Mensajería Publish-Subscribe con RabbitMQ

Juan Diego Roa Porras - 2210086

Kevin Dannie Guzmán Duran - 2211875

Facultad de Ingenierías Fisicomecánicas, UIS

28091: Principios y Prácticas de Desarrollo Orientado a Objetos - C1

Docente Gabriel Rodrigo Pedraza Ferreira

27 de marzo de 2025

TABLA DE CONTENIDOS

1. INTRODUCCIÓN

En el desarrollo de software, existen diversas arquitecturas, como la monolítica, la de microservicios, la de cliente-servidor y la de comunicación asíncrona. Cada una de ellas presenta ventajas y particularidades diseñadas para cumplir con requisitos funcionales específicos.

Este documento tiene como propósito servir de guía y documentación para un ejercicio práctico en el que se implementa una arquitectura de comunicación asíncrona, específicamente el modelo publish/subscribe. En este esquema, intervienen tres actores principales: un productor, encargado de enviar mensajes; un broker de mensajería, responsable de distribuirlos según el tipo de intercambio especificado; y un consumidor, que recibe los mensajes en función de los temas a los que está suscrito.

El ejercicio se ha desarrollado con fines didácticos, buscando no solo implementar la misma lógica en distintas tecnologías, sino también automatizar su despliegue mediante contenedores. Para ello, se ha utilizado Docker, incluyendo herramientas clave como Dockerfile para la creación de imágenes y Docker Compose para la orquestación y gestión eficiente de los contenedores.

2. DESARROLLO

2.1. Front-End

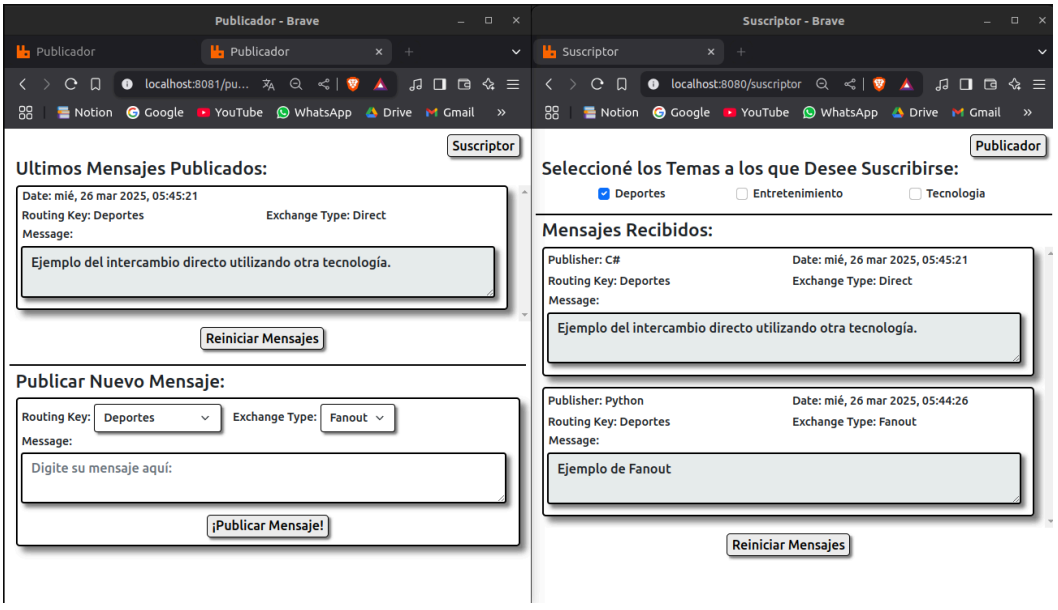
Si bien no era un requisito obligatorio, para este ejercicio se desarrolló un **front-end unificado** con el objetivo de **facilitar** las pruebas y verificar de manera rápida y

visual el funcionamiento del sistema. Para su implementación, se utilizaron **HTML, CSS y JavaScript clásico**, y se creó una imagen de Docker que permite ejecutarlo en un servidor **Nginx**.

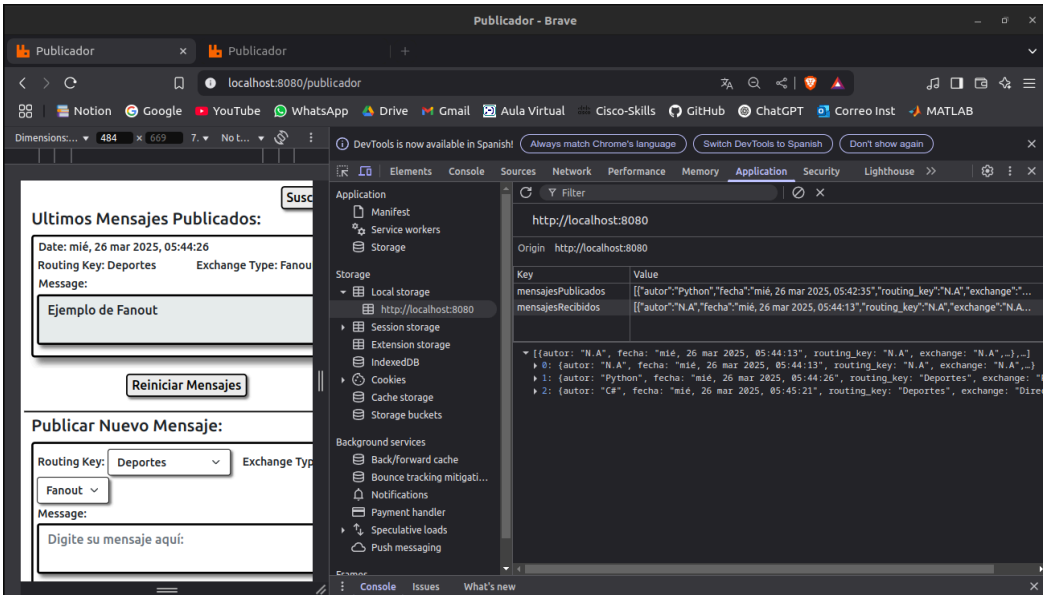
El diseño de la interfaz permite gestionar tanto la funcionalidad del **productor** como la del **consumidor** desde un mismo contenedor. La vista del productor incluye un **formulario sencillo** para publicar nuevos mensajes y un **contenedor de mensajes enviados exitosamente.** Por otro lado, la vista del **consumidor** permite suscribirse a múltiples temas a través de **checkboxes** y visualizar los mensajes recibidos, los cuales se muestran ordenados del más reciente al más antiguo.

