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.

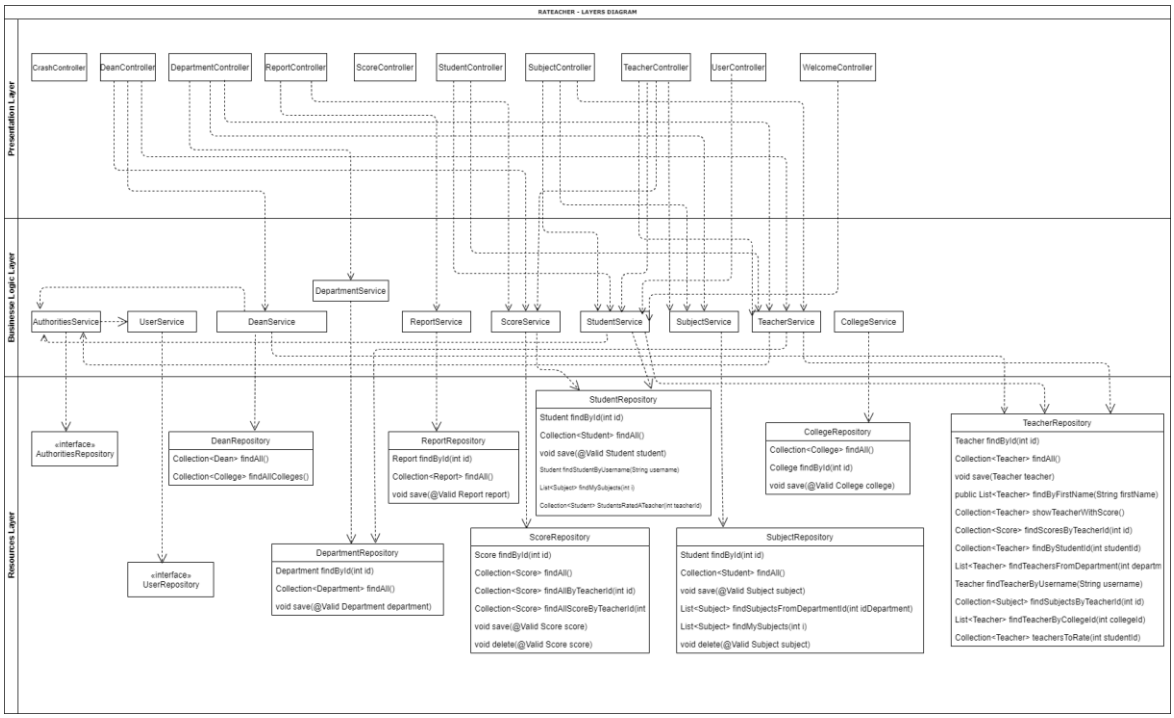
El diagrama UML de dicho documento muestra las entidades con sus atributos y respectivas restricciones. El diagrama también muestra las relaciones implementadas en el proyecto entre entidades junto a su multiplicidad.

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: Roles y acceso a funciones

Descripción del problema:

Una cosa muy importante que aprendimos con esta asignatura es el concepto de "rol" y todo lo que puede y no puede hacer un rol en concreto. Para evitar comportamientos no esperados como SQLInjections o la invasión de datos privados de los usuarios o de la propia aplicación, decidimos repartir las funciones entre los distintos roles de manera que solo sean accesibles si y solo si el usuario actual es un rol determinado.

Alternativas de solución evaluadas:

Alternativa 1.a: Ocultar botones y funciones a los roles que no deben tener acceso a dichas funciones.

Ventajas:

- Fácil de implementar gracias al "framework" de Petclinic.

Inconvenientes:

- Nunca antes habíamos hecho algo así por lo que al principio no sabíamos como hacerlo.

Alternativa 1.b: Dejar un único rol permitido al usuario final, siendo dicho rol el de "Estudiante".

Ventajas:

- Al tener un único rol como usuario final significa que es el único que puede tener acceso a las funciones de la aplicación por lo que solo aparecerían las funciones que puede realizar un estudiante y serían los propios

administradorse los que paralelamente agregarían los profesores y el resto de datos manual o sistemáticamente, evitando el resto de roles.

Inconvenientes:

- Se reduce muchísimo la funcionalidad de la aplicación y por tanto la experiencia como usuario final, lo cual no es permisible en este sector.

Alternativa 1.c: No ocultar ningún botón ni ninguna función pero solo podrá hacer uso de dichos elementos el usuario cuyo rol se lo permita.

Ventajas:

- El trabajo sería mucho mas sencillo y nos ahorraríamos algunos quebraderos de cabeza en cuanto a visualización final de la aplicación.

Inconvenientes:

- La aplicación tendria un aspecto visual poco llamativo y caótico, pues el usuario está viendo funcionalidades que ni siquiera puede usar.
- Se estaría dando información sobre muchas de las funcionalidades de la aplicación, cuyo uso podría ser destinado maliciosamente.

Justificación de la solución adoptada

Como entendemos que las funciones de nuestra aplicación van destinadas a distintos roles, los cuales también queremos que participen como usuarios finales de la misma, consideramos que la Alternativa de diseño 1.a es la más acertada, pues nos acerca más al comportamiento real de aplicaciones similares que podemos encontrar en Internet, lo cual facilita el uso del usuario final.

Decisión 2: Relación asignatura - departamento

Descripción del problema:

Una historia de usuario que nos facilitaba muchas conexiones entre las entidades era mostrar asignaturas por departamento, relación la cual no estaba contemplada en ningún diagrama-

Alternativas de solución evaluadas:

Alternativa 1.a: Añadir una nueva relación que satisfaga el problema.

