DP1 2020-2021

Documento de Diseño del Sistema

Proyecto RATEACHER

https://github.com/gii-is-DP1/dp1-2020-g3-18

Miembros:

- Cuevas Carrasco, David
- Fernández Angulo, Francisco
- Pardo López, Luis
- Portero Montaño, Juan Pablo
- Rodríguez Rodríguez, Tomás Francisco
- Rojas Romero, José Joaquín

Tutor: José María García Rodríguez

GRUPO G3-18

Versión 2

10/01/21

Historial de versiones

Fecha	Versión	Descripción de los cambios	Sprint
10/01/2021	V1	• Creación del documento	3

Introducción

El proyecto consiste en una página web orientada hacia la calificación de los profesores de distintas facultades, de cara a que el alumnado pueda acceder a dicha página y conocer los distintos puntos de vista acerca de los profesores de las asignaturas que van a tener o han tenido.

Diagrama(s) UML:

Diagrama de Dominio/Diseño

Un ejemplo de diagrama para los ejercicios planteados en los boletines de laboratorio sería (hemos omitido las generalizaciones hacia BaseEntity para simplificar el diagrama):

El diagrama de dominio/diseño se encuentra en el documento "DP_Sprint3_G3-18", este documento ha sido modificado con respecto al Sprint anterior. Las modificaciones están explicadas en el mismo documento al comienzo.

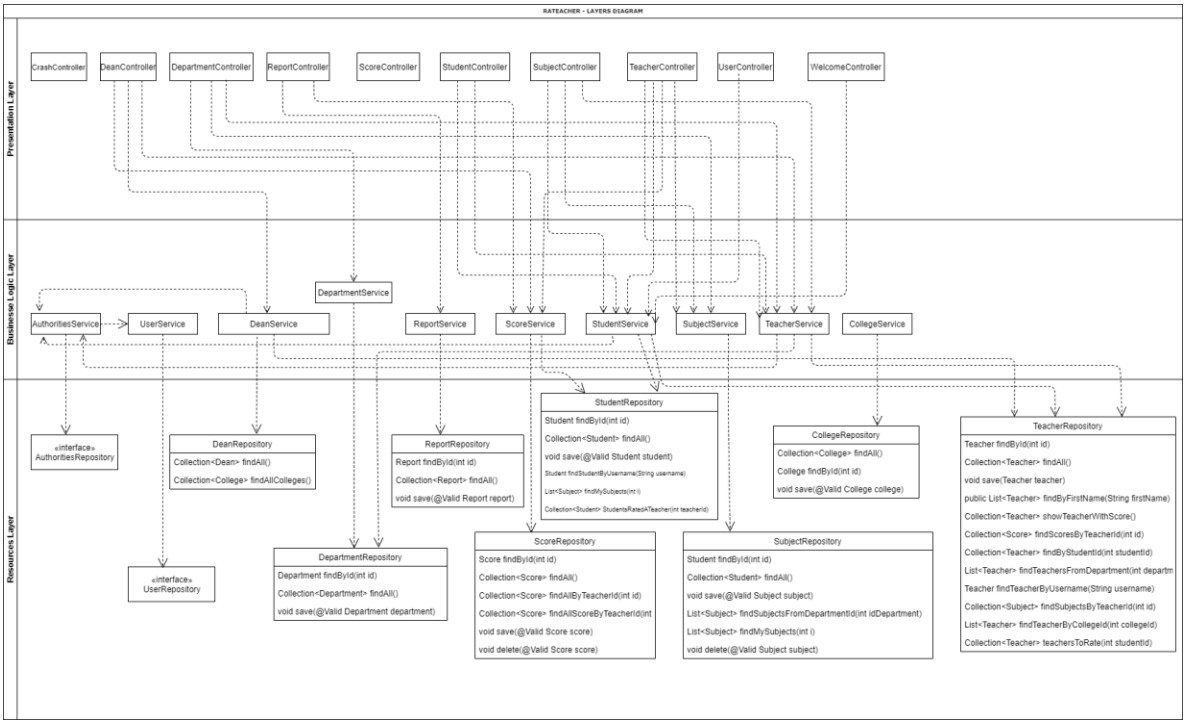
El diagrama UML de dicho documento muestra las entidades con sus atributos y respectivas restricciones. El diagrama también muestra las relaciones implementadas en el proyecto entre entidades junto a su multiplicidad.

