**Proyecto Fin de Máster**

Inicio

Preparamos el entorno, para ello necesitaremos lo siguiente:

* Eclipse STS
* Lombok
* Postman
* MongoDB Compass
* Github Desktop
* Visual Studio Code

Como vamos a empezar por la parte de backend, empezamos creado el proyecto con Spring starter. Necesitaremos lo siguiente:

* Spring Data MongoDB
* Spring Web
* Spring Dev Tools
* Lombok

Modelo de Dominio

Emplezamos por el domain model, vamos a crear las siguientes clases:

* Department.java
* Employee.java
* User.java
* Vacation.java

Tendremos dos Enums para los roles dentro de la empresa y los roles de los usuarios de la app:

* EmployeeRoleEnum.java
* UserRoleEnum.java

En una primera instancia, no iba a añadir excepciones de en dominio, pero puede ser buena practica para añadir comprobaciones para que no se violen las reglas del negocio.

Vamos a usar RuntimeException, ya que en el dominio no tiene sentido capturar reglas. Simplemente lo que queremos es interrumpir el flujo porque se ha violado una regla.

Al ir añadiendo excepciones, tuve dudas sobre donde implementar las excepciones en la clase Employee, en la cual utilicé el patrón builder. Por descarte decido añadirlas antes del return del método public Employee build().

En el paquete de excepciones añadiremos lo siguiente:

* EmployeeDomainException.java
* UserDomainException.java
* VacationDomainException.java
* DepartmentDomainException.java

Creamos una clase en el paquete útil para validar las contraseñas:

* PasswordRules.java
* DniRules.java
* EmailRules.java

Con esto se termina de momento la parte del modelo de dominio de la aplicación, aplicando la arquitectura hexagonal.

Capa de Aplicación

Ahora vamos a continuar con la capa de aplicación, donde pensaremos que se puede hacer con las entidades que acabamos de definir en el dominio. Son los conocidos “casos de uso”, que representan las funcionalidades.

Una duda que me surgió en este punto es si la estructura de paquetes que definí para la capa de aplicación era correcta. Una vez creado el primer puerto de entrada y la primera clase de datos encapsulados para su uso, me di cuenta de que necesitaría servicios para registrar el usuario (quizá también repositorios para guardarlos), la solución fue preguntarle a la IA que debería hacer.

Otra duda que tuve fue si crear puertos de salida por cada caso de uso o crear un repositorio con todos.

La siguiente duda fue también, si crear un servicio que agrupase todos los casos de uso de un empleado por ejemplo o separarlo para cada caso de uso, me decido por la segunda.

Operaciones o funcionalidades:

* Registrar un empleado o dar de alta un empleado.
  + En este caso la idea es que se haga de dos maneras:
    - La primera es que los empleados del departamento de RRHH den de alta un empleado mediante una vista.
    - La segunda es que tenga la opción de subir un Excel para poder dar de alta varios empleados de forma masiva.
  + Al ser una arquitectura hexagonal el puerto no cambia a pesar de querer realizar esto, ya que el caso de uso es el mismo, lo que sí que cambiara serán los adaptadores que llaman al puerto.
  + Por otro lado, empecé haciendo los puertos con DTOs, pero según las buenas prácticas de arquitectura hexagonal y DDD lo correcto sería usar Commands:
    - Encapsula los datos de entrada.
    - Permite hacer validaciones previas al uso.
    - Logramos desacoplarlo de la interfaz.
    - Cliff utilizo Commands varias veces en las clases.

Capa de Infraestructura

Ahora tenía dos opciones, seguir con los casos de uso en la capa de aplicación o pasar a la infraestructura para comprobar que lo que he ido haciendo funciona.

Entonces vamos a pasar a la capa de infraestructura, comprobar que el registro de usuarios funciona y más adelante volveremos a la capa de aplicación para completar los casos de uso.

Empezaremos con la entidad de base de datos mongo:

* EmployeeEntity.java

Una vez creada la entidad toca pensar, ¿cómo vamos a conectar todo lo anteriormente realizado con el exterior?

