

**Taller**

**Patrón Publisher-Subscriber**

**Tutor**

Ing. Eduardo Mauricio Campaña Ortega

**Integrantes**

Luis Corales

Diego Hiriart

Christian Samaniego

**Fecha**

19/07/2023

## Tabla de contenido

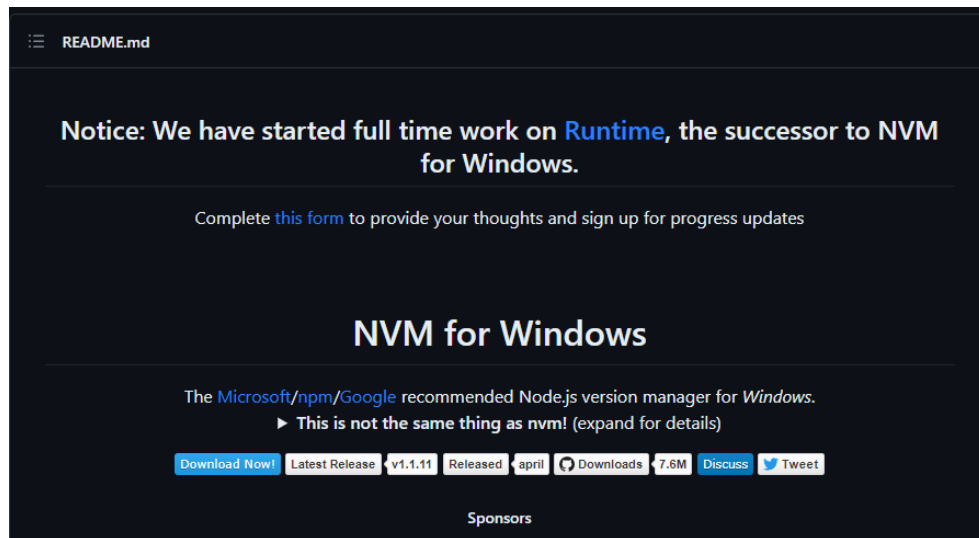
Pre-requisitos .....	1
Instalación Node.js .....	1
Instalación Docker .....	4
Desarrollo .....	7
Configuración Inicial .....	7
Interfaces .....	12
Providers .....	13
Redis .....	13
Zeromq .....	14
Kafka .....	15
Ejecución .....	19
Conclusiones .....	21
Recomendaciones .....	21

## Pre-requisitos

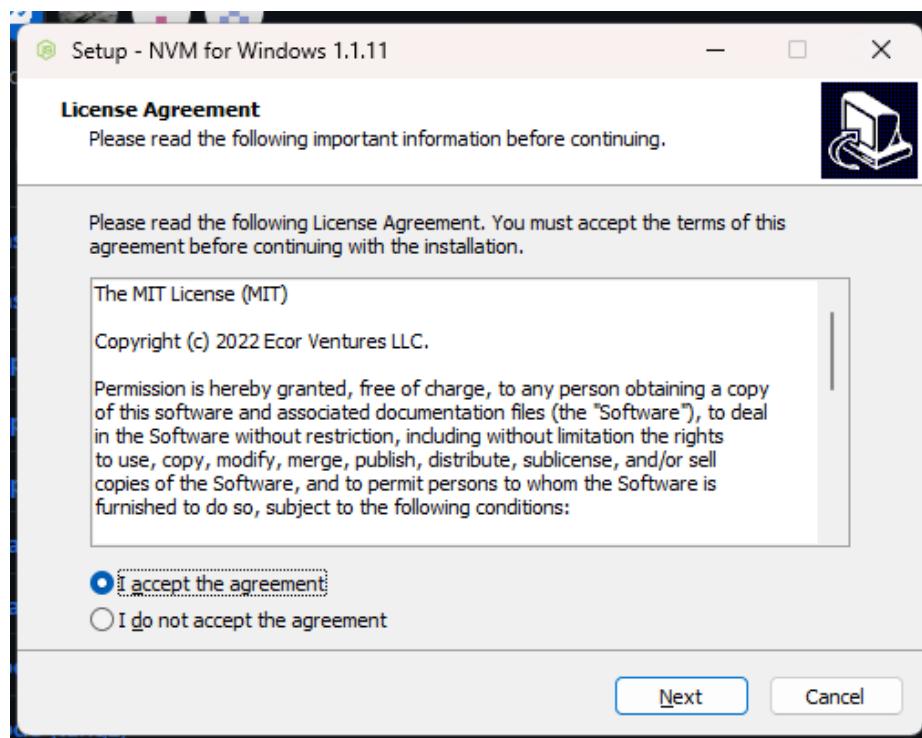
### Instalación Node.js

Para la implementación de esta solución es necesario contar con una instalación de Node.js, para lo cual se utilizará la herramienta nvm la cual permite instalar y gestionar distintas versiones de node de forma fácil.

