

Segundo Parcial 2021

Hecho por: Centurión, Franco; Idañez, Lucía; Lamas, Chabela

[Segundo Parcial Modelo - 2021.pdf](#)

Dudas

- Vale la pena poner los atributo el promedio_de_trabajo, cantidad_traducciones, frecuencia? Si no se pusieran, no sería menos performante hacer los calculos (por ejemplo ,para calcular la cantidad de traducciones habria que agarrar todas las traducciones y hacer un size)? PONER EN MISMA TABLA

+ metrica_traductor		
PK	id reputacion frecuencia cantidad_traducciones promedio_tiempo_resolucion	INTEGER INTEGER INTEGER INTEGER INTEGER

Campos que PODRÍA tener la tabla metrica_traductor

- Está bien crear una tabla archivo? Para determinar el archivo fuente y el final de la traducción SI, LLAMARLO DOCUMENTO
- Esta bien tener un balanceador de carga en microservicios? y si fuera así, cómo manejamos el tema de las sesiones? ya que tenemos tokens. SI, SERVIDOR STATELESS.
- Si o si un microservicio es un cliente pesado? TIENDE A SERLO

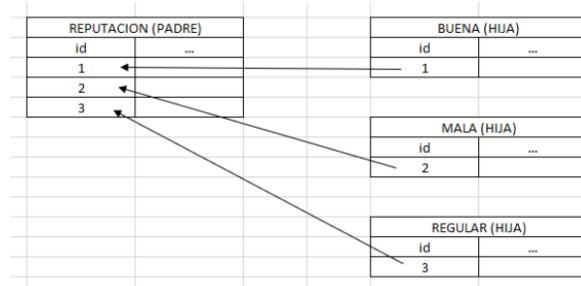
- En la pregunta 4 de arquitectura, no podría ser una cola de mensajes para el pago también?
 Como el caso que se había discutido en clase, sobre el telepase. Pensamos que el procesamiento de un pago debería ser algo más de estilo sincrónico para dejarle al usuario con la tranquilidad de que se pago fue recibido. PROCESO DE PAGO ES SINCRONICO → API REST. SI ES ASINCRONICO → COLA DE MENSAJES

1. PERSISTIR COMPOSITE Y DECORATOR

2. CUANDO SE USA MVVM Y MVP

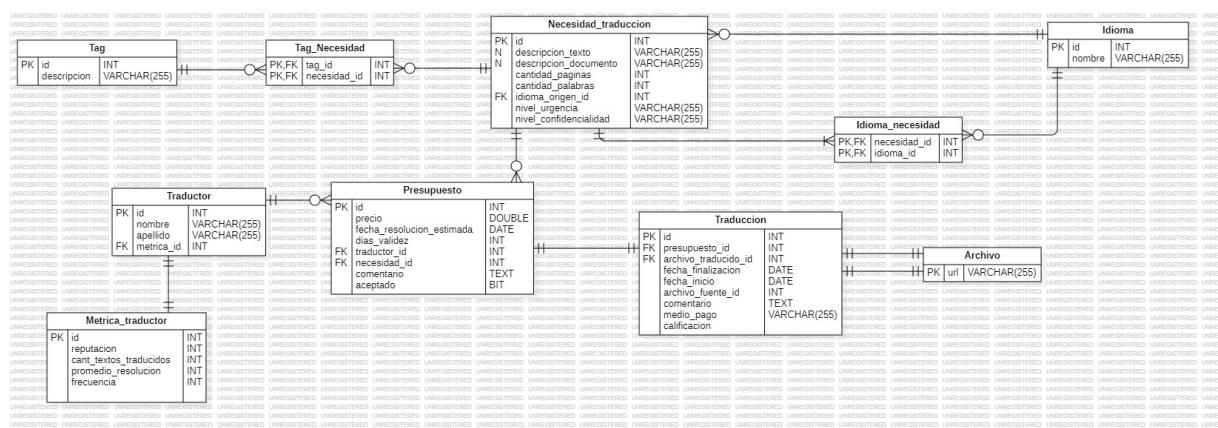
Notas

- FK siempre del many
- Text es más largo que VARCHAR
- JOIN ventajas: nos podemos traer de la clase padre, todas las hijas (porque tiene sus ids) y es útil ésta cuando las hijas tienen atributos en común



Ejemplo JOIN

Modelado de datos



- **nivel_urgencia** y **nivel_confidencialidad** decidimos modelarlo con un ENUM, ya que consideramos que no es un atributo que tenga gran frecuencia de cambio. Por este motivo, a la hora de modelarlo, se guardará en la tabla de Necesidad_traducion como un campo de tipo VARCHAR, y se utilizará un converter para pasar de este tipo de dato al tipo de dato del ENUM y viceversa.
 - Otra alternativa hubiera sido hacer tablas para el nivel de urgencia y el de confidencialidad, para tenerlos previamente tipados ahí, pero no lo hicimos de esta manera ya que es menos

performante debido a la necesidad de hacer joins, y además no son campos que varíen mucho en el tiempo.

- La entidad **necesidad_traducción** debe ser persistida debido a que se debe permitir la búsqueda web de la misma
- Creamos una tabla traductor ya que consideramos necesario mantener una trazabilidad de quién genera el presupuesto
- Decidimos hacer una entidad **Metrica_traductor** para guardar toda la información relacionada con las estadísticas del mismo. PODRÍA ESTAR EMBEBIDO EN TRADUCTOR
- Creamos una entidad traducción que sea la oficialización de la **necesidad_traducción** con el presupuesto seleccionado

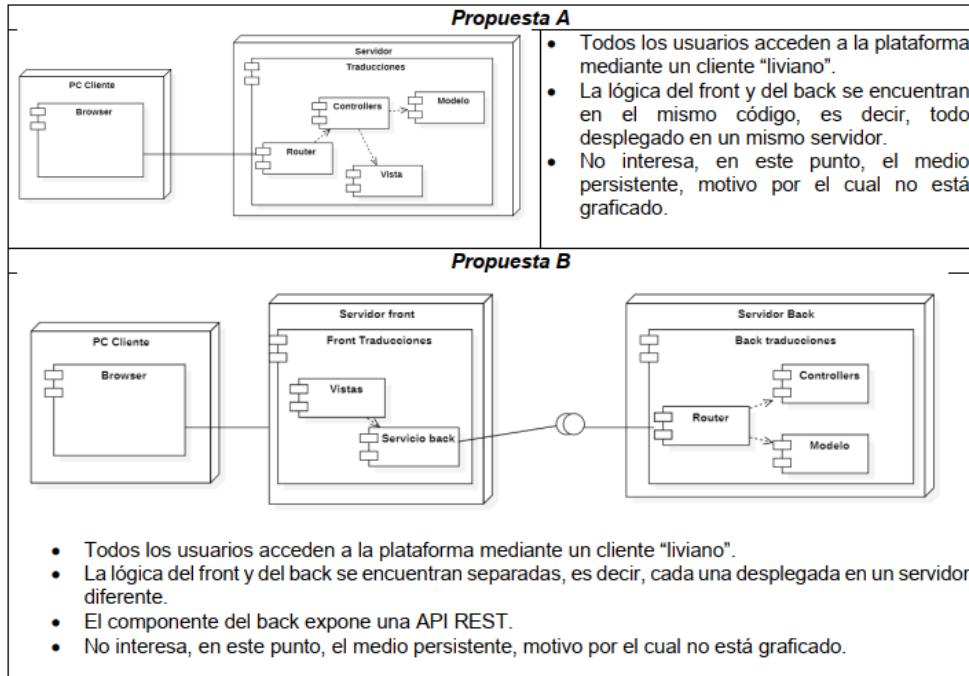
Arquitectura

1. Sabiendo que los traductores tendrán una frecuencia de uso alta del aplicativo y que los dueños de los textos y/o documentos no necesariamente tendrán la misma frecuencia de uso; ¿considera que podría ser beneficioso utilizar un SSO (Single Sign On)? ¿Por qué? Justifique adecuadamente su respuesta

RTA: Sí vale la pena el uso de SSO ya que debemos tener los usuarios registrados al éstos no ser esporádicos. Los traductores tienen frecuencia alta y los dueños de textos y documentos más baja, por ello no se debería optar por una manera compleja de generar los inicios de sesión.

2. Compare las siguientes propuestas de arquitectura en base a los siguientes atributos de calidad, detallando los aspectos (pivotes) que tuvo en cuenta para su comparación:

- Performance
- Mantenibilidad
- Escalabilidad
- guardan cosas → repo, cubo → nodo, cuadrados → componentes, back traducciones proyecto



	Propuesta A	Propuesta B
Performance	En este caso, la propuesta puede ser más performante en cuanto a <u>procesamiento</u> ya no hay tiempo de espera por la respuesta del servidor backend	Es menos performante debido a que el back-end y el front-end se encuentran conectadas por API REST, causando un tiempo de respuesta del lado del servidor. A raíz de ello, se puede ver afectado el <u>procesamiento</u> de la respuesta del Servidor Back por parte del Servidor Front.
Mantenibilidad	En este caso, al estar todos los componentes dentro de un mismo servidor, puede mejorar la <u>analizabilidad</u> de los errores. Sin embargo, supone un problema porque si se hace un cambio ya sea en el back o en el front, se debe volver a <u>deployar</u> todo el aplicativo.	En este caso, al estar separados ambos servidores (backend y frontend) resulta más complejo <u>rastrear el error</u> entre las transacciones realizadas. La ventaja de esto, es que ante un cambio en uno de los dos <u>nodos</u> , solo de debe <u>deployar</u> aquel en el que se implementó el mismo.
Escalabilidad	Al encontrarse todo en un mismo <u>servidor</u> , al momento de <u>extender las funcionalidades</u> debemos " <u>escalarlo a la misma velocidad</u> "	Es más escalable debido a que el servidor se encuentra dividido en el back y en el front con uso de una API REST, permitiendo que cada uno de los <u>nodos extiendan sus funcionalidades</u> individualmente y a <u>distintas velocidades</u> ya sea horizontalmente como verticalmente.

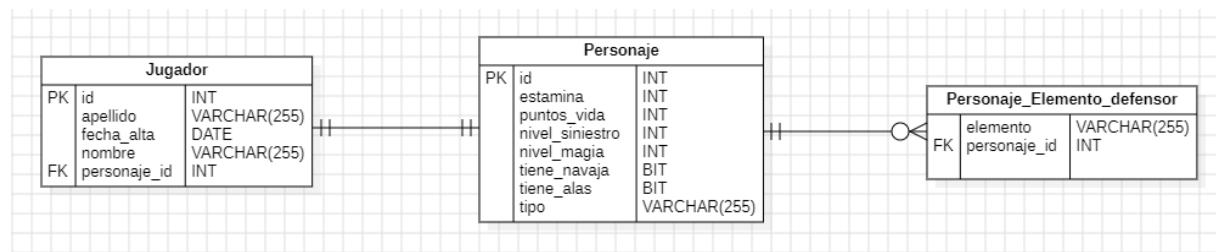
3. En base a lo desarrollado en el punto anterior y sabiendo que, debido al escaso tiempo para la ejecución del proyecto, se tomó la decisión de "salir a producción" con un front que pronto cambiará, ¿con cuál de las dos propuestas se quedaría? ¿Por qué? Tenga en cuenta que puede seleccionar alguna de ellas y proponer un cambio o mejora

RTA: La B debido a que se integra mediante el API REST procurando una mejor comunicación que con el uso de un monolito, por ejemplo. Además, esta propuesta al tener las lógicas separadas se pueden desplegar por separando, aumentando la mantenibilidad y escalabilidad. De la misma manera, se podría aplicar un microservicio en vez de un monolito

4. Sabiendo que todos los medios de pagos candidatos a ser soportados en la plataforma permiten una integración mediante API REST y mediante cola de mensajes; ¿Qué estrategia de integración de las anteriormente mencionadas elegiría? ¿Por qué? Justifique adecuadamente su respuesta

RTA: Se podría utilizar una cola de mensajes asíncrona para la facturas de los pagos pero para la efectuar el pago, se puede realizar la misma de manera sincrónica con una API REST (como la de Mercado Pago). Tomamos estas decisiones teniendo en cuenta la experiencia de usuario, en pos de darle la seguridad a ellos que se efectuaron las transacciones.

Persistencia



- Para elemento defensor decidimos guardar la referencia de la clase ya que los tipos de elementos son completamente stateless, por ende no amerita que se persista en la base de datos. Se usa un converter para transformar entre el string que se guarda en la base de datos y el tipo de dato correspondiente en el mundo de objetos.

```

public class ElementoDefensorConverter implements AttributeConverter<ElementoDefensor, String> {
    @Override
    public String convertToDatabaseColumn(ElementoDefensor elemento) {
        String elementoEnBase = null;

        if(elemento.getClass().getName().equals("Arco")) {
            elementoEnBase = "arco";
        }
        else if(medioDeNotificacion.getClass().getName().equals("Espada")) {
            elementoEnBase = "espada";
        }
        else if(elemento.getClass().getName().equals("Escudo")) {
            elementoEnBase = "escudo";
        }
        return elementoEnBase;
    }

    public ElementoDefensor convertToEntityAttribute(String elementoEnBase) {
        ElementoDefensor elementoDefensor = null;
        if(elementoEnBase.equals("arco"))
            elementoDefensor = new Arco();
        else if(elementoEnBase.equals("espada"))
            elementoDefensor = new Espada();
        else if(elementoEnBase.equals("escudo"))
            elementoDefensor = new Escudo();
        return elementoDefensor;
    }
}
  
```

```

    }

    @Override
    public ElementoDefensor convertToEntityAttribute(String s) {
        ElementoDefensor elemento = null;

        if(s.equals("arco")) {
            elemento = new Arco();
        }
        else if(s.equals("espada")) {
            elemento = new Espada();
        }
        else if(s.equals("escudo")) {
            elemento = new Escudo();
        }
        return elemento ;
    }

}

```

Asimismo, del lado de personaje esto será mapeado utilizando la anotación `@ElementCollection` que nos permite mapear una colección de Strings (recordemos que al mapear con referencia a clase, en realidad estamos generando una colección de Strings)

```

public class Personaje {

    ...

    @ElementCollection
    @Convert( converter = ElementoDefensorConverter.class)
    @Column(name = "elementoDefensor")
    private List<ElementoDefensor> elementos;

    ...
}

```

- Para el caso particular del ladrón y del mago, decidimos mappear la herencia con la estrategia **Single Table** porque para recuperar la información no haría falta hacer joins y se podría especificar el tipo de cada personaje mediante la columna discriminadora **Tipo_Personaje**. Sin embargo, en el caso de que se agregaran más personajes con distintos atributos, se recomendaría cambiar la estrategia a un Joined (dado que las clases entre sí tienen elementos en común)