Ventajas:

- Fácil de implementar y de añadir al completo de la aplicación.
- Soluciona muchos otros problemas de relación entre entidades como los profesores y los decanos.

Inconvenientes:

- Tuvimos que alterar un poco la direccionalidad de otras relaciones y ajustar un nuevo diagrama UML lo cual implicó grandes cambios en el flujo de datos.

Alternativa 1.b: No añadir la relación.

Ventajas:

- Al no añadir la relación, el problema se elimina y podemos seguir con el desarrollo del resto de la aplicación.

Inconvenientes:

- Se complica mucho las relaciones entre todas las entidades pues la supresión de la relacion asignatura-departamento involucraba cambios en todos los modelos que dificultaban el entendimiento de las relaciones.

Alternativa 1.c: Eliminar los departamentos.

Ventajas:

- Al no existir departamentos, las asignaturas no estarían agrupadas y sería elementos únicos sueltos los cuales pueden ser buscados y encontrados por cualquier alumno.

Inconvenientes:

- Se dificultaría un poco encontrar asignaturas en concreto al no poder filtrar por departamentos.
- Como existen tantas asignaturas de todas las facultades, las tablas y el acceso a la BD podria ralentizarse considerablemente al encontrarse tan atomizado.

Justificación de la solución adoptada

Como grupo decidimos la Alternativa 1.a codificar la relación necesaria porque además de agilizar muchisimo el trabajo posterior, nos supuso un reto que supimos superar pensando entre todos y nos motivó a detectar más fallos en las relaciones y corregirlos.

Decisión 3: Incompatibilidad de sentencias psql y h2-console

Descripción del problema:

Las "querys" de h2-console no servían en los repositorios del proyecto.

Alternativas de solución evaluadas:

Alternativa 1.a: Hacer única y exclusivamente consultas psql.

Ventajas:

- Es lo que llevamos viendo en algunas asignaturas de la carrera, por lo que a efectos prácticos, deberíamos ser capaces de implementar.

Inconvenientes:

- Obliga a unificar a todos los miembros del equipo de desarrollo bajo un mismo método de trabajo, lo cual era problemático al haber compañeros con conocimiento exclusivo sobre psql y otros compañeros sobre h2-console.

Alternativa 1.b: Añadir @Query(nativeQuery = true) en las consultas en el repositorio.

Ventajas:

- Con ese código se pueden realizar consultas como en h2-console.

Inconvenientes:

- Los desarrolladores que no entiendan bien h2-console, no pueden usar consultas de sus compañeros para tomarlas como ejemplo.

Alternativa 1.c: Hacer única y exclusivamente consultas h2-console.

Ventajas:

- Nos unificaría a todos en un mismo método de trabajo.

Inconvenientes:

- Nunca antes habíamos visto consultas en h2-console por lo que nos implicaría destinar parte de nuestro tiempo a entender nuevos conceptos.

Justificación de la solución adoptada

Como grupo decidimos la Alternativa 1.b pues de este modo cada uno podía trabajar de la manera en la que se sintiese más cómodo.

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: Obtener el id del usuario logueado da error

Descripción del problema:

Obtener el id del usuario logueado en la vista principal daba un error porque no se podía obtener id de usuario logueado ya que la primera vez que se accedía al sistema aún no se está logueado.

Alternativas de solución evaluadas:

Alternativa 1.a: : Implementar esta funcionalidad directamente en la vista necesaria.

Ventajas:

- Es la manera más directa que hemos encontrado y que funcione correctamente.

Inconvenientes:

- Hay que ir vista por vista comprobando si hace falta o no implementar los cambios, y en caso de tener que hacerlo, hay que aplicar los cambios tantas veces como vistas tengamos en el proyecto

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

Ventajas:

- Se concentra todo lo relacionado a obtener datos del usuario logueado en un único sitio.

Inconvenientes:

- Dificultad a la hora de implementar dicho cambio debido a que no sabemos como hacerlo correctamente.
- Falta de tiempo, pues esta decisión fue tomada a última hora una semana antes de la entrega y por tanto nos quitó tiempo que nos gustaría haber empleado en el desarrollo de mas tests.

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 el código implementado no era mucho y creemos que no tenemos que modificar mucho mas las vistas por lo que nos pareció una buena solución a corto-medio plazo.

Decisión 6: 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.

Decisión 7: La funcion delete, en general, no nos funciona correctamente

Descripción del problema:

Al aplicar la operacion "CRUD" delete, obteníamos algunos borrados de datos exitosos y otros no existosos (no se borraba), lo cual creemos que se debía a una mala codificación de las "Foreign Keys" del proyecto

Alternativas de solución evaluadas:

Alternativa 1.a: : Añadir al fichero "data.sql" la línea de código "SET FOREIGN_KEY_CHECKS=0;"

Ventajas:

- Hace que funcione perfectamente

Inconvenientes:

- No sabemos que incovenientes podría tener, por lo que de momento el inconveniente sería ser capaz de encontrar dicha línea de código.

Alternativa 1.b: No permitir que se pueda usar la funcion "delete"

Ventajas:

- Evitamos errores y agilizamos el trabajo

Inconvenientes:

- Tendríamos que limitar la cantidad de funciones "add" para que no haga falta el uso de funciones "delete"

Alternativa 1.c: Emplear otro método distinto para llevar a cabo las operaciones CRUD, en este caso, el "delete".

Ventajas:

- Podríamos obtener satisfactoriamente el resultado esperado.

Inconvenientes:

- No conocemos otra manera de implementar las operaciones CRUD en petclinic que no sea la proporcionada por los vídeos de teoría.

Justificación de la solución adoptada

Como grupo decidimos la Alternativa 1.a pues soluciona el problema a la perfección con el mínimo esfuerzo y recursos.