**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](#_Toc140718804)

[Instalación Node.js 1](#_Toc140718805)

[Instalación Docker 4](#_Toc140718806)

[Desarrollo 7](#_Toc140718807)

[Configuración Inicial 7](#_Toc140718808)

[Interfaces 12](#_Toc140718809)

[Providers 13](#_Toc140718810)

[Redis 13](#_Toc140718811)

[Zeromq 14](#_Toc140718812)

[Kafka 15](#_Toc140718813)

[Ejecución 19](#_Toc140718814)

[Conclusiones 21](#_Toc140718815)

[Recomendaciones 21](#_Toc140718816)

# Pre-requisitos

## Instalación Node.js

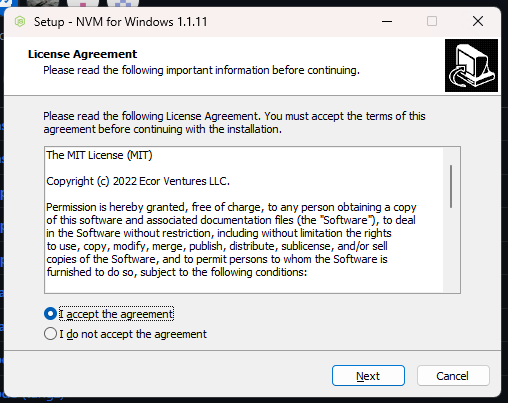
Para la implementación de esta solución es necesario contar con una instalación de Node.js, para lo cual se utilizará la herramienta nvm la cual permite instalar y gestionar distintas versiones de node de forma fácil.

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!”

A screenshot of a computer program

Description automatically generated

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

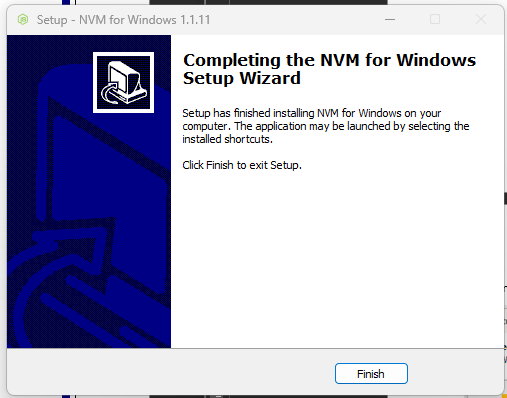


Seleccionamos el directorio de instalación y damos click en Next:

A screenshot of a computer

Description automatically generated

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**

**A screenshot of a computer

Description automatically generated**

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

**nvm install lts**

**A screenshot of a computer

Description automatically generated**

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

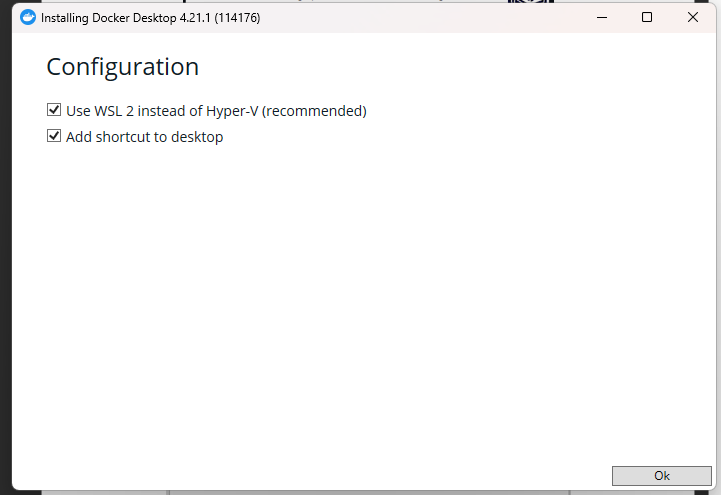
Al correr **node -v** podemos confirmar que se utiliza esa versión.

## 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.

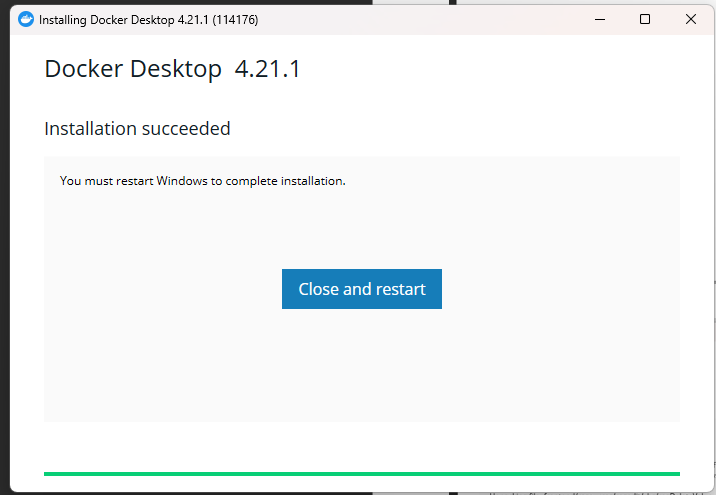


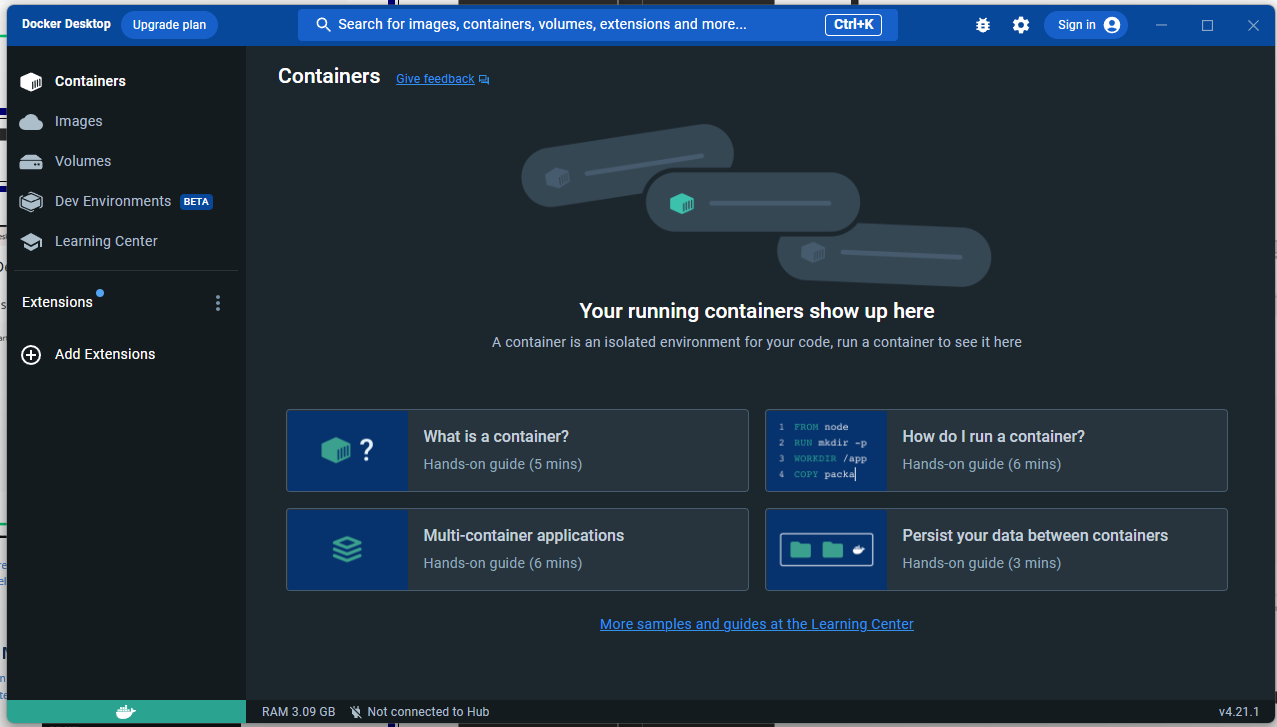
Comenzará la instalación de Docker Desktop

A screenshot of a computer

Description automatically generated

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





# Desarrollo

## Configuración Inicial

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.ts**

{

    "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 especifico 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 envian 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')

    }

  }

}

Creamos 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**

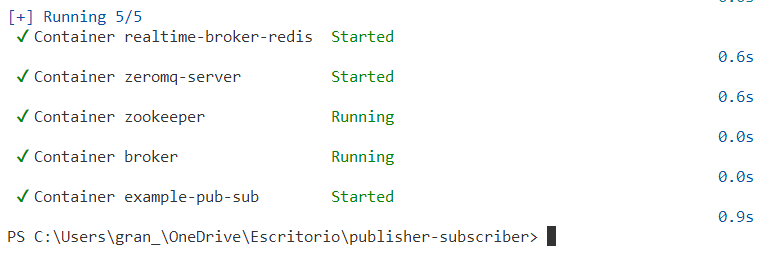
y al correr **docker ps** se puede comprobar que los contenedores están corriendo:

A white background with black text

Description automatically generated

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

**docker-compose up -d**



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

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

**A screen shot of a computer

Description automatically generated**

Se puede observer 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:

A screen shot of a computer

Description automatically generated

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:

A screenshot of a computer program

Description automatically generated

# Conclusiones

* El Patrón Publisher-Subscriber permite reducir el acoplamiento entre componentes ya que os 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.