ARQUITECTURA DE SOFTWARE

TALLER 1 - PRESENTACIÓN

ANDRÉS DAVID PÉREZ CELY
DANIEL FERNANDO GONZALEZ CORTÉS
JUAN DIEGO REYES RODRIGUEZ

PROFESOR
ANDRES ARMANDO SANCHEZ MARTIN

BOGOTÁ D.C

Repositorio Git Tag:

https://github.com/talleres-arqui/presentacion-1/releases/tag/v1.0

Patrón publicador/suscriptor

- Definición

El patrón Publisher–Subscriber (publicador–suscriptor) es un patrón de diseño de arquitectura de software que permite la comunicación asíncrona y desacoplada entre componentes de un sistema. En este modelo, los publicadores (publishers) generan y envían mensajes sobre determinados temas (*topics*) sin conocer quién los recibirá, mientras que los suscriptores (subscribers) expresan interés en esos temas y reciben automáticamente las notificaciones cuando se publican nuevos eventos. La comunicación entre publicadores y suscriptores se realiza a través de un intermediario llamado broker o gestor de mensajes, que actúa como un canal central encargado de distribuir los mensajes.

- Características
  - Desacoplamiento total: Los emisores y receptores no se conocen entre sí; solo interactúan mediante temas (topics). Esto permite modificar, agregar o eliminar componentes sin afectar a los demás.
  - Comunicación asíncrona: Los mensajes no requieren respuesta inmediata. El suscriptor puede recibir datos en cualquier momento, incluso si el publicador ya terminó su tarea.
  - Basado en eventos: La información fluye como una secuencia de eventos, lo que facilita la reacción en tiempo real a cambios o actualizaciones.
  - Escalabilidad y flexibilidad: Permite integrar múltiples publicadores y suscriptores sin colisiones. Ideal para microservicios, IoT y sistemas distribuidos.
  - Fiabilidad y control: Los brokers (como MQTT) manejan mecanismos de entrega garantizada (QoS), persistencia y control de sesiones.
  - Multiprotocolo: Aunque el patrón es conceptual, puede implementarse con distintas tecnologías: MQTT, Redis Pub/Sub, RabbitMQ, Kafka, etc.
  - Aplicabilidad amplia: Utilizado en aplicaciones IoT, notificaciones en tiempo real, monitoreo, chat, y sistemas de mensajería.

- Rol del Broker en el Patrón Publisher–Subscriber

El broker (intermediario o gestor de mensajes) es el núcleo del patrón Pub/Sub. Su función es recibir, filtrar y distribuir los mensajes entre los publicadores (publishers) y los suscriptores (subscribers) según los topics o temas de interés.

Funciones principales del broker:

  - Recepción de mensajes: recibe las publicaciones que envían los publicadores.

- Gestión de tópicos: organiza los mensajes por temas (topics) para que solo los suscriptores interesados los reciban.
- Entrega garantizada: aplica reglas de calidad de servicio (QoS) para asegurar que los mensajes lleguen correctamente.
- Almacenamiento temporal: puede retener mensajes para suscriptores desconectados (según la configuración del protocolo).
- Seguridad: maneja autenticación, autorización y cifrado para evitar accesos no autorizados.
- Monitoreo y control: registra métricas sobre conexión, latencia y tráfico de mensajes.

- Niveles de Calidad de Servicio (QoS) en MQTT

El protocolo MQTT incluye tres niveles de Quality of Service (QoS) que determinan cómo se garantiza la entrega de los mensajes entre el publicador, el broker y el suscriptor. Esto permite ajustar el equilibrio entre rendimiento, confiabilidad y consumo de red según el caso de uso.

Nivel	**Nombre**	**Descripción**	**Uso recomendado**
QoS 0	At most once (como máximo una vez)	El mensaje se envía sin confirmación. Si se pierde, no se reintenta. Es el modo más rápido, pero menos confiable.	Telemetría no crítica, sensores frecuentes.
QoS 1	At least once (al menos una vez)	El publicador espera confirmación del broker. Puede llegar más de una vez si hay retransmisión.	Comunicación general o eventos donde duplicados no sean un problema.
QoS 2	Exactly once (exactamente una vez)	Intercambio de cuatro pasos entre publicador y broker para garantizar una única entrega sin duplicados. Es el más confiable pero también el más lento.	Transacciones críticas o datos financieros.

- Suscripción y temas (Topics)

Un topic es el canal lógico por el que se enrutan los mensajes en Pub/Sub (MQTT). Los publicadores envían mensajes a un topic y los suscriptores reciben solo los de los topics a los que están suscritos.

- Jerarquía y sintaxis

  - Los topics se organizan jerárquicamente con / como separador.
  - No hay registro previo: un topic existe cuando se publica o se suscribe.
  - Ejemplos:

    - `noticias/publicadas`
    - `noticias/categoria/politica`
    - `usuarios/123/actividad`

- Comodines (wildcards)

  - (un nivel): coincide con un segmento.
    - `noticias/categoria/+` → recibe `noticias/categoria/politica` y `noticias/categoria/deportes`.
  - (varios niveles): coincide con cero o más segmentos al final.
    - `noticias/#` → recibe todo bajo `noticias/…`
  - Reglas: los wildcards solo se usan al suscribirse, no al publicar.

- Historia y evolución

El patrón Publisher–Subscriber (Publicador–Suscriptor) surge de la necesidad de desacoplar emisores y receptores en sistemas distribuidos desde finales de los 70 e inicios de los 80, cuando los modelos cliente–servidor empezaron a mostrar límites de escalabilidad y flexibilidad. A diferencia de las llamadas directas, Pub/Sub propone comunicación asíncrona basada en eventos, donde los publicadores emiten mensajes sin conocer a los receptores, y los suscriptores reciben solo lo que les interesa (por *topics*).

Durante los 90, con el auge del middleware (CORBA, JMS) y la arquitectura dirigida por eventos (EDA), el patrón se formalizó: aparecieron brokers y colas como elementos estándar para enrutar mensajes. En los 2000, la expansión web y los servicios distribuidos empujaron implementaciones más ligeras (RabbitMQ/AMQP, Redis Pub/Sub). El salto masivo llegó con IoT, que exigió protocolos ultraeficientes y confiables; así se popularizaron MQTT y, para *event streaming* a gran escala, Kafka, consolidando Pub/Sub como pilar de microservicios y sistemas en tiempo real.

Línea de Evolución

- 1970s–1980s: primeras formulaciones de comunicación asíncrona y modelos de eventos para reducir acoplamiento entre módulos.
- 1990s: estandarización conceptual con JMS, CORBA y EDA; brokers y *topics* se vuelven patrones comunes.
- 2000s: adopción en web y SOA; aparecen RabbitMQ (AMQP) **y** Redis Pub/Sub para mensajería ligera.
- 2010–2015: auge de microservicios y cloud; Kafka impulsa *event streaming* de alto rendimiento; MQTT crece con IoT industrial y doméstico.
- 2016–2020: madurez del ecosistema; integraciones administradas en nubes (AWS SNS/SQS, Google Pub/Sub, Azure Event Grid); seguridad y observabilidad mejoran.

- 2021 en adelante: Pub/Sub se integra con arquitecturas reactivas, edge/IoT, WebSockets y serverless; prácticas de event-driven se vuelven estándar en data/IA y tiempo real.

Contexto actual

Hoy, Pub/Sub es la columna vertebral de sistemas distribuidos: facilita escalabilidad horizontal**,** resiliencia**,** integración entre servicios **y** tiempo real. Se implementa con brokers y servicios gestionados (MQTT/Mosquitto, Kafka, RabbitMQ, SNS/SQS, Google Pub/Sub) y convive con APIs REST/GraphQL. En tu arquitectura (FastAPI + Mosquitto + Vue), el patrón habilita notificaciones inmediatas, bajo acoplamiento y extensibilidad por *topics*, alineado con EDA y microservicios modernos.

- Ventajas y desventajas

Ventajas

- Desacoplamiento: Los publicadores y suscriptores no dependen entre sí; pueden evolucionar o cambiar sin afectar a los demás componentes del sistema.
- Escalabilidad: Permite agregar nuevos suscriptores o publicadores fácilmente, sin necesidad de modificar el flujo de mensajes existente.
- Comunicación asíncrona: Los componentes pueden comunicarse sin esperar respuesta inmediata, optimizando el uso de recursos y la eficiencia.
- Flexibilidad y extensibilidad: Es posible conectar distintos sistemas, tecnologías o plataformas a través del broker sin alterar la lógica del negocio.
- Tolerancia a fallos: Si un suscriptor no está disponible, el broker puede almacenar o reenviar mensajes (según el protocolo), garantizando la entrega.
- Soporte para tiempo real: Facilita la actualización inmediata de datos entre dispositivos o aplicaciones conectadas, ideal para IoT, notificaciones o sistemas de monitoreo.
- Mantenibilidad: La separación de responsabilidades facilita la depuración, el escalamiento y la evolución del sistema con menor riesgo de errores colaterales.