Este ejercicio no implementa persistencia mediante bases de datos o volúmenes de Docker. Sin embargo, para mejorar la experiencia del usuario, se decidió almacenar un **"caché" de mensajes en el LocalStorage** del navegador. De esta forma, los mensajes permanecerán disponibles hasta que el usuario los elimine manualmente, ya sea mediante los botones diseñados para esa función o utilizando el atajo **Ctrl + Shift + R** para forzar la recarga de la página.

El resultado final fue el siguiente:



Donde el caché de mensajes se ve así:



Respecto a la implementación, como era de esperarse, se recargó fuertemente en Javascript donde se destacan algunos bloques de código comunes tanto para la vista del productor como, la del consumidor:

- Para estructurar los mensajes y facilitar su mapeo en los contenedores de la interfaz, se definió una clase Mensaje:

```
//Clase para almacenar los mensajes publicados:
let Mensaje = class {
    constructor(autor = window.PUBLISHER, fecha, routing_key,
exchange, mensaje){
        this.autor = autor
        this.fecha = fecha;
        this.routing_key = routing_key;
        this.exchange = exchange;
        this.mensaje = mensaje;
    }
}
```

- Dado que el sistema necesita mostrar la fecha y hora correctamente, se implementó una función que devuelve la hora en formato UTC, considerando la zona horaria de Bogotá (America/Bogota):

```
//Devuelve la fecha actual en formato UTC teniendo en cuenta la zona
horaria
function UTCZonaHoraria(zona="America/Bogota") {
    let fecha = new Date();
```

```
    return fecha.toLocaleString("es-GB", {
        timeZone: zona,
        weekday: 'short',
        day: '2-digit',
        month: 'short',
        year: 'numeric',
        hour: '2-digit',
        minute: '2-digit',
        second: '2-digit',
        hour12: false
    });
}
```

- Para conservar un historial de los mensajes enviados, se almacena un **caché de mensajes** en localStorage. Al iniciar la aplicación, estos mensajes se recuperan y se muestran en pantalla:

```
//Arreglo que carga los mensajes guardados en el localStorage
let mensajesGuardados = localStorage.getItem("mensajesPublicados");
if (mensajesGuardados) {
    mensajesPublicados = JSON.parse(mensajesGuardados);
} else {
    mensajesPublicados = [mensajePorDefecto];
}

function mostrarMensaje(mensaje){
    let divMensaje = document.createElement("div");
    divMensaje.classList.add("mensajePublicado");

    divMensaje.innerHTML = `
        <p class="col">Date: ${mensaje.fecha}</p>
        <div class="row">
            <p class="col">Routing Key: ${mensaje.routing_key}</p>
            <p class="col">Exchange Type: ${mensaje.exchange}</p>
        </div>

        <p>Message:</p>
        <textarea readonly class="form-control" rows="2" >
           ${mensaje.mensaje}
        </textarea>
    `;
    //Agrega el mensaje como el primer hijo del contenedor:
    contenedorMensajes.prepend(divMensaje);
}
```

```
mensajesPublicados.forEach(mensaje => {
    mostrarMensaje(mensaje);
});
```

En general, tanto **publisher.js** como **subscriber.js** comparten gran parte de la lógica, con pequeñas adaptaciones según su funcionalidad específica. Algunas de estas diferencias incluyen la carga dinámica de los checkboxes a partir de un arreglo de temas en el subscriber, y la inclusión del autor del mensaje en el contenedor de mensajes recibidos, entre otros pequeños ajustes.

No obstante, la principal diferencia entre ambas vistas radica en los endpoints que consumen. Aunque esto se abordará con más detalle en la sección sobre el despliegue, ambos archivos **.js** utilizan dos variables de entorno clave:

- **API_URL:** Define la dirección IP y el puerto donde se ejecuta el backend correspondiente.
- **PUBLISHER:** Permite identificar fácilmente qué tecnología de backend está siendo consumida por ese frontend específico.

Donde la solicitud hecha con fetch en **publisher.js** se ve de la siguiente manera:

```
//Función para enviar el mensaje consumiendo la API
async function mensajeEnviado(nuevoMensaje) {
    const respuesta = await fetch("http://"
                                  + window.API_URL
                                  + "/publisher/publishMessage",
            {
            method: "POST",
            body: JSON.stringify(nuevoMensaje),
            headers: {
              "Content-type": "application/json; charset=UTF-8"
            }
        })
        //Manejo de excepciones
}

//Event Listener para publicar los mensajes:
document.addEventListener("DOMContentLoaded", function () {
    document.getElementById("publicarMensaje")
            .addEventListener("click", async function () {
        let nuevoMensaje = new Mensaje(
```

```
            window.PUBLISHER,
            UTCZonaHoraria(),
            document.getElementById("selectRK").value,
            document.getElementById("selectExchange").value,
            document.getElementById("areaMensaje").value
        );

        const envioExitoso = await mensajeEnviado(nuevoMensaje);
        if (!envioExitoso) {
            return;
        }

        //Lógica para mostrar el mensaje en el contenedor

        });
```

Y la solicitud hecha en **subscriber.js** se ve de esta manera:

```
//Se llamará cada tres segundos la función consumirMensajes
const intervalID = setInterval(consumirMensajes, 3000);


//Función para recibir y mostrar los nuevos mensajes de rabbitMQ
async function consumirMensajes() {
    try {
        const respuesta = await fetch("http://"
                                      + window.API_URL
                                      + "/consumer/getMessages",
            {
            method: "POST",
            body: JSON.stringify(temasSuscritos),
            headers: {
              "Content-type": "application/json; charset=UTF-8"
            }
        })

        if (!respuesta.ok) {
            const errorData = await respuesta.json();
            alert("Error al consumir mensajes");
            return false
        }

        const data = await respuesta.json();
        if (data != null){
            data.mensajes.forEach(mensaje => {
                let nuevoMensaje = new Mensaje(
                    mensaje.autor,
                    mensaje.fecha,
                    mensaje.routing_key,
                    mensaje.exchange,
```

```
                    mensaje.mensaje
                );

            //Lógica para mostrar el mensaje en el contenedor

    } catch (error) {
        alert("Error al consumir mensajes: " + error.message);
        return false;
    }
}
```