Las validaciones del proyecto están implementadas según la complejidad de las reglas de negocio expuestas con anterioridad. Solo se han realizado validadores en caso de que alguna regla de negocio fuese suficientemente compleja como para no poder ser resuelta por una notación simple. Ejemplo: @Email.

Toda la documentación se encuentra en el proyecto en una carpeta llamada "Entrega Sprint 3".

Diagrama de Capas (incluyendo Controladores, Servicios y Repositorios)

NUESTRO DIAGRAMA DE CAPAS DE RATEACHER:



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: Layers + MVC

Tipo: Arquitectónico

Contexto de Aplicación

Nuestro equipo ha decidido seguir con el patrón arquitectónico por defecto que implementa el "framework" de Petclinic. Se trata de una combinación de un patrón arquitectónico en capas o "layers" donde diferenciamos 3 grandes capas transversales conectadas cada cual con las colindantes y obteniendo siempre datos de la capa inferior hacia la superior, combinado con el patrón MVC (Model Vista Controlador) el cual nos permite tener bien diferenciados los grupos que conforman la vista o

presentación de nuestra aplicación (gráfica), del Dominio (Servicios y entidades) y de los Recursos (Repositorios y acciones sobre la base de datos).

Clases o paquetes creados

Para simplificar un poco este apartado, simplemente comentar que las clases y paquetes creados conforman todos los patrones de diseño que se comentan justo debajo de las ventajas de usar este patrón, por lo que aquellas clases y paquetes que permiten dicho patrón arquitectónico se encuentran descritas en los distintos patrones de diseño.

Ventajas alcanzadas al aplicar el patrón

La principal ventaja que podemos destacar a cerca del patrón arquitectónico elegido es que al tratarse de una combinación de dos de los patrones arquitectónicos más usados a lo largo de la historia del desarrollo de aplicaciones web, podemos encontrar información fácilmente a cerca de la implementación, además de que es el patrón más desarrollado en las clases teóricas y por tanto tenemos a nuestra disposición muchos más recursos para poder implementar adecuadamente dicho patrón. A parte de esto, otra ventaja notoria sería la de la manera en la que las capas se relacionan entre si, pues dichas relaciones nos permiten "trackear" el flujo de los datos y saber mejor donde se encuentran los fallos o donde tenemos que realizar modificaciones más fácilmente que en otros patrones arquitectónicos más complejos, como el patron de Microservicios, por ejemplo.

Patrón: Template View

Tipo: de Diseño

Contexto de Aplicación

Las partes de la aplicación donde se ha aplicado el patrón "Template View" en nuestro proyecto se encuentran dentro de la ruta: "src/main/webapp/WEB-INF" donde encontramos 2 carpetas llamadas "jsp" y "tags" respectivamente, dentro de las cuales encontramos los archivos ".jsp" que permiten la correcta visualizacion gráfica de los datos y funciones que se estén llevando a cabo en cualquier momento.

Clases o paquetes creados

No se han creado ni clases ni paquetes, sino carpetas (mencionadas en el párrafo anterior) en las cuales encontramos los archivos ".jsp" correspondientes a cada tipo de dato.

Ventajas alcanzadas al aplicar el patrón

Este patron nos ha resultado interesante al aplicarlo, pues nos permite tener aisladas las vistas del resto de funcionalidades con un mínimo de conexion entre estas y así poder modificar la visualización de los datos de una manera más sencilla y limpia.

Patrón: Intercepting Filter

Tipo: de Diseño

Contexto de Aplicación

Las partes de la aplicación donde se ha aplicado el patrón "Template View" en nuestro proyecto se encuentran dentro de la ruta: "src/main/java/rateacher/configuration" donde las dos ultimas direcciones representan paquetes, siendo el último de estos el que incluye las clases necesarias para configurar la seguridad de nuestra aplicación a través de permisos que tienen los distintos roles sobre las rutas de URL posibles.