Desventajas

- Complejidad en la gestión: Requiere un broker o middleware adicional, lo que implica configuración, monitoreo y mantenimiento constante.
- Dificultad en el seguimiento de flujos: Al ser asíncrono, rastrear el recorrido de un mensaje o depurar errores puede ser más complejo.
- Riesgo de mensajes perdidos: Si no se configuran correctamente los niveles de calidad de servicio (QoS), algunos mensajes pueden no llegar al destino.
- Sobrecarga de red: En sistemas con muchos suscriptores o eventos, el tráfico de mensajes puede crecer rápidamente, afectando el rendimiento.
- Dependencia del broker: Si el broker falla y no hay redundancia, el sistema completo puede verse afectado.

- Seguridad: Al existir múltiples puntos de entrada y distribución de datos, se requieren mecanismos de autenticación y cifrado robustos.

- Casos de Uso (Situaciones y/o problemas donde se pueden aplicar)

  1. Notificaciones en tiempo real

     - Situación: En una plataforma de noticias o red social, los usuarios deben recibir nuevas publicaciones o comentarios sin necesidad de recargar la página.
     - Aplicación: El servidor (publicador) emite un evento cada vez que se genera contenido nuevo; los clientes (suscriptores) reciben automáticamente las actualizaciones mediante WebSockets o MQTT.

  2. Internet de las Cosas (IoT)

     - Situación: Una red de sensores distribuidos (por ejemplo, temperatura, humedad o movimiento) envía datos periódicamente a un sistema central.
     - Aplicación: Los sensores publican datos en tópicos específicos (por ejemplo, "sensores/temperatura"), mientras los sistemas de monitoreo o control se suscriben solo a los tópicos relevantes.

  3. Monitoreo y telemetría en sistemas distribuidos

     - Situación: Una infraestructura de microservicios necesita recolectar métricas de rendimiento, logs o alertas en tiempo real.
     - Aplicación: Cada servicio publica eventos sobre su estado o rendimiento, y los sistemas de observabilidad (como Prometheus o Grafana) se suscriben para recopilar la información.

  4. Aplicaciones financieras o de trading

     - Situación: En bolsas de valores o plataformas de criptomonedas, los precios y órdenes cambian constantemente y deben distribuirse a miles de clientes simultáneamente.
     - Aplicación: Los servicios de precios publican actualizaciones en tópicos (por ejemplo, "mercado/acciones/XYZ"), y los clientes suscritos reciben la información instantáneamente.

  5. Comunicación entre microservicios

     - Situación: Un sistema modular con múltiples microservicios requiere intercambiar eventos sin generar dependencias directas entre ellos.
     - Aplicación: Cada microservicio publica eventos cuando ocurre un cambio (por ejemplo, "usuario/creado"), y los demás servicios interesados se suscriben para reaccionar.

  6. Control industrial y automatización

- Situación: En una planta industrial, distintos equipos necesitan coordinar operaciones en tiempo real (como el encendido de motores, apertura de válvulas o medición de presión).
- Aplicación: Los controladores publican estados y comandos en canales MQTT, y los dispositivos se suscriben según su función.

7. Chats y mensajería instantánea

- Situación: En aplicaciones de mensajería, los usuarios deben recibir nuevos mensajes o actualizaciones de estado en tiempo real.
- Aplicación: Cada sala o conversación es un tópico al que los usuarios se suscriben, recibiendo los mensajes tan pronto son publicados.

- Casos de aplicación (Ejemplos y casos de éxito en la industria)

1. Netflix – Arquitectura de microservicios y mensajería interna

- Netflix utiliza el patrón Pub/Sub dentro de su arquitectura basada en microservicios para comunicar eventos entre componentes de forma asíncrona. Cada microservicio publica notificaciones sobre cambios de estado o métricas internas, mientras otros servicios se suscriben para reaccionar o almacenar datos en tiempo real.
- Resultado: Mayor resiliencia del sistema, reducción del acoplamiento y escalabilidad a nivel global, soportando millones de usuarios simultáneos.

2. Uber – Comunicación de eventos entre servicios

- Uber implementa una infraestructura de mensajería basada en Kafka, que sigue el patrón Publisher–Subscriber, para coordinar la asignación de conductores, el seguimiento de viajes y la actualización de precios en tiempo real.
- Resultado: Sincronización eficiente entre pasajeros, conductores y servidores; alta disponibilidad y tolerancia a fallos en su red global.

3. Meta (Facebook) – Sistema de notificaciones y actividad en tiempo real

- Ejemplo: Facebook utiliza un modelo Pub/Sub para propagar eventos a millones de usuarios simultáneamente: likes, comentarios, mensajes o reacciones. Su infraestructura de notificaciones usa brokers distribuidos para enviar actualizaciones instantáneas a los clientes conectados.
- Resultado: Experiencia fluida y en tiempo real para los usuarios, con mínima latencia y alta confiabilidad.

4. Amazon Web Services (AWS) – Servicios SNS y SQS

- Ejemplo: AWS Simple Notification Service (SNS) y Simple Queue Service (SQS) implementan el patrón Pub/Sub como servicio administrado. SNS

actúa como publicador, y SQS como canal de distribución para múltiples suscriptores o colas.
    - Resultado: Miles de empresas integran notificaciones, alertas o pipelines de datos sin desarrollar su propio broker, reduciendo costos y tiempo de implementación.

5. Google Cloud – Pub/Sub como servicio administrado

    - Ejemplo: Google Cloud Pub/Sub permite manejar flujos de datos de millones de eventos por segundo entre aplicaciones distribuidas, analítica en BigQuery o procesamiento con Dataflow.
    - Resultado: Integración en tiempo real de datos de múltiples fuentes (IoT, logs, APIs) y procesamiento en streaming de gran escala.

6. Tesla – Telemetría vehicular en tiempo real

    - Ejemplo: Los vehículos Tesla envían constantemente información sobre rendimiento, posición y estado del sistema mediante un modelo Pub/Sub basado en MQTT y servicios en la nube.
    - Resultado: Monitoreo continuo, actualizaciones OTA (Over The Air) y mejora del rendimiento mediante análisis en tiempo real.

7. Spotify – Sincronización de dispositivos y reproducción simultánea

    - Ejemplo: Spotify utiliza el patrón Pub/Sub para mantener sincronizados múltiples dispositivos del usuario (móvil, escritorio, altavoz). Cuando se cambia de canción o dispositivo, se publica un evento que actualiza todos los clientes suscritos.
    - Resultado: Experiencia coherente y sincronizada, con comunicación ligera y estable en diferentes redes y plataformas.

8. Industria de IoT – Manufactura y automatización

    - Ejemplo: Empresas como Siemens, Schneider Electric y Bosch emplean el patrón Pub/Sub con MQTT para conectar sensores, controladores industriales y sistemas SCADA.
    - Resultado: Comunicación confiable entre miles de dispositivos, control remoto en tiempo real y mayor eficiencia energética.

VueJs

- Definición

Vue.js es un framework de JavaScript que se utiliza para crear interfaces web interactivas y dinámicas. Se destaca por ser fácil de aprender, ya que permite empezar con proyectos pequeños e ir aumentando su complejidad de manera progresiva. Su funcionamiento se basa en un sistema de componentes y una reactividad que actualiza automáticamente la página

cuando cambian los datos. Gracias a su estructura sencilla y flexible, Vue.js facilita el desarrollo de aplicaciones modernas, rápidas y organizadas.

- Características

    1. Facilidad de uso: Vue.js tiene una sintaxis sencilla y clara, lo que lo hace ideal para quienes están empezando a trabajar con frameworks de JavaScript.
    2. Reactividad: Cuando los datos cambian, la interfaz también se actualiza automáticamente sin necesidad de recargar la página.
    3. Basado en componentes: Permite dividir una aplicación en partes más pequeñas y reutilizables llamadas componentes, lo que facilita el mantenimiento y la organización del código.
    4. Integración progresiva: Se puede usar en proyectos ya existentes sin necesidad de rehacer todo desde cero, ya que se adapta fácilmente a diferentes entornos.
    5. Enlace de datos (data binding): Con Vue.js, los datos y la interfaz están conectados, así que cualquier cambio en uno se refleja en el otro.
    6. Ecosistema completo: Ofrece herramientas como Vue Router (para manejar rutas entre páginas) y Pinia o Vuex (para gestionar el estado de la aplicación).
    7. Rendimiento y rapidez: Es muy liviano, lo que permite que las aplicaciones carguen rápido y funcionen sin problemas.

- Historia y evolución

    - 2014: Vue.js fue creado por Evan You, un exdesarrollador de Google que buscaba una alternativa más ligera y sencilla a AngularJS. Ese año lanzó la primera versión, enfocada en ser fácil de usar y progresiva.
    - 2015: Se lanza Vue 1.0, la primera versión estable. Su sistema reactivo y las plantillas declarativas lo hicieron popular rápidamente entre los desarrolladores.
    - 2016: Aparece Vue 2.0, una versión completamente reescrita que introduce el Virtual DOM, mejora el rendimiento y consolida la arquitectura basada en componentes.
    - 2017–2019: Vue se expande internacionalmente. Se lanza la CLI oficial (herramienta para crear proyectos más rápido) y se fortalecen sus librerías complementarias como Vue Router y Vuex. Durante estos años, se convierte en una de las principales opciones frente a React y Angular.
    - 2020: Llega Vue 3.0, con grandes mejoras: la Composition API (una nueva forma de estructurar el código), soporte completo para TypeScript y mejor rendimiento general.
    - 2021–2024: El ecosistema de Vue 3 se consolida. Surgen herramientas como Vite, que optimizan el desarrollo, y se posiciona como un framework moderno, rápido y adaptable a distintos entornos.

- 2025: Vue.js continúa evolucionando con mejoras en rendimiento y compatibilidad, manteniéndose como uno de los frameworks más usados gracias a su equilibrio entre simplicidad, eficiencia y flexibilidad.

- Ventajas y desventajas

Ventajas	Desventajas
Fácil de aprender: Tiene una sintaxis simple y clara, por lo que es ideal para quienes están comenzando con frameworks de JavaScript.	**Menor respaldo empresarial**: No pertenece a una gran compañía como Google o Meta, lo que puede afectar su promoción.
Alta reactividad: Los cambios en los datos se reflejan automáticamente en la interfaz sin recargar la página	**Ecosistema dividido**: Existen muchas librerías no oficiales, lo que puede generar confusión al elegir herramientas.
Componentes reutilizables: Permite organizar el código en partes pequeñas y reutilizables, facilitando el mantenimiento.	**Curva de aprendizaje en proyectos grandes**: Aunque es fácil de usar, puede volverse más complejo en sistemas muy grandes.
Buen rendimiento: Es un framework liviano, rápido y eficiente al cargar las aplicaciones.	**Menor demanda laboral**: En comparación con React o Angular, hay menos ofertas de trabajo centradas en Vue.js.
Excelente documentación: Cuenta con guías completas y fáciles de entender que ayudan al aprendizaje.	**Problemas de compatibilidad entre versiones**: Pasar de una versión a otra (como de Vue 2 a Vue 3) puede requerir ajustes en el código.
Gran comunidad: Tiene muchos desarrolladores activos que comparten recursos, librerías y soluciones a problemas comunes.	**Dependencia de la comunidad**: Al no tener un respaldo corporativo, su evolución depende mucho de los aportes de los usuarios.

- Casos de Uso (Situaciones y/o problemas donde se pueden aplicar)

1. Desarrollo de aplicaciones web interactivas: Vue.js se usa para crear interfaces dinámicas que responden rápidamente a las acciones del usuario, como paneles de control, formularios y páginas con actualizaciones en tiempo real.
2. Single Page Applications (SPA): Permite desarrollar aplicaciones de una sola página, donde todo el contenido se carga sin necesidad de recargar el navegador, mejorando la experiencia del usuario.
3. Integración en proyectos existentes: Gracias a su estructura progresiva, se puede incorporar fácilmente en sitios ya desarrollados, por ejemplo, para agregar componentes interactivos sin cambiar toda la página.

4. Desarrollo de paneles administrativos (dashboards): Vue es muy usado para crear paneles de gestión con gráficos, tablas y estadísticas actualizadas.
5. Aplicaciones móviles o multiplataforma: Con herramientas como Quasar o NativeScript-Vue, es posible desarrollar aplicaciones móviles usando la misma base de código de Vue.
6. Tiendas en línea o plataformas de comercio electrónico: Muchas empresas lo usan para crear catálogos interactivos, carritos de compra dinámicos y sistemas de búsqueda rápidos.

- Casos de aplicación (Ejemplos y casos de éxito en la industria)

1. Alibaba
   Esta reconocida plataforma de comercio electrónico utiliza Vue.js en varias partes de su sitio web para mejorar la velocidad de carga y la interactividad con los usuarios. Gracias a Vue, pudieron crear interfaces más ligeras y eficientes para manejar millones de productos.
2. Xiaomi
   La empresa china de tecnología emplea Vue.js en su página oficial y tiendas en línea, aprovechando su capacidad para construir interfaces rápidas y adaptables, optimizando la experiencia de compra en distintos dispositivos.
3. Nintendo
   Utiliza Vue.js en algunos de sus portales web para la gestión de contenidos y promociones, ya que el framework facilita el manejo de animaciones y componentes visuales dinámicos, ideales para su estilo interactivo.
4. Behance
   Parte del sitio de Behance, donde los diseñadores publican sus portafolios, está construido con Vue.js. Su estructura basada en componentes permitió mejorar el rendimiento y la carga dinámica de proyectos sin interrupciones.
5. GitLab
   Esta plataforma de desarrollo colaborativo usa Vue.js en su interfaz de usuario para ofrecer una experiencia fluida y reactiva a los programadores al gestionar repositorios, tareas y revisiones de código.
6. Grammarly
   Vue.js también se ha usado en partes de la interfaz de Grammarly, ayudando a manejar de forma eficiente las sugerencias en tiempo real y la interacción del usuario sin recargar la página.

FastAPI

- Definición

FastAPI es un framework web moderno y rápido desarrollado en Python para crear APIs robustas, eficientes y seguras. Está construido sobre Starlette para la parte web y asíncrona, y Pydantic para la validación de datos y manejo de esquemas. Su diseño aprovecha las anotaciones de tipo de Python (type hints), lo que permite a FastAPI validar automáticamente

las solicitudes y respuestas, generar documentación interactiva con Swagger UI y ReDoc, y ofrecer autocompletado y detección de errores desde el entorno de desarrollo.

Además, al estar basado en ASGI (Asynchronous Server Gateway Interface), soporta programación asíncrona (async/await), lo que mejora el rendimiento y la escalabilidad frente a frameworks tradicionales como Flask o Django, especialmente en aplicaciones que manejan muchas peticiones simultáneas o integraciones con servicios externos.

- Características

1. Alto rendimiento: Está construido sobre Starlette y Uvicorn, lo que permite un rendimiento comparable al de frameworks como Node.js o Go, gracias al soporte nativo para operaciones asíncronas (async/await).
2. Validación automática: Utiliza Pydantic para validar datos de entrada y salida según los tipos definidos en Python, reduciendo errores y simplificando el manejo de datos.
3. Documentación interactiva: Genera de forma automática documentación OpenAPI, accesible mediante interfaces visuales como Swagger UI y ReDoc, sin necesidad de configuraciones adicionales.
4. Basado en tipado de Python: Aprovecha las type hints para mejorar la legibilidad del código, la detección de errores en tiempo de desarrollo y la autocompletación en editores.
5. Simplicidad y rapidez de desarrollo: Permite construir APIs funcionales con pocas líneas de código, reduciendo la carga de trabajo del desarrollador y acelerando los tiempos de entrega.
6. Compatibilidad con estándares: Cumple con los estándares modernos como OpenAPI (anteriormente Swagger) y JSON Schema, facilitando la interoperabilidad con otros sistemas y lenguajes.
7. Seguridad integrada: Incluye soporte sencillo para autenticación y autorización mediante OAuth2, JWT y otros esquemas de seguridad.
8. Escalabilidad: Ideal para arquitecturas de microservicios o aplicaciones distribuidas gracias a su diseño asíncrono y modular.
9. Fácil integración: Se integra fácilmente con bases de datos (SQL y NoSQL), colas de mensajes, sistemas de cacheo y herramientas modernas como Docker y Kubernetes.

- Historia y evolución

FastAPI es un framework relativamente reciente dentro del ecosistema Python, pero su adopción ha sido extraordinariamente rápida debido a su enfoque moderno, su alto rendimiento y la facilidad de uso.

Su creador es Sebastián Ramírez, un ingeniero de software colombiano que comenzó el desarrollo de FastAPI alrededor de 2018. En ese momento, los frameworks más usados en Python para crear APIs eran Flask y Django REST Framework, ambos muy populares, pero

con limitaciones frente a las necesidades de las aplicaciones modernas que requerían alto rendimiento, concurrencia y validación automática de datos.

Ramírez identificó varios problemas comunes en los proyectos de backend desarrollados con Python:

- La validación de datos era repetitiva y propensa a errores.
- La documentación de las APIs debía hacerse manualmente.
- El soporte para programación asíncrona no estaba bien integrado.
- La falta de tipado fuerte dificultaba el mantenimiento y la detección temprana de errores.

A partir de esas limitaciones, diseñó FastAPI con una filosofía clara: combinar el poder del tipado estático de Python con la velocidad y eficiencia de ASGI, además de seguir los estándares abiertos de documentación.

Línea de evolución

- 2018: lanzamiento oficial de FastAPI 0.1.0, construido sobre Starlette (para el manejo de peticiones asíncronas) y Pydantic (para validación de datos). Desde el inicio se distinguió por generar automáticamente documentación OpenAPI y por su facilidad para desarrollar endpoints REST con validaciones completas.
- 2019: la comunidad de desarrolladores empezó a adoptarlo en proyectos de microservicios y machine learning, debido a su compatibilidad con frameworks como TensorFlow, PyTorch y scikit-learn, así como con Docker y Kubernetes.
- 2020: FastAPI alcanzó más de 20 mil estrellas en GitHub, siendo reconocido por su rendimiento comparable a Node.js y Go, y por su simpleza en la creación de APIs robustas y bien documentadas. Durante este año, su uso creció notablemente en servicios de datos y en la comunidad de DevOps.
- 2021–2022: el framework consolidó su reputación como la mejor opción para crear APIs modernas y microservicios en Python. Muchas empresas tecnológicas comenzaron a adoptarlo en producción (Netflix, Microsoft, Uber, Explosion AI, entre otras). También se fortaleció la comunidad de extensiones y librerías complementarias, como FastAPI Users (autenticación), SQLModel (ORM oficial del mismo creador) y FastAPI-Admin.
- 2023 en adelante: FastAPI se consolidó como parte del ecosistema estándar de desarrollo backend con Python. Ha mantenido una comunidad activa, actualizaciones frecuentes y una base sólida para integrar inteligencia artificial, APIs de datos y servicios en la nube. Además, su integración con asyncio y WebSockets lo posicionó como una herramienta ideal para aplicaciones en tiempo real y arquitecturas orientadas a eventos.

Contexto actual

Hoy en día, FastAPI es considerado uno de los frameworks más influyentes del ecosistema Python moderno. Su documentación oficial, ejemplos y comunidad activa en GitHub y

Discord lo han convertido en una opción favorita tanto para proyectos académicos como para entornos empresariales y de producción a gran escala. La evolución de FastAPI refleja una tendencia clara en el desarrollo web: mayor velocidad, menos código repetitivo, mejor tipado y cumplimiento estricto de estándares, todo sin sacrificar simplicidad. Es la respuesta moderna de Python al desarrollo de APIs de alto rendimiento y uno de los ejemplos más exitosos de innovación abierta dentro del lenguaje.

- Ventajas y desventajas

Ventajas

- Alto rendimiento. Gracias a su base en ASGI, Starlette y Uvicorn, FastAPI logra una velocidad comparable a frameworks como Node.js o Go, siendo uno de los más rápidos dentro del ecosistema Python.
- Validación automática de datos. Usa Pydantic para validar, transformar y documentar los datos sin escribir código adicional. Esto reduce errores y simplifica la gestión de entradas y salidas en los endpoints.
- Documentación integrada. Genera automáticamente documentación OpenAPI y JSON Schema, accesible mediante Swagger UI y ReDoc, lo que facilita la comunicación entre desarrolladores y clientes.
- Programación asíncrona (async/await). Permite manejar muchas solicitudes simultáneas sin bloquear el servidor, ideal para microservicios, IoT, WebSockets y aplicaciones en tiempo real.
- Basado en tipado estático. Aprovecha las type hints de Python para ofrecer autocompletado, detección de errores y documentación más precisa, mejorando la productividad y mantenibilidad del código.
- Fácil integración. Se integra sin dificultad con bases de datos SQL y NoSQL, herramientas de seguridad (JWT, OAuth2), y frameworks de machine learning o DevOps (Docker, Kubernetes, etc.).
- Curva de aprendizaje corta. Su sintaxis es clara y similar a Flask, pero con funcionalidades más modernas y potentes, lo que facilita adoptarlo rápidamente.

Desventajas

- Relativamente nuevo. Al ser un framework joven (desde 2018), todavía no tiene el mismo nivel de madurez, soporte o ecosistema de extensiones que Django o Flask.
- Menor cantidad de tutoriales avanzados. Aunque la documentación oficial es excelente, aún hay menos recursos y foros con ejemplos complejos en comparación con frameworks más antiguos.
- Dependencia de tipado. El uso intensivo de type hints puede resultar poco familiar o confuso para desarrolladores que no estén acostumbrados al tipado estático en Python.
- Overhead asíncrono en casos simples. En aplicaciones pequeñas o con pocas peticiones simultáneas, el uso de async/await puede añadir complejidad innecesaria.

- Manejo de sesiones limitado. FastAPI no incluye un sistema de autenticación ni sesiones por defecto; se requiere integrar librerías externas o desarrollarlas manualmente.

- Casos de Uso (Situaciones y/o problemas donde se pueden aplicar)

    1. APIs para sistemas empresariales o microservicios
        - Situación: una organización necesita dividir su sistema monolítico en servicios más pequeños, independientes y escalables.
        - Aplicación: FastAPI permite crear microservicios ligeros y eficientes que se comunican entre sí mediante REST o mensajería asíncrona, facilitando la integración entre módulos (usuarios, pagos, reportes, etc.).
    2. Plataformas de Machine Learning y analítica de datos
        - Situación: un equipo de IA necesita exponer modelos de predicción entrenados en Python a través de una API accesible por aplicaciones externas.
        - Aplicación: FastAPI facilita la creación de endpoints que reciben datos, ejecutan el modelo (por ejemplo, con scikit-learn, TensorFlow o PyTorch) y devuelven resultados en formato JSON, todo con excelente rendimiento.
    3. Aplicaciones en tiempo real o de alta concurrencia
        - Situación: se requiere manejar múltiples conexiones simultáneas, como en chats, dashboards en vivo o sensores IoT.
        - Aplicación: gracias a su soporte para WebSockets y async/await, FastAPI permite construir aplicaciones reactivas y de baja latencia, ideales para sistemas de monitoreo, trading o mensajería instantánea.
    4. Integración con sistemas externos o pasarelas de comunicación
        - Situación: una empresa necesita conectar su aplicación con servicios externos (pagos, correo, autenticación, etc.) de manera confiable y rápida.
        - Aplicación: FastAPI puede actuar como middleware o API Gateway, gestionando peticiones, validando tokens, y redirigiendo tráfico entre servicios de terceros.
    5. Aplicaciones móviles o web que consumen APIs REST
        - Situación: una aplicación móvil requiere un backend rápido y seguro para manejar registro, autenticación, y consumo de datos.
        - Aplicación: FastAPI proporciona endpoints seguros con JWT y OAuth2, gestionando peticiones REST y sirviendo contenido dinámico hacia aplicaciones frontend (Vue, React, Angular, Flutter, etc.).
    6. Automatización y comunicación entre dispositivos IoT
        - Situación: una red de sensores necesita enviar datos constantemente a un servidor central.

    - Aplicación: FastAPI puede recibir, procesar y almacenar datos en tiempo real desde múltiples dispositivos, integrándose fácilmente con MQTT o WebSockets.
7. Desarrollo rápido de prototipos y pruebas
    - Situación: un equipo de desarrollo necesita construir y probar rápidamente un API funcional para validar una idea o servicio.
    - Aplicación: con FastAPI, se puede desarrollar un prototipo completo con documentación y validaciones en cuestión de horas, listo para pruebas o demostraciones.

- Casos de aplicación (Ejemplos y casos de éxito en la industria)

1. Netflix — Gestión interna de datos y automatización

    - Descripción: Netflix utiliza FastAPI en varios de sus servicios internos para procesamiento y análisis de datos.
    - Aplicación: gracias a su rendimiento y facilidad para construir APIs rápidas, lo emplean para exponer endpoints que permiten automatizar flujos de datos y tareas analíticas internas.
    - Impacto: reducción significativa en los tiempos de respuesta y mantenimiento, mejorando la eficiencia de herramientas usadas por los equipos de ingeniería.

2. Microsoft — Integración con servicios de Machine Learning

    - Descripción: Microsoft adoptó FastAPI en algunos proyectos del ecosistema Azure Machine Learning.
    - Aplicación: se usa para desplegar modelos de IA como servicios REST, integrando Python con sus soluciones en la nube.
    - Impacto: permitió acelerar la creación de endpoints de inferencia para modelos, reduciendo la complejidad del despliegue y aumentando la productividad de los equipos de ciencia de datos.

3. Uber — Backend de servicios de datos y análisis

    - Descripción: Uber emplea FastAPI en ciertos microservicios relacionados con su infraestructura de datos.
    - Aplicación: sirve para crear APIs ligeras que procesan solicitudes analíticas y comunican resultados entre servicios distribuidos.
    - Impacto: facilita la escalabilidad y la eficiencia en el manejo de grandes volúmenes de datos, manteniendo baja latencia.

4. Explosion AI — spaCy y Prodigy

    - Descripción: La empresa desarrolladora de spaCy y Prodigy, dos de las herramientas más usadas en procesamiento de lenguaje natural (NLP), usa FastAPI como su framework principal.

- Aplicación: sirve para conectar interfaces de usuario con modelos NLP y flujos de anotación de datos.
- Impacto: posibilita una comunicación rápida entre la interfaz de usuario y los modelos de IA, optimizando el tiempo de desarrollo y despliegue.

5. DocuBrain — Solución de automatización documental

   - Descripción: startup que implementó FastAPI como backend para un sistema de análisis inteligente de documentos.
   - Aplicación: expone modelos de reconocimiento de texto (OCR) y clasificación de archivos como APIs.
   - Impacto: incrementó el rendimiento del procesamiento documental en un 40% frente a su versión anterior con Flask.

MySQL

- Definición

MySQL es un sistema de gestión de bases de datos relacional que utiliza el lenguaje SQL (Structured Query Language) para administrar y consultar la información. Es uno de los sistemas más populares del mundo debido a que es gratuito, rápido, estable y de código abierto.

Permite almacenar, organizar y acceder a grandes volúmenes de datos de forma eficiente, por lo que es muy usado en páginas web, aplicaciones y sistemas empresariales. Además, que puede trabajar junto con diferentes lenguajes de programación como Java, PHP, Python o C#, lo que lo hace muy versátil.

- Características
  1. Lenguaje estándar SQL: Utiliza el lenguaje SQL para crear, modificar y consultar datos dentro de las bases de datos.
  2. Modelo relacional: Organiza la información en tablas relacionadas entre sí, lo que facilita la búsqueda y el manejo de grandes volúmenes de datos.
  3. Multiplataforma: Funciona en distintos sistemas operativos como Windows, Linux y macOS, lo que le da gran flexibilidad.
  4. Código abierto: Es un software gratuito y de código abierto, lo que permite modificarlo y adaptarlo según las necesidades del usuario o la empresa.
  5. Alta velocidad y rendimiento: Está optimizado para manejar consultas complejas de manera rápida, incluso con bases de datos grandes.
  6. Seguridad y control de acceso: Permite establecer usuarios, contraseñas y permisos, protegiendo los datos de accesos no autorizados.
  7. Soporte para transacciones: Garantiza la integridad de los datos mediante el uso de transacciones (operaciones que deben cumplirse completamente o no ejecutarse).

8. Escalabilidad: Puede usarse desde pequeños proyectos personales hasta sistemas empresariales con millones de registros.

- Historia y evolución
  - **1994**: MySQL fue creado por Michael Widenius (Monty) y David Axmark en Suecia. La idea surgió como una forma más rápida y sencilla de manejar bases de datos que otros sistemas de la época.
  - **1995**: Se lanza oficialmente la primera versión pública de MySQL. Su éxito se debió a que era gratuito, de código abierto y fácil de instalar, lo que atrajo rápidamente a la comunidad de desarrolladores.
  - **2000**: MySQL se vuelve totalmente software libre bajo la licencia GNU GPL, lo que impulsa aún más su crecimiento. Durante estos años, muchas empresas comienzan a usarlo como parte del conjunto de tecnologías conocido como LAMP (Linux, Apache, MySQL, PHP/Python/Perl).
  - **2005**: La empresa MySQL AB, encargada del desarrollo del sistema, es comprada por Sun Microsystems, lo que permitió invertir más en soporte y mejoras de rendimiento.
  - **2010**: Oracle Corporation adquiere Sun Microsystems y, con ello, MySQL pasa a formar parte de Oracle. A partir de entonces, surgen versiones comerciales y comunitarias (Community Edition y Enterprise Edition).
  - **2013–2019**: Se lanzan versiones más modernas con mejoras en rendimiento, seguridad y replicación de datos. También aparecen proyectos derivados como MariaDB, creado por el propio fundador original, como una alternativa 100 % libre.
  - **2020–2025**: MySQL sigue evolucionando, con compatibilidad en la nube (MySQL HeatWave), mayor integración con servicios web y mejoras en la velocidad de procesamiento y seguridad. Hoy en día, sigue siendo uno de los gestores de bases de datos más usados en el mundo, tanto en entornos académicos como empresariales.

- Ventajas y desventajas

Ventajas	Desventajas
Gratuito y de código abierto: Puede descargarse y usarse sin costo, y su código puede modificarse según las necesidades del proyecto.	**Limitaciones con datos muy complejos:** Aunque es rápido, puede presentar dificultades con consultas muy pesadas o bases de datos extremadamente grandes.
Rendimiento y velocidad: Está optimizado para ejecutar consultas rápidamente, incluso en bases de datos con gran cantidad de información.	**Licencia dual (con Oracle):** Algunas funciones avanzadas solo están disponibles en la versión Enterprise, que es de pago.

Compatibilidad multiplataforma: Funciona en sistemas operativos como Windows, Linux y macOS, lo que lo hace muy flexible.	**Pocas herramientas gráficas avanzadas integradas:** Aunque existen programas externos, el sistema base no incluye un entorno visual tan completo como otros gestores.
Facilidad de uso: Su instalación y configuración son sencillas, por lo que es ideal tanto para principiantes como para profesionales.	**Escalabilidad limitada en entornos empresariales grandes**: En sistemas muy grandes, algunas empresas prefieren soluciones como PostgreSQL u Oracle Database por su mayor robustez.
Alta seguridad: Ofrece autenticación de usuarios, cifrado de contraseñas y control de privilegios, protegiendo los datos almacenados.	**Gestión limitada de transacciones complejas:** No siempre maneja de forma óptima operaciones muy complicadas que involucran varias tablas o procesos simultáneos.
Amplia comunidad y soporte: Al ser tan popular, cuenta con gran cantidad de documentación, foros y tutoriales disponibles.	**Dependencia de Oracle:** Desde que fue adquirido, parte de su desarrollo depende de las decisiones comerciales de Oracle, lo que genera preocupación en la comunidad.

- Casos de Uso (Situaciones y/o problemas donde se pueden aplicar)

1. Gestión de bases de datos en sitios web dinámicos: MySQL se usa ampliamente para almacenar y manejar información en sitios web donde el contenido cambia constantemente, como tiendas en línea, blogs o foros. Ejemplo: guardar usuarios, productos, comentarios o publicaciones de manera organizada.
2. Aplicaciones empresariales y sistemas administrativos: Se aplica en empresas que necesitan manejar grandes volúmenes de datos, como facturación, nómina o inventarios. Ejemplo: un sistema de gestión que registra ventas, clientes y pagos en tiempo real.
3. Aplicaciones educativas o académicas: MySQL se usa en plataformas de gestión escolar, bibliotecas digitales o sistemas de calificaciones, donde se necesita almacenar información estructurada y segura.
4. Sistemas de reservas y logística: Es ideal para manejar información de reservas de hoteles, vuelos, citas médicas o pedidos, ya que permite realizar consultas rápidas y confiables.
5. Aplicaciones móviles y servicios en la nube: Muchas apps móviles usan MySQL como base de datos principal o en combinación con servicios web para sincronizar datos entre usuarios y servidores.

6. Análisis de datos y reportes: Se usa en proyectos donde se requiere recopilar, consultar y analizar información, por ejemplo, estadísticas de uso, desempeño o ventas.

- Casos de aplicación (Ejemplos y casos de éxito en la industria)

1. Facebook
   Aunque usa varias tecnologías, MySQL ha sido parte clave en su infraestructura desde sus inicios. Se utiliza para almacenar datos de usuarios, publicaciones y relaciones, aprovechando su capacidad para manejar millones de consultas por segundo de forma eficiente.
2. YouTube
   Antes de pertenecer a Google, YouTube usaba MySQL para gestionar los datos de sus videos, usuarios y comentarios. Su rendimiento y estabilidad fueron esenciales para soportar el rápido crecimiento de la plataforma.
3. Twitter
   MySQL fue uno de los sistemas de base de datos originales en Twitter, ayudando a manejar millones de tuits y perfiles durante sus primeros años de expansión.
4. Netflix
   Utiliza MySQL para el almacenamiento de datos críticos, como información de usuarios, registros de dispositivos y preferencias. Se combina con otras tecnologías para ofrecer una experiencia personalizada a cada usuario.
5. Airbnb
   MySQL forma parte de la base de datos que permite a Airbnb gestionar reservas, usuarios y propiedades a nivel mundial, garantizando disponibilidad y coherencia en la información.
6. Shopify
   Esta plataforma de comercio electrónico usa MySQL para manejar millones de transacciones y productos en miles de tiendas en línea, destacando por su fiabilidad y escalabilidad.

MQTT

- Definición

MQTT (*Message Queuing Telemetry Transport*) es un protocolo ligero de mensajería basado en el modelo publicador–suscriptor (pub/sub), diseñado para la comunicación entre dispositivos con recursos limitados o redes de baja calidad. Se ejecuta sobre TCP/IP y permite el intercambio eficiente de mensajes entre múltiples clientes a través de un broker, que actúa como intermediario gestionando las suscripciones y la entrega de datos.

Su diseño se enfoca en la eficiencia, confiabilidad y bajo consumo de ancho de banda, lo que lo convierte en un estándar ampliamente utilizado en el Internet de las Cosas (IoT), la telemetría industrial y los sistemas embebidos. MQTT permite mantener conexiones persistentes y seguras mediante mecanismos de confirmación de entrega y distintos niveles

de calidad de servicio (QoS), garantizando que los mensajes lleguen incluso en condiciones de red inestables o con dispositivos intermitentes.

- Características

1. Modelo publicador–suscriptor: Utiliza una arquitectura donde los dispositivos publican mensajes en *topics* (temas), y otros dispositivos suscritos a esos temas los reciben a través de un broker, eliminando la dependencia directa entre emisor y receptor.
2. Ligero y eficiente: Su encabezado mínimo (apenas 2 bytes) y bajo consumo de recursos lo hacen ideal para dispositivos con poca memoria, energía o ancho de banda, como sensores o microcontroladores.
3. Orientado a conexión persistente: Mantiene una conexión TCP/IP estable con el broker, lo que permite la comunicación en tiempo real y reduce la necesidad de reconexiones frecuentes.
4. Calidad de servicio (QoS): Ofrece tres niveles de entrega de mensajes:
    - QoS 0: "como máximo una vez" (sin confirmación).
    - QoS 1: "al menos una vez" (confirmación requerida).
    - QoS 2: "exactamente una vez" (garantiza entrega única).
5. Soporte para retención de mensajes: El broker puede guardar el último mensaje publicado en un tema para entregarlo automáticamente a nuevos suscriptores.
6. Mensajes retenidos y persistentes: Permite definir mensajes persistentes y mantener sesiones activas incluso si el cliente se desconecta temporalmente, útil para redes inestables.
7. Compatibilidad con seguridad: Soporta autenticación, autorización y cifrado SSL/TLS, garantizando comunicaciones seguras entre dispositivos y el broker.
8. Extensibilidad y compatibilidad: Puede integrarse con otros protocolos y servicios (HTTP, WebSockets, AMQP), y es compatible con brokers populares como Eclipse Mosquitto, HiveMQ y EMQX.
9. Multiplataforma y estandarizado: Es un protocolo abierto (ISO/IEC 20922) y cuenta con implementaciones en múltiples lenguajes y plataformas, lo que facilita su adopción en diversos entornos de desarrollo.

- Historia y evolución

MQTT (Message Queuing Telemetry Transport) nació para resolver un problema muy específico: comunicación confiable y barata entre dispositivos remotos en redes inestables y de bajo ancho de banda. Fue creado en 1999 por Andy Stanford-Clark (IBM) y Arlen Nipper (Arcom/Cirrus Link) para monitoreo industrial (oleoductos, telemetría). Su diseño minimalista, orientado a pub/sub y con encabezados muy pequeños, lo hizo ideal para equipos con recursos limitados y enlaces satelitales costosos.

Con el auge del Internet de las Cosas (IoT) en la década de 2010, MQTT pasó de nicho industrial a estándar de facto para M2M: miles de sensores, gateways y servicios en la nube lo adoptaron por su eficiencia energética, sesiones persistentes y niveles de QoS. La apertura

del protocolo y su estandarización consolidaron un ecosistema de brokers (Mosquitto, EMQX, HiveMQ) y librerías en casi cualquier lenguaje.

Línea de evolución

- 1999: Diseño inicial por Stanford-Clark y Nipper para telemetría industrial en redes de alta latencia/bajo ancho de banda.
- 2003–2010: Adopción en automatización y SCADA; maduración del patrón publicador–suscriptor sobre TCP/IP.
- 2013: IBM libera MQTT como protocolo abierto; la comunidad y los vendors comienzan a contribuir activamente.
- 2014: OASIS estandariza MQTT 3.1.1, clarificando interoperabilidad (codificación, keep-alive, ack, etc.).
- 2016: Se publica como ISO/IEC 20922, elevando su estatus a estándar internacional.
- 2018–2019: Llega MQTT 5.0, gran actualización con propiedades de mensaje, razones de desconexión, respuesta del servidor, sesiones mejoradas y control de flujo, manteniendo compatibilidad conceptual con 3.1.1.
- 2020 → presente: Consolidación en nubes públicas (AWS IoT Core, Azure IoT, GCP), soporte por WebSockets, bridging entre brokers, y herramientas de observabilidad/gestión empresarial.

Contexto actual

Hoy, MQTT es el protocolo dominante en IoT y telemetría: funciona bien desde sensores alimentados por batería hasta flotas de vehículos y fábricas conectadas. Su combinación de ligereza, QoS, retained messages, last-will, y seguridad vía TLS/autenticación lo hace fiable en producción. La versión 5.0 habilitó casos más complejos (ruteo fino, error handling rico, metadatos) y una mejor interoperabilidad entre clientes y brokers, asegurando su vigencia para arquitecturas event-driven y sistemas distribuidos modernos. MQTT evolucionó de una solución industrial cerrada a un estándar abierto y global, motor clave de la conectividad IoT gracias a su eficiencia, simplicidad y foco en la confiabilidad bajo restricciones reales de red y dispositivo.

- Ventajas y desventajas

Ventajas

- Ligereza y eficiencia. MQTT utiliza un encabezado muy pequeño (2 bytes), lo que reduce el uso de ancho de banda y energía, ideal para dispositivos con recursos limitados o redes inestables.
- Modelo publicador–suscriptor. Separa los emisores de los receptores mediante un broker, simplificando la comunicación entre múltiples clientes sin necesidad de conexión directa.

- Confiabilidad mediante niveles de QoS. Permite definir tres niveles de entrega (QoS 0, 1 y 2) para ajustar el equilibrio entre rendimiento y seguridad en la entrega de mensajes.
- Persistencia y retención de mensajes. Puede guardar mensajes y reenviarlos cuando un dispositivo vuelve a conectarse, garantizando continuidad incluso ante desconexiones temporales.
- Escalabilidad y tiempo real. Diseñado para manejar miles de conexiones simultáneas, lo que lo hace ideal para sistemas de IoT, monitoreo remoto o redes de sensores distribuidas.
- Soporte para seguridad. Implementa autenticación, autorización y cifrado mediante TLS/SSL, protegiendo la comunicación entre clientes y broker.
- Interoperabilidad. Es un estándar abierto (ISO/IEC 20922) con soporte en múltiples lenguajes, plataformas y brokers, lo que facilita su integración en soluciones empresariales y de nube.

Desventajas

1. Dependencia de un broker central. Toda la comunicación pasa por el broker; si este falla o se sobrecarga, el sistema completo puede verse afectado.
2. Requiere una conexión TCP constante. Aunque eficiente, necesita conexiones persistentes, lo que puede ser problemático en redes intermitentes o con alto consumo energético.
3. Seguridad no predeterminada. MQTT no cifra la comunicación por defecto; el cifrado debe configurarse manualmente mediante TLS, lo que aumenta la complejidad inicial.
4. Limitaciones en transmisión de grandes volúmenes de datos. Está optimizado para mensajes pequeños y frecuentes; no es ideal para transferencia de archivos grandes o multimedia.
5. Falta de control de flujo avanzado. En versiones anteriores (3.1.1), el control de errores y la gestión detallada de sesiones eran limitados; aunque MQTT 5.0 mejora esto, no todos los brokers lo soportan plenamente.

- Casos de Uso (Situaciones y/o problemas donde se pueden aplicar)

    1. Monitoreo y control de dispositivos IoT

        - Situación: una red de sensores distribuidos (temperatura, humedad, presión, movimiento) necesita enviar datos constantemente a un servidor central.
        - Aplicación: MQTT permite que cada sensor publique sus lecturas en un *topic*, mientras un servidor o dashboard se suscribe para recibirlas en tiempo real, incluso con conexiones intermitentes.

    2. Automatización industrial y telemetría

- Situación: en una planta industrial, múltiples máquinas deben reportar su estado operativo y recibir órdenes desde un sistema central.
- Aplicación: MQTT posibilita la comunicación confiable entre controladores, PLCs y sistemas SCADA, reduciendo latencia y garantizando entrega de datos críticos.

3. Domótica y hogares inteligentes

    - Situación: diferentes dispositivos del hogar (luces, cámaras, termostatos) deben comunicarse entre sí o con una aplicación móvil.
    - Aplicación: MQTT gestiona la comunicación en tiempo real entre los dispositivos y el servidor local o en la nube, permitiendo control remoto y automatización eficiente.

4. Vehículos conectados y transporte inteligente

    - Situación: una flota de vehículos necesita reportar ubicación, velocidad o alertas al centro de control de tráfico.
    - Aplicación: MQTT, por su bajo consumo de datos y capacidad de conexión persistente, es ideal para transmitir telemetría desde vehículos a servidores o aplicaciones móviles.

5. Sistemas de energía y redes eléctricas inteligentes (Smart Grids)

    - Situación: se requiere monitorear y ajustar en tiempo real el consumo y la generación de energía en diferentes puntos de la red.
    - Aplicación: MQTT permite la comunicación constante entre medidores inteligentes, estaciones de control y sistemas de análisis centralizados.

6. Salud digital y dispositivos médicos

    - Situación: hospitales y clínicas necesitan recibir datos continuos de dispositivos biomédicos (monitores de ritmo cardíaco, presión arterial, oxímetros).
    - Aplicación: MQTT garantiza la entrega fiable de estos datos al sistema central de monitoreo, incluso si hay interrupciones temporales en la red.

7. Agricultura inteligente

    - Situación: fincas y cultivos requieren controlar riego, temperatura o calidad del suelo desde zonas rurales con conectividad limitada.
    - Aplicación: MQTT se usa para conectar sensores y actuadores que transmiten datos al sistema central o aplicación móvil del agricultor, optimizando recursos y decisiones.

8. Notificaciones y mensajería en tiempo real

    - Situación: una aplicación necesita enviar alertas instantáneas o actualizaciones sin sobrecargar el servidor.
    - Aplicación: MQTT puede actuar como sistema de mensajería ligera, donde los clientes se suscriben a temas de interés (alertas, actualizaciones, eventos).

- Casos de aplicación (Ejemplos y casos de éxito en la industria)

1. Facebook Messenger

    - Descripción: Facebook adoptó MQTT como base de su sistema de mensajería instantánea para millones de usuarios en todo el mundo.
    - Aplicación: MQTT permite enviar y recibir mensajes en tiempo real con bajo consumo de batería y datos, algo esencial en dispositivos móviles.
    - Impacto: Logró mejorar la velocidad y confiabilidad de la mensajería en condiciones de red limitadas, optimizando el rendimiento en países con conectividad inestable.

2. Amazon Web Services (AWS IoT Core)

    - Descripción: AWS IoT Core utiliza MQTT como protocolo principal para la comunicación entre dispositivos IoT y la nube.
    - Aplicación: Permite conectar sensores, microcontroladores y sistemas industriales que envían datos a la nube para análisis, automatización y control remoto.
    - Impacto: Proporciona una infraestructura escalable para millones de dispositivos conectados, asegurando comunicación segura y eficiente en tiempo real.

3. Microsoft Azure IoT Hub

    - Descripción: Azure también integra MQTT como canal de comunicación entre dispositivos y servicios en la nube.
    - Aplicación: Empresas lo utilizan para monitorear maquinaria, flotas vehiculares o sensores ambientales.
    - Impacto: Garantiza interoperabilidad, seguridad y confiabilidad en la transmisión de datos industriales, fortaleciendo la analítica y el mantenimiento predictivo.

4. Volkswagen Group — Vehículos conectados

    - Descripción: El grupo Volkswagen emplea MQTT para conectar vehículos con plataformas de análisis y mantenimiento remoto.
    - Aplicación: Los autos publican información de diagnóstico, geolocalización y rendimiento en temas específicos que son consumidos por sistemas en la nube.

- Impacto: Ha permitido mejorar los servicios de asistencia, actualizaciones remotas (*over-the-air*) y monitoreo de desempeño de flotas.

5. Siemens — Automatización industrial

   - Descripción: Siemens integra MQTT en soluciones de Industrial IoT (IIoT) para comunicación entre sensores, controladores y sistemas SCADA.
   - Aplicación: Facilita la recolección de datos de procesos industriales y su envío a plataformas analíticas o de mantenimiento predictivo.
   - Impacto: Incrementa la eficiencia de las operaciones industriales y reduce los costos de mantenimiento no planificado.

6. Ericsson y Telefónica — Smart Cities

   - Descripción: Estas compañías utilizan MQTT para sistemas de ciudades inteligentes, incluyendo alumbrado público, transporte y monitoreo ambiental.
   - Aplicación: Los sensores urbanos publican datos (tráfico, calidad del aire, energía) y las plataformas centralizadas los procesan en tiempo real.
   - Impacto: Optimización del consumo energético, mejora en la movilidad urbana y toma de decisiones basada en datos.

7. Nest (Google) — Hogares inteligentes

   - Descripción: Los termostatos y dispositivos inteligentes de Google Nest usan MQTT para comunicarse entre sí y con la nube.
   - Aplicación: Publican estados (temperatura, energía, movimiento) y reciben comandos desde la aplicación móvil.
   - Impacto: Mejora la eficiencia energética y ofrece al usuario una experiencia de control doméstico fluida y confiable.

8. Agricultura conectada — John Deere y Agri-Tech startups

   - Descripción: En el sector agrícola, empresas como John Deere usan MQTT para conectar maquinaria, sensores de campo y estaciones meteorológicas.
   - Aplicación: Los dispositivos publican datos sobre humedad del suelo, riego y condiciones climáticas.
   - Impacto: Permite decisiones automatizadas, ahorro de recursos hídricos y mejora en el rendimiento de los cultivos.

- **Que tan común es el stack designado**

  Sí, bastante, sobre todo en IoT y dashboards. Sin embargo no es tan común en web ya que se usa otros frameworks lo demás se usa mucho y es una elección sólida.

  ¿Por qué?

1. **Vue.js**: rápido de montar, ideal para SPA y paneles.
2. **FastAPI**: APIs Python modernas, tipadas y muy rápidas.
3. **MySQL**: súper estable y ubicuo en producción.
4. **MQTT**: estándar de facto para IoT/telemetría.

¿Qué tan comunes son cada uno?

1. **MySQL** = Muy común (en todo tipo de proyectos).
2. **Vue.js** = Común (especialmente en dashboards; React tiene más cuota global).
3. **FastAPI** = Cada vez más común para APIs nuevas en Python.
4. **MQTT** = Muy común en IoT (poco común fuera de IoT).

- **Matriz de análisis de Principios SOLID vs Temas**

Principio SOLID	Vue.js (Frontend)	FastAPI (Backend)	MySQL (Base de Datos)	MQTT (Integración)
S – Responsabilidad Única	Cada componente maneja una función específica (formulario, lista de noticias, etc.).	Cada endpoint se encarga de una sola operación (crear noticia, obtener suscriptores, etc.).	Cada tabla almacena datos de una sola entidad (usuarios, noticias, mensajes).	Cada tópico MQTT se dedica a un tipo de información (por ejemplo, noticias/nacionales).
O – Abierto/Cerrado	Se pueden agregar nuevos componentes sin modificar los existentes.	Se pueden extender servicios agregando nuevas rutas sin alterar las ya creadas.	Es posible añadir nuevas tablas o campos sin afectar las consultas principales.	Se pueden crear nuevos tópicos o suscriptores sin modificar los anteriores.
L – Sustitución de Liskov	Componentes derivados mantienen la estructura y comportamiento base (por ejemplo, un CardNoticia hereda de CardBase).	Servicios o clases hijas respetan la interfaz esperada de las clases base (por ejemplo, PublicadorService y SuscriptorService).	Vistas o relaciones entre tablas pueden sustituirse sin romper consultas.	Un nuevo cliente MQTT puede reemplazar otro mientras siga las reglas del protocolo.
I – Segregación de Interfaces	Los componentes solo reciben las props que necesitan (no se cargan	Las rutas de la API están divididas por módulos (por ejemplo, /noticias,	Cada consulta o vista solo devuelve los campos necesarios.	Los suscriptores se conectan únicamente a los tópicos que les interesan.

	funciones innecesarias).	/usuarios, /suscriptores).		
D – Inversión de Dependencias	Los componentes dependen de datos externos (API) y no directamente del backend.	FastAPI usa inyección de dependencias (por ejemplo, el servicio MQTT o la conexión a BD).	El acceso a datos se hace mediante un ORM (SQLAlchemy) , no consultas directas.	Los servicios publican y reciben mensajes a través del broker, no de forma directa entre ellos.

- **Matriz de análisis de Atributos de Calidad vs Temas**

Atributos de calidad (ISO/IEC 25010)	**VueJS**	**FastAPI**	**MySQL**	**MQTT**
Adecuación funcional	UI reactiva; facilita cumplir requerimientos funcionales en el cliente.	Validación y serialización con Pydantic; OpenAPI automático.	ACID; SQL rico; vistas/funciones/triggers para reglas de negocio.	Pub/Sub ligero; entrega por QoS; retained/last will.
Eficiencia de desempeño	DOM virtual y reactividad eficiente; buen rendimiento en SPA.	ASGI + async/await; muy baja latencia; alto throughput.	Índices, caché de consultas; buen rendimiento OLTP.	Encabezado mínimo; excelente en enlaces de baja banda.
Compatibilidad	Integra con REST/WebSockets; convive con librerías externas.	Interopera vía OpenAPI; fácil orquestación con microservicios.	Conectores amplios; replica y federa con otras BD.	Interopera con HTTP/WebSockets/bridges entre brokers.
Usabilidad	Curva amable, DX sólida; ecosistema de componentes.	Docs interactivas (Swagger/ReDoc) ; DX excelente.	Herramientas maduras (Workbench); lenguaje estándar SQL.	Modelo simple de topics; fácil de adoptar en IoT.
Confiabilidad	Depende del navegador; pruebas E2E/unitarias mejoran estabilidad.	Starlette estable; pruebas con TestClient; manejo de errores estructurado.	Replicación, backups, InnoDB; alta disponibilidad con clusters.	Sesiones persistentes; reintentos vía broker; tolerante a cortes.

Seguridad	XSS/CSRF mitigables con buenas prácticas; requiere hardening.	OAuth2/JWT, dependencias, TLS vía proxy; fácil de reforzar.	Cifrado en tránsito/descanso; usuarios/roles/GRANTs.	TLS/SSL, auth por usuario/ACLs; requiere configuración cuidadosa.
Mantenibilidad	Arquitectura por componentes y tipado opcional (TS) facilitan cambios.	Código tipado; modular; fácil refactor con type hints.	Esquemas versionables; migraciones (Liquibase/Flyway).	Clientes simples; topologías escalables; bajo acoplamiento.
Portabilidad	Funciona en navegadores modernos; SSR/Nuxt para distintos entornos.	Despliegue en Uvicorn/Gunicorn; contenedores; nubes múltiples.	Multiplataforma; contenedores; cloud managed (RDS, etc.).	Corre en múltiples brokers y dispositivos; apto edge/cloud.

- **Matriz de análisis de Tácticas vs Temas**

Tácticas / Temas	**VueJS**	**FastAPI**	**MySQL**	**MQTT**
Tácticas de rendimiento	DOM virtual, lazy loading, rendering reactivo.	Uso de async/await, alto rendimiento con ASGI y Uvicorn, caché de respuestas.	Uso de índices, optimización de consultas, caché de resultados y buffer pool.	Mensajes ligeros, encabezado mínimo, comunicación asíncrona persistente.
Tácticas de seguridad	Protección contra XSS/CSRF, validación en formularios, uso de HTTPS.	Autenticación con OAuth2 y JWT, cifrado TLS, validación automática con Pydantic.	Roles y permisos, cifrado de datos en tránsito y en reposo, auditorías.	Autenticación por usuario/clave, TLS/SSL, listas de control de acceso (ACL).
Tácticas de disponibilidad	Recuperación del estado del componente y reactividad en tiempo real.	Despliegue en múltiples instancias, soporte de balanceo de carga y recuperación ante fallos.	Replicación, clustering, backups automáticos, recuperación ante fallos.	Sesiones persistentes, QoS, brokers redundantes para alta disponibilidad.

Tácticas de mantenibilidad	Componentes reutilizables y desacoplados; separación lógica de vista y estado.	Código modular, tipado fuerte, dependencias bien definidas.	Normalización de esquemas, scripts de migración y consistencia ACID.	Topología desacoplada, simplicidad de clientes y modularidad del protocolo.
Tácticas de escalabilidad	Carga diferida de componentes, división de código, optimización de recursos.	Arquitectura de microservicios, contenedorización, integración con orquestadores.	Sharding, replicación maestro-esclavo y balanceo de carga de consultas.	Soporte para miles de clientes concurrentes, brokers escalables, bridging entre nodos.
Tácticas de usabilidad	Binding bidireccional, interfaz intuitiva, comunicación fluida con el usuario.	Documentación automática (Swagger/ReDoc), endpoints claros y bien estructurados.	Herramientas gráficas y sintaxis SQL estándar ampliamente conocida.	Modelo simple de topics, facilidad de adopción en entornos IoT y bajo consumo de recursos.

- **Matriz de análisis de Patrones vs Temas**

Patrones / Temas	**VueJS**	**FastAPI**	**MySQL**	**MQTT**
Modelo–Vista–Controlador (MVC)	Implementa la Vista y parte del Controlador en el cliente mediante componentes y reactividad.	Cumple el rol de Controlador y parte del Modelo en aplicaciones RESTful.	Representa el Modelo dentro del patrón MVC, almacenando y estructurando los datos.	No aplica al MVC clásico; se centra en la comunicación entre servicios.
Cliente–Servidor	Funciona como cliente que consume servicios REST o GraphQL desde el servidor.	Actúa como servidor que expone endpoints al cliente (VueJS u otros).	Opera como servidor de base de datos que responde a consultas SQL del backend.	Opera como middleware dentro de una arquitectura cliente–servidor extendida.
Microservicios	Consume microservicios a través de endpoints; se	Soporta arquitectura de microservicios mediante APIs	Cada microservicio puede tener su	Integra microservicios a través del intercambio de

	integra fácilmente con APIs independientes.	independientes y contenedorización.	propia instancia o base dedicada.	mensajes asincrónicos.
Publicador–Suscriptor (Pub/Sub)	Puede suscribirse a canales de datos en tiempo real mediante WebSockets o MQTT.	Puede publicar o suscribirse a mensajes usando brokers MQTT o Redis.	Puede almacenar mensajes o logs generados por sistemas Pub/Sub.	Es la implementación por excelencia del patrón Publicador–Suscriptor.
Repository / DAO	No aplica directamente; consume datos gestionados por el backend.	Implementa repositorios y modelos para la capa de acceso a datos.	Es la base del patrón Repository, centralizando la persistencia de datos.	Puede integrarse con bases de datos o servicios para almacenar mensajes persistentes.
Arquitectura dirigida por eventos (Event-Driven Architecture, EDA)	Reacciona a eventos del usuario y actualiza la interfaz de forma dinámica.	Gestiona peticiones asíncronas y respuestas tipo evento con async/await.	Recibe eventos de inserción/actualización y puede disparar triggers o notificaciones.	Sustