DP1 2020-2021

Documento de Diseño del Sistema

TeamWorks

https://github.com/gii-is-DP1/dp1-2020-g1-08

Miembros:

- Alonso Pontiga, Pedro
- Castro Bonilla, María
- de Ory Carmona, Nicolás
- Montiel Nieves, José
- Navarro Blázquez, Ignacio
- Rabasco Ledesma, Fernando

Tutora: Irene Bedilia Estrada Torres

GRUPO G1-08

Versión <0.6>

24/12/2020

Historial de versiones

Fecha	Versión	Descripción de los cambios	Sprint
24/12/2020	V0.1	● Creación del documento	3
28/12/2020	V0.2	● Añadidas varias decisiones de diseño	3
08/01/2020	V0.3	● Más decisiones de diseño y explicación de patrones	3
10/01/2021	V0.4	● Añadido diagrama de dominio y de capas	3
05/02/2021	V0.5	● Nuevo diagrama de capas	4
08/02/2021	V0.6	● Añadida la política de logs ● Añadido patrón MVC	4
09/02/2021	v0.7	● Corregido el Modelo de Dominio ● Añadida Leyenda ● Añadido índice de figuras ● Añadido formato ● Modificada descripción ● Corregido diagrama de capas ● Corregida introducción ● Añadidas decisión 6, 7 y 8 ● añadido patrón de diseño dependency injection	4

Contents

Introducción

TeamWorks es una aplicación web orientada hacia la gestión de equipos de trabajo, así como a la comunicación entre distintas empresas a través de un cliente de correo electrónico.

A través de la aplicación, empresarios o jefes de equipo pueden crear equipos de trabajo y añadir nuevos usuarios a estos, asignándole a cada uno los datos de acceso. Una vez se crea un equipo o empresa, se habilita un canal a través del cual los miembros pueden comunicarse por medio de mensajes de correo, sin revelar las direcciones de correo electrónico personales. El sistema permite adjuntar archivos y añadir etiquetas u otros elementos que se comparten entre los miembros de un equipo.

Por otro lado, en cuanto a la gestión de equipos, la aplicación permite al empresario dividir la empresa o equipo en varios departamentos. Se puede designar, para cada uno, un gestor de departamento que se encargará de añadir o retirar usuarios, crear nuevos proyectos o editar otros atributos específicos.

Los proyectos son la unidad mínima de organización de la aplicación. Puede crearse dentro de un departamento, y asignarse a varios miembros de este, así como designar un jefe de proyecto que se encargue de su gestión. Los proyectos pueden utilizarse para organizar un grupo pequeño de personas para llevar a cabo un desarrollo o conjunto de tareas contextualizadas.

Con el fin de facilitar la organización a los equipos de desarrollo ágil, la aplicación proporciona una interfaz de asignación de tareas e hitos. Se puede dividir el desarrollo del proyecto en hitos con fecha límite, y añadirle a cada uno un conjunto de tareas. El jefe de proyecto designado puede crear tareas, asignarlas a miembros del proyecto, y marcarlas con etiquetas.
Una vez acordadas y programadas las tareas a completar antes de una fecha determinada, los miembros del proyecto involucrados podrán visualizar en la pantalla principal un listado de tareas que podrán ir marcando como completadas cuando el usuario las haya llevado a cabo.

El desarrollo de la aplicación ha planteado la oportunidad de aprender nuevas tecnologías a lo largo de todo el stack. En backend hemos tratado en profundidad con Spring, un framework de java, mientras que en frontend se ha utilizado React para facilitar la estructuración de la aplicación como una SPA. Si bien todo ello ha supuesto numerosos desafíos, el resultado general ha sido muy satisfactorio.

