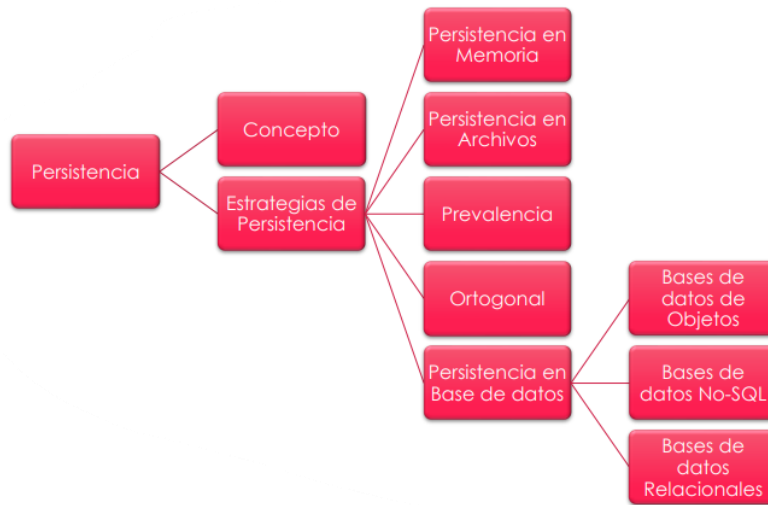


Persistencia de Datos

Persistencia en medios relacionales mediante ORM



Persistencia

Cuando hablamos de PERSISTENCIA en términos informáticos, nos estamos refiriendo a que el estado de un sistema sobrevive más allá del proceso que lo generó, esto quiere decir que el estado de mi sistema se almacena o se guarda en algún entorno persistente para que, cuando volvamos a ejecutar el proceso, el estado sobreviva.

Este medio persistente puede ser una base de datos o archivos.

No todos los sistemas tienen datos persistentes, algunos pueden ser solo comportamiento.

Estrategias de Persistencia

Existen varias estrategias de persistencia de datos en un aplicativo:

- Persistencia en Memoria
- Persistencia en Archivos
- Prevalencia
- Ortogonal
- Persistencia en Base de Datos
 - Bases de datos de Objetos
 - Bases de datos No-SQL
 - Bases de datos Relacionales

Persistencia en Memoria (Memoria RAM)

La PERSISTENCIA EN MEMORIA es la capacidad de un dato u objeto de seguir existiendo luego de ser utilizado en distintas operaciones.

Si bien la memoria RAM es un medio volátil (es decir que, cuando se corta la energía, los datos se borran), lo que contradice con el concepto de persistencia puro, existen memorias persistentes.

La persistencia en memoria es utilizada mayormente para realizar test en los sistemas. Sin embargo, debemos recordar que al estar trabajando en memoria RAM, ganamos velocidad, por ende, muchos sistemas en algunas partes utilizan la persistencia en memoria para cachear datos que necesitan tener rápidamente.

También existen implementaciones de Bases de Datos que persisten sus datos en memoria, como SQLite.

Persistencia en Archivos

La PERSISTENCIA EN ARCHIVOS involucra guardar el estado de un sistema en uno o varios archivos para que luego el estado pueda ser recuperado y el sistema continúe funcionando/ejecutando desde el mismo punto, es decir que se ejecuta “como si nada hubiera pasado”.

Existen varios tipos de archivos o formatos que se puede utilizar para persistir o transmitir datos, tales como:

- XML
- CSV
- Ancho Fijo
- JSON
- Otros

Prevalencia

La PREVALENCIA es una técnica que almacena el Estado de un Sistema en la Memoria Principal, pero regularmente genera “snapshots” a disco para evitar la pérdida completa de datos.

- Los snapshots generan un archivo con los datos y luego se transfieren al disco para no tener una pérdida completa de los datos.
- Se combina con la técnica de persistencia en archivos.

Las implementaciones de esta técnica tienen la capacidad de manejar transacciones.

Una de las principales ventajas de la prevalencia es que no existen transformaciones de datos, ya que éstos son persistidos en el formato que el aplicativo utiliza o “entiende”. Como consecuencia, se obtiene un alto grado de transparencia y una mejora notable en la performance.

Como desventaja existe una falta de interoperabilidad de los datos, ya que otros sistemas no podrán “acceder” a nosotros si no trabajan con nuestros mismos recursos y formatos, además de que debe considerarse una alta capacidad de Memoria ya que la misma debe poder alojar al aplicativo completo.

Ortogonal

La persistencia ORTOGONAL hace referencia a que la Persistencia del Estado de un Sistema se implementa como una propiedad intrínseca de su entorno de ejecución.

- Que sea intrínseco quiere decir que nosotros no tenemos que estar dando órdenes de que se ejecuten ciertas acciones (guardar, ejecutar, borrar, etc.) sino que el mismo entorno lo realiza de forma automática sin que nos demos cuenta.

Por este motivo, no se requieren acciones específicas para poder guardar o recuperar datos, ya que esto ocurre de forma “transparente”.

Este tipo de persistencia es adoptado por los Sistemas Operativos, permitiendo funcionalidades como la Hibernación, y en Sistemas de Virtualización de Plataformas como VMware o VirtualBox.

Persistencia de Base de Datos

En la PERSISTENCIA DE BASE DE DATOS el estado de un Sistema es persistido en una o varias bases de datos.

Los datos persistentes del aplicativo necesitan una transformación antes de ser persistidos, ya que puede que existan conflictos en los tipos de datos que hayamos usado y los que admite la base de datos.

La interoperabilidad es un atributo de calidad que se maximiza en este caso, ya que los datos se persisten de forma independiente a los sistemas que los manipulan.

Las bases de datos otorgan mecanismos para recuperarse en caso de fallos, además de brindar reglas para resguardar la integridad de los datos.

Persistencia de Base de Datos Orientadas a Objetos

Cada una de las Bases de Datos Orientadas a Objetos está atada a un lenguaje de programación en particular, ya que en cada lenguaje los objetos tienen una representación interna diferente.

Como ventaja se destaca la no transformación de los datos, esto proporciona una mejor performance. Sin embargo, la desventaja más grande es la interoperabilidad de los datos.

- No voy a poder compartir mis datos con otros sistemas, ya que este tipo de bases están acopladas al lenguaje que estoy usando.

Modelo Relacional



El MODELO RELACIONAL fue postulado por Edgar Frank Codd (IBM) en 1970.

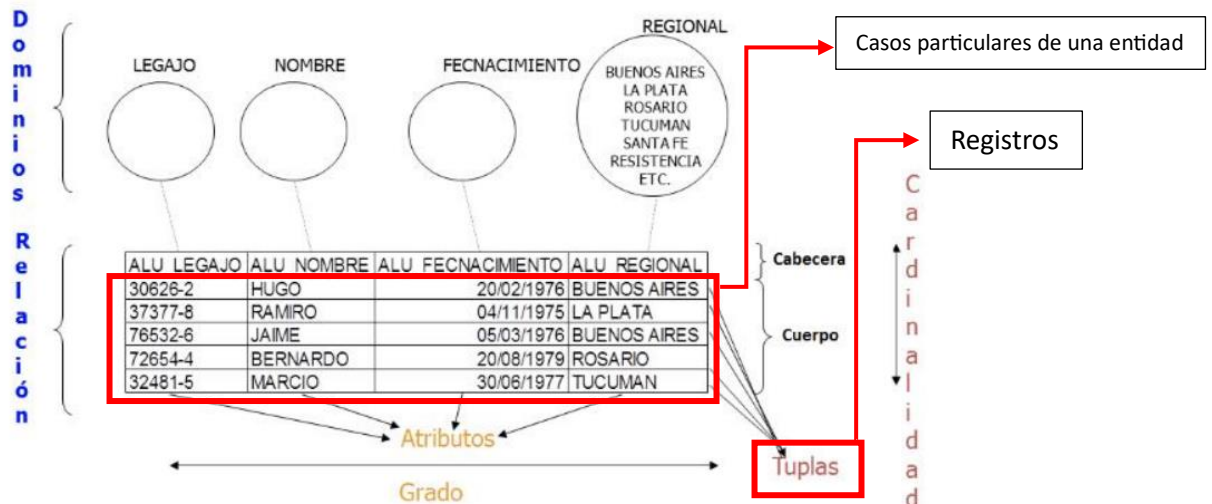
Este modelo surgió como una nueva forma de organizar los datos. Todos los datos son almacenados en RELACIONES.

Cada una de estas relaciones es un conjunto de datos organizados, llamados TUPLAS.

El modelo relacional facilita la organización de grandes volúmenes de datos y garantiza la INTEGRIDAD de estos.

PALABRAS CLAVE

- **Relación:** es la “entidad” donde se almacenan los datos.
- **Tupla:** es el registro.
- **Atributo:** es un CAMPO de la Tupla.
- **Dominio:** es un conjunto de valores posibles que puede tomar un Atributo. Es la menor unidad semántica de información.
 - En general, el Dominio hace referencia a todos los valores posibles que puede tomar un atributo
- **Cardinalidad:** número de tuplas. → La cantidad de filas que tiene una tabla.
- **Grado:** número de Atributos. → La cantidad de columnas que tiene una tabla.
- **Clave Primaria (PK):** es el identificador único de cada tupla.



Base de Datos Relacional

La BASE DE DATOS RELACIONAL es un tipo de base de datos que cumple con el Modelo Relacional.

IDAÑEZ, LUCIA MARÍA 🐱

En una BDR, todos los datos se almacenan y se accede a ellos por medio de relaciones que fueron establecidas. Estas relaciones almacenan datos y son llamadas RELACIONES BASE y su implementación es llamada TABLA.

Como mencionamos, en una BDR los datos están organizados en relaciones llamadas Tablas, donde cada tupla es representada por una FILA y que, a su vez, contiene múltiples COLUMNAS (Atributos), y donde cada celda puede tomar alguno de los valores definidos en su dominio.

CLAVE PRIMARIA

La CLAVE PRIMARIA es el Atributo que identifica de manera unívoca a cada fila de la tabla.

Existen tres tipos de claves primarias: Naturales, Subrogadas o Compuestas.

ÍNDICES

El ÍNDICE es una estructura de datos que mejora la velocidad de las operaciones, por medio de un identificador único de cada fila de una tabla. Permite un rápido acceso a los registros de una tabla en una base de datos.

CLAVE FORÁNEA

La CLAVE FORÁNEA es un grupo de una o más columnas en una tabla que referencian a la clave primaria de otra tabla. Una Foreign Key puede ser parte de una Primary Key.

Tanto las PK como las FK son índices

INTEGRIDAD REFERENCIAL

La INTEGRIDAD REFERENCIAL es una propiedad que establece que la clave externa de una tabla de referencia siempre debe corresponder a una fila válida de la tabla a la que se haga referencia. La integridad referencial garantiza que la relación entre dos tablas permanezca sincronizada durante las operaciones de actualización y eliminación.

TIPOS DE DATOS

- VARCHAR: Cadenas de caracteres compuestas por letras, números y caracteres especiales.
- INTEGER: Números enteros naturales.
- DOUBLE: Valores numéricos con punto flotante.
- DATETIME: Formato para manejo de fechas.

CONSTRAINTS

Las CONSTRAINTS son restricciones al modelo que se utilizan para limitar el tipo de dato que puede ingresarse en una tabla. Se especifican cuando la tabla se crea por primera vez, o realizando una actualización después.

Las constraints más comunes son:

- Not Null: no acepta valores nulos
- Unique: no acepta valores repetidos
- Check (o enum): verifica que los valores cumplan determinada condición
- Primary Key: clave primaria
- Foreign Key: clave foránea

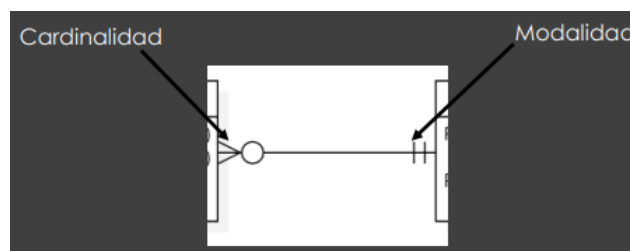
RELACIONES

Las RELACIONES son asociaciones entre tablas que se describen en la estructura de la base de datos.

Las relaciones describen cómo se vinculan una o más tablas entre sí.

CARDINALIDAD Y MODALIDAD

- La CARDINALIDAD es la cantidad máxima de relaciones que tiene un registro de una entidad (tabla) con respecto a la otra tabla.
- La MODALIDAD es la cantidad mínima de relaciones que tiene un registro de una entidad (tabla) con respecto a la otra tabla.

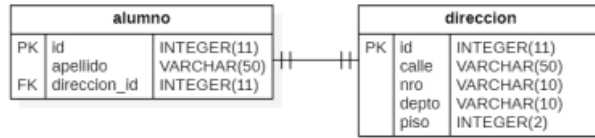


RELACIÓN ONE TO ONE

- Es una relación entre dos entidades (tablas) A y B, en la cual un registro de la entidad A se corresponde con un único registro de la entidad B.
- La relación puede ser unidireccional, en cualquier sentido, o bidireccional

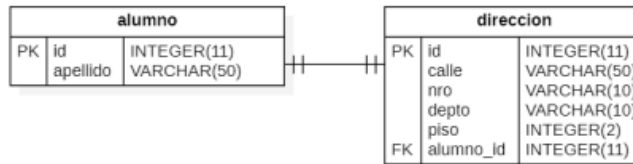
Unidireccional desde el lado A: existe un registro en la tabla B que está siendo referenciado por un único registro de la tabla A.

- En este caso, UN Alumno tiene la referencia a UNA Dirección, por ende, la relación es unidireccional desde el lado del A.



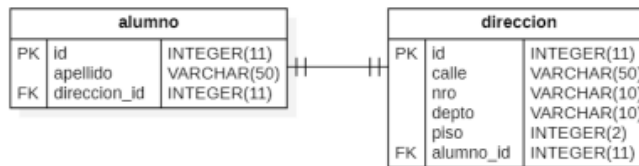
Unidireccional desde el lado B: existe un registro en la tabla A que está siendo referenciado por un único registro de la tabla B.

- En este caso, la Dirección hace referencia a un único registro en la tabla Alumno, por ende, la relación es unidireccional desde el lado de la Dirección.



Bidireccional: existe un registro en la tabla A que está siendo referenciado por un único registro en la tabla B; y a su vez, este registro de la tabla B está siendo referenciado por el registro de la tabla A.

- En este caso, tanto la tabla de Alumno como la tabla Dirección tienen una referencia el uno con el otro, por ende, es una relación bidireccional.



RELACION ONE TO MANY

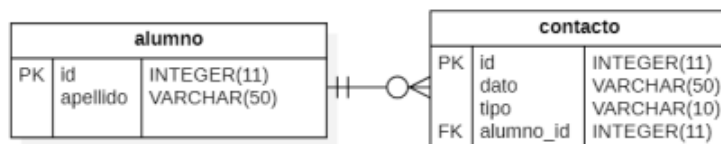
- Es una relación entre dos entidades (tablas) A y B en la cual un registro de la entidad A está siendo referenciado por muchos registros de la entidad B.
- Significa que “A tiene muchos B”, porque cada registro de la tabla B pertenece a un registro de la tabla A

RELACIÓN MANY TO ONE

- Es una relación entre dos entidades (tablas) A y B en la cual muchos registros de la entidad A están referenciando al mismo registro de la entidad B.
- Significa que “A pertenece a un B, y B tiene muchos A”, porque cada registro de la tabla A esta siendo referenciado por uno o muchos registros de la tabla B.
- También se puede leer como “Muchos A pertenecen al mismo B”

Veamos el siguiente ejemplo:

“Un alumno puede tener muchos contactos”



En este caso podemos ver que todos los contactos referencian/pertenece a un mismo alumno (Many to One) y un solo alumno referencia/tiene muchos contactos (One to Many).

MANY TO ONE VS. ONE TO MANY

Si se tienen en cuenta dos entidades A y B podríamos afirmar que:

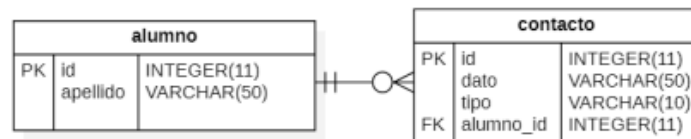
- Si A tiene una relación One to Many con la entidad B, entonces B tiene una relación Many to One contra la entidad A.
- Si A tiene una relación Many to One contra la entidad B, entonces B tiene una relación One to Many contra la entidad A.

La relación entre las entidades A y B, en este caso, pueden llamarse de una u otra forma dependiendo el “lado” que miremos. En otras palabras, es una cuestión de perspectiva.

REGLA PRÁCTICA: si existe una relación One to Many, visto desde A hacia B, entonces “del otro lado” existe una relación Many to One (visto desde B hacia A). La inversa también es válida.

Veamos el siguiente ejemplo:

- Un contacto pertenece a un alumno, y un alumno puede tener muchos contactos
- Muchos contactos pueden pertenecer al mismo alumno



MANY TO MANY

- La relación MANY TO MANY es una relación entre dos entidades (tablas) A y B en la cual un registro de la entidad A puede estar siendo apuntado por muchos registros de la entidad B, y un registro de la entidad B puede estar siendo apuntado por muchos registros de la entidad A.
- Significa que **“A tiene muchos B y B tiene muchos A”**
- Esta relación no es posible realizarla en el modelo relacional, ya que sería tener múltiples FKs en ambas tablas involucradas en la relación.