Para gestionar correctamente el acceso a los archivos **CSS**, **JavaScript** y, en general, a las distintas rutas y documentos del frontend, fue necesario implementar el siguiente archivo de configuración para **Nginx**:

```
events {}

http {
    include mime.types;
    default_type  application/octet-stream;
    server {
        listen 80;

        root /usr/share/nginx/html;
        index publisher.html;

        # Redirige "/" → "/publicador", manteniendo host y puerto
        location = / {
            return 302 http://$http_host/publicador;
        }

        location /publicador {
            try_files $uri /publisher.html;
        }

        location /suscriptor {
            try_files $uri /subscriber.html;
        }

        location /estaticos/ {
            alias /usr/share/nginx/html/estaticos/;
            try_files $uri =404;
        }
    }
}
```

2.2. Back-End

2.2.1. Python

El primer backend implementado fue el de Python, utilizando **FastAPI** para crear la API, la cual, por defecto, se ejecuta en el puerto **8000**. Para facilitar su despliegue, se creó un archivo **requirements.txt** que incluye todas las dependencias necesarias (**FastAPI, Pika, Uvicorn y Pydantic**). Además, se incorporó un archivo **env.py** para gestionar la única variable de entorno del proyecto: la dirección IP del host de **RabbitMQ**. Finalmente, se agregó un archivo **models.py**, cuya función es estructurar los datos de los mensajes, simplificando su publicación y consumo, similar a lo realizado en el frontend.

Archivo **env.py**:

```
import os
#Variable de entorno con la URL de rabbitMQ, por default será localhost
RABBIT_URL = os.getenv("RABBIT_URL", "localhost")
```

Archivo **models.py:**

```
class Message(BaseModel):
    autor: str
    fecha: str
    routing_key: str
    exchange: str
    mensaje: str
```

En el archivo **main.py**, se configura el **CORS** mediante un middleware, lo que permite gestionar las solicitudes desde el frontend. Además, se integran los endpoints utilizando **routers**, asignándoles prefijos específicos para garantizar la coherencia con las solicitudes **HTTP** del frontend. Finalmente, la aplicación se ejecuta utilizando la librería **Uvicorn**.

```
app = FastAPI()

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_methods=["*"],
```

```
    allow_headers=["*"]
)

# Incluir routers -> Acceso a los endpoints de consumer y publisher
app.include_router(consumer_router, prefix="/consumer")
app.include_router(publisher_router, prefix="/publisher")

# El host=0.0.0.0 permite acceder a la API desde fuera del contenedor
if __name__ == "__main__":
    uvicorn.run("main:app", host="0.0.0.0", port=8000)
```

A continuación, se explica el funcionamiento del archivo **publisher.py**, enfocándose en el endpoint **/publishMessage**. En este, por cada solicitud se establece una conexión con **RabbitMQ**, se recibe una entidad de tipo **Mensaje**, y utilizando su **exchange** y **routing key**, el mensaje se publica correctamente. Posteriormente, la conexión se cierra automáticamente, lo que no afecta el funcionamiento de la aplicación, ya que es el **broker (Rabbit)** el encargado de redirigir el mensaje.

```
router = APIRouter()

#Conexión a RabbitMQ
def getConnectionInfo():
    connection =
pika.BlockingConnection(pika.ConnectionParameters(host=RABBIT_URL))
    return connection, connection.channel()

#EndPoint Para publicar mensajes:
@router.post("/publishMessage")
async def publishMessage(message: Message):
    connection, channel = getConnectionInfo()
    channel.basic_publish(
        exchange = 'amq.' + message.exchange.lower(),
        routing_key = message.routing_key.lower(),
        body = message.model_dump_json())
    channel.close()
    connection.close()
    return message
```

El archivo **consumer.py** establece una conexión con **RabbitMQ** y crea una **cola temporal**. Para simplificar la implementación, se permitió que **RabbitMQ** asigne automáticamente el nombre de la cola. Cuando el backend deja de ejecutarse, tanto la conexión con el **broker** como la cola temporal se eliminan. Además, se configura la cola para recibir cualquier mensaje proveniente de un **exchange** de tipo **fanout**.

Posteriormente, se implementó una función que actualiza dinámicamente los **bindings** de la cola temporal del consumidor a partir de un arreglo denominado **"temas suscritos"**. Si el cliente se suscribe a un tema, se crean dos **bindings** con la misma **routing key**: uno asociado a un **exchange** de tipo **direct** y otro a un **exchange** de tipo **topic**. Esta decisión fue puramente didáctica, ya que en la práctica ambos **bindings** serían equivalentes. En caso de que el consumidor deje de estar suscrito a un tema, los **bindings** correspondientes se eliminan de manera preventiva. **RabbitMQ** no genera excepciones al intentar eliminar un **binding** inexistente, lo que simplifica la lógica al no requerir verificaciones previas.

Finalmente, el endpoint **/getMessages** recibe un arreglo de cadenas de texto que se pasa a la función encargada de actualizar los **bindings**. Luego, se recuperan los mensajes almacenados en la cola y se devuelven en formato **JSON** como una lista de mensajes recibidos.

```
router = APIRouter()

#Conexión a RabbitMQ
connection =
pika.BlockingConnection(pika.ConnectionParameters(host=RABBIT_URL))
channel = connection.channel()

#Creación de cola temporal específica para cada cliente
result = channel.queue_declare(queue='', exclusive=True)
queue_name = result.method.queue

#Se especifica el binding de fanout
channel.queue_bind(exchange='amq.fanout', queue=queue_name)

def updateBindings(temas_suscritos):
    aux = ['Deportes', 'Entretenimiento', 'Tecnologia']
    for tema in aux:
        if tema not in temas_suscritos:
            #Eliminamos los bindings
            channel.queue_unbind(exchange='amq.direct',
                queue=queue_name, routing_key=tema.lower())
            channel.queue_unbind(exchange='amq.topic',
                queue=queue_name, routing_key=tema.lower())
        else:
            #Creamos los bindings
            channel.queue_bind(exchange='amq.direct', queue=queue_name,
                routing_key=tema.lower())
            channel.queue_bind(exchange='amq.topic', queue=queue_name,
                routing_key=tema.lower())
```

```python
@router.post("/getMessages")
async def getMessages(temas_suscritos: List[str] = Body(...)):
    updateBindings(temas_suscritos)

    mensajes = []

    while True:
     method_frame, header_frame, body = channel.basic_get(
         queue=queue_name, auto_ack=True)
     if method_frame:
         body_str = body.decode("utf-8")
         mensaje = json.loads(body_str)
         mensajes.append(mensaje)
     else:
         break  # No hay más mensajes en la cola
     return {"mensajes": mensajes}
```

2.2.2. C#

El segundo backend implementado fue el de **C#**, utilizando **.NET** para crear la API. En retrospectiva, para facilitar su despliegue, se debió configurar el puerto de ejecución a través de **Kestrel**, permitiendo que la aplicación recibiera solicitudes **HTTP** desde fuera del contenedor. Finalmente, el backend se ejecutó en el puerto **5040**, y la configuración correspondiente en appsettings.json se vería de la siguiente manera:

```json
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "Kestrel": {
    "Endpoints": {
      "Http": {
        "Url": "http://0.0.0.0:5040"
      }
    }
  }
}
```

Luego se especificó el **modelo** de mensaje:

```
public class Message {
    public required string autor {get; set;}
    public required string fecha {get; set;}
    public required string routing_key {get; set;}
    public required string exchange {get; set;}
    public required string mensaje {get; set;}
}
```

En el archivo **Program.cs**, se definió la configuración del **CORS**, la habilitación de **Swagger** para la documentación, la inicialización del servicio de **RabbitMQ** y el acceso a los endpoints de la API.

En C# y .NET, era necesario declarar todas estas configuraciones en el **builder** antes de crear la aplicación. Luego, una vez inicializada, se debía indicar explícitamente que utilizara dicha configuración, lo que implicaba agregar algunas líneas de código adicionales.

```
using Microsoft.OpenApi.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddPolicy("AllowAll", policy =>
    {
        policy
            .AllowAnyOrigin()
            .AllowAnyMethod()
            .AllowAnyHeader();
    });
});

builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen(c => {
    c.SwaggerDoc("v1", new OpenApiInfo {Title = "Backend C# API",
Description="Documentation for the Backend with C#", Version="v1"});
});

builder.Services.AddSingleton<RabbitMqConsumerServiceAsync>();

//Especificar el puerto donde se va a ejecutar
builder.WebHost.UseKestrel().UseUrls("http://0.0.0.0:5040");

var app = builder.Build();
app.UseCors("AllowAll");
```

```
var rabbitService =
app.Services.GetRequiredService<RabbitMqConsumerServiceAsync>();
await rabbitService.InitializeAsync();

if (app.Environment.IsDevelopment() || true) //Permite acceder al Swagger
en "Producción"
{
    app.UseSwagger();
    app.UseSwaggerUI(
        c => {
            c.SwaggerEndpoint("/swagger/v1/swagger.json",
                              "Backend C# API V1");
        }
    );
}

app.PublisherEndpoints();
app.ConsumerEndpoints();

app.Run();
```

El endpoint para publicar mensajes siguió la misma lógica que en el backend anterior. Se accedió a la variable de entorno **RABBIT_URL**, asignándole un valor por defecto en caso de que no estuviera definida. Luego, se estableció la conexión con **RabbitMQ** y se realizó un preprocesamiento del objeto de tipo **mensaje**, recibido en el **body** de la solicitud, convirtiéndolo en una lista de **bytes** para su envío al **broker**. Finalmente, tras publicar exitosamente el mensaje, se cerró la conexión.

```
using System.Text;
using System.Text.Json;
using RabbitMQ.Client;

public static class Publisher {
    public static void PublisherEndpoints(this WebApplication app){

        app.MapPost("publisher/publishMessage", async (Message mensaje) => {
            //Conexion a RabbitMQ

            //Consumir de la variable de entorno
            var RABBIT_URL = Environment.GetEnvironmentVariable("RABBIT_URL")
                             ?? "localhost";

            var factory = new ConnectionFactory { HostName = RABBIT_URL };
            using var connection = await factory.CreateConnectionAsync();
            using var channel = await connection.CreateChannelAsync();

            //Convertir el mensaje en bytes
```

```
            string mensajeAsString = JsonSerializer.Serialize(mensaje);
            byte[] mensajeAsBytes = Encoding.UTF8.GetBytes(mensajeAsString);

            //Publicar el mensaje
            await channel.BasicPublishAsync(
                exchange: "amq."+ mensaje.exchange.ToLower(),
                routingKey: mensaje.routing_key.ToLower(),
                body: mensajeAsBytes
            );

            //Cerrar la conexión
            await channel.CloseAsync();
            await connection.CloseAsync();
        });
    }
}
```

El endpoint **getMessages** se simplificó en gran medida al delegar la gran mayoría de la lógica a la instancia del servicio de RabbitMQ:

```
public static class Consumer
{
    public static void ConsumerEndpoints(this WebApplication app)
    {
        app.MapPost("consumer/getMessages", async
(RabbitMqConsumerServiceAsync rabbitService, string[] temasSuscritos) =>
        {
            await rabbitService.UpdateBindingsAsync(temasSuscritos);
            var mensajes = await rabbitService.GetMessagesAsync();
            return Results.Json(new { mensajes });
        });
    }
}
```

El servicio de RabbitMQ se inicializa estableciendo la conexión, declarando una cola temporal para el consumidor y asignándole un **binding** a un **exchange** de tipo *fanout*. Además, se implementa una función para actualizar los **bindings**, siguiendo la misma lógica que el backend en Python, y otra función encargada de recuperar uno o varios mensajes de la cola, iterando a través de un *do-while* según su estado.

```
public class RabbitMqConsumerServiceAsync
{
    private IConnection connection;
    private IChannel channel;
    private string queueName;
```

```
// Inicialización
public async Task InitializeAsync()
{
    var RABBIT_URL = Environment.GetEnvironmentVariable("RABBIT_URL")
                        ?? "localhost";

    var factory = new ConnectionFactory { HostName = RABBIT_URL };
    connection = await factory.CreateConnectionAsync();
    channel = await connection.CreateChannelAsync();

    var result = await channel.QueueDeclareAsync(
                                queue: "",
                                durable: false,
                                exclusive: true,
                                autoDelete: true);
    queueName = result.QueueName;

    await channel.QueueBindAsync(queue: queueName,
                                exchange: "amq.fanout",
                                routingKey: "");
}

public async Task UpdateBindingsAsync(string[] temasSuscritos)
{
    string[] aux = { "Deportes", "Entretenimiento", "Tecnologia" };
    foreach (string tema in aux)
    {
        if (!temasSuscritos.Contains(tema, StringComparer.OrdinalIgnoreCase))
        {
            await channel.QueueUnbindAsync(queue: queueName,
                                           exchange: "amq.direct",
                                           routingKey: tema.ToLower());

            await channel.QueueUnbindAsync(queue: queueName,
                                           exchange: "amq.topic",
                                           routingKey: tema.ToLower());
        }
        else
        {
            await channel.QueueBindAsync(queue: queueName,
                                         exchange: "amq.direct",
                                         routingKey: tema.ToLower());

            await channel.QueueBindAsync(queue: queueName,
                                         exchange: "amq.topic",
                                         routingKey: tema.ToLower());
        }
    }
}
```

```
    public async Task<List<Message>> GetMessagesAsync()
    {
        List<Message> mensajes = new();
        BasicGetResult result;
        do
        {
            result = await channel.BasicGetAsync(queueName, autoAck: true);
            if (result != null)
            {
                byte[] body = result.Body.ToArray();
                string jsonString = Encoding.UTF8.GetString(body);
                Message? mensaje = JsonSerializer.Deserialize<Message>
                                    (jsonString);
                if (mensaje is not null) mensajes.Add(mensaje);
            }
        } while (result != null);

        return mensajes;
    }
}
```

2.2.3. Java

El último backend implementado fue el de Java, utilizando Spring Boot. Al igual que en los anteriores, fue necesario especificar la variable de entorno **RABBIT_URL**, definida en el archivo **application.properties**. Además, se creó una clase de configuración para gestionar el CORS, permitiendo solicitudes HTTP desde cualquier IP. También se implementó una clase **Message**, que sirvió como modelo, y una clase para interactuar con RabbitMQ, similar al servicio utilizado en el backend de C#. Finalmente, se desarrollaron los correspondientes **Rest Controllers** para exponer los endpoints.

application.properties:

```
spring.application.name=tallerRabbitMQ

server.port=7070
spring.rabbitmq.host = ${RABBIT_URL:localhost}
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
```

CORSConfig.java:

```
@Configuration
public class CorsConfig implements WebMvcConfigurer{
    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**")
                // Usa allowedOriginPatterns en lugar de allowedOrigins
                .allowedOriginPatterns("*")
                .allowedMethods("GET", "POST", "PUT", "DELETE")
                .allowedHeaders("*")
                .allowCredentials(true); // Permite credenciales
    }
}
```

Message.java:

```
@Data
public class Message implements Serializable{
    private String autor;
    private String fecha;
    private String routing_key;
    private String exchange;
    private String mensaje; //Con sus respectivos Getters and Setters
}
```

RabbitMQ Config:

```
@Configuration
public class RabbitMQConfig {
    public static final String FANOUT_EXCHANGE = "amq.fanout";
    public static final String DIRECT_EXCHANGE = "amq.direct";
    public static final String TOPIC_EXCHANGE = "amq.topic";

    @Bean
    public FanoutExchange fanoutExchange() {
        return new FanoutExchange(FANOUT_EXCHANGE);
    }

    @Bean
    public DirectExchange directExchange() {
        return new DirectExchange(DIRECT_EXCHANGE);
    }

    @Bean
    public TopicExchange topicExchange() {
        return new TopicExchange(TOPIC_EXCHANGE);
    }

    @Bean
    public RabbitAdmin rabbitAdmin(ConnectionFactory connectionFactory) {
```

```
        return new RabbitAdmin(connectionFactory);
    }

    @Bean
    public AmqpTemplate rabbitMQTemplate(ConnectionFactory connectionFactory) {
        RabbitTemplate rabbitTemplate = new RabbitTemplate(connectionFactory);
        rabbitTemplate.setMessageConverter(jsonMessageConverter());
        return rabbitTemplate;
    }

    @Bean
    public MessageConverter jsonMessageConverter() {
        return new Jackson2JsonMessageConverter();
    }
}
```

Destacándose que el uso de la notación **@Bean** es fundamental, porque le da las habilidades a Spring Boot para que maneje automáticamente la conexión y configuración de RabbitMQ, facilitando el envío y recepción de mensajes en la aplicación.

Publisher.java:

```
@Component
@RestController
@RequestMapping("/publisher")
public class Publisher {
    private final RabbitTemplate rabbitTemplate;

    @Autowired
    public Publisher(RabbitTemplate rabbitTemplate) {
        this.rabbitTemplate = rabbitTemplate;
    }

    @PostMapping("/publishMessage")
    public ResponseEntity<Message> publishMessage(@RequestBody Message message)
{
        String exchangeName = "amq." + message.getExchange().toLowerCase();
        String routingKey = message.getRouting_key().toLowerCase();
        rabbitTemplate.convertAndSend(exchangeName, routingKey, message);
        return ResponseEntity.ok(message);
    }
}
```

