Documento de Decisiones de Diseño

Servicio: Fusionador de Comunidades

Desde nuestro “Sistema de Monitoreo de Estado de Servicios de Transporte Público y de Establecimientos”, utilizaremos el servicio para fusionar comunidades según los criterios impuestos por él. Sin embargo, la utilización de este servicio no se limita a las comunidades, ya que en ningún momento de su utilización hacemos referencia a una. Esto genera un desacoplamiento interesante que podría hacer que nuestro servicio sea más genérico y así llegar a más usuarios. Es por esto que decidimos llamarlo *“Fusionador de Organizaciones”*.

Implementamos este servicio mediante una API REST (Stateless) utilizando Javalin, debido a que Java es el lenguaje que veníamos utilizando y es con el que todos nos sentimos cómodos.

A nivel de utilización por terceros, decidimos usar tres endpoints para separar bien las responsabilidades de cada uno. Uno se utilizará para obtener las propuestas de fusión, otro para aceptar una fusión y otro para rechazarla.
También nos aseguramos de devolver mensajes de error en caso de que haya algún error relacionado con el parseo y/o con la fusión. A esto le dimos una gran importancia debido a que quien esté utilizando nuestro servicio, lo único que podrá saber de nosotros es lo que les llega mediante una respuesta. Para facilitar el proceso de detección de errores a nuestros usuarios, diseñamos un formato específico sobre cómo llegan los mensajes de error si los hubiera.

A nivel objetos, optamos por separar bien las responsabilidades para lograr un diseño modular, obedeciendo el primer principio SOLID. Cada endpoint tiene su propio Handler y maneja su lógica internamente.
El modelado de criterios se hizo con un patrón composite para facilitar la extensibilidad y la modularidad. Cada criterio es una clase, y luego hay un criterio AND, que hace que todos los criterios seleccionados tengan que cumplirse para que no haya un error.
Para obtener las posibles fusiones diseñamos el Relacionador de Organizaciones. Esta clase recibe Organizaciones y devuelve propuestas de fusion entre ellas.
Hubieron cosas que no modelamos por simplicidad, por ejemplo:

- La fusión entre más de dos organizaciones: Sólo consideramos el caso de una fusión binaria (una organización con otra)
- Jerarquización de las fusiones: No se consideró a una fusión mejor que otra, simplemente la fusión propuesta para un par de organizaciones es la primera que se encuentra.

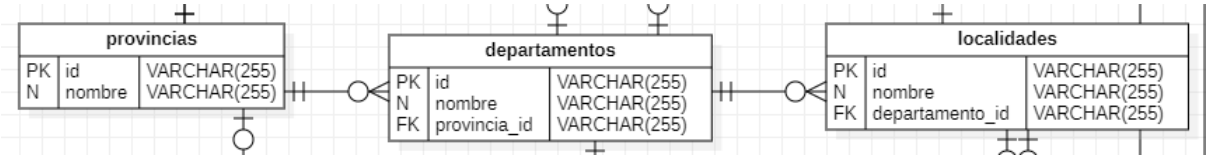
Para fusionar efectivamente las comunidades, diseñamos el Fusionador de Comunidades, que toma dos comunidades e intenta fusionarlas.

Persistencia

Diseño de Datos

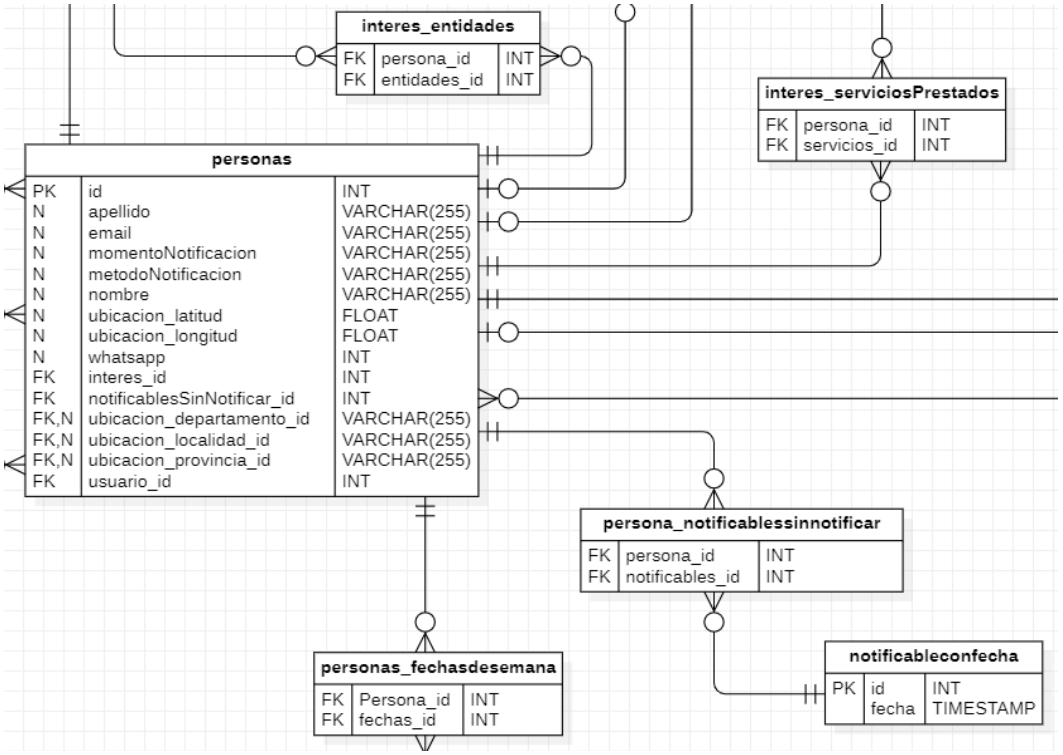
En cuanto a la persistencia de las entidades del Sistema, destacamos algunas decisiones importantes.

La primera de ellas es la forma en que decidimos persistir las **localidades**, los **departamentos** y las **provincias**.



Decidimos persistirlas para no tener que recurrir a la **API GeoRef** siempre que requiramos utilizarlas. Esto es posible ya que la recurrencia con la que podrían cambiar las entidades en el tiempo es baja. En el único momento en el que recurrimos a la API es cuando, dada una latitud y una longitud, necesitamos la localidad, el departamento y la provincia.

Otra decisión relevante fue la de persistir el **Interes** y el **ListadoNotificables** que tiene la Persona, como Embedabbles en la tabla personas. Esto es así ya que las clases Interes y ListadoNotificables tienen atributos de listas ManyToMany y comportamiento específico asociado. Como esas clases solo tienen comportamiento y listas, decidimos que en las tablas intermedias de esos ManyToMany se haga referencia a la persona en lugar del Interes o el ListadoNotificables, para evitar tablas que tengan solamente un id.



Por otro lado, en la **Persona** decidimos persistir el metodoNotificacion como un String (ya que así se utiliza en el Notificador), y la **EstrategiaMomentoNotificacion** usando un Converter que transforma “alMomento” a la clase **NotificacionAlMomento** y “sinApuro” a la clase **NotificacionSinApuro**.
Finalmente, decidimos incluir la última ubicación (latitud, longitud, localidad, departamento, provincia) registrada de cada persona en la tabla personas.

personas		
PK	id	INT
	apellido	VARCHAR(255)
N	email	VARCHAR(255)
	momentoNotificacion	VARCHAR(255)
	metodoNotificacion	VARCHAR(255)
	nombre	VARCHAR(255)
N	ubicacion_latitud	FLOAT
N	ubicacion_longitud	FLOAT
N	whatsapp	INT
FK	interes_id	INT
FK	notificablesSinNotificar_id	INT
FK,N	ubicacion_departamento_id	VARCHAR(255)
FK,N	ubicacion_localidad_id	VARCHAR(255)
FK,N	ubicacion_provincia_id	VARCHAR(255)
FK	usuario_id	INT

Anotaciones

OneToMany

Se utiliza OneToMany en las entidades del dominio que presentan una relación de composición con otras entidades “hijas” o “componente”, por ejemplo:

- **OrganismoControl**: esta entidad sólo tiene sentido al tener bajo su control **EntidadPrestadora**.
- **EntidadPrestadora:** varias instancias son contenidas por un OrganismoControl. No puede haber una EntidadPrestadora que no sea controlada por nadie.

Cascade

En las relaciones **OneToMany** especificadas se agregó además cascade:
`CascadeType.PERSIST, CascadeType.MERGE, CascadeType.REMOVE`
Con el fin de automatizar la generación de sentencias de inserción, actualización y remoción de las entidades hijas asociadas.
Esto nos ahorra tener que escribir el código para la inserción de dichas entidades.

Observación

- Estas relaciones cascadas son útiles en el contexto actual de pruebas unitarias bajo el cual desarrollamos la aplicación y podrían no ser las relaciones finales en un contexto productivo.
- Estas relaciones pueden llegar a generar problemas al trabajar con entidades que poseen grandes colecciones de datos: generando automáticamente

muchísimas sentencias insert / update / delete cuando se modifica una entidad padre.

- Las relaciones cascade también pueden generar situaciones donde se eliminan más registros de los necesarios. Por ejemplo si se aplica un CascadeType.REMOVE en relaciones ManyToMany.