Para poder implementar esta relación debemos partir la relación en dos relaciones One To Many. Por un lado, la entidad A debe tener una relación ONE TO MANY contra una entidad C, considerada ENTIDAD INTERMEDIA; lo mismo sucede con la entidad B, es decir que la entidad B debe tener una relación ONE TO MANY contra la ENTIDAD INTERMEDIA C.

La entidad C se llama TABLA INTERMEDIA y debe poseer una FK a la entidad A y otra FK a la entidad B. Además, la entidad C podría guardar algún dato extra en la relación.

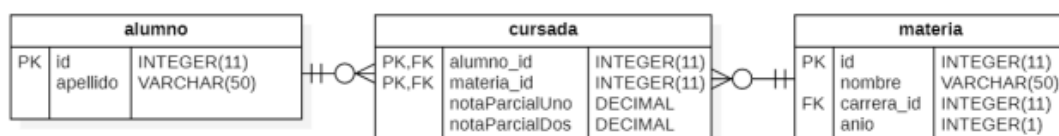
Veamos el siguiente ejemplo:

- En este caso, un alumno puede estar cursando varias materias, y una materia puede ser cursada por muchos alumnos.



- En este caso, suponemos que no nos interesa guardar ningún dato extra sobre las materias que está cursando un alumno.
- Si necesitamos, por ejemplo, las notas del primer y segundo parcial, el diseño cambia y “alumno_materia” deja de ser una tabla intermedia normal.

Si el dominio lo amerita y necesitamos guardar algún dato extra, el diseño se podría mejorar.



Estrategias de Persistencia

PERSISTENCIA EN BASES DE DATOS RELACIONALES

El estado de un Sistema puede ser persistido en una o varias bases de datos relacionales.

La persistencia de un Sistema que está pensado y escrito bajo el Paradigma Orientado a Objetos no es directa.

Mapeo Objeto-Relacional

El Mapeo Objeto-Relacional es una técnica usada para convertir los tipos de datos con los que se trabaja en un lenguaje orientado a objetos a tipos de dato con los que se trabaja en base de datos relacional.

ORM es la técnica que transforma los datos del mundo de objetos para que encajen en el mundo relacional.

También se conoce como ORMs a los frameworks que implementan esta técnica.

La técnica ORM nos permite desacoplar nuestro sistema de la base de datos relacional. Estas herramientas introducen una capa de abstracción entre la base de datos y el desarrollador, lo que evita que el desarrollador escriba consultas para recuperar, insertar, actualizar o eliminar registros de la base de datos, ya que el encargado de hacer todo esto será el ORM.

Al tratar de encontrar esta correspondencia entre el mundo de objetos y el mundo de datos, hay cosas que no se puede imitar, a estas cosas las llamamos IMPEDANCE MISMATCH.

Impedance Mismatch

Identidad

- Un objeto cumple con las características de identidad y unicidad, ya que el objeto reside en un espacio de memoria y ningún otro puede residir en ese mismo espacio de memoria.
- Un registro, en una tabla de una base de datos relacional, necesita una identificación unívoca a través de un PK (Primary Key).

Las claves posibles para una entidad en una BDR son:

- Clave natural: algún campo propio de la entidad que lo identifique de forma unívoca.
- Clave subrogada: clave ficticia (al azar, generada automáticamente, etc.)

Conversión de Tipos de Datos

Los tipos de datos de ambos mundos no tienen una correlación directa.

Algunas conversiones con las que vamos a trabajar son:

- String: en el mundo de objetos no tenemos limitaciones de tamaño mientras que en las BDR tenemos los VARCHAR/CHAR que requieren definir una longitud.
- Campos numéricos: en las BDR se requiere una precisión de decimales o el rango de valores máximos y mínimos.
- Fechas: las fechas tienen tipos no siempre compatibles entre sí, por ejemplo:
 - LocalDate de Java vs. Date-Datetime-Time-tinyblob del motor
- Booleanos
- Enumerados: son como los enums de Java

Manejo de la Cardinalidad

En el POO, las relaciones pueden ser unidireccionales y bidireccionales, es decir que tenemos relación de una a otra. Sin embargo, en el mundo de las BDR las relaciones entre las entidades no son bidireccionales (salvo algunos casos).

Mapeo de la Herencia

En el POO, existe el mecanismo de herencia, el cual nos permite reutilizar lógica de una clase y/o extender su comportamiento, sin embargo, este concepto no existe en el mundo relacional, es decir que no existe algo así como “una herencia de tablas”. Entonces, ¿cómo solucionamos esto?

Existen cuatro estrategias de mapeo de herencias:

1. SINGLE TABLE: Una única tabla.
2. JOINED: Una tabla por la superclase y una tabla por cada una de las clases hijas.
3. TABLE PER CLASS: Una tabla por cada clase concreta.
4. MAPPED SUPERCLASS: Los atributos de la superclase son persistidos en las tablas de las clases hijas.

MAPPED SUPERCLASS

Ejemplo de Eze con Proservices

Como todas nuestras clases persistentes poseen el atributo **ID**, vamos a crear una clase abstracta **Persistente** y allí vamos a colocar el atributo ID.

```
public abstract class Persistente {  
    no usages  
    @Id  
    @GeneratedValue  
    private Long id;  
}
```

TRAZABILIDAD EN BASES DE DATOS, poner cuándo se crea, cuándo se da de baja un registro y la última modificación
- Hacer baja lógica en vez de baja física, esto se puede hacer con un campo “activo/estado”

Una vez hecho esto, en nuestras clases persistentes que tengan el atributo ID, haremos que hereden de **Persistente**.

```
public class Disponibilidad extends Persistente {
```

Entonces como todas nuestras clases heredan de **Persistente**, queremos que cada una de ellas tenga su atributo ID, para esto debemos mapear a **Persistente** con una notación **@MappedSuperClass**. Con esto estamos diciendo que todos los atributos definidos en la superclase, en este caso **Persistente**, van a “bajar” a cada una de las tablas de las clases hijas.

DEFINICIÓN

MAPPED SUPERCLASS nos permite que aquellos atributos de una superclase sean persistidos en las tablas de las clases hijas; y la superclase no es considerada una Entidad, no es persistente.

Un uso muy común suele ser cuando todas las clases persistentes tiene una PK subrograda y/o un campo “activo” (boolean), pero entre ellas no existe nada más en común.

Generalmente este mapeo se utiliza cuando la superclase tiene un comportamiento y/o algunos atributos en común.

SINGLE TABLE

Ejemplo de Eze con Proservices

Contexto: Forzamos un poco las cosas a través de una clase **Reputación**, que es abstract, y tenemos 3 (tres) clases hijas: Buena, Media y Mala.

En la clase padre **Reputación** vamos a anotarla como **@Entity**, al ser una Entidad podemos pedirle al ORM todas las reputaciones, sin embargo, al ser una herencia, elegimos una estrategia: **SINGLE_TABLE**, entonces como esta estrategia nos va a generar una única tabla, debemos definir el **@Table**. Además de esto, debemos establecer la columna que actuará como el campo discriminador de la tabla mediante la anotación **@DiscriminatorColumn**.

```
@Entity  
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)  
@Table(name = "reputacion")  
@DiscriminatorColumn(name = "tipo")  
public abstract class Reputacion extends Persistente {
```


IDAÑEZ, LUCIA MARÍA 🐱

Las clases hijas de **Reputación** también deben estar mapeadas, cada una de ellas será una Entidad, por lo tanto tendrán un `@Entity` y a eso le debemos agregar una anotación `@DiscriminatorValue`, la cual servirá para que al momento de insertar una instancia de una clase hija en particular en la columna discriminadora le setee un valor en particular, como por ejemplo:

```
@Entity
@DiscriminatorValue("buena")
public class ReputacionBuena extends Reputacion {
```

Quedando de esta manera:

	tipo	id	nombre	nombreDeMala	nombreDeRegular	nombreDeBuena
	buena	NULL	asdas	NULL	NULL	Eze
	mala	NULL	ueiwueiq	Eze	NULL	NULL
	reg...	NULL	odiwo...	NULL	Eze	NULL

DEFINICIÓN

SINGLE TABLE establece que existe una única superclase y una cantidad **N** de clases hijas.

El resultado de mapear la herencia con la estrategia de SINGLE TABLE será una única tabla en la base de datos. Esta única tabla contendrá una columna por cada uno de los atributos persistentes de la superclase + una columna por cada atributo persistente de cada una de las clases hijas.

Además, tendrá una columna que actuará de **campo discriminador**, esta columna nos va a decir a qué instancia de una clase pertenece cada registro/fila.

Si consideramos una superclase con N atributos persistentes; una clase hija A con M atributos persistentes y otra clase hija B con P atributos persistentes; entonces:

- Si se persiste una instancia de la clase A (con todos sus atributos seteados, incluidos los de la superclase) entonces quedarán P columnas nulas;
- Si se persiste una instancia de la clase B (con todos sus atributos seteados, incluidos los de la superclase) entonces quedarán M columnas nulas.

Como consecuencia de usar esta estrategia, varias columnas pueden quedar nulas, sin embargo, se obtiene una buena performance, ya que solamente se debe consultar una única tabla sin necesidad de hacer JOINS.

- Hacer JOINS le quita performance a la consulta.

JOINED

Ejemplo de Eze con Proservices

En la clase padre **Reputación** vamos a anotarla como `@Entity`, al ser una Entidad podemos pedirle al ORM todas las reputaciones, sin embargo, al ser una herencia, elegimos una estrategia: JOINED, debemos definir el `@Table`.

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
@Table(name = "reputacion")
public abstract class Reputacion extends Persistente {
```

Siguiendo el ejemplo anterior, en vez de tener la anotación `@DiscriminatorValue` en cada clase hija, tendremos `@Table(name = ...)`

```
@Entity
@Table(name = "reputacion_regular")
public class ReputacionRegular extends Reputacion {
```

En este caso, serán necesarios hacer JOINS, ya que nos debemos fijar si la reputación de un prestador es Buena, Mala o Regular y, en base a eso, hacer luego un JOIN contra la tabla correspondiente a la reputación.

- Nos debemos traer el registro que está en la clase padre y luego el registro asociado a la clase hija para recuperar la instancia completa del objeto.
- La “super tabla/tabla padre” crece más rápido que las subtablas.
- Se debe explicitar la columna discriminadora, así el ORM sabe a qué tabla tiene que joinearse.

DEFINICIÓN

IDAÑEZ, LUCIA MARÍA 🐱

JOINED establece que cada una de las PKs de las tablas que representan a las clases hijas, a su vez serán una FK a la tabla que representa a la superclase. Entonces, para recuperar un objeto con todos sus atributos (propios + los que están en la superclase), el ORM debe joinear las tablas.

TABLE PER CLASS

Ejemplo de Eze con Proservices

Siguiendo el ejemplo anterior, En la clase padre **Reputación** vamos a anotarla como `@Entity`, al ser una Entidad podemos pedirle al ORM todas las reputaciones, sin embargo, al ser una herencia, elegimos una estrategia:

`TABLE_PER_CLASS`, en este caso la clase no tendrá su propia tabla.

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Reputacion extends Persistente {
```

Pero las clases hijas de **Reputación** sí tendrán su propia tabla, sin embargo, cada clase hija va a tener las columnas definidas para su clase + las columnas que le correspondan a los atributos de la super clase, esto quiere decir que los atributos de la clase padre BAJAN a las clases hijas.

- Esta estrategia es recomendable cuando las tablas no tienen nada en común entre sí.
- La clase padre es una entidad, pero NO es persistente, es una `@Entity` pero sin tabla.
- No podríamos tener un **id** a su tabla padre (Reputación), deberíamos tener la relación del otro lado.
 - o De Reputación -> Prestador, no de Prestador -> Reputación

DEFINICIÓN

`TABLE PER CLASS` establece que existe una única superclase y **N** clases hijas concretas. El resultado de mapear la herencia con la estrategia Table Per Class genera, en la base de datos, **N** tablas: una por cada clase hija.

Todos los atributos persistentes de la superclase serán persistidos en cada una de las tablas que mapean contra las clases hijas. Entonces, para recuperar polimórficamente todos los objetos, el ORM debe realizar **unions** entre todas las tablas que mapean contra las clases hijas.

Notaciones

Las anotaciones que estamos usando son de JPA que es una estándar de Java que nos permite realizar la persistencia de datos de nuestras clases, pero sin quedar pegados a un ORM en particular, esto quiere decir que JPA establece como “la interfaz” mientras que Hibernate (el ORM que estamos usando) implementa esa interfaz de JPA para realizar la persistencia.

EJERCICIO DE PROSERVICES

@Entity

- Con esta anotación determinamos que la clase es persistente

@Table

- Esto es para establecer que la clase va a tener una tabla
- También le podemos aclarar al ORM el nombre que va a tener la tabla:
 - o `@Table(name = “nombre_de_tabla”)`
→ Esto nos va a permitir que nuestro modelo de datos en la BDR no quede acoplado a nuestro modelo de objetos

@Id

- A nuestra clase le debemos agregar un atributo **id** (Long id) que, mediante la anotación `@Id`, nos establece que esa será la PK de nuestra tabla
- Para los **id** se recomienda usar la anotación `@GeneratedValue` que determinará que nuestro **id** será auto incremental por default (este es el más recomendado), sin embargo, el uso de UU-Id es mucho más seguro
 - o Para el `@GeneratedValue` se puede establecer una **strategy** (distintas formas de otorgar un id)

@Column

- Esta notación sirve para generar una columna en la base en la tabla que le corresponde a la clase
- Podemos aclarar el nombre de la columna:
 - o `@Column(name = “nombre_columna”)`

@Transient

- Le indicamos al ORM que no persista ese atributo de la clase

Regla Práctica: Siempre que tengo una colección es una relación que termina en Many
Puede ser OneToMany o ManyToMany

IDAÑEZ, LUCIA MARÍA 

FORMAS DE MAPPEAR RELACIONES

Para el ejemplo con ProServices

Bidireccional

Contexto: Un SERVICIO puede tener muchas TAREA, es decir que Servicio tiene una lista de tareas y Tarea (a propósito) tiene la referencia al Servicio.

Vamos a la clase Tarea y agregamos las notaciones necesarias como @Entity, @Table y las @Column.

Parándonos en **Tarea** vemos que la relación que tiene con **Servicio** es **MANY TO ONE**, por ende, sobre el atributo **Servicio** colocamos esa notación **@ManyToOne**, sin embargo, debemos declarar bien nuestra relación, para ello agregamos la notación **@JoinColumn** donde especificamos el nombre de la columna que va a tener nuestra FK en la tabla **Tarea** y contra quién de la otra tabla (Servicio) va a ir, quedando de la siguiente manera:

```
@ManyToOne
@JoinColumn(name = "servicio_id", referencedColumnName = "id")
private Servicio servicio;
```

Nombre de la columna
que va a tener la FK

A la columna que
referencia en la otra tabla

En la tabla **Servicio** vamos a tener una colección de **Tarea**, por ende, si nos paramos allí la relación que vamos a tener en **ONE TO MANY**, por ellos sobre este atributo colocamos la notación **@OneToMany** y, como la relación es bidireccional, es importante que agreguemos el **mappedBy** que especifica **CÓMO NOS TIENE QUE ENCONTRAR EN LA OTRA TABLA**.

```
@OneToMany(mappedBy = "servicio")
private List<Tarea> tareas;
```

Acá estamos diciendo que
encuentre a cada una de las
tareas en la clase Tarea con el
atributo "servicio"

Unidireccional

(Del lado del One to Many)

Contexto: Un PRESTADOR tiene una colección de Disponibilidad.

Vamos a la clase Prestador y, parándonos sobre Prestador, vemos que la relación que tiene con Disponibilidad es ONE TO MANY, por ende, sobre el atributo de la colección de Disponibilidad colocamos la notación **@OneToMany**, pero nos estaría faltando la referencia que tendría Disponibilidad, por ende, falta la notación **@JoinColumn** con las especificaciones del nombre de la columna y a quién hace referencia, quedando:

```
@OneToMany
@JoinColumn(name = "prestador_id", referencedColumnName = "id")
private List<Disponibilidad> disponibilidades;
```

Nombre de la columna que va a tener la
FK en la tabla DISPONIBILIDAD

A la columna que referencia la
tabla Prestador, es decir donde
estamos parados

Unidireccional

(Del lado del Many to One)

Contexto: Un SERVICIO OFRECIDO va a tener una referencia al PRESTADOR.

Vamos a la clase Servicio Ofrecido y, parándonos sobre Servicio Ofrecido, tenemos una relación MANY TO ONE hacia Prestador, por ende, nos paramos sobre el atributo de Prestador y colocamos la notación **@ManyToOne**, también debemos agregar la referencia que tendría Servicio Ofrecido a nuestro Prestador, es decir la FK que está en la tabla de Servicio Ofrecido que apunta hacia el Prestador, para generar esta FK utilizamos la notación **@JoinColumn** con las especificaciones del nombre la columna y a quién hace referencia, quedando de la siguiente manera:

```
@ManyToOne
@JoinColumn(name = "prestador_id", referencedColumnName = "id")
private Prestador prestador;
```

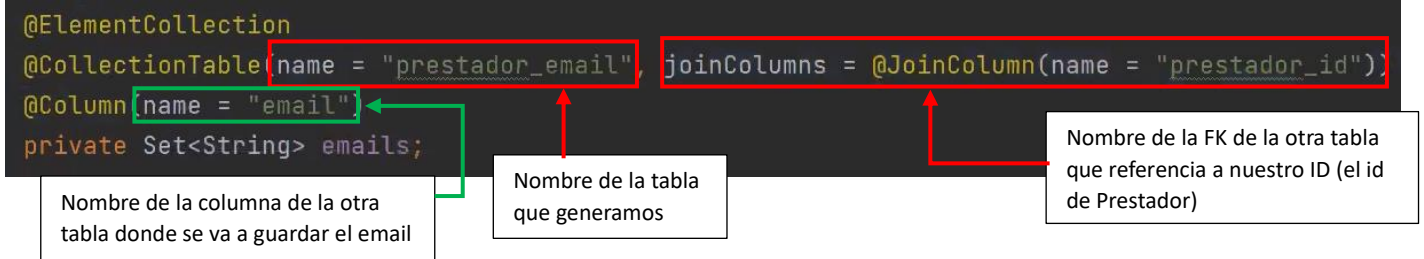
Nombre de la columna que va a tener la
FK en la tabla SERVICIO

La columna a la que hace
referencia/apunta en la tabla
Prestador

MAPEO DE COLECCIONES DE VALUE OBJECT

En el caso de que tengamos que mapear una colección de algún elemento que no sea una clase, es decir que es una colección de Value Objects (string, int, etc.), vamos a usar la notación `@ElementCollection` y cuando usamos esta notación tenemos que especificar cómo se va a llamar la tabla, para eso utilizamos la notación `@CollectionTable`, a su vez dentro de la notación tenemos que especificar cómo se va a llamar la FK de la OTRA TABLA QUE ESTAMOS INVENTANDO. También, en la tabla que estamos parados, tenemos que decir cómo se va a llamar la columna donde se va a guardar esa referencia a la tabla que hicimos, quedando de esta manera:

```
@ElementCollection
@CollectionTable(name = "prestador_email", joinColumns = @JoinColumn(name = "prestador_id"))
@Column(name = "email")
private Set<String> emails;
```

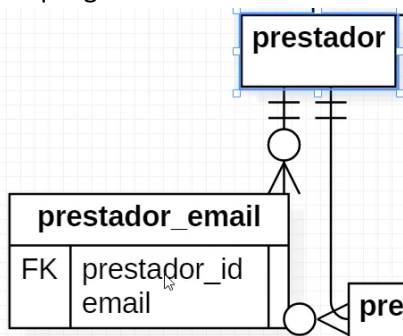


Nombre de la columna de la otra tabla donde se va a guardar el email

Nombre de la tabla que generamos

Nombre de la FK de la otra tabla que referencia a nuestro ID (el id de Prestador)

Lo que generamos:



Embebido

Muchas veces, de acuerdo con el dominio, no es necesario que todas las clases de nuestro mundo de objetos tengan una tabla; para mapear estos casos colocaremos, arriba del atributo donde se haga referencia a aquel objeto que no queremos que tenga una tabla, la notación `@Embedded` y arriba de la clase de ese atributo colocaremos la notación `@Embeddable`, esto significa que todos los atributos de esa clase van a ser columnas que van a ir a parar a la tabla donde se encuentra el atributo.

Ejemplo:

Contexto: La clase Trabajo tiene un atributo dirección que tiene una clase Dirección.

CLASE TRABAJO

```
@Embedded
private Direccion direccion;
```

CLASE DIRECCIÓN

```
@Embeddable
@Setter
@Getter
public class Direccion {
```

Converters

Para algunos casos de mapeo van a ser necesarios CONVERSORES, estos son implementados por JPA y por Hibernate, que con clases que nosotros mismos creamos para transforman ciertos atributos para que puedan ser persistidos en la base.

¿Cómo creo un Converter?

Para crear un converter debemos hacer que nuestra clase implemente una interface llamada `AttributeConverter<...,...>` donde en primer lugar tendrá el tipo de dato Java que recibe el converter y en segundo lugar el tipo de dato SQL al que transforma.

Ejemplo con fechas:

```
public class LocalDateAttributeConverter implements AttributeConverter<LocalDate, Date>
```

Cuando nuestra clase implementa esa interfaz está obligada a tener dos métodos:

- `convertToDatabaseColumn(...)`, que recibe el tipo de dato Java y lo convierte a tipo de dato SQL (el tipo de dato que va a ser persistido en la base)
- `convertToEntityAttribute(...)`, que recibe el tipo de dato SQL y lo convierte a tipo de dato Java

Para que el ORM, en este caso Hibernate, aplique automáticamente esta conversión se debe poner sobre la clase del converter la notación:

```
@Converter(autoApply = true)
```

Ejemplo:

```
@Column(name = "fechaNacimiento", columnDefinition = "DATE")  
private LocalDate fechaNacimiento;
```

En este mapeo, Hibernate, al momento de hacer un INSERT o un UPDATE, se da cuenta que esto es un `LocalDate` y va a utilizar el converter de fechas, por ende, llama al método **`convertToDatabaseColumn`** para persistir ese atributo en la base.

De la misma forma, cuando se intente hacer el proceso de HIDRATACIÓN, es decir cuando traigamos los registros de la base, va a hacer el proceso contrario, Hibernate se da cuenta que hay un `Date` en el registro y llama al método **`convertToEntityAttribute`** del conversor.

CON ENUMERADOS

LOS ENUMERADOS NO SE TRANSFORMAN EN TABLAS

(Formas de mapeo, ponemos de ejemplo a los días de la semana)

Una forma de mapear un enumerado es mediante la notación `@Enumerated`, seguido de un `@Column` con el nombre de la columna. A esta forma también podemos definir cómo queremos que sea el enum:

- `Ordinal`, mediante un número;
- `String`, se guarda la palabra que le corresponde al enumerado.

Otra forma de mapeo es mediante un conversor.

Ejemplo: Nosotros tenemos a `DayOfWeek` como uno de nuestros atributos, pero como está en inglés y nuestro modelo es en español, implementamos un **`DiaDeSemanaConverter`** el cual toma un `DayOfWeek` como tipo de dato Java y da un `String` como "tipo de dato SQL" (en realidad sería un `varchar`).

```
public class DiaDeSemanaConverter implements AttributeConverter<DayOfWeek, String> {
```

Para el método **`convertToDatabaseColumn`** recibimos un `DayOfWeek` (como tipo de dato Java) y devolvemos un `String`; para **`convertToEntityAttribute`** recibimos un `String` y devolvemos un `DayOfWeek` (para el proceso de hidratación).

Para este caso debemos llamar explícitamente al conversor que quiero utilizar para ese atributo, de esta manera:

```
@Convert(converter = DiaDeSemanaConverter.class)  
@Column(name = "dia")  
private DayOfWeek dia;
```


Entity Manager

Todos los ORM de tipo Data Mapper tienen un objeto llamado **Entity Manager** que es el responsable de ayudarnos con las interacciones con la base, es decir con las acciones de guardar, modificar, buscar o eliminar un registro en la base.

El Entity Manager pertenece a la capa de datos y es el encargado de trabajar en el medio del mundo de objetos y el mundo de relacional.

TRANSACCIÓN

Se ejecutan todos los procesos o transacciones y en caso de que uno falle, todos fallan, esto quiere decir que se vuelven para atrás (**rollback**) todos los procesos y de esta manera nuestra base de datos queda en un estado consistente.

Algunas transacciones que utilizaremos son INSERT, UPDATE y DELETE.

¿Cómo se realizan las transacciones?

Para realizar alguna de estas transacciones, le pedimos al Entity Manager una transacción:

```
EntityManager tx = entityManager().getTransaction();
```

Con esto le decimos al Entity Manager que todo lo que venga después, es decir todas las transacciones que vengan después, las “prepare” pero que aún no las ejecute, nuestro Entity Manager va a ejecutar esas transacciones cuando llamemos al transaction commit:

```
tx.commit();
```

Y recién en este punto va a ir a la base.

Ejemplo de una transacción:

```
EntityManager tx = entityManager().getTransaction();
tx.begin();
entityManager().persist(unServicio); //INSERT INTO ....
tx.commit();
```

Proceso de Hidratación

¿Cómo recupera un dato el ORM?

Cuando el ORM recupera un objeto desde la base de datos ejecuta el PROCESO DE HIDRATACIÓN, este proceso consiste en **instancias la clase** del objeto que se quiere recuperar y **populador el objeto**, esto quiere decir que el ORM se encarga de asignarle al objeto cada uno de sus atributos (esto varía si es lazy o eager loading) recuperados desde la base de datos.

Veamos un ejemplo:

Para realizar una sentencia como un SELECT el ORM utiliza un metalenguaje llamado HQL (Hibernate Query Language).

```
Servicio unServicio = (Servicio) entityManager().createQuery(
    "from " + Servicio.class.getName() + " where nombre = :nombre")
    .setParameter("nombre", "Jardineria")
    .getSingleResult();
```

- En este ejemplo el ORM se encarga de recuperar el registro que se pide desde la base y de instanciar el objeto (SERVICIO) que le corresponde a ese registro y de “rellenar” los atributos de ese objeto con las columnas que recupere desde la base (como el nombre).
- Dato: veamos que allí dice **Servicio.class.getName()** lo cual devuelve ‘Servicio’, el ORM se va a encargar de ir a la clase Servicio y fijarse cómo está mapeada para luego saber a qué tabla ir, es decir que si en nuestra clase Servicio tenemos un `@Table(name = “servicio”)` va a ir contra la tabla “servicio” a realizar la query.

```
List<Servicio> servicios = entityManager().createQuery("from " + Servicio.class.getName()).getResultList();
```

- En este caso, el ORM va a HIDRATAR todos los Servicio que sean recuperados/traídos de la base.

HIDRATACIÓN LAZY VS. EAGER

Cuando el ORM está realizando el proceso de Hidratación debe prestar atención a si debe, o no, popular (rellenar) determinado atributo, esto se basa de acuerdo con cómo está marcado el atributo, EAGER o LAZY.

- **EAGER**, el atributo se seteará
- **LAZY**, el atributo no se seteará
 - Este atributo solo será populado por el ORM, cuando sea llamado explícitamente.

En el caso de **LAZY LOADING**, solo se realiza la carga en memoria de los objetos sólo al momento de su utilización.

EAGER LOADING realiza la carga en memoria de los objetos independientemente de si van a ser utilizados o no.

En el caso de Hibernate, solo nos deja marcar como LAZY o EAGER a los atributos con relación a otros objetos, en caso de que sea un vale object como un entero o string no nos dejara marcarlo.

Clasificación de la Arquitectura de los ORM

El mundo de los ORM se clasifica en dos arquitecturas: **ACTIVE RECORD** o **DATA MAPPER**.

Hibernate, el ORM que utilizamos, posee arquitectura DATA MAPPER.

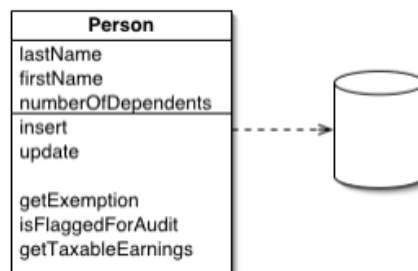
Active Record

En la arquitectura ACTIVE RECORD, los ORMs decoran las clases de entidades persistentes del dominio, agregándoles funcionalidades o responsabilidades. Estas responsabilidades están relacionadas a las acciones de Alta (save), Baja (delete) y Modificación (update) de la entidad (ABM), además poseen otras funcionalidades que permite la búsqueda de registros o elementos de dicha entidad como find, findBy, findAll, etc.

Esta arquitectura elimina el papel del entity manager, dado que sus entidades poseen las funcionalidades que les van a permitir persistirse, actualizarse y eliminarse de la base de datos.

La arquitectura ACTIVE RECORD es más conveniente cuando el aplicativo es más orientado a datos, sin mucho comportamiento, ya que el modelo de objetos queda más acoplado a la capa de datos.

Con esta arquitectura ganamos performance al eliminar al entity manager, sin embargo, poseemos un nivel de acoplamiento muy alto a nuestro mundo de objetos ya que, por cada clase, se generará una tabla; esto no sucede con el Data Mapper, ya que en ese caso podíamos elegir como “mapear” cada clase, teníamos embebidos, herencias, tablas únicas, etc., con este tipo de arquitectura no existen esas cosas ya que por cada clase persistente, se generará una tabla.



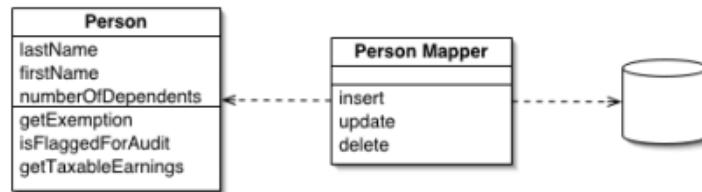
En este caso, la clase **Person** sabe persistirse, hacer update y eliminarse de la base de datos, es decir que no existe un componente intermedio entre el dominio (el modelo de objetos) y la capa de datos.

Data Mapper

En la arquitectura DATA MAPPER los ORMs agregan un nuevo componente intermedio entre la base de datos y las entidades del dominio. Este componente se llama ENTITY MANAGER y se encarga de buscar, agregar, modificar y eliminar (entre otras cosas) objetos de las entidades persistentes.

Este tipo de arquitectura es más conveniente cuando tenemos más comportamiento en nuestro aplicativo y cuando nuestros datos sufren muchas transformaciones.

En este tipo de arquitectura mis clases de dominios no están acopladas con la base de datos, ya que podemos tener clases que no necesariamente serán persistentes.



Métodos de Cascada

Las relaciones entre entidades, a veces, dependen de la existencia de otra entidad. Por ejemplo, la relación **Persona**>>**Dirección**, suponiendo que una dirección le pertenece a una persona, es decir que, sin la Persona, la entidad Dirección no tienen ningún sentido de existencia propio. Cuando eliminamos la entidad Persona, nuestra entidad Dirección también debería eliminarse.

- En este caso, a nivel objetos, estamos hablando de una relación de COMPOSICIÓN.
- La Dirección es contenida por la Persona y la Persona contiene la Dirección.

La CASCADA implica que, cuando realicemos alguna acción sobre la entidad objetivo –sobre la entidad que contiene a la entidad asociada-, la misma acción será aplicada a la entidad asociada.

Existen varios tipos de Cascadas:

- ALL
 - Propaga el impacto de una o varias operaciones desde la entidad contenedora hasta las entidades contenidas.
- PERSIST
 - Este tipo de cascada propaga la persistencia desde la entidad contenedora hasta las entidades asociadas, esto quiere decir que cuando guardamos la entidad principal también se guardan sus entidades secundarias.
- MERGE
 - Este tipo de cascada propaga la operación de actualización desde una entidad contenedora hasta las entidades asociadas, es decir que cuando actualizamos la entidad principal también se actualizan las entidades secundarias.
- REMOVE
 - Este tipo de cascada propaga la operación de eliminación de una entidad contenedora a una entidad asociada, es decir que cuando eliminamos la entidad principal, la entidad secundaria también se elimina.
- REFRESH
 - La operación de “refresh” repopula todos los atributos de un objeto, es decir que refresca los datos contra los que están en la base.
 - Este tipo de cascada propaga la operación desde una entidad contenedora hasta una entidad asociada, esto quiere decir que cuando una entidad principal sea refrescada/repopulada, la entidad secundaria también lo será.
- DETACH
 - “detach” elimina de memoria la entidad/objeto, ya que el ORM no tiene en memoria los objetos todo el tiempo, solo los mantiene un momento y cada tanto los va sacando.
 - Este tipo de cascada propaga la operación desde una entidad contenedora hasta las entidades asociadas, esto quiere decir que cuando una entidad principal sea eliminada de la memoria, la entidad secundaria también lo será.

Sin embargo, no todos estos están disponibles en todos los ORMs.

Cache

Los ORM tienen la capacidad de almacenar en cache, de forma transparente, los datos recuperados desde el medio persistente. Con esto, nos ahorramos de ir a la base (al medio persistente), en términos de tiempo y recursos, para aquellos datos que son consultados con una alta frecuencia.

Existen dos tipos de caches en los ORM:

IDAÑEZ, LUCIA MARÍA 

- Cache de PRIMER NIVEL
- Cache de SEGUNDO NIVEL (no todos los ORMs cuentan con esta cache)

CACHE DE PRIMER NIVEL

La CACHE DE PRIMER NIVEL tiene un alcance de sesión que garantiza que cada instancia de una entidad se cargue solamente una vez en el contexto persistente. Una vez que la sesión se termina o se cierra, la cache de primer nivel se termina.

Esta cache permite que las sesiones concurrentes trabajen con instancias de forma aislada.

Por cada request que llega a nuestro sistema, se atiende en un hilo aparte. Las peticiones se realicen esa sesión de la request, es decir las idas a la base que hagamos durante esa sesión, se “recuerdan” durante la sesión mientras esté abierta, de esta manera nos ahorramos de ir nuevamente a la base a cada rato, ya que tenemos esa información en memoria. Esto el ORM lo hace de forma automática.

CACHE DE SEGUNDO NIVEL

La CACHE DE SEGUNDO NIVEL tiene un alcance de SessionFactory (en Hibernate), esto significa que es compartida por todas las sesiones creadas con la misma factory.

- Por cada request que se escuche en un hilo aparte, va a necesitar tener su propio entity manager instanciado, todos los entity manager se crean en base a un sessionFactory. Todos aquellos entity manager que hayan sido creados por el mismo SessionFactory pueden llegar a compartir objetos, esto quiere decir que, en vez de ir a la base, van a traer la información de los mismos espacios de memoria.

Cuando se busca una instancia de una entidad por su id y la cache de segundo nivel está habilitada, sucede alguno de los siguientes escenarios:

- Si la instancia ya está presente en la cache de primer nivel, es devuelta desde allí.
- Si la instancia no está presente en la cache de primer nivel, pero la instancia está almacenada en la cache de segundo nivel, entonces se obtienen los datos desde allí, se hidrata el objeto y se devuelve la instancia.
- Si no se encuentra en ninguno de los anteriores casos, entonces se realiza el proceso de búsqueda e hidratación común y corriente.