Consumer.java:

```java
@Component
@RestController
@RequestMapping("/consumer")
public class Consumer {
    // Atributos + Constructor
    private String declareTempQueue() {
        Queue tempQueue = new AnonymousQueue();
        rabbitAdmin.declareQueue(tempQueue);
        return tempQueue.getName();
    }

    // Metodo para hacer update de los bindings

    // Endpoint getMessages
    }
```

```java
private void updateBindings(List<String> temasSuscritos) {
    List<String> temasDisponibles = List.of(e1:"Deportes", e2:"Entretenimiento", e3:"Tecnologia");

    for (String tema : temasDisponibles) {
        if (!temasSuscritos.contains(tema)) {
            rabbitAdmin.removeBinding(BindingBuilder.bind(new Queue(queueName)).to(topicExchange).with(tema.toLowerCase()));
            rabbitAdmin.removeBinding(BindingBuilder.bind(new Queue(queueName)).to(directExchange).with(tema.toLowerCase()));
        } else {
            rabbitAdmin.declareBinding(BindingBuilder.bind(new Queue(queueName)).to(topicExchange).with(tema.toLowerCase()));
            rabbitAdmin.declareBinding(BindingBuilder.bind(new Queue(queueName)).to(directExchange).with(tema.toLowerCase()));
        }
    }
}
```

```java
@PostMapping("/getMessages")
public ResponseEntity<Map<String, List<Map<String, String>>>> getMessages(@RequestBody List<String> temasSuscritos) {
    updateBindings(temasSuscritos);

    List<Map<String, String>> mensajes = new ArrayList<>();
    Message message;

    while ((message = rabbitTemplate.receive(queueName)) != null) {
        String body = new String(message.getBody(), StandardCharsets.UTF_8);

        // Convertimos el JSON del mensaje a un Map
        ObjectMapper objectMapper = new ObjectMapper();
        try {
            Map<String, String> mensajeMap = objectMapper.readValue(body, new TypeReference<Map<String, String>>() {});

            // Asegurar que tenga la estructura correcta
            Map<String, String> mensajeFinal = new HashMap<>();
            mensajeFinal.put(key:"autor", mensajeMap.getOrDefault(key:"autor", defaultValue:"Desconocido"));
            mensajeFinal.put(key:"fecha", mensajeMap.getOrDefault(key:"fecha", LocalDateTime.now().toString()));
            mensajeFinal.put(key:"routing_key", mensajeMap.getOrDefault(key:"routing_key", defaultValue:"Sin clave"));
            mensajeFinal.put(key:"exchange", mensajeMap.getOrDefault(key:"exchange", defaultValue:"Desconocido"));
            mensajeFinal.put(key:"mensaje", mensajeMap.getOrDefault(key:"mensaje", defaultValue:"Sin contenido"));

            mensajes.add(mensajeFinal);
        } catch (JsonProcessingException e) {
            e.printStackTrace();
        }
    }

    // Devolver la estructura correcta
    Map<String, List<Map<String, String>>> response = new HashMap<>();
    response.put(key:"mensajes", mensajes);

    return ResponseEntity.ok(response);
}
```

3. DESPLIEGUE

3.1. Front-End

Como se mencionó anteriormente, el frontend se desplegó en un servidor Nginx, utilizando la siguiente configuración:

front > nginx.conf

```
events {}

http {
    include mime.types;
    default_type  application/octet-stream;
    server {
        listen 80;

        root /usr/share/nginx/html;
        index publisher.html;

        # Redirige "/" → "/publicador", manteniendo host y puerto
        location = / {
            return 302 http://$http_host/publicador;
        }

        location /publicador {
            try_files $uri /publisher.html;
        }

        location /suscriptor {
            try_files $uri /subscriber.html;
        }

        location /estaticos/ {
            alias /usr/share/nginx/html/estaticos/;
            try_files $uri =404;
        }
    }
}
```

Este archivo define el puerto en el que se ejecutará la aplicación, configura la redirección automática a la ruta **/publicador** y garantiza la correcta carga de los archivos de estilos y scripts **.js**.

Para el Dockerfile, se utilizó una imagen base de Nginx, y siguiendo nuestras preferencias personales, se instalaron **bash** y **nano** para facilitar el acceso y la edición de archivos dentro del contenedor si fuese necesario. Luego, se copió el código fuente y la configuración a sus respectivos directorios de Nginx, junto con el script script.sh, que se encargó de generar dinámicamente la variable de entorno utilizada por los archivos **.js**. Finalmente, Nginx se ejecuta por defecto.

```
FROM nginx:alpine

#Instalamos nano y bash
RUN apk update && \
    apk add bash nano

#Copiamos la carpeta frontend a la ruta de nginx
```

```
COPY ./frontend/ /usr/share/nginx/html/

#Copiamos la configuración de nginx
COPY nginx.conf /etc/nginx/nginx.conf

#Copiamos el script para generar la variable de entorno en config.js
COPY docker-entrypoint.sh /docker-entrypoint.sh

#Le damos permisos de ejecución al script
RUN chmod +x /docker-entrypoint.sh

#Ejecutamos el script
ENTRYPOINT ["/docker-entrypoint.sh"]

#Exponemos el puerto
EXPOSE 80

#Ejecutamos nginx
CMD ["nginx", "-g", "daemon off;"]
```

Donde el mencionado **docker-eentrypoint.sh** se ve de la siguiente manera:

front > docker-entrypoint.sh

```
1  #!/bin/sh
2
3  # Ruta donde esta el config.js dentro del contenedor
4  CONFIG_PATH=/usr/share/nginx/html/estaticos/config.js
5
6  # Escribe la variable de entorno en el config.js dentro de la carpeta correcta
7  echo "window.API_URL = '${API_URL}';" > $CONFIG_PATH
8  echo "window.PUBLISHER = '${PUBLISHER}';" >> $CONFIG_PATH ## >> Significa Append
9
10 exec "$@"
```

3.2. Back-End

3.2.1. Python

La creación de la imagen para el backend de Python fue bastante sencilla. Se utilizó una imagen base liviana de Python y, al igual que en el frontend, se instalaron **bash** y **nano** para facilitar el acceso al contenedor en caso de ser necesario. Se asignó un valor por defecto a la variable de entorno **RABBIT_URL**, luego se copió el archivo **requirements.txt** y se instalaron las dependencias con **pip install -r**. Finalmente, se expuso el **puerto** correspondiente y se configuró la ejecución de main.py por defecto.

```
FROM python:3.10.16-slim

# Actualiza e instala bash y nano
RUN apt-get update && \
    apt-get install -y --no-install-recommends \
    bash \
    nano \
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/*

#Generamos la variable de entorno de la URL de RabbitMQ
ENV RABBIT_URL=taller_rabbitmq

#Definimos el directorio de la aplicaciøn
WORKDIR /home/python

#Copiamos e Instalamos las dependencias necesarias para correr el back
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

#Copiamos el codigo fuente del backend
COPY . .

#Exponemos el puerto
EXPOSE 8000

#Se ejecutará por default el main.py
ENTRYPOINT ["python3", "main.py"]
```

3.2.2. C#

Este Dockerfile sigue una estrategia de dos etapas para optimizar el despliegue de una aplicación en C# con .NET 8.0. En la primera etapa **(build)**, se utiliza la imagen del SDK de .NET para restaurar dependencias, compilar el código y generar una versión optimizada en modo **Release**. Luego, en la segunda etapa (**runtime**), se usa una imagen más liviana de ASP.NET, donde se copian los archivos compilados y se expone el puerto **5040**. Además, se define el valor por defecto de la variable de entorno para RabbitMQ y se establece el comando de inicio para ejecutar la aplicación.

Cabe destacar que aunque se buscó optimizar lo máximo posible el dockerfile, igual se terminó generando una imagen que pesaba más de 200mb y que hacer el **docker build** tomaba alrededor de 10 a 15 segundos.

```
# Etapa 1: Build
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build

# Establecer directorio de trabajo
WORKDIR /app

# Copiar archivos de proyecto y restaurar dependencias
COPY *.csproj ./
RUN dotnet restore

# Copiar el resto del código
COPY . ./

# Publicar la aplicación en modo release
RUN dotnet publish -c Release -o out

#-------------------------------------------------------

# Etapa 2: Runtime
FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS runtime

# Establecer directorio de trabajo
WORKDIR /app

# Copiar desde la etapa de build
COPY --from=build /app/out ./

# Exponer el puerto
EXPOSE 5040

# Variable de entorno opcional para RabbitMQ (puede sobreescribirse)
ENV RABBIT_URL=taller_rabbitmq

# Comando para ejecutar la app
ENTRYPOINT ["dotnet", "C#.dll"]
```

3.2.3. Java

Se tomó como base el Dockerfile del repositorio utilizado en el curso, manteniendo una estrategia de construcción en dos etapas. Primero, se utiliza una imagen de **Maven** para compilar el código fuente y generar el archivo **.jar**, copiando el pom.xml y el código fuente dentro del contenedor. Luego, en la segunda etapa, se parte de una imagen de Java, donde se define la variable de entorno **RABBIT_URL**, se copia el .jar generado en la etapa anterior y se ejecuta la aplicación por defecto. Finalmente, se expone el puerto 7070 para permitir el acceso a la API.

```
#--- BUILDING ---#

#Utilizamos la imagen de maven para crear el artefacto desplegable (jar) del
proyecto
FROM maven:3.9.6-eclipse-temurin-17 AS maven

#Copiamos el codigo fuente dentro de la imagen
COPY ./src /usr/local/app/src

#Copiamos el pom dentro de la imagen y en la raiz del directorio de la app
COPY ./pom.xml /usr/local/app

#Definimos el directorio de la aplicaiøn
WORKDIR /usr/local/app

#Ejecutamos los comandos clean y package propios de maven para generar el jar,
#saltamos los test para que no de error
RUN mvn clean package -DskipTests

#--- DESPLIEGUE ---#

#Partimos ahora de una imagen de java
FROM eclipse-temurin:17

#Se definen las variables de entorno
ENV RABBIT_URL=taller_rabbitmq

# Copia el jar ejecutable de la imagen auxiliar alias mavn
COPY  --from=maven /usr/local/app/target/tallerRabbitMQ-0.0.1-SNAPSHOT.jar
/usr/share/backJava.jar

# Lanza el ejecutable usando java
CMD ["java", "-jar", "/usr/share/backJava.jar"]

# Expone el puerto 7070
EXPOSE 7070
```

4. DOCKER COMPOSE

Para facilitar el despliegue y realizar pruebas de manera eficiente, se definieron tres versiones del entorno de ejecución, cada una adaptada a diferentes configuraciones del total de equipos y con características específicas. A continuación, se detallarán estas versiones del archivo **docker-compose.yaml.**

4.1. Versión 1 (v1)

Se desplegarán un total de siete contenedores: uno para RabbitMQ, que funcionará como broker, tres para los backends (uno por cada tecnología) y tres para los frontends, cada uno conectado a un backend diferente. Dado que todos los contenedores se ejecutarán en la misma máquina, las variables de entorno compartirán el mismo valor, ya sea **localhost** o la dirección IP de una máquina virtual (**10.6.101.107**).

Además, se creará una red interna que permitirá a los backends comunicarse con el broker utilizando el **nombre del contenedor**. Como los backends dependen de su conexión con RabbitMQ, en el caso de las APIs de Spring Boot y .NET, si no se establece correctamente la conexión, la ejecución del servicio se detiene. Para evitar estos inconvenientes, se emplearon dos mecanismos: **healthcheck**, que monitorea periódicamente el estado del broker, y **condition**, que garantiza que los backends solo se inicien cuando el contenedor de RabbitMQ esté completamente operativo.

```
services:
  rabbitmq:
    image: rabbitmq:3-management
    container_name: taller_rabbitmq
    hostname: taller_rabbitmq
    networks:
      - taller_rabbitmq
    ports:
      - 5672:5672
      - 15672:15672
    healthcheck:
      test: ["CMD", "rabbitmqctl", "status"]
      interval: 10s
      timeout: 5s
      retries: 5

  back_python:
    image: back_python
    container_name: back_python
    networks:
      - taller_rabbitmq
    environment:
      - RABBIT_URL=taller_rabbitmq
    depends_on:
      rabbitmq:
        condition: service_healthy
    ports:
      - 8000:8000
```

```
  front_python:
    image: front_taller
    container_name: front_python
    environment:
      - API_URL=localhost:8000
      - PUBLISHER=Python
    ports:
      - 8080:80

  back_dotnet:
    image: back_dotnet
    container_name: back_dotnet
    networks:
      - taller_rabbitmq
    environment:
      - RABBIT_URL=taller_rabbitmq
    depends_on:
      rabbitmq:
        condition: service_healthy
    ports:
      - 5040:5040

  front_dotnet:
    image: front_taller
    container_name: front_dotnet
    environment:
      - API_URL=localhost:5040
      - PUBLISHER=C#
    ports:
      - 8081:80

  #Back en java (cambia la imagen) y front en java (cambia la API URL)

networks:
  taller_rabbitmq:
    driver: bridge
```

4.2. Versión 2 (v2)

En esta versión, nuevamente se desplegarán siete contenedores, pero con una diferencia clave: el contenedor de RabbitMQ estará **separado** de los contenedores del backend y frontend. Esto simplifica significativamente la estructura del docker-compose, aunque implica algunos **cambios en la configuración**.

Dado que RabbitMQ ya no forma parte del mismo docker-compose, no será posible utilizar **healthcheck** ni **depends_on** con **condition** para gestionar el inicio de los backends. En su lugar, se emplea **restart: always**, lo que permitirá que los contenedores del backend se reinicien automáticamente hasta que logren conectarse con RabbitMQ. Además, la variable de entorno **RABBIT_URL** se actualizará según corresponda en cada caso.

Docker-compose.yaml para el rabbit host:

```yaml
services:
  rabbitmq:
    image: rabbitmq:3-management
    container_name: taller_rabbitmq
    hostname: taller_rabbitmq
    ports:
      - 5672:5672
      - 15672:15672
```

Docker-compose.yaml para el host del back y el front:

```yaml
services:
  #Faltan los de Python para ahorrar espacio

  back_dotnet:
    image: back_dotnet
    container_name: back_dotnet
    environment:
      - RABBIT_URL=10.6.101.107 #MV de Rabbit
    restart: always
    ports:
      - 5040:5040

  front_dotnet:
    image: front_taller
    container_name: front_dotnet
    environment:
      - API_URL=localhost:5040
      - PUBLISHER=C#
    ports:
      - 8081:80

  back_java:
    image: back_java
    container_name: back_java
    environment:
      - RABBIT_URL=10.6.101.107 #MV de Rabbit
    restart: always
    ports:
      - 7070:7070
```

```
  front_java:
    image: front_taller
    container_name: front_java
    environment:
      - API_URL=localhost:7070
      - PUBLISHER=Java
    ports:
      - 8082:80
```

4.3. Versión 3 (v3)

En esta versión final, nuevamente se desplegarán siete contenedores, pero con una diferencia clave: se utilizarán tres máquinas en lugar de dos. Estas pueden ser dos máquinas virtuales y una máquina local, o tres máquinas virtuales con direcciones IP independientes.

Como resultado, el contenedor de RabbitMQ permanecerá aislado, manteniendo exactamente la misma configuración de la versión anterior. Sin embargo, el backend y el frontend se ejecutarán en equipos separados, distribuyendo así la carga entre las máquinas. A continuación, se presentan los archivos docker-compose.yaml correspondientes a esta configuración.

Docker-compose.yaml para el rabbit host:

```
services:
  rabbitmq:
    image: rabbitmq:3-management
    container_name: taller_rabbitmq
    hostname: taller_rabbitmq
    ports:
      - 5672:5672
      - 15672:15672
```

Docker-compose.yaml para el host de los backs:

```
services:
  back_python:
    image: back_python
    container_name: back_python
    environment:
      - RABBIT_URL=10.6.101.107 #MV de Rabbit
    restart: always
    ports:
      - 8000:8000
```

```
  back_dotnet:
    image: back_dotnet
    container_name: back_dotnet
    environment:
      - RABBIT_URL=10.6.101.107 #MV de Rabbit
    restart: always
    ports:
      - 5040:5040

  back_java:
    image: back_java
    container_name: back_java
    environment:
      - RABBIT_URL=10.6.101.107 #MV de Rabbit
    restart: always
    ports:
      - 7070:7070
```

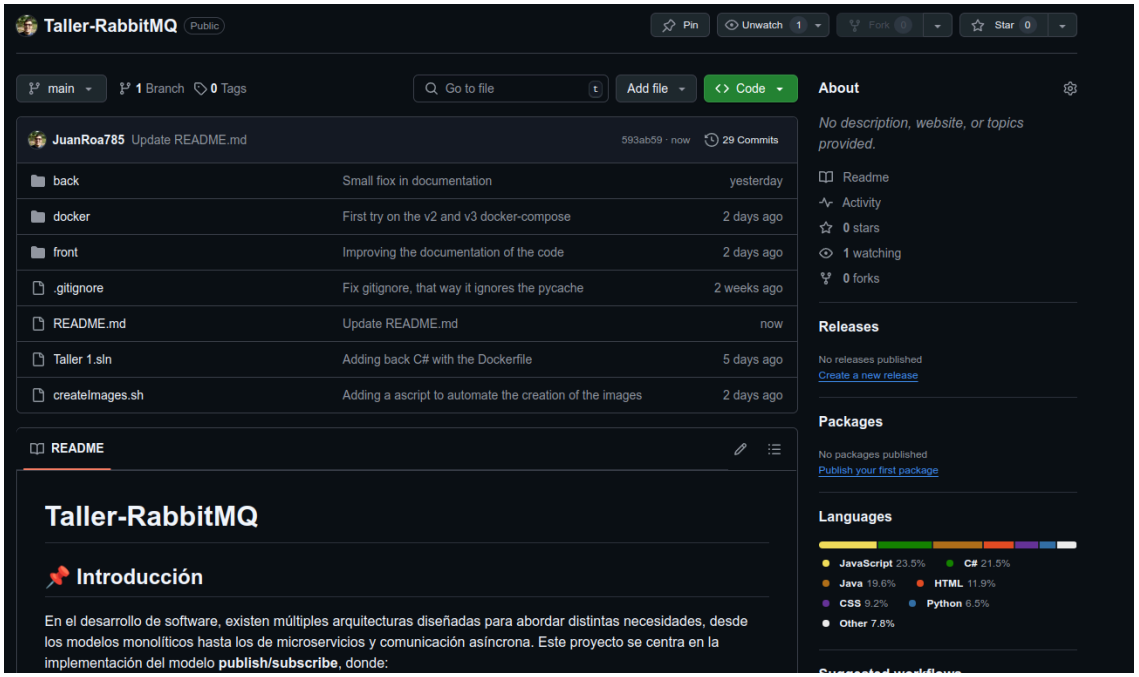
Docker-compose.yaml para el host de los fronts:

```
services:
  front_python:
    image: front_taller
    container_name: front_python
    environment:
      - API_URL=10.6.101.100:8000 #IP contenedor Back (Mismo puerto)
      - PUBLISHER=Python
    ports:
      - 8080:80

  front_dotnet:
    image: front_taller
    container_name: front_dotnet
    environment:
      - API_URL=10.6.101.100:5040 #IP contenedor Back (Mismo puerto)
      - PUBLISHER=C#
    ports:
      - 8081:80

  front_java:
    image: front_taller
    container_name: front_java
    environment:
      - API_URL=10.6.101.100:7070 #IP contenedor Back (Mismo puerto)
      - PUBLISHER=Java
    ports:
      - 8082:80
```

5. DOCUMENTACIÓN

5.1. Repositorio de Github

Enlace: https://github.com/JuanRoa785/Taller-RabbitMQ.git



5.2. Video Explicativo Proyecto

Enlace: https://www.youtube.com/watch?v=SD9hrMAeWh4

Mensajería Publish-Subscribe con RabbitMQ