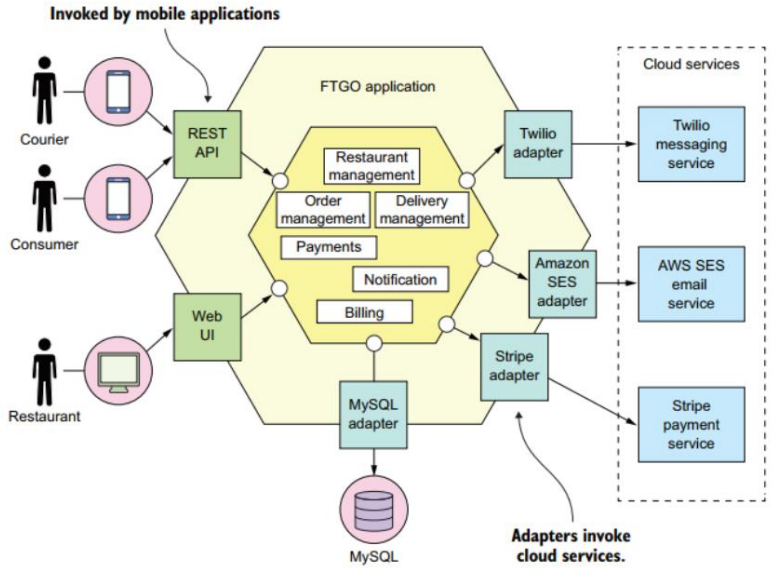
Conocimientos básicos:

1. Arquitectura de Microservicios

Arquitectura Monolítica (WAR)

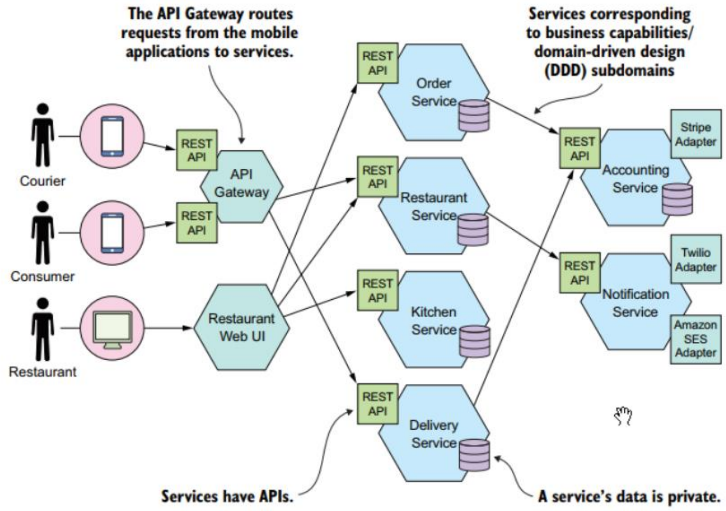
La arquitectura de la aplicación FTGO



- Microservicios: Es un estilo arquitectónico que descompone funcionalmente una aplicación en un conjunto de servicios.
- Los microservicios son altamente modulares.
- Son pequeños y fáciles de mantener.
- Son escalables (pueden modificarse de manera independiente unos de otros).
- Pueden aislar fácilmente sus fallas.
- Permite la flexibilidad tecnológica (se puede migrar de una tecnología a otra).

Arquitectura Microservicios

La arquitectura de microservicios FTGO



Continuos Delivery/Deployment:

- Capacidad de prueba requerida.
- Capacidad de implementación requerida.
- Equipos de desarrollo autónomos y débilmente acoplados.

2. Principios SOLID

Principios para el desarrollo de software de manera eficaz:

- **Principio de responsabilidad única:** Cada objeto (clase) debe hacer únicamente una cosa para evitar cambios innecesarios.
- **Principio Open/Closed:** Las clases deben estar abiertos a la extensión y cerrados al cambio.
- **Principio de Sustitución Liskov:** Cuando extendemos una clase la clase hija no debe modificar el comportamiento de la clase padre. La prueba se realiza probando que código siga siendo valido si se reemplaza la clase padre con su clase hija.
  Esto garantiza el correcto uso de la herencia.
- **Principio de segregación de interfaces:** Ninguna clase debería depender de métodos que no utiliza. Esto obliga a que los métodos detallados en las interfaces deben representar los comportamientos del objeto que abstraen; en caso contrario se deben crear interfaces más pequeñas para desacoplar módulos entre sí.
  Se debe evitar tener funciones en una interface que cuando extiendan no tengan implementación, estos casos son conocidos como fat interface y se dan cuando se desnaturalizo el propósito de la interface.
- **Principio de inversión de dependencias:** Este principio ayuda al código a ser testeable y mantenible, indica que las clases de alto nivel no deben depender de las clases de bajo nivel. Para ello las abstracciones no deberían depender de los detalles sino al contrario los detalles deberían depender de las abstracciones (clases).

S ingle Resposibility Principle
A class should have only a single responsibility (i.e. only one potential change in the software's specification should be able to affect the specification of the class)

O pen / Closed Principle
A software module (it can be a class or method) should be open for extension but closed for modification.

L iskov Substitution Principle
Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

I nterface Segregation Principle
Clients should not be forced to depend upon the interfaces that they do not use.

D ependency Inversion Principle
Program to an interface, not to an implementation.

3. Programación Funcional

3.1. Interfaz Funcional: Las interfaces funcionales son interfaces que tienen un método a implementar, es decir, un método abstracto. Esto significa que cada interfaz creada que respeta esta premisa se convierte automáticamente en una interfaz funcional :

- Supplier: método get(): Solo devuelve un valor.
- Consumidor y Biconsumidor: Reciben variables pero no devuelve ningún valor. Ejemplo:
  Consumidor: accept(T);
  Biconsumidor: Biconsumer<Integer,Integer> printSum = (number1,number2) → System.out.println(number1+number2);
- Predicado y BiPredicado: Es una interface que recibe parámetros y realiza una condición de validación devolviendo true o false. Ejemplo:
  Predicado: filter(number -> number % 2 == 0)
  BiPredicado: Bipredicate<String,Integer> valid = (string1,integer1) -> {return string1.length() == integer1;**}**
  * Esta interface siempre debe devolver true o false.
- Función y BiFunción: Está interface es la más común y como indica su nombre es una función que recibe parámetros y devuelve un valor según el caso. Ejemplo:
  Función: map(number -> number.doubleValue())

BiFunción: Bifunction<Integer,Integer,Integer> sumNumber = (number1,number2) -> number1 + number2;
Integer sumAct = sumNumber.apply(t:1,u:2);

- UnaryOperator y BinaryOperator: Es igual a la interface función y bifunción pero solo recibe y devuelve un tipo de parámetro. Ejemplo:
  UnaryOperator: UnaryOperator<Integer> value = x -> x+2;
  BinaryOperator: number.stream().reduce((n1,n2)->n1+n2).ifPresent(System.out::println);
  * Esta interface se usa poco, normalmente se usa function la cual no limita el tipo de entregable.

3.2. Expresiones Lambda: Es una característica del java8 consiste en usar el operador lambda (->) y tiene varias características que permiten mejorar la lectura del código así como reducir la cantidad de líneas de código (considerando que por lo que se observa en el sonarqube no se pueden usar muchas líneas de código).

3.3. Method References: Similar a la expresión lambda pero en lugar de usar la lambda (->) se usa la referencia (::), se puede usar de 3 maneras:

- A un método estático: Para usar esto se necesita usar una instancia de la clase donde está declarado el método estático. Por ejemplo:
  Sayable sayable = MethodReference::saySomething;
  *De esta manera el objeto sayable (el cual es una interface) puede usar el método saySomething en como propio.

```
interface Sayable{
    void say();
}
public class MethodReference {
    public static void saySomething(){
        System.out.println("Hello, this is static method.");
    }
    public static void main(String[] args) {
        // Referring static method
        Sayable sayable = MethodReference::saySomething;
        // Calling interface method
        sayable.say();
    }
}
```

- A una instancia: Para referenciar métodos no estáticos se necesita una instancia de la clase donde esta implementado el método a referenciar.

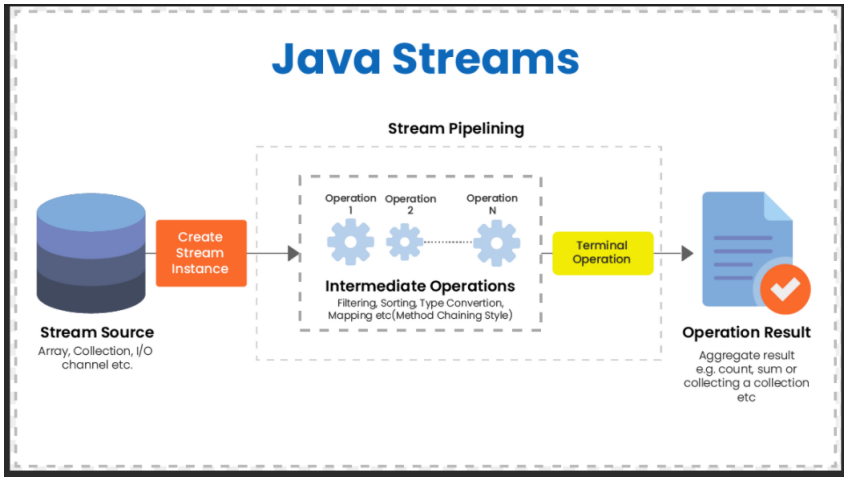
```java
interface Sayable{
    void say();
}
public class InstanceMethodReference {
    public void saySomething(){
        System.out.println("Hello, this is non-static method.");
    }
    public static void main(String[] args) {
        InstanceMethodReference methodReference = new InstanceMethodReference(); // Creating object
        // Referring non-static method using reference
            Sayable sayable = methodReference::saySomething;
        // Calling interface method
            sayable.say();
            // Referring non-static method using anonymous object
            Sayable sayable2 = new InstanceMethodReference()::saySomething; // You can use anonymous object also
            // Calling interface method
            sayable2.say();
    }
}
```

- Como un constructor: Se puede referenciar un constructor de manera que se puedan usar los métodos de la interface.

```java
interface Messageable{
    Message getMessage(String msg);
}
class Message{
    Message(String msg){
        System.out.print(msg);
    }
}
public class ConstructorReference {
    public static void main(String[] args) {
        Messageable hello = Message::new;
        hello.getMessage("Hello");
    }
}
```

3.4. **Stream API:** Interface de Java 8 que permite trabajar con flujos mediante la tipificación de objetos como stream(), esto se trabaja mediante manejo de colecciones iterables (que se puedan recorrer).

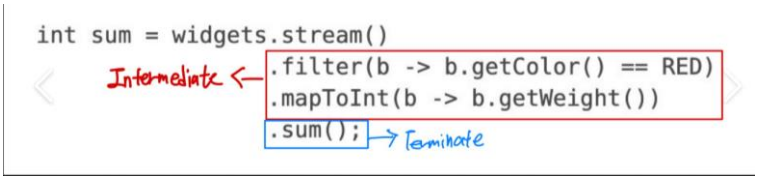
Ejemplo de estructura stream():



El stream tiene 2 tipos de operaciones:

- **Intermedias: Trabajan con los datos iterados de la colección y no se ejecutan si no tienen un operador terminal. Podemos poner varios de estos para considerar varias operaciones dentro de un flujo de trabajo.**
- **Finales: Devuelven un valor el cual está definido como un tipo de dato.**

Cada operación tiene un grupo de funciones que se pueden utilizar un valor.



3.5. **Clase Optional:** Es una clase nuevas incluida en el JDK de java 8 la cual controla las ejecuciones que generan las excepciones por null

Java Optional Example: If Value is not Present

```
import java.util.Optional;
public class OptionalExample {
   public static void main(String[] args) {
      String[] str = new String[10];
      Optional<String> checkNull = Optional.ofNullable(str[5]);
      if(checkNull.isPresent()){  // check for value is present or not
         String lowercaseString = str[5].toLowerCase();
         System.out.print(lowercaseString);
      }else
         System.out.println("string value is not present");
   }
}
```

Output:

```
string value is not present
```

Java Optional Example: If Value is Present

```
import java.util.Optional;
public class OptionalExample {
   public static void main(String[] args) {
      String[] str = new String[10];
      str[5] = "JAVA OPTIONAL CLASS EXAMPLE";// Setting value for 5th index
      Optional<String> checkNull = Optional.ofNullable(str[5]);
      if(checkNull.isPresent()){  // It Checks, value is present or not
         String lowercaseString = str[5].toLowerCase();
         System.out.print(lowercaseString);
      }else
         System.out.println("String value is not present");
   }
}
```

Output:

```
java optional class example
```

4. **Programación Reactiva:** Se analiza el manifiesto reactivo el cual establece los siguientes objetivos para un sistema:
   - Responsivo

- Resiliente
- Orientado a Mensajes (no bloqueante y todo llega a su ritmo)
- Elásticos

La comunicación no debe ser orientada a conexiones sino a envío de mensajes y se realiza mediante flujos de datos considerando los siguientes elementos:

- Observer: Observador que quiere obtener información.
- Observable: Generador de información.
- Suscribe: Entrega la información existente a los observadores suscritos.

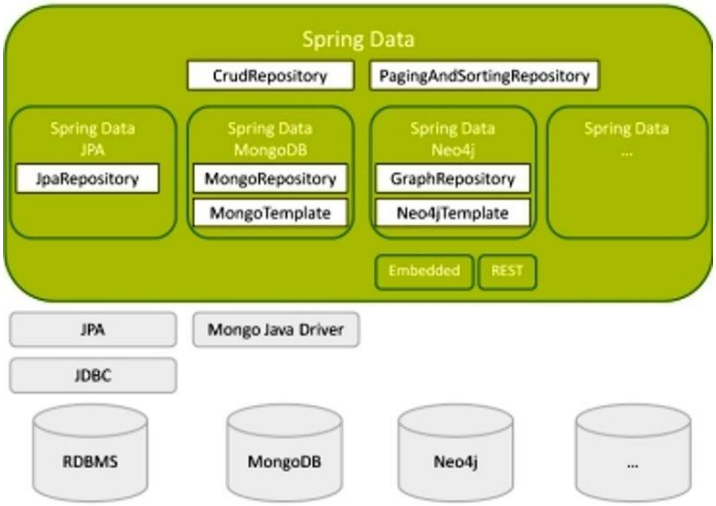
5. **Desarrollo de Microservicio:**
   - Spring Boot: Modelo de diseño de un proyecto que usa el MVC para desarrollo de aplicaciones web:
     - Endpoints (controller)
     - Services (Interfaces e implementaciones).
     - Datos (Persistencia con JPA o algo similar).
   - Spring WebFlux

     Es una versión de spring que utiliza llamadas asíncronas para el uso de hilos (threads) aprovechando la programación reactiva (reactive streams).

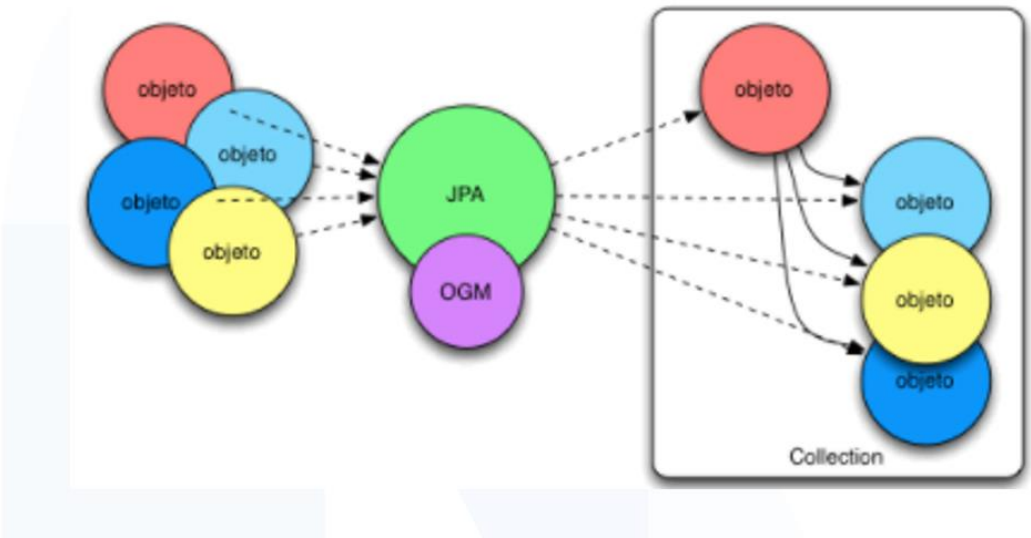
     * Considerar que se puede realizar de varias maneras, en el material de referencia se trabaja con streams (flujos) que manejan maps(flujo de objetos) y flux (colección de flujos).
   - Lombok: Es un proyecto que consiste en el desarrollo de una librería de apoyo para el JDK 8 en adelante que permite el ahorro de código mediante el uso de etiquetas.

   - Spring Data JPA: Manejo de la capa de datos de spring

     Estructura:



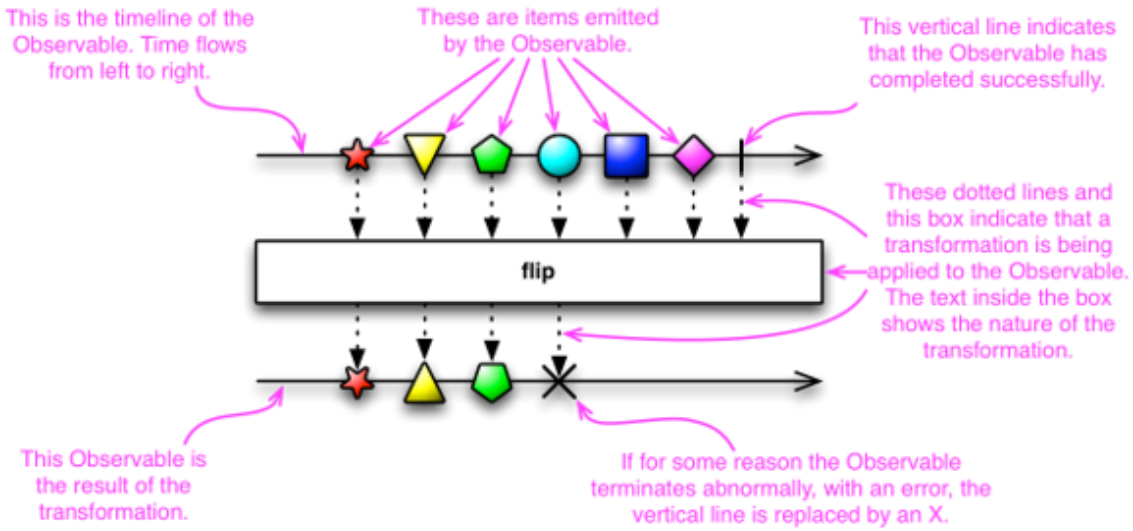
Funcionalidad:



El mongo DB es una base de datos No SQL por lo que trabaja basándose en entidades por lo que la información se obtiene a través de BSON (estos se convierten en JSON de clases de tipo @Entity) a través de un api de azure llamada cosmos DB.

Este diseño implica que cada elemento se considere una BD de una tabla y no exista anidación de objetos (cambio en el paradigma normal de construcción de clases y de normalización de las mismas).

- **RxJava**
  Estilo de programación con java 8 en adelante para acceso veloz a la información mediante el manejo de 3 actores:
  1. Observer: Espera la información.
  2. Observable: Emite la información.
  3. Suscriptor: Observador que desea obtener información de un observable, esta información viaja a través del suscriptor.



6. **Pruebas Unitarias:** Es la prueba de componentes de manera individual simulando los componentes que realizan entradas con la finalidad de probar las salidas, para ello se usa la librería okhttp3.mockwebserver. MockWebServer la cual permite simular servicios web. Los objetos que son necesario serán instanciados como mocks.

   Ver ejemplo: w

7. **Azure Básico:**
   SQL Database: Herramienta de azure para uso de BD a través de los servicios clouds lo que permite beneficios de disponibilidad y ahorro.
8. **DevOps**

DevOps es un marco de trabajo y una filosofía en constante evolución que promueve un mejor desarrollo de aplicaciones en menos tiempo y la rápida publicación de nuevas o revisadas funciones de software o productos para los clientes.

1. Git, GitFlow
   a. Git: Repositorio de fuentes que permite el control de versiones y el uso de ramas para trabajo distribuido.
   b. GitFlow: Aplicación del git mediante el cual se manejan dos rama estables (master y develop) en base a las cuales se desarrollan ramas para cada nuevo desarrollo (feature), adicionalmente se manejan dos conceptos de ramas release la cual se usa para corregir defectos encontrados en el develop sin necesidad de crear una rama feature nueva (nota personal nuna he usado esta rama) y la rama hotfix que se usa para soluciones de emergencia encontrados en el master (corregir errores de desarrollos ya liberados en producción sin pasar por el gitFlow, se usa poco y debe evitarse en extremo llegar a esto) ambas ramas la release y la hotfix deben ser eliminadas una vez que se termina su uso (para ello deben haber hecho merge con las ramas develop y master).
   * Recuerda que el tag existe para control de versiones.

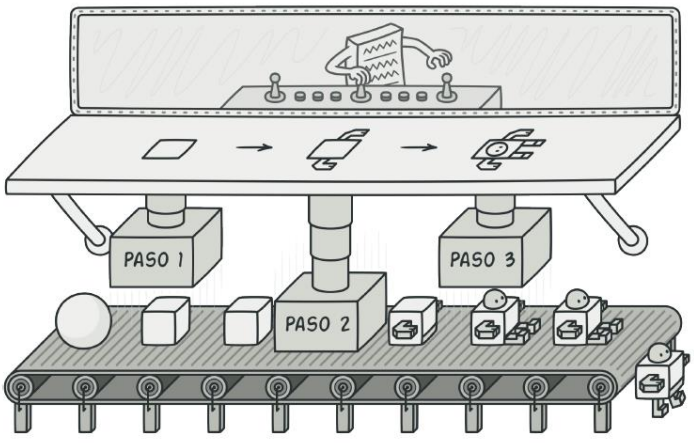
2. Jenkins: Orquestador que se encarga de ejecutar de manera automatizada un flujo para despliegue de aplicativos en ambientes indicados, facilita algunos aspectos de entrega continua.
3. SonarQube: Validador de código para controlar la calidad del mismo de manera estática. Se controlan 3 situaciones:
    - Code smells: codificación que funciona pero que podría estar mejor o directamente no respeta el estándar del mismo.
    - Bugs: Errores que no permiten la compilación del código.
    - Vulnerabilities: Problemas relacionados a la seguridad que potencialmente podría afectar la seguridad del ambiente donde se despliegue el aplicativo.

    La presencia de estos representa una deuda técnica, mientras mayor sea la presencia de estos mayor la deuda técnica y eso haría más complicada la labor de desarrollo a futuro.

4. Checkstyle: Plugin para revisión de código estático (es un sonar cube antes de subir al sonar cube) por lo que se puede ejecutar de manera local.

Conocimientos Intermedios:

1. **Patrones de Diseño de Software**
    - Creacionales: Patrones de creaciones de objetos enfocado en la flexibilidad y reutilización de objetos:
        - Factory Method: Creación de objetos en una superclase, mientras permite a las subclases alterar el tipo de objeto.
        - Abstract Factory: Permite producir familias de objetos relacionadas sin especificar sus clases concretas.
        - Builder: Permite construir objetos complejos paso a paso. Este patrón nos permite producir distintos tipos de representaciones de un objeto empleando el mismo código de construcción.



BUILDER

Elementos:

Builder: Interface donde se definen los elementos comunes de todos los objetos a construir. Se deben indicar funciones genéricas definiendo los parámetros de ingreso y salida.

builders/Builder.java: Common constructor interface

```java
package refactoring_guru.builder.example.builders;

import refactoring_guru.builder.example.cars.CarType;
import refactoring_guru.builder.example.components.Engine;
import refactoring_guru.builder.example.components.GPSNavigator;
import refactoring_guru.builder.example.components.Transmission;
import refactoring_guru.builder.example.components.TripComputer;

/**
 * Builder interface defines all possible ways to configure a product.
 */
public interface Builder {
    void setCarType(CarType type);
    void setSeats(int seats);
    void setEngine(Engine engine);
    void setTransmission(Transmission transmission);
    void setTripComputer(TripComputer tripComputer);
    void setGPSNavigator(GPSNavigator gpsNavigator);
}
```

Producto Builder [Constructores concretos]: Clase que hace implementación del builder. Debe contener las variables utilizadas por los métodos de la interface e implementar los métodos (según el principio de sustitución de Liskov y como prueba del builder ninguna función debe quedar sin implementar o devolviendo null).

```java
public class CarBuilder implements Builder {
    private CarType type;
    private int seats;
    private Engine engine;
    private Transmission transmission;
    private TripComputer tripComputer;
    private GPSNavigator gpsNavigator;

    public void setCarType(CarType type) {
        this.type = type;
    }

    @Override
    public void setSeats(int seats) {
        this.seats = seats;
    }

    @Override
    public void setEngine(Engine engine) {
        this.engine = engine;
    }

    @Override
    public void setTransmission(Transmission transmission) {
        this.transmission = transmission;
    }

    @Override
    public void setTripComputer(TripComputer tripComputer) {
        this.tripComputer = tripComputer;
    }

    @Override
    public void setGPSNavigator(GPSNavigator gpsNavigator) {
        this.gpsNavigator = gpsNavigator;
    }

    public Car getResult() {
        return new Car(type, seats, engine, transmission, tripComputer, gpsNavigator);
    }
}
```

Producto: Clase donde se define el producto que se va a construir (es una abstracción sin embargo debe ser más concreta que los builders).

```java
/**
 * Car is a product class.
 */
public class Car {
    private final CarType carType;
    private final int seats;
    private final Engine engine;
    private final Transmission transmission;
    private final TripComputer tripComputer;
    private final GPSNavigator gpsNavigator;
    private double fuel = 0;

    public Car(CarType carType, int seats, Engine engine, Transmission transmission,
               TripComputer tripComputer, GPSNavigator gpsNavigator) {
        this.carType = carType;
        this.seats = seats;
        this.engine = engine;
        this.transmission = transmission;
        this.tripComputer = tripComputer;
        if (this.tripComputer != null) {
            this.tripComputer.setCar(this);
        }
        this.gpsNavigator = gpsNavigator;
    }

    public CarType getCarType() {
        return carType;
    }

    public double getFuel() {
        return fuel;
    }

    public void setFuel(double fuel) {
        this.fuel = fuel;
    }

    public int getSeats() {
        return seats;
    }

    public Engine getEngine() {
        return engine;
    }

    public Transmission getTransmission() {
        return transmission;
    }

    public TripComputer getTripComputer() {
        return tripComputer;
    }

    public GPSNavigator getGpsNavigator() {
        return gpsNavigator;
    }
}
```

*El código del ejemplo define muchas clases para elementos que son auxiliares como engine, transmisión, etc. y que no son relevantes para representar el patrón.

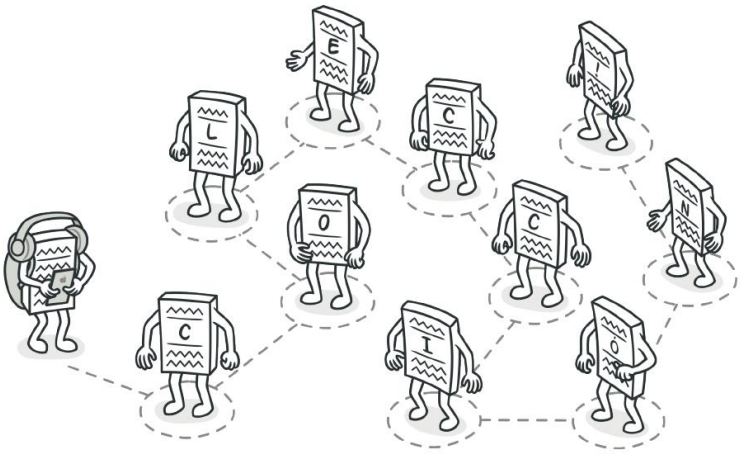
Director: Clase que define las acciones para la construcción de productos. Las ejecuciones de los builder se producen en esta clase mediante la invocación de sus funciones las cuales según las características implementará un producto distinto.

```
public class Director {

    public void constructSportsCar(Builder builder) {
        builder.setCarType(CarType.SPORTS_CAR);
        builder.setSeats(2);
        builder.setEngine(new Engine(3.0, 0));
        builder.setTransmission(Transmission.SEMI_AUTOMATIC);
        builder.setTripComputer(new TripComputer());
        builder.setGPSNavigator(new GPSNavigator());
    }

    public void constructCityCar(Builder builder) {
        builder.setCarType(CarType.CITY_CAR);
        builder.setSeats(2);
        builder.setEngine(new Engine(1.2, 0));
        builder.setTransmission(Transmission.AUTOMATIC);
        builder.setTripComputer(new TripComputer());
        builder.setGPSNavigator(new GPSNavigator());
    }

    public void constructSUV(Builder builder) {
        builder.setCarType(CarType.SUV);
        builder.setSeats(4);
        builder.setEngine(new Engine(2.5, 0));
        builder.setTransmission(Transmission.MANUAL);
        builder.setGPSNavigator(new GPSNavigator());
    }
}
```

Cliente: El “main” del patrón de diseño en la que se solicitará que el director construya el producto que se necesita invocar:

```
public class Demo {

    public static void main(String[] args) {
        Director director = new Director();

        // Director gets the concrete builder object from the client
        // (application code). That's because application knows better which
        // builder to use to get a specific product.
        CarBuilder builder = new CarBuilder();
        director.constructSportsCar(builder);

        // The final product is often retrieved from a builder object, since
        // Director is not aware and not dependent on concrete builders and
        // products.
        Car car = builder.getResult();
        System.out.println("Car built:\n" + car.getCarType());


        CarManualBuilder manualBuilder = new CarManualBuilder();

        // Director may know several building recipes.
        director.constructSportsCar(manualBuilder);
        Manual carManual = manualBuilder.getResult();
        System.out.println("\nCar manual built:\n" + carManual.print());
    }

}
```

    - Prototye: Permite copiar objetos existentes sin que el código dependa de sus clases.
    - Singleton: Permite asegurarnos que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.
    - Estructurales: Explican como ensamblar objetos y clases en estructuras más grandes, a la vez que se mantiene la flexibilidad y eficiencia de estas estructuras:
    - Adapter: Permite la colaboración entre objetos con interfaces incompatibles.
    - Bridge: Permite dividir una clase grande o un grupo de clases estrechamente relacionadas, en dos jerarquías separadas (abstracción e implementación) que pueden desarrollarse independientemente la una de la otra.
    - Composite: Permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales.
    - Decorator: Permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores especiales que contiene estas funcionalidades.
    - Facade: Proporciona una interfaz simplificada a una biblioteca, un framework o cualquier otro grupo complejo de clases.
    - Flyweight: Permite mantener más objetos dentro de la cantidad disponible de memoria RAM compartiendo las partes comunes del estado entre varios objetos en lugar de mantener toda la información en cada objeto.
    - Proxy: Permite proporcionar un sustituto o marcador de posición para otro objeto. Un proxy controla el acceso al objeto original, permitiéndote hacer algo antes o después de que la solicitud llegue al objeto original.
- De Comportamiento: Los patrones de comportamiento tratan con algoritmos y la asignación de responsabilidades entre objetos:
    - Chain of responsability: Permite pasar solicitudes a lo largo de una cadena de manejadores. Al recibir una solicitud, cada manejador decide si la procesa o si la pasa al siguiente manejador de la cadena.
    - Command: Convierte una solicitud en un objeto independiente que contiene toda la información sobre la solicitud. Esta transformación te permite parametrizar los métodos con diferentes solicitudes, retrasar o poner en cola la ejecución de una solicitud y soportar operaciones que no se pueden realizar.
    - Iterator: Permite recorrer elementos de una colección sin exponer su representación subyacente (lista, pila, árbol, etc.).



Objetivo:
Recorrer colecciones de manera que se pueda acceder a sus diferentes elementos de manera rápida y sencilla sin exponer el método.
Elementos:

- Iterador: Realiza el recorrido de colecciones mediante un método único devuelve elementos de la colección que recorre.

  Los iteradores guardan los métodos necesarios para recorrer la colección, por ejemplo, extraer el siguiente elemento, recuperar la posición actual, Etc.

  Técnicamente son interfaces iteradoras.

iterators/ProfileIterator.java: Defines profile interface

```
package refactoring_guru.iterator.example.iterators;

import refactoring_guru.iterator.example.profile.Profile;

public interface ProfileIterator {
    boolean hasNext();

    Profile getNext();

    void reset();
}
```

- Comportamiento: Conjunto de pasos para encontrar un elemento mediante el recorrido de una colección.

  Técnicamente son iteradores concretos.

```java
public class FacebookIterator implements ProfileIterator {
    private Facebook facebook;
    private String type;
    private String email;
    private int currentPosition = 0;
    private List<String> emails = new ArrayList<>();
    private List<Profile> profiles = new ArrayList<>();

    public FacebookIterator(Facebook facebook, String type, String email) {
        this.facebook = facebook;
        this.type = type;
        this.email = email;
    }

    private void lazyLoad() {
        if (emails.size() == 0) {
            List<String> profiles = facebook.requestProfileFriendsFromFacebook(this.emai
            for (String profile : profiles) {
                this.emails.add(profile);
                this.profiles.add(null);
            }
        }
    }

    @Override
    public boolean hasNext() {
        lazyLoad();
        return currentPosition < emails.size();
    }

    @Override
    public Profile getNext() {
        if (!hasNext()) {
            return null;
        }

        String friendEmail = emails.get(currentPosition);
        Profile friendProfile = profiles.get(currentPosition);
        if (friendProfile == null) {
            friendProfile = facebook.requestProfileFromFacebook(friendEmail);
            profiles.set(currentPosition, friendProfile);
        }
        currentPosition++;
        return friendProfile;
    }

    @Override
    public void reset() {
        currentPosition = 0;
    }
}
```

- Colección: Interfaz que declara uno o varios métodos para obtener el comportamiento que trabaje mejor con la colección. Siempre debe devolver una interfaz iteradora que se le pueda enviar a una interfaz concreta.

social_networks/SocialNetwork.java: Defines common social network interface

```java
package refactoring_guru.iterator.example.social_networks;

import refactoring_guru.iterator.example.iterators.ProfileIterator;

public interface SocialNetwork {
    ProfileIterator createFriendsIterator(String profileEmail);

    ProfileIterator createCoworkersIterator(String profileEmail);
}
```

- Colección concreta: Devuelven nuevas instancias de una clase iteradora particular, está se emite a solicitud del cliente.

```java
public class Facebook implements SocialNetwork {
    private List<Profile> profiles;

    public Facebook(List<Profile> cache) {
        if (cache != null) {
            this.profiles = cache;
        } else {
            this.profiles = new ArrayList<>();
        }
    }

    public Profile requestProfileFromFacebook(String profileEmail) {
        // Here would be a POST request to one of the Facebook API endpoints.
        // Instead, we emulates long network connection, which you would expect
        // in the real life...
        simulateNetworkLatency();
        System.out.println("Facebook: Loading profile '" + profileEmail + "' over the ne

        // ...and return test data.
        return findProfile(profileEmail);
    }

    public List<String> requestProfileFriendsFromFacebook(String profileEmail, String co
        // Here would be a POST request to one of the Facebook API endpoints.
        // Instead, we emulates long network connection, which you would expect
        // in the real life...
        simulateNetworkLatency();
        System.out.println("Facebook: Loading '" + contactType + "' list of '" + profile

        // ...and return test data.
        Profile profile = findProfile(profileEmail);
        if (profile != null) {
            return profile.getContacts(contactType);
        }
        return null;
    }

    private Profile findProfile(String profileEmail) {
        for (Profile profile : profiles) {
            if (profile.getEmail().equals(profileEmail)) {
                return profile;
            }
        }
        return null;
    }

    private void simulateNetworkLatency() {
        try {
            Thread.sleep(2500);
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }
```

```
    @Override
    public ProfileIterator createFriendsIterator(String profileEmail) {
        return new FacebookIterator(this, "friends", profileEmail);
    }

    @Override
    public ProfileIterator createCoworkersIterator(String profileEmail) {
        return new FacebookIterator(this, "coworkers", profileEmail);
    }

}
```

- Cliente: Es el actor que solicita un elemento de una colección mediante un iterador, esto técnicamente se realiza interactuando con las interfaces para no acoplarse con clases concretas.

  En casos muy específicos los clientes pueden crear iteradores cuando tienen comportamientos específicos.

```
public class SocialSpammer {
    public SocialNetwork network;
    public ProfileIterator iterator;

    public SocialSpammer(SocialNetwork network) {
        this.network = network;
    }

    public void sendSpamToFriends(String profileEmail, String message) {
        System.out.println("\nIterating over friends...\n");
        iterator = network.createFriendsIterator(profileEmail);
        while (iterator.hasNext()) {
            Profile profile = iterator.getNext();
            sendMessage(profile.getEmail(), message);
        }
    }

    public void sendSpamToCoworkers(String profileEmail, String message) {
        System.out.println("\nIterating over coworkers...\n");
        iterator = network.createCoworkersIterator(profileEmail);
        while (iterator.hasNext()) {
            Profile profile = iterator.getNext();
            sendMessage(profile.getEmail(), message);
        }
    }

    public void sendMessage(String email, String message) {
        System.out.println("Sent message to: '" + email + "'. Message body: '" + messag
    }
}
```

Se adiciona la clase main para efectos del ejemplo donde se puede observar que se invoca el :

```
public class Demo {
    public static Scanner scanner = new Scanner(System.in);

    public static void main(String[] args) {
        System.out.println("Please specify social network to target spam tool (default:F
        System.out.println("1. Facebook");
        System.out.println("2. LinkedIn");
        String choice = scanner.nextLine();

        SocialNetwork network;
        if (choice.equals("2")) {
            network = new LinkedIn(createTestProfiles());
        }
        else {
            network = new Facebook(createTestProfiles());
        }

        SocialSpammer spammer = new SocialSpammer(network);
        spammer.sendSpamToFriends("anna.smith@bing.com",
                "Hey! This is Anna's friend Josh. Can you do me a favor and like this po
        spammer.sendSpamToCoworkers("anna.smith@bing.com",
                "Hey! This is Anna's boss Jason. Anna told me you would be interested in
    }

    public static List<Profile> createTestProfiles() {
        List<Profile> data = new ArrayList<Profile>();
        data.add(new Profile("anna.smith@bing.com", "Anna Smith", "friends:mad_max@ya.co
        data.add(new Profile("mad_max@ya.com", "Maximilian", "friends:anna.smith@bing.co
        data.add(new Profile("bill@microsoft.eu", "Billie", "coworkers:avanger@ukr.net")
        data.add(new Profile("avanger@ukr.net", "John Day", "coworkers:bill@microsoft.eu
        data.add(new Profile("sam@amazon.com", "Sam Kitting", "coworkers:anna.smith@bing
        data.add(new Profile("catwoman@yahoo.com", "Liza", "friends:anna.smith@bing.com"
        return data;
    }
}
```

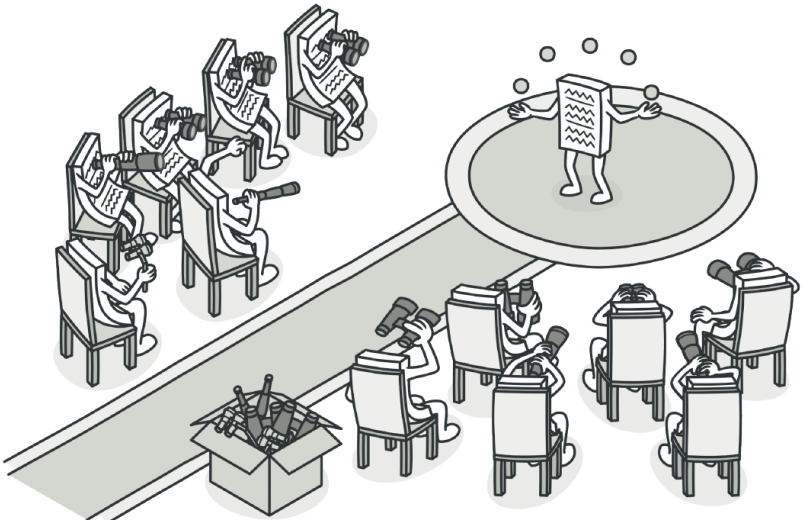
Una interfaz puede declarar varios iteradores los cuales cumplen diferentes comportamientos.

Con la implementación a una sola interfaz el código se hace compatible con cualquier tipo de colección o algoritmo de recorrido (esto depende de que exista el iterador adecuado para el tipo de búsqueda).

Cuando se desarrolla un nuevo comportamiento únicamente se crea un nuevo iterador lo que conserva la colección y el cliente.

Este patrón se utiliza para reducir la duplicación en el código de recorrido a lo largo de la aplicación.

- Mediator: Permite reducir las dependencias caóticas entre objetos. El patrón restringe las comunicaciones directas entre los objetos, forzándolos a colaborar únicamente a través de un objeto mediador.
- Memento: Permite guardar y restaurar el estado previo de un objeto sin revelar los detalles de su implementación.
- Observer: Permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando.

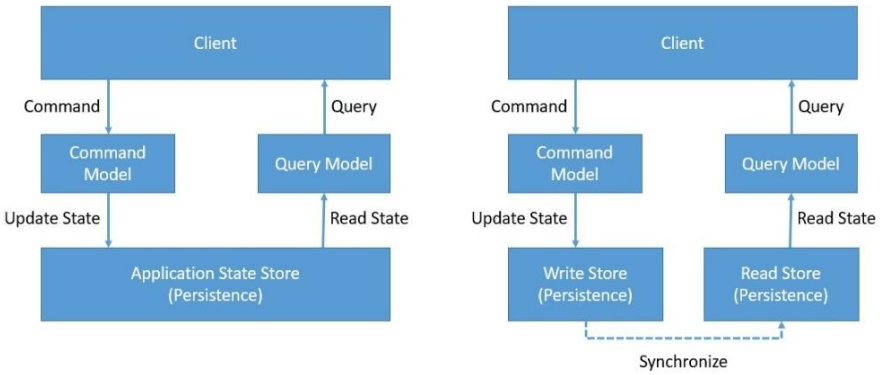


- State: Permite a un objeto alterar su comportamiento cuando su estado interno cambia. Parece como si el objeto cambiara su clase.
- Strategy: Permite definir una familia de algoritmos, colocar cado uno de ellos en una clase separada y hacer sus objetos intercambiables.
- Template Method: Define el esqueleto de un algoritmo en la superclase pero permite que las subclases sobrescriban pasos del algoritmo sin cambiar su estructura.
- Visitor: Permite separar algoritmos de los objetos sobre los que operan.

2. Patrones de Diseño de Software en Microservicios

a. Patrón Circuit Breaker: Patrón que implementa un control de servicios que no se encuentran disponibles y permite controlar los cortes de servicios de manera que se pueda controlar las funcionalidades para evitar que los procesos fallen de manera total.
b. Patrón Retry: **Patrón** que administra el reintento de operaciones para casos de errores transitorios (errores que se resuelven solos pasados un tiempo como puede ser un tema de accesos porque los recursos están ocupados).
c. Patrón Gateway Aggregation: Patrón para consolidar múltiples solicitudes a servicios para trabajarlos de manera simple.
d. Patrón Backends for Frontends: Patrón que implica que por cada front end se use un back end que lo respalde evitando un back end centralizado para no complicar las dependencias con el requerimiento de cada front end.
e. Patrón CQRS: Comand Query Request Segregation es un patrón de desarrollo de sistemas que consiste en un desarrollo dividido en dos partes Query (consultas) y Comands (Ordenes).

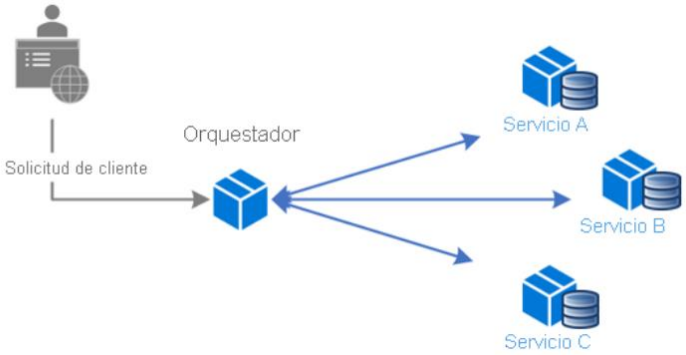
   Al realizar las consultas a parte de las ordenes y manejaras como si fueran dos repositorios independientes estos deben realizar una acción sincronía cada que se realicen modificaciones en la data para que las consultas sigan trayendo data vigente.



f. Patrón SAGA

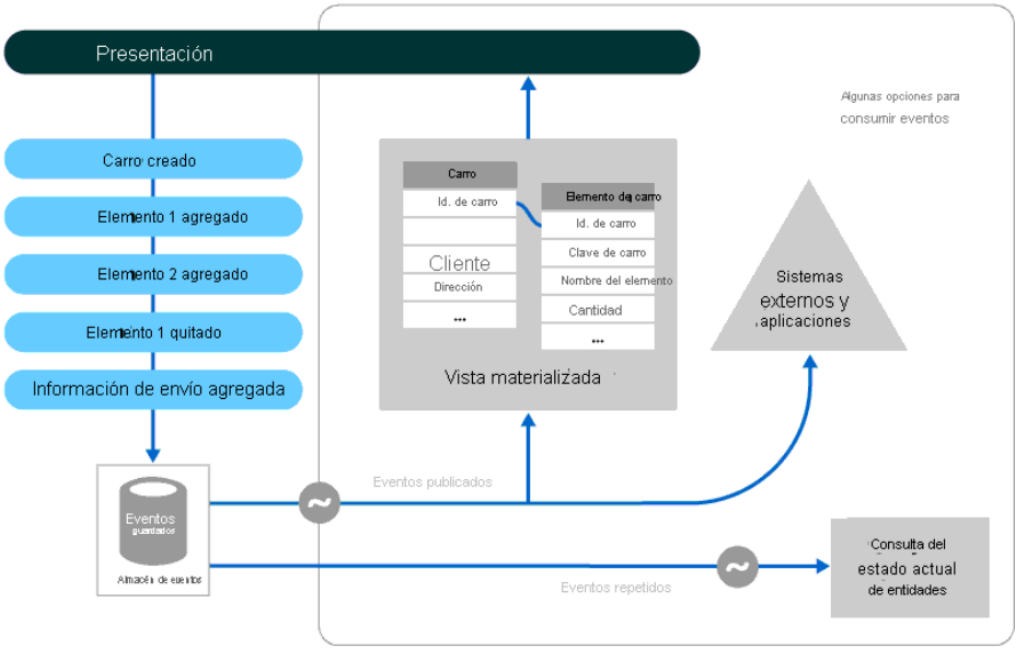
Patrón de administración de transacciones mediante transacciones locales. La transacción local es un esfuerzo de trabajo realizado por un participante de la saga, cada transacción local actualiza la base de datos y dispara un evento para la siguiente transacción.

Enfoques de patrón saga coreografía y orquestación, la única diferencia es el uso de un orquestador para administrar los eventos en la coreografía no hay orquestador y los eventos se disparan directamente contra los servicios lo que hace útil esta forma para sistemas pequeños sin embargo no presenta un control de compensación correcto pues no se controlan los envíos de eventos; es por eso que cuando el proyecto es grande se usa la orquestación.



g. Patrón event sourcing: Patrón que genera un almacén de eventos mediante objetos atómicos y de bajo nivel, esto facilita la flexibilidad y extensibilidad para el sistema. Este patrón facilita la integración con otros servicios.

Este patrón suele trabajar con el patrón CQRS.



3. **MongoDB**
4. **Topicos Kafka**

   Kafka es una plataforma de distribución de flujos para el manejo de eventos de desarrollo en tiempo real (entre otras cosas).

   Es una aplicación distribuida por lo que puede ejecutarse a través de múltiples servidores o centro de data. Permite un nivel de clonación y particionamiento elevado para un consumo masivo simultaneo sin perder desempeño.

   Tiene como características:
   - Produce consume.
   - Fast
   - High acuracy
   - In order
   - Fault tolerant

   Sus fortalezas son:

   - Desacoplamiento de dependencia de sistemas.
   - Se maneja por flujos de eventos.
   - El emisor solo genera un evento y no le importa quién lo esté escuchando.
   - Si un receptor desea conocer cuando iniciar un evento debe suscribirse. Reciben la información y se disparan los eventos.
   - Las suscripciones manejan la información mediante mensajes (Kafka en sí es un servicio de entrega de mensajes que “disparan” eventos).
   - Permite rastrear la ruta de los mensajes enviados para asegurar el flujo y la llegada al destino.
   - Data gathering → proceso de coleccionar data para desarrollar métricas relacionadas al transito de datos en el SW.

   Núcleos de Kafka:

- Producer API: Generadores de data.
- Topic: Lista ordenada de eventos.
- Consumer API: Suscrito a uno o varios tópicos para consumir su data y disparar eventos en tiempo real.
- Streams API: Nivela la producción y el consumo de API mediante el consumo de uno o varios tópicos pasando por un proceso de procesamiento y transformación de la data generando nuevos flujos para tópicos nuevos o para los ya existentes.
- Conector API: Genera conectores que serán reutilizados por varios desarrolladores evitando la duplicidad de generación de tópicos para desarrollo de eventos (por ejemplo el uso de una base de datos Mongo DB).