contenedor.

Patrón: < Template view >

Tipo: Diseño

Contexto de Aplicación

En concreto se ha aplicado “JSP/JSTL”. Sirve para generar las vistas de la aplicación . En estas vistas se pueden incluir anotaciones que el framework interpreta para generar el código html que le llega al cliente. También permite el uso de etiquetas, bien de la librería JSP, la de Spring o creadas por nosotros mismos, que sirven para reutilizar código que hayamos creado o que nos facilitan el desarrollo al darnos parte de la implementación ya hecha, un ejemplo de esto es la tag “<form:input>”.

Clases o paquetes creados

El paquete WEB-INF.

Ventajas alcanzadas al aplicar el patrón

El patrón “Template view” permite organizar y desarrollar de una manera más intuitiva las vistas de una aplicación web, frente al patrón “Transform view”. Además, otra gran ventaja que tiene es que todos los miembros del grupo conocían jsp y html de antes, por lo que podíamos ponernos a trabajar de inmediato, sin tener que formarnos previamente.

Patrón: < Domain model >

Tipo: Diseño

Contexto de Aplicación

La lógica de dominio de la aplicación se ha organizado en base a este patrón. Spring viene preparado para usar el patrón “Domain model", solo hay que crear las entidades y Spring Data se encarga de, a partir de las entidades, crear la base de datos. En concreto estamos usando el patrón (Meta) Data mapper.

Clases o paquetes creados

Indicar las clases o paquetes creados como resultado de la aplicación del patrón.

Ventajas alcanzadas al aplicar el patrón

Este patrón permite implementar una lógica de negocio más compleja que la de los otros dos que hemos dado en la asignatura, como son “Table module” o “Transaction script”. Otra ventaja es que como el framework está preparado para usar este patrón pues la implementación es muy simple, nosotros solo nos tenemos que preocuparnos de crear los objetos y, como hemos dicho antes, Spring se encarga de generar y gestionar la base de datos.

Patrón: < Service layer >

Tipo: Diseño

Contexto de Aplicación

Este patrón forma parte de la lógica de negocio de la aplicación.
Describir las partes de la aplicación donde se ha aplicado el patrón. Si se considera oportuno especificar el paquete donde se han incluido los elementos asociados a la aplicación del patrón.

Clases o paquetes creados

El paquete org.springframework.samples.petclinic.service.

Ventajas alcanzadas al aplicar el patrón

Este patrón, provee a la aplicación de una mayor separación de responsabilidades y disminuye el acoplamiento. También le añade un poco más de seguridad al no ser el controlador el que hace las llamadas a la base de datos.

Patrón: < Repository pattern >

Tipo: Diseño

Contexto de Aplicación

Este patrón encapsula la lógica requerida para acceder a los datos. El objetivo es que de la impresión de que se está accediendo a los datos como si de una colección de objetos en memoria se tratase.

Clases o paquetes creados

El paquete org.springframework.samples.petclinic.repository.

Ventajas alcanzadas al aplicar el patrón

Favorece el bajo acoplamiento entre la parte de datos y la lógica de negocio. Esto facilita migraciones de la capa de persistencia a otras tecnologías. Además, gracias a esto mismo, hace que trabajar con los datos sea mucho más fácil.

Patrón: < Model View Controller >

Tipo: Arquitectónico

Contexto de Aplicación

La estructura de la aplicación se basa en este patrón. Esto es así por el uso de POJOs para el modelo, de controladores y de páginas JSP para las vistas.

Clases o paquetes creados

Los paquetes org.springframework.samples.petclinic.model, org.springframework.samples.petclinic.web y WEB-INF (en este tenemos las páginas JSP y las tags, que las considero parte de las vistas).

Ventajas alcanzadas al aplicar el patrón

La posibilidad de que varios desarrolladores puedan trabajar simultáneamente en diferentes partes de la aplicación. Soporte para múltiples vistas. Favorece la cohesión, el bajo acoplamiento y la separación de responsabilidades.

Patrón: < (Meta) Data mapper >

Tipo: Diseño

Contexto de Aplicación

Este patrón se ha aplicado por imposición de la asignatura. En JPA se llama Entity Manager. Normalmente, no usamos Entity Manager directamente, sino que usaremos repositorios en su lugar.

Clases o paquetes creados

Se considera, tal y como se ha indicado antes, que el repositorio es la forma en la que este patrón de diseño está aplicado en la aplicación. Por esto coincide el paquete de repositorios como las clases o paquetes creados.

Ventajas alcanzadas al aplicar el patrón

Permite mover datos entre objetos y una base de datos mientras los mantiene independientes entre sí y del propio mapeador.

Decisiones de diseño

Decisión 01: Direccionalidad de las relaciones entre entidades.

Descripción del problema:

Al modelar las relaciones entre las distintas entidades, teníamos que indicar de qué tipo serían.