Diagrama(s) UML:

Diagrama de Dominio/Diseño

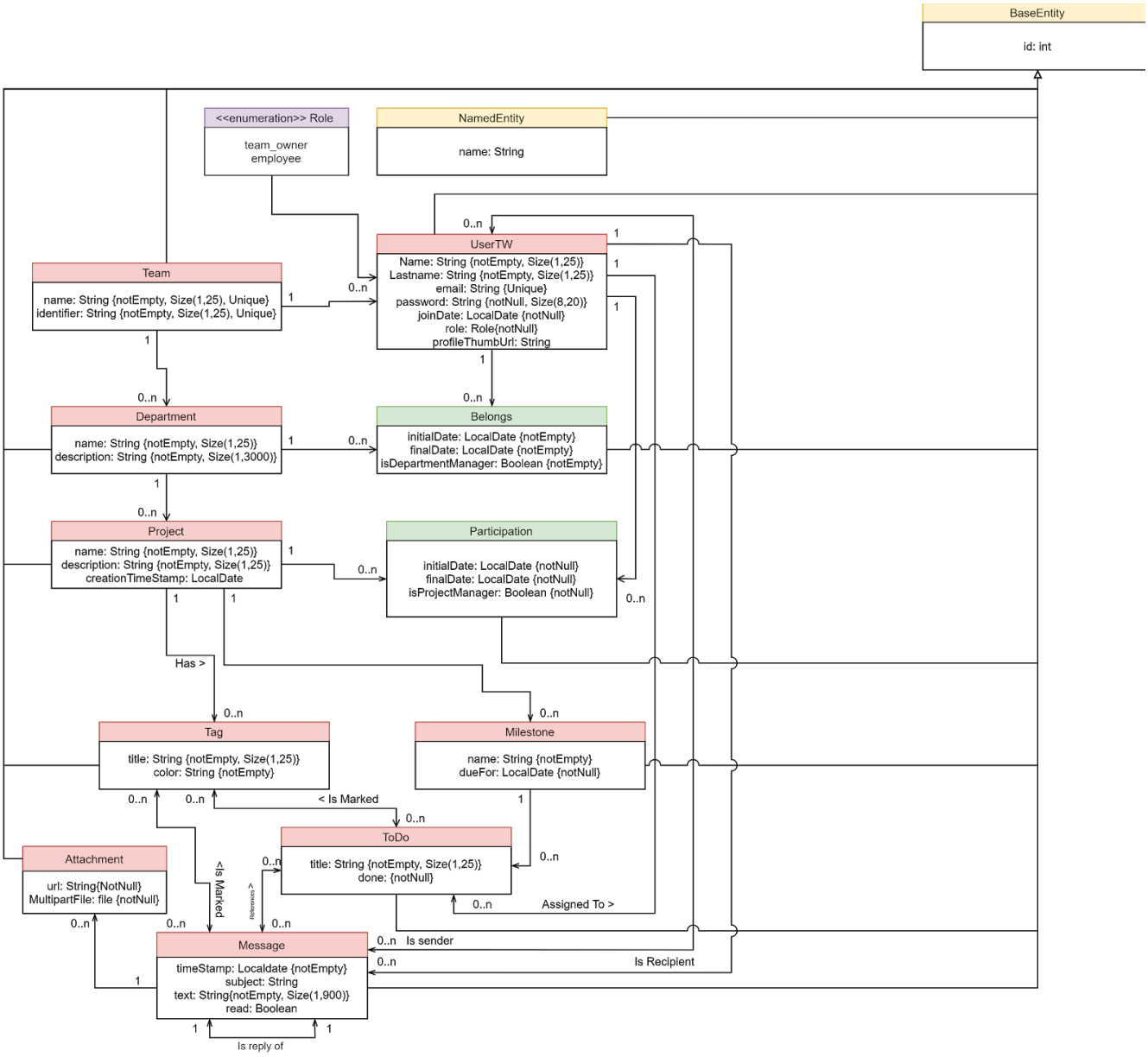

Figura 1. Diagrama de dominio

Leyenda UML

Rojo: Entidades
Verde: Entidades intermedias
Amarillo: Entidades Base
Violeta: Enumerados

Diagrama de Capas (incluyendo Controladores, Servicios y Repositorios)

Puesto que hay un gran número de relaciones y quizá no se aprecien todas correctamente, hemos dividido el diagrama en dos:

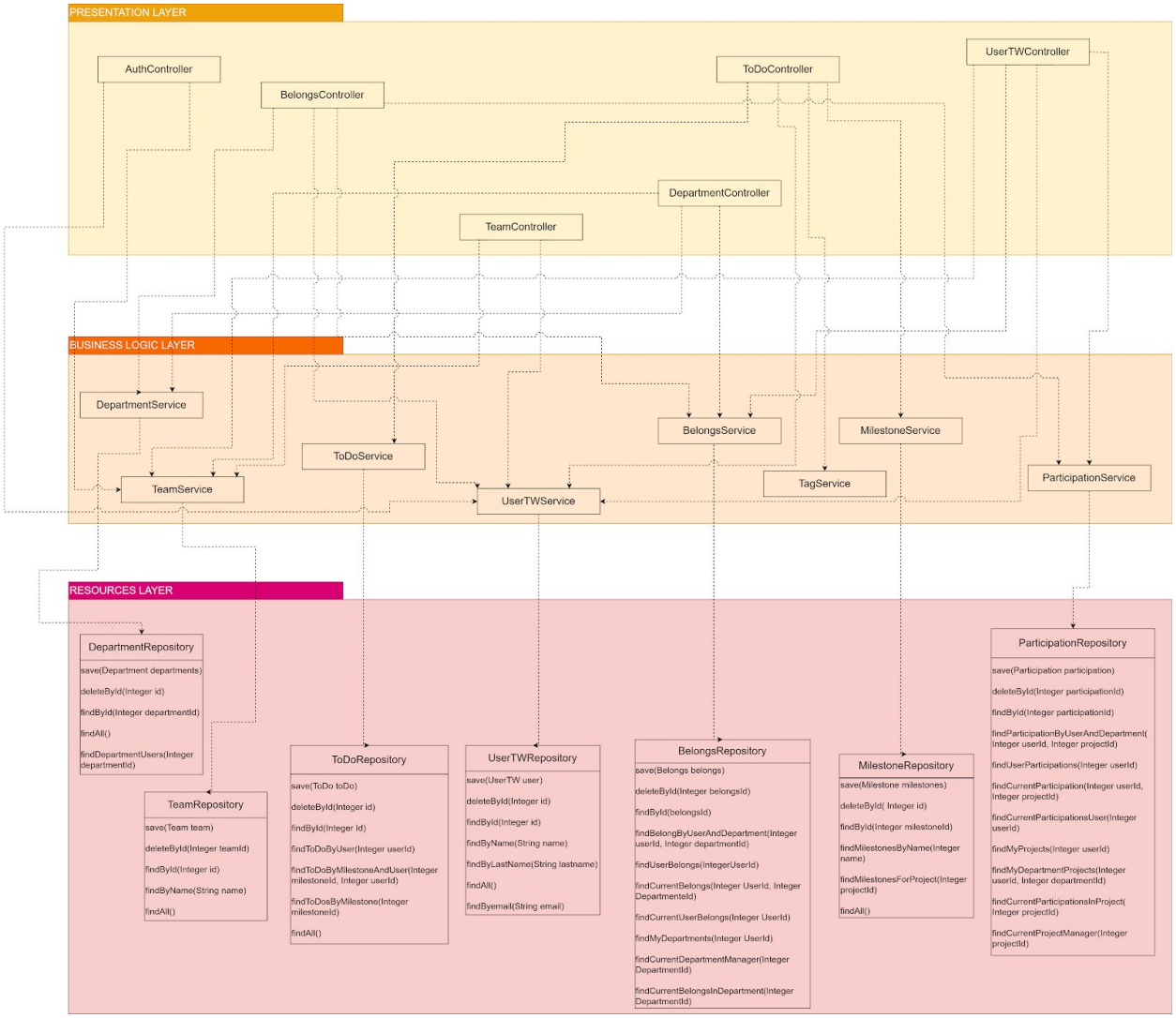

Figura 3. Diagrama de capas subdividido

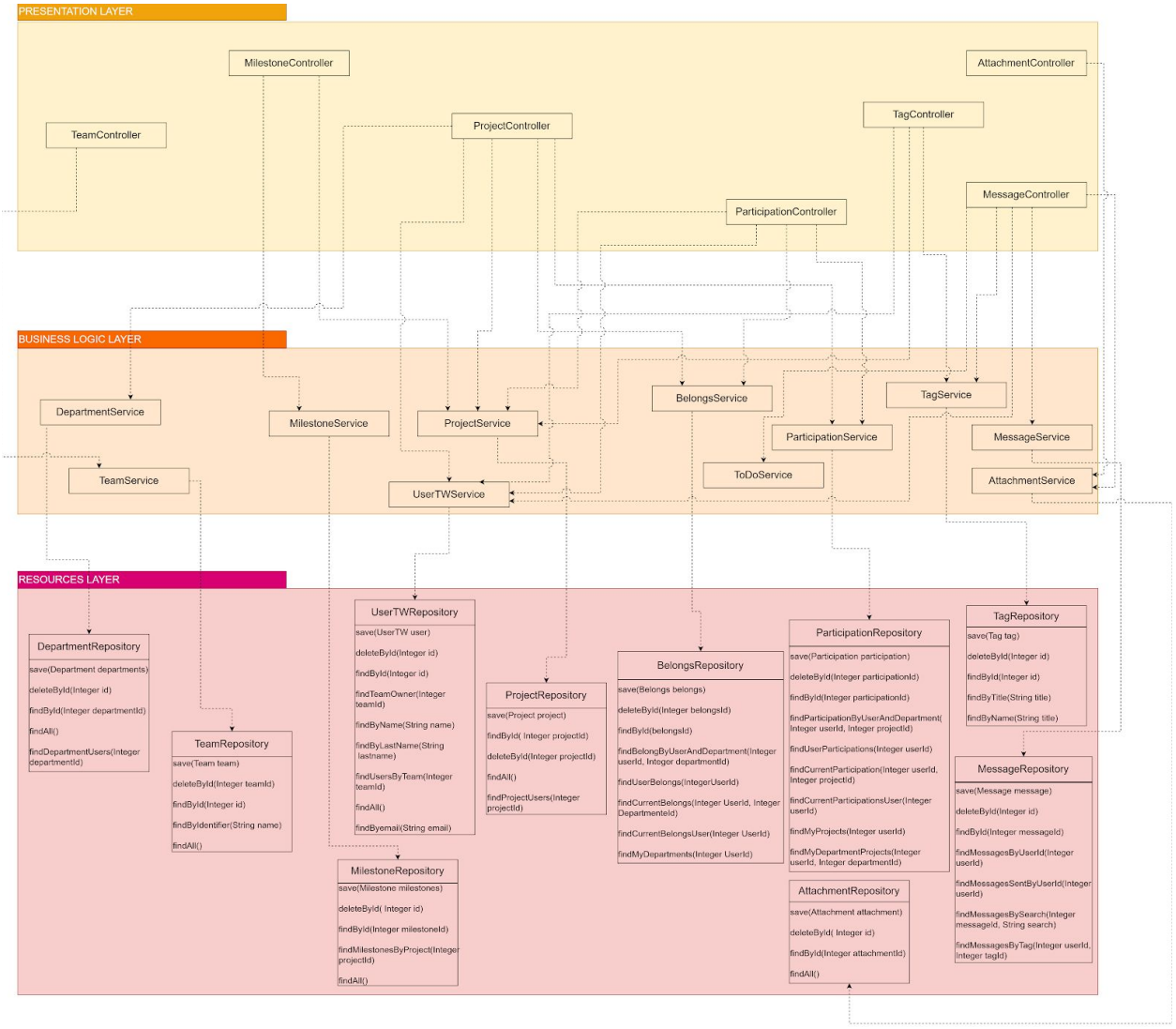

Figura 4. Diagrama de capas subdividido segunda parte

Patrones de diseño y arquitectónicos aplicados

Patrón: MVC

Tipo: Arquitectónico

Contexto de Aplicación

Hemos usado el patrón MVC en el proyecto al completo que se ha realizado con el framework de Spring. Para ser más exactos, la capa de presentación está formada por los controladores y las vistas, las cuales usan la tecnología ReactJS, y varios paquetes de esta. La capa de la lógica de negocio está formada por los servicios y por las entidades de nuestra aplicación, y por último, la capa de datos, donde se encuentran los repositorios.