Flujo:

* Recibir los datos desde fuera, desde las vistas de la app (formato JSON posiblemente).
* Transformarlos a un DTO (o Command).
* Transferirlos al servicio para que haga sus funciones con él.
* Persistirlo en MongoDB (así comprobamos que funciona).

¿Cómo continuamos?

Lo suyo sería comenzar por los datos que se reciben desde el exterior, es decir los request. Como las buenas prácticas dicen que no hay que trabajar con las entidades, necesitaremos un DTO que habrá que mapear.

* EmployeeRequestDto.java

Luego el mapper:

* EmployeeRequestDtoMapper.java

El mapper lo dejamos para luego, vamos con el controlador y cuando necesitemos el mapper lo creamos.

* EmployeeController.java

Problemas encontrados:

El primer problema encontrado ha sido que en el controlador, al realizar el POST para registrar los empleados, con Cliff devolvíamos una URI con el ID. SI bien es cierto que por ejemplo podría haberla devuelto con el DNI u otro campo, decidí utilizar el ID.

En el command y utilizado en la capa de aplicación no incluí el campo id (no es buena práctica) y en el DTO tampoco. La solución fue que al guardar el empleado en el servicio, este devolviese el id generado por Mongo. Cambiamos de void a Long.

Luego, tuve otro problema, y es que MongoDB genera por defecto IDs de tipo string (valueObject), entonces, cambié todos los ids de la aplicación a tipo String en vez de usar Long.

Por otro lado, tuve dudas sobre los repositorios utilizados en la capa de infraestructura, obviamente necesitaba uno que se conectase con Mongo para persistir los datos, pero me faltaba un adaptador que conectase con la capa de aplicación. Esto se le pregunto a la IA.

Entonces la estructura de paquetes y ficheros de la capa de infraestructura para el registro de empleados quedaría de la siguiente manera:

Texto

El contenido generado por IA puede ser incorrecto.

Ahora vamos a probar que todo funciona con Postman:

* Primero preparamos el POST en Postman:

Captura de pantalla de un celular

El contenido generado por IA puede ser incorrecto.

* Recibimos el siguiente error:

Texto

El contenido generado por IA puede ser incorrecto.

* Lo que significa que nuestras validaciones del dominio están funcionando.
* Como aun no hemos preparado los departamentos para trabajar con ellos, le añadimos un id de departamento ficticio = “departmentIds”: [1]
* Nos ha dado el mismo error… El error esta aquí:

Texto

El contenido generado por IA puede ser incorrecto.

* El Builder espera que le llegue una lista con los departamentos, como instancio un arraylist vacío, aun que en el JSON ponga un 1 nunca le llegan. Se crea un departamento temporal en el servicio para comprobar el registro y se prueba:

Captura de pantalla de un celular

El contenido generado por IA puede ser incorrecto.

En MongoDB quedaría así:

Captura de pantalla de un celular

El contenido generado por IA puede ser incorrecto.

Algo que no me cuadra es que no veo que se haya persistido el departamento, vamos a investigar… Vale, actualmente no hay un mapper que mapee los departamentos para guardarlos ni un servicio, con lo cual en principio está bien.

**¡No olvidarse de los departamentos, TODOs en el código como recordatorio!**

**Otra cosa importante es, que cada vez que se inicie la aplicación, los datos que había con anterioridad no se borren. Esto lo realice en el TFG utilizando JPA, un modo que creaba la BBDD y eliminaba todo y un segundo modo que una vez creada simplemente la actualizaba (investigar en MongoDB como sería).**

**Falta también el registro de usuario mediante un Excel, sería lo más cómodo si se quiere dar de alta a varios trabajadores o para la primera vez que se utilice la app en una empresa.**

Continuará…

Ahora vamos a añadir una llamada para obtener la lista de todos los empleados que hay en la empresa dados de alta:

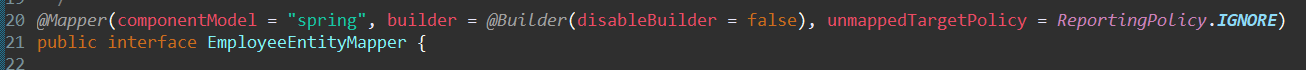
* GET getAllEmployees

Despues de añadir todo lo necesario en el servicio, puertos, adaptador y mappers, probamos la llamada y obtenemos el siguiente error:  
Captura de pantalla de computadora

El contenido generado por IA puede ser incorrecto.