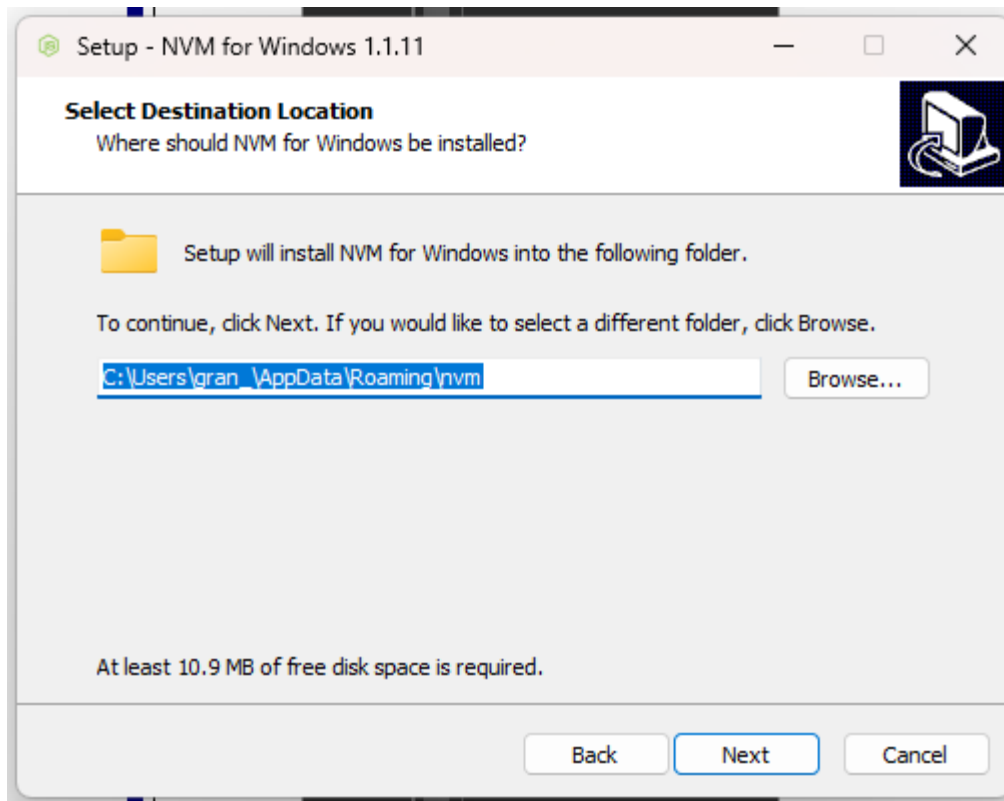
En el caso de utilizar Windows, se utilizará la herramienta nvm-windows. Para instalar la herramienta es necesario visitar el repositorio en github: <https://github.com/coreybutler/nvm-windows> y dar click en "Download Now!"



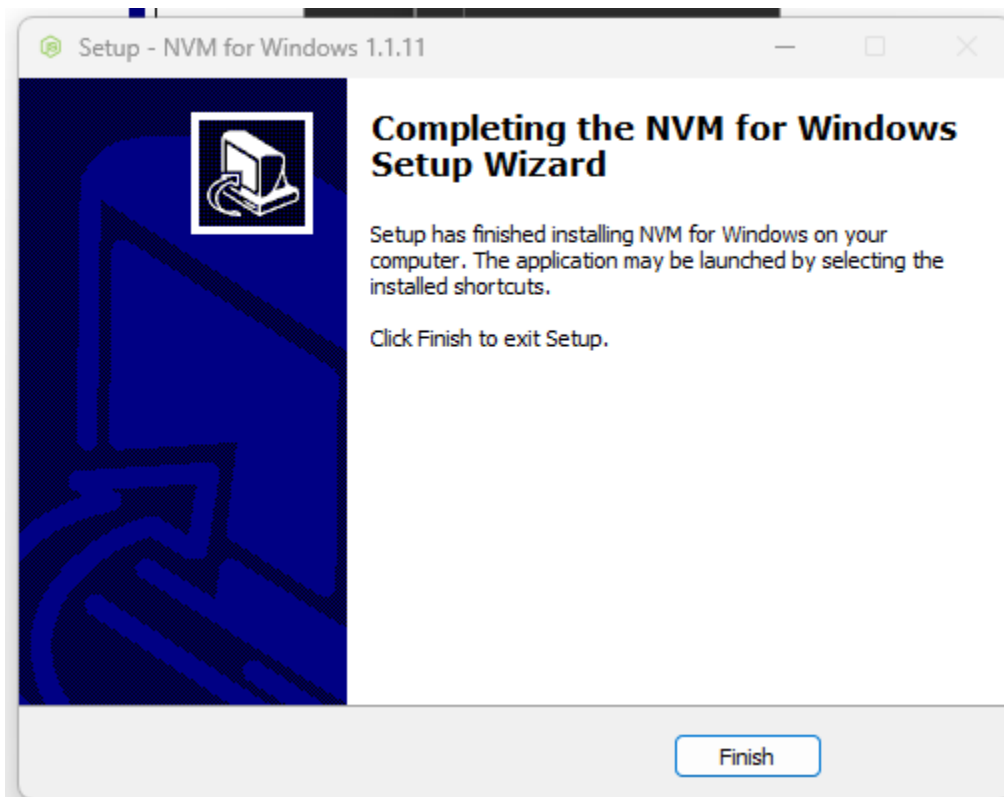
Descargamos el archivo nvm-setup.exe y lo ejecutamos. A continuación se mostrará un wizard de instalación. Aceptamos y damos click en Next.



Seleccionamos el directorio de instalación y damos click en Next:

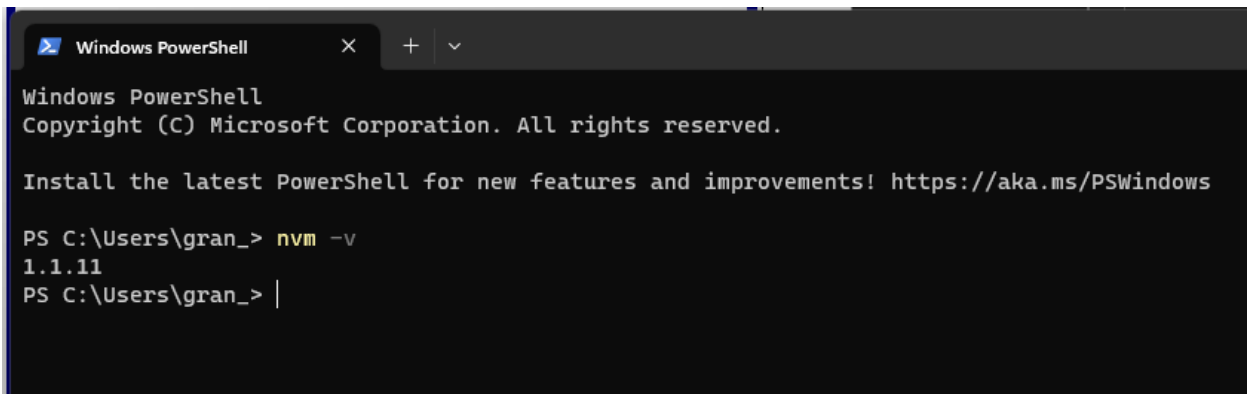


Tardará unos segundos en instalar y se mostrará la siguiente ventana de confirmación:



Podemos confirmar la instalación de nvm corriendo el siguiente comando en Powershell:

**nvm -v**



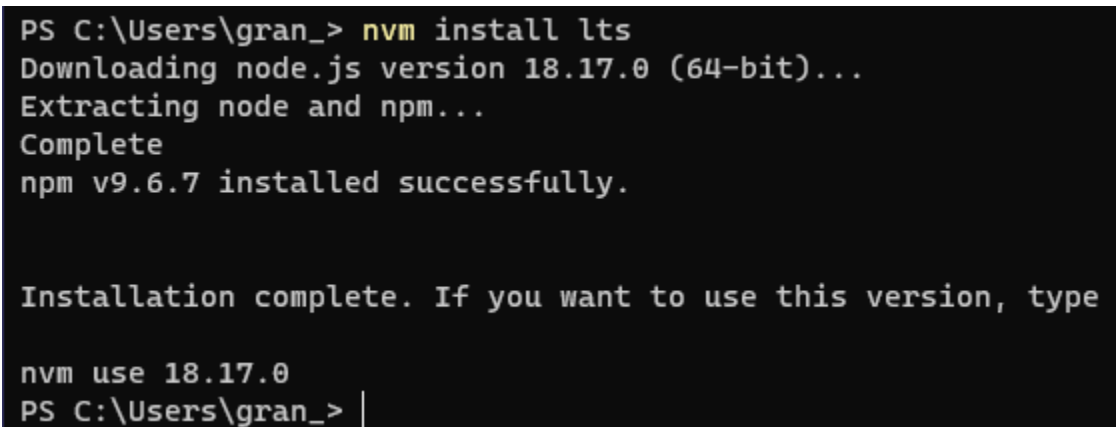
```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\gran_> nvm -v
1.1.11
PS C:\Users\gran_> |
```

Ahora procedemos a instalar una versión de Node. En este caso instalaremos la versión estable más reciente corriendo el comando:

**nvm install lts**



```
PS C:\Users\gran_> nvm install lts
Downloading node.js version 18.17.0 (64-bit)...
Extracting node and npm...
Complete
npm v9.6.7 installed successfully.

Installation complete. If you want to use this version, type

nvm use 18.17.0
PS C:\Users\gran_> |
```

Finalmente corremos el comando **nvm use 18.17.0** para utilizar esta versión.

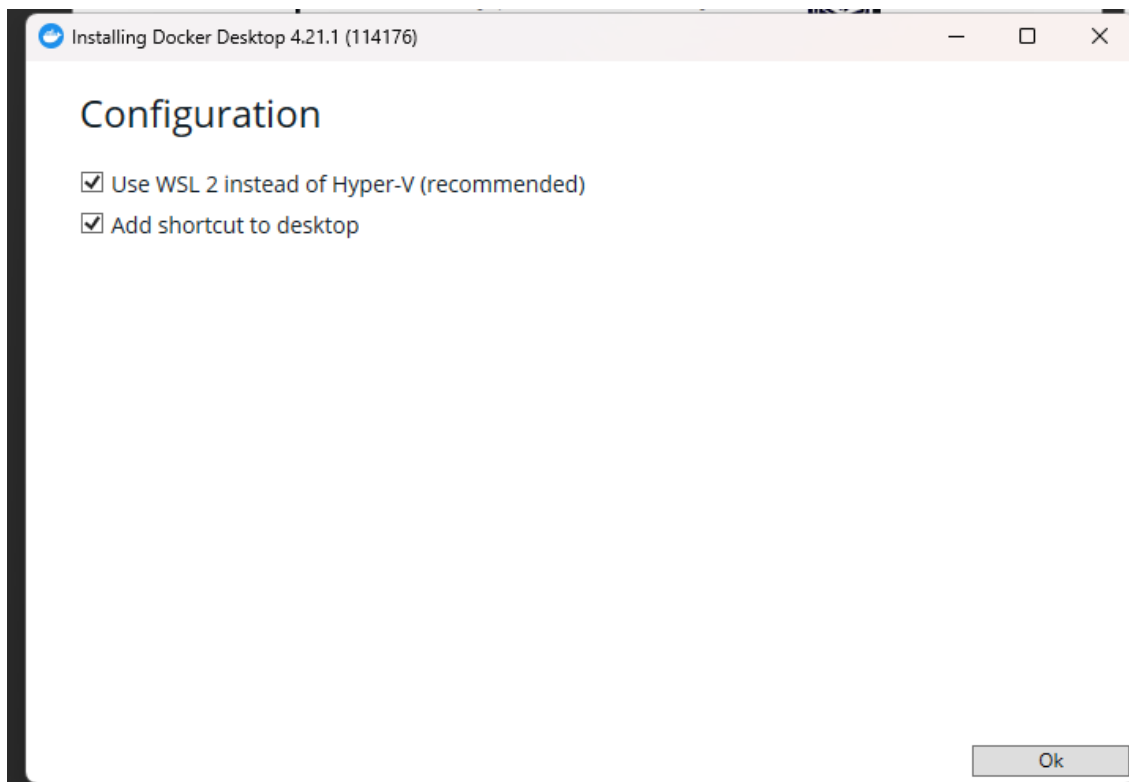
Al correr **node -v** podemos confirmar que se utiliza esa versión.