Clases o paquetes creados

Excepto React que se encuentra en dp1-2020-g1-08/src/main/webapp/resources/frontend/ , todo el patrón MVC (controladores, servicios, repositorios,clases...) está contenido en dp1-2020-g1-08/src/main/java/org/springframework/samples/petclinic/ .

Ventajas alcanzadas al aplicar el patrón

Gracias a este patrón, hemos conseguido que nuestra aplicación tenga una alta cohesión y un bajo acoplamiento.
Los componentes de la aplicación tienen responsabilidades separadas y la aplicación soporta múltiples vistas.

Patrón: Dependency Injection

Tipo: de Diseño

Contexto de Aplicación

La Inyección de Dependencias es un elemento fundamental de Spring que consiste en que el framework gestiona y llama a las implementaciones de los subsistemas que conforman la aplicación, incluyendo servicios, controladores y repositorios. Esto permite a clases como aquellas del paquete org.springframework.samples.petclinic.web solicitar la inyección de las implementaciones de la capa de servicio.

Clases o paquetes creados

Las clases del paquete org.springframework.samples.petclinic.service son un ejemplo de clases que se registran en el sistema de inyección de dependencias de spring

Ventajas alcanzadas al aplicar el patrón

Utilizar este patrón permite una sintaxis mucho más fluida a la hora de desarrollar la aplicación, pudiendo construir objetos con unas dependencias determinadas, que Spring resolverá en tiempo de ejecución inyectando la implementación que considere más oportuna.

Patrón: RESTful API

Tipo: Arquitectónico

Contexto de Aplicación

El patrón RESTful en el diseño de APIs se puede ver aplicado en todos los controladores situados en el paquete `org.dp1.teamworks.web`.

Clases o paquetes creados

Todos los controladores REST están implementados en el paquete `org.dp1.teamworks.web`.

Ventajas alcanzadas al aplicar el patrón

Este patrón se complementa con la aplicación del patrón SPA, descrito más abajo, puesto que dado que el renderizado se delega al cliente, este se beneficia mucho de una estructura REST en el backend, que devuelva datos en formato JSON. Esto supone ventajas como una menor carga en el servidor, así como una mayor extensibilidad y facilidad de integración con el cliente.

Patrón: Middleware

Tipo: de Diseño

Contexto de Aplicación

El patrón middleware facilita la ejecución de lógica de negocio común a varios controladores, antes de que llegue el control al propio controlador. Se puede ver aplicado en la gran mayoría de controladores situados en el paquete `org.dp1.teamworks.web`. Spring facilita la integración de este patrón mediante su configuración en la clase `org.dp1.teamworks.configuration.WebConfig`.

Clases o paquetes creados

Todos los controladores middleware están implementados en el paquete `org.dp1.teamworks.middleware`.

Ventajas alcanzadas al aplicar el patrón

Utilizar interceptores de Spring (la nomenclatura que se utiliza en este framework) permite la ejecución de código ante una petición HTTP antes de que el control llegue a la clase controlador a la que iba dirigida la petición. Esto permite realizar comprobaciones y ejecutar lógica de negocio, entre las cuales se encuentran el que el usuario tenga una sesión válida, o tenga privilegios para realizar la acción solicitada.

Patrón: Single Page Application (SPA)

Tipo: de Diseño

Contexto de Aplicación

La interfaz de usuario de la aplicación tiene una complejidad considerable y tiene muchos elementos dinámicos, por lo que para evitar estrés innecesario en el servidor debido al renderizado de vistas, se ha optado por el renderizado en cliente. Por lo tanto, cuando el usuario hace una petición a la ruta raíz, se devuelve todo el código de la aplicación, que es una mezcla de HTML, CSS y Javascript.

Clases o paquetes creados

Todo el código dentro del directorio /src/main/webapp/resources/frontend/.