Despues de investigar, esto sucede por que en el dominio hemos utilizado el patrón Builder. MapStruct lo que intenta hacer es simplemente new Employee(); pero nuestra clase no lo admite.

La solución posible es indicarle al mapper que use el builder que hemos creado en la clase Employee:



Probamos la llamada y obtenemos el mismo error, toca seguir investigando…

Lo que vamos a hacer es mapear a mano la EmployeeEntity la entidad de dominio y luego hacer lo mismo con la lista:  
Esto está explicado en el código

Ahora recibimos un error diferente, el cual hemos recibido anteriormente y es que el empleado debe pertenecer a un departamento, no a una lista vacía que es como había puesto. Creamos un departamento ficticio y volvemos a probar:

Captura de pantalla con letras y números

El contenido generado por IA puede ser incorrecto.

Luego implementados la llamada para obtener un empleado por ID:

* GET getEmployee(id)

Todo OK.

DEPARTAMENTOS

Aquí empecé creado un command pensando en el caso de uso, pero después de darle unas vueltas, creo que lo mejor es precargarlos en MongoDB y sino hardcodearlos directamente.

Empezamos creando la entidad:

* DepartmentEntity.java

Creamos el repositorio:

* DepartmentRepository.java

Ahora tenemos que crear una clase componente que cargue los datos al inicio de la app, en internet pone que en arquitectura hexagonal se suele crear un paquete llamado Bootstrap para ello.

* DepartmentLoad.java

Esta clase creara los departamentos al iniciar la aplicación, se añade la manera de crear los departamentos solo si no existen ya en la BBDD para así poder añadir y que no elimine los existentes.

VOLVEMOS A LOS EMPLEADOS

En los empleados hemos puesto TODOs con todo lo relacionado a los departamentos, ya que no los teníamos creados. Ahora vamos a completarlos.

Empezamos creando un puerto de salida para los Departamentos:

* DepartmentRepositoryOutputPort.java

Luego creamos el adaptador para que se comunique con la capa de aplicación:

* DepartmentRepositoryAdapter.java

Aquí nos damos cuenta de que necesitaremos un mapper para transformar los datos a la entidad de dominio, creamos el mapper:

* DepartmentEntityMapper.java

Ahora inyectamos el puerto de salida de los departamentos en el servicio de registro de empleados, obtenemos los ids y damos de alta al empleado con los datos obtenidos.

En un futuro será la responsabilidad del frontend el proporcionar mediante un formulario los ids de los departamentos a los que pertenecerá el empleado.

Vale, ahora también tenemos que hacer unos cambios en el mapper de employeeEntity, en el también utilice un departamento ficticio para el mapeo (se utiliza en el employee controller).

Esto fue más complicado y llevo un rato, se añadió la siguiente anotación aquí:

Texto

El contenido generado por IA puede ser incorrecto.

Como estamos utilizando Builder con MapStruct, este detectara departmentsToIds() y si coincide la validación hará el set de los ids que este obtenga. Esto lo hace gracias al component model = spring y al unmmapedTargetPolicy = IGNORE.

En el método lo dejamos así:

Texto

El contenido generado por IA puede ser incorrecto.

Pruebas:

Captura de pantalla de un celular

El contenido generado por IA puede ser incorrecto.

Texto

El contenido generado por IA puede ser incorrecto.

Probamos el GET por ID por si acaso:

Nos falla con el siguiente error ->

Texto

El contenido generado por IA puede ser incorrecto.

Vamos a ver por qué falla…

Hemos llegado al fallo, el cual está en la validación que hemos puesto en el dominio:

Texto

El contenido generado por IA puede ser incorrecto.