## Instalación Docker

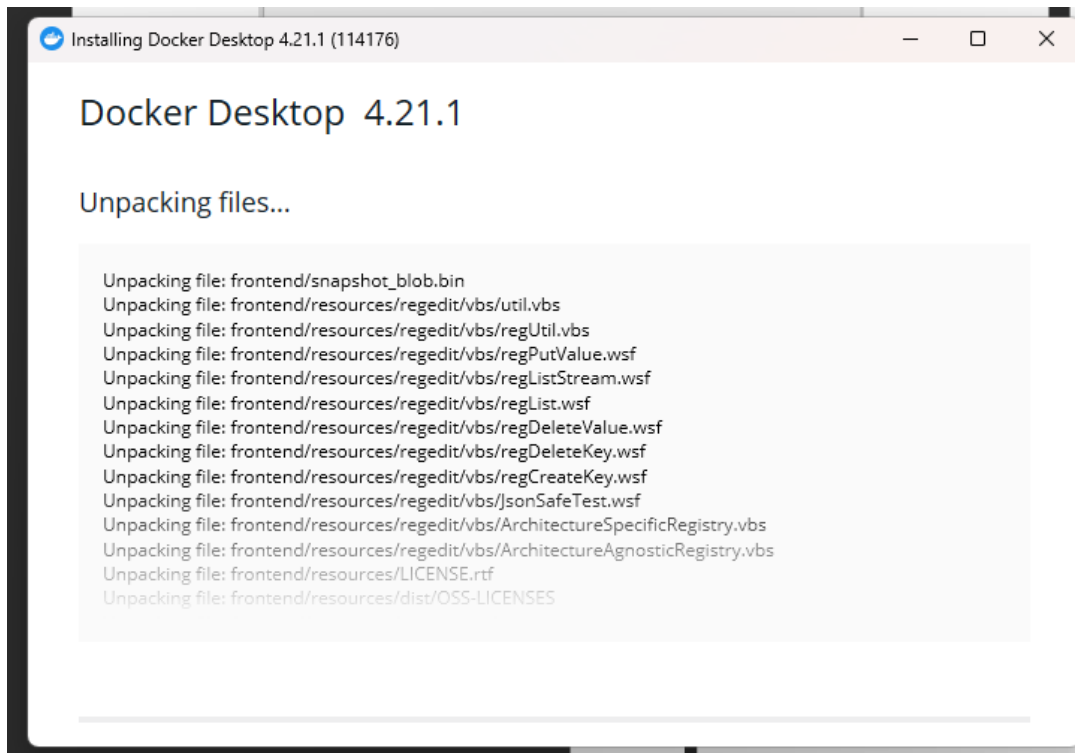
A continuación, continuaremos con la instalación de Docker. Para esto visitaremos la página de descarga oficial <https://docs.docker.com/desktop/install/windows-install/> y damos click en el botón para descargar el instalador de Docker Desktop:



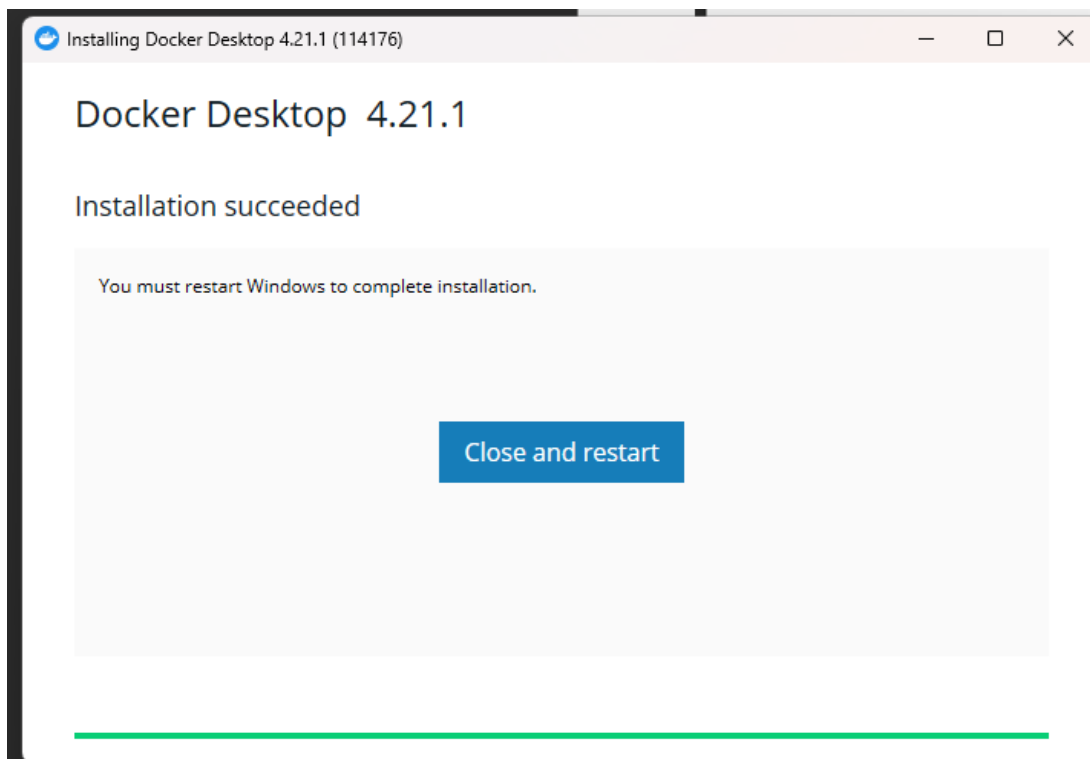
Es recomendable utilizar WSL 2 en lugar de Hyper-V. Damos click en Ok.