Ventajas alcanzadas al aplicar el patrón

Reduce considerablemente el estrés sobre el servidor, ya que una gran parte del trabajo de renderizado de vistas dinámicas se delega al cliente. Además, permite estructurar fácilmente el backend como una API RESTful lo más simplificada posible y basar las peticiones al modelo de datos en consultas AJAX. Facilita la gestión del estado puesto que toda la aplicación se carga en memoria en la petición inicial.

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: Utilizar React para frontend

Descripción del problema:

Las complejas vistas e interfaces de usuario que se esbozaron inicialmente para el proyecto han planteado dudas respecto a la tecnología a utilizar para renderizar las mismas. Se propusieron las siguientes alternativas:

Alternativas de solución evaluadas:

Nota: Por renderizar se entiende procesar los datos de los repositorios y transformarlos en una vista HTML que a continuación mostrará el cliente.

Alternativa 1.a: Renderizado en servidor

Ventajas:

- Mantiene una gran porción de la base de código en Java
- Sintaxis familiar con HTML mediante tags JSP
- Facilidad de mantenimiento
- Requisitos mínimos en el cliente, puesto que no tiene que renderizar la vista.

Inconvenientes:

- Deja de ser tan práctico cuando la interfaz es compleja
- Poco soporte para interfaces dinámicas (no resultan tan idiomáticas)

Alternativa 1.b: Renderizado en cliente sin framework

Ventajas:

- Permite escribir secciones de la aplicación con templates JSP y otras con Javascript o HTML puro.
- Mucha flexibilidad en la implementación
- Gran cantidad de documentación y facilidad de implementación de patrones de diseño conocidos.

Inconvenientes:

- Resulta muy complicado implementar patrones arquitectónicos como SPA (supone reinventar la rueda)
- Tendencia a diseñar aplicaciones monolíticas, muy difíciles de mantener
- Requiere consensuar entre todos los miembros del equipo la estructura de la aplicación: monolítica, en módulos, toolchain...

Alternativa 1.c: Renderizado en cliente con React.

Ventajas:

- Facilita el diseño de la aplicación como una SPA, con las ventajas que ello supone, descritas arriba.
- Cumple 100% con la división en capas de la aplicación.
- No afecta al trabajo diario de desarrollo y pruebas de la aplicación
- Código extremadamente modular
- Facilidad de mantenimiento
- Toolchains preparadas para producción
- Buena compatibilidad con APIs RESTful
- Muchos recursos didácticos en formato de guías, tutoriales y ejemplos

Inconvenientes:

- Añade un overhead en el renderizado de la página en el cliente
- Requiere conocer el framework por parte de todos los miembros implicados en el desarrollo frontend de la aplicación
- Requiere un conocimiento importante de Javascript
- Incrementa el tamaño de la aplicación
- Para aplicaciones simples, resulta una solución demasiado compleja

Justificación de la solución adoptada

Consideramos que la aplicación tiene una complejidad considerable, que cuando se junta con interfaces ricas y dinámicas, requiere recurrir a herramientas cuyo éxito a la hora de utilizarlas en proyectos de producción está más que demostrado. Por tanto, hemos optado por la alternativa C, aprovechando esta oportunidad para aprender un nuevo framework y crear una aplicación muy modular y que sea fácil de mantener en el futuro.

Decisión 2: Seguridad en los Endpoints

Descripción del problema:

Dado que se ha estructurado el backend de la aplicación como una API REST, se barajaron varias posibilidades respecto a los métodos de autenticación y seguridad en los endpoints. Se propusieron las siguientes alternativas:

Alternativas de solución evaluadas:

Alternativa 1.a: Utilizar la solución integrada Spring Security

Ventajas:

- Es la solución propuesta por defecto para la autenticación en Spring Boot
- Abstrae detalles de implementación sobre el middleware implicado

Inconvenientes:

- Poca flexibilidad
- Obliga a adaptar la solución de seguridad y los roles a Spring Security

Alternativa 1.b: Utilizar Interceptors de Spring

Ventajas:

- Es muy fácil utilizarlos para implementar el patrón middleware
- Control total sobre el código que se ejecuta antes de cada petición a controlador
- Abstrae lógica de negocio repetida en los controladores (DRY code)
- Permite comprobar lógica de roles más compleja

Inconvenientes:

- Es una solución “from scratch” y requiere investigar cómo funcionan los Interceptors, que son una infraestructura a más bajo nivel

Justificación de la solución adoptada

Para nuestro caso de uso, Spring Security no era lo suficientemente flexible y habría supuesto un esfuerzo de adaptación de la capa de datos para que funcionase con esta abstracción. Por lo tanto, optamos por la alternativa B.