Para la solución se consulta a la IA

**“Mover la responsabilidad del mapeo completo (IDs -> Department) a donde tengas el acceso al repositorio”.**

**Esto lo que quiere decir es que hay que moverlo al EmployeeRepositoryAdapter, que es quien tiene acceso a la BBDD, los mappers no deben acceder a repositorios, no es buena práctica.**

1. **Inyectamos el puerto de salida de los departamentos en el adaptador de empleados.**
2. **Cambiar el método findEmployeeById actual**

**Texto

El contenido generado por IA puede ser incorrecto.**

**Por:  
Texto

El contenido generado por IA puede ser incorrecto.**

**Lo que hacemos es cargar aquí los departamentos desde Mongo.**

* + **Recupera EmployeeEntity desde la BBDD**
  + **Obtiene los IDs de los departamentos**
  + **Consulta Mongo para obtener los departamentos completos**
  + **Validar que los recuperados coinciden con los solicitados**
  + **Delegar en el mapper que reciba también la lista de departamentos**

1. **Cambiamos el EmployeeEntityMapper:**
   * **Antes creábamos un departamento con new Department(id, null);**
   * **LO cual hacía que saltase la validación y por lo tanto el error**
2. **Cambiamos también el método toDomainList(…) a lo siguiente:**

**Texto

El contenido generado por IA puede ser incorrecto.**

* + **Lo que hace ahora es mapear cada EmployeeEntity a su correspondiente Employee incluyendo los departamentos reales (con nombre, en vez de solo el ID)**
  + **Ya no utiliza toDomain(entity) el cual antes llamaba al método que creaba departamentos con nombre null.**
  + **Recibe una lista de departamentos desde el repositorio**
  + **Relaciona cada EmployeeEntity con sus departamentos correctos, filtrando por ID**

1. **Ahora tenemos que cambiar el método findAll() del adaptador para:**
   * **Consulta primero todos los EmployeeEntity**
   * **Consulta todos los departamentos**
   * **Pasárselos al mapper**

**Texto

El contenido generado por IA puede ser incorrecto.**

1. **Tenemos que añadir también un método en el DepartmentRepositoryAdapter que para poder consultar todos los departamentos:**

**Texto

El contenido generado por IA puede ser incorrecto.**

1. **Cambiamos el EmployeeResponseDto para que en la lista de departamentos le pasemos el DepartmentResponseDto:**

**Texto

El contenido generado por IA puede ser incorrecto.**

* + **También creamos el DepartmentResponseDto**

1. **Ahora tenemos que mapear también los departamentos a DepartmentResponseDto**

**Texto

El contenido generado por IA puede ser incorrecto.**

**En principio ya debería funcionar, lo comprobamos:**

Interfaz de usuario gráfica, Texto

El contenido generado por IA puede ser incorrecto.

DepartmentController

Pensando en que vamos a necesitar con los departamentos en el frontend, seguramente necesitemos mostrar los departamentos a la hora de dar de alta un empleado para seleccionarlos, entonces vamos con ello.

Creamos:

* DepartmentController.java

Creamos un método GET para obtener todos los departamentos:

Captura de pantalla de un celular

El contenido generado por IA puede ser incorrecto.

Creamos un método GET para obtener un departamento por ID:

Aquí me doy cuenta de que hemos estado haciendo algo mal, he utilizado en el controlador el output port de los departamentos cuando lo suyo es utilizar un input por.

Vamos a crearlo:

* DepartmentGetInputPort.java

Aquí principalmente tenemos que realizar los mismo que en el EmployeeGetInputPort, crear los métodos para obtener todos los departamentos y un departamento por ID.

Luego crear un servicio DepartmentGetService e implementarlos.

Vamos al adaptador he inyectamos el puerto input con los métodos.

Reemplazamos el repositoryOutPUtPort de controlador de los departamentos por este puerto de entrada.

Probamos que sigue funcionando todo:

Captura de pantalla de un celular

El contenido generado por IA puede ser incorrecto.

Antes de continuar vamos a repasar todo el código y comentarlo, aprovechando que lo tenemos reciente.

Hay que acordarse de implementar Swagger para docuementar la API cuando tengamos todos los controladores y endpoints implementados.

