Programación Funcional

https://perficient.udemy.com/course/functional-programming-and-reactive-programming-in-java

1 - 14

Lambda:

Beneficios:

- Se puede pasar el comportamiento de una función como parámetro
- Programación declarativa: Se indica que hacer en vez de como hacerlo. La prog. Funcional es una parte pequeña de esta.

Interfaces Funcionales:

- Predicate: Test condition (T->boolean)
- Consumer: Takes/Consumes something (T->void)
- Function: Takes something returns something (T->R)
- Supplier: Supplies/returns something (()->T)
- Unary Operator: Takes something returns something like the entry (T -> T)
- Binary Operator: Takes 2 things and returns something like the entries ((T, T)->T)
- BiPredicate: Test condition with two arguments ((L, R) -> boolean)
- BiConsumer: Takes/Consumes two arguments ((T, U) -> void)
- BiFunction: Takes two arguments returns something ((T, U) -> R)

Generics: <T, R>

Method References:

- object :: instanceMethod
- Class :: staticMethod
- Class :: instanceMethod
- Class :: new

Programación Funcional:

Es un paradigma de programación basado en funciones matemáticas puras, donde se evita cambiar estados o información variante.

Es una forma de prog. declarativa que se enfoca en el "que resolver" en vez de como resolver.

Se basa en **Lambda Calculus** y se pueden pasar funciones a funciones por parámetro.

Conceptos:

- **Functions as first class citizens:** Se pasan funciones por parámetro a otras funciones para obtener otra función y asignando esta a otras variables o estructuras de datos
- **Pure functions:**
    1. La salida de una función pura depende solo de sus parámetros de entrada y su algoritmo interno
    2. No tiene efectos secundarios, no escribe o modifica archivos y no consume web services ni bases de datos. No modifica nada externo. Siempre retorna la misma salida para las mismas entradas
    3. Debe tener una salida
    4. Claridad de pensamiento
    5. Fácil de razonar
    6. Nunca se modifican estado compartidos o variables
- **Higher order functions:** Son funciones que reciben funciones por parámetro y/o retornan funciones
- **No side effects:** No se afecta ni se interactúa con algo externo
- **Referential Transparency:** Es una propiedad de una función, variable o expresión donde la expresión puede ser reemplazada por su valor (evaluado) sin afectar el comportamiento del programa

Técnicas:

- **Function Chaining:**
  Llamar funciones sobre un método, donde cada función permite encadenar los llamados a otras funciones y así simplificar código. Se suele colocar en el método concatenado un lambda con la ejecución del método del objeto (interfaz) actual y la del objeto (interfaz) recibido, retornando el lambda que representa la Interfaz funcional con lo ejecutado de ambos objetos (interfaz).
  En otras ocasiones se ejecuta lo del obj actual y el resultado lo pasa por parámetro a la ejecución del otro obj.
  Por lo que se ejecuta siempre primero el actual y luego el otro.
- **Functional Composition:** Es similar a la cadena, pero se ejecuta primero el otro obj y luego el actual.
- **Closures:** Es una función la cual hace referencia a variables libres en su contexto léxico
- **Currying:** La descomposición de un método en varios métodos que reciben entradas simples pero cada método genera el siguiente método hasta que un método entrega la respuesta simple.

- **Lazy Evaluation:** Al dejar las acciones en memoria, se pueden inicializar variables o crear instancias solo cuando se necesiten. Tambien evaluar condiciones solo cuando sea necesario
- **Tail Call Optimization:** Es un tipo específico de recursión. En particular, se dice que una función recursiva usa tail recursion si la llamada a sí misma ocurre como la última acción de la función.
  En las recursiones de maneja una pila de llamados que con recursión normal es habitual que se desborde, pero con la optimización el compilador es capaz de reemplazarla con un simple bucle, aunque lastimosamente Java no la incluye.

Dessign Pattern Functionally

- **Iterator Design Pattern:**

  Acceder a los elementos de un contenedor sin exponer la estructura interna de este. Permite tener una manera de acceder a los elementos de una colección de forma secuencial sin necesidad de conocer su representación interna

- **Strategy Design Pattern:**
  Es usado cuando se tienen multiples soluciones o algoritmos para solucionar una tarea en específico y el cliente decide la implementación a ser usada en tiempo de ejecución.
  Ej: filter
- **Decorator Pattern:**
  Es usado para modificar la funcionalidad de un objeto en tiempo de ejecución sin afectar otras instancias de la misma clase
- **Creating fluent Interfaces:**
  Provee una interfaz fluida y facil de leer, que suele imitar el dominio de un lenguaje en específico. Al usar este patrón, el codigo puede ser leido de forma más sencilla y similar al del lenguaje humano.
  Para simplificar código se pueden encadenar métodos haciendo que estos retornen el mismo objeto para seguir invocando métodos de corrido.
  Se ve mejor, más elegante y conciso.
- **Factory Design Pattern:**
  Es un patrón creacional que consiste en crear objetos sin exponer instanciación lógica.
  Es una forma de instanciar una clase dentro de un método designado conocido como Factory Method.
  Factory es un objeto que le es posible crear otros objetos
- **Builder Design Pattern:**

Se trata de proveer flexibilidad al momento de crear un objeto. La idea es aislar la construcción de un objeto complejo (muchos argumentos en su constructor) de su representación

- **Command Design Pattern:**
  Encapsular una petición como un objeto, parametrizar clientes con diferentes peticiones y realizar operaciones correspondientes

En programación funcional no se modifica el estado de los objetos por lo que se retorna una nueva instancia con dicha modificación

Streams:

Se usan Collections o List que almacenan datos y se procesan los datos al recorrer estas colecciones normalmente mediante iteradores.

Es una secuencia de elementos de una fuente que soporta el procesamiento de dato mediante operaciones para luego contener los datos resultantes

Características:

- Declarativo: más conciso y entendible
- Flexible: No necesariamente se maneja sobre una colección, tambien sobre archivos o input/output resourses
- Paralelizable: Para procesar grandes cantidades de datos en colecciones se pueden procesar en paralelo

¿Como funciona?

1. El stream toma la información de la fuente
2. Realiza el procesamiento
3. Retorna la información en un contenedor que el usuario elija o simplemente consume los datos

Stream Pipeline: Compuesto por:

1. Una Fuente
2. Cero o más operaciones intermedias
3. Una operación terminal

Streams no son contenedores de datos

Solo se pueden usar una vez, son inmutables y se vacían en cuanto se realiza una operación. Por lo que las funciones como filter retornan un nuevo Stream.

Funciones de alto nivel:

1. Filter: Funcion equivalente a un if y recibe un predicado
2. Map: Realiza una operación y reemplaza cada elemento de la lista. No se debe modificar el estado de un objeto mediante un map, por lo que se da una excepción de modificación concurrente.
3. Reduce: Realiza una operación acumulativa con los valores del stream, con un valor identidad, el cual es un valor que al ser operado con otro elemento da como resultado el otro elemento.

Las operaciones intermedias de Streams son Lazy, pues se ejecutan solo cuando se realiza una colección del stream resultante o una operación Terminal

Streams Numericos: Evita el encapsulamiento de int, long o double

- IntStream
- LongStream
- DoubleStream

Map retorna un Stream de Objects

MapToDouble retorna un Stream de doubles primitivos (sin el wrapper Double)

FlatMap: recibe una función de identidad (que retorna el mismo elemento que recibe). Une los Streams dentro de otro Stream, por lo que retorna un Stream (Flatered Stream) que unifica los contenidos de los Streams internos.

Parallel Streams: Multiprocesamiento, aumenta la velocidad de procesamiento de los elementos del stream.

Streams son:

- Stateless

- No interfiere en el estado de la fuente de recursos, no modifica a la fuente mediante las operaciones
- Associative: El resultado de las operaciones no debe ser afectado por el orden de los datos manejados

Stateless Operations:

No requiere ningun tipo de información del exterior

Stateful Operations:

Usa información externa

Hay un problema al usar la función skip (statefull) cuando se usa paralelismo en un stream, pues es dificil saber cuál es el primer elemento al ser procesados por diferentes núcleos. Por lo que hay que evitar usar operaciones Statefull

Configuración de paralelismo:

- Common Fork-Join Pool: Usado por todas las operaciones paralelas de streams y toma todos los procesadores disponibles

Custom Streams

Spliterator: Es una interfaz que se usa cuando se desea usar un stream sobre una fuente personalizada y accede a los datos.

Es un objeto especial sobre el cual se contruye un stream. Tambien se puede usar sobre collections, lo cual es lo que ocurre por debajo al crear un stream.

Contiene 4 métodos abstractos y otros por defecto.

- **tryAdvance**: Si un elemento existe ejecuta la acción sobre este y retorna un booleano
- **trySplit**: Divide los elementos y es importante cuando se usa procesamiento paralelo
- **estimateSize**: retorna la cantidad de elementos que se encuentran por un foreach transversal

- **characteristics**: retorna un conjunto de caracteristicas del spliter. El resultado es una operación OR entre las características (en representación binaria). Las características pueden ser:
    1. **Ordered**: Importa el orden de los elementos
    2. **Distinct**: No hay duplicados en el spliterator
    3. **Sorted**: El stream está ordenado
    4. **Sized**: Se conoce el tamaño del spliterator
    5. **NonNull**: No existen valores nulos
    6. **Inmutable**: El stream es inmutable
    7. **Concurrent**: Stream construido en una estructura concurrida como un hashmap concurrido
    8. **SubSized**: Se conoce el tamaño y se refiere a que los spliterator resultantes del método trySplit van a ser adaptados al tamaño cuando se paralelise

Las caracteristicas sirven para optimizar código de manera que no se ejecuten operaciones repetitivas e innecesarias.

Collectors

Sirven para procesar grandes cantidades de información en memoria.

Recolecta la información de un stream y la encapsula en una colección.

Todos sus métodos son estáticos y retornan un Collector en específico

Existen Collectors.to: List, Set, Map

También existe .toUnmodifiable para que la estructura no pueda ser modificada

Existe un .toCollection que recibe un CollectionSupplier y transforma a cualquier collection.

Custom Collectors

Collector: Es una interfaz con 5 métodos abstractos

- supplier: retorna un Supplier
- accumulator: retorna un BiConsumer
- combiner: retorna un BinaryOperator
- finisher: retorna un Function

- characteristics: retorna un conjunto con las carácteristicas

Métodos de Collections:

- Traversing: Iterar con un ciclo for normal o mejorado de forma imperativa
- Filtering: Elementos de filtrado o de busqueda
- Sorting: Ordenar elementos
- Mapping: Mapear elementos para convertirlos en otros
- Reducing: Operar los elementos para retornar un único valor

SQL Bootcamp

https://perficient.udemy.com/course/the-complete-sql-bootcamp/learn

Statements Fundamentales:

- Select
- Distinct: Valores únicos
- Count
- Where
- Order By: ASC/DESC
- Limit: Limita las filas que se retornan
- Not
- Between _ and _
- In: _ IN (option1, option2, … , optionn)
- Like: case-sensitive
    - **%:** Cualquier secuencia de caracteres
    - **_:** Cualquier carácter (solo uno)
- ILike: case-insensitive

Ejemplos:

```
Select DISTINCT(rating) From film;


Select Count(*) From film
Where rental_rate > 4 And replacement_cost >= 19.99 And rating = 'R';
```

```
Select store_id, first_name, last_name From customer
ORDER BY store_id DESC, first_name ASC;

Select * From payment
Order by payment_date Desc
Limit 5;

Select * From payment
Where payment_date BETWEEN '2007-02-01' and '2007-02-15';

Select * From payment
Where amount in (0.99, 1.98, 1.99)
Order by amount;

Select * From customer
Where first_name Like '%er%' and last_name ILike 'j%';
```

Group By Statements

Aggregation functions: **SOLO en el Select o Having**

- **AVG:** promedio, como valor punto flotante. Se puede usar Round para especificar la precisión decimal
- **COUNT:** cantidad de filas
- **MAX:** máximo valor
- **MIN:** mínimo valor
- **SUM:** suma de los valores

Se pueden combinar columnas que retornen un único valor mediante estas funciones pero no se puede combinar con columnas que no sean de este estilo.

Ejemplo:

Select min(replacement_cost), max(replacement_cost), Round(avg(replacement_cost), 2), SUM(replacement_cost)

from film;

Group By:

Debe ir después de un From o un Where

En el select debe ir únicamente un aggregate function o una (o varias) columna indicada en el llamado Group By

Los aggregate functions se ejecutan solo cuando se ejecuta el Group By, por lo que no se puede hacer un where de ese aggregate function. Para eso se usa el **Having**

Ejemplo:

Select customer_id, SUM(amount) as asum, Count(amount) from payment

Group by customer_id

Order by asum DESC

Limit 5;

Select staff_id, customer_id, SUM(amount) from payment

Group by staff_id, customer_id

Order by staff_id, customer_id

Limit 5;

Select Date(payment_date) as p_date, SUM(amount) as amount_sum from payment

Group by p_date

Order by amount_sum DESC;

//Date extrae solo la parte de día, mes y año de un timestamp

Having:

Se usa después de un Group By.

Permite realizar validaciones de aggregate functions, pero no permite la comparación de columnas renombradas.

Ejemplo:

Select customer_id, SUM(amount) as total_spend from payment

Where customer_id Not In (184, 87, 477)

GROUP BY customer_id

HAVING SUM(amount) > 150

Order By total_spend DESC;

Assessment Test 1

1. Return the customer IDs of customers who have spent at least $110 with the staff member who has an ID of 2.

   Select customer_id from payment
   Where staff_id = 2
   GROUP By customer_id
   HAVING SUM(amount) > 110;

2. How many films begin with the letter J?

   Select Count(*) from film
   Where title Like 'J%';

3. What customer has the highest customer ID number whose name starts with an 'E' and has an address ID lower than 500?

   Select first_name, last_name from customer
   Where first_name Like 'E%' and address_id < 500
   Order By customer_id DESC
   Limit 1;

JOINS

NOTA: Los alias del as no se pueden usar en el where y tampoco en el having debido a que se coloca hasta el final del query

1. **Inner Join (JOIN):** Combina los elementos que tienen en común la columna indicada las dos tablas y deja todas las columnas

```
Select payment_id, payment.customer_id, first_name
From
payment Inner Join customer
on payment.customer_id = customer.customer_id;


Select film.title
From
film join
(film_actor join actor
On film_actor.actor_id = actor.actor_id)
On film.film_id = film_actor.film_id
Where actor.first_name = 'Nick' and actor.last_name = 'Wahlberg';
```

2. **Full outer join:** Combina todos los elementos de ambas tablas en una única tabla, donde si no hay valores coincidentes (en la columna indicada) entre las tablas coloca null en las columnas que no tendría información. Para obtener los que no coinciden se hace un where con las columnas coincidentes de ambas tablas con estas en Is null.

```
Select *
From
payment Full Outer Join customer
on payment.customer_id = customer.customer_id
Limit 5;
```

3. **Left outer join:** Crea una tabla con las filas de la tabla a la izquierda con las filas coincidentes con la tabla de la derecha. Deja en null las columnas de la derecha de las filas que no coinciden. Para obtener solo los elementos que no coinciden se puede hacer un where con la columna coincidente de la derecha Is null.

```
Select film.film_id, film.title, inventory.inventory_id
From
film Left Join inventory
```

on inventory.film_id = film.film_id

Limit 5;

4. **Right outer join:** Crea una tabla con las filas de la tabla a la derecha con las filas coincidentes con la tabla de la izquierda. Deja en null las columnas de la izquierda de las filas que no coinciden. Para obtener solo los elementos que no coinciden se puede hacer un where con la columna coincidente de la izquierda Is null.

5. **Union:** Se usa para combinar el resultado de dos o más **selects**, deben tener las mismas columnas

Comandos avanzados SQL

Nota: Show sirve para mostrar variables de ejecución

- **Timestamps and EXTRACT:** https://www.postgresql.org/docs/10/functions-formatting.html
  - ✓ **Tipos de dato:**
    - **Time:** Solo tiempo en horas
    - **Date:** Solo fechas en día, mes y año
    - **Timestamp:** Fecha y tiempo
    - **Timestamptz:** Fecha, tiempo y zona horaria
  - ✓ **Funciones:**
    - **Timezone:** Indica la zona horaria de la DB -> Show
    - **Now():** Indica un timestamp con zona horaria de la fecha y tiempo actual -> Select
    - **TimeOfDay():** Similar a Now pero con un string del día y mes -> Select
    - **Current_time:** Solo retorna el tiempo actual
    - **Current_day:** Solo retorna la fecha actual
  - ✓ **Extracts Functions:**
    - **Extract():** Permite obtener un subcomponente de un tipo de dato Date. Ej: **Extract(Year from date)**
      - **Year**
      - **Month**
      - **Day**
      - **Week**
      - **Quarter**
      - **Dow:** días de la semana. Domingo = 0
    - **Age():** Calcula la distancia entre una fecha y la actual. Ej: **Age(date)**

        - **To_Char():** Convierte datos de fecha en texto. Ej: To_Char(date, 'MM-dd-yyyy')
- **Math Functions:** https://www.postgresql.org/docs/10/functions-math.html
- **String Fuctions:** https://www.postgresql.org/docs/10/functions-string.html
- **Sub-Query:** Permite realizar queries complejas, realizando un query sobre el resultado de otro query. Hay más de un SELECT.
    - ✓ Se encierra entre paréntesis el subquery y dependiendo del retorno se pueden usar comparadores lógicos o matemáticos
    - ✓ Se puede usar **IN** si se tiene un retorno en forma de lista
    - ✓ Se puede usar **EXIST(query)** para comprobar la existencia de una fila
    - ✓ Ejemplos:
        - Select title, rental_rate
          From film
          Where rental_rate >
          (Select AVG(rental_rate) From film);

        - Select film_id, title From film
          Where film_id IN
          (Select inventory.film_id
          From rental
          Inner Join inventory On inventory.inventory_id = rental.inventory_id
          Where return_date BETWEEN '2005-05-29' And '2005-05-30')
          Order By title;

        - Select first_name, last_name
          From customer as c
          Where Exists
          (Select * From payment as p
           Where p.customer_id = c.customer_id
           And amount > 11);

- **Self-Join:** Es un query donde una tabla hace join consigo misma
    - ✓ No hay sintaxis para esto, se usa el Join con la tabla duplicada y nombradas de distinta forma
    - ✓ Ejemplo:

```
Select f1.title, f2.title, f1.length
From film as f1
Inner Join film as f2
On f1.film_id != f2.film_id
And f1.length = f2.length;
```

Assestment Test 2:

1. **How can you retrieve all the information from the cd.facilities table?**

   Select * From cd.facilities;

2. **You want to print out a list of all of the facilities and their cost to members. How would you retrieve a list of only facility names and costs?**

   Select "name", membercost From cd.facilities;

3. **How can you produce a list of facilities that charge a fee to members?**

   Select * From cd.facilities
   Where membercost != 0;

4. **How can you produce a list of facilities that charge a fee to members, and that fee is less than 1/50th of the monthly maintenance cost? Return the facid, facility name, member cost, and monthly maintenance of the facilities in question.**

   Select facid, "name", membercost, monthlymaintenance From cd.facilities
   Where membercost != 0 AND membercost < monthlymaintenance/50;

5. **How can you produce a list of all facilities with the word 'Tennis' in their name?**

   Select * From cd.facilities
   Where "name" like '%Tennis%';

6. **How can you retrieve the details of facilities with ID 1 and 5? Try to do it without using the OR operator.**

   Select * From cd.facilities
   Where facid in (1,5);

7. **How can you produce a list of members who joined after the start of September 2012? Return the memid, surname, firstname, and joindate of the members in question.**

   Select memid, surname, firstname, joindate From cd.members
   Where joindate >= '2012-09-01';

8. **How can you produce an ordered list of the first 10 surnames in the members table? The list must not contain duplicates.**

   ```
   Select Distinct(surname) From cd.members
   Order By surname
   Limit 10;
   ```

9. **You'd like to get the signup date of your last member. How can you retrieve this information?**

   ```
   Select joindate From cd.members
   Order By joindate DESC
   Limit 1;
   ```

   OR

   ```
   SELECT MAX(joindate) AS latest_signup FROM cd.members;
   ```

10. **Produce a count of the number of facilities that have a cost to guests of 10 or more.**

    ```
    Select Count(*) From cd.facilities
    where guestcost >= 10;
    ```

11. **Produce a list of the total number of slots booked per facility in the month of September 2012. Produce an output table consisting of facility id and slots, sorted by the number of slots.**

    ```
    Select facid, Sum(slots) as slots From cd.bookings
    Where starttime BETWEEN '2012-09-01' And '2012-10-01'
    Group By facid;
    ```

12. **Produce a list of facilities with more than 1000 slots booked. Produce an output table consisting of facility id and total slots, sorted by facility id**

    ```
    Select facid, Sum(slots) as total_slots From cd.bookings
    Group By facid
    Having Sum(slots) > 1000
    Order By facid;
    ```

13. **How can you produce a list of the start times for bookings for tennis courts, for the date '2012-09-21'? Return a list of start time and facility name pairings, ordered by the time.**

```
Select bookings.starttime, facilities."name"
From cd.bookings as bookings
Join cd.facilities as facilities
On bookings.facid = facilities.facid
Where facilities."name" Like '%Tennis Court%'
And To_Char(bookings.starttime, 'yyyy-MM-dd') = '2012-09-21'
Order By bookings.starttime;
```

14. **How can you produce a list of the start times for bookings by members named 'David Farrell'?**

```
Select bookings.starttime
From cd.bookings
Join cd.members ON
members.memid = bookings.memid
Where members.firstname = 'David'
And members.surname = 'Farrell';
```

Data Types

https://www.postgresql.org/docs/9.5/datatype.html

- **Boolean:** true o false
- **Character:** char, varchar y text
- **Numeric:** integer y punto flotante
- **Temporal:** date, time, timestamp, e interval
- **UUID:** Universal Unique Identifier
- **Array:** Almacena varios Strings o números, etc
- **JSON**
- **Hstore key-value pair**
- Tipos especiales como la dirección de red e información geométrica

Llaves Primarias y Foráneas:

- **Primary Key:** Identificador único de una fila en una tabla
- **Foreign Key:** La tabla que contiene una de estas se llama tabla de referencia o tabla hija. La tabla a la que se hace referencia se llama Tabla referenciada o tabla padre. Una tabla puede tener muchas llaves foráneas dependiendo de sus relaciones.

Constraints:

- **Column Constraints:**
  - **Not Null**
  - **Unique**
  - **Primary Key**
  - **Foreign Key**
  - **Check:** Se asegura de que todos los valores de una columna satisfagan ciertas condiciones
  - **Exclusion:** Se asegura de que al comparar dos filas en una columna en específico o expresión usando el operador especificado, no todas las comparaciones van a retornar true
- **Table constraints:**
  - **Check(condition):** Para validar una condición al insertar o actualizar filas
  - **References:** Para restringir el valor almacenado en una columna que debe existir en alguna columna de otra tabla.
  - **Unique(Column_list):** Obliga a las columnas a ser únicas
  - **Primary Key(Column_list):** Permite definir la primary key que consiste de múltiples columnas

Estructura Create Table:

```
Create Table table_name (
        column_name Type column_constraint,
        column_name Type column_constraint,
        table_constraint table_constraint
) Inherits existing_table_name
```

Ejemplo:

```
Create Table players (
        player_id Serial Primary Key,
        age SmallInt Not Null
)
```

- **Type Serial:** Esto crea una secuencia donde asigna un valor a la columna dependiendo del valor por defecto y el ya asignado. No se ajusta si se elimina un elemento ya asignado

Estructura Insert

Insert Into table (column1, column2, …) Values (value1, value2, …), (value1, value2, …);

- Se pueden insertar resultados de un query de la siguiente forma:

  Insert Into table (column1, column2, …) Select column1, column2, … From another_table Where condition;

- Los valores insertados deben coincidir con las columnas
- Las columnas Serial no requieren un valor proporcionado

Ejemplo: Ver archivo "Inserts Example.txt"

Estructura Update

Update table

Set column1 = value1, column2 = value2

Where condition;

Ejemplos:

Update account

Set last_login = Current_Timestamp

Where last_login is Null;

Update Join:

Update TableA

Set original_col = TableB.new_col

From TableB

Where TableA.id = TableB.id

Retornar filas afectadas:

Update account

Set last_login = created_on

Returning account_id, last_login;

Estructura Delete:

Delete From table Where row_id = 1;

Usando info de otras tablas:

Delete From TableA

Using TableB Where TableA.id = TableB.id;

PELIGRO: Borrar todas las filas de una tabla

Delete From table;

Retornar filas eliminadas:

Delete From table

Returning column1, column2, …;

Alter Table

Permite cambiar la estructura de las tablas existentes.

- Añadiendo, renombrando o eliminando columnas

Alter Table table_name

Add Column new_col Type; // Drop Column col_name; // Rename Column old_name To new_name;

- Cambiar el tipo de dato de las columnas

- Colocar valores por defecto a las columnas

  Alter Table table_name
  Alter Column col_name
  Set Default value; // Drop Default; // Set Not Null; // Drop Not Null;

- Agregar Check constraints

  Alter Table table_name
  Alter Column col_name
  Add Constraint constraint_name;

- Renombrar una tabla

  Alter Table table_name
  Rename To new_name;

Drop Table

- Para remover todas las dependencias de esta columna:

  Alter Table table_name
  Drop Column col_name Cascade;

- Para validar la existencia de una columna:

  Alter Table table_name
  Drop Column If Exists col_name;

- Eliminar varias columnas de una tabla:

  Alter Table table_name
  Drop Column col_one
  Drop Column col_two;

- Eliminar una table:

  Drop Table table_name;

- Validar existencia de tabla:

  Drop Table If Exists table_name;

Check Constraint

Sirve para validar una columna antes de insertarla o modificarla

Ejemplo:

Create Table example(

ex_id Serial Primary key,

age SmallInt **Check(age > 21),**

parent_age SmallInt **Check (parent_age > age)**

);

Conditional Expressions and Procedures

- **Case:**
  - **General:** Se suele usar como una columna en el Select

    Case
    When condition1 Then resul1,
    When condition2 Then result2,
    Else some_other_result
    End

    Ejemplo:
    Select a,
    Case When a = 1 Then 'One'
    When a = 2 Then 'Two'
    Else 'other' As label
    End
    From test;

  - **Expresión:**

    Case expression
    When value1 Then resul1,
    When value2 Then result2,
    Else some_other_result
    End

Ejemplo:

```
Select a,
        Case a When 1 Then 'One'
                When 2 Then 'Two'
        Else 'other' As label
        End
From test;
```

- **Coalesce:**
  Recibe ilimitados argumentos y retorna el primer elemento distinto de null en caso de que exista, sino retorna null. Se puede usar cuando hay un retorno que puede ser null y se desea reemplazar ese valor

- **Nullif:**
  Recibe dos valores, en caso de que sean iguales retorna null y en caso contrario, retorna el primer elemento. Útil cuando se requieren divisiones y se pueden dar divisiones por 0

- **Cast:**
  Transforma un tipo de dato en otro teniendo lógica en la conversión
  Ejemplo:

  ```
  Select Cast('5' As Integer)
  Select '5'::Integer
  Select Cast(date As Timestamp)
  ```

- **Views:**
  Son resultados de Queries en memoria. Como una tabla virtual.

  Create or Replace View view_name As query;

POO

Pilares:

1. **Encapsulación:** Que todos los datos referentes a un objeto queden "encerrados" dentro de éste y sólo se puede acceder a ellos a través de los miembros que la clase proporcione
2. **Abstracción:** La clase debe representar las características de la entidad hacia el mundo exterior, pero ocultando la complejidad que llevan aparejada mediante el uso de atributos y comportamientos
3. **Herencia:** Una clase que hereda de otra obtiene todos los rasgos de la primera y añade otros nuevos y además también puede modificar algunos de los que ha heredado

4. **Polimorfismo:** Varios objetos de diferentes clases, pero con una base común, se pueden usar de manera indistinta.

Polimorfismo:

- **Estática:** Se basa en la sobrecarga de métodos, donde varios métodos pueden tener el mismo nombre, pero cambian sus argumentos (ya sea en cantidad o sus tipos) y el tipo de retorno.
- **Dinámica:** Se basa en la herencia entre clases y la sobreescritura de los métodos heredados para implementarlos a su manera. La idea es que sin importar que hijo sea, pero teniendo implementado ese método, se pueda invocar dicho método tratando de igual forma a los hijos como se trataría al padre.

Interfaces: Son usadas para indicar qué métodos debe obligatoriamente implementar (contener) una Clase (aunque no tienen por qué comportarse del mismo modo).

Clases Abstractas: Son algo abstracto, no representan algo específico y las podemos usar para crear otras clases. No pueden ser instanciadas, por lo que no podemos crear nuevos objetos con ellas.

Diferencias entre Interfaz y clase abstracta:

- Una clase abstracta puede heredar o extender cualquier clase (independientemente de que esta sea abstracta o no), mientras que una interfaz solamente puede extender o implementar otras interfaces.
- Una clase abstracta puede heredar de una sola clase (abstracta o no) mientras que una interfaz puede extender varias interfaces de una misma vez.
- Una clase abstracta puede tener métodos que sean abstractos o que no lo sean, mientras que las interfaces sólo y exclusivamente pueden definir métodos abstractos.
- En una clase abstracta, los métodos abstractos pueden ser public o protected. En una interfaz solamente puede haber métodos públicos.
- En una clase abstracta pueden existir variables static, final o static final con cualquier modificador de acceso (public, private, protected o default). En una interfaz sólo puedes tener constantes (public static final).

SOLID

- **S**ingle responsibility:
  - Cada clase debería tener una única responsabilidad
  - No debe haber más de una razón para cambiar la clase
  - Las clases deben ser pequeñas
  - Separar clases grandes en clases más pequeñas
- **O**pen Closed:
  - Clases abiertas a extensiones y cerradas a modificaciones

- Se debe poder extender una clase sin tener que modificarla
- Usar variables privadas con getters y setters -> Solo cuando sean necesarias
- Usar Clases base Abstractas (Java interfaces and abstract classes)
- **Open:** Hacer uso del polimorfismo para extender clases
- **Close:** No cambiar el código fuente de lo que no debe ser modificado
- Usar clases abstractas o interfaces para que sin importar la implementación que se esté recibiendo se haga el mismo procedimiento confiando en que la implementación retorne lo deseado.

- **L**iskov Substitution:
  - Los Objetos pueden ser reemplazados con instancias hijas sin alterar el correcto funcionamiento del programa
  - Se puede usar la prueba de "es un", por ejemplo: Un cuadrado "es un" rectángulo, pero un rectángulo "NO es un" cuadrado.
- **I**nterface segregation:
  - Hacer interfaces especializadas para un cliente en específico
  - A veces es mucho mejor tener una interfaz especializada que una con un propósito general
  - Mantener los componentes enfocados y con baja dependencia entre ellos
  - Trata de segregar una "Fat Interface" en interfaces más pequeñas y con alta cohesión, conocidas como "Role Interfaces". Cada interfaz de rol declara uno o más métodos para cada comportamiento en específico.
  - Ejemplo:
    - Cuando se tiene una interfaz con firmas de ciertos métodos, véase animal, y como tal varios pueden implementar dichas firmas en caso de que se pueda. Dado el caso del método de caminar, un perro lo puede hacer, pero un pez no lo puede hacer, por lo que no tendría sentido que la interfaz manejara ese método obligando a lanzar una excepción o un error cuando al animal pez de le pida caminar. https://alexandrefreire.com/principios-solid/segregacion-interfaces/

- **D**ependency Inversion:
  - Las abstracciones no deben depender de los detalles, sino los detalles depender de la abstracción
  - Hacer referencia a interfaces, donde no importa cómo se implementen los métodos el programa seguirá funcionando
  - No es lo mismo que Dependency Injection

Clean Code

¿Qué es?:

- Escribir código entendible y de fácil comprensión
- Que el código tenga significado

- Que se pueda entender de forma rápida
- Debe ser conciso
- Debe evitar el uso de nombre no intuitivos, anidación compleja y grandes bloques de código
- Debe seguir las mejores prácticas y patrones
- Debe ser divertido de escribir y de mantener

APIGEE

En los últimos años se ha visto un crecimiento drástico en el uso y acceso a plataformas web por lo que se ha visto necesario aumentar el ritmo de cambio y adaptación a las necesidades de cada plataforma de manera óptima y cumpliendo con estándares de calidad. Es aquí donde entra Apigee API Manegment Plataform.

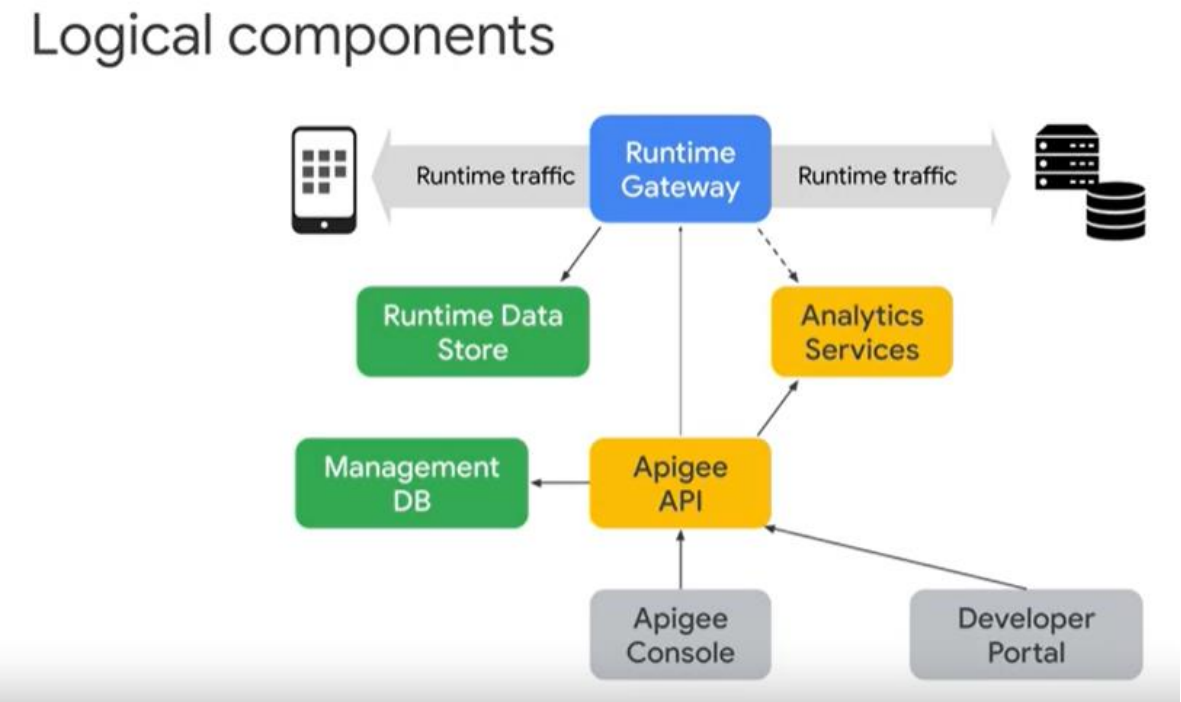
El API funciona como un puente entre la conexión de una aplicación (y sus desarrolladores) con el backend mismo de la aplicación. Existiendo un API Team que desarrolla el API. Como tal la aplicación no se conecta directamente al backend, sino que se conecta al API para realizar las operaciones que requiera.

Apigee API Manegment Plataform:

- **API Runtime:** Se encarga de vigilar el trafico en tiempo de ejecución. Las plataformas a nivel empresarial pueden usarse como SaaS que se corren en la nube de Google y es administrada por ellos
    - Google-Managed Runtime
    - Multi-cloud Runtime
    - On-Premises Runtime
    - Apigee Adapter for Envoy
- **Mediation:** Permite identificar y manipular las peticiones y respuestas de las llamadas al API mediante Apigee
    - Seguridad
    - Transformación
    - Orchestration
    - API Abuse Prevention
- **API Analytics:** Apigee genera información que puede ser analizada por equipos de operación para la toma de decisiones
    - Business Metrics
    - Operational Metrics
    - API program metrics

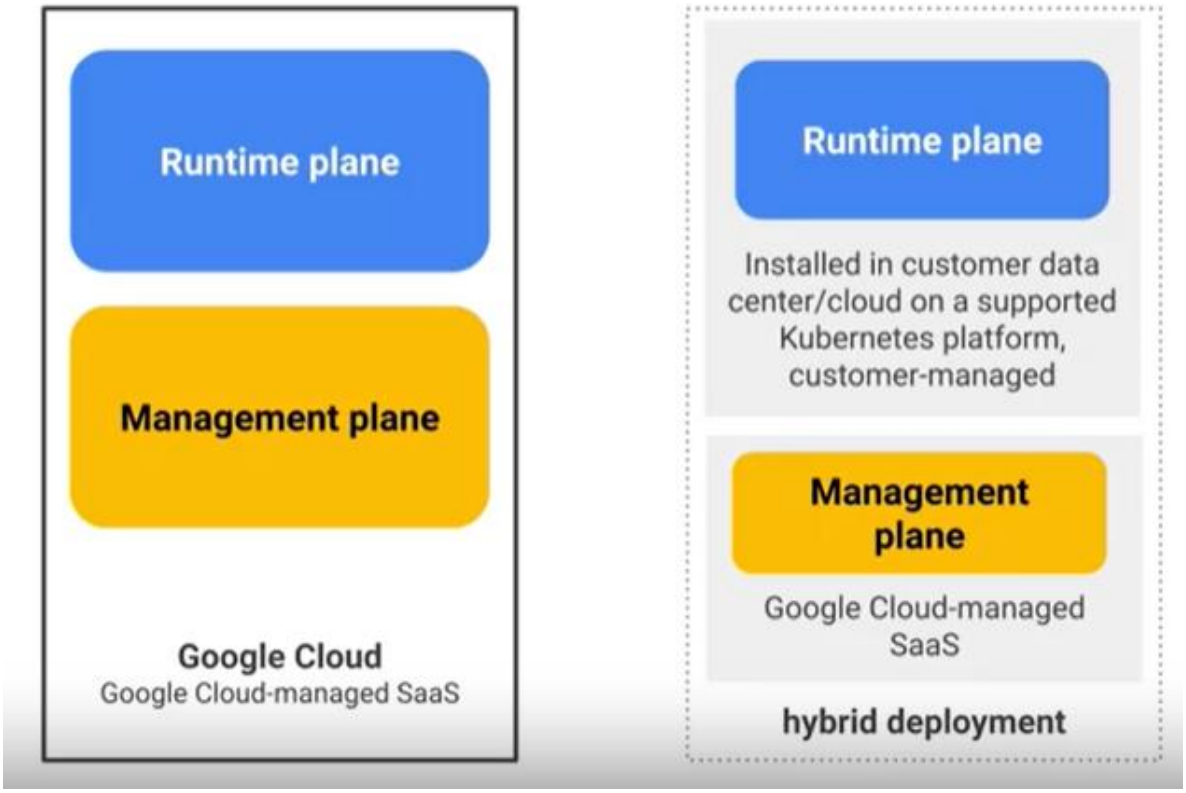
    - API Monitoring and Alerting
- **Developer Ecosystem:** Es un ambiente en el que el API se monta en un catalogo junto con los productos API y en una plataforma de desarrollo donde es posible ver la documentación del API y otras métricas
    - API catalog
    - API products
    - API Monetization
    - API Marketplace

Componentes Lógicos



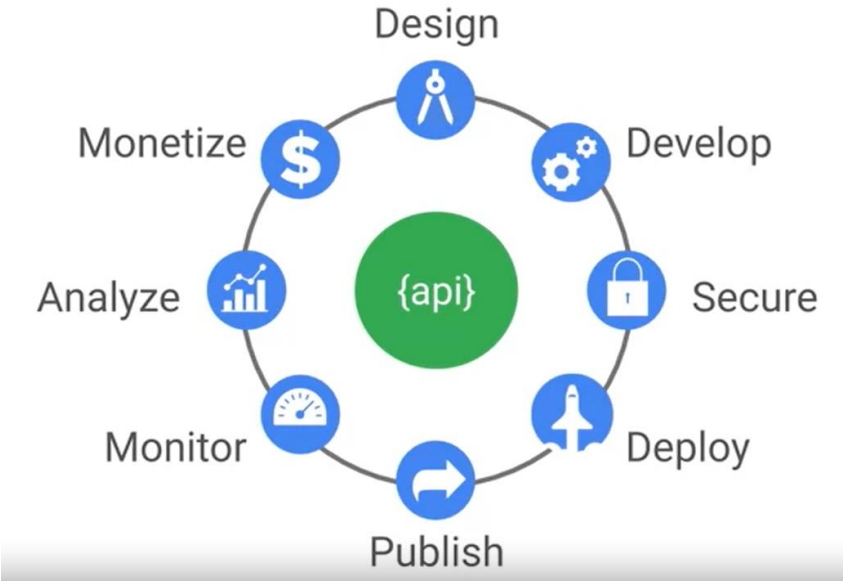
- **Runtime Gateway:** Se ubica en la zona de trafico en tiempo de ejecución. Su componente mas importante es el "procesador de mensajes" el cual se encarga de ejecutar el API en respuesta a las peticiones al API
- **Runtime Data Store:** Almacena los datos usados por el API en tiempo de ejecución
- **Analytic Services:** Los eventos generados para analítica con procesados de manera asíncrona para dar reportes analíticos y poder tomar decisiones
- **Apigee API:** Administra la plataforma Apigee, permite montar o desmontar un API, hacer revisiones de proxy, monitorear, configurar ambientes. Ayuda a la automatización de procesos como CICD
- **Apigee Console y Developer Portal:** Usan el Apigee API, donde permiten controlar todos los aspectos del API (como el ciclo de vida del API y ver los reportes analíticos), así como registrar alguna aplicación
- **Management DB:** Almacena cambios de configuración.

Apigee ofrece una plataforma flexible con opciones de “deployment” (desplegar)



La opción más viable es la primera, permite a los desarrolladores dejar el trabajo de mantenimiento a Google Cloud y concentrarse en los requisitos de sus clientes. La otra opción deja que el desarrollador se encargue de manejar y mantener los contenedores en una especie de kubernates, donde usen servicios de Google para administrar su funcionamiento.

API LifeCycle:



- **Design:** Diseño del API y aprobación por los stakeholders.
  - Para diseñar un API se puede usar la herramienta OpenAPI, con el fin de definir bien la interfaz y sus especificaciones
  - Es una especificación sin entrar en detalle de la implementación
  - Con el fin de que los desarrolladores puedan explorar, entender y probar el API
  - También se puede usar la herramienta para generar un proxy STUB del API, con una plantilla para construir un API ligado a la especificación definida
- **Develop:** Desarrollo del API
  - Apigee permite realizar proxies del API usando políticas, que son funciones predefinidas que pueden ser configuradas sin código
  - De ser necesario uno puede escribir código en Java para los casos de uso más complejos
  - El proxy puede ser debuggeado usando la herramienta Apigee's Trace Tool
  - Así se pueden identificar errores en producción
- **Secure:** Realizar capa de seguridad al API
  - Se puede **agregar** seguridad al api, incluso si este no tiene manejo de ello
  - Se puede agregar OAuth, SAML, Json Web Tocken and HMAC
  - Tambien se puede usar protección de hilos contra ataques basados en contenido
  - Sirve como una detección temprana de posibles amenazas antes de que lleguen las peticiones al backend
  - Al estar montado en la nube de Google se usan varios servicios de seguridad como restricciones de IP
- **Deploy:** Desplegar el API en producción

    - El proceso incluye repetidas pruebas para verificar el correcto funcionamiento del API
    - Se provee un Mangement API para crear, modificar o desplegar proxies y configuraciones de artefactos
    - El proceso es algo como:
        - Check-in
        - Build
        - Test
        - Deploy
- **Publish:** Publicar el API en “app developers”
    - Se publica la especificación junto con su documentación para que otros desarrolladores puedan usar tu API
    - Se proveen dos tipos de portales para desarrolladores
        - Drupal-based Portal: ofrece un sistema de administración de contenido personalizable
        - Un portal integrado hosted el cual requiere menos esfuerzo, pero carece de algunas mejoras.
- **Monitor:** Monitorear la salud y uso del API
    - Se provee un tablero con datos lo mas cercanos al tiempo real
    - Ayuda a diagnosticar problemas
    - Captura métricas de rendimiento
- **Analyze:** Analizar los puntos débiles y de mejora del API
    - Captura métricas de negocio, mediante la captura de datos del uso del API
    - Se puede integrar esta información con otras herramientas
- **Monetize:** Dependiendo del modelo de negocio se puede o no monetizar el acceso al API
    - Empresas con APIs publicas o los que ofrezcan productos digitales a aliados pueden usar esta herramienta para obtener algún beneficio según los desarrolladores usen el API o compartir la ganancia con los desarrolladores que mantienen tu negocio

Apigee Organizations

Una organización está asociada a un único proyecto de Google Cloud, y es la entidad de más alto nivel en Apigee

- **Access:** Los usuarios tienen acceso a la organización asociándolos con uno o más roles. Un Administrador de la organización Apigee tiene acceso de “superuser” dentro de la organización. Otros roles son diseñados para operaciones, negocios o miembros del equipo de desarrollo del API
- **APIs:** Son implementadas usando API proxies, los cuales son construidos usando políticas que proveen una funcionalidad especifica como parte de las peticiones del proxy y el flujo de respuesta. Las políticas pueden ser combinadas en flujos compartidos (deben ser usados en múltiples proxies) para patrones en común.
- **Environments:** API proxies y flujos compartidos se despliegan en ambientes, los cuales son usados frecuentemente para modelar y aplicar un ciclo de vida de desarrollo a un API. Los usuarios pueden tener diferentes permisos para cada ambiente. Es un contenedor en tiempo de ejecución para API proxies.
    - **Deployments:** Revisiones de proxy y flujo compartido pueden ser desplegadas a un ambiente. Estas son inmutables.
    - **Connectivity:** Los “Target servers” desacoplan los endpoints concretos (de las Urls) del código del proxy. Las “Keystores y Truststores” almacenan certificados TLS y llaves privadas para permitir conexiones seguras de entrada y salida.

    - **Runtime Data:** Usa cache para eliminar trafico innecesario al backend o servicios terceros. Los “Key Values Maps” pueden ser usados para configuraciones de ambiente especificas.
    - **Config:** Los “Debug Sessions” capturan el tráfico del API recibido cuando se rastrea un API proxy. Los “Flow Hooks” permiten compartir automáticamente flujos a ser unidos con cada proxy en el ambiente. Los “Resouce Files” permiten compartir código entre proxies en un ambiente.
- **Publishing:** Las APIs son productivas al exponerlas en un su portal de desarrollo como Productos API. Los desarrolladores de apps registran sus apps para que puedan usar uno o más productos API.
- **Runtime Data:** Las apps tienen “Llaves API” y “OAuth tokens” para acceder a las APIs deseadas. Los productos API especifican el ambiente permitido para las apps.
- **Config:** Los “Key Value Maps” en el ámbito de la organización pueden ser usados para configurar toda la organización. Los “Environment Groups” mapea los hostnames al ambiente Apigee. Los “Data Collectors” son usados para recolectar información personalizada y realizar reportes.
- **Analytics:** La “Analytics Data” es capturada para proveer visibilidad para todo el tráfico del API. Realiza reportes personalizados para la organización

REST

Antes se usaba por preferencia el protocolo SOAP, el cual era un tanto complejo por lo que era necesario el uso de un documento llamado WSDL donde se especificaba el uso del servicio web y se comunicaban mediante archivos XML. En cambio, REST es un estilo arquitectural más sencillo (más no es un estándar) que surgió gracias a los conceptos que trajo el protocolo HTTP, y que usualmente usa JSON para enviar mensajes. Gracias a esto REST es el estilo más usado en las API web (RESTful).

Verbos:

- **Get:** Consultar uno o varios recursos
- **Post:** Crear un recurso
- **Put/Patch:** Modificar un recurso. Put modifica por completo el recurso, patch modifica únicamente los campos especificados
- **Delete:** Eliminar un recurso

Para el manejo de recursos se usa en la URL el nombre del recurso en plural (/orders) y cuando se quiere uno en específico se indica el id del recurso después de un “/” (/orders/{id}). No se deben usar nombres internos de la compañía, mejor usar nombres comprensibles.

Resource	POST create	GET read	PUT/PATCH update	DELETE delete
/dogs	Create a new dog	List all or matched dogs	Bulk update all or matched dogs	Delete all or matched dogs
/dogs/1234	N/A	Retrieve "Toto"	If exists, update "Toto" If not, error	Delete "Toto"

Hay que evitar el uso de verbos en las rutas de la URL, estos son difíciles de recordad y para operaciones CRUD es mejor solo manejar las operaciones con los verbos HTTP. Para las operaciones NO CRUD, se debería considerar el uso de solo POST y el uso de un parámetro query para realizar una acción en un recurso (POST /dogs/1234?action=walk).

Resource	POST create	GET read	PUT/PATCH update	DELETE delete
/dogs	Create a new dog	List all or matched dogs	Bulk update all or matched dogs	Delete all or matched dogs
/dogs/1234	**Non-CRUD operations with query param**	Retrieve "Toto"	If exists, update "Toto" If not, error	Delete "Toto"

Se debe considerar la opción de usar los parámetros query en vez de realizar llamadas repetitivas. Se pueden usar los parámetros para filtrar los resultados de una consulta. Esto es más trabajo para el programador. También se pueden usar los parámetros para indicar al API que atributos debe retornar.

```
GET /employees/5678?fields=id,fullName,geo.latitude,geo.longitude
{
  "id": 5678,
  "fullName": "Joe Apigeek",
  "geo": {
    "latitude": 37.4049103,
    "longitude": -122.0216585
  }
}
```

Consejos al desarrollar un API:

Diseñar desde la perspectiva de los usuarios que van a utilizar el API y realizar la interfaz del API antes de su implementación.

- Outside-in thinking: Diseñar valor para el consumidor como prioridad
- Construir algo para entregar ese valor al consumidor
- Centrarse en el consumo de la app
- Que sea divertida de usar

Esto ayudará a darse cuenta de errores en etapas tempranas del desarrollo. Promueve consistencia, usabilidad y buenas prácticas de diseño. Además, incrementa la habilidad para realizar desarrollo en paralelo.

OpenAPI Specs

En un inicio las API con SOAP se realizaban por medio de WADL, pero esté era complejo y difícil de leer. Luego llegó Swagger que especificaba la interfaz Restful en un archivo tipo Json o Yml, mucho más sencillo de usar y de construir. Este formato ahora es usado por OpenAPI siendo un formato open source.

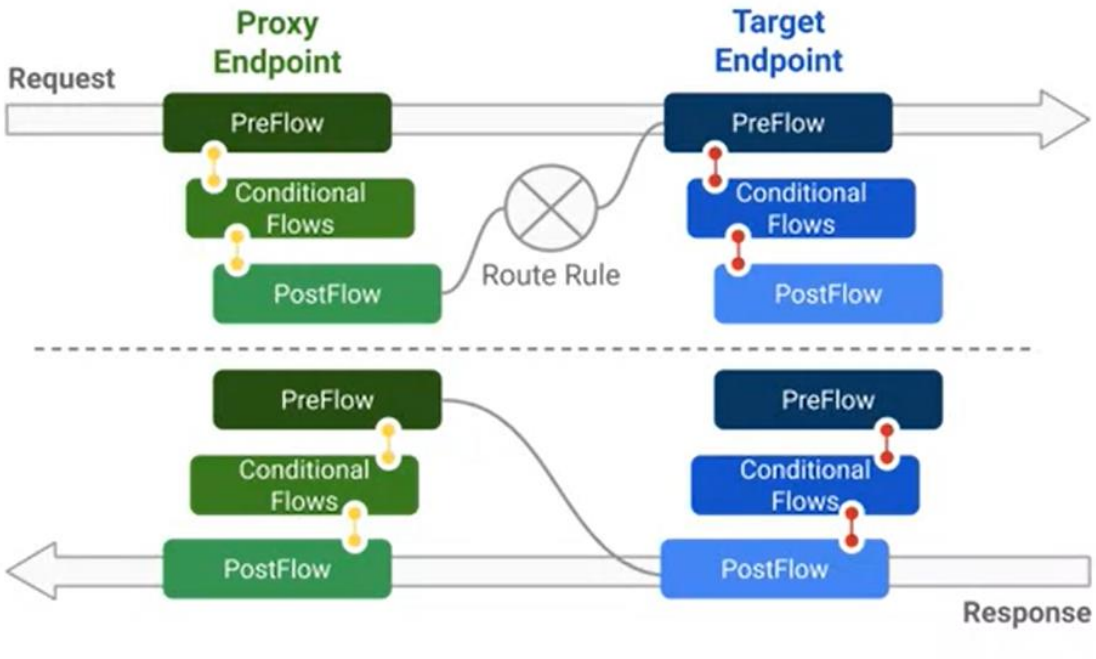
- Describe el contrato de uso del API
- Mucho más comprensible para humanos y maquinas
- Define los paths disponibles y las operaciones permitidas para cada path
- Especifica el formato de petición y respuesta, incluyendo los códigos de error y éxito, como detalles de autenticación
- Sirve para generar API proxy Stubs
- Sirve para validar las peticiones entrantes

API Proxies

Es un intermediario entre un cliente y un proveedor, donde el cliente no requiere conocer acerca del proveedor solo que el proxy le retorna esa información sin importar lo que haya detrás. Así mismo, el proveedor no necesita saber quién es el cliente, solo sabe que el proxy lo contacta.

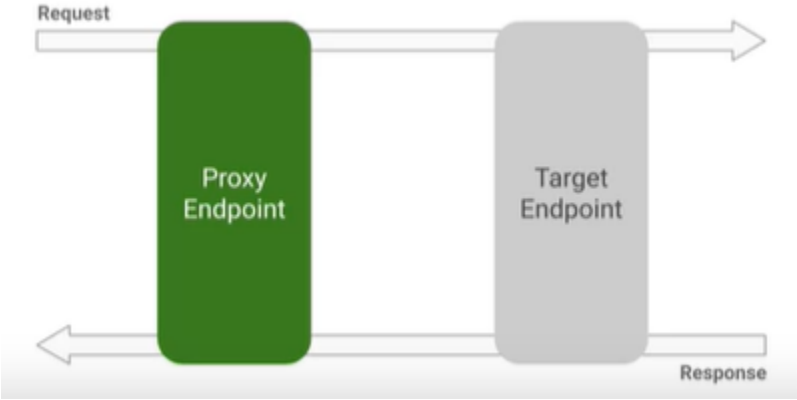
Cumple con el objetivo de desacoplar al Consumidor del API de los servicios de backend e implementa una API orientada al consumidor diseñada específicamente para los

desarrolladores de apps. Protege a los desarrolladores de apps de la complejidad de los sistemas backend y permitiendo mayor modificabilidad sin afectar las apps que lo usen.



- El proxy recibe los request del API basado en combinaciones del hostname específico, el esquema y la ruta URL
- Este agrega seguridad, escalabilidad y “rate limiting” al API al adjuntar políticas al flujo
- Puede llamar múltiples servicios y combinaciones, además de transformar las respuestas durante el llamado al API
- Captura información de las llamadas para realizar análisis (Analytics)
- Puede que incremente la latencia de las llamadas a cambio de los beneficios como:
    - Se gana control y conocimiento
        - Verificar la seguridad del API mediante tokens y otras políticas de seguridad rígidas
        - Realizar administración del tráfico y atender requests desde cache para llamadas que no requieran llegar al backend y la respuesta esté almacenada
        - Recolectar información para analytics y obtener conocimiento sobre el uso del API y el rendimiento
    - Desacopla al consumido del API de los servicios del backend
        - Crear APIs que combinen backend con servicios de terceros
        - Innovar en APIs orientadas al consumidor sin afectar los servicios del backend
        - Mover y actualizar el backend sin dañar las apps consumidoras

Proxy y Target Endpoint:



El tráfico de las peticiones (con su protocolo, host alias, y URL base) es redirigido al Proxy endpoint, donde este analiza la petición y verifica la petición entrante. Maneja políticas de tráfico “rate-limit” y de protección contra peticiones invalidas o maliciosas. Típicamente, este también construye la respuesta del API.

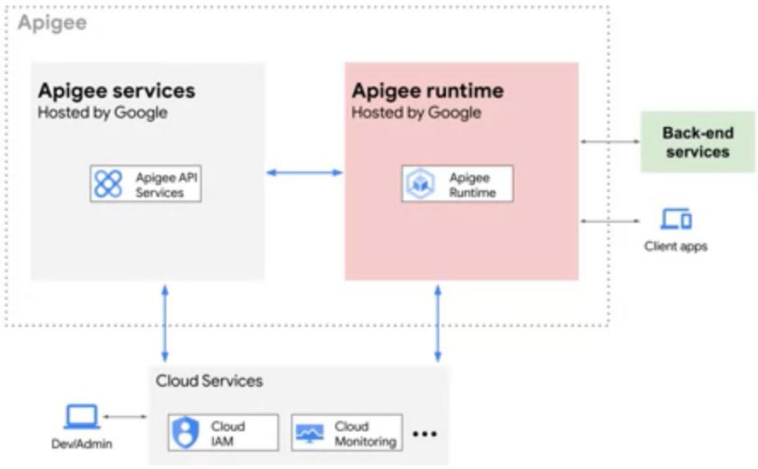
El Target endpoint especifica el servicio backend a llamar, construye las peticiones API para el servicio backend. Especifica la conexión y timeouts (I/O) de entrada y salida de las llamadas al servicio backend. También valida los códigos de estado y payloads de la respuesta del backend.

Conectándose a un API proxy:

Cada organización puede tener múltiples ambientes. Los cuales sirven para controlar el ciclo de vida de un API, por ejemplo: se inicia en el ambiente de desarrollo que luego es ascendido al ambiente de pruebas y eventualmente ascendido al ambiente de producción.

El Apigee en tiempo de ejecución puede recibir tráfico usando diferentes hostnames (nombres de dominio).

Arquitectura de Apigee:



Consiste en dos componentes primarios: Apigee Services y Apigee runtime

- **Apigee Services:** Provee las APIs que se usan para crear, administrar y desplegar los proxies API que se desarrollen.
- **Apigee runtime:** Es un conjunto de servicios de tiempo real contenidos y ejecutados en un cluster de Kubernates administrado por Google. Permite a los API proxies llamar servicios backend

Todo el tráfico del API pasa a través y es procesado por es los servicios runtime o servicios de Google Cloud.

Para crear una organización: se debe crear un proyecto de Google cloud (uno por cada organización Apigee), luego el cliente debe crear una nube virtual o privada con una red VPC y una conexión para ser usada por el Google managed Apigee runtime. Ahora si se puede crear la organización.

El “Apigee Runtime”, que contiene los servicios runtime, será creado en un proyecto de inquilino único para la organización. A esta se accede mediante una dirección IP privada. Toda llamada entrante debe pasar a través del proyecto del cliente mediante la dirección IP.

El Apigee Runtime puede llamar directamente los servicios backend. El cliente tiene control completo sobre el enrutamiento del proyecto. Los recursos dentro del proyecto, como máquinas virtuales, pueden realizar llamados a APIs usando la red interna. Apigee también puede llamar recursos en la conexión peered VPC.

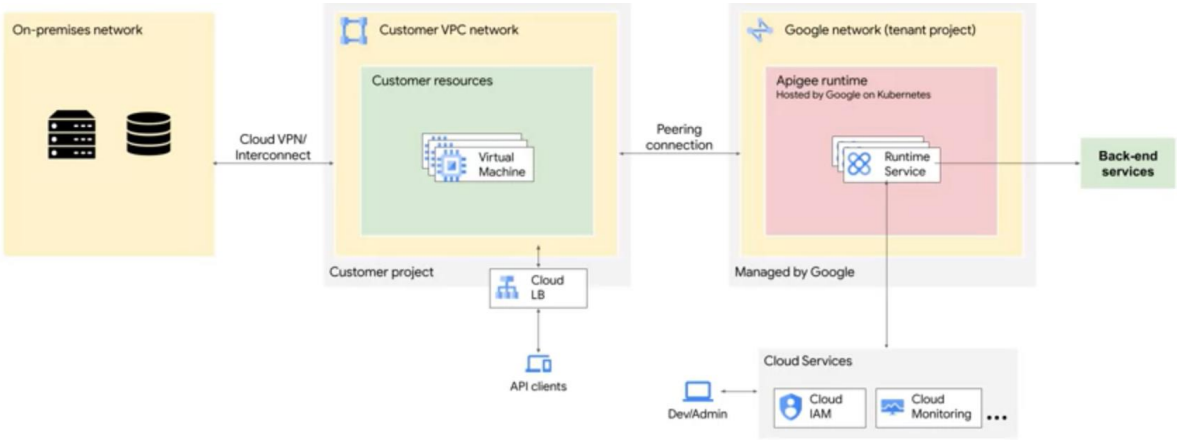
También se pueden usar Cloud VPN o Cloud Interconnect para conectar tu red On-premise con tu red privada Google Cloud Virtual. De tal forma Apigee puede llamar servicios en tu red On-premise sin salir de la red privada.

Si el cliente desea exponer APIs fuera de la red o de Google Cloud, un balanceador de carga (LB) puede ser creado para proveer acceso seguro a Apigee mediante la red peer. Ya sea que el request venga de la red interna o desde afuera de Google Cloud, se sabe que todos los requests entran al servicio Runtime mediante una única IP.

Dependiendo del Environment Group se sabe a qué API proxy se debe enrutar el request, pues una organización API puede contener uno o más Environment Groups, los cuales son usados para mapear hostnames a los ambientes (environments).

El Apigee runtime puede recibir llamadas API que usen distintos dominios, donde un dominio solo puede ser asignado a un único Environment Group. Por lo que cada que llegue un request con un dominio, solo un ambiente en ese Environment Group puede manejarlo. Un Environment Group puede tener asignados más de un dominio.

Un Environment Group puede tener más de un ambiente, y de igual forma un ambiente puede estar en más de un Environment Group.



Base Path: Es la primera parte de la URL que sigue al dominio, se configura en un Proxy Endpoint (HTTPProxyConnection). Para desplegar una revisión Proxy a un ambiente: todos los base path endpoints deben ser únicos en ese ambiente.

En un Environment Group, más de un proxy puede ser desplegado en diferentes ambientes con el mismo base path. En este caso se daría un conflicto de enrutamiento al momento del despliegue con opción a abortar el despliegue. Si se decide no abortar, el primer proxy recibirá todos los request y no será hasta que este sea “undeployed” que el segundo proxy empezará a recibirlos.

HTTP y HTTPS:

- Las APIs deben ser diseñadas para usar HTTP, HTTPS (Encriptado mediante un certificado TLS – Transport Layer Security) o ambas
- Una instancia runtime solo acepta HTTPS y usa un certificado auto firmado

- Para llamar APIs vía HTTPS, el certificado TLS debe estar ubicado en un balanceador de carga en el proyecto del cliente. La conexión TLS con el el balanceador de carga se termina antes de que el request alcanza el Apigee Runtime
- Si se requiere permitir HTTP request, la conexión debe también terminar en un proyecto balanceador de carga del cliente
- Se recomienda siempre usar HTTPS

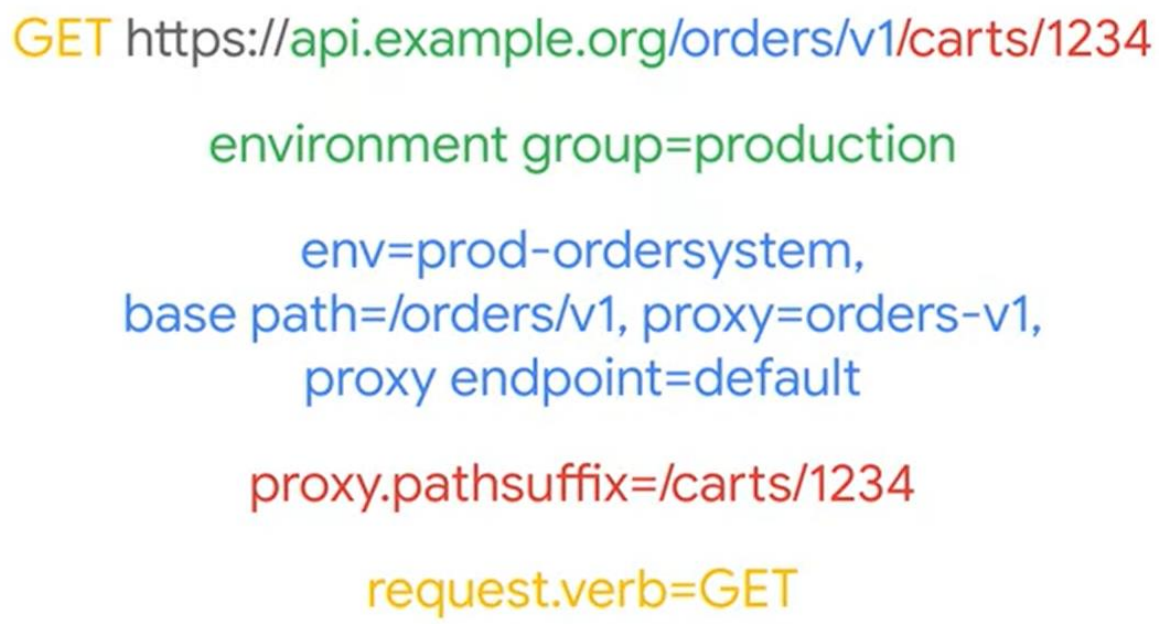
Proxy Endpoints:

Las organizaciones típicamente tienen muchos proxies desplegados para un ambiente dado.
Los Proxies típicamente tiene un único proxy endpoint, pero puede tener más de uno.
Un Proxy Endpoint es el punto de entrada para las llamadas al API

Condiciones, Flujos y Políticas:

Variables: Se suelen predefinir variables que son usadas automáticamente durante una llamada al API. A veces brindan información del request entrante y otras pueden ser del response también. Si se desea, se pueden crear variables personalizadas para construir responses o usar en analytics.

Proxy path suffix: El nombre de la variable es “proxy.pathsuffix”, el cual es colocado a la parte de la URL entrante seguido del base path sin incluir los query parameters. El verbo y el proxy path sufix indican operaciones a ser realizadas.



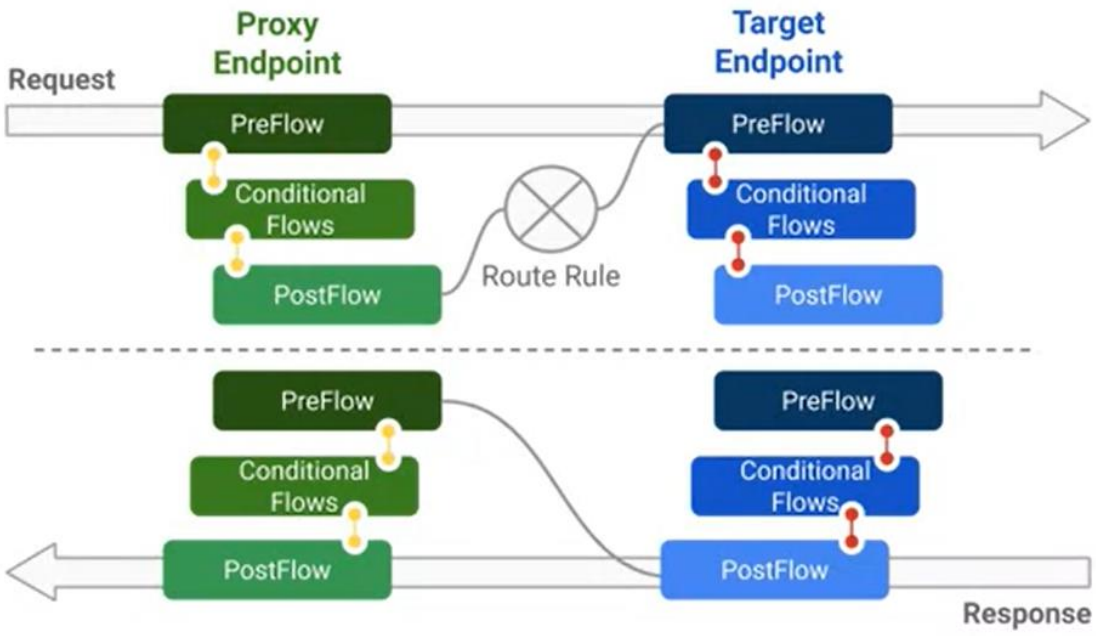
Condiciones:

Evalúan cuando algo es verdadero o falso, se pueden usar las variables predefinidas o personalizadas, así como datos primitivos y condicionales AND y OR.

```
<Condition>request.queryparam.action != null</Condition>

<Condition>response.status.code >= 500</Condition>

<Condition>request.verb == "GET" AND proxy.pathsuffix MatchesPath "/orders/*"</Condition>
```

Se pueden usar operaciones de coincidencia como coincidencia completa (==), coincidencia parcial e inicial (Matches), coincidencia mediante un Regex o expresiones regulares (JavaRegex) y coincidencia de rutas (MatchesPath) donde se le indica cuantos / va a tomar en cuenta. Todos los operadores son sensibles a las mayúsculas y las minúsculas.

Flujo:



- **Orden de evaluación:**
  - Proxy endpoint request: El request se valida en esta sección
  - Route Rules: Determina con las reglas que target debe continuar con el procesamiento
  - Target endpoint request: Se procesan el request original y se construye el request a ser enviado al backend
  - El Request es enviado al backend y retorna el response
  - Target endpoint response: Se realiza un Parse al response en este paso

    - Proxy endpoint response: El proceso continúa hacia el proxy endpoint response, donde se construye el API response y se retorna al consumidor.

- **Flujos:**
    - Para cada sección, las políticas en PreFlow se evalúan siempre de primeras
    - Los Conditional Flows son evaluados en orden
    - Solo las políticas en el primer flujo condicional con una condición que dé true son ejecutadas
    - Si no hay condiciones en true, ninguna es ejecutada
    - Finalmente, se evalúan las políticas en PostFlow

- **Policies (Políticas):**
    - Son modelos preconstruidos con funciones limitadas y específicas
    - Sus funciones indican rate limit, transformaciones, mediación y seguridad.
    - Se configuran mediante XML
    - Pueden correr código personalizado
    - Pueden ser ejecutadas condicionalmente
    - Tipos:
        - Traffic Management
        - Security
        - Mediation
        - Extension
    - Se pueden adjuntar políticas en uno o más flujos como pasos, los cuales pueden ser asignados a una condición, ejecutándose únicamente si la condición da true.
    - Para el nombre de una política se debe usar un prefijo para especificar el tipo de política, seguido de un nombre que explique lo que hace. La convención es: {policy type abbrevation} – {name}

Target Endpoints, Route Rules y Target Servers:

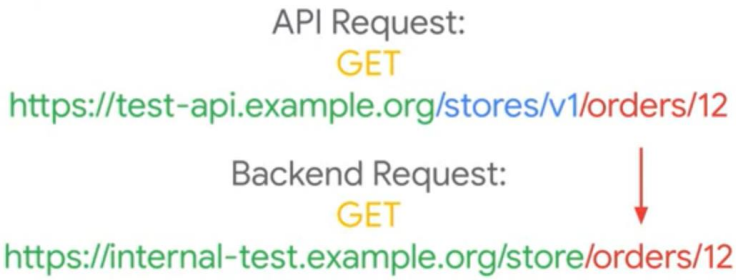
Target Endpoints: se ubican dentro de un proxy, puede haber mas de uno en un proxy o incluso puede que no haya ninguno (cuando no se requiere un llamado al backend).

Route Rules: Después de que termine el flujo del Proxy Endpoint, se evalúan las configuraciones de Route Rules para decidir a que target Endpoint se debe dirigir la petición. La primera regla indica el Target Endpoint. El response es retornado a través de los mismos proxy y target endpoints. Las reglas pueden indicar que no se debe usar un target endpoint por lo que la ejecución continua con el response del proxy endpoint.

```
<ProxyEndpoint name="store">
  ⋮
  <RouteRule name="auth-notarget">
    <Condition>proxy.pathsuffix == "/auth"</Condition>
  </RouteRule>
  <RouteRule name="test-store-server">
    <Condition>request.queryparam.test != null</Condition>
    <TargetEndpoint>test-store-backend</TargetEndpoint>
  </RouteRule>
  <RouteRule name="default">
    <TargetEndpoint>store-backend</TargetEndpoint>
  </RouteRule>
</ProxyEndpoint>
```

Para especificar la ruta backend en el target endpoint:

```
<TargetEndpoint name="store-backend">
  ⋮
  <HTTPTargetConnection>
    <URL>https://internal-test.example.org/store</URL>
  </HTTPTargetConnection>
</TargetEndpoint>
```



Se puede crear un Target Server Configuration con el mismo nombre en cada ambiente, en donde se definen el dominio y el puerto a utilizar, junto con la información del certificado necesario para la validación del cliente o del servidor.

Para referenciar el target server se usa una sección LoadBalancer con una etiqueta server que contenga el nombre del server, además de una etiqueta Path con la ruta que sigue después del dominio.

```
<TargetEndpoint name="store-backend">
  ⋮
  <HTTPTargetConnection>
    <LoadBalancer>
      <Server name="store-backend"/>
    </LoadBalancer>
    <Path>/store</Path>
  </HTTPTargetConnection>
</TargetEndpoint>
```

Al usar LoadBalancer se pueden definir más de un server y un algoritmo para distribuir el trafico a los servicios del backend. Puede identificar automáticamente fallos del backend en fuera de servicio. Además se puede indicar el uso de un monitor de salud del backend mediante TCP y HTTP para detectar cuando un backend está de vuelta en servicio.

```
<TargetEndpoint name="my-backend">
  ⋮
  <HTTPTargetConnection>
    <LoadBalancer>
      <Algorithm>RoundRobin</Algorithm>
      <Server name="backend-1"/>
      <Server name="backend-2"/>
      <MaxFailures>5</MaxFailures>
      <ServerUnhealthyResponse>
        <ResponseCode>503</ResponseCode>
      </ServerUnhealthyResponse>
    </LoadBalancer>
    <HealthMonitor>
      <IsEnabled>true</IsEnabled>
      <IntervalInSec>10</IntervalInSec>
      <TCPMonitor>
        <ConnectTimeoutInSec>5</ConnectTimeoutInSec>
        <Port>80</Port>
      </TCPMonitor>
    </HealthMonitor>
    <Path>/v1</Path>
  </HTTPTargetConnection>
</TargetEndpoint>
```

Se pueden usar propiedades para el target endpoint entre las cuales están:

- connect.timeout.milis -> connection timeout
- io.timeout.milis -> Socket read/write timeout

```
<TargetEndpoint name="my-backend">
  ⋮
  <HTTPTargetConnection>
    <URL>https://internal-test.example.org/store</URL>
    <Properties>
      <Property name="connect.timeout.millis">2000</Property>
      <Property name="io.timeout.millis">15000</Property>
      <Property name="compression.algorithm">gzip</Property>
    </Properties>
  </HTTPTargetConnection>
</TargetEndpoint>
```

API Products, Developers, Apps y API Keys

API Product: Es un paquete de operaciones API (como los casos de uso encapsulados en un producto) que realiza un control de acceso a nuestras API's, manejar distintos niveles de acceso a los servicios (solo lectura, lectura-escritura, interno/publico, libre/estándar/premium)

- **Configuración:**
  - **Custom Attributes:** Son parejas nombre-valor que se convertirá en una variable cuando llegue un request

- **Publicar el API:**
  - Se encapsulan las API y las operaciones API en un API Product
  - Los custom attributes deben ser adjuntados al API producto y a sus operaciones API
  - Los desarrolladores crean apps consumiendo APIs
  - Se pueden usar credenciales (keys de consumo o secrets) para que una app quede asociada con uno o más productos
  - Una Key de consumidor puede ser usada como una API key para acceder a operaciones API protegidas

App Developers: Son configurados con Nombre, Correo y Username. A estos se les pueden asignar Custom Attributes.

Apps: Son asociadas a un desarrollador y también se les puede asignar Custom Attributes. Uno o más conjuntos de credenciales pueden ser creadas para una App, y estas pueden ser asociadas con uno o más API products. Las credenciales pueden ser removidas tanto por el desarrollador como por el API team.

API Key: Una Key de consumo (enviada con un API request) es usada como una API key que identifica al App. Permite el acceso a operaciones API protegidas por políticas de VerifyAPIKey; si el API Key es válido, las variables son creadas para asociar la app, el desarrollador y los atributos del API producto (incluyendo custom attributes).

Para especificar dicha política se debe indicar en el PreFlow en el Request del ProxyEndpoint, como un paso a realizar. Adicionalmente, hay que definir el nombre de la política con ciertas configuraciones y el API Key en una etiqueta de VerifyAPIKey. Así mismo se crean a partir de los nombres las variables de la siguiente forma:

ProxyEndpoint

```
<ProxyEndpoint name="orders">
  <PreFlow name="PreFlow">
    <Request>
      <Step>
        <Name>VK-VerifyKey</Name>
      </Step>
    </Request>
  :
</ProxyEndpoint>
```

VerifyAPIKey policy

```
<VerifyAPIKey continueOnError="false"
  enabled="true" name="VK-VerifyKey">
  <APIKey ref="request.header.apikey"/>
</VerifyAPIKey>
```

Example variables
format: verifyapikey.{policy_name}.{var_name}

Developer variables
verifyapikey.VK-VerifyKey.developer.firstName
verifyapikey.VK-VerifyKey.developer.lastName
verifyapikey.VK-VerifyKey.developer.email
verifyapikey.VK-VerifyKey.developer.{custom_attr_name}

App variables
verifyapikey.VK-VerifyKey.app.name
verifyapikey.VK-VerifyKey.{custom_app_attr_name}
verifyapikey.VK-VerifyKey.app.apiproducts

API product variables
verifyapikey.VK-VerifyKey.apiproduct.name
verifyapikey.VK-VerifyKey.apiproduct.{custom_attr_name}
verifyapikey.VK-VerifyKey.apiproduct.developer.quota.*

API Product Strategies:

Un producto tradicional tiene las siguientes características:

- Se deben conocer las necesidades de la audiencia a la que va dirigida la app
- La documentación es muy importante, pues describe las funcionalidades y el cómo usarlas
- Incluye el precio del producto

Un API Product es bastante similar a uno tradicional:

- Se deben conocer las necesidades del desarrollador consumidor
- Documentación que describe las funcionalidades del API product y el cómo usarlas
- Incluye el precio de uso del API producto
    - Se pueden monetizar las APIs para generar ganancias por el uso de estas o retribuir a los desarrolladores que invierten y generan nuevos ingresos comerciales
    - Registrarse, indicar métodos de pago o billetera y acceder a reportes de monetización en el Portal de Desarrolladores

Projects: Componentes de un proyecto

- Entregables del proyecto y fecha de entrega
- Pasos para completar
- Línea de Tiempo para completar
- Siguientes objetivos después de completar el proyecto

Diferencias entre un Product y un Project

- Products son diseñados basados en requerimientos de negocio
- Products se adaptan a las necesidades del mercado
- Products no tienen una fecha para completarse
- Se recomienda adoptar una mentalidad hacia el producto

Diseño de REST API

Status Codes: Señales para identificar fallos o éxito de una operación

https://developer.mozilla.org/es/docs/Web/HTTP/Status

- **Grupos:**
  - 1xx: Informar
  - 2xx: Éxito
  - 3xx: Redirección
  - 4xx: Error en el cliente (del request)
  - 5xx: Error en el servidor (no es del cliente)
- **Códigos comunes:**
  - **200: Ok**
  - **400: Bad Request** - el servidor no pudo interpretar la solicitud dada una sintaxis inválida
  - **401: Unauthorized** - Es necesario autenticar para obtener la respuesta solicitada. Esta es similar a 403, pero en este caso, la autenticación es posible.
  - **403: Forbidden -** El cliente no posee los permisos necesarios para cierto contenido, por lo que el servidor está rechazando otorgar una respuesta apropiada.
  - **404: Not Found -** El servidor no pudo encontrar el contenido solicitado. Este código de respuesta es uno de los más famosos dada su alta ocurrencia en la web.
  - **500: Internal Server Error -** El servidor ha encontrado una situación que no sabe cómo manejarla.
  - **201: Created -** La solicitud ha tenido éxito y se ha creado un nuevo recurso como resultado de ello. Ésta es típicamente la respuesta enviada después de una petición PUT.
  - **204: No Content -** La petición se ha completado con éxito, pero su respuesta no tiene ningún contenido, aunque los encabezados pueden ser útiles. El agente de usuario puede actualizar sus encabezados en caché para este recurso con los nuevos valores.

- **304: Not Modified -** Esta es usada para propósitos de "caché". Le indica al cliente que la respuesta no ha sido modificada. Entonces, el cliente puede continuar usando la misma versión almacenada en su caché.
- **405: Method not Allowed -** El método solicitado es conocido por el servidor, pero ha sido deshabilitado y no puede ser utilizado. Los dos métodos obligatorios, GET y HEAD, nunca deben ser deshabilitados y no deberían retornar este código de error.
- **406: Not Acceptable –** Cuando un request entrante tiene un Accept header que pregunta por un formato de response que no es soportado por el API.
- **409: Conflict -** Esta respuesta puede ser enviada cuando una petición tiene conflicto con el estado actual del servidor.
- **503: Service Unavailable -** El servidor no está listo para manejar la petición

- **Recomendaciones:**
  - Es bueno dar contexto de los errores para identificar la razón de este, esto sin filtrar información interna y delicada del servidor
  - Si se manejan recursos como usuarios, es preferible enviar un 404 a pesar de que el usuario exista para que un hacker no trate de hackear ese usuario al recibir un 403

- **Paginación:**
  - Paginar los resultados para búsquedas y evitar realizar consultas repetidas o pesadas
  - Usarlo para gran cantidad de resultados
  - **Tipos:**
    - **Offset y Limit:**
      - Funciona bien con la mayoría de casos de uso de UI, pero no permite realizar paginación consistente a través de todos los recursos
      - Actualizar recursos puede causar que su localización en la lista cambie, lo que causaría que elementos se retornaran más de una vez u otros no sean retornados

    - **Basado en Valor:**
      - Devuelve un atributo que se puede devolver con la siguiente llamada para recuperar el siguiente bloque/grupo de recursos
      - Un “ultimo” parámetro es retornado con el último recurso retornado
      - Al especificar el parámetro, se retornan los recursos desde el siguiente al parametro

DDD

Domain: Se refiere a una temática específica para la cual se está desarrollando el proyecto

DDD (Domain Driven Design):

Tiene como objetivo reducir la complejidad requerida para desarrollar una solución al implementarla lo más cercana posible al dominio de la solución al problema. Para esto el desarrollador debe trabajar de mano con el experto en el dominio, creando así un lenguaje que ambos comprenden (Domain Language o lenguaje Ubiquitous).

El objetivo final de DDD es ayudar a construir sistemas complejos que son más mantenibles, entendibles y que nos ayuden a alcanzar la funcionalidad de negocio requerida modelando nuestro sistema en el dominio mismo.

Ventajas y desventajas:

Ventajas	Desventajas
Comunicación entre desarrolladores y conocedores del negocio	Requiere experiencia en el dominio
Flexibilidad	Costoso (en tiempo y recursos)
Mantenibilidad	

Mejor para proyectos que son:

- A largo plazo
- Tenga una alta complejidad de dominio
- Tenga claros beneficios en comunicación usando un lenguaje de dominio

Conceptos básicos:

- **Domain:** El área temática en la cual nuestro programa va a ser constuido
- **Domain Model:** Un modelo conceptual que representa las diferentes partes del dominio, las cuales serán usadas en nuestro software. Incluyendo tanto comportamiento como datos.
- **Bounded Context:** El contexto al cual se le puede aplicar un modelo. En un proyecto complejo usualmente se tienen múltiples dominios, cada uno con su propio bounded context
- **Ubiquitous language:** El lenguaje en común usado entre el equipo y el código en si

- **Entity (Entidad):** Un objeto que es definido por un hilo de continuidad y es identificado mediante sus atributos
- **Value:** Un objeto que tiene atributos, pero no identidad. Puede ser tratado como inmutable
- **Aggregate:** Una colección de values y entities las cuales están agrupadas juntas por una entidad raíz, conocida como "Aggregate root". Encapsula estos objetos y los objetos externos no deben hacer referencia a sus miembros, lo cual garantiza la consistencia de los cambios realizados en el agregado.
- **Domain Event:** Un evento directamente relacionado al dominio

Patrones de diseño:

- **Factory:** Patrón creacional para encapsular la lógica creando Domain Objects tales como entidades y agregados. Previene al cliente de conocer el funcionamiento interno de la manipulación de objetos para que estos puedan intercambiarse sin su conocimiento.

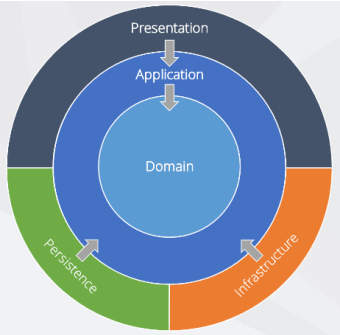
  Funcionamiento: El código llama un método factory, donde este invoca un constructor privado y retorna una instancia del objeto de dominio

- **Repository:** Es un patrón relacionado a la persistencia. Coloca una capa de abstracción en la parte superior de la capa de acceso a datos. De esta manera, los métodos para recuperar y almacenar objetos de dominio deben dedicarse a un objeto Repository especializado. Lo que significa que la lógica de la aplicación no es consciente de la implementación de estos métodos relacionados con el almacenamiento de datos y nos da la ventaja de que las implementaciones alternativas puedan intercambiarse fácilmente sin cambiar la lógica de la aplicación o del dominio.

- **Service/Infraestructura:** Es un patrón para crear objetos que manipulan nuestros principales objetos de dominio sin ser parte del dominio en sí. El servicio y sus operaciones, que no pertenecen como tal al dominio, deben ser stateless y deben tener terminología de lenguaje ubiquitous. Son usados por los Domain Objects para realizar esas operaciones que no pertenecen al dominio.

Component Architecture: Onion Architecture

Existen capas en donde las capas externas pueden depender de las capas internas, pero las capas internas no pueden depender o referenciar las capas externas



- **Capa de dominio:** Contendrá nuestros modelos de dominio y su lógica relacionada. En Spring serían las clases
- **Capa de Aplicación:** Se encuentran la lógica a nivel de la aplicación. En Spring serían los @Service
- **Capa de Service:** Dependiendo del componente puede representar lógica de un Web API o interfaz de usuario. En Spring los @Controller.
  Esta también suele ser la capa de infraestructura, en la cual se implementan conexiones a servicios externos a la app como DB o APIs. En Spring pueden ser implementaciones de Hibernate

Pros y contras de Microservicios frente a monolito:

- **Pros:**
  - Mejoras en atributos de calidad como la escalabilidad, la disponibilidad y la mantenibilidad
  - Se pueden realizar equipos de gran cantidad de desarrolladores y que estos programen al mismo tiempo distintas partes.
  - Realizar cambios es mas sencillo y menos costoso que en el monolito
- **Contras:**
  - El costo, puede llevar mucho tiempo y recursos realizar un sistema basado en microservicios

Pub/Sub:

Pub/Sub permite que los servicios se comuniquen de forma asíncrona, con latencias del orden de 100 milisegundos.

Pub/Sub permite crear sistemas de productores y consumidores de eventos, llamados publishers y subscribers. Los publishers se comunican con los subscribers de forma asíncrona mediante la difusión de eventos, en lugar de mediante “synchronous remote procedure calls” (RPCs).

- **Conceptos básicos**
  - **Topic:** Un recurso con nombre al que los editores envían mensajes.

- **Subscription**: Un recurso con nombre que representa el flujo de mensajes de un único tema específico, que se entregará a la aplicación suscriptora.
- **Message**: La combinación de datos y atributos (opcionales) que un editor envía a un tema y que finalmente se entrega a los suscriptores.
- **Message attribute**: Un par clave-valor que un editor puede definir para un mensaje. Por ejemplo, la clave iana.org/language_tag y el valor es podrían añadirse a los mensajes para marcarlos como legibles por un suscriptor de habla inglesa.
- **Publisher**: Una aplicación que crea y envía mensajes a un tema o temas.
- **Subscriber**: Una aplicación con una suscripción a un tema(s) para recibir mensajes del mismo.
- **Acknowledgement (or "ack")**: Una señal enviada por un suscriptor a Pub/Sub después de haber recibido un mensaje con éxito. Los mensajes aceptados se eliminan de la cola de mensajes de la suscripción.
- **Push and pull:** Los dos métodos de entrega de mensajes. Un suscriptor recibe los mensajes ya sea porque Pub/Sub los empuja al punto final elegido por el suscriptor, o porque el suscriptor los extrae del servicio.

Containerization

Container

Dentro de ellos podemos alojar todas las dependencias que nuestra aplicación necesite para ser ejecutada: empezando por el propio código, las librerías del sistema, el entorno de ejecución o cualquier tipo de configuración. Desde fuera del contenedor no necesitamos mucho más. Dentro están aislados para ser ejecutados en cualquier lugar. Es como una instancia de la Imagen

Image

Una imagen es una representación estática de la aplicación o el servicio y de su configuración y las dependencias. Para ejecutar la aplicación o el servicio, se crea una instancia de la imagen de la aplicación para crear un contenedor, que se ejecutará en el host de Docker. Las imágenes se almacenan en un registro, que sirve como biblioteca de imágenes.

Docker

Docker es una plataforma de software que le permite crear, probar e implementar aplicaciones rápidamente. Docker empaqueta software en unidades estandarizadas llamadas contenedores que incluyen todo lo necesario para que el software se ejecute, incluidas bibliotecas, herramientas de sistema, código y tiempo de ejecución. Con Docker, puede implementar y ajustar la escala de aplicaciones rápidamente en cualquier entorno con la certeza de saber que su código se ejecutará.

Kubernates

Kubernetes es una plataforma portable y extensible de código abierto para administrar cargas de trabajo y servicios. Kubernetes facilita la automatización y la configuración declarativa. Tiene un ecosistema grande y en rápido crecimiento. El soporte, las herramientas y los servicios para Kubernetes están ampliamente disponibles.

CI/CD

CI:

La integración continua consiste en hacer integraciones automáticas de un proyecto lo más a menudo posible. Se entiende por integración la compilación y ejecución de pruebas de todo un proyecto. El proceso se lleva a cabo normalmente cada cierto tiempo y su función es la descarga de las fuentes desde el control de versiones, su posterior compilación, la ejecución de pruebas y generar informes.

Jenkins:

Es un software de integración continua que permite a los desarrolladores detectar y solucionar problemas de integración de forma continua, evitando el caos de última hora cuando se acercan las fechas de entrega. Además, siempre se dispone de una versión para pruebas, demos o no lanzamientos anticipados. Se pueden ejecutar de forma inmediata las pruebas unitarias y hay una monitorización continua de las métricas de calidad del proyecto.

Jenkins es un sistema que corre en un servidor que es un contenedor de servlets, como hace también Apache Tomcat. Puede trabajar con herramientas de control de versiones como CVS, Subversion, Git, Mercurial, Perforce y Clearcase y es capaz de ejecutar proyectos basados en Apache Ant y Apache Maven, así como scripts de shell y programas batch de Windows.

CD:

Se refiere a la distribución o la implementación continuas, los cuales son conceptos relacionados que suelen usarse indistintamente. Ambos conceptos se refieren a la automatización de las etapas posteriores del canal, pero a veces se usan por separado para explicar la cantidad de automatización que se está incorporando.

La distribución continua se refiere a los cambios que implementa un desarrollador en una aplicación, a los que se les realizan pruebas de errores automáticas y que se cargan en un repositorio (como GitHub o un registro de contenedor), para que luego el equipo de operaciones pueda implementarlos en un entorno de producción en vivo.

Spring

Creado por Rod Johnson

Dependency Injection:

- Cuando una dependencia necesitada es inyectada por otro objecto
- La clase a la que se le inyecta no tiene responsabilidad de instanciar el objeto inyectado (No hay new)
- Tipos de inyección:
  - ¿Propiedad?: Por medio de un atributo publico/privado donde se le asigna directamente una instancia de la clase necesitada
  - Por medio de los setters
  - Por medio del **constructor**
- Recomendable usar interfaces, debido a que:
  - Permite decidir la implementación a inyectar en tiempo de ejecución
  - Sigue el principio de segregación de interfaces SOLID
  - Código mejor testeable
- Se refiere a la composición de las clases, pues uno compone sus clases con la inyección de dependencias en mente

- **Inversión de control:**
  - Es una técnica que permite inyectar dependencias en tiempo de ejecución
  - Las dependencias no están predeterminadas
  - Le da poder a los frameworks para funcionar como esqueletos extensibles
  - Es el ambiente de ejecución del código o el contexto
- **Buenas practicas**
  - Usar la inyección por constructor
  - Usar propiedades finales para componentes inyectados
  - Programar respecto a una interfaz
- El Autowired solo es necesario cuando se usa DI por medio de una propiedad o por medio de los setters. Con constructores no es necesario.
- Al tener más de una implementación de una interfaz: Spring no sabe cuál inyectar por lo que hay que usar la anotación @Qualifier y @Primary o @Profile
  - @Qualifier(“nombreBean”): Se usa antes del parámetro de la dependencia necesaria en el constructor del que la requiere y se le pasa el nombre de la implementación deseada con la primera letra en minúscula. También se puede poner la anotación encima del método setter o de la propiedad.
  - @Primary: Se coloca en la implementación que se desea sea priorizada, por lo que en todo lado donde no haya un @Qualifier se va a usar dicha implementación.
  - @Profile(“nombrePerfil”): Sirve para definir un tipo de idioma o implementación que se va a manejar durante todo el programa. La anotación se coloca en la implementación deseada para dicho perfil y dependiendo del perfil definido en el .properties (spring.profiles.active = nombrePerfil) se usa una implementación u otra. Se puede tener más de un perfil designado para una

implementación, así mismo la que tenga el perfil “default” va a ser el que se use si no se especifica un perfil en el .properties.

- Para cambiarle el nombre a un bean de Spring se coloca en la anotación el nombre deseado. Por ejemplo: @Service(“nombre”)

Ciclo de vida de un Bean:

1. Se instancia el objeto
2. Se inyectan propiedades desde el archivo .properties
3. Se llama el método setBeanName de la interfaz de BeanNameAware que puede ser implementado si uno lo desea a su manera
4. Se llama el método setBeanFactory de la interfaz BeanFactoryAware
5. Se llama el método setApplicationContext de la interfaz ApplicationContextAware
6. PreInicialización con BeanPostProcessors
7. Se llama el método afterPropertiesSet de la interfaz InitializingBeans
8. Se puede realizar una inicialización personalizada (Custom Init Method)
9. Post Inicialización mediante la interfaz BeanPostProcessors
10. Ya se puede usar el Bean

Cuando se apaga el servidor/contenedor:

1. Se llama el método destroy() de la interfaz DisposableBean, el cual se puede implementar
2. Después de eso el Bean ha sido destruido

Interfaces CallBack

- InitializingBean.afterPropertiesSet()
  - Paso 6
- DisposableBean.destroy()
  - Se llama al apagar el API Rest

Anotaciones de Ciclo de Vida

- @PostConstruct: Se coloca en los métodos que van a ser llamados después de que el vean haya sido creado/construido, pero antes de que sea retornado al objeto que lo requiere
- @PreDestroy: Se llama justo antes de que el bean sea destruido por el contenedor

Bean Post Processors:

Da una razón para interactuar con el ciclo de vida del contexto de Spring y con los beans mientras son procesados. Se debe implementar la interfaz BeanPostProcessor:

- postProcessBeforeInitialization: Se llama antes del método de inicialización del Bean
- postProcessAfterInitilization: Se llama después de haber inicializado el bean

Aware Interfaces:

Spring tiene alrededor de 14 Interfaces 'Aware', las cuales son usadas para acceder a la infraestructura de Spring Framework. Son utilizados en gran medida dentro del framework pero rara vez los usan los desarrolladores de Spring.

Aware Interface	Description
ApplicationContextAware	Interface to be implemented by any object that wishes to be notified of the ApplicationContext that it runs in.
ApplicationEventPublisherAware	Set the ApplicationEventPublisherthat this object runs in.
BeanClassLoaderAware	Callback that supplies the bean class loader to a bean instance.
BeanFactoryAware	Callback that supplies the owning factory to a bean instance.
BeanNameAware	Set the name of the bean in the bean factory that created this bean.
BootstrapContextAware	Set the BootstrapContext that this object runs in.

Aware Interface	Description
LoadTimeWeaverAware	Set the LoadTimeWeaver of this object's containing ApplicationContext.
MessageSourceAware	Set the MessageSource that this object runs in.
NotificationPublisherAware	Set the NotificationPublisher instance for the current managed resource instance.
PortletConfigAware	Set the PortletConfig this object runs in.
PortletContextAware	Set the PortletContext that this object runs in.
ResourceLoaderAware	Set the ResourceLoader that this object runs in.
ServletConfigAware	Set the ServletConfig that this object runs in.
ServletContextAware	Set the ServletContext that this object runs in.

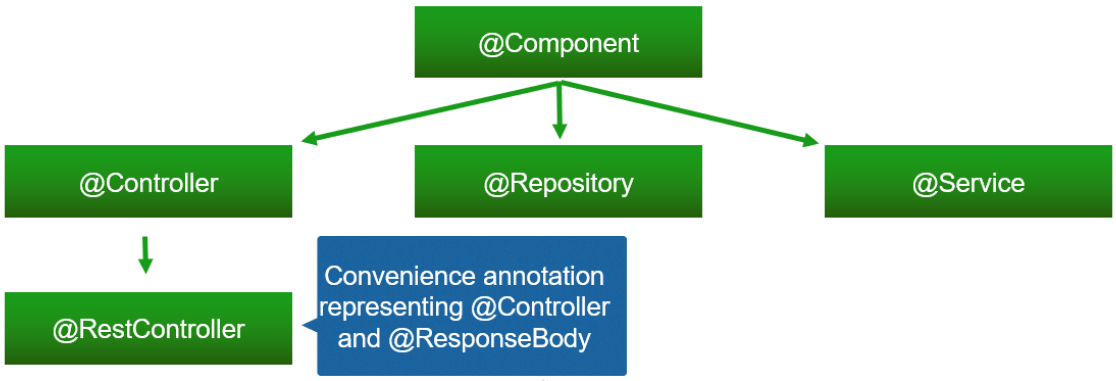
Notas:

- Para generar un banner personalizado: www.patorjk.com. Se debe colocar el resultado en un txt llamado "Banner.txt" y ubicado en resources
- Es recomendable usar interfaces para los servicios
- Se puede manejar una interfaz genérica estilo CrudService<T, ID> para que las interfaces de los servicios extiendan de dicha interfaz (Con métodos genéricos)
- Si no se usa una base de datos se puede manejar un Servicio abstracto donde se maneje un mapa que guarde los objetos.

Configuración

- **Estereotipo:** Una imagen fija o un conjunto de características las cuales representan un tipo particular de persona o cosa

- En Spring, los estereotipos son anotaciones usadas para definir Spring Beans
- Las clases anotadas son detectadas por medio del ComponentScan para agregar una instancia de la clase al contexto de Spring
- Estereotipos disponibles: @Component, @Controller, @RestController, @Repository, @Service
- Spring Boot provee una configuración predefinida para Spring Framework

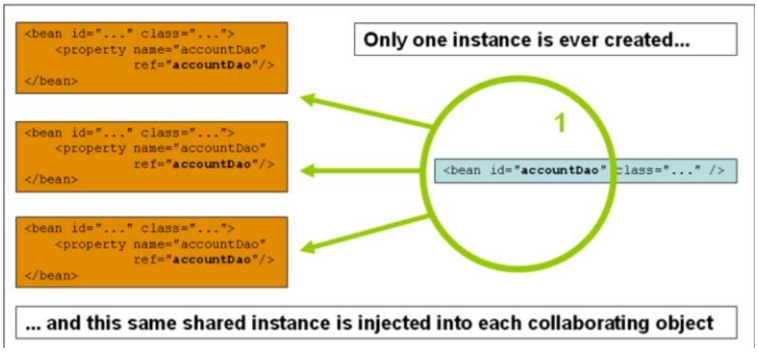


- **ComponentScan:** Permite indicarle a Spring desde que paquete o paquetes debe escanear para identificar Beans
- **@Configuration:**
  - Se puede tener más de uno en la aplicación
  - La clase con esta anotación permite crear Beans dentro de este mediante métodos que retornen la instancia del Bean y que este esté marcado con la anotación @Bean, y se le puede indicar el nombre del Bean.
  - Así mismo se pueden usar sobre estos Beans las anotaciones de @Primary y @Profile.
  - Para un Bean que reciba por parámetro otro, solo basta con crear el bean que recibe e indicarlo en los parámetros del que lo necesita.
  - **Factory Bean:** Se puede definir un Bean de una clase Factory que tenga un método que retorne una instancia de implementación de una interfaz dependiendo de un indicador y que en la configuración se creen los beans que llamen al Factory con su respectivo indicador e indicando con un @Profile cual se va a activar
- Si se desea se puede manejar un xml con los beans a utilizar en vez de colocarlos en el código.
  - Se deben usar etiquetas tipo bean dentro de la etiqueta beans. Donde se especifica el nombre del bean y la ubicación de la clase

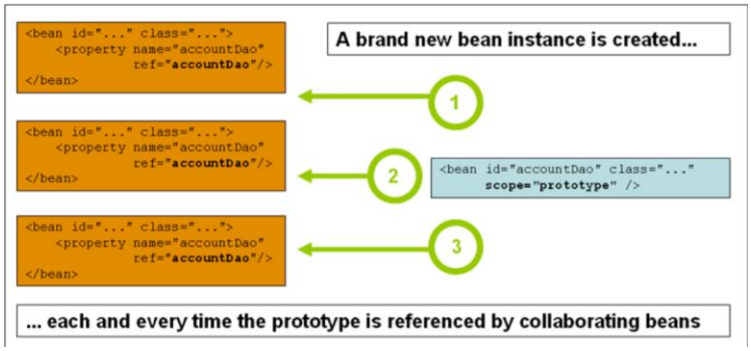
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean name="constructorGreetingService" class="guru.springframework.sfgdi.services.ConstructorGreetingService"/>
</beans>
```

- Se debe colocar la anotación @ImportResource(“classpath:archivo.xml”) en la clase del @SpringBootApplication o en la clase del @Configuration

- **Scopes:**
  - **Singleton:** Solo una instancia del bean es creada en el contenedor IoC



  - **Prototype:** Se crea una instancia cada vez que se requiera el bean



  - **Request:** Se crea una instancia por cada http request. Solo está disponible en el contexto de un web-aware SpringApplicationContext
  - **Session:** Se crea una instancia por cada http session. Solo está disponible en el contexto de un web-aware SpringApplicationContext
  - **Global-Session:** Se crea una instancia por cada sesión global. Se usa típicamente en un contexto Portlet. Solo está disponible en el contexto de un web-aware SpringApplicationContext
  - **Application:** El bean tiene alcance dentro el ciclo de vida de un ServletContext. Solo está disponible en el contexto de un web-aware SpringApplicationContext
  - **Websocket:** El bean tiene alcance dentro el ciclo de vida de un ServletContext. Solo está disponible en el contexto de un web-aware SpringApplicationContext
  - **Custom Scope:** Uno puede crear su propio Scope implementando Spring ‘Scope’ interface. No se pueden sobreescribir el built in los Scopes de Singleton y Prototype.
  - El scope se puede especificar tanto en el XML como con una anotación @Scope sobre el bean

Propiedades Externas

- **¿Por qué usarlos?:**
  - Valores de codificación rígida que puedan cambiar son considerados una mala practica
  - Hacen tu aplicación rígida y difícil de cambiar
  - Que la aplicación sea portable: El artefacto de despliegue (jar/war) debe poderse desplegar en diferentes ambientes

- **¿Qué puede cambiar?**
  - Nombres de usuarios
  - Contraseñas
  - Urls
  - API Keys
  - Paths
  - Queue names

- **¿Cómo indicar las propiedades externas?**
  - Argumentos por línea de comandos
  - SPRING_APPLICATION_JSON – Objeto JSON por línea de comandos o variable de entorno
  - JNDI
  - Variables de entorno del Sistema Operativo
  - Archivos de Propiedades (.properties o .yaml)

- **Spring Cloud Config:**
  - La aplicación de Spring inicia y obtiene valores de la configuración de un servidor
  - El proceso Bootstrap se realiza antes del inicio normal de Spring

- Se usa la anotación @PropertySource("classpath: archivo.properties") para indicar un .properties distinto al application.properties.
- Se usa @Value("${propiedad}") para indicar que el parámetro de un bean se llena por propiedad
- Desde línea de comandos se puede modificar la propiedad: --propiedad = valor
- En orden de prioridad están ubicadas de la siguiente forma: Línea de comandos, Variables de entorno y .properties/yml
- Se pueden crear varios application.properties de tal forma que pueda haber más de un perfil para usar el .properties (application-perfil.properties). Como puede ser un application-dev.properties o un application-qa.properties

- Para indicar el perfil o los perfiles de properties a usar se debe colocar el nombre del perfil en la sección spring.profiles.active y el ultimo perfil indicado sobreescribe las propiedades que ya existan
- En el .yml se estructura en forma de árbol las propiedades y se pueden indicar varias configuraciones en el mismo archivo con solo colocar "---" para separarlas.
- **Properties Binding:** Colocando sobre un Bean de configuración la anotación @ConfigurationProperties("prefijo_propiedad") se indica que los atributos de esa clase tienen una propiedad con el prefijo especificado y Spring setea los valores en los atributos de ese Bean. Usando Setters
- **Constructor Properties Binding:** Se debe colocar sobre el bean de configuración la anotación @ConstructorBinding y el @ConfigurationProperties("prefijo_propiedad") para que ingrese los datos de las propiedades mediante el constructor.
- En caso de que no sea un bean de configuración se debe indicar en un bean de configuración con la anotación @EnableConfigurationProperties(bean.class)
- Se pueden manejar los mensajes de la aplicación mediante properties

Developer Tools

- Se agrega a los proyectos mediante artefacto de Maven: 'spring-boot-devtools'
- Se desactiva en cuanto se ejecuta una aplicación empaquetada
- Tiene un reinicio automático:
    - Reacciona al cambio en las clases y reinicia el contexto de Spring
    - Usa dos cargadores de clases: Uno para la aplicación y otro para las dependencias proyectos jar
    - El reinicio es muy rápido, solo se recargan las clases de la aplicación
- En IntelliJ toca:
    - Por defecto se requiere seleccionar 'Build/Make Project'
- Template Caching:
    - Por defecto las plantillas son cacheadas por rendimiento
    - Caching requiere que el contenedor se reinicie para refrescar la cache
    - Developer Tools desactivará Template Caching, por lo que no se requiere reiniciar para ver cambios
- LiveReload: Es una tecnología que permite actualizar el navegador a penas hayan cambios

JPA – Hibernate:

- **@OneToOne:** El lado propietario es donde se define la llave foránea. Se coloca en el atributo de relación en ambas clases. La definición de cascada se coloca en el lado propietario

- **@OneToMany:** El lado propietario es donde está el Many (donde está la llave foránea). El cascade se coloca aquí, además de colocar el mappedBy = "nombre de la variable al lado del many"
- **@ManyToOne:** El lado propietario es donde está el Many (donde está la llave foránea). Se usa un @JoinColumn(name = "columna")
- **@ManyToMany:** Se coloca la anotación a ambos lados y en el lado propietario se coloca una anotación

  @JoinTable(name = "tabla", joinColumns = @JoinColumn(name = "columna"), inverseJoinColumns = @JoinColumn(name = "columnaInversa"))

  Y en el lado no propietario se coloca:
  @ManyToMany(mappedBy="atributo_propietario")

- Al usar el 'mappedBy' se define el atributo que posee la referencia de la relación
- **Fetch Type:**
  - Lazy Fetch Type: Los datos no son consultados hasta que sean referenciados
  - Eager Fetch Type: Los datos son consultados "up front" (¿en la delantera?)
  - Por defecto:
    - **@OneToOne:** Eager
    - **@OneToMany:** Lazy
    - **@ManyToOne:** Eager
    - **@ManyToMany:** Lazy
- **JPA Cascade Types:** Controla como los cambios de estado son activados en Cascada desde los objetos padre hacia los hijos
  - **Persist:** Operaciones sin riesgo van a afectar en cascada las entidades relacionadas
  - **Merge:** Las entidades relacionadas son unidas cuando la entidad propietaria es unida
  - **Refresh:** Las entidades relacionadas son refrescadas cuando la entidad propietaria es refrescada
  - **Remove:** Remueve todas las entidades relacionadas cuando la entidad propietaria es eliminada
  - **Detach:** Se separan todas las entidades si una separación manual ocurre
  - **All:** Aplica todas las opciones anteriores
  - Por defecto, no se aplica ninguna operación
- **Embeddable Types:**
  - Son usadas para definir un conjunto de propiedades comunes
  - Suelen ser clases que encapsulan información, pero no tiene como tal una tabla en la DB, sino que la clase tiene la información de una tabla donde esta clase es un tipo embebido.
- **Herencia:**

    - **MappedSuperClass:** Las entidades heredan de una super clase. Una tabla no es creada por la super clase.
    - **Single Table (Opción por defecto en hibernate):** Una tabla es usada por todas las subclases
    - **Joined Table:** Clases base y subclases tienen sus propias tablas. Consultar una entidad de subclase requieres un Join con la tabla padre
    - **Table per Class:** Cada subclase tiene su propia tabla

- **Crear y actualizar Timestamps:**
    - Usualmente la mejor práctica para crear y actualizar timestamps en sus entidades para propósitos de auditoria
    - JPA soporta @PrePersit y @PreUpdate los cuales pueden ser usados para soportar timestamps de auditoria mediante callbacks de ciclos de vida de JPA
    - Hibernate provee @CreationTimestamp y @UpdateTimestamp

- **@Lob:** Anotación para objetos grandes, representa objetos en la db como CLOBS (Characters - String) o BLOBS (Binary – Byte[])
- Si se necesita un enumerado basta con usar la anotación @Enumerated(value=EnumType.Type) sobre el atributo deseado
- **Hibernate DDL (Data Definition Language) Auto:**
    - La propiedad en Spring es “spring.jpa.hibernate.ddl-auto”
    - Opciones: none, validate, update, create, create-drop
    - Para DB en memoria como lo es h2, spring usará siempre create-drop
    - Para inicializar la DB se puede usar un import.sql ubicado en la carpeta de resources y la opción de DDL-Auto debe estar en create o create-drop
    - **DML:** Data Modeling Language
    - DDL sirve para crear la estructura de las tablas
    - DML sirve para realizar operaciones a las tablas como inserts o updates

- **Spring JDBC**
    - Spring DataSource carga por defecto un schema.sql y un data.sql
    - También cargará de schema-${plataform}.sql y data-${plataform}.sql
        - Esto definiendo la propiedad “spring.datasource.platform” con el nombre de la plataforma
    - Puede hacer conflicto con la inicialización de la propiedad de Hibernate DDL-Auto, se puede evitar el conflicto colocando dicha propiedad en ‘none’ o ‘validate’

- **Spring Data JPA Query Methods:**
    - Existen métodos del repositorio que no es necesario asignarles un Query explicito pero que, con solo definir la firma, el framework interpreta y sabe que query debe ir en ese método. Por ejemplo: findByName

- En relaciones bidireccionales se recomienda modificar un poco los setters para recibir el objeto y colocarle la relación con esta clase

Testing

Unit Testing:

- Diseñadas para probar secciones especificas de código, e idealmente se debe tener una cobertura del 70-80%
- Deben ser únicas y se deben ejecutarse muy rápido y no deben tener dependencias externas.

Integration Test:

- Diseñado para probar comportamientos entre objetos y partes del sistema completo
- Tiene un alcance mucho más amplio
- Puede incluir el contexto Spring, bases de datos y mensajes de brokers
- Van a correr mucho más lento que las unitarias

Functional Tests:

- Se prueba la aplicación mientras se encuentra corriendo
- La aplicación se encuentra desplegada en un ambiente conocido
- Se prueban puntos de contacto funcionales

TDD (Test Driven Development): Escribir los tests primero, lo cual va a fallar para que ahora se escriba el código que arreglará el test

BDD (Behavior Driven Development): Sigue TDD e indica que la prueba de cualquier unidad de software puede ser especificada en términos del comportamiento deseado de la unidad

Mock: Una implementación falsa de clases usadas en el testing

Spy: Es un mock parcial que permite sobre escribir métodos selectos de una clase real

La dependencia spring-boot-starter-test incluye dependencias como las de Junit, Mockito entre otras.