Alternativas de solución evaluadas:

Alternativa 1: Usar relaciones unidireccionales.

Ventajas:

- Gestión sencilla; no requiere implementación de cascadas y se elude la problemática de borrar las dos entidades relacionadas si suprime una de ellas. Por ejemplo, si la entidad 1 y la entidad 2 mantienen una relación unidireccional, si se elimina la entidad 1, no se hace con la entidad 2.

.

Inconvenientes:

- Mayor complicación en ciertas circunstancias. Verbigracia, si la relación entre dos entidades es unidireccional y 1:n, la clave ajena la contiene la entidad n; por lo que, si queremos comprobar desde la entidad si está asociada con otra n, no se puede comprobar directamente; no obstante, sí se puede probar al revés, es decir, desde la entidad n, puesto que es la que contiene la clave ajena de la entidad 1 asociada.

Alternativa 2: Emplear relaciones bidireccionales.

Ventajas:

- Mayor facilidad al momento de efectuar ciertas comprobaciones. Verbigracia, si dos entidades mantienen una relación bidireccional, siendo una asociación 1:n, se puede verificar la relación desde la entidad 1, porque contiene las claves ajenas del conjunto de instancias de la entidad n. En cambio, si fuera unidireccional, no se podría comprobar como anteriormente.

Inconvenientes:

- Implementación y gestión complicada: las relaciones deben implementar cascada, cerciorándose de que, si se elimina una entidad de la relación en cuestión, también se ha de suprimir la asociada.

Justificación de la solución adoptada

Elegimos las relaciones unidireccionales por costumbre y por eludir los riesgos que conllevan la implementación de relaciones bidireccionales.

Decisión 02: Restricciones simples.

Descripción del problema:

Ciertas validaciones no requieren complejidad.

Alternativas de solución evaluadas:

Caben dos posibilidades: uso de validaciones sintácticas o semánticas.

Alternativa 1: Usar validaciones sintácticas, concretamente, las anotaciones “@” que proporciona Spring por defecto “Spring built-int Validator”.

Ventajas:

- Soporta validaciones sintácticas.
- Se desarrollan fácilmente con Spring.
- BindingResult provee de numerosos métodos útiles.

Inconvenientes:

- No soporta validación semántica.

Alternativa 2: Emplear validadores personalizados.

Ventajas:

- Soporta validaciones sintácticas y semánticas.
- Se desarrollan fácilmente con Spring.

- BindingResult provee de numerosos métodos útiles.
- Proporciona alta cohesión cuando una regla de negocio para una entidad es validada por un validador personalizado.

Inconvenientes:

- Un controlador solo puede tener un validador personalizado asociado, por lo que complica las relaciones n:n.
- Las reglas de negocios se validan en los controladores, pero no en las clases que implementan la lógica de negocio.
- Las validaciones sintácticas incluidas en la entidad deben invocarse manualmente desde el validador personalizado.

Alternativa 3: Emplear excepciones.

Ventajas:

- Soporta validaciones sintácticas y semánticas.
- Añade mucha expresividad a los mensajes de error.
- Enfoque de manejo de errores habitual y fácil en idiomas OO

Inconvenientes:

- Poca cohesión.
- Solo se puede lanzar una excepción al mismo tiempo.
- Aumenta la complejidad del código fuente.
- Puede haber otras partes del código que no se comprueben; inconsistencias en la base de datos.

Alternativa 4: Emplear anotaciones de validación personalizadas.

Ventajas:

- Soporta validaciones sintácticas y semánticas.
- Se desarrollan fácilmente con Spring.
- Alta cohesión.
- Se pueden desarrollar para reusar código.
- Soporta checkeo de reglas de negocio y capas de lógica de negocio.

Inconvenientes:

- Es complicado de reutilizar.
- Requiere una interfaz para cada una.
- Mayor dificultad para disponer de mensajes de error concretos.

Justificación de la solución adoptada

Elegimos para validaciones sencillas la alternativa 1, el uso de anotaciones de Spring, ya que son fáciles de usar y suficientes para validar este tipo de restricciones.

Decisión 03: Uso de herencias.

Descripción del problema:

Tenemos que decidir su uso herencias, aprovechando el uso de las clases padres o, por el contrario, no lo hacemos.

Alternativas de solución evaluadas:

Alternativa 1: Uso de superclases y herencias.

Ventajas:

- No se duplica el código. Por ejemplo, aquellas entidades que contengan un atributo id y un nombre, pueden extender de “NameEntity”, en lugar de establecer para cada una los susodichos.

Inconvenientes:

- Relaciones existentes que deben respetarse; ha de considerarse al realizar los modelos y implementarlas.

Alternativa 2: No emplear herencias.

Ventajas:

- Menor acoplamiento: las entidades no dependen de una clase padre.

Inconvenientes:

- Código duplicado. Verbigracia, si las entidades que se han de implementar no extienden de “BaseEntity”, para cada una de ellas habrá que establecer un atributo id que aparecerá en todas, puesto que requieren de un identificador.

Justificación de la solución adoptada

Utilizamos herencia, así no duplicamos código y aprovechamos el existente.

Decisión 4: Importación de datos reales para demostración

Descripción del problema:

Necesitamos disponer de un conjunto mínimo de instancias de cada entidad para poder probar las distintas funcionalidades implementadas.

Alternativas de solución evaluadas:

Alternativa 1: Incluir manualmente las respectivas instancias en el fichero “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 2: Crear un script adicional que contenga las instancias sumadas 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 3: 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 los 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

Elegimos la alternativa 1, ya que, nos parece más sencilla e intuitiva que las demás y estamos acostumbrados a proceder de esta manera.

Decisión 5: Validar reglas de negocio.

Descripción del problema:

Necesitamos validar que se cumplen las distintas reglas de negocio.

Alternativas de solución evaluadas:

Disponemos de tres posibilidades:

Alternativa 1: Emplear validadores personalizados.

Ventajas:

- Soporta validaciones sintácticas y semánticas.
- Se desarrollan fácilmente con Spring.
- BindingResult provee de numerosos métodos útiles.
- Proporciona alta cohesión cuando una regla de negocio para una entidad es validada por un validador personalizado.

Inconvenientes:

- Un controlador solo puede tener un validador personalizado asociado, por lo que complica las relaciones n:n.
- Las reglas de negocios se validan en los controladores, pero no en las clases que implementan la lógica de negocio.
- Las validaciones sintácticas incluidas en la entidad deben invocarse manualmente desde el validador personalizado.

Alternativa 2: Emplear excepciones.

Ventajas:

- Soporta validaciones sintácticas y semánticas.
- Añade mucha expresividad a los mensajes de error.
- Enfoque de manejo de errores habitual y fácil en idiomas OO

Inconvenientes:

- Poca cohesión.
- Solo se puede lanzar una excepción al mismo tiempo.
- Aumenta la complejidad del código fuente.
- Puede haber otras partes del código que no se comprueben; inconsistencias en la base de datos.

Alternativa 3: Emplear anotaciones de validación personalizadas.

Ventajas:

- Soporta validaciones sintácticas y semánticas.

- Se desarrollan fácilmente con Spring.
- Alta cohesión.
- Se pueden desarrollar para reusar código.
- Soporta checkeo de reglas de negocio y capas de lógica de negocio.

Inconvenientes:

- Es complicado de reutilizar.
- Requiere una interfaz para cada una.
- Mayor dificultad para disponer de mensajes de error concretos.

Justificación de la solución adoptada

Optamos por las excepciones (alternativa 2), puesto que estamos habituados a su uso y nos permiten mayor riqueza en los mensajes de error y más versatilidad.

Decisión 6: Diseño para la trazabilidad.

Descripción del problema: Necesitamos información de utilidad o sobre algún problema significante del sistema.

Alternativas de solución evaluadas:

Alternativa 1: Uso de logs.

Ventajas:

- Se pueden controlar eventos que suceden mientras el programa se está ejecutando. Especialmente eventos de interés para el programador.
- Podemos seleccionar el tipo de logs que nos convengan. Nosotros nos centraremos en los de tipo info y los de tipo error.
- Suelen ser una alternativa más adecuada y refinada a los clásicos "outprintln".

Inconvenientes:

- Se debe usar de manera moderada, es decir, no podemos usar logs en todos lados dado que el rendimiento se vería afectado.

Alternativa 2: Uso de auditorías.

Ventajas:

- Son una buena alternativa a los logs. Su uso está más orientado a las entidades. Por ejemplo, para comprobar si una entidad ha sido creada.

Inconvenientes:

- Su uso principalmente está orientado únicamente a administradores.
- Su implementación es más compleja que los logs.
- Hay que tener cuidado con su uso ya que cada operación implica la creación de una o más revisiones de auditoría

Justificación de la solución adoptada

Elegimos la alternativa 1., ya que nos parece más sencilla e intuitiva además de que se adapta mejor a nuestras necesidades.

Decisión 7: Diseño de páginas de errores.

Descripción del problema: Necesitamos implementar algún tipo de páginas de errores.

Alternativas de solución evaluadas:

Alternativa 1: Uso de páginas de errores personalizadas.

Ventajas:

- Se puede personalizar las páginas de errores al gusto y con ello ofrecer una mayor sensación de calidad.

Inconvenientes:

- Requiere un mayor esfuerzo de implementación y un mayor trabajo en general.

Alternativa 2: Uso de páginas de error por defectos.

Ventajas:

- Apenas necesita implementación.

Inconvenientes:

- No se pueden personalizar.

Justificación de la solución adoptada

Elegimos la alternativa 1., ya que nos parece que es la que mejor se puede adaptar a nuestro proyecto ya que implementaremos páginas de errores siguiendo un poco nuestra temática.