VACACIONES

Vamos a continuar con las vacaciones, ya que, si queremos seguir con el modelo de dominio de los usuarios, como estas implementar roles para el acceso (en un futuro) a diferentes funciones, vistas, …) necesitaríamos implementar spring security.

* Revisamos lo que hicimos en el dominio.

Texto

El contenido generado por IA puede ser incorrecto.

* Aquí faltan cosas, por ejemplo, días pedidos. Hacemos lista de lo que falta y lo implementamos:
  + Días pedidos
  + Estado -> Esto es importante, porque las vacaciones una vez las solicitas estas pasaran por diferentes estados. Una vez la envías estará en pendiente, pero si te las validan pasara a estar en aprobadas, sin embargo, si te las rechazan, a rechazadas.

**VACATON - APPLICATION**

Actualizamos el anterior punto y comenzamos con la capa de aplicación:

La implementación de las vacaciones se parece en cierto modo al registro de empleados, así que seguimos la misma estructura que tienen estos.

Creamos una clase command para encapsular los datos que se van a utilizar para registrar unas vacaciones. Se utilizan como objetos intermedios entre las capas.

* VacationRegisterCommand.java

No incluimos el ID en el command.

Ahora seguimos con los puertos, empezando por el de entrada:

* VacationRegisterInputPort.java

Este realizará las operaciones que el controlador quiere hacer, en este caso el registro de las vacaciones.

Continuamos con el puerto de salida:

* VacationRepositoryOutputPort.java

Este define las operaciones que el repositorio de las vacaciones va a necesitar (adaptadores también se incluyen).

* save
* findByEmployeeIdAndDateOverlap

El primero guardará las vacaciones. El segundo lo que hará será buscar en la base de datos las vacaciones del empleado y comprobará que no se solapen con las introducidas en esta petición.

Seguimos con el servicio de la capa de aplicación, que tiene la lógica principal para el registro de las vacaciones:

* VacationRegisterService.java

Contiene diferentes validaciones de los datos y utiliza los puertos para sus funciones.

**VACATION – INFRASTRUCTURE**

**USER – INFRASTRUCTURE**

**LOGIN – INFRASTRUCTURE**

**FRONTEND**

Vale, empezamos con el frontend de la aplicación.

En la última clase con Rubén dejé preparada una estructura para el proyecto de Angular la cual aprovecharemos.

Vamos a empezar con la pantalla de login, es decir la vista por defecto de la app será la página de login, donde el empleado introducirá su usuario y contraseña para acceder a la aplicación.

Servicio de autenticación:

* auth.service.ts

Páginas del login:

* login.page.ts
* login.page.html
* login.page.css

Actualizamos los archivos:

* app.routes.ts
* app.component.html
* app.config.ts

Aquí me doy cuenta de un problema grave, cuando el usuario inicia sesión en la app, según el rol que tenga se le mostrara una vista u otra.

Al probar el inicio de sesión en el local storage estoy guardando la información de DTO que he recibido del backend, la cual es la siguiente:

Una captura de pantalla de una computadora

El contenido generado por IA puede ser incorrecto.

Que tenga el rol del usuario está bien, ya que, a parte de los empleados, habrá un rol de usuario reservado para el administrador, el cual tendrá más privilegios sobre la app.

Sin embargo, para mostrar las vistas necesito el rol del empleado, ya que según el rol de empleado que tenga se le mostrara la vista correspondiente.

Toca volver al backend para cambiar la respuesta de la API.

Después de hacer los cambios en el backend…

\*Aquí ponemos los cambios realizados en el backend, hay que consultarlo con Cliff\*

Antes de continuar vamos a definir estilos globales que reutilizaremos durante la aplicación, para ello editamos el archivo styles.css ubicado en src.

Una vez definidos los estilos, vamos a probarlos en el login, modificamos levemente también el archivo propio login.page.css.

Como la idea es tener una aplicación coherente, necesitamos que todas las vistas sigan un patrón en el diseño. Para ello la idea es crear un layout que reutilizaremos en cada vista, este componente será la base donde trabajemos.

Para ello generamos el componente layout:

* ng generate component shared/components/layout

Lo siguiente es preparar el html del layout que servirá como plantilla: