Pontificia Universidad Javeriana

Entrega Presentación 1 - 2430

Arquitectura de Software

Integrantes:

Javier Alejandro Moyano Cipamocha

Juan Francisco Ramirez Escobar

Esteban Salazar Arbelaez

Presentado a:

Andrés Armando Sánchez

28/09/2024

Bogotá, Colombia

Patrón PUB/SUB

El **patrón Publish/Subscribe (PUB/SUB)** es un patrón arquitectónico en el cual los componentes se comunican principalmente a través de mensajes asíncronos, a veces denominados "eventos" o "temas". Los publicadores no tienen conocimiento de los suscriptores, y los suscriptores solo conocen los tipos de mensajes.

Los sistemas que usan el patrón de publicación-suscripción dependen de la invocación implícita; es decir, el componente que publica un mensaje no invoca directamente a ningún otro componente. Los componentes publican mensajes sobre uno o más eventos o temas, y otros componentes registran su interés en la publicación. En tiempo de ejecución, cuando se publica un mensaje, el bus de publicación-suscripción (o de eventos) notifica a todos los elementos que registraron interés en el evento o tema. De esta manera, la publicación del mensaje provoca una invocación implícita de (métodos en) otros componentes. El resultado es un acoplamiento débil entre los publicadores y los suscriptores.

El patrón de publicación-suscripción tiene tres tipos de elementos:

- Componente publicador. Envía (publica) mensajes.
- Componente suscriptor. Se suscribe y luego recibe los mensajes.
- Bus de eventos. Gestiona las suscripciones y el envío de mensajes como parte de la infraestructura en tiempo de ejecución.

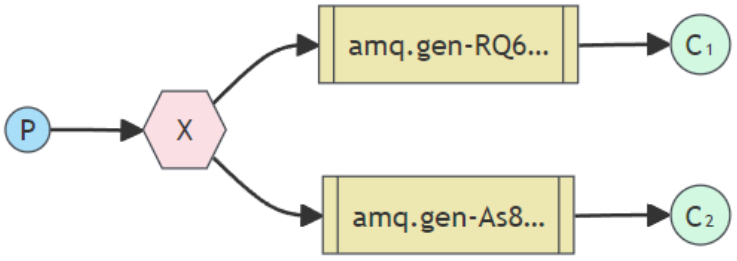

Imagen 1: Pub/Sub

Características:

- **Desacoplamiento:** Los *publishers* y *subscribers* no necesitan conocerse ni interactuar directamente, lo que facilita el diseño de sistemas distribuidos y escalables.
- **Escalabilidad:** Al distribuir la comunicación entre diferentes componentes, este patrón permite que los sistemas crezcan de manera eficiente sin generar cuellos de botella.
- **Asincronía:** Los mensajes se envían y reciben de manera asincrónica, lo que significa que las aplicaciones pueden continuar funcionando sin tener que esperar respuestas inmediatas.
- **Flexibilidad:** Es posible añadir o eliminar suscriptores sin afectar a los emisores, lo que facilita la evolución y adaptación del sistema a nuevas necesidades.
- **Tolerancia a fallos:** Dependiendo de la implementación, los mensajes pueden ser almacenados temporalmente si los suscriptores no están disponibles, garantizando la entrega eventual**.**

Historia:

El patrón Publish/Subscribe (PUB/SUB) ha sido fundamental para la comunicación en sistemas distribuidos, permitiendo una separación clara entre los emisores y receptores de mensajes. A lo largo del tiempo, su implementación ha sido clave en sistemas orientados a eventos y arquitecturas que requieren escalabilidad y manejo de notificaciones en tiempo real.

- **Década de 1990:** Surge como una solución para la comunicación asincrónica en sistemas distribuidos, inicialmente adoptado en sistemas como CORBA y plataformas de eventos.
- **2000s:** El patrón comienza a ser utilizado en arquitecturas orientadas a servicios y gana popularidad con el aumento de los sistemas de mensajería en tiempo real.
- **2010s en adelante:** Con el crecimiento del cloud computing y microservicios, PUB/SUB se consolida como un patrón clave en sistemas distribuidos, con soluciones modernas como Apache Kafka, RabbitMQ y Google Pub/Sub.

Ventajas:

- **Desacoplamiento**: Facilita la evolución independiente de los componentes al eliminar la dependencia directa entre emisores y receptores.
- **Escalabilidad**: Al permitir la distribución de mensajes entre múltiples suscriptores, mejora la capacidad del sistema para manejar grandes volúmenes de mensajes.
- **Asincronía**: Elimina la necesidad de comunicación en tiempo real entre los participantes, lo que reduce la latencia y permite el procesamiento en segundo plano.
- **Modularidad**: Facilita la adición de nuevos componentes o servicios sin impactar en la lógica existente, ya que nuevos suscriptores pueden añadirse fácilmente sin modificar los *publishers*.
- **Tolerancia a fallos**: En implementaciones más avanzadas, el patrón PUB/SUB permite la reintención de mensajes o persistencia temporal, asegurando que los datos se entreguen incluso si el suscriptor está desconectado en ese momento.

Desventajas:

- **Complejidad en la gestión de mensajes**: Puede requerir una infraestructura adicional para asegurar la entrega de mensajes, especialmente en sistemas distribuidos y de alta disponibilidad.
- **Sobrecarga en sistemas pequeños**: En aplicaciones más simples o con bajo volumen de mensajes, la implementación de PUB/SUB puede ser innecesariamente compleja.
- **Complejidad de configuración**: Requiere una configuración y gestión más avanzada, especialmente cuando se utilizan intermediarios de mensajes o sistemas distribuidos.

Casos de Uso.

- **Sistemas de notificaciones en tiempo real**: Ideal para aplicaciones que necesitan enviar actualizaciones inmediatas a múltiples usuarios, como notificaciones push en aplicaciones móviles o alertas en aplicaciones web.
- **Sistemas de mensajería interna**: Usado en arquitecturas de microservicios, donde diferentes servicios se comunican de manera asincrónica sin bloquearse mutuamente.
- **Actualización de información en tiempo real**: Plataformas de deportes en tiempo real (como tu proyecto FutScores), donde se necesitan actualizaciones instantáneas para mostrar datos en vivo como marcadores de partidos o estadísticas de jugadores.
- **Sistemas de eventos distribuidos**: Implementaciones de sistemas distribuidos que necesitan procesar eventos, como sensores IoT o plataformas de monitoreo.

RabbitMQ

Es un sistema de mensajería que utiliza el protocolo AMQP (Advanced Message Queuing Protocol) para facilitar la integración entre diferentes aplicaciones y servicios. Actúa como un intermediario que recibe, almacena y envía mensajes, permitiendo que los sistemas intercambien información de manera confiable. Además, RabbitMQ soporta colas persistentes, lo que asegura que los mensajes no se pierdan en caso de fallos, y permite la distribución de carga entre múltiples consumidores.

Imagen 3: RabbitMQ Logo

Modelos de Comunicación

RabbitMQ soporta varios modelos de comunicación que permiten flexibilidad y eficiencia en la distribución de mensajes entre diferentes componentes del sistema:

- **Publish/Subscribe**: Permite a los productores enviar mensajes sin preocuparse por los consumidores específicos. Los mensajes se publican en un "exchange" y son distribuidos a todas las colas que estén suscritas a dicho "exchange". Este modelo es ideal para notificaciones o eventos en los que varios receptores deben recibir la misma información simultáneamente.
- **Work Queues**: Este modelo está diseñado para distribuir tareas entre varios trabajadores. El objetivo es repartir la carga de trabajo de manera equitativa, asegurando que cada tarea sea procesada por un solo consumidor. Es útil en sistemas donde es necesario realizar procesamiento intensivo en segundo plano, como generación de informes o procesamiento de imágenes.
- **Routing**: RabbitMQ permite el enrutamiento de mensajes a través de exchanges que distribuyen los mensajes según reglas específicas basadas en claves de enrutamiento (routing keys). Esto permite que los mensajes lleguen solo a aquellos consumidores

interesados en ciertas categorías de mensajes, permitiendo una segmentación eficiente de la información.

- **Topics**: Este modelo extiende el enrutamiento basado en claves de enrutamiento, permitiendo una mayor flexibilidad al permitir el uso de patrones de coincidencia. Los mensajes se distribuyen según claves jerárquicas que permiten que múltiples consumidores reciban solo los mensajes que coinciden con los temas (topics) de interés.
- **RPC (Remote Procedure Call)**: RabbitMQ también permite la implementación de patrones RPC, donde los clientes envían solicitudes y esperan respuestas. Esto permite la comunicación sincrónica entre servicios, útil en sistemas que necesitan respuesta inmediata a peticiones específicas.

Características.

- **Soporte para múltiples protocolos:** Como se mencionó anteriormente, aunque RabbitMQ se basa en AMQP, también soporta otros protocolos como MQTT y STOMP, lo que lo hace versátil para diferentes escenarios.
- **Persistencia de mensajes:** Los mensajes pueden ser almacenados en disco, asegurando su entrega incluso en caso de fallos del sistema o interrupciones.
- **Balanceo de carga:** RabbitMQ distribuye los mensajes entre varios consumidores, equilibrando la carga de trabajo y optimizando el rendimiento.
- **Enrutamiento avanzado:** Permite definir reglas complejas para la entrega de mensajes, asegurando que los mensajes lleguen a los consumidores adecuados.

Historia.

Su implementación del protocolo AMQP ha permitido que diversas aplicaciones y servicios interactúen de manera eficiente a través de mensajes distribuidos, desempeñando un papel crucial en arquitecturas de microservicios.

- **2007**: Es lanzado como un sistema de mensajería basado en el protocolo AMQP por Rabbit Technologies, enfocado en facilitar la comunicación entre aplicaciones.
- **2010**: VMware adquiere Rabbit Technologies, y RabbitMQ pasa a formar parte del portafolio de soluciones empresariales de la compañía, ganando estabilidad y nuevas funcionalidades.
- **2015 en adelante**: Se convierte en una pieza clave en sistemas distribuidos y microservicios, con soporte para múltiples protocolos y

Ventajas:

- **Fiabilidad**: RabbitMQ es conocido por su capacidad para manejar grandes volúmenes de mensajes de manera confiable, con colas persistentes que aseguran la entrega de mensajes incluso en caso de fallos.
- **Durabilidad de mensajes**: RabbitMQ permite configurar colas persistentes, lo que asegura que los mensajes no se pierdan incluso si el servidor se reinicia.
- **Enrutamiento avanzado**: Permite definir reglas sofisticadas para el enrutamiento de mens ajes, lo que facilita su uso en sistemas complejos.

Desventajas:

- **Sobrecarga operativa**: La administración de RabbitMQ puede requerir conocimientos especializados, ya que la configuración, el monitoreo y el ajuste del rendimiento pueden ser complejos.
- **Escalabilidad limitada**: Aunque RabbitMQ es confiable, no escala tan bien como otras soluciones de mensajería más modernas como Apache Kafka.
- **Consumo de recursos**: RabbitMQ puede ser intensivo en términos de recursos de memoria y procesamiento cuando maneja un gran volumen de mensajes.
- **Escalabilidad limitada en grandes volúmenes**: Aunque RabbitMQ es excelente para un volumen moderado de mensajes, puede no ser la mejor opción para escenarios que requieren una capacidad masiva de mensajes por segundo, donde Kafka puede ser una mejor alternativa.
- **Dependencia de almacenamiento en disco**: En algunos casos, las colas persistentes de RabbitMQ pueden sufrir una ralentización debido a la dependencia de operaciones de disco.

Casos de Uso.

- **Integración de microservicios**: En arquitecturas de microservicios, RabbitMQ es ideal para coordinar la comunicación entre diferentes servicios de manera desacoplada y eficiente, permitiendo que cada microservicio se comunique mediante mensajes.
- **Sistemas de procesamiento por lotes**: RabbitMQ es útil en sistemas que necesitan procesar grandes volúmenes de trabajo en lotes, distribuyendo las tareas entre varios consumidores para balancear la carga.
- **Colas de tareas**: En aplicaciones web que requieren la ejecución de tareas en segundo plano, como el procesamiento de imágenes o la generación de informes, RabbitMQ permite manejar estas tareas sin sobrecargar el sistema principal.

Astro JS

Astro es un framework moderno para el desarrollo web, diseñado específicamente para optimizar la generación de sitios estáticos y dinámicos. Su principal ventaja es la capacidad de generar HTML estático por defecto, lo que resulta en tiempos de carga más rápidos. Además, Astro solo carga JavaScript cuando es estrictamente necesario, lo que lo hace ideal para sitios web que prioriza el rendimiento y la eficiencia, mientras sigue siendo compatible con tecnologías populares como React y Vue

Imagen 2: Astro Logo

Características.

- **Optimización de rendimiento:** Genera HTML estático, reduciendo la cantidad de recursos que necesita el navegador, lo que mejora considerablemente los tiempos de carga.
- **Carga de JavaScript bajo demanda:** Astro solo incluye el código JavaScript necesario cuando el usuario lo requiere, minimizando el tamaño total de las páginas.
- **Compatibilidad con múltiples frameworks:** Permite la integración de componentes de React, Vue y otros frameworks populares dentro de un mismo proyecto, lo que le da flexibilidad en términos de desarrollo.
- **Desarrollo modular con el enfoque de Islas:** Una de las características más innovadoras de Astro es su enfoque en las "islas". El concepto de Islas de Interactividad implica que las partes interactivas de una página (por ejemplo, formularios, menús dinámicos, o cualquier componente que requiera JavaScript) se aíslen en islas independientes. Cada isla se carga y ejecuta solo cuando es necesario, mientras que el resto del contenido se sirve como HTML estático.

  Este enfoque proporciona dos grandes beneficios:

  - Minimización de JavaScript innecesario: Solo se ejecuta JavaScript en aquellas áreas que realmente lo requieren, lo que mejora considerablemente el rendimiento general de la página.

    - Facilidad de mantenimiento y escalabilidad: Al aislar cada componente interactivo, los desarrolladores pueden trabajar de forma modular, permitiendo que el código sea más fácil de mantener y escalar. Cada isla se desarrolla y despliega de forma independiente sin afectar otras partes del sitio.

Historia.

Astro es un framework que nació como respuesta a los problemas de rendimiento de sitios web modernos, optimizando la forma en que se carga el contenido estático y dinámico.

- **2021**: Es lanzado como un framework de desarrollo web por los creadores de Snowpack, con el objetivo de optimizar sitios web estáticos y dinámicos, minimizando la carga de JavaScript.
- **2022**: Astro gana tracción rápidamente debido a su enfoque en rendimiento, y se integra con populares frameworks de JavaScript como React, Vue y Svelte, expandiendo sus capacidades y popularidad en la comunidad de desarrolladores.
- **2023 en adelante**: Astro continúa evolucionando, añadiendo soporte para funcionalidades avanzadas como la renderización dinámica y consolidándose como una opción líder en desarrollo web optimizado.

Ventajas:

- **Optimización de rendimiento**: Astro permite generar sitios altamente optimizados, mejorando los tiempos de carga al cargar solo el JavaScript necesario.
- **Compatibilidad con múltiples frameworks**: Soporta la integración con componentes de frameworks populares como React, Vue y Svelte.
- **Modularidad**: Facilita la creación de sitios modulares y escalables, donde los componentes están aislados y pueden desarrollarse de forma independiente.
- **Carga mínima de JavaScript**: Al ser un framework optimizado para HTML estático, Astro minimiza la carga innecesaria de JavaScript, lo que mejora el rendimiento y reduce el consumo de datos, especialmente en dispositivos móviles.

Desventajas.

- **Relativa inmadurez**: Al ser un framework relativamente nuevo, aún no tiene la madurez ni la comunidad de soporte de otros frameworks más establecidos.
- **Curva de aprendizaje**: Para desarrolladores acostumbrados a otros frameworks de JavaScript, adaptarse a las particularidades de Astro puede requerir algo de tiempo.
- **Comunidad pequeña**: Comparado con frameworks como React o Vue, Astro tiene una comunidad más pequeña, lo que puede limitar el acceso a tutoriales, documentación detallada o soporte.
- **Compatibilidad con herramientas de terceros**: Aunque Astro soporta varios frameworks populares, la integración con otras herramientas o bibliotecas de terceros podría requerir configuraciones adicionales.

Casos de Uso.

- **Sitios web de contenido estático**: Ideal para blogs, landing pages o sitios corporativos donde la mayor parte del contenido es estático y no cambia frecuentemente.
- **E-commerce optimizado**: En tiendas online con productos que no requieren actualizaciones constantes, Astro permite una experiencia rápida para los usuarios, con tiempos de carga mínimos.
- **Sitios web con carga selectiva de JavaScript**: Para aplicaciones web que tienen algunas secciones interactivas pero cuyo contenido principal puede ser servido estáticamente, como sitios informativos con formularios de contacto o áreas de login.

Redis

Es una base de datos en memoria que ofrece un rendimiento extremadamente rápido para el almacenamiento y recuperación de datos. Funciona principalmente bajo un esquema clave-valor, lo que lo hace muy útil en aplicaciones que requieren accesos rápidos a datos temporales o frecuentemente solicitados, como sesiones de usuario, sistemas de caché o colas de mensajes. Redis también incluye soporte para operaciones complejas y tipos de datos avanzados, lo que lo convierte en una herramienta versátil.

Imagen 4: Redis Logo

Características:

- **Almacenamiento en memoria:** Al operar principalmente en memoria, Redis ofrece tiempos de respuesta extremadamente rápidos, lo que lo hace ideal para sistemas que requieren acceso en tiempo real.
- **Soporte para estructuras de datos avanzadas:** A diferencia de otras bases de datos, Redis permite trabajar con listas, conjuntos, hashes y sorted sets, ampliando sus posibilidades de uso.
- **Soporte para operaciones atómicas:** Las operaciones sobre los datos son atómicas, lo que asegura la coherencia en entornos con múltiples operaciones simultáneas.
- **Capacidad para replicación y clustering:** Redis soporta la replicación de datos y la creación de clústeres distribuidos, permitiendo una alta disponibilidad y escalabilidad.

Historia.

Desde su creación, ha evolucionado rápidamente, proporcionando soluciones tanto para almacenamiento rápido como para operaciones complejas en memoria.

- **2009**: Salvatore Sanfilippo crea Redis para mejorar la escalabilidad de su aplicación de análisis de tráfico, comenzando como una base de datos clave-valor en memoria.
- **2011-2015**: Redis evoluciona rápidamente con la adición de soporte para nuevos tipos de datos (listas, conjuntos, hashes), persistencia en disco y replicación, lo que lo

convierte en una herramienta indispensable para la gestión de datos temporales y en tiempo real.

- **2016 en adelante**: Redis se establece como una base de datos crítica para aplicaciones de alto rendimiento, con capacidades de clustering y soluciones avanzadas de caching, siendo utilizado por gigantes tecnológicos como Twitter, GitHub y Snapchat.

Ventajas:

- **Rendimiento extremadamente rápido**: Al operar en memoria, Redis ofrece tiempos de acceso y escritura muy rápidos, lo que lo hace ideal para aplicaciones en tiempo real.
- **Versatilidad**: Redis soporta varios tipos de datos y puede usarse para múltiples casos de uso como almacenamiento en caché, gestión de sesiones, y colas de mensajes.
- **Simplicidad**: Su naturaleza de base de datos clave-valor lo convierte en una solución fácil de implementar y administrar en muchos escenarios.

Desventajas:

- **Persistencia limitada**: Aunque Redis puede persistir datos en disco, su principal enfoque es el almacenamiento en memoria, lo que puede resultar en pérdida de datos en caso de un fallo del sistema.
- **Consumo de memoria**: Al ser una base de datos en memoria, su uso intensivo de memoria RAM puede limitar su escalabilidad en sistemas con grandes volúmenes de datos.
- **Escalabilidad limitada sin clustering**: Si no se utiliza Redis en modo cluster, su escalabilidad está limitada por la cantidad de memoria disponible en un solo nodo.
- **Costos de hardware para grandes volúmenes**: Si Redis almacena grandes volúmenes de datos en memoria, los costos de hardware pueden aumentar rápidamente, especialmente en entornos donde se maneja una gran cantidad de datos.

Casos de Uso.

- **Almacenamiento en caché**: Redis es excelente para almacenar en caché datos que se consultan frecuentemente, como resultados de consultas a bases de datos, para mejorar el rendimiento y reducir la carga en los servidores.

- **Gestión de sesiones**: En aplicaciones web, Redis puede utilizarse para almacenar información de sesión del usuario, manteniendo las sesiones activas entre múltiples servidores o solicitudes sin recurrir a una base de datos más lenta.
- **Colas de mensajes simples**: Redis puede actuar como una cola de mensajes ligera para tareas de procesamiento en segundo plano que necesitan ser ejecutadas rápidamente.

Go (GoLang)

Go es un lenguaje de programación desarrollado por Google que se destaca por su simplicidad y eficiencia, especialmente en sistemas que requieren un alto grado de concurrencia. Go combina la velocidad de un lenguaje compilado con una sintaxis clara y fácil de entender, lo que facilita la creación de aplicaciones escalables y de alto rendimiento. Su capacidad para gestionar tareas concurrentes de manera eficiente lo hace especialmente útil en el desarrollo de servicios backend y aplicaciones distribuidas.

Imagen 5: GoLang Logo

Características:

- **Soporte nativo para concurrencia:** Go tiene un sistema de concurrencia integrado basado en goroutines, lo que facilita la creación de aplicaciones que manejan múltiples tareas simultáneamente sin complicaciones adicionales.
- **Rendimiento elevado:** Al ser un lenguaje compilado, Go ofrece un rendimiento comparable al de lenguajes como C o C++, pero con una sintaxis más simple y moderna.
- **Simplicidad en la sintaxis:** Ha sido diseñado para ser fácil de aprender y utilizar, con una sintaxis clara que evita la complejidad innecesaria, lo que permite un desarrollo rápido y eficiente.
- **Compatibilidad cruzada:** Puede compilarse para diferentes plataformas sin necesidad de modificar el código fuente, lo que facilita el despliegue en diversos entornos.
- **Recolección automática de basura:** Go incluye un sistema de *garbage collection* que gestiona la memoria de manera eficiente, reduciendo la posibilidad de errores asociados a la gestión manual de la memoria.

Historia.

Su enfoque en la concurrencia y el alto rendimiento ha hecho que se convierta en una herramienta clave en el desarrollo de backend y plataformas de infraestructura.

- **2009**: Google lanza Go, desarrollado por ingenieros de renombre como Robert Griesemer, Rob Pike y Ken Thompson, con el objetivo de mejorar la gestión de concurrencia y simplificar el desarrollo de aplicaciones escalables.
- **2013**: Go alcanza la versión 1.0, marcando su adopción formal en proyectos de producción, destacando por su capacidad para manejar tareas concurrentes con goroutines.
- **2014-2017**: Se convierte en el lenguaje preferido para desarrollar sistemas como Docker y Kubernetes, consolidándose como una opción fundamental para el desarrollo de backend y cloud computing.
- **2020 en adelante**: Go sigue evolucionando, con mejoras en su manejo de memoria y herramientas de desarrollo, manteniéndose como una elección sólida para aplicaciones distribuidas, escalables y de alto rendimiento.

Ventajas:

- **Concurrencia eficiente**: Go tiene un excelente manejo de la concurrencia gracias a las goroutines, lo que permite ejecutar múltiples tareas simultáneamente sin complicaciones.
- **Rendimiento**: Como lenguaje compilado, Go genera binarios que son muy rápidos y eficientes, lo que lo hace ideal para aplicaciones de backend y servicios que requieren alta capacidad de procesamiento.
- **Simplicidad**: Go tiene una sintaxis simple y directa, lo que facilita su aprendizaje y su implementación en proyectos complejos.

Desventajas:

- **Ecosistema relativamente joven**: Aunque Go ha crecido mucho en popularidad, su ecosistema y algunas bibliotecas aún están en desarrollo en comparación con lenguajes más establecidos como Java o Python.
- **Manejo limitado de bibliotecas de terceros**: El soporte para algunas bibliotecas y herramientas aún puede ser limitado, especialmente en comparación con lenguajes que tienen décadas de desarrollo.

- **Sistema de manejo de dependencias**: Aunque ha mejorado con herramientas como Go Modules, el manejo de dependencias en Go ha sido históricamente un área de críticas y dificultades para los desarrolladores.

Casos de Uso.

- **APIs y servicios backend de alto rendimiento**: Go es ideal para construir APIs REST o servicios backend que necesitan manejar grandes volúmenes de tráfico con una baja latencia, como aplicaciones en la nube o plataformas de streaming.
- **Aplicaciones concurrentes**: Aplicaciones que necesitan manejar múltiples procesos simultáneamente, como servidores de chat, sistemas de procesamiento de imágenes o servicios que manejan grandes volúmenes de datos en tiempo real.
- **Microservicios**: Gracias a su eficiencia y capacidad para manejar múltiples tareas concurrentes, Go es una excelente opción para desarrollar microservicios escalables que se comunican mediante HTTP o colas de mensajes.

¿Qué tan común es el stack designado?

El stack tecnológico que se ha elegido para el proyecto **FutScores** es altamente común en la industria. Este stack es especialmente adecuado para aplicaciones que necesitan responder a múltiples solicitudes en tiempo real y manejar grandes volúmenes de datos y comunicaciones entre componentes distribuidos. Aunque algunas tecnologías como Astro son relativamente nuevas, su adopción está en crecimiento, lo que lo hace un stack moderno y orientado al futuro.

1. Patrón PUB/SUB

Gracias a su capacidad para manejar gran escala, el patrón PUB/SUB es común en microservicios y aplicaciones que se ejecutan en la nube, como en AWS, Google Cloud Pub/Sub y Azure Event Grid.

- **WhatsApp**: Utiliza el patrón PUB/SUB para gestionar las notificaciones de mensajes entre los usuarios, permitiendo que múltiples dispositivos reciban notificaciones al mismo tiempo cuando llega un nuevo mensaje.
- **Slack**: En Slack, el patrón PUB/SUB se utiliza para distribuir mensajes en canales y notificaciones entre diferentes usuarios y dispositivos, asegurando la entrega rápida y eficiente.

2. Astro JS

Comparado con frameworks más establecidos como React o Vue, Astro todavía está en una fase de crecimiento, pero su uso en proyectos de rendimiento crítico está aumentando.

- **Skyscanner**: Un popular sitio web de búsqueda de vuelos que utiliza Astro para optimizar el rendimiento de su landing page, asegurando tiempos de carga rápidos y una mejor experiencia de usuario.
- **Ecommerce y sitios corporativos**: Empresas de ecommerce de bajo volumen y sitios corporativos han implementado Astro para mejorar la experiencia de usuario, especialmente en dispositivos móviles, con la carga mínima de JavaScript.

3. RabbitMQ

RabbitMQ es una de las soluciones de mensajería más comunes en el mercado. Es ampliamente adoptado en arquitecturas de microservicios y sistemas distribuidos para manejar la comunicación entre aplicaciones y servicios.

- **Instagram**: Utiliza RabbitMQ para gestionar la cola de notificaciones de los usuarios cuando reciben un comentario o un like en una foto. Esto permite que la aplicación procese grandes volúmenes de notificaciones en tiempo real.
- **Pinterest**: Usa RabbitMQ para distribuir eventos de "pinning" a través de diferentes servidores, lo que permite gestionar las actualizaciones de contenido de los usuarios y sus seguidores.

4. Redis

Redis es extremadamente común en aplicaciones que requieren un acceso rápido a datos temporales o de alta frecuencia, siendo utilizado para almacenamiento en caché, gestión de sesiones, y colas de trabajo en tiempo real.

- **Twitter**: Twitter utiliza Redis para gestionar el almacenamiento en caché de las timelines de los usuarios, mejorando el rendimiento en la entrega de actualizaciones en tiempo real.
- **GitHub**: Redis se usa en GitHub para gestionar la cola de tareas de procesamiento en segundo plano, como la compilación de repositorios y la entrega de notificaciones a los usuarios.
- **Snapchat**: Usa Redis para manejar la gestión de sesiones y el almacenamiento en caché de datos temporales, lo que mejora la velocidad de carga de las interacciones en la aplicación.

5. GoLang

Go es un lenguaje de programación muy popular en la actualidad, especialmente en el desarrollo de sistemas backend, microservicios, y plataformas de infraestructura en la nube. Su adopción ha crecido significativamente debido a su simplicidad y eficiencia.

- **Uber**: Uber utiliza Go en varias de sus microservicios backend, particularmente aquellos que requieren alta concurrencia y baja latencia para manejar el tráfico en tiempo real de su plataforma de transporte.
- **SoundCloud**: La plataforma de streaming SoundCloud eligió Go para construir APIs que soportan gran cantidad de usuarios y reproducen música en tiempo real, garantizando un rendimiento óptimo.

I. Matriz de análisis de Principios SOLID vs Temas

Tecnología	SRP (Single Responsibility Principle)	OCP (Open/Closed Principle)	LSP (Liskov Substitution Principle)	ISP (Interface Segregation Principle)	DIP (Dependency Inversion Principle)
Patron PUB/SUB	Cada componente tiene una función clara (emisor o receptor).	El patrón es extensible, puedes agregar nuevos suscriptores sin modificar la lógica	Se puede mantener la lógica si los suscriptores siguen el contrato del sistema de mensajería.	No aplica directamente, ya que no hay interfaces estrictas.	El emisor y receptor dependen de abstracciones como los canales de mensajería.
Astro JS	Cada componente de Astro sigue el SRP	Astro permite extender las funcionalidades del sitio web sin modificar el código base	No aplica	Permite crear interfaces seguras para cada componente del sitio.	Depende de la integración con otros frameworks, puede implementar DIP con la infraestructura adecuada.
RabbitMQ	Cada cola y mensaje tiene una responsabilidad clara dentro del sistema.	RabbitMQ permite agregar más consumidores sin modificar los productores	Siempre que los consumidores sigan el protocolo de mensajes, se puede sustituir sin problemas.	En sistemas que dependen de RabbitMQ, las interfaces se crean de manera natural al conectar con los servicios.	Los servicios dependen del broker y no directamente de otros servicios.
Redis	Redis sigue el SRP al ser usado principalmente para almacenamiento en caché o datos temporales.	Permite añadir nuevas características o cambiar la estructura sin afectar su comportamiento general.	No aplica	Redis es simple, sin embargo, las interfaces de acceso pueden ser encapsuladas para cumplir con ISP.	Redis sigue el DIP en la mayoría de las arquitecturas donde se integra, ya que los módulos de aplicación dependen de abstracciones.
GoLang	Go promueve la creación de módulos simples con responsabilidades claras	Permite crear software extensible mediante paquetes modulares sin alterar el código existente.	La composición es preferida sobre la herencia	Go fomenta la creación de interfaces pequeñas y específicas	Enfoque en la dependencia de interfaces y abstracciones

El análisis del stack revela una clara alineación con los principios SOLID, lo que nos garantiza un diseño arquitectónico escalable, flexible y mantenible. Tecnologías clave como RabbitMQ, Redis y el patrón PUB/SUB cumplen con la separación de responsabilidades y permiten la extensibilidad del sistema sin necesidad de modificar su estructura base

II. Matriz de análisis de Atributos de Calidad vs Temas

Tecnología	Rendimiento	Escalabilidad	Mantenibilidad	Seguridad	Disponibilidad
Patron PUB/SUB	Permite una alta eficiencia en la entrega de mensajes, ya que los emisores no necesitan esperar a los suscriptores.	Escalable debido a la capacidad de agregar más suscriptores sin afectar a los publishers.	Fácil de mantener dado que los componentes están desacoplados	Los mensajes pueden estar expuestos si no se asegura el canal.	Alta, ya que los mensajes pueden reenviarse cuando los suscriptores están disponibles.
Astro JS	Ofrece un rendimiento optimizado al generar HTML estático y reducir la carga de JavaScript.	Escalable en términos de tráfico web, pero limitado en cuanto a aplicaciones altamente dinámicas.	Mantenible gracias a su modularidad	Seguridad gestionada por el front-end, depende de las medidas aplicadas en el backend.	Alta disponibilidad, ya que los sitios estáticos son menos propensos a caídas.
RabbitMQ	Excelente rendimiento en el manejo de grandes volúmenes de mensajes en tiempo real.	Altamente escalable mediante la distribución de colas	Mantenimiento moderado, requiere conocimientos específicos	Ofrece mecanismos de seguridad como TLS y autenticación, pero puede requerir configuraciones avanzadas.	Alta, al permitir replicación y distribución de colas.
Redis	Extremadamente rápido gracias a su almacenamiento en memoria.	Escalable, especialmente para manejar grandes volúmenes de datos.	Fácil de mantener, con una sintaxis simple	Necesita medidas adicionales para asegurar los datos en memoria.	Alta, ya que puede replicar datos entre nodos para evitar fallos.
GoLang	Rendimiento elevado, especialmente en sistemas concurrentes y de alto tráfico.	Escalable debido a su manejo eficiente de la concurrencia	Mantenibilidad alta debido a su simplicidad	Herramientas de seguridad sólidas para manejar conexiones seguras	Alta, gracias a su capacidad de manejo de múltiples peticiones concurrentes

Cada tecnología del stack contribuye de manera efectiva a diferentes atributos de calidad, asegurando que el sistema del proyecto sea eficiente, escalable y seguro.

III. Matriz de análisis de Tácticas vs Temas

Tecnología	Manejo de Concurrencia	Redundancia	Particionamiento	Balanceo de carg	Caché
Patron PUB/SUB	Soporta múltiples mensajes concurrentes entre productores y consumidores.	Los mensajes pueden reenviarse a suscriptores cuando estén disponibles	Los mensajes se pueden particionar en diferentes temas o canales para mejorar la organización.	No maneja balanceo de carga directamente	No aplica directamente en el patrón PUB/SUB.
Astro JS	No está diseñado específicamente para manejar concurrencia, ya que es un framework orientado al front-end.	Pueden tener redundancia natural en servidores distribuidos	No aplica, aunque puede segmentar contenido por páginas.	El contenido estático puede distribuirse a través de CDNs para balancear la carga.	No aplica directamente, pero el contenido estático no requiere caché adicional.
RabbitMQ	Permite manejar múltiples consumidores concurrentes.	Ofrece redundancia mediante la replicación de colas y mensajes en diferentes nodos.	Las colas pueden ser particionadas en diferentes nodos o clusters.	RabbitMQ distribuye automáticamente la carga entre varios consumidores.	No aplica directamente
Redis	Soporta operaciones concurrentes con múltiples accesos a la base de datos en memoria.	Ofrece replicación de datos, proporcionando redundancia en caso de fallos.	Permite el particionamiento a través de clústeres, distribuyendo datos en múltiples nodos.	No aplica directamente, pero se puede combinar con otros sistemas para balancear el acceso a datos.	Es uno de los principales sistemas de caché en memoria, reduciendo la carga en las bases de datos.
GoLang	Tiene un soporte nativo para concurrencia a través de goroutines, lo que permite manejar múltiples tareas al mismo tiempo.	No implementa redundancia por sí solo, pero puede ser parte de una arquitectura que la soporte.	Go es ideal para sistemas distribuidos y puede colaborar en la partición de servicios en microservicios.	Puede implementar fácilmente balanceadores de carga dentro de aplicaciones distribuidas.	Go se integra bien con sistemas de caché como Redis para optimizar el rendimiento.

Demuestra que el stack de tecnologías son muy eficaces en el manejo de tareas concurrentes y la distribución de cargas, asegurando que el sistema pueda procesar múltiples solicitudes de manera eficiente. La combinación de estas tecnologías permite construir una arquitectura escalable, resiliente y de alto rendimiento, capaz de manejar grandes volúmenes de datos y usuarios sin comprometer la estabilidad ni la velocidad.

IV. Matriz de análisis de Patrones vs Temas

Tecnología	Patrón de Diseño en Capas	Patrón MVC	Patrón de Microservicios	Arquitectura Hexagonal
Astro JS	Encaja principalmente en la capa de presentación dentro de una arquitectura en capas, proporcionando una interfaz rápida y optimizada.	Astro JS puede actuar como la vista en un sistema MVC, gestionando la presentación de datos al usuario de manera eficiente.	No se relaciona directamente con microservicios, pero puede consumir APIs backend basadas en microservicios.	No es directamente aplicable en la arquitectura hexagonal, ya que está enfocado en la presentación.
RabbitMQ	Puede formar parte de una arquitectura en capas, actuando como middleware para la capa de comunicación entre diferentes capas de la aplicación.	Aunque no está directamente relacionado con MVC, RabbitMQ puede gestionar la comunicación entre el controlador y otras capas de la arquitectura.	Es una herramienta comúnmente usada en arquitecturas de microservicios, facilitando la mensajería entre servicios distribuidos.	Puede integrarse en una arquitectura hexagonal para gestionar la comunicación entre los puertos (interfaces) y adaptadores externos.
Redis	Actúa como un mecanismo de soporte para la capa de datos, mejorando el rendimiento del sistema en un diseño en capas.	Puede servir de soporte al modelo en un sistema MVC, almacenando datos en memoria para mejorar la rapidez en la interacción con el modelo.	Se usa comúnmente en microservicios como un almacenamiento en caché compartido o una base de datos en memoria.	Redis puede integrarse en la capa de infraestructura de una arquitectura hexagonal para proporcionar almacenamiento rápido de datos.
Go (GoLang)	Go es ideal para implementar la lógica de negocio en la capa intermedia de una arquitectura en capas, siendo eficiente y escalable.	Go puede ser utilizado para implementar el controlador y el modelo en un sistema MVC, manejando la lógica de negocio y la interacción con la base de datos.	Go es ampliamente utilizado en microservicios debido a su rendimiento y capacidad para manejar múltiples servicios concurrentes.	Go se adapta bien a la arquitectura hexagonal, actuando como la capa de dominio o aplicación, separando la lógica del sistema de las dependencias externas.

Las tecnologías del stack pueden integrarse en diversas capas del sistema, permitiendo una separación clara de responsabilidades y garantizando una comunicación fluida entre componentes. Además, apoyan arquitecturas que favorecen la mensajería asíncrona, el almacenamiento en caché, la segregación de comandos y consultas, y la integración de servicios desacoplados.

V. Matriz de análisis de Mercado Laboral vs Temas

Tecnología	Demanda de Empleo	Oportunidades de crecimiento profesional
Patron PUB/SUB	Alta, especialmente en sistemas distribuidos, IoT, y aplicaciones en tiempo real.	Oportunidades significativas para ingenieros de backend y desarrolladores especializados en sistemas distribuidos.
Astro JS	Moderada, debido a su reciente adopción, pero en aumento entre empresas que priorizan la optimización del rendimiento web.	Crecimiento moderado, con oportunidades en áreas de desarrollo web moderno y performance optimization.
RabbitMQ	Alta, con una demanda constante en empresas que utilizan arquitecturas de microservicios y sistemas de mensajería.	Oportunidades amplias para arquitectos de software y desarrolladores en sistemas distribuidos y mensajería.
Redis	Muy alta, Redis es ampliamente adoptado para el almacenamiento en caché y bases de datos en memoria.	Altas oportunidades en roles de backend, DevOps, y administración de bases de datos NoSQL.
GoLang	Muy alta, especialmente en el desarrollo backend y servicios en la nube.	Gran crecimiento profesional, con fuerte demanda de desarrolladores de Go en el desarrollo de microservicios, APIs y plataformas cloud.

Herramientas como RabbitMQ, Redis, y el patrón PUB/SUB son muy populares en sistemas distribuidos y microservicios, lo que ofrece muchas oportunidades para desarrolladores backend y arquitectos de sistemas. Go sigue creciendo en popularidad, siendo muy buscado para proyectos de alta concurrencia y escalabilidad en la nube. Aunque Astro JS es más nuevo, está ganando terreno rápidamente en el desarrollo web, especialmente en sitios que buscan mejorar el rendimiento

Referencias

I. Longo, E., & Redondi, A. E. (2023). Design and implementation of an advanced MQTT broker for distributed pub/sub scenarios. *Computer Networks*, *224*, 109601.

II. Jones, B., Luxenberg, S., McGrath, D., Trampert, P., & Weldon, J. (2011). RabbitMQ performance and scalability analysis. *project on CS*, *4284*.

III. Matallah, H., Belalem, G., & Bouamrane, K. (2020). Evaluation of nosql databases: Mongodb, cassandra, hbase, redis, couchbase, orientdb. *International Journal of Software Science and Computational Intelligence (IJSSCI)*, *12*(4), 71-91.

IV. Meyerson, J. (2014). The go programming language. *IEEE software*, *31*(5), 104-104

V. RabbitMQ. (n.d.). *RabbitMQ Documentation*. https://www.rabbitmq.com/documentation.html

VI. Astro. (n.d.). *Astro Framework Documentation*. https://docs.astro.build/