



# Segundo Parcial 2022 - DentalBA

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

[Segundo Parcial Modelo - 2022.pdf](#)

## Dudas

- Es correcta la relación con las fotos de las consultas y las etapas en el Modelo de Entidad Relación? SI
- Uno de nosotros hizo esta resolución del ejercicio 2, está bien?



2.

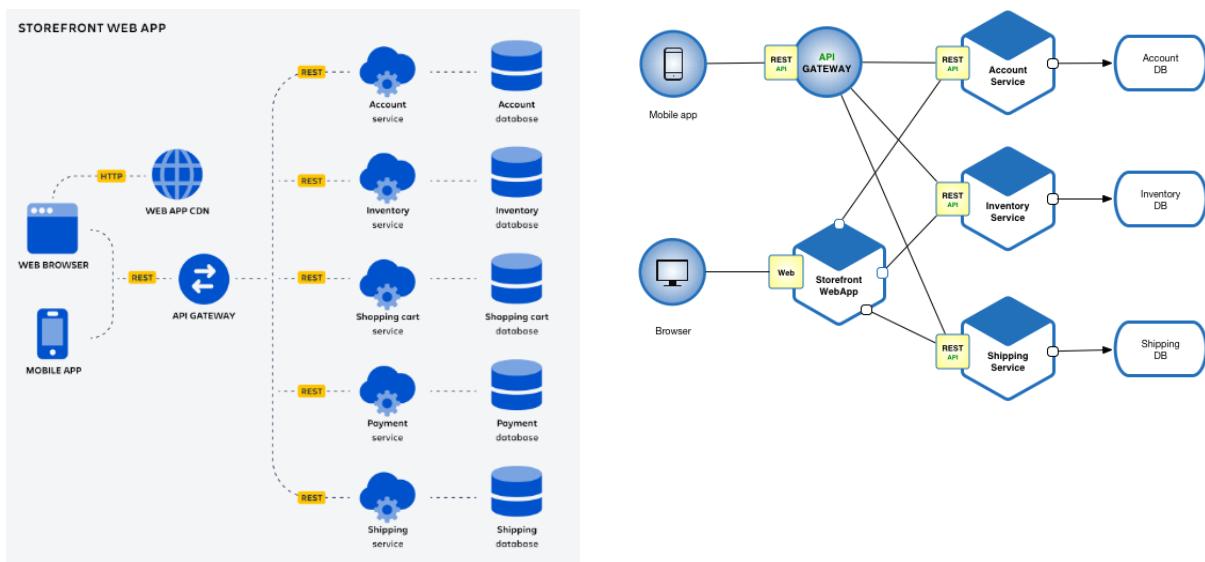
- a. El problema involucra a la securización de las rutas. Probablemente no haya una securización correcta de las rutas, pues los usuarios no cuentan con los roles y/o permisos necesarios para poder trabajar en la plataforma. El atributo de calidad que está involucrado aquí es la seguridad.
- b. La solución al problema es definir un esquema correcto de roles y permisos, para que los usuarios puedan acceder a las rutas que les corresponde.

Mal. El problema es que se está perdiendo la sesión porque, probablemente, haya varios servidores que no están compartiendo sesión y el balanceador me lleva a cualquiera sin importar donde YO tengo la sesión. Posible solución: Sticky session.

- La dinámica con un API Gateway y un balanceador de carga es así:  
 API GATEWAY → BALANCEADOR DE CARGA → SERVICIO?

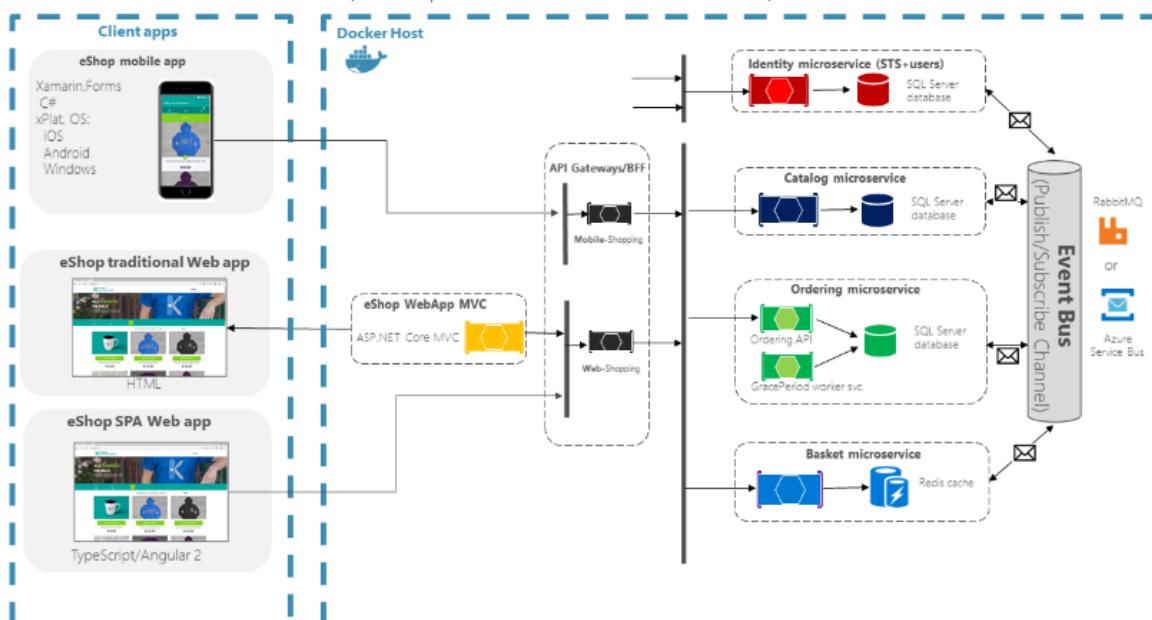
### Notas

- El concepto de SPA está muy relacionado al Cliente Pesado pero no necesariamente todo cliente Pesado lo es al 100%, ya que podrías tomar la decisión de delegar cierta parte de tu lógica de presentación (UI) a otro cliente pesado, lo que llevaría a una recarga de la página. Pero en general sí, son conceptos fuertemente relacionados
- Definición **API Gateway**: “Los **API Gateway** son un componente que ayuda a encaminar las solicitudes de API, agrega respuestas de API y aplica acuerdos de nivel de servicio mediante funciones. Los gateways conducen a APIs y servicios de back-end que define la empresa y los presenta en una capa regulable mediante una solución de gestión de APIs.”



## eShopOnContainers reference application

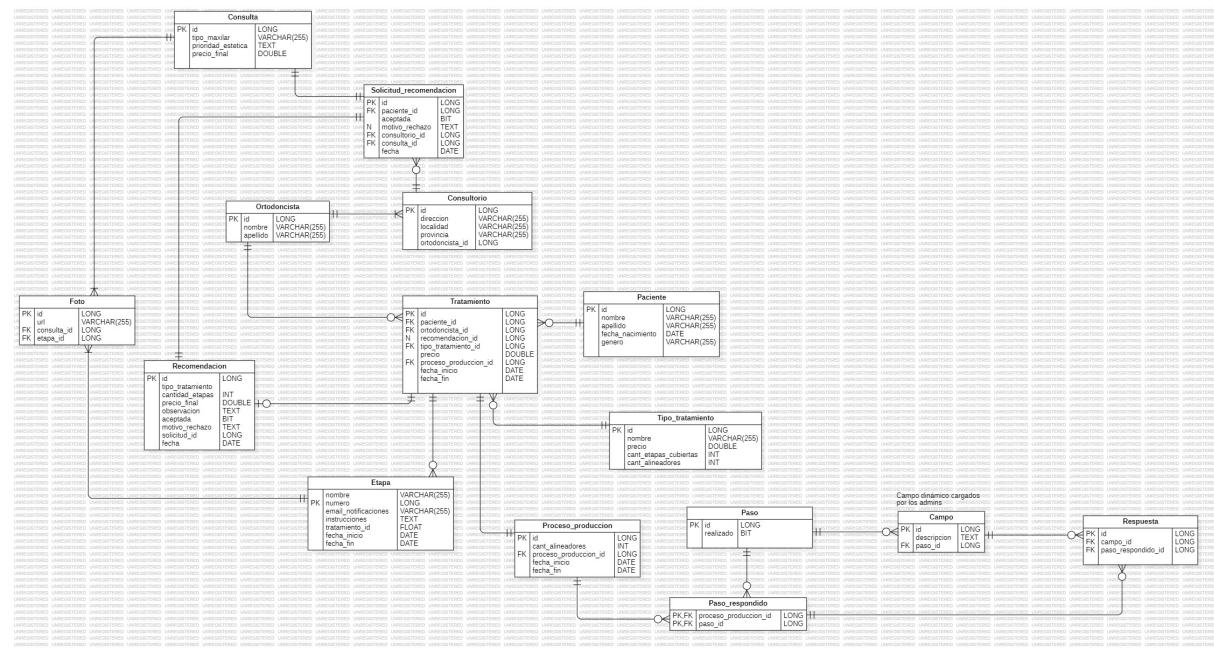
(Development environment architecture)



## Modelado de datos

### Entidades principales

- **Solicitud\_recomendación:** cuenta con un paciente\_id en donde luego en la capa de presentación se va a poder mostrar los datos del paciente como (nombre, apellido, fecha nacimiento, género), una consulta\_id para poder relacionarlo con los datos de la misma (fotos de la cara y dentadura desde distintos ángulos (obligatorias), maxilares a tratar (superior, inferior o ambos), prioridad estética (texto libre) y consulta profesional (texto libre))
- **Consulta:** cuenta con los atributos mencionados anteriormente
- **Paciente:** posee un nombre, apellido, fecha de nacimiento y género
- **Consultorio:** que se relaciona con una tabla ubicación
- **Ortodoncista**
- **Tratamiento**
- **Tipo tratamiento**
- **Recomendación**
- **Proceso productivo**
- **Etapa**
- **Paso**



## Decisiones de diseño

- Consideramos que un ortodoncista es dueño de varios consultorios pero que un consultorio tiene un solo dueño.
- El tipo maxilar que se encuentra en la tabla consulta es un enum (SUPERIOR, INFERIOR, AMBOS) que se utiliza un converter para poder persistirlo en nuestra base de datos.
- La obligatoriedad de las fotos de las consulta la determina la capa de presentación es por ello que en esta tabla marcamos una relación de 1 a muchos para el path de las fotos.
- En la solicitud\_recomendación, añadimos un atributo llamada "aceptada" que es un boolean que permite la trazabilidad y auditoría de las mismas.
- Al poder ser iniciado un tratamiento luego de una solicitud o de manera independiente, lo representamos con la modalidad de la relación entre tratamiento y recomendación.
- Como existen distintos tipos de tratamiento, optamos por realizar una tabla tipo\_tratamiento donde se puede especificar las etapas que cubren y el presupuesto. A nivel objetos, esto se podría ver representado como una instancia de la clase TipoTratamiento ya que hacer una clase abstracta no sería lo recomendable ya que no tienen comportamientos distintos.
- Se realizó una desnormalización por consistencia de datos en la tabla tratamiento al sumar el atributo "precio" ya que, de esta manera, se permite fijar el precio del mismo en caso de que el presupuesto del tipo de tratamiento se actualice y así cumplir con el requerimiento de trazabilidad.

- Consideramos que radiografía panorámica en nuestro sistema es una URL a la imagen de la radiografía.
- Realizamos una relación de la tabla paso con una tabla observación para poder representar los campos variables que se mencionan en el enunciado.
- Por desnormalización de datos, en la tabla de proceso de producción tiene la cantidad de alineadores para cumplir con el requerimiento de auditoría.
- Existe una relación bidireccional a nivel datos entre Tratamiento y Proceso de producción ya que consideramos importante que cada una de las tablas tengan el id de la otra por consistencia de datos.
- Decidimos embeber la clase de Ubicación con la de Consultorio de la siguiente manera:

```

@Embeddable
public class Ubicacion{
    ...
}

@Entity
@Table(name="consultorio")
public class Consultorio{
    ...
    @Embedded
    private Ubicacion ubicacion;
}

```

## Arquitectura

1. Sabiendo que el aplicativo tendrá una frecuencia de uso alta por parte de los ortodoncistas y por parte de los profesionales de DentalBA; teniendo en cuenta que el Sistema debe ser desplegado en una infraestructura propia y considerando que se prevé realizar mejoras notables a nivel visual (o inclusive cambiar por completo la interfaz según necesidad):

a. Compare las siguientes alternativas respecto al diseño arquitectónico de la capa de presentación del Sistema en base a, al menos, dos atributos de calidad que considere que apliquen. Debe quedar en claro qué aspecto/s está considerando para la comparación.

b. ¿Qué alternativa escogería? ¿Por qué? Justifique.

Alternativas

1. Cliente liviano (Server Side Render)
2. Cliente pesado (Client Side Render), con un backend que expone una API REST para la manipulación de los recursos.
3. Cliente liviano (Server Side Render) desacoplado del backend, el cual expone una API REST para permitir la manipulación de los recursos.

**RTA:**

a.

|                | <b>Cliente liviano</b>   | <b>Cliente pesado, con un backend que expone una API REST</b>   | <b>Cliente liviano desacoplado del backend que expone API REST</b>   |
|----------------|--|---|--|
| Performance    | <ul style="list-style-type: none"> <li>- Hay un menor uso de memoria del lado del cliente pero mayor del lado del servidor. Al existir una alta concurrencia de usuarios en la plataforma, tenemos una carga importante del lado de los servidores, ya que se generan espacios de memoria y recuperación de los mismos por los accesos. - No hay tiempo de respuesta del lado del servidor, ya que no se integra por API REST</li> </ul> | <ul style="list-style-type: none"> <li>- Mejora la performance del lado del servidor, al no implementar una sesión, disminuimos la carga sobre los mismos ya que se puede evitar guardar las sesiones usando tokens. - Del lado del cliente se disminuye la performance ya que es este el que renderiza las vistas. Sin embargo, el cliente gozará de una mejor experiencia de usuario dado que no existirán</li> </ul> | <ul style="list-style-type: none"> <li>- Es la opción 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.</li> </ul>   |
| Mantenibilidad | <ul style="list-style-type: none"> <li>- Para este caso, como se prevé realizar cambios parciales o totales en la interfaz de usuario, se podrían causar conflictos debido a que se debería deployar todo el componente nuevamente generando un impacto en el resto de los componentes.</li> </ul>   | <ul style="list-style-type: none"> <li>- Es más mantenible ya que la lógica de la interfaz de usuario queda del lado del cliente. Por consiguiente, si se realizan cambios notables en alguna de las capas, como se prevé en este caso, puede realizarse de manera más directa y segura, sin impactar tanto en el componente del servidor backend, y sin necesidad de deployar todo el aplicativo</li> </ul>            | <ul style="list-style-type: none"> <li>- Es más mantenible que la primera alternativa pero menos mantenible que el segundo. Debido a que sigue existiendo un problema con el deploy como en el caso de la opción 1 pero que a su vez permite realizarse de manera que impacte menos en el servidor back</li> </ul> |

|  |                        |   |  |
|--|------------------------|---|--|
|  | <b>Cliente liviano</b> | <b>Cliente pesado, con un backend que expone una API REST</b> | <b>Cliente liviano desacoplado del backend que expone API REST</b> |
|  |                        | nuevamente, sino los componentes necesarios                   |  |

|                | <b>Cliente liviano</b>   | <b>Cliente pesado, con un backend que expone una API REST</b>   | <b>Cliente liviano desacoplado del backend que expone API REST</b>  |
|----------------|--|---|---|
| Performance    | La carga inicial de la página es manejada por el servidor, lo que puede resultar en tiempos de carga más rápidos para los clientes. Sin embargo, las interacciones adicionales después de la carga inicial requieren comunicación constante con el servidor, lo que podría aumentar la latencia. | Una vez cargada la aplicación, las interacciones pueden ser más rápidas ya que se realizan en el lado del cliente sin la necesidad de recargar la página completa. La carga inicial puede ser más lenta, ya que el cliente necesita descargar el código y los recursos antes de renderizar la interfaz. | Permite una carga inicial rápida al renderizar en el servidor, mientras que las interacciones posteriores pueden manejarse en el lado del cliente para una experiencia más fluida. Puede haber una complejidad adicional al gestionar la lógica de presentación en el servidor. |
| Mantenibilidad | Cambios significativos en la interfaz pueden requerir actualizaciones en el servidor y en el cliente.  | La interfaz del cliente puede evolucionar de forma independiente al backend, permitiendo actualizaciones más rápidas y flexibles.   | La interfaz del cliente puede actualizarse de manera más independiente, pero aún hay cierta dependencia en la comunicación con el backend.  |

- **Performance:** representa el desempeño relacionado a la cantidad de recursos utilizados bajo condiciones determinadas. En este caso en particular vamos a analizar el criterio de comportamiento temporal, que establece los tiempos de respuesta y procesamiento cuando se lleva a cabo sus funciones bajo condiciones determinadas.
- **Mantenibilidad:** capacidad que representa un producto SW para ser modificado de manera efectiva y eficiente con el fin de satisfacer las necesidades evolutivas, correctivas o perfectivas. En este caso en particular vamos a analizar la capacidad de ser modificado

b. La alternativa escogida es la 2 “**Cliente pesado, con un backend que expone una API REST**” debido a que según nuestros criterios evaluados, dado el contexto de la consigna, conviene tener desacoplada la lógica del front y la del back. De la misma manera, el cliente pesado es más escalable que el liviano y más seguro, al utilizar tokens que permita que el inicio de sesión sea securizado.

2. Suponiendo y considerando que el Sistema cuenta con una arquitectura web Stateful, con varios servidores atendiendo solicitudes de forma simultánea, lea el siguiente escenario y conteste las preguntas: “Varios usuarios han reportado que, mientras utilizan la plataforma (previamente logueados), el Sistema los vuelve a llevar a la pantalla de Inicio de Sesión, teniendo que volver a ingresar sus credenciales para continuar con su labor.”
- ¿Cuál cree que es el problema involucrado? Además de mencionarlo y detallarlo, indique qué atributo de calidad, calidad de diseño o concepto general arquitectónico, se está viendo afectado.
  - Proponga una solución al problema presentado.

**RTA:**

a. Al mencionar que los usuarios deben ingresar sesión para continuar con sus labores, se podría considerar que es un error de balanceador de cargas. Debido a que se podría estar implementando un algoritmo incorrecto, como es el ejemplo del Round Robin que te manda a un servidor distinto ante cada solicitud. Aquí se ven afectados los siguientes atributos de calidad:

- **Usabilidad:** los usuarios no son capaces de usar la app dado que la problemática no les permite acceder a las distintas rutas.
- **Accesibilidad:** No logra ser capaz de que personas con distintas características y discapacidades puedan utilizar la página ya que la constante necesidad de iniciar sesión puede dificultarle la comprensión
- **Escalabilidad:** esta problemática dificulta la posibilidad de escalar, dado que, si agregamos más servidores (en un futuro), por cada cliente que realice una **request** será dirigido a un servidor distinto.

De la misma manera, algunos de los criterios de calidad de diseño involucrados son:

- **Extensibilidad y Mantenibilidad:** Por cada solicitud te mandan a un servidor distinto causando dificultades para modificar o extender el aplicativo.

b. Como solución a este problema proponemos la implementación de un algoritmo como **Sticky Session**, de manera que cada usuario tenga su sesión en un servidor específico y que cada vez que haga log in, se lo redirija a ese mismo servidor.

3. Teniendo en cuenta que se requiere enviar notificaciones por email a los ortodoncistas frente a cada cambio que afecten a sus tratamientos/etapas y que el envío de emails puede demorar varios segundos/minutos; ¿Cómo resolvería el envío de emails para que no se vea afectado el tiempo de respuesta a las requests de los clientes? Detalle su solución dejando en claro cuestiones de sincronismo/asincronismo e integración (de ser necesario).

**RTA:** Para que no se vea afectado el tiempo de respuesta a las request de los clientes, implementaría una cola de mensajería, en donde se guardaría la notificación que se debe enviar, y luego un consumidor toma eso y envia la notificación a medida que puede. Es importante abordar el envío de mails desde un punto de vista asincrónico, ya que si un mail tarda mucho en enviarse, puede afectar al funcionamiento del sistema. En este sentido, la cola de mensajería nos brinda la posibilidad de trabajar esto de forma asíncrona.

4. Sabiendo que no todos los módulos del Sistema se extienden en funcionalidad con la misma frecuencia; que ciertos módulos poseen mayor concurrencia que otros (por ejemplo, el módulo de “Proceso de producción de alineadores” es el más utilizado); y que podrían agregarse nuevos módulos: ¿cree que una arquitectura de Microservicios podría aplicar para este caso? ¿Por qué? Justifique adecuadamente su respuesta.

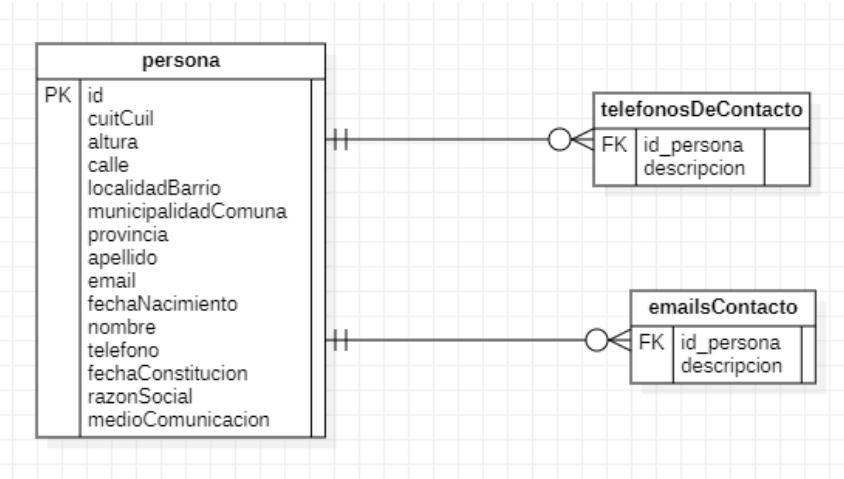
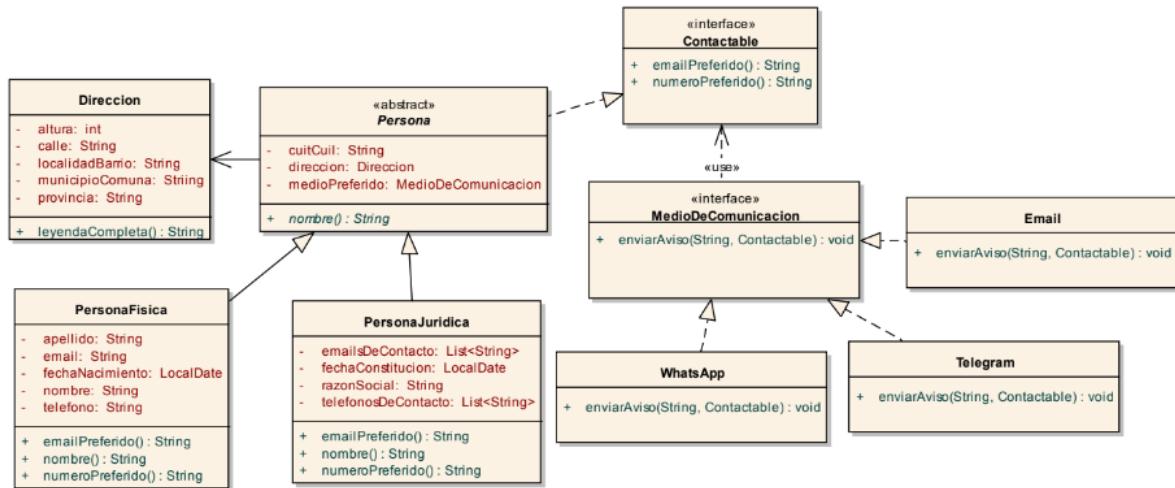
**RTA:**

Si, en este caso considero que una arquitectura de microservicios resulta adecuada por los siguientes motivos:

- Si no todos los módulos del sistema se extienden en funcionalidad con la misma frecuencia, entonces es conveniente microservicios ya que permite trabajar sobre esos módulos en particular y extender los mismos, y solo se hará una compilación y deployment de estos módulos en particular, es decir, no será necesario volver a deployar toda la aplicación.
- Si hay ciertos módulos que poseen mayor concurrencia que otros, se pueden tomar acciones específicas para esos módulos (por ejemplo, agregar una caché a ese módulo en particular, escalar verticalmente).
- Si se pueden agregar nuevos módulos, resulta conveniente una arquitectura de microservicios por los mismos motivos que mencioné en al punto 1, es decir, tenemos la

posibilidad de escalar fácilmente la aplicación sin afectar al sistema entero, y no habrá necesidad de compilar y deployar la app de nuevo.

## Persistencia



## Elementos necesarios de persistir

- **Persona**
- **PersonaFisica**
- **PersonaJuridica**
  - Se usa un `@ElementCollection` para los atributos `emailsDeContacto` y `telefonoDeContacto`

Nota general: Estas son las conversiones que realizaríamos para persistir los datos:

| Tipo de dato objetos | Tipo de dato relacional |
|----------------------|-------------------------|
| LocalDate            | SMALLDATETIME           |
| String               | VARCHAR(255)            |
| Int                  | INTEGER(11)             |

## Decisiones

Por una parte, para persistir las personas decidimos usar la estrategia de mapeo de herencias llamada Single Table. Pondremos en la misma, todos los atributos de Persona Física y Jurídica en la misma teniendo de esta manera algunos atributos null de acuerdo a la clase mapeada.

Esta decisión la tomamos ya que consideramos que el agregado de nuevos tipos de personas no será muy frecuente. Asimismo, las consultas resultarán más performantes que si se usara otra estrategia (como el Join) dado que se consultará una sola tabla, sin necesidad de hacer distintas

De la misma manera, para el atributo de medio de comunicación, decidimos hacer un converter para determinar en la base de datos el tipo de interfaz asociada y de esta manera realizar una referencia a clase:

```
public class MedioComunicacionConverter implements AttributeConverter<MedioComunicacion, String> {

    @Override
    public String convertToDatabaseColumn(MedioComunicacion medio) {
        String medioEnBase = null;

        if(medio.getClass().getName().equals("Email")) {
            medioEnBase = "Email";
        }
        else if(medio.getClass().getName().equals("Telegram")) {
            medioEnBase = "Telegram";
        }
        else if(medio.getClass().getName().equals("Whatsapp")) {
            medioEnBase = "Whatsapp";
        }
        return medioEnBase;
    }

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

        if(s.equals("Email")) {
            elemento = new Email();
        }
        else if(s.equals("Whatsapp")) {
            elemento = new Whatsapp();
        }
        else if(s.equals("Telegram")) {
            elemento = new Telegram();
        }
    }
}
```

```

        }
        return elemento ;
    }

}

```

Por otra parte, decidimos no persistir la tabla Dirección ya que es un atributo de la tabla persona, que es únicamente informativo. No tiene sentido de existencia propio (depende de la existencia de Persona), es por ello que decidimos embeberlo en la entidad persona, agregándolo como una columna:

```

@Embeddable
public class Direccion {
    ...
}

@Entity
@Table(name="persona")
public class Persona {
    ...
    @Embedded
    private Direccion direccion;

}

@Entity
@DiscriminatorValue("PersonaJuridica")
public class PersonaJuridica extends Persona{
    @ElementCollection
    @CollectionTable(name="emailsContacto", joinColumns= @JoinColumn(name="id_persona"))
    @Column(name = "descripcion")
    List<String> emailsDeContacto;

}

```

A su vez, decidimos no persistir la interfaz Contactable y Medio de Comunicación ya que como todas las clases que implementan esta interfaz son stateless, no resulta necesario persistir, pues no tenemos ningún estado que guardar.