

1:23:08




➡  Flujo de petición PUT?  
`/fields/{fieldId}`

## Controller

- No implementado
- Pero vemos

`configuration/access/fastapi_api/users_api/router.py:`  
como ejemplo ya que hay muchas similitudes

 Enviar DTO en lugar de argumentos  
por separado al caso de uso

- Está guapo porque si se mete un command bus de por medio simplemente hay que renombrar DTO a Command y enviarlo al bus en lugar del caso de uso.
- Problema si decidimos renombrar algún campo del DTO.
- Acoplamiento muy fuerte entre `CreateUserPayload` y `UserCreatorDTO`
- Ejemplo `organization_id` se quita del Payload para cuando lo importe el DTO tenga los demás elementos.

### Propuestas (a elegir)

1. Enviar los parámetros explícitos directamente a los casos de uso.
2. O cuando se creen los DTOs hacerlo parámetro a parámetros.

Con cualquiera de las dos, se elimina reticencia al refactor de rename al ser automático + posibles bugs introducidos por ello.

## Gestión de errores

- Actualmente todos los controllers tienen un try catch que si falla cualquier cosa es un `400`.
- Por otro lado, por el `AuthenticatedUserHook` se pueden lanzar `401`.
- En los errores `400` se le pasa de body el mensaje de error. Si se ha caído la base de datos también entraría allí.
- Le estamos dando información a los usuarios que no necesitan y que puede ser hasta peligroso.

### Propuesta

1. Controlar cada caso de error por separado. Ejemplo:

```
except InvalidFirstName as error:
    return JsonResponse(status_code=status.HTTP_400_BAD_REQUEST,
content={'error': str(error)})
except InvalidLastName as error:
    return JsonResponse(status_code=status.HTTP_400_BAD_REQUEST,
content={'error': str(error)})
except InvalidEmail as error:
    return JsonResponse(status_code=status.HTTP_400_BAD_REQUEST,
```

```

content={'error': str(error)})
except Exception as error:
    return JSONResponse(status_code=status.HTTP_500_SERVER_ERROR,
content={'error': 'Server error'})

```

1. Esto se va a empezar a repetir en cada controller. Se puede crear una función utilitaria que nos simplifique esta parte:

```

return await withErrorHandling(
    def():
        ...
        user_created = await user_creator.create(
            user_creator_dto=user_creator_dto,
authorization=authorization
        )
        ...
        return user,
    {
        InvalidFirstName: status.HTTP_400_BAD_REQUEST,
        InvalidLastName: status.HTTP_400_BAD_REQUEST,
        InvalidEmail: status.HTTP_400_BAD_REQUEST
    }
)()

```

- Con esto simplificamos cada controller + no hemos de añadir el código del 500 en cada lado.
- Próximamente lanzaremos un curso sobre esto. 🙌

## 🔑 ¿Dónde generar el ID?

- Ante la duda la llamada no puede ser un PUT , así que si hemos de soportar ambos casos, seguimos siendo un POST .
- Actualmente, si no hay ID se genera dentro de los DTOs  
user.py:41 , field.py:40
  - Se está generando un identificador fuera de la capa de infraestructura.
  - Dificulta el testing.
  - Dificulta el debugging.
- Una alternativa sería ponerlo en el caso de uso, pero estaríamos cubriendo un caso de “los clientes no nos saben utilizar” cuando

nosotros queremos que el ID venga desde fuera.

- También hemos de devolver el id creado, si lo hiciéramos en el caso de uso, nuestro caso de uso tendría que devolver valores (y no respetaríamos CQS).

## Propuesta

- Generar el identificador en el `PostController` .
  - Inyectando un `UuidGenerator` para que sea fácil inyectar una implementación diferente en testing.
  - Gracias a esto los tests de los casos de uso quedan igual y la generación de este ID si no existe no queda escondida.
  - Va muy de la mano de no acoplar los Payloads a los DTOs.
- Por otro lado, tener también el `PutController` el cuál no genera ID.



## Caso de uso FieldCreator



## Dónde añadir el código de autorización

- Estamos añadiendo magia en nuestra capa de aplicación.
- Ahora mismo el `authorizer` depende del `AuthenticatedUserHook` que depende de que sea una llamada API para hacer el control de autorización.
- ¿Qué pasa si podemos crear un `Field` directamente desde un controller y desde un subscriber?

## Propuestas (a elegir)

1. Hacer la autorización en el controller o middleware
  - Aquí la magia no molesta ya que es infraestructura.
  - Peligro de reusar el caso de uso desde otro sitio y olvidarnos el control.
2. Hacer la autorización más explícita en los casos de uso
  - Se inyecta en los caso de uso un `Authorizer` el cuál se le pasan explícitamente los argumentos que necesita.

- El Authorizer puede lanzar una excepción si no está autorizado para evitar en cada caso de uso hacer el `if !authorized` `throw UnauthorizedError`.
- Se puede hacer más limpio creando un servicio de dominio en `shared` para ello.



## Cómo llamar a otro caso de uso



Complementar información como patrón lo vemos a nivel macro después

- El código que checkea si ya existe el field y si es así exception es candidato a ser extraído a servicio.
- Actualmente ya existe un `FieldFinder`
  - Con nombre `Field Finder` se espera que busque un field por su id y que si no lo encuentra lance un error.
  - Esta clase hace más, depende lo que se le pase busca por una cosa u otra.
  - Propuesta de renombrado a: `FieldByCriteriaSearcher` o `FieldByFiltersSearcher` (diferenciar `Searcher` de `Finder`).
- Para nuestro caso de uso podríamos crear un servicio de dominio `FieldSearcher` donde simplemente hace
 

```
*self*.field_repository.get_by_filter(filter*="{\"id\":field_create_dto.id}")
```

  - Luego hablamos de los métodos de los repositorios.
  - Con este simplemente extraeríamos una línea de código a una clase.

### Propuesta

- Si este patrón se va a repetir mucho sacarlo a una clase, si no primero a un método.
- La clase se podría llamar `FieldDoesNotExistEnsurer`, que busca el Field y si ya existe lanza el error.
- Si es un método el nombre podría ser `ensureFieldDoesNotExist`.



## Crear Field

- Si usamos y queremos que el DTO llegue hasta field, el único en poder aceptar un `FieldCreatedDTO` debería de ser el método `create`
  - Actualmente también lo utiliza `delete`.
  - Si vemos `field_deleter.py:46` vemos que no hay necesidad de ello. Podríamos hacer directamente `field_to_remove.delete()`.
  - Luego hablamos del método `delete` del repository.
- Para no tener un método `Validate` y otro `fromPrimitives` que hacen casi lo mismo, y dado que el `validate` sólo se debería de llamar desde el `create`, centralizarlo todo desde el `fromPrimitives` y que el `create` llame allí.
- Estamos acoplando los primitivos de field al contenido del evento de dominio. Esto está bien porque nos permite ir más rápidos, pero pasa igual que con los DTOs: Añade miedo al refactor.
  - Mejor ser explícitos y pasar todos los parámetros.
  - Así podemos mantener un esquema de los eventos de dominio pese a hacer algún refactor de un método interno.



## Relacionar field con account

- Entendemos que un account puede tener múltiples fields y que un field puede pertenecer a diversos accounts.
- Tal y cómo está el código estamos acoplando nuestro código a cómo estaría implementado en base de datos.
- Ahora estamos haciendo que el repository de `Field` conoce tanto a `Field` como a `FieldAccount`.
- `FieldAccount` no es nada más que una relación de `field_id` y `account_id`.

### Propuesta

- No tratar a `account` como una entidad, sino como un VOs.
- Transformarlo en una lista `account_ids`.
- En base de datos puede estar modelado de la misma propuesta actual, pero en el momento de recuperarlo se pueden unir los datos.

- Alternativamente se podría guardar todo en el mismo campo de json.
- Importante valorar quién más consume de la base de datos. Si el data lake consume directamente de estas tablas, seguramente una tabla separada vaya mejor.
- De esta forma el mismo evento de `FieldCreatedDomainEvent` puede llevar también el `accountId`.
- En un futuro si se añaden más cuentas o se quitan, se dispara el evento de `AccountAssignedToFieldDomainEvent` o `AccountRemovedFromFieldDomainEvent` (adaptado a vuestro lenguaje ubicuo).
  - El nombre actual es `FieldAccountCreatedDomainEvent`, creemos que el nombre propuesto es más específico y habla más de domino.
- Si os interesa saber quién creó el `field` se puede añadir un campo `account_creator_id`

## Logging

- Actualmente el logger está añadido dentro de los casos de uso.
- Aunque inyectemos una interfaz, estamos inyectando código de infra en nuestra aplicación.
- El logger no es un requisito de negocio.

## Propuesta

- Añadir la capa de logging dentro de las implementaciones de la capa de infraestructura.
- Podemos añadir uno en la implementación de guardar del repository y otro en el publish del event bus.
- De esta forma ensuciamos la infraestructura y no las otras capas.
- Diferenciar logging de monitoring de telemetría.

## Qué devuelve un caso de uso

- Ahora mismo devuelve un `FieldDTO` que extiende de `FieldCreatedDTO` para añadirle un `factory method`.
- Los casos de uso de creación no deberían de devolver nada. Si hace falta por el ID que se ha generado, si lo hemos hecho desde el

controller eliminamos esa necesidad.

- Pero hay casos de uso (Finders...) donde sí que hace falta. Para ellos:

### Propuesta

- Devolver siempre lo que devuelve el `toPrimitives`.
- De esta forma no hace falta crear tantos DTOs.
- De esta forma tampoco hacemos leaks de cómo están implementado internamente nuestros agregados.



## Repositorios



### Tener repositorio base como `GeneralEntityRepository`

- Estamos obligando a implementar muchos métodos a nuestras implementaciones
- Seguramente, al no utilizar todas nuestras implementaciones quedarán vacías
- Hace más complicado saber qué hace cada repository
- Si un repository sólo puede guardar y buscar por id, es mucho más fácilmente verlo en la interfaz que buscar la llamada a cada método

### Propuesta

- Eliminar esa clase y que cada repository tenga sus métodos



### save vs update

- Aunque técnicamente estamos acostumbrados, cuando pensamos en un repositorio (por ejemplo una biblioteca), me da igual que un libro esté vacío o lo acabe de rellenar, lo que quiero es guardarlo allí.
- Concepto update no existe en NoSQL, entonces sabríamos detalle de implementación SQL



## Propuesta

Dejar sólo el método `save` y este se encarga de hacer `update` si es necesario.

## Método de delete

- Ejemplo `field_deleter.py:40`
- El patrón va a ser:
  1. Buscamos `field`.
  2. Ejecutamos método `.delete()`.
  3. `repository.delete(entity)`.
    - Le pasamos la entidad entera
    - Porque es posible que tenga que borrar datos de otras tablas que están en esa entidad. Ejemplo `accounts`.
    - Por lo tanto se encarga de eliminar todos los datos de su agregado aunque esté repartido en otras tablas.
  4. `entity.pullDomainEvents()`.
    - Si hay otros agregados que han de borrar datos suyos lo hacen reaccionando a este evento.



## Procesos en batch

- Es posible que hayan veces que queramos hacer cosas con diversas entidades.
- Ahora mismo estamos haciendo operación a operación (cosa que es costosa a nivel de BD).

## Propuesta

- Modelar nuestra propia clase `Collection`. En este caso `Fields`
- Ejemplo `field_deleter.py:40`
  1. `fields: **Fields** *await*`  
`*self*.field_repository.searchBy(...`
  2. `fields.deleteAll()`
    1. `self.fields.forEach(delete())`
  3. `*await* *self*.field_repository.deleteAll(fields)`
  4. `fields.pullDomainEvents()`