CACHE – ESTRATEGIAS DE CONCURRENCIA

Existen varias estrategias de concurrencia de cache de segundo nivel:

READ_ONLY: se debería utilizar con entidades que nunca cambian.

NONSTRICT_READ_WRITE: la cache se actualiza después de que se haya confirmado que una transacción cambió los datos afectados. Esta estrategia es adecuada si se puede tolerar una mínima inconsistencia eventual.

READ_WRITE: esta estrategia garantiza una fuerte consistencia que se logra mediante la utilización de bloqueos “suaves”. Cuando se actualiza una entidad en la cache, también se almacena un bloqueo suave para esa entidad en la cache, que se libera después de que se confirma la transacción.

CACHE – REPRESENTACIÓN INTERNA

Las entidades no se almacenan en la cache como objetos, sino que solamente se guarda su estado (valores de los atributos).

Los atributos transitorios no se guardan.

Las colecciones no se guardan (a menos que se haya explicitado lo contrario)

En las relaciones ...ToOne solamente se almacena el id de la entidad externa.

Patrón Repositorio


El PATRON REPOSITORIO no es un patrón de diseño, sino que es un PATRÓN ARQUITECTONICO.

Este patron nos ayuda a estructurar el aplicativo mediante una buena separación de las responsabilidades, apoyándose sobre el estilo de la estructura en capas. Propone crear una capa de persistencia para que la misma se encargue del acceso a los datos. En esta capa existen objetos “Repository”, cada repositorio debería poder agregar(unObjeto), modificar(unObjeto), eliminar(unObjeto), buscar y buscarTodos.

El PATRÓN REPOSITORIO puede implementarse haciendo uso de los objetos DAOs.

Los DAO son Data Access Object, esto quiere decir que son los responsables de acceder a los datos. Pueden acceder a una base de datos relacional, a una base de datos no relacional, a la memoria, a archivos, etc., es decir que son objetos que pueden acceder a cualquier medio persistente en busca de información.

El patrón repositorio puede combinarse con el patrón de diseño Strategy para utilizar los DAOs. Cada repositorio podría delegar su responsabilidad en un DAO concreto.

IDAÑEZ, LUCIA MARÍA 

Para los clientes de los repositorios debería ser indistinto el DAO concreto que utilizan. Esto quiere decir que, los clientes de los repositorios no deberían enterarse cuál es el medio persistente.

REPOSITARIOS

La instanciación de un objeto está en el CONTROLLER, pero quien se encarga de guardar ese objeto en la base de datos es el REPOSITORIO, ya que es responsabilidad de la capa de datos, por ende, el CONTROLLER instancia el objeto y llama al REPOSITORIO para que persista esa instancia en la base de datos.

- El Controller se puede comunicar con los Repositorios.

Clase 26/09 (Curso Martes)

Los encargados de atender las peticiones de los usuarios son los controladores (controllers), para esto creamos rutas para las cuales los usuarios crearan peticiones y cada una de esas rutas será atendida por un controlador.

Sesión

Las **sesiones** sirven para identificar al usuario que está intentando acceder a un recurso o ejecutar determinada funcionalidad sobre el Sistema.

Una **SESIÓN** puede almacenar temporalmente información relacionada con las actividades del usuario mientras que se encuentre conectado.

No siempre es necesario tener el concepto de sesión, ya que depende del tipo de arquitectura que se haya establecido para el aplicativo.

Cookies

Por definición las COOKIES son:

“Un archivo pequeño que se guarda en las computadoras (cualquier dispositivo) de las personas para ayudar a almacenar preferencias y demás información que se utiliza en las páginas web que visitan.

Las COOKIES pueden guardar la configuración de las personas en algunos sitios web y, a veces, se pueden utilizar para realizar un seguimiento de cómo los visitantes acceden a los sitios web e interactúan con ellos.”

- Google

Una **COOKIE** es un pequeño archivo creado por el Servidor, pero almacenado en el cliente, es decir que se guarda en el navegador del usuario.

Las cookies viajan al servidor que las creó en cada request (petición) que el cliente realice.

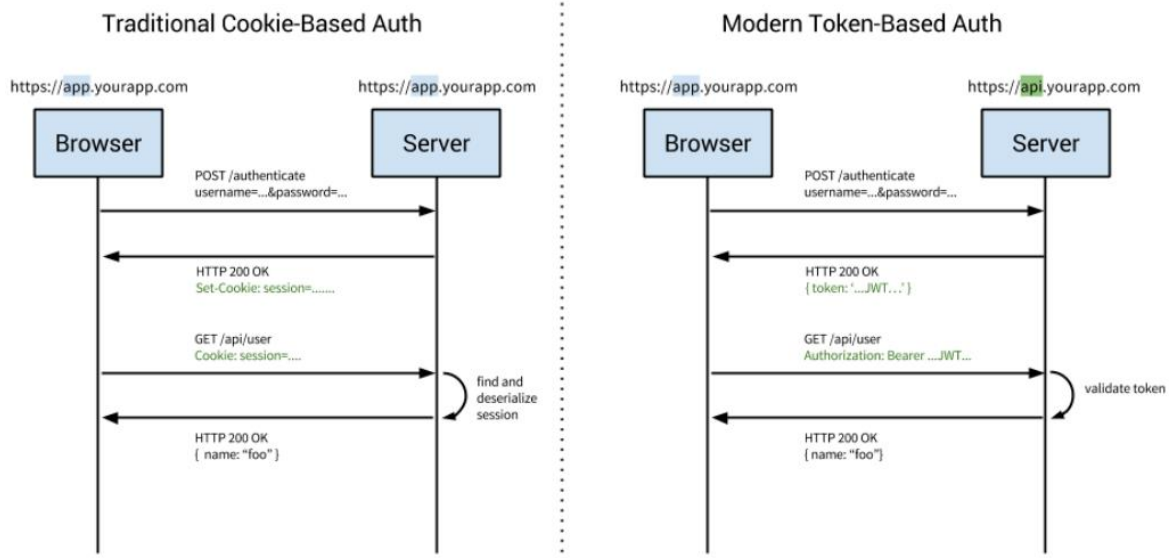
En un sentido más amplio, sirven para que el servidor pueda identificar y reconocer al usuario que está realizando la request.

Stateless vs. Statefull (WEB)

En el sentido arquitectónico de Stateless y Statefull, el concepto de sesión:

STATELESS	STATEFULL
<ul style="list-style-type: none">• No se almacena sesión del lado del servidor.• El usuario se identifica en cada solicitud que realiza.• Para identificarse, el usuario envía un token al servidor. El servidor decodifica este token y descubre quién es el usuario que está realizando la solicitud.	<ul style="list-style-type: none">• Se almacena la sesión del lado del servidor.• El usuario se identifica una sola vez, no en cada solicitud realizada al servidor.• En la solicitud donde el usuario inicia sesión (se validan las credenciales), el servidor genera un espacio para el usuario, ya sea en memoria, archivos u otro medio, otorgándole un id.• El ID que se le otorga al usuario es la COOKIE del usuario.• En posteriores solicitudes, el servidor revisa la cookie del usuario (que llega en la request) y recupera la sesión por ese id.

Las sesiones tradicionales son de arquitectura Statefull mientras que las sesiones más actuales son de arquitectura Stateless.



En la sesión tradicional, usamos espacio del servidor para mantener los datos del usuario, en cambio en las sesiones modernas no estamos usando espacio del servidor para mantener las sesiones, sino que se manda un JWT y en ese JWT hay información del usuario.

Para guardar cosas de la sesión usamos `ctx.sessionAttribute('key', value);` y para recuperar ese atributo se hace `ctx.sessionAttribute('key_usada_antes');`

Recursos del Sistema Web

Cuando diseñamos un sistema con arquitectura web debemos pensar en qué recursos vamos a permitir manipular. Podemos decir que un recurso es una entidad que puede ser manipulada: puede ser dada de alta, modificada, eliminada o listada (buscada).

Para poder identificar qué recursos debe exponer un sistema se deben tener en cuenta los casos de uso, para esto debemos tener en cuenta las routes.

Recursos independientes o autónomos	Recursos dependientes o anidados
<ul style="list-style-type: none"> Recursos que no dependen de nadie, que tienen sentido de existencia propio. 	<ul style="list-style-type: none"> Recursos que no son autónomos y que dependen de la existencia de otros. No tiene sentido listar estos recursos sin que sean condicionados por su recurso contenedor. Estos recursos se suelen ver involucrados en una relación de composición.

Protección de Recursos

Nos va a interesar proteger los recursos que expone un Sistema Web, para que estos recursos no sean accedidos por todos los tipos de usuarios existentes.

En ese sentido, existen al menos tres niveles de protección:

- Autenticación
- Autorización basada en roles
- Autorización basada en roles y permisos

AUTENTICACIÓN

Se dice que los recursos que está protegidos por la autenticación exigen que el usuario se identifique antes de acceder a ellos.

Un ejemplo:

- En un sistema web con capa de presentación visual, se puede redirigir al usuario a una pantalla de LOGIN en el caso que quiera acceder a una ruta protegida sin haber iniciado sesión.
- En un sistema web con una capa de presentación de datos REST, podríamos devolverle al usuario un código 401 (no autorizado) en el caso de que no esté autenticado.

ROLES

Este tipo de protección depende de la protección de Autenticación, esto quiere decir que es necesario que el usuario se identifique.

Los recursos protegidos por roles exigen que el usuario cuente con un rol en particular para poder acceder a ellos.

Este tipo de protección permite que no todos los usuarios manipulen los mismos recursos.

ROLES Y PERMISOS

Este tipo de protección depende de la protección de Autenticación, esto quiere decir el usuario debe estar identificado.

Los recursos protegidos por roles y permisos exigen que el usuario cuente con un rol en particular y con uno o varios permisos para acceder a ellos.

Este tipo de protección permite que no todos los usuarios con mismos roles accedan a los mismos recursos.

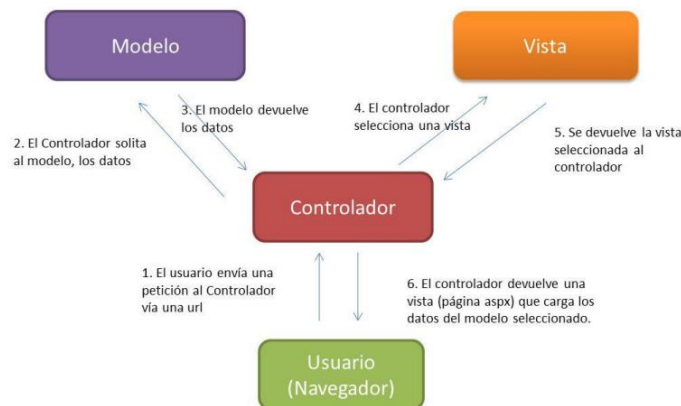
También permite que usuarios con distintos roles cuenten con el mismo permiso.

Patrones Arquitectónicos de Interacción (Web)

Es necesario pensar en los diferentes alcances que tienen los recursos, como: propio, propios y todos.

Los patrones arquitectónicos nos sesgan la estructura de nuestro aplicativo, en este caso el patrón arquitectónico de interacción web consiste en la interacción que tenemos entre los usuarios finales y nuestro sistema.

MVC: MODELO VISTA CONTROLADOR



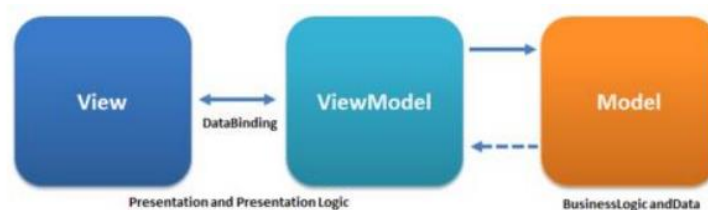
En este patrón de diseño se separan los datos de una aplicación, la interfaz de usuario, y la lógica de negocio en tres componentes distintos.

El usuario a través de su navegador envía una solicitud (request) la cual va a ser captada por un controlador en particular. El controlador, en caso de que sea necesario, solicita los datos al modelo a través de los repositorios.

Una vez que el modelo devuelve los datos al controlador, el controlador selecciona una vista en particular y, gracias a la acción del template engine, los datos que devolvió el modelo y la vista seleccionada por el controlador se combinan. Una vez hecho esto, se devuelve la plantilla al controlador y este la devuelve al navegador.

La capa de middlewares actúa entre el usuario (navegador) y el controlador.

MVVM: MODELO VISTA VISTAMODELO



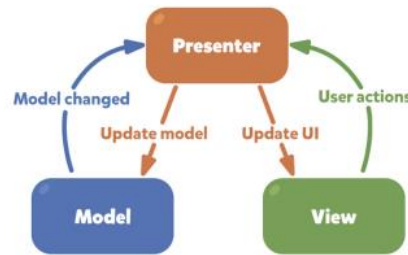
En este patrón de diseño se separan los datos de la aplicación y la interfaz de usuario, pero en vez de controlar manualmente los cambios en la vista o en los datos, estos se actualizan directamente cuando sucede un cambio en ellos, por ejemplo, si la vista actualiza un dato que está presentando se actualiza el modelo automáticamente y viceversa.

IDAÑEZ, LUCIA MARÍA 🐱

Las principales diferencias entre MVC y MVVM son que en MVVM el “Controller” cambia a “ViewModel” y hay un “binder” que sincroniza la información en vez de hacerlo un controlador “Controller” como sucede en MVC. El MVVM puede ser entendido como una correlación con el patrón de diseño Observer.

MVP: MODELO VISTA PRESENTADOR

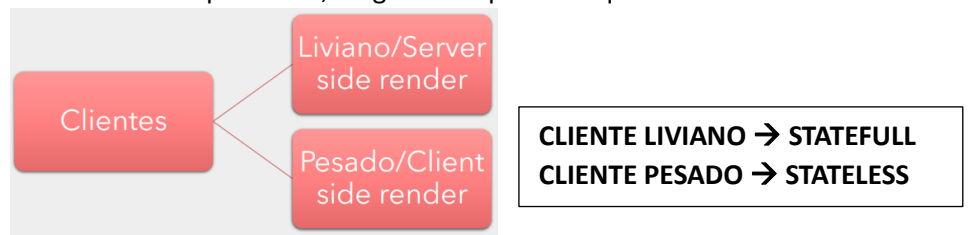
MVP Architecture Pattern



El Modelo-Vista-Presentador o patrón MVP se estructura a partir de las relaciones entre sus componentes. Primero, se genera la petición desde el usuario. Luego, el presentador lleva la solicitud hacia el modelo y allí se selecciona la información que se ha de presentar. Por último, el contenido seleccionado es expuesto, gracias al presentador, en la vista.

Tipos de Clientes

Cuando se tiene un sistema diseñado bajo una arquitectura web y se necesita contar con capa de presentación visual, surge la necesidad de definir quién tendrá la responsabilidad de generar las vistas, esto quiere decir que se discute quién tiene la responsabilidad de generar una vista en particular; surgen dos tipos de arquitecturas:



CLIENTE LIVIANO

En el caso del cliente liviano, las vistas son generadas del lado del servidor, es decir que el navegador (nuestro cliente) no tiene la responsabilidad de generarlas.

El cliente/navegador solo se encarga de mostrar la respuesta a la petición; casi siempre estas respuestas (request) serán en HTML.

En algunos casos, se puede contener lógica escrita en JavaScript del lado del cliente.

En este caso, tenemos más procesamiento del lado del servidor, ya que las vistas son generadas del lado del servidor lo cual consume procesamiento, al igual que con la memoria, debido a que el servidor es quien se encarga de llamar al template engine, procesar todas las vistas, etc.

CLIENTE PESADO

En el caso del cliente pesado, las vistas son generadas del lado del cliente, es decir en el navegador. El cliente posee toda la lógica necesaria para presentar los datos recibidos desde el servidor, mediante código JavaScript.

Las respuestas a las peticiones del usuario con siempre JSON (o algún otro tipo de formato de intercambio).

En el caso del cliente pesado, el cliente es quien tiene mayor procesamiento, ya que es el encargado de recibir los JSON que vienen del servidor, generar las vistas, rellenarlas con los datos, etc.

Mi backend va a ser totalmente STATELESS en un cliente pesado, por ende, debo mandarle el token para interactuar.

	Cliente Liviano	Cliente Pesado
Generación de la Vista	Servidor	Cliente
Stream (dato que se transmite)	HTML	JSON
Procesamiento	<ul style="list-style-type: none"> Mayor en Servidor Menor en Cliente 	<ul style="list-style-type: none"> Mayor en Cliente Menor en Servidor
Uso de Memoria	<ul style="list-style-type: none"> Mayor en Servidor Menor en Cliente 	<ul style="list-style-type: none"> Mayor en Cliente Menor en Servidor
Lógica de Interacción	Servidor	Cliente
Usabilidad (*)	-	+
Escalabilidad (*)	-	+
Mantenibilidad (*)	-	+

Concurrencia: cantidad de usuarios que debo soportar en un mismo momento

Usabilidad: En un cliente pesado, se maximiza la usabilidad ya que se evitan las recargas de la página, por ende, la experiencia de usuario es “mejor”, mientras que en un cliente liviano se debe esperar a la recarga de la página. Sin embargo, el cliente pesado en dispositivos más antiguos (con no muchos recursos a nivel hw), no provee una buena experiencia de usuario, ya que el cliente pesado al tirar la lógica del lado del cliente consume muchos recursos de procesamiento.

Escalabilidad: En un cliente pesado, se maximiza la escalabilidad ya que hay menos trabajo del lado del servidor, por ende, podemos “agrandarlo” mucho más.

Mantenibilidad: En un cliente pesado es más cohesivo, por ende, se maximiza la mantenibilidad ya que es más fácil de testear tanto el back como el front, debido a que están completamente separados, hay más modularidad. La lógica del back y del front está separada, se “levantan” los proyectos por separado (repositorios separados).

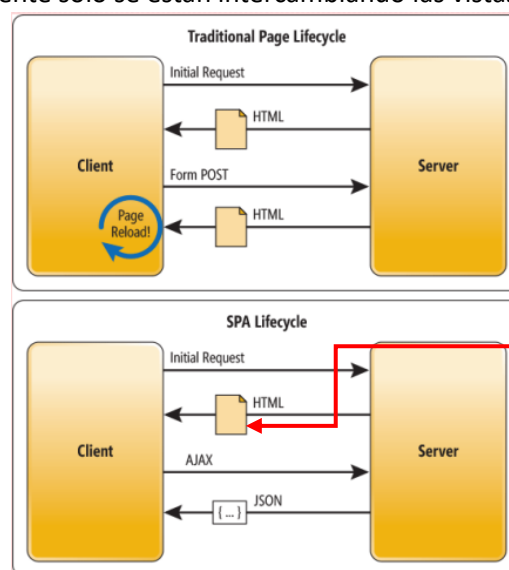
SPA: Single Page Application

Un concepto: “SPA son las siglas de **Single Page Application**. Es un tipo de aplicación web donde todas las pantallas las muestra en la misma página, sin recargar el navegador.”

Este concepto nos dice que un SPA es un cliente pesado.

Otro concepto: “Una **SPA** es un sitio donde existe un único punto de entrada, generalmente el archivo **index.html**. En la aplicación no hay otro archivo HTML al que se pueda acceder de manera separada y que nos muestre un contenido o parte de la aplicación, toda la acción se produce dentro del mismo **index.html**.”

Aunque solo tengamos una página, tendremos varias vistas, es decir que tendremos varias pantallas. Entonces, en la misma página, dependiendo de la interacción, se irán intercambiando varias vistas, produciendo el efecto de que se tienen varias páginas, cuando realmente solo se están intercambiando las vistas.



Esta request inicial es mucho más pesada que el resto ya que me traigo toda la lógica del front

Diagrama de Despliegue y Componentes

DIAGRAMA DE COMPONENTES

IDAÑEZ, LUCIA MARÍA 🐱

- El DIAGRAMA DE COMPONENTES es un diagrama de arquitectura UML que presenta un nivel de abstracción más alto que el diagrama de clases.
- Es un diagrama estático ya que busca presentar todos los componentes de software existentes en el Sistema y sus relaciones.

COMPONENTE

Un COMPONENTE se representa como un rectángulo, con dos pequeños rectángulos en el lado izquierdo uno arriba de otro.



Para que dos componentes de Software interactúen entre ellos, al menos uno debe exponer una interface. Esta interface podría ser una API REST o un conjunto de funciones.

Gráficamente, que un componente exponga una interfaz se representa como un “círculo”. A su vez, para representar que otro componente usa esa interfaz, se utiliza una “semicircunferencia”.

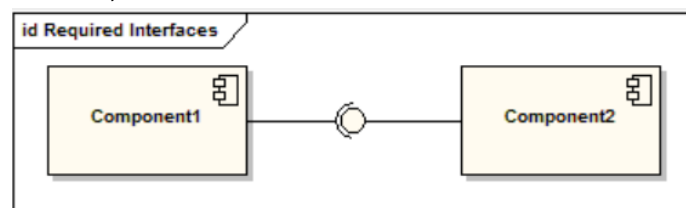


DIAGRAMA DE DESPLIEGUE

- El Diagrama de Despliegue es un diagrama de arquitectura UML.
- El Diagrama de Despliegue es un diagrama estático-físico que busca presentar todos los nodos existentes en la arquitectura de un Sistema y los componentes de Software que contienen.

Elementos:

Artefacto: es un producto desarrollado por el software, se representa por un rectángulo con el nombre y la palabra ‘artefacto’ encerrado por flechas dobles.

Asociación: es una línea que indica un mensaje u otro tipo de comunicación entre nodos.

Componente: es un rectángulo con dos pestañas que indica un elemento de software.

Dependencia: es una línea discontinua que termina en una flecha, indica que un nodo o componente depende de otro.

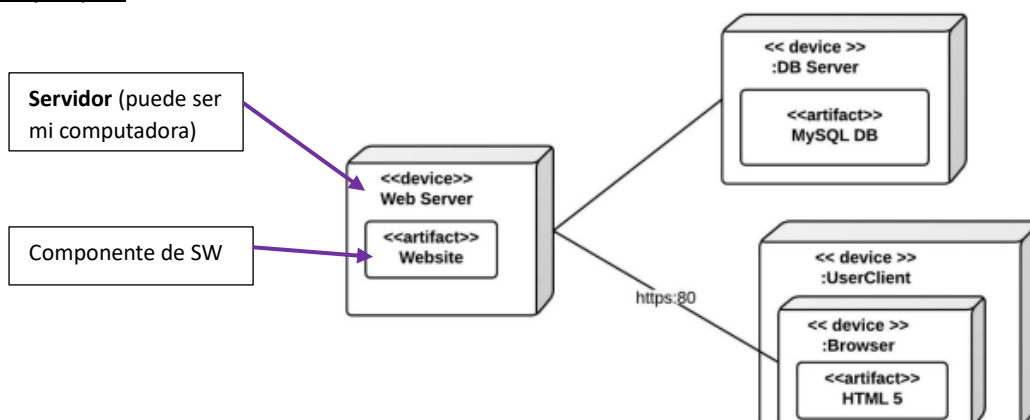
Interfaz: es un círculo que indica una relación contractual. Aquellos objetos que se dan cuenta de que la interfaz debe completar cierto tipo de obligación.

Nodo: es un objeto hardware o software, mostrado por un cubo (representa un servidor). Un NODO es todo objeto que tienen capacidad de cómputo, es decir que debe poder procesar algo.

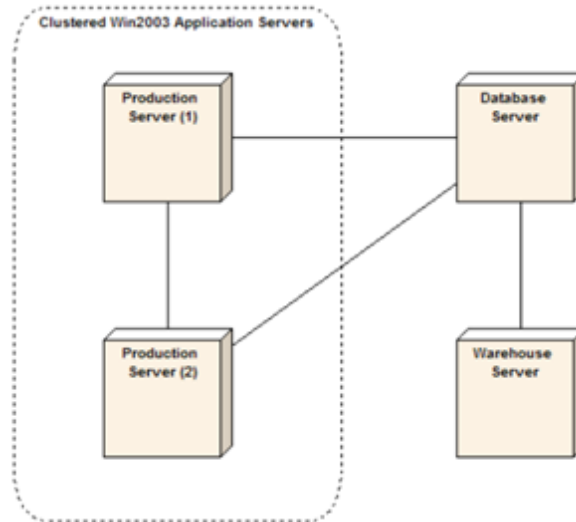
Nodo como contenedor: es un nodo que contiene a otro nodo dentro de sí, en el que los nodos contienen componentes.

Estereotipo: es un dispositivo contenido dentro del nodo, presentado en la parte superior del nodo.

Un ejemplo:



Otro ejemplo:



¿Qué **COMPONENTES** (Cajas) podemos encontrar en una Arquitectura?

- Navegador Web
- Aplicación Móvil
- Motor de Base de Datos
- Load Balancer (Balanceador de Carga)
- Caché
- Biblioteca
- Otros.

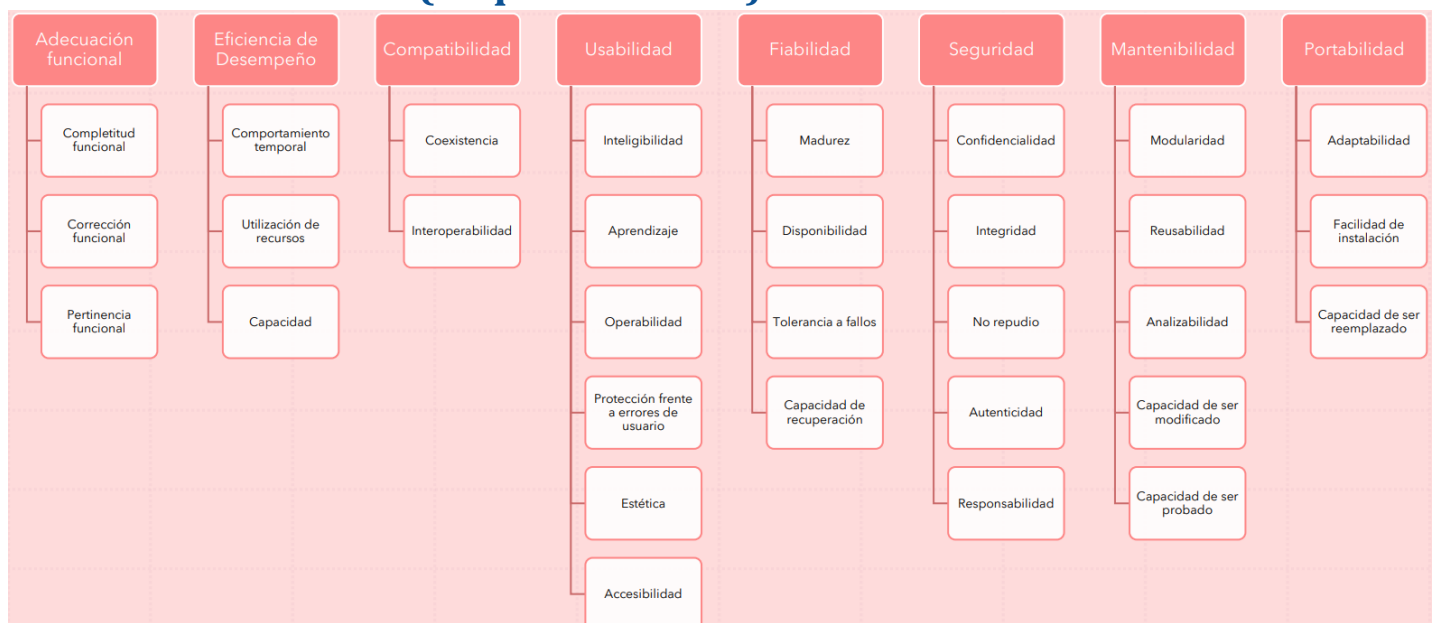
Los **COMPONENTES** son **CÓDIGO**

Los **NODOS** son **FÍSICOS**

¿Qué **NODOS** (Cubos) podemos encontrar en una Arquitectura?

- Servidor de Base de Datos
- Servidor de Aplicación
- Servidor de Archivos
- Dispositivo Móvil
- Computador Personal
- Dispositivo Embebido
- Otros.

Atributos de Calidad (Arquitectónicos)



Aquellos atributos de calidad que casi siempre podemos evaluar son:

- Eficiencia de Desempeño – Utilización de recursos → Performance
- Fiabilidad – Disponibilidad

IDAÑEZ, LUCIA MARÍA 🐱

- Fiabilidad – Tolerancia a fallos
- Seguridad (en general)

Tradeoffs (tira y afloje) entre Atributos de Calidad

Los **atributos de calidad** pueden **entrar en conflicto** unos con otros:

- Performance vs. Seguridad
- Seguridad vs. Disponibilidad (aunque estén vinculados)
- Performance vs. Modificabilidad

Se deben **evaluar** los múltiples **atributos de calidad** con el objetivo de **Diseñar** un **Sistema** “Good Enough” (lo suficientemente bueno) para los **Stakeholders**.

Escalabilidad

¿Qué sucedería si un único Servidor tuviera que atender solicitudes de un número muy elevado de clientes a la vez?
¿Podría hacerlo solo?

Si pensamos que un solo servidor atiende un elevado número de peticiones, nuestro servidor podría ser un cuello de botella y sucederían cosas como:

- La **PERFORMANCE** podría degradarse sustancialmente, es decir que los tiempos de respuesta pueden ser muy elevados.
- La **DISPONIBILIDAD** podría verse afectada ya que, si el Servidor se cae, todos los clientes quedarían sin poder realizar sus solicitudes.

Sin embargo, podemos enfrentar estos problemas de dos formas diferentes:

- Crecer verticalmente
- Crecer horizontalmente

Solucionar problemas metiendo otros componentes (más capas) lleva a tener otros problemas

ESCALABILIDAD VERTICAL

- La **escalabilidad vertical** implica agregarle MÁS RECURSOS al Servidor físicamente (memoria, disco, procesamiento, etc.) para que éste pueda atender más solicitudes de forma concurrente.
- Existe una limitación física ya que llegará un punto en el cual no se podrán agregar más recursos al Servidor, ya que todo producto hardware llega a una “capacidad máxima” de expansión.
- Este tipo de escalabilidad NO SOLUCIONA el problema de Disponibilidad, sino que pretende resolver el problema de la PERFORMANCE.

ESCALABILIDAD HORIZONTAL

- La **escalabilidad horizontal** implica agregar MÁS EQUIPOS Servidores para atender las solicitudes de los clientes.
 - De esta manera se distribuye la carga entre los distintos servidores en los momentos de concurrencia.
- No existe limitación física para este tipo de escalabilidad, se puede agregar tantos equipos como se dispongan.
- Posee una cierta complejidad ya que se deben distribuir las solicitudes entre todos los equipos Servidores.
 - La distribución se puede realizar por algún algoritmo: Sticky Session, Round Robin, etc.
 - Con un algoritmo Sticky Session se mantiene el mismo servidor con el que se inició la sesión.
- Es necesario contar con un Balanceador de Cargas que está delante de los servidores.
 - El balanceador de carga se encarga de la distribución de las solicitudes, en este componente se puede configurar el algoritmo con el cual se van a distribuir las solicitudes.
- Este tipo de escalabilidad pretende solucionar el problema de Disponibilidad y Performance.

Nota:

Si nuestro backend es statefull, ¿cómo hago para mantener la sesión entre servidores? En el caso de que la distribución de solicitudes sea con el algoritmo Round Robin, para esto se debe tener algo para mantener la sesión en común entre todos los servidores, por ejemplo, un file system compartido, una base de datos para todas las sesiones, un servidor exclusivo solo para tener todas las sesiones (en caso de que estén en memoria), etc.

Clase 11/10

WEBHOOK

Una integración mediante webhooks, para que un sistema no le esté preguntando a cada rato a otro sobre novedades o información, un sistema le pide a otro una dirección de CALLBACKS, estas sirven para que cada vez que ocurra una novedad, un sistema le informe a otro a través de esa dirección.

- Se expone un endpoint, API REST, donde el sistema que utiliza este componente conoce la request que se obtiene.

COLA DE MENSAJES/TRABAJO

Una cola de mensajes es un componente de software que admite la comunicación de forma asincrónica entre dos componentes de Software. La arquitectura básica de una cola de mensajes consiste en un **productor/publicador** que crea mensajes y los publica en la cola. Luego, otros servicios o servidores llamados **consumidores/suscriptores** se suscriben a la cola y comienzan a recibir los mensajes de la cola, estos mensajes son procesos por ejecutar.

- La cola funciona como integración entre dos sistemas o componentes distintos.

SINGLE SIGN ON (SSO)

El **SINGLE SIGN ON (SSO)** es un método de autenticación que le permite a los usuarios autenticarse con múltiples aplicaciones y sitios web mediante un solo conjunto de credenciales.

REPLICA DE BASE DE DATOS

La replicación de bases de datos puede ser utilizada en muchos de los sistemas. Para que las distintas instancias de una misma base de datos estén actualizadas, podríamos implementar una arquitectura MAESTRO-ESCLAVO.

- Podríamos establecer que en la instancia ESCLAVO leo mientras que en la MAESTRO escribo y leo a la vez.

Esta arquitectura opera con, mínimo, dos bases de datos:

- **Base de datos máster/maestro:** soporta operaciones de escritura (insert, update y delete)
- **Base de datos réplicas/esclavo:** únicamente soporta operaciones de escritura (select)

Las ventajas de la replicación de datos son:

- ✓ Rendimiento: la cantidad de consultas que se procesan en simultáneo aumenta debido a que las consultas se distribuyen en distintas bases de datos y no siempre las atiende la misma base (máster)
- ✓ Fiabilidad: ante la pérdida de una base de datos se puede encontrar una copia de datos en otra base

Aplicación Nativa

La aplicación nativa no es portable, se necesitan dos o más aplicativos para distintos entornos de despliegue (Android/iOS). La mantenibilidad es complicada ya que, si se toca código en un aplicativo, en el otro también se debe modificar porque deben quedar iguales ambos. Las aplicaciones nativas acceden a todos los recursos del teléfono y maximizan su uso, son más performantes.

Aplicaciones Híbridas

Una aplicación no está escrita en tecnología nativa, una aplicación híbrida ejecuta un navegador embebido. Una ventaja es que el código de una aplicación híbrida es único, ya que, al ser un navegador, se puede ejecutar en distintos dispositivos (es portable), por ende, su mantenibilidad es más sencilla. Al trabajar con tecnología híbrida no dispongo de todos los recursos del dispositivo/celular, por ende, no es tan performante.

25/10

Arquitectura de SW

Aplicaciones Monolíticas y No Monolíticas

APLICACIONES MONOLÍTICAS

Se caracterizan por tener toda la lógica de negocio en un único proyecto, esto quiere decir que está todo en un solo repositorio.

Si pensamos en un sistema que tiene una Arquitectura Web Monolítica, un cliente liviano, vemos que todos los componentes están en un solo código fuente que se ejecuta del lado del Servidor.

Aunque estas aplicaciones son más fáciles de desarrollar, no siempre son una buena opción, dado que se complejiza la situación si son varias personas trabajando sobre un mismo proyecto, ya que todos trabajan sobre el mismo código. Además, si se realiza algún cambio en el código fuente, todo el aplicativo debe volver a desplegarse (como en el TP).

APLICACIONES NO MONOLÍTICAS

Las APLICACIONES NO MONOLÍTICAS se basan en que la lógica está distribuida en muchos componentes.



SOA – Service Oriented Architecture

SOA es un estilo de arquitectura que se basa en romper la lógica de una aplicación en unidades independientes denominadas SERVICIOS.

PRINCIPIOS

- El valor del negocio por encima de la estrategia técnica → VALOR NEGOCIO > ESTRATEGIA TÉCNICA
 - Hay cuestiones del negocio que van a mejorar por aplicar una cierta arquitectura, siempre se piensa en mejorar las cosas del negocio.
- Las metas estratégicas por encima de los beneficios específicos de los proyectos
 - SOA se piensa a nivel organización, no se aplica a un proyecto sí y a otro no
- La INTEROPERABILIDAD intrínseca por encima de la integración personalizada
 - Los componentes de la arquitectura se deben integrar de manera estándar
- Los Servicios Compartidos por encima de las implementaciones de propósito específico
 - Los servicios que utilizamos en la organización deben ser comunes a todas las aplicaciones/sistemas de la organización
- La Flexibilidad por encima de optimización
 - Maximizar la facilidad de cambios
- El Refinamiento Evolutivo por encima de la perfección inicial
 - Cuando nuestro componente está listo para integrarse, los demás componentes no se deben ver impactados tanto en su integración como en su actualización/mejora

ACTORES

- Consumidor de Servicios (Cliente)
- Proveedor de Servicios (Servicio)
- Registro de Servicios



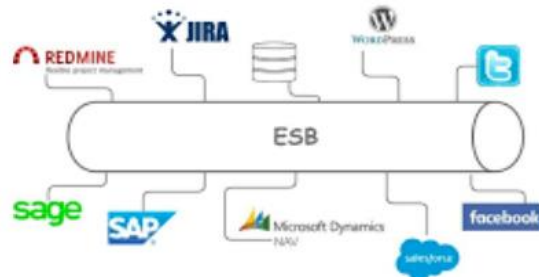
COMPONENTES

- **Consumidor de servicios:** aplicación, módulo de sw u otro servicio que demanda la funcionalidad proporcionada por un servicio

- **Proveedor de servicios:** entidad que acepta y ejecuta consultas de consumidores y publica sus servicios y su contrato de interfaces en el registro de servicios para que el consumidor pueda descubrir y acceder al servicio
- **Registro de servicios:** es un repositorio de servicios disponibles, permite visualizar las interfaces de los proveedores de servicios a los consumidores

¿Qué es un ESB?

Un **BUS DE SERVICIO EMPRESARIAL** (ESB, como indican sus siglas en inglés) es un **componente de arquitectura** de sw que gestiona la comunicación entre múltiples servicios web. Se enfoca en resolver el problema que surge cuando los servicios web dentro de una organización son muchos, lo que hace necesario que se tengan que desarrollar conectores que permitan comunicar las diferentes aplicaciones.



Debo desarrollar componentes que se integren al ESB, no entre componentes.

Con el ESB puedo perder un poco de performance, ya que hay una “capa” o componente intermedio, pero gano flexibilidad y desacoplo componentes.

Para hacer la integración al ESB, primero se debe tener un conector, este conector expone una interfaz a la cual me tengo que conectar.

- Quien sea “menos importante” (en cuanto a negocio) desarrolla el conector al ESB
 - Un conector es un código que se integra contra el ESB

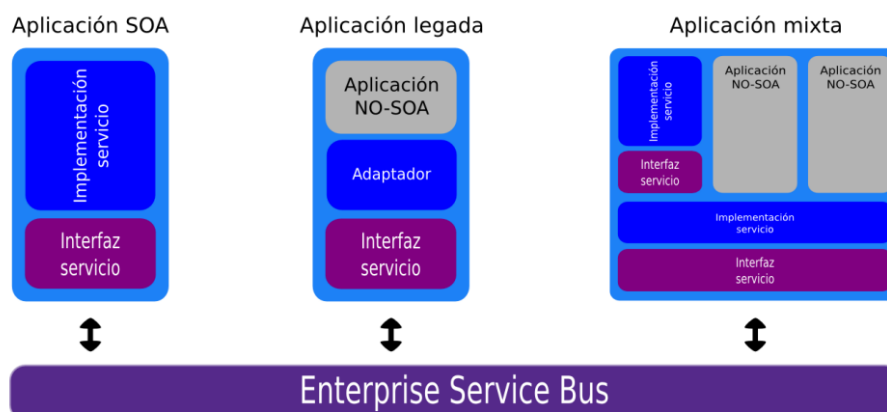
En un ESB los servicios no interactúan directamente, sino que la comunicación es a través de un conector.

- El ESB identifica los servicios y establece la comunicación entre ellos, los servicios no tienen la necesidad de conocer dónde están y quiénes son los otros participantes en la comunicación.
- El ESB permite el flujo de mensajes a través de los diferentes protocolos.
- También permite que, si el otro servicio está caído, dejar el mensaje en el Bus y que cuando el otro servicio esté disponible, lo reciba.

Algunos ejemplos de ESB son OpenESB, Oracle ESB, Azure Service Bus, IBM WebSphere ESB, etc.

Tipos de Aplicaciones

Tipos de aplicaciones SOA



Microservicios

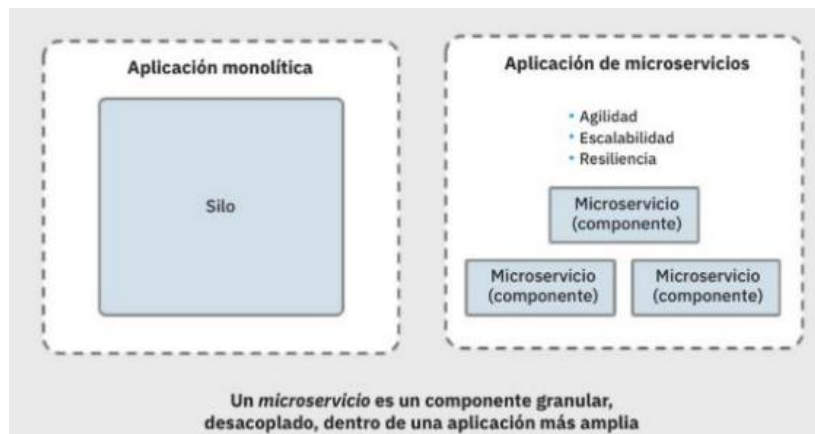
MICROSERVICIOS es un ESTILO ARQUITECTÓNICO con un enfoque para desarrollar UNA ÚNICA APLICACIÓN como un **conjunto de pequeños servicios**. Cada uno de estos servicios ejecuta un propio proceso y resuelve UNA SOLA COSA (Single Responsibility), COMUNICÁNDOSE con mecanismos ligeros, mayormente una API REST.

Estos servicios se basan en capacidades empresariales, es decir que cada servicio debe representar una parte del negocio. Cada servicio debe estar desplegado en un contenedor de forma independiente, cada servicio debe estar en un NODO SEPARADO, siempre el microservicio tiene que estar asociado al nodo en particular para que después el despliegue de un microservicio no afecte al despliegue de otro y las herramientas de despliegue sean totalmente automatizadas.

Hay una parte de GESTIÓN CENTRALIZADA de servicios, esto quiere decir que, si bien cada microservicio puede estar implementado en diferentes tecnologías, **cada servicio debe gestionar su propia persistencia**.

Microservicios vs. Monolítica

Cada servicio debería ser autónomo, con su propia base de datos



SOA → Integración con aplicaciones

Microservicio → Cómo se desarrolla una aplicación en particular

Características

- En una ARQUITECTURA DE MICROSERVICIOS, los servicios son pequeños e independientes y están débilmente acoplados.
- Cada servicio tiene un código base independiente.
- Los servicios se pueden implementar de manera independiente. Un equipo puede actualizar un servicio existente sin tener que volver a desplegar toda la aplicación.
- Los servicios son los responsables de conservar sus propios datos o estado externo.
- Los servicios se comunican entre sí mediante API bien definidas.
- No es necesario que los servicios sean implementados en la misma tecnología.
- Cada servicio puede escalar de manera independiente

Sin embargo, algunas desventajas son:

- Problemas de consistencia, referencia, etc.
 - Cuando tenemos muchos servicios separados, deberemos consultar a cada servicio por los datos que necesitamos, por ejemplo: si yo tengo un servicio **compras** y otro **producto** voy a necesitar saber qué productos se compraron, por ende, voy a tener una FK en compras que reference a producto
- Las consultas entre servicios no gestionamos nosotros el tiempo de respuesta, debemos tener en cuenta en acoplamiento entre servicios, por ejemplo: si mi módulo de **compras** depende de **producto** y realizo una consulta, debo tener cuidado con el tiempo de respuesta de cada uno y con su disponibilidad
 - Tiempo de respuesta y Disponibilidad
- También debo tener en cuenta los fallos porque las transacciones se generan en distintos nodos, por ende, no puedo rastrear el error (son distintos componentes) → Problemas de Trazabilidad
- Se debe seguir manteniendo la compatibilidad de la interfaz con todos aquellos servicios que se integran a mi servicio → Versionar los endpoints/APIs

Separar grandes monolitos en muchos servicios pequeños:

Cada servicio ejecuta su propio proceso. Esta regla se denomina un servicio por contenedor.

Optimizar los servicios para una función única:

Una sola función empresarial por servicio. Esto hace que cada servicio sea más pequeño y simple para escribir y mantener. Esto se denomina Principio de responsabilidad única (SRP).

Comunicarse a través de las API REST y los Buses de Mensajes:

Los microservicios tienden a evitar el estrecho acoplamiento introducido por la comunicación implícita a través de una base de datos.

Aplicar CI/CD por servicio:

En una gran aplicación compuesta por muchos servicios, los diferentes servicios evolucionan a velocidades diferentes. Cada servicio tiene una única y continua tubería de permisos de integración/entrega que avanza a un ritmo natural.

La CI (integración continua) es la práctica de integrar cambios de código en un repositorio múltiples veces al día. La CD tiene dos significados: la entrega continua automatiza las integraciones de código, mientras que la implementación continua entrega automáticamente las versiones finales a los usuarios finales. Las pruebas frecuentes de CI/CD reducen los errores y defectos del código.

Aplicar decisiones de alta disponibilidad (HA) / “clustering” por servicio:

En el enfoque monolítico se escalan todos los servicios del monolito al mismo nivel y esto hace que se usen en exceso los recursos. En un sistema grande, no todos los servicios necesitan ser ampliados, muchos pueden ser desplegados en un número mínimo de servidores para conservar los recursos.

COMPONENTES NECESARIOS

- **Administración:** Este componente es responsable de la colocación de servicios en los nodos, la identificación de errores, el reequilibrio de servicios entre nodos, etc.
- **Detección de Servicios:** Mantiene una lista de servicios y los nodos en que se encuentran. Permite la búsqueda de servicios para localizar el punto de conexión de un servicio.
- **Puerta de enlace de API:** Esta puerta es el punto de entrada para los clientes. Los clientes llaman a la puerta de enlace de API, la cual reenvía la llamada a los servicios en el back-end.

Opciones de Despliegue

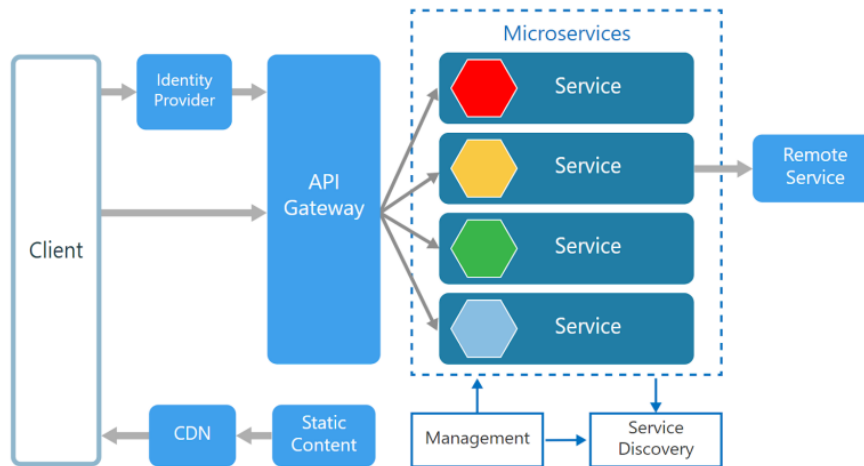


← Servidor físico vs. Serverless →

¿CUÁNDO DEBERÍAMOS USAR MICROSERVICIOS?

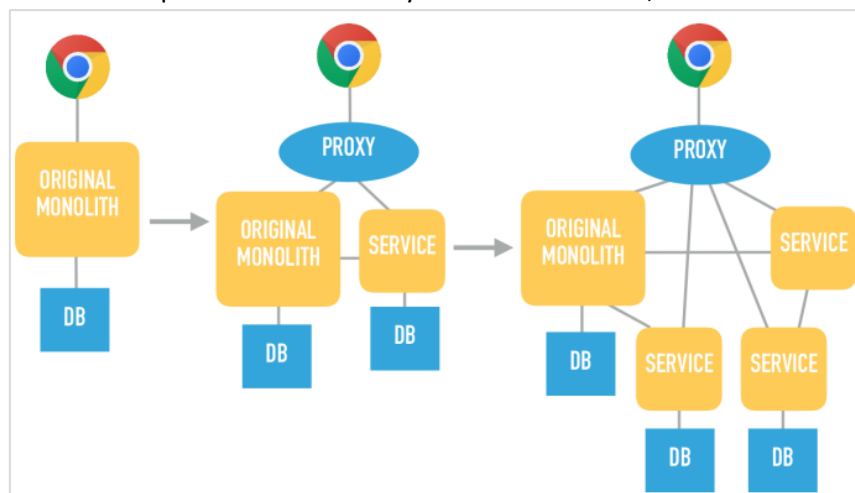
- Aplicaciones grandes
- Aplicaciones complejas que tendrán una gran escalabilidad y no sé cuándo va a escalar (escalar de forma elástica)
- Aplicaciones que requieren escalar de manera dinámica
- Aplicaciones que crecen rápidamente a nivel funcional
- Aplicaciones con dominios o subdominios complejos o variados
- Una organización que tenga equipos de desarrollo pequeños

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.



Algunos patrones (a nivel arquitectónico)

- ✓ El **patrón fachada** define una interfaz a través de una **API externa** especificada para un sistema o subsistema.
- ✓ Los **patrones de entidad (entity)** y **agregado (aggregate)** son útiles para identificar conceptos específicos de la empresa que se relacionan directamente con el microservicio.
 - Entidad → Se debe crear un microservicio por entidad, es decir que la funcionalidad de nuestro microservicio se crea alrededor de una entidad
 - Agregado → Combinación de entidades para crear el microservicio
- ✓ El **patrón servicios** ofrece una forma de mapear operaciones que no corresponden a una sola entidad o agregado, esto quiere decir que se deja de lado la visión de que cada entidad es un microservicio
 - Servicios → Crear un concepto sobre una entidad, no necesariamente un microservicio debe basarse sobre una entidad sino más bien sobre un procesos o conceptos relacionado a ella
- ✓ El **patrón adaptador de microservicios** (Microservicios adaptadores) adapta entre dos API diferentes (conceptualmente). Se adapta una API heredada o servicio tradicional SOAP a una API RESTful o técnicas de mensajería ligeras.
- ✓ El **patrón strangler** (estrangulador) propone realizar un REINGENIERÍA de una aplicación monolítica y orientarla a microservicios.
 - Rompe partes de una aplicación monolítica y a cada una de esas, la vuelve un microservicio



- ✓ El **patrón service registry** (registro de servicio) está relacionado a identificar servicios y poder intercambiarlos de forma fácil.
- ✓ El **patrón correlation ID** y **Log Aggregator** resuelve problemas de trazabilidad, descubrimiento y seguimiento de errores.
- ✓ El **patrón circuit breaker** está vinculado a detectar y cortar el llamado a servicios de manera incorrecta.

Herramientas concretas:

- Service Register y Discovery
- Circuit Breaker
- API Gateway
- Observabilidad y Gestión
- Orquestador

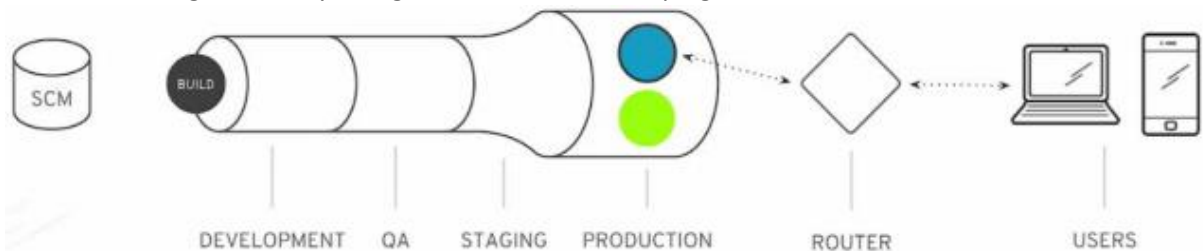
Blue/Green Deployment

A grandes rasgos, es importante tener separados los entornos de desarrollo, testing, QA, staging, etc.

Un aspecto importante en el despliegue es mantener la disponibilidad, por ello en el despliegue blue/green lo que hace es desplegar la aplicación completa en nodos que van a correr en paralelo a los nodos que corren actualmente y, de pronto a través del router, hacen un cambio a qué versión de aplicación estoy apuntando.

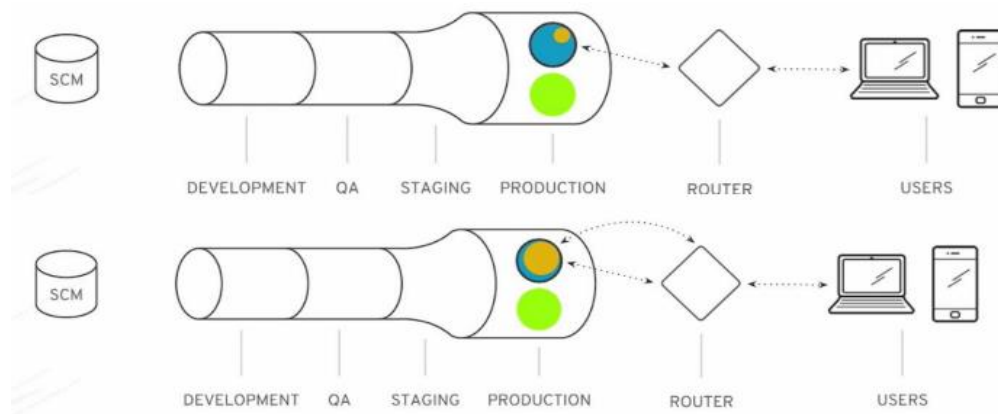
En ese cambio, como lo hago con el despliegue ya realizado de las nuevas versiones, no tengo caída de servicio.

- El cambio lo hago cuando ya tengo toda la solución desplegada



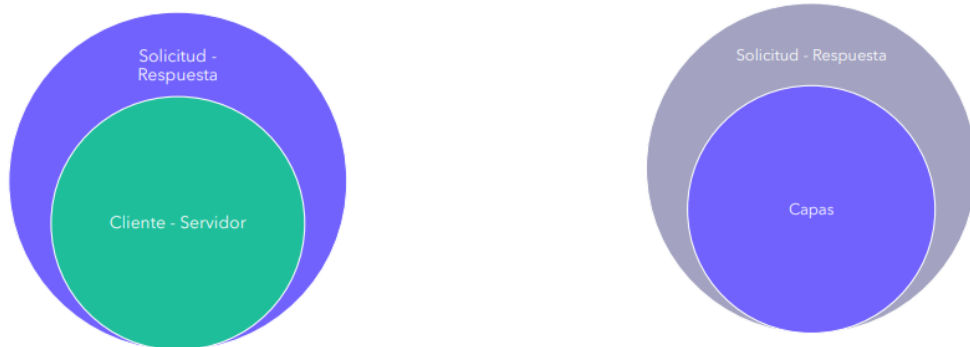
Canary Deployment

El canary no despliega toda la solución en todos los nodos, sino que lo hace solo en una parte, entonces una parte de usuarios finales está en la nueva versión mientras que otros mantienen la vieja versión. Se va aumentando el porcentaje de la nueva versión, esto provoca que no tengamos caída del servicio, mantiene la disponibilidad. A su vez, permite que se pueda volver a la versión anterior (en caso de que no esté toda la nueva versión desplegada).



Patrones Arquitectónicos tipo “Call & Return”

El patrón Cliente – Servidor es de tipo solicitud – respuesta, la arquitectura en capas también, el POO es de tipo solicitud – respuesta, el protocolo HTTP, etc.



Cliente – Servidor



La limitación de esto se basa en que se necesita una SOLICITUD para tener una RESPUESTA, es decir que el servidor debe esperar una solicitud para que el cliente obtenga una respuesta.

Para cambiar esto, aparecen otros patrones.

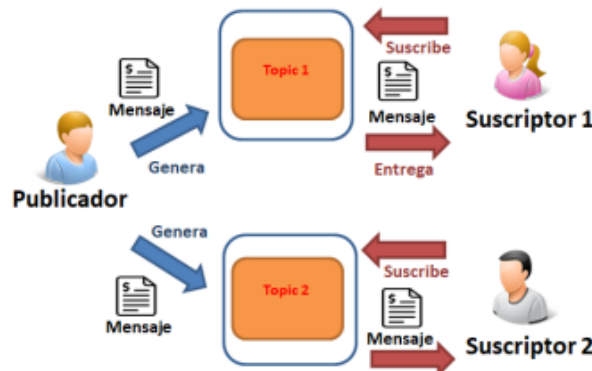
Patrón Suscripción de Eventos



La solicitud que realiza el cliente es la suscripción a eventos. Cuando surgen los eventos, el servidor le avisa al cliente (notificación push).

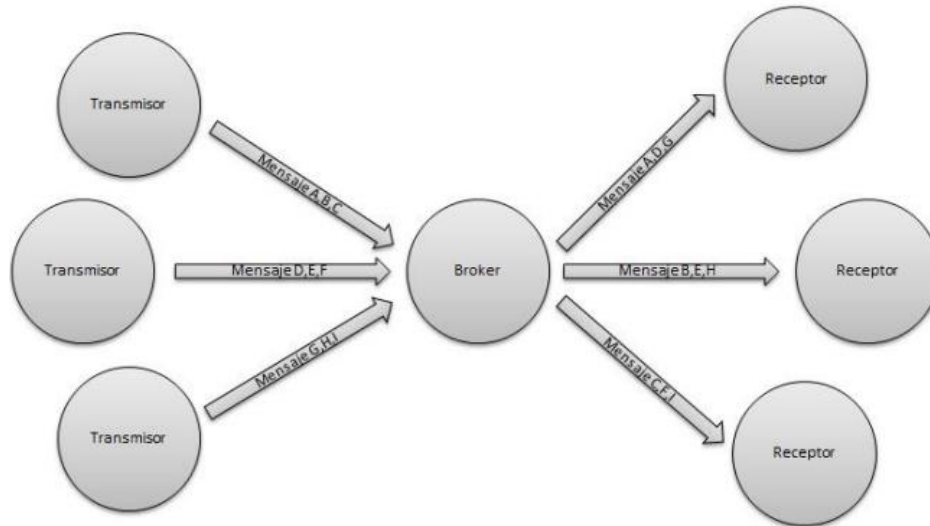
Patrón Publicador – Suscriptor

- Este patrón es de tipo Mensajería
- Define, el lugar de cliente – servidor, al suscriptor y el publicador
- Define un Topic (tema) al cual me puedo suscribir para ser notificado
- Minimiza el acoplamiento (Cliente - Servidor)
 - El componente Suscriptor y el componente Publicador no van a estar conectados entre sí sino que hay un componente intermedio



Patrón Broker

- El bróker implementa un Publicador – Suscriptor
- El bróker es un intermediario del Publicador – Suscriptor
- Iguala a los participantes, ambos pueden iniciar la comunicación (permite a los componentes alternar los roles)
- Resuelve los problemas de Firewall
- Servicios de Autenticación → El cliente gestiona la autenticación



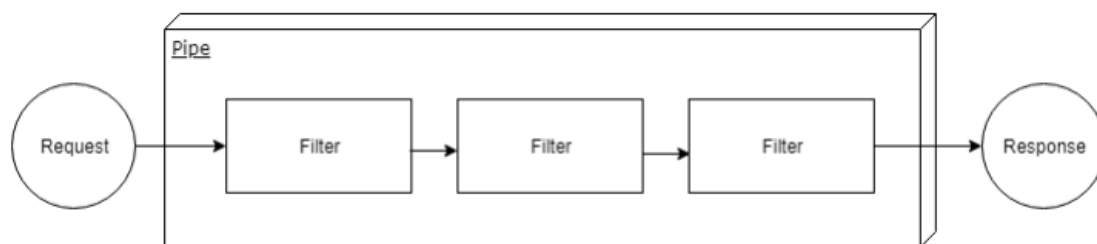
Patrones Arquitectónicos tipo “Flujo de Datos”

Stream → flujo de datos/respuesta

Pipe & Filters

Descompone una tarea compleja en una serie de elementos/componentes independientes que se pueden volver a utilizar. Cada uno de esos elementos/componentes hace un proceso particular y se los llama FILTERS.

Cada conexión entre cada elemento es un PIPE, esto quiere decir que conecta el stream de datos con el otro filter. Este patrón puede mejorar el rendimiento, la escalabilidad y la capacidad de reutilización, ya que, al romper una tarea en partes más pequeñas, permite que cada una se escale e implemente por separado.



Lo importante del Pipe & Filter es que se rompe ese “monolito” de procesamiento de datos y se lleva ese procesamiento a servicios más pequeños, reusables y escalables.

P2P – Peer to Peer

- Cualquier par (peer) es libre de comunicarse con cualquier otro par, sin necesidad de utilizar un servidor central.
- Los “pares” o compañeros suelen localizarse entre sí mediante el intercambio de listas de forma automática.
- Cada par es capaz de actuar tanto como cliente, puede realizar solicitudes, o como servidor, puede atender solicitudes, siendo cliente o servidor, ambos al mismo tiempo.

La arquitectura P2P es como si uniéramos el Cliente y el Servidor en una sola aplicación, lo que permite conectarse a otras computadoras de la red para consumir los recursos expuestos por los otros nodos de la red.

Algunos objetivos del P2P son:

- ✓ **Descentralización:** no hay un nodo central que maneje nodos secundarios, todos pueden dar respuesta a cualquier solicitud.
- ✓ **Replicación:** diseñada para el despliegue en gran cantidad de nodos, permite la replicación en diversos centros de datos.

- ✓ **Tolerancia a fallos:** cuando un nodo no funciona, la base de datos sigue trabajando normalmente con los restantes nodos, se puede añadir y quitar nodos sin detener el servicio.
- ✓ **Escalabilidad:** para escalar a nivel base de datos, solo se deben instanciar más nodos.

Clase 01/11

Integración de Sistemas

Primero, ¿por qué se integran los sistemas?

- Para reusar una funcionalidad de otro componente
- Tenemos una restricción que me obliga a integrarme con otro sistema
 - Necesidad vinculada al negocio
- Hay algo que no podemos resolver, pero lo resuelve un componente externo o al marco legal
- Por los datos, necesitamos los datos de otro sistema y ese sistema necesita mis datos (muy importante)
- Aumentar la cohesión

Pero ¿qué es integrar un sistema?

- ➔ Integrar un sistema consiste en establecer una comunicación entre componentes, con el fin de compartir datos y funcionalidades; además abrimos el universo cerrado sobre el que fue creado nuestro sistema y aumentamos la cohesión.

¿INTEGRACIÓN SINCRÓNICA O ASINCRÓNICA?

Esta decisión dependerá de las necesidades del negocio y/o de las restricciones técnicas. En general, el negocio es quien nos va a obligar a tomar esta decisión.

- ➔ También se debe tener en cuenta lo que sucede y lo que queremos que se vea, por ejemplo: un pago en MP, parece instantáneo, pero no está implementado de esa manera.

Ejemplos:

Necesidad de Negocio

Se tiene un E-Commerce que necesita procesar pagos a través de la plataforma.

Necesidades vinculadas al marco legal

Se tiene un sistema que necesita realizar facturas electrónicas conforme a la reglamentación actual.

- ¿Haríamos esta facturación sincrónica o asincrónica?
 - Como no sabemos cuánto tardará el proceso de generar la facturación, conviene una facturación asincrónica, ya que, teniendo en cuenta que es un E-Commerce, el cliente “necesita” una respuesta rápida para que no abandone la compra.

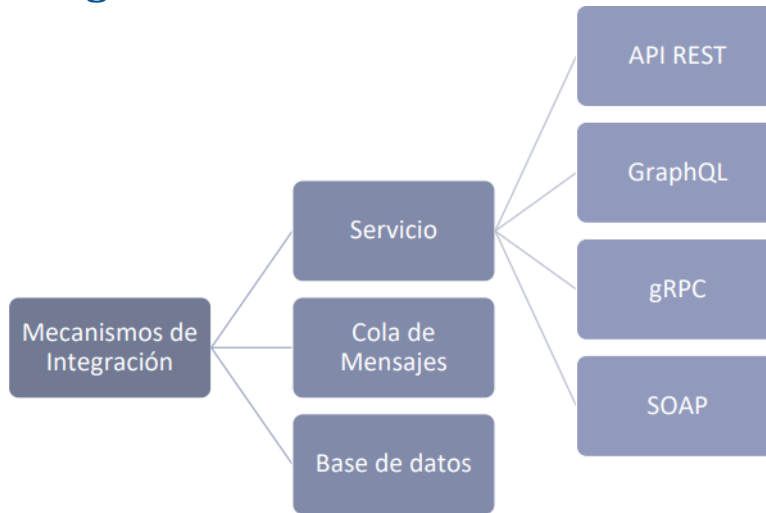
Sistema de Telepase

Sistema de Telepase - ¿Qué integraciones involucra?

- Sincronización Cabinas de Peaje <-> Unidad Central
- Unidad Central -> Gateway de Pago (Generación Pagos)
- Gateway de Pago -> Unidad Central (Bajas por Falta de Pago)
- Gateway de Pago <-> Tarjeta de Crédito
- Tarjeta de Crédito <-> Entidad Bancaria

Mientras más intermediarios tengo, aumentan los costos

Mecanismos de Integración



En los mecanismos de integración tenemos vinculado:

- Servicios:
 - API REST (u otro tipo)
 - GraphQL
 - gRPC
 - SOAP
- Cola de Mensajes
- Base de Datos (compartida)

¿Qué es un Web Service?

Un WEB SERVICE es un término que se utiliza para hacer referencia a un componente de software interoperable de máquina a máquina, el cual está alojado en una ubicación a la que se puede acceder mediante una red.

El WEB SERVICE posee una interfaz, oculta los detalles de implementación para que se pueda utilizar independientemente de la plataforma y del stack tecnológico, se puede utilizar solo o con otros servicios web y busca implementar tecnología cruzada, orientada a componentes y poco acopladas.

¿Qué es una API?

Una API es una **interfaz de programación de aplicaciones**, es decir que es un conjunto de herramientas, definiciones y protocolos que se usa para diseñar e integrar aplicaciones.

Una API permite que un producto o servicio se comuniquen con otros productos y servicios, sin tener que saber cómo se implementa cada uno.

Existen distintos tipos de APIs:

- Local
 - Proporciona servicios de sistema operativo, bibliotecas o middleware
- Web
 - Proporciona servicios entre componentes distribuidos y sobre el protocolo HTTP
- Programa
 - Proporciona la posibilidad de ejecución de código de forma remota entre componentes distribuidos (RPC)

Alcance de APIs:

- Public (Pública) → Todos pueden usarlas
- Partner (Para Terceros) → Yo hago una API para que otro consuma ese servicio (consumen mi API)
- Private (Interna) → APIs que solo se consumen dentro de mi sistema

APIs de tipo Web:

- REST
- SOAP
- gRPC
- GraphQL

REST

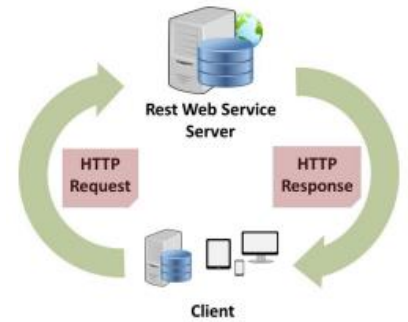
REST o Representational State Transfer es un ESTILO O TÉCNICA de Arquitectura a la hora de realizar una comunicación entre cliente y servidor.

REST se apoya sobre el Protocolo HTTP.

Se crea una solicitud HTTP, creamos una REQUEST (petición) que viaja al servidor, la cual contiene toda la información que necesitamos y solo esperamos una RESPONSE, es decir una respuesta a nuestra request.

REST se apoya en los métodos básicos de HTTP:

- POST: Para crear recursos nuevos (para todo, en general)
- GET: Para obtener un listado o un recurso en concreto
- PUT: Para modificar un recurso
- PATCH: Para modificar un recurso (en forma parcial)
- DELETE: Para borrar un recurso



Características

- Todos los recursos de manipulan mediante un identificador único de recursos o URI
- Las REQUEST y/o RESPONSE utilizan el formato de intercambio JSON/XML

Códigos de Estados HTTP

- ✓ **1xx:** Respuestas informativas. Indica que la petición ha sido recibida y se está procesando.
- ✓ **2xx:** Respuestas correctas. La petición ha sido procesada correctamente.
- ✓ **3xx:** Respuestas de redirección. Indica que el cliente necesita realizar más acciones para finalizar la petición.
- ✓ **4xx:** Errores causados por el cliente. El cliente hizo algo mal.
- ✓ **5xx:** Errores causados por el servidor. Fallo en el servidor.

SOAP

Los **SOAP (Web Services)** son servicios que basan su comunicación bajo el protocolo **SOAP (Simple Object Access Protocol)**.

- SOAP es un protocolo estándar que define la comunicación entre dos componentes mediante el intercambio de datos XML

Funcionan por lo general a través del protocolo HTTP, sin embargo, SOAP no está limitado al protocolo HTTP, sino que puede ser enviado por FTP, POP3, SMTP, TCP, colas de mensajería, etc.

Utiliza **WSDL (Web Services Description Language)**, el cual se usa para generar una interfaz del Servicio Web, la cual será un punto de entrada. Al **punto de entrada** se lo conoce como archivo wsdl o contrato, el cual es accesible en una red por medio de una URL.

Estructura

Solicitud

```
<?xml version='1.0' encoding='UTF-8'>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  </soap:Header>
  <soapenv:Body>
    <aplicarVentaRequest xmlns="http://elements.blogsoa.com/Productos">
      <codigoProducto>1KL9796Y</codigoProducto>
      <cantidadProducto>2</cantidadProducto>
      <almacen>Centro</almacen>
      <precioTotal>7600.72</precioTotal>
    </aplicarVentaRequest>
  </soapenv:Body>
</soapenv:Envelope>
```

Respuesta

```
<?xml version='1.0' encoding='UTF-8'>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  </soap:Header>
  <soapenv:Body>
    <aplicarVentaResponse xmlns="http://elements.blogsoa.com/Productos">
      <codigo>0</codigo>
      <descripcion>La venta se ha registrado exitosamente</descripcion>
      <numeroOrden>356-MNC298</numeroOrden>
    </aplicarVentaResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

- WDSL: En el archivo WDSL se describe todo el servicio web, las operaciones disponibles y las estructuras de datos de los mensajes de cada operación.
- XSD: En el archivo XSD se describen las estructuras de datos de los mensajes de cada operación.

Característica	REST	SOAP
Formato de Intercambio	XML / JSON	XML
Llamada/Respuesta	Sincrónica	Asincrónica/Sincrónica
Interfaz	Fijas y Simples	Dinámicas y Complejas
Descripción del Servicio	Documentación	WSDL
Acoplamiento	Bajo	Alto
Protocolos	HTTP	HTTP / SMTP / TCP
Seguridad	HTTPS	WS-Security
Manejo de Estados	Sin Estados (Stateless)	Con Estado
Tipo de Aplicación	Web - Mobile	Enterprise
Público	General	Específico

SOAP se usa para aplicaciones con estado y específicas. Puede ser asincrónica también.

gRPC

gRPC es un **framework de llamada a procedimiento remoto (RPC)** de alto rendimiento que puede ejecutarse en cualquier entorno.

- Puede **conectar servicios de manera eficiente** en y entre centros de datos con soporte para balanceo de carga, rastreo, verificación de estado y autenticación;
- Es aplicable para conectar dispositivos, aplicaciones móviles y navegadores a servicios de backend.
- Transporta el flujo de datos mediante HTTP/2;
- Gestiona la estructura y la distribución de los datos mediante los **Protocol Buffers**
 - Los Protocol Buffers son archivos de texto plano con la extensión '.proto'
- Los flujos se transmiten en datos binarios compactos que surgen en la serialización y la deserialización

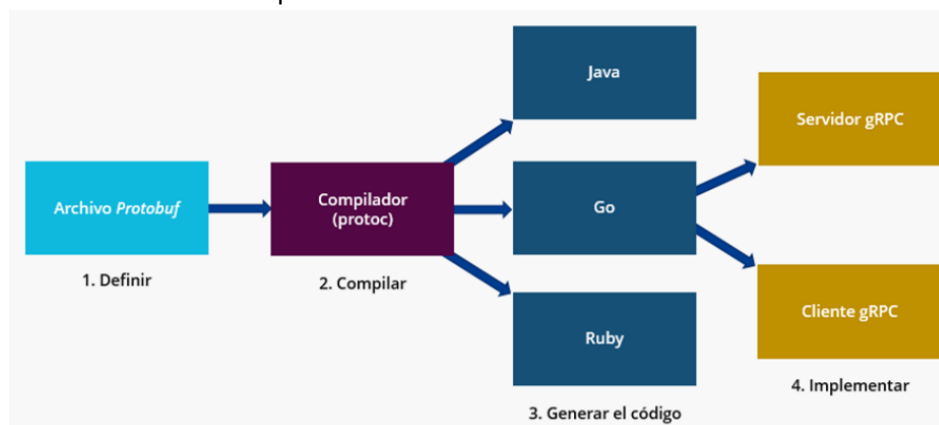
PROTOBUF

Protocol Buffers (abreviado como Protobuf) es un formato de intercambio de datos desarrollado, en principio, para el uso interno en Google y luego lanzado en 2008 como proyecto de código abierto.

- Este formato binario permite a las aplicaciones almacenar e intercambiar datos estructurados fácilmente, incluso si los programas están compilados en diferentes lenguajes.
- Centrado en la Comunicación entre Servicios/Microservicios
- Soporte Multilenguaje
- Flujo de Intercambio más pequeño, muy eficiente
- Streaming Bidireccional

Pasos para la implementación:

1. Definición del contrato de servicio
2. Generación del código gRPC del archivo '.proto' (en el lenguaje deseado)
3. Implementación del servidor en el lenguaje deseado.
4. Creación del stub del cliente que llama al servicio.



GraphQL

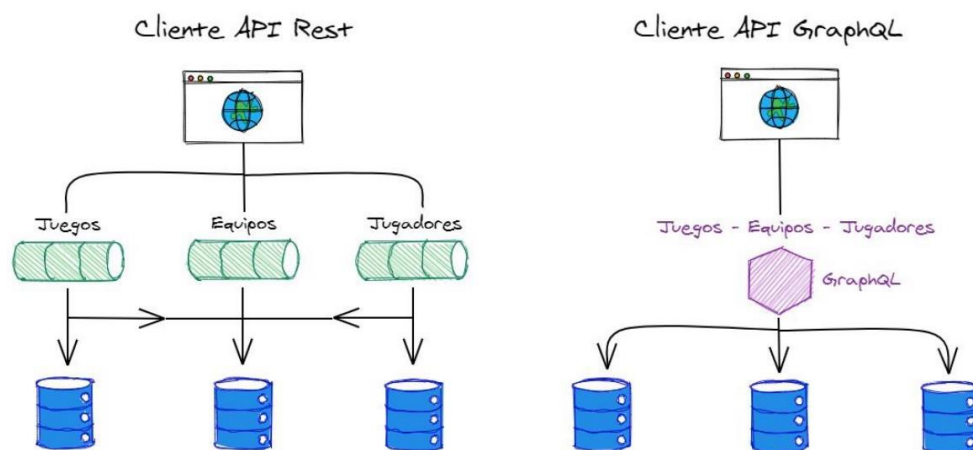
GraphQL es un **lenguaje de consulta para API** y un tiempo de ejecución del servidor para las interfaces de programación de aplicaciones (API).

- Brinda a los clientes exactamente los datos que solicitan y nada más
- Proporciona una descripción completa y comprensible de los datos en su API
- Facilita la evolución de las API con el tiempo y habilita poderosas herramientas para los desarrolladores

Ventajas de GraphQL:

- Simplifica la consulta
- Se apoya sobre HTTP
- Esquema tipado

GraphQL vs. API REST



COMPARATIVA

REST	SOAP	GraphQL	gRPC
Sincrónico HTTP	Sincrónico/Asincrónico HTTP, FTP, SMTP	Sincrónico/Asincrónico HTTP, AMQP, MQTT	Sincrónico/Asincrónico HTTP/2
Basado en HTTP (Estados, Métodos, URI)	Basado en intercambio de Mensajes	Basado en intercambio de Mensajes	Basado en intercambio de Mensajes
Múltiples Estándares (Open API, RAML)	Estándar de Especificación de Esquema mediante XML	Estándar de Especificación de Esquema mediante IDL/SDL	Estándar de Especificación de Protobuf
Formato de intercambio JSON	Formato de intercambio XML	Formato de intercambio JSON	Formato de intercambio Serializado (protobuf)

COLA DE MENSAJES

Una **COLA DE MENSAJES** es una **forma de comunicación asíncrona de servicio a servicio** que se usa generalmente en arquitecturas de microservicios y sin servidor.

Los **mensajes** se almacenan en la cola hasta que se procesan y eliminan. Cada mensaje se procesa una sola vez, por un solo consumidor.

- Las colas de mensajes se pueden usar para desacoplar procesos pesados, para acumular trabajo y para clasificar cargas de trabajo



- Proporcionan la comunicación y la coordinación para aplicaciones distribuidas
- Pueden simplificar la escritura de código para aplicaciones desacopladas
- Permite a diferentes partes de un sistema comunicarse y procesar las operaciones de forma asíncrona
- Ofrece un búfer ligero que almacena temporalmente los mensajes (cada mensaje lo consume un solo worker)
- Ofrece puntos de enlace que permiten a los componentes de software conectarse a la cola para enviar y recibir mensajes
- Los mensajes suelen ser pequeños y pueden ser cosas como solicitudes, respuestas, mensajes de error o datos
- Para enviar un mensaje, el PRODUCTOR (un componente) añade un mensaje a la cola
- El mensaje se almacena en la cola hasta que otro componente, llamado CONSUMIDOR, lo recupera y hace algo con él
- **Muchos productores y consumidores pueden utilizar la cola, pero solo un consumidor procesa cada mensaje una sola vez** (uno a uno). Ésta es la diferencia con el Broker.
- Cuando más de un consumidor debe procesar un mensaje, las colas de mensajes se pueden combinar implementando el patrón publicación-suscripción → en este patrón hay tópicos donde varios consumidores se suscriben al mismo tópico (abanico de salida)

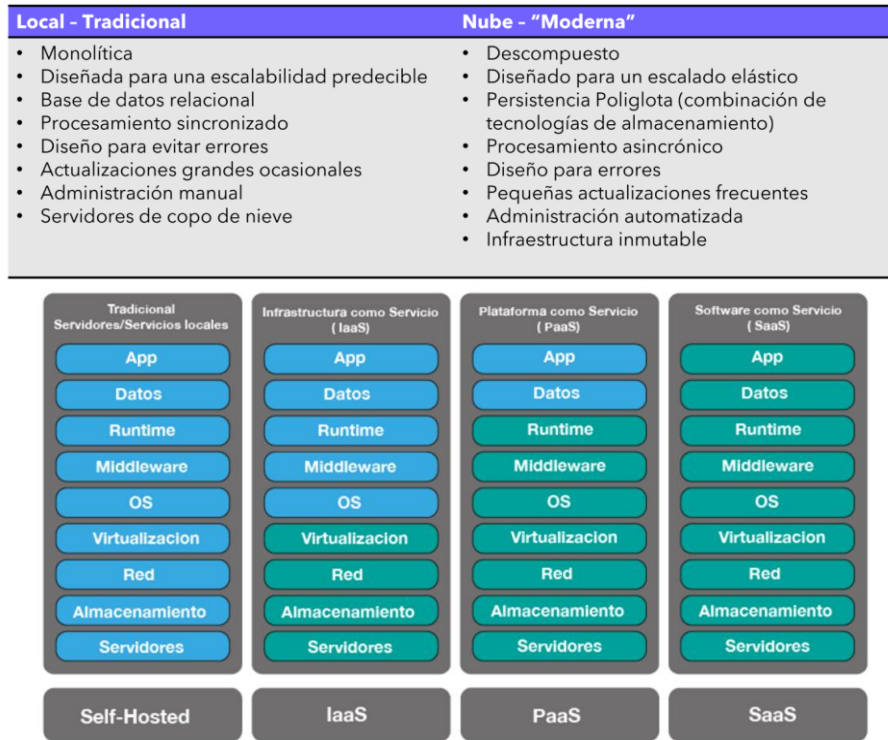
BASES DE DATOS COMPARTIDAS

- Se utiliza una Base de Datos central para integrar los componentes
- Se centra en la integración de datos, no de uso de funcionalidad
- Permite integración asíncrona y síncrona
 - En la integración asíncrona se debería hacer un pulleo cada cierto tiempo
- Es simple de implementar
- Cada componente puede estar resuelto en diferentes stacks tecnológicos
- Cada componente de lógica de negocio es independiente en su disponibilidad
- Puede generar problemas de performance (siempre que hablamos de pulleo)
- El acoplamiento a los modelos puede ser elevada (porque otro componente conoce la estructura de mis entidades/modelos)
- Es difícil de cambiar a nivel estructural (rompo componentes fuertemente acoplados)
- Se debe considerar la seguridad (CIA) de los datos
- El uso entre sistemas que estén al poder de desarrollo y de la misma organización

Aplicaciones en la Nube vs. Tradicionales

Aplicativos en la Nube → Desafíos

- Cambia la forma en que se diseñan las aplicaciones
- En lugar de ser “monolitos”, las aplicaciones se descomponen en servicios menores y descentralizados
- Los servicios se comunican a través de API o mediante el uso de eventos o de mensajería asíncrona
- Las aplicaciones se escalan horizontalmente, agregando nuevas instancias, tal y como exigen las necesidades
- El estado de las aplicaciones se distribuye (por tener el listado distribuido)
- Las operaciones se realizan en paralelo y de forma asíncrona (por haber crecido horizontalmente)
- Las aplicaciones deben ser resistentes cuando se produzcan errores
- Las implementaciones deben estar automatizadas (compilo y se despliega sola la aplicación) y ser predecibles
- La supervisión y la telemetría son fundamentales para obtener una visión general del sistema



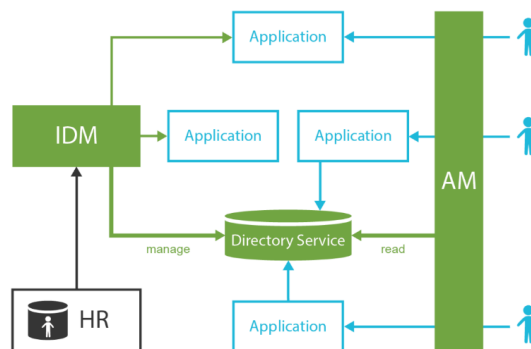
Componentes

IAM / IDM

- Es un componente que gestiona las identidades y los accesos
- IAM
 - **Acceso:** Define Estrategias para el acceso a través de varias tecnologías, como contraseñas, biometría, autenticación multifactor y otras identidades digitales
- IDM
 - **Identidad:** Se centra más en la identidad de un usuario (o nombre de usuario), sus roles y permisos, y los grupos a los que pertenece

Autenticación vs. Autorización

- La AUTENTICACIÓN confirma que los usuarios son quienes dicen ser, validando la identidad del usuario
- La AUTORIZACIÓN otorga a los usuarios autenticados los permisos para acceder a uno o varios recursos



SSO

El **SSO** es un componente que permite tener acceso a múltiples aplicaciones ingresando mediante un solo acceso a una cuenta.

- Se desea evitar el ingreso repetitivo de las credenciales de una cuenta cada vez que el usuario se desconecte del servicio
- Para los usuarios supone una gran comodidad ya que identificándose solo una vez es posible mantener la sesión válida para el resto de las aplicaciones que hacen uso del SSO
- Acelera el acceso de los usuarios a sus aplicaciones
- Reduce la carga de memorizar diversas contraseñas

IDAÑEZ, LUCIA MARÍA 🐱

- Fácil de implementar y conectar a nuevas fuentes de datos
- Al fallar SSO se pierde acceso a todos los sistemas relacionados (Desventaja)
- Suplantación de identidades

CDN

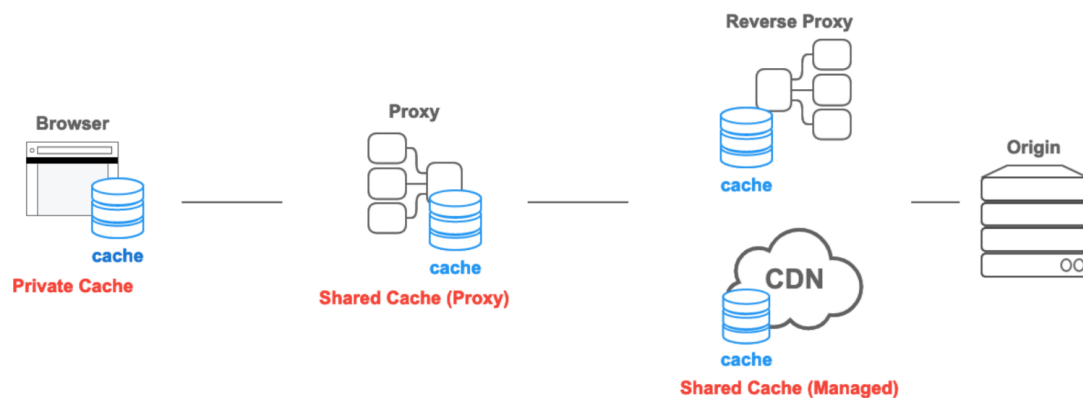
Una **red de entrega de contenido (CDN)** está formada por un grupo de servidores distribuidos geográficamente que trabajan juntos para ofrecer una entrega rápida de contenido de Internet.

- Permite la transferencia rápida de activos necesarios para cargar contenido de Internet, incluidas páginas HTML, archivos js, css, imágenes y vídeos.
- Puede ayudar a proteger sitios web contra algunos ataques maliciosos comunes, como los ataques de denegación de servicio distribuido (DDOS).
- Ventajas:
 - Mejora de los tiempos de carga del sitio web
 - Reducción de los costos de ancho de banda
 - Aumento de la disponibilidad de contenido y la redundancia
 - Mayor seguridad del sitio web

CACHÉ

- Almacena una respuesta asociada con una solicitud y reutiliza la respuesta almacenada para solicitudes posteriores
- Hay varias ventajas de reusabilidad:
 - Dado que no es necesario enviar la solicitud al servidor origen, mientras más cerca esté el cliente y la caché, más rápida será la respuesta
- El servidor de origen no necesita procesar la solicitud, por lo tanto, reduce la carga en el servidor.
- El funcionamiento adecuado de la memoria caché es fundamental para la salud del sistema.
- Características:
 - Privada (Usuario)
 - Compartida
 - Proxy Reverso
 - CDN
 - Tiempo de Expiración

Arquitectura con cache:



Y con esto, se termina el resumen del segundo cuatri de Diseño de Sistemas ¡Éxitos en el parcial!