Decisión 4: Modularidad en React

Descripción del problema:

La intención del equipo al adoptar React como tecnología de renderización de vistas en cliente, era utilizar una estructura en módulos y componentes. Esto se podía hacer de diferentes maneras, las cuales se detallan a continuación:

Alternativas de solución evaluadas:

Alternativa 3.a: Dividir por cada vista, y esta a su vez en los componentes que la componen.

Ventajas:

- Fácil de leer para alguien poco familiarizado con la base de código

Inconvenientes:

- No resulta muy modular
- Es complicado reutilizar los componentes en otras vistas

Alternativa 3.b: Abstraer la gran mayoría de elementos HTML, de manera que cualquier fragmento o componente de una vista pueda reutilizarse en otra.

Ventajas:

- Elementos mucho más pequeños y simples
- Modularidad máxima.
- Reutilización de código.

Inconvenientes:

- Más compleja
- Más esfuerzo para llevarla a cabo
- Requiere consenso entre los miembros del equipo

Justificación de la solución adoptada

Como buscamos la máxima modularidad y facilidad de mantenimiento de la aplicación, apostamos por abstraer el código lo máximo posible, aunque sea más complejo. Por lo tanto, escogimos la alternativa B.

Decisión 5: Loggers

Descripción del problema:

Se necesita seguir la traza de los controladores para poder tener información sobre las variables del entorno y los servicios llamados. Nos surge la duda de qué cuáles son los componentes que debemos loguear en el proyecto y a qué nivel loguearlos.

Alternativas de solución evaluadas:

Alternativa 5.a: Loguear todo a nivel trace

Ventajas:

- Información detallada de cada componente y sus variables
- Fácil debug de cualquier componente

Inconvenientes:

- Problemas de rendimiento
- Logear eventos que no aporten nada
- Dificultad de entendimiento debido a ser muchos mensajes
- Gran tamaño de archivo de log

Alternativa 5.b: Loguear controladores a nivel info

Ventajas:

- Información de las variables y eventos que ocurren en el controlador
- Fácil debug de controladores
- Facilidad de detectar fallos generales en servicios
- Pocos mensajes de log por lo que son fáciles de entender
- Se ocupa poco almacenamiento en memoria

Inconvenientes:

- No se registra información detallada de los errores de los services

Alternativa 5.c: No loguear

Ventajas:

- Pequeño aumento de rendimiento debido a no tener loggers
- Menos trabajo
- No hay archivo de log, por lo que no ocupa tamaño

Inconvenientes:

- Sin información sobre eventos relevantes

- Dificultad de detección de fallos

Justificación de la solución adoptada

Como buscamos un término medio, hemos decidido que la mejor opción sería la de loguear solo eventos importantes en los controladores . Por lo tanto, escogimos la alternativa B.

Decisión 6: Eliminaciones en cascada

Descripción del problema:

Se necesita actualizar la base de datos cuando se solicita la eliminación de una entidad que se relaciona con varios objetos. Se busca tener un control de la eliminación en cascada de todas las entidades, caso por caso.

Alternativas de solución evaluadas:

Alternativa 6.a: Utilizar las anotaciones @ManyToMany, @OneToMany con los atributos “mappedBy” y “cascade”

Ventajas:

- Aclara quién es la entidad dueña de la relación
- Necesario para garantizar un estado consistente de la base de datos
- Permite la eliminación en cascada de la mayoría de los casos

Inconvenientes:

- La eliminación en cascada en relaciones @ManyToMany es compleja y existen algunos casos límite que rompen la aplicación
- No tiene mucha flexibilidad a la hora de realizar la eliminación en cascada, en ocasiones eliminando más registros de la cuenta

Alternativa 6.b: Implementar lógica de eliminación en cascada personalizada

Ventajas:

- Fino control del comportamiento del sistema cuando se solicita la eliminación de una entidad
- Funciona en todos los casos

Inconvenientes:

- Requiere un mayor esfuerzo de implementación, teniendo que programar manualmente todos los métodos de eliminación en cascada
- Puede provocar que el modelo de datos se quede en un estado inconsistente.

Justificación de la solución adoptada

A pesar de que la alternativa 6.b suponía más esfuerzo de implementación, la alternativa 6.a no permitía realizar todas las operaciones en cascada que se necesitaban realizar, por lo que se ha optado por la segunda propuesta.

Decisión 7: Esquema RESTful

Descripción del problema:

Se busca utilizar una nomenclatura estándar para los endpoints de la API REST que expone la aplicación.

Alternativas de solución evaluadas:

Alternativa 7.a: Utilizar microendpoints.

Los microendpoints consisten en exponer de forma muy granular la funcionalidad de la aplicación. La nomenclatura tendría la siguiente regla:

Para endpoints de actualización – un endpoint por cada campo que se puede actualizar. Por ejemplo, si en una tarea se puede actualizar la lista de etiquetas y el nombre, habría dos endpoints: POST /api/toDo/updateTitle y POST /api/toDo/setTags

Para endpoints con métodos POST, PATCH o DELETE – explicitar el método en la llamada. Por ejemplo, para crear un proyecto: POST /api/project/create; para eliminarlo: DELETE /api/project/delete.

Ventajas:

- Acceso muy granular a la aplicación
- Al tener nombres distintos, permite un control mucho más fino sobre el nivel de autorización requerido para realizar cada acción

Inconvenientes:

- Muy difícil de mantener y documentar. Documentación Swagger casi obligatoria para que desarrolladores frontend puedan integrar la aplicación con el backend.

Alternativa 7.b: Reutilizar endpoints y cambiar el comportamiento según el método HTTP.

Utilizar un mismo endpoint para un grupo de operaciones CRUD, como /api/project, y realizar distintas operaciones en función del método utilizado: GET, POST, DELETE, PATCH...

Ventajas:

- Esquema muy claro y fácil de seguir a la hora de integrarlo con frontend
- Fácilmente testable

Inconvenientes:

- El registro de interceptors sólo toma como parámetro un patrón path, que no distingue entre métodos. Por lo tanto, habría que cambiar drásticamente la implementación de los interceptores para que sean conscientes del método HTTP con el que se llaman
- Podría provocar errores fácil de depurar en frontend si se llama al endpoint con un método incorrecto

Justificación de la solución adoptada

Puesto que esta decisión apareció más tarde en el ciclo de desarrollo, cuando aparece la necesidad de poner distintos niveles de autorización para distintos métodos, se vio el equipo obligado a migrar a la alternativa 7.a, a pesar de que inicialmente se comenzó con la 6.b, puesto que la lógica de interceptores era ya muy robusta y difícil de cambiar.

Decisión 8: Reglas de negocio

Descripción del problema:

Se busca elegir una solución para implementar la validación de las reglas de negocio

Alternativas de solución evaluadas:

Alternativa 8.a: Utilizar validadores personalizados de spring

Ventajas:

- Soportada por spring facilitandonos el uso de muchos recursos útiles.

Inconvenientes:

- No se pueden hacer validaciones semánticas complejas de las relaciones del dominio

Alternativa 8.b: Utilizar excepciones

Ventajas:

- Permite validaciones semánticas complejas de relaciones de entidades de datos
- Permite dar mensajes personalizados y detallados

Inconvenientes:

- Solo se lanza una excepción a la vez, lo que conlleva la imposibilidad de saber si el resto de reglas de negocio han sido incumplidas o no.

Justificación de la solución adoptada

Se adopta la alternativa 8.b ya que la mayoría de reglas de negocio tenían restricciones relacionales semánticas complejas. A causa de la complejidad de las relaciones entre las entidades involucradas en

las reglas de negocio, se descartó la opción 8.b. Sin embargo se han realizado también validadores en el binding de las clases concretas para controlar las restricciones simples.