Clases o paquetes creados

Los paquetes son los dos últimos mencionados en el párrafo anterior los cuales incluyen las siguientes clases ".java": "ExceptionHandlerConfiguration.java", "GenericIdToEntityConverter.java", "WebConfig.java" y "SecurityConfiguration.java", siendo esta última clase la que lleva a cabo las tareas descritas en el contexto de la aplicación.

Ventajas alcanzadas al aplicar el patrón

Como pequeñísima introducción a la seguridad web y debido a su sencillez, este patrón nos ha parecido bastante útil para poder contemplar algunos protocolos de seguridad en un muy corto plazo y por ello hemos decidido seguir con el y no cambiarlo ni buscar alternativas.

Patrón: Metadata Mapper

Tipo: de Diseño

Contexto de Aplicación

Las partes de la aplicación donde se ha aplicado el patrón "Metadata Mapper" en nuestro proyecto se encuentran dentro de la ruta: "src/main/java/rateacher/model", siendo paquetes las dos últimas direcciones, cuyo contenido conforma la base de las entidades que forman parte del dominio de la aplicación.

Clases o paquetes creados

Los paquetes son los dos últimos mencionados en el párrafo anterior los cuales incluyen las siguientes clases: Authorities.java, BaseEntity.java, College,java, Dean.java, Department.java, ExternalEvaluation.java, NamedEntity.java, Person.java, PersonalExperience.java, ProfessionalExperience.java, Report.java, Reports.java, ResearchExperience.java, Score.java, Student.java, Students.java, Subject.java, Subjects.java, Teacher.java, Teachers.java, TeachingExperience.java, TeachingPlan.java y por último User.java. De aqui en adelante se mostrarán solo los nombres de las clases ".java".

Ventajas alcanzadas al aplicar el patrón

Las ventajas más importantes y la razón por la que hemos adoptado este patrón se debe a que mediante simples anotaciones (Ejemplos: "@Size, @Required,etc) podemos tener controladas algunas reglas de negocio muy sencillas en una primera fase que es la de la creación de las propias entidades, por lo que el trabajo se simplifica mucho.

Patrón: Data Mapper + Table Data Gateway

Tipo: de Diseño

Contexto de Aplicación

Las partes de la aplicación donde se ha aplicado la combinación de patrones "Data Mapper + Table Data Gateway" en nuestro proyecto se encuentran dentro de la ruta: "src/main/java/rateacher/repository", siendo paquetes las dos últimas direcciones, cuyo contenido conforma la base de las entidades que forman parte del dominio de la aplicación.

Clases o paquetes creados

Los paquetes son los dos últimos mencionados en el párrafo anterior los cuales incluyen las siguientes clases: AuthoritiesRepository, CollegeRepository, DeanRepository, DepartmentRepository, ReportRepository, ScoreRepository, StudentRepository, SubjectRepository, TeacherRepository y finalmente UserRepository.

Ventajas alcanzadas al aplicar el patrón

Las ventajas más importantes y la razón por la que hemos adoptado este patrón se debe a que nos permite disponer de una capa de "Recursos" conectada con la capa de "Dominio" de manera que podemos, mediande consultas "psql" y "h2-console" tratar directamente sobre los repositorios y acceder a los datos de una manera más controlada y segmentada, lo que nos facilita el desarrollo en equipo y se disminuyen notablemente los errores y la abstracción de estos.

Decisiones de diseño

Decisión 1: Implementación de reglas de negocio (decisión de diseño)

Descripción del problema:

A la hora de implementar las reglas de negocio en nuestro proyecto teníamos varias alternativas disponibles, pues se podrían codificar tanto en el controlador como en el servicio.

Alternativas de solución evaluadas:

Alternativa 1.a: Implementar reglas de negocio en una clase validadora empleanto el método "*Validator*".

Ventajas:

- Es mucho más efectivo y se pueden comprobar gran catidad de reglas de negocio de una manera mas detallada.
- Existe un tipo ("Validator") exclusivo para este apartado.

Inconvenientes:

- Difícil de implementar por la manera en la que se codifica ya que tienes que seguir un orden específico de ejecuciones de métodos para probar correctamente la regla de negocio que se esté tratando.

Alternativa 1.b: Implementar reglas de negocio directamente en los atributos del modelo.

Ventajas:

- Si la regla de negocio es básica y sencilla, lo más rápido es usar anotaciones, como en nuestro caso, "*@Size(min=1, max=5)*", directamente en los atributos. ***(RN-4** Rango de puntiación de 1 a 5).*

Inconvenientes:

- Esta alternativa solo cubre relgas de negocio muy sencillas.
- No sería buena práctica actuar directamente sobre los atributos.

Alternativa 1.c: Implementar reglas de negocio en la parte de los servicios.

Ventajas:

- Esta alternativa tiene una complejidad mas baja y también es efectiva como emplear una clase validadora.
- Hace uso de etiquetas.

Inconvenientes:

- El modelo relacional que tenemos dificulta un poco este proceso al tener relaciones un poco más complejas entre, por ejemplo, la puntuación de un profesor y el alumno que le ha puntuado.

Justificación de la solución adoptada

Nuestra decision fue hacer uso de las tres alternativas (a, b y c) pues cubrimos un espectro de reglas de negocio desde simples etiquetas en el modelo hasta la creación de una única clase validadora para probar un método en concreto ("*ScoreValidator*").

Decisión 2: Relación de las querys

Descripción del problema:

Una decisión importante fue la de elegir entre la relaciones etiquetadas *"@OneToMany"*, *"@ManyToOne"* , "@OneToOne" o *"@ManyToMany"* ya que obteníamos errores en el fichero *"data.sql"* originados por el etiquetado en el fichero *"model"* de la entidad en cuestión.

Alternativas de solución evaluadas:

Alternativa 1.a: Hacer uso de todas las etiquetas menos "*@OneToMany*" ya que no entendíamos como solucionar el error o si el propio framework dificultaba la manera de usarla.

Ventajas:

- Cubre inmediatamente muchas relaciones de nuestro modelado UML.
- Fácil de implementar en el modelo de una entidad.

Inconvenientes:

- La falta de esta etiqueta obliga a usar el resto, teniendo que elegir a veces la etiqueta que más se parezca al comportamiento que esperamos y estando más limitados.

Justificación de la solución adoptada

Como grupo decidimos la Alternativa 1.a pues no encontramos otras alternativas y fuimos capaces de cubrir todas las relaciones satisfactoriamente.

Decisión 3: Creación de un profesor como nuevo usuario del sistema

Descripción del problema:

Una decisión clave del proyecto era la de cómo crear uno nuevo profesor en el sistema, pues como se verá a continuación había varias alternativas.

Alternativas de solución evaluadas:

Alternativa 1.a: Hacer uso del rol Decano, es decir, solo un Decano podría crear un profesor y por tanto debería de existir un decano antes de .

Ventajas:

- Si solo un decano puede crearlo aumenta la seguridad de que alguien que no sea un profesor realmente se cree una cuenta como profesor, pues solo el decano puede hacerlo y este sabe con antelacion quienes son los profesores reales.
- Un cambio sencillo seria añadir en el fichero "*security*" la autorización del decano para crear el profesor.

Inconvenientes:

- Aumenta la complejidad de la entidad "*dean*", sobre todo la clase controladora y tambien aumenta la complejidad de algunas consultas "psql" en el repositorio.

Alternativa 1.b: Permitir al nuevo usuario registrarse como profesor.

Ventajas:

- Un profesor que aún no sea usuario y quiera darse de alta como profesor en el sistema solo tendría que rellenar el formulario del login indicando que es profesor.

Inconvenientes:

- Cualquier persona podría registrarse como profesor y eso no es lo que queremos. Una posible solución a eso sería pedir un código de confirmación en el registro de profesor el cual se enviaría previamente al correo para garantizar la autoridad.

Alternativa 1.c: El administrador es el que se encarga de ir introduciendo los profesores.

Ventajas:

- Un administrador se encarga de esta tarea en concreto por lo que periódicamente revisaría las peticiones de los decanos y estos los agregarían manualmente, por lo que habría también más seguridad.

Inconvenientes:

- Con este método necesitaríamos de un administrador que se encargase exclusivamente de esto y tendría que introducir en el fichero "data.sql" cada
  profesor para que este luego pudiese logearse.

Justificación de la solución adoptada

Como grupo decidimos la Alternativa 1.c, pues la vimos hasta el momento como la más efectiva y segura mientras pensamos en algún sistema mejor.

Decisión 4: No sabíamos como obtener el "ID" del usuario actual

Descripción del problema:

Necesitábamos saber el "ID" del usuario actual logeado para facilitar el paso de parámetros en las "requests" de los controladores y pasar satisfactoriamente de una vista a otra.

Alternativas de solución evaluadas:

Alternativa 1.a: Emplear funcionalidad de Spring que nos permitía obtener el "username" actual y de él obtener los datos necesarios.

Ventajas:

- Es la manera más correcta que hemos encontrado y que nos han enseñado para no complicar el proyecto.

Inconvenientes:

- Es fácil implementar la funcionalidad de Spring que nos da el "username" actual, lo difícil es encontrar dicha funcionalidad y entender como funciona perfectamente y los recursos que nos deja a nuestra disposición.

Alternativa 1.b: Crear una clase única la cual esta destinada a la obtencion del datos del usuario actual.

Ventajas:

- En una sola clase tendríamos todo lo necesario para tratar los datos del usuario.

Inconvenientes:

- Tendriamos que aprender a codificar dicha clase.
- No nos parecería un buen hábito dejar datos tan importantes alojados en un único sitio, pues podría vulnerar la seguridad de la aplicación.

Alternativa 1.c: Dejar este problema para un Sprint próximo y disponer de más tiempo para acabar otras funciones de la aplicación.

Ventajas:

- Ganamos tiempo para poder desarrollar otra parte del proyecto.

Inconvenientes:

- No es una buena práctica posponer problemas de desarrollo a fases más tardías pues tenemos muy bien aprendido que los errores tienen que solucionarse en las etapas más tempranas del desarrollo software para evitar mayores costes en el futuro.

Justificación de la solución adoptada

Como grupo decidimos la Alternativa 1.a pues de este modo podemos fácilmente ingresar parámetros en la URL y tener siempre disponible cualquier dato del usuario actual.

Decisión 5: Cambiar toda referencia a "Petclinic" por "Rateacher"

Descripción del problema:

Una vez obtenido un proyecto sólido se nos comentó en una reunión que debíamos borrar todo rastro de "Petclinic" pues no sería una costumbre lógica el tener nombres y referencias erróneas de proyectos pasados en nuestro proyecto actual, lo cual puede dar lugar a confusiones o a pérdida de tiempo al buscar los distintos ficheros.

Alternativas de solución evaluadas:

Alternativa 1.a: : Crear un proyecto limpio nuevo desde 0 y desarrollar una arquitectura de carpetas similar al del Framework de Petclinic a la que ir agregando los distintos paquetes y archivos desarrollados durante los últimos Sprints.

Ventajas:

- Es la solución mas lenta pero también la más efectiva al tener un proyecto controlado al 100% y saber lo que se puede modificar y lo que no.

Inconvenientes:

- Nos requeriría de un tiempo que no tenemos ya que habría que replicar la misma arquitectura de carpetas para poder seguir los mismos patrones de diseño y arquitectónicos comentados con anterioridad y además dejamos pasar esta decisión hasta que un profesor nos lo comentó en días cercanos a la entrega.

Alternativa 1.b: Cambiar únicamente los elementos más visibles.

Ventajas:

- Solo habría que localizar elementos renombrables fácilmente visibles y renombrarlos.

Inconvenientes:

- No estaríamos renombrando correctamente todas las apariciones de "Petclinic" en el proyecto por lo que no estariamos dando una solución acertada a este problema.

Alternativa 1.c: Hacer una busqueda profunda en toda la arquitectura de carpetas del proyecto y renombrar todo aquello que no provoque fallos en las rutas.

Ventajas:

- Consideramos que es una solución rápida y consistente pues nos ahorra tiempo en comparación a las anteriores alternativas.

Inconvenientes:

- No creemos recordar ningun aspecto teórico en el que den las bases de como renombrar los distintos archivos del proyecto por lo que al principio hicieron falta varios intentos fallidos en los que en cada uno el proyecto quedaba inservible.

Justificación de la solución adoptada

Como grupo decidimos la Alternativa 1.c pues pensamos que era la mejor relación de esfuerzo/tiempo que pudimos aplicar y que nos ha resultado exitosa.