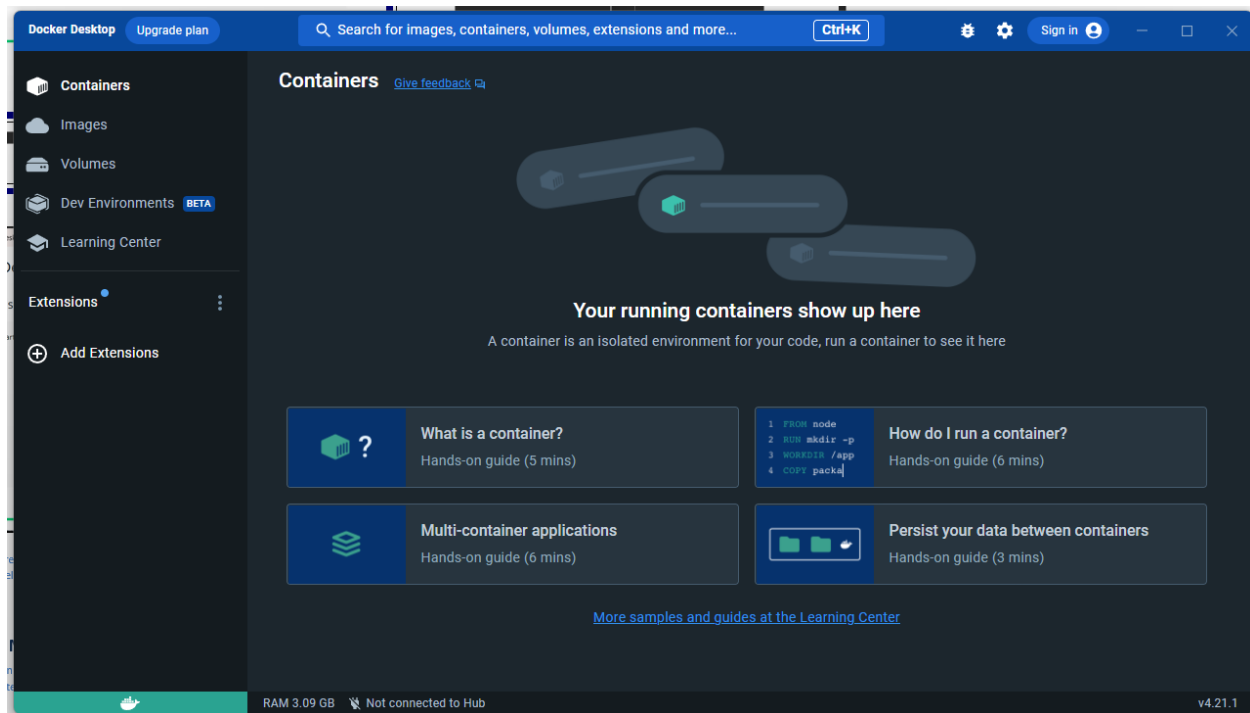


Comenzará la instalación de Docker Desktop



Una vez finalizado se mostrará la siguiente pantalla de confirmación solicitando reiniciar Windows para completar la instalación:





## Desarrollo

### Configuración Inicial

El código para este proyecto, en caso de necesitarse como guía, se puede encontrar en el siguiente repositorio:

- [Talleres-Integracion-HCS/publisher-subscriber Corales Hiriart Samaniego at main · Diego-Hiriart/Talleres-Integracion-HCS · GitHub](https://github.com/Diego-Hiriart/Talleres-Integracion-HCS)

Para la creación inicial del proyecto se creará un proyecto mediante el comando

**npm init -y**

Este comando generará un archivo **package.json** en el cual se especificarán las dependencias, sus correspondientes versiones y scripts:

```
"scripts": {
  "build": "tsc --build",
  "start:dev": "nodemon"
},
"devDependencies": {
  "@types/node": "^20.4.2",
  "nodemon": "^3.0.1",
  "ts-node": "^10.9.1",
  "typescript": "^5.1.6"
},
"dependencies": {
  "ioredis": "^4.27.6",
```



```

    "kafkajs": "^2.2.4",
    "zeromq": "^5.3.1"
  }

```

Para instalar las dependencias se debe correr el comando **npm install**

Se utilizará TypeScript el cual es un superset de JavaScript que agrega tipado estático, interfaces, clases y genéricos, entre otras características. Permite detectar errores en tiempo de compilación y mejora la escalabilidad y legibilidad del código. El siguiente archivo contempla una configuración básica de typescript.

#### tsconfig.json

```

{
  "compilerOptions": {
    "module": "commonjs",
    "declaration": true,
    "removeComments": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "allowSyntheticDefaultImports": true,
    "target": "es2017",
    "sourceMap": true,
    "outDir": "./dist",
    "baseUrl": "./",
    "incremental": true
  }
}

```

#### tsconfig.build.json

```

{
  "extends": "./tsconfig.json",
  "exclude": ["node_modules", "test", "dist", "**/*spec.ts"],
  "include": ["src/**/*.ts"]
}

```

También se utilizará **nodemon** es una herramienta de desarrollo para aplicaciones de Node.js que facilita el proceso de desarrollo al monitorizar cambios en los archivos de tu proyecto y reiniciar automáticamente el servidor cada vez que detecta algún cambio.

#### nodemon.json

```

{
  "watch": ["src"],
  "ext": "ts,json",

```

```

    "ignore": [".git", "node_modules/"],
    "exec": "ts-node ./src/app.ts"
  }
}

```

Se hará uso del siguiente archivo de configuración de Docker:

### Dockerfile

```

FROM node:14-stretch

ENV workdir /var/prod

WORKDIR ${workdir}

COPY *.json .
COPY src ./src

RUN npm install
RUN npm run build

EXPOSE 8200
CMD ["npm", "run", "start:dev"]

```

Finalmente, crearemos un archivo **docker-compose** que incluirá los servicios necesarios para la ejecución de Kafka: Zookeeper y bróker. Adicionalmente incluye la imagen de redis y por último la imagen de zeromq-server:

```

//docker-compose.yml
version: '3.5'

services:
  zookeeper:
    image: confluentinc/cp-zookeeper:6.0.1
    hostname: zookeeper
    container_name: zookeeper
    ports:
      - "2181:2181"
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000
    networks:
      - example-pub-sub-net

  broker:
    image: confluentinc/cp-server:6.0.1
    hostname: broker

```

```

    container_name: broker
    depends_on:
      - zookeeper
    ports:
      - "9092:9092"
      - "9101:9101"
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: 'zookeeper:2181'
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_ADVERTISED_LISTENERS:
PLAINTEXT://broker:29092,PLAINTEXT_HOST://localhost:9092
      KAFKA_METRIC_REPORTERS:
io.confluent.metrics.reporter.ConfluentMetricsReporter
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_GROUP_INITIAL_REBALANCE_DELAY_MS: 0
      KAFKA_CONFLUENT_LICENSE_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_CONFLUENT_BALANCER_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
      KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
      KAFKA_JMX_PORT: 9101
      KAFKA_JMX_HOSTNAME: localhost
      KAFKA_CONFLUENT_SCHEMA_REGISTRY_URL: http://schema-registry:8081
      CONFLUENT_METRICS_REPORTER_BOOTSTRAP_SERVERS: broker:29092
      CONFLUENT_METRICS_REPORTER_TOPIC_REPLICAS: 1
      CONFLUENT_METRICS_ENABLE: 'true'
      CONFLUENT_SUPPORT_CUSTOMER_ID: 'anonymous'
    networks:
      - example-pub-sub-net

redis:
  container_name: realtime-broker-redis
  image: redis:alpine
  networks:
    - example-pub-sub-net
  ports:
    - "6379:6379"

zeromq-server:
  build:
    context: ./zeromq-server
    dockerfile: ./Dockerfile
  container_name: zeromq-server
  environment:

```

```

    - ZMQ_BIND_ADDRESS=tcp://*:3000
networks:
  - example-pub-sub-net

example-pub-sub:
  container_name: example-pub-sub
  build: .
  environment:
    - NODE_ENV=develop
  volumes:
    - ./src:/var/prod/src
  networks:
    - example-pub-sub-net
  depends_on:
    - broker
    - zeromq-server
    - redis

networks:
  example-pub-sub-net:
    name: example-pub-sub-net
    driver: bridge

```

Finalmente, crearemos una carpeta llamada zeromq-server la cual funcionará similarmente a las imágenes de karfa y redis. Sirve para conectarse a un servidor específico y enviar un mensaje de prueba al topic.

```

//zeromq-server\app.js
const zmq = require('zeromq')
const socket = zmq.socket('pub')
const address = process.env.ZMQ_BIND_ADDRESS || `tcp://*:3000`

socket.bindSync(address)

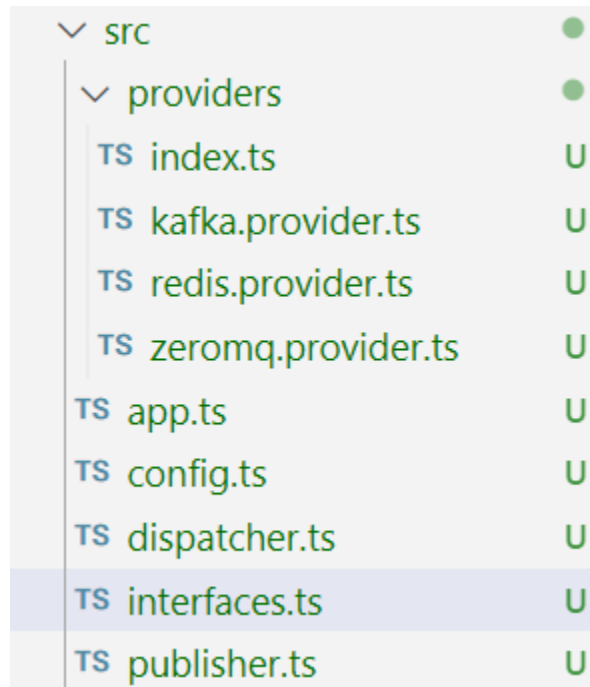
const sendMessage = () => {
  const message = { message: 'Hola !' }
  socket.send(['test', JSON.stringify(message)], (err) => {
    console.log(err)
  })
}

setInterval(sendMessage, 5000)

```

## Interfaces

Se manejará la siguiente estructura inicial de archivos. La carpeta src/ incluirá nuestro código fuente, interfaces y archivo de entrada y nuestros componentes de providers, publishers y dispatcher.



Comenzaremos por definir las interfaces que serán implementadas por nuestras clases en el archivo interfaces.ts

**MessageFromEvent:** Se encarga de definir el formato estándar de como se van a enviar los mensajes al dispatcher.

```
export interface MessageFromEvent {
  data: Record<string, unknown>
  topic: string
  provider: string
}
```

**BaseProvider:** Es la interfaz que implementarán todas las clases. Incluye el método para leer los mensajes de los componentes topic.

```
export interface BaseProvider {
  readonly providerName: string
  subscribe(topics: string[]): Promise<void>
  readMessagesFromTopics(callback: (data: MessageFromEvent) => void)
}
```

Adicionalmente se definen las tres siguientes interfaces que definen las propiedades que necesita cada cliente para conectarse correctamente:

```
export interface KafkaConnectionOptions {
  clientId: string
  brokers: string[]
  groupId: string
}
```

```
export interface RedisConnectionOptions {
  host: string
  port: number
  db: number
  family?: number // 4 (IPv4) or 6 (IPv6)
  password?: string
}
```

```
export interface ZeromqConnectionOptions {
  host: string
  port: number
}
```

## Providers

### Redis

Crearemos la clase Redis provider, la cual implementa los métodos y propiedades de la interfaz **BaseProvider**. Es importante mencionar que al leer los mensajes del topic, los mensajes se envían como string y mediante el método `JSON.parse` se lo parsea al objeto correspondiente. El mismo proceso se realizará para los providers de **Kafka** y **Zeromq**

```
//src/providers/redis.provider.ts
import Redis from 'ioredis'
import { RedisConnectionOptions, BaseProvider, MessageFromEvent } from
'../interfaces'

export class RedisProvider implements BaseProvider {
  readonly providerName = 'redis'
  private provider: Redis.Redis
```

```

constructor(opts: RedisConnectionOptions) {
    this.provider = new Redis({
        port: opts.port,
        host: opts.host,
        family: opts.family,
        password: opts.password,
        db: opts.db,
    })
}

async subscribe(topics: string[]): Promise<void> {
    await this.provider.subscribe(...topics)
}

async readMessagesFromTopics(callback: (data: MessageFromEvent) => void):
Promise<void> {
    this.provider.on('message', async (topic: string, message: string) => {
        const data = JSON.parse(message) as unknown as Record<string, unknown>
        callback({ data, provider: this.providerName, topic })
    })
}

async testPublisher(topic: string, message: string): Promise<void> {
    this.provider.publish(topic, JSON.stringify({ message }))
}
}

```

## Zeromq

Algo importante en zeromq es que se debe especificar a la función socket() si el componente es un subscriber mediante el string 'sub' o Publisher mediante el string 'pub'. Una diferencia con Redis es que los topic se pasan separados por espacios.

```

// src/providers/zeromq.provider.ts
import * as zmq from 'zeromq'
import { ZeromqConnectionOptions, BaseProvider, MessageFromEvent } from
'../interfaces'

export class ZeromqProvider implements BaseProvider {
    readonly providerName = 'zeromq'
    private provider: zmq.Socket

    constructor(opts: ZeromqConnectionOptions) {
        this.provider = zmq.socket('sub')
        this.provider.connect(`${opts.host}:${opts.port}`)
    }
}

```

```

    }

    async subscribe(topics: string[]): Promise<void> {
        this.provider.subscribe(topics.join(' '))
    }

    async readMessagesFromTopics(callback: (data: MessageFromEvent) => void) {
        this.provider.on('message', async (topic: Buffer, message: Buffer) => {
            const data = JSON.parse(message.toString()) as unknown as Record<string,
unknown>
            callback({
                data,
                provider: this.providerName,
                topic: topic.toString(),
            })
        })
    }
}

```

## Kafka

A diferencia de Redis y Zeromq, para la implementación del provider en Kafka es necesario que el topic exista previamente antes de leerlo. Adicionalmente al igual que el provider de Redis, se implementa una función **testPublisher** para publicar los mensajes.

```

// src/providers/kafka.provider.ts
import { Admin, Consumer, EachMessagePayload, Kafka, Producer } from 'kafkajs'
import { KafkaConnectionOptions, BaseProvider, MessageFromEvent } from
'../interfaces'

export class KafkaProvider implements BaseProvider {
    readonly providerName = 'kafka'

    private provider: Kafka
    private consumer: Consumer
    private producer: Producer
    private admin: Admin

    constructor(opts: KafkaConnectionOptions) {
        this.provider = new Kafka({
            clientId: opts.clientId,
            brokers: opts.brokers,
        })

        this.consumer = this.provider.consumer({ groupId: opts.groupId })
        this.producer = this.provider.producer()
    }
}

```



```

    this.admin = this.provider.admin()

    this.createTopic().then()
}

async subscribe(topics: string[]): Promise<void> {
    await Promise.all(
        topics.map((topic: string) => this.consumer.subscribe({ topic,
fromBeginning: true })))
    )
}

async readMessagesFromTopics(callback: (data: MessageFromEvent) => void) {
    await this.consumer.run({
        eachMessage: async ({ topic, message }: EachMessagePayload) => {
            const data = JSON.parse(message.value.toString()) as unknown as
Record<string, unknown>
            callback({ data, provider: this.providerName, topic })
        },
    })
}

async createTopic() {
    try {
        await this.admin.connect()
        await this.admin.createTopics({ waitForLeaders: true, topics: [{ topic:
'test' }] })
    } catch (err) {
        console.log(err)
    }
}

async testPublisher(topic: string, message: string): Promise<void> {
    try {
        await this.producer.connect()
        await this.producer.send({
            topic: topic,
            messages: [{ value: JSON.stringify({ message }) }]
        })
    } catch (err) {
        console.log(err, 'error')
    }
}
}

```

Creemos un archivo **Index.ts** el cual es el encargado de exportar las tres clases creadas anteriormente:

```
// src/providers/index.ts
export { KafkaProvider } from './kafka.provider'
export { RedisProvider } from './redis.provider'
export { ZeromqProvider } from './zeromq.provider'
```

Adicionalmente, se deben definir configuraciones de conexión para cada cliente:

```
// src/config.ts
export const KAFKA_OPTS = {
  clientId: "exmaple-app",
  brokers: ["broker:29092"],
  groupId: "example",
};
export const REDIS_OPTS = {
  host: "redis",
  port: 6379,
  db: 0,
};
export const ZEROMQ_OPTS = {
  host: "tcp://zeromq-server",
  port: 3000,
};
```

También vamos a definir una función **dispatcher** la cual va a recibir todos los mensajes desde distintos providers que únicamente imprimirá los mensajes que reciba.

```
// src/dispatcher.ts
import { MessageFromEvent } from './interfaces';

export const dispatcher = (message: MessageFromEvent) => {
  console.log(message)
}
```

Ahora crearemos nuestro archivo main, en el cual instanciaremos cada uno de nuestros providers y emitiremos un evento de Node que es el encargado de enviar la información a un lugar centralizado (ie. Topic). Finalmente iteraremos a través de los providers y llamaremos los métodos para suscribirnos y leer los mensajes.

```
// src/app.ts
import * as events from 'events'
```

```

import { dispatcher } from './dispatcher'
import { KafkaProvider, ZeromqProvider, RedisProvider } from './providers'
import { REDIS_OPTS, KAFKA_OPTS, ZEROMQ_OPTS } from './config'
import { MessageFromEvent } from './interfaces'

const eventName = 'main'
const topics = ['test']
const event = new events.EventEmitter()
const provider = [
  new RedisProvider(REDIS_OPTS),
  new KafkaProvider(KAFKA_OPTS),
  new ZeromqProvider(ZEROMQ_OPTS),
]

event.on(eventName, dispatcher);
const sendMessageToFromEvent = (message: MessageFromEvent) => {
  event.emit(eventName, message)
}
const main = async () => {
  await Promise.all(
    provider.map(async (p) => {
      await p.subscribe(topics)
      await p.readMessagesFromTopics(sendMessageToFromEvent)
    })
  )
}

main()

```

Finalmente crearemos la función para publicar los mensajes en Redis y Zeromq. Simplemente instancia objetos de estas clases y llama al método para publicar mensajes cada cierto intervalo de tiempo:

```

// src/publisher.ts
import { KafkaProvider, RedisProvider } from './providers'
import { KAFKA_OPTS, REDIS_OPTS } from './config'

const kafka = new KafkaProvider(KAFKA_OPTS)
const redis = new RedisProvider(REDIS_OPTS)

export const msg = () => {
  setInterval(async () => {

```

```

    await kafka.testPublisher('test', 'hola from kafka')
    await redis.testPublisher('test', 'hola from redis')
  }, 5000)
}

```

## Ejecución

En primer lugar, debemos inicializar los servicios zookeeper y bróker que son necesarios para la ejecución de Kafka. Ejecutamos el siguiente comando:

**docker-compose up -d zookeeper broker**

y al correr **docker ps** se puede comprobar que los contenedores están corriendo:

```

PS C:\Users\gran_\OneDrive\Escritorio\publisher-subscriber> docker-compose up -d zookeeper broker
[+] Running 17/2
✓ broker 12 layers [████████████████████] 0B/0B Pulled 2
✓ zookeeper 3 layers [████████] 0B/0B Pulled 2
[+] Running 3/3
✓ Network example-pub-sub-net Created
✓ Container zookeeper Started
✓ Container broker Started
PS C:\Users\gran_\OneDrive\Escritorio\publisher-subscriber> docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED        STATUS        PORTS                                                                 NAMES
5788a4686c3a   confluentinc/cp-server:6.0.1       "/etc/confluent/dock..." 15 seconds ago Up 13 seconds 0.0.0.0:9092->9092/tcp, 0.0.0.0:9101->9101/tcp broker
20dfe1be6af1   confluentinc/cp-zookeeper:6.0.1    "/etc/confluent/dock..." 16 seconds ago Up 14 seconds 2888/tcp, 0.0.0.0:2181->2181/tcp, 3888/tcp zookeeper

```

Una vez levantados estos dos servicios podemos proceder a levantar los demás servicios con el siguiente comando:

**docker-compose up -d**

```

[+] Running 5/5
✓ Container realtime-broker-redis Started 0.6s
✓ Container zeromq-server Started 0.6s
✓ Container zookeeper Running 0.0s
✓ Container broker Running 0.0s
✓ Container example-pub-sub Started 0.9s
PS C:\Users\gran_\OneDrive\Escritorio\publisher-subscriber> █

```

Para observar los logs del contener debemos correr el siguiente comando:

**docker logs -f example-pub-sub**

```

{"level":"ERROR","timestamp":"2023-07-20T08:34:35.794Z","logger":"kafkajs","message":"[Connect
ion] Connection error: getaddrinfo EAI_AGAIN broker","broker":"broker:29092","clientId":"examp
le-app","stack":"Error: getaddrinfo EAI_AGAIN broker\n    at GetAddrInfoReqWrap.onlookup [as o
ncomplete] (dns.js:71:26)"}
{ data: { message: 'Hola !' }, provider: 'zeromq', topic: 'test' }
{ data: { message: 'Hola !' }, provider: 'zeromq', topic: 'test' }
{ data: { message: 'Hola !' }, provider: 'zeromq', topic: 'test' }
{ data: { message: 'Hola !' }, provider: 'zeromq', topic: 'test' }
{ data: { message: 'Hola !' }, provider: 'zeromq', topic: 'test' }
{ data: { message: 'Hola !' }, provider: 'zeromq', topic: 'test' }
{ data: { message: 'Hola !' }, provider: 'zeromq', topic: 'test' }
{ data: { message: 'Hola !' }, provider: 'zeromq', topic: 'test' }
{ data: { message: 'Hola !' }, provider: 'zeromq', topic: 'test' }
{ data: { message: 'Hola !' }, provider: 'zeromq', topic: 'test' }

```

Se puede observar que se envían los mensajes de zeromq, sin embargo, no se observan mensajes de Kafka ni de redis. Para eso necesitamos ejecutar la función que definimos en **src/Publisher.ts**

Primero obtendremos acceso al Shell del contenedor corriendo con el siguiente comando desde una nueva terminal:

**Docker exec -it example-pub-sub bash**

Y procedemos a correr ts-node ejecutando el paquete directamente desde node\_modules. Debemos importar la función msg y llamarla:

```

PS C:\Users\gran_\OneDrive\Escritorio\publisher-subscriber> docker exec -it example-pub-sub bash
root@031fa2d40cbc:/var/prod# node_modules/ts-node/dist/bin.js
> import {msg} from './src/publisher'
undefined
> msg()
{"level":"WARN","timestamp":"2023-07-20T08:53:11.586Z","logger":"kafkajs","message":"KafkaJS v2.0.0 switc
hed default partitioner. To retain the same partitioning behavior as in previous versions, create the pro
ducer with the option \"createPartitioner: Partitioners.LegacyPartitioner\". See the migration guide at h
ttps://kafka.js.org/docs/migration-guide-v2.0.0#producer-new-default-partitioner for details. Silence thi
s warning by setting the environment variable \"KAFKAJS_NO_PARTITIONER_WARNING=1\""}
undefined
> {"level":"ERROR","timestamp":"2023-07-20T08:53:11.624Z","logger":"kafkajs","message":"[Connection] Resp
onse CreateTopics(key: 19, version: 3)","broker":"broker:29092","clientId":"example-app","error":"Topic c
reation errors","correlationId":2,"size":50}
> █

```

Y desde la otra terminal podemos observar que se envían mensajes de los tres servicios: Kafka, redis y zeromq demostrando la funcionalidad correcta del patrón Publisher-Subscriber:

```
data: { message: 'hola from redis' },
provider: 'redis',
topic: 'test'
}
{ data: { message: 'Hola !' }, provider: 'zeromq', topic: 'test' }
{
  data: { message: 'hola from kafka' },
  provider: 'kafka',
  topic: 'test'
}
{
  data: { message: 'hola from redis' },
  provider: 'redis',
  topic: 'test'
}
{ data: { message: 'Hola !' }, provider: 'zeromq', topic: 'test' }
{
  data: { message: 'hola from kafka' },
  provider: 'kafka',
  topic: 'test'
}
}
```

## Conclusiones

- El Patrón Publisher-Subscriber permite reducir el acoplamiento entre componentes ya que los publicadores y sus suscriptores no necesitan conocerse entre sí directamente, lo que facilita cambios y extensiones en el sistema.
- El patrón Publisher-Subscriber permite la comunicación asincrónica entre componentes, lo que puede mejorar el rendimiento y la capacidad de respuesta del sistema, especialmente en entornos en tiempo real.
- Al utilizar el patrón Publisher-Subscriber, es posible agregar o quitar suscriptores sin afectar el funcionamiento del sistema. Esto permite una mayor escalabilidad y flexibilidad para manejar múltiples eventos y escenarios.
- La utilización de Docker y Docker-compose para el levantamiento de servicios son una excelente opción para crear ambientes replicables en diferentes contextos.

## Recomendaciones

- Es importante verificar que las dependencias que se utilizan en un proyecto sean las apropiadas para garantizar el correcto funcionamiento del sistema. En este caso fue necesario realizar una revisión de las dependencias de desarrollo ya que no permitían la ejecución del servidor.
- Al usar herramientas de contenedores como Docker, se debe comprobar que los cambios realizados en el código se vean reflejados en el contenedor. Es importante forzar la recreación del contenedor para así evitar que el código fuente se mantenga en el cache.
- Antes de implementar el patrón Publisher-Subscriber, es importante identificar los eventos significativos que pueden ocurrir en el sistema y determinar qué componentes pueden ser notificados cuando esos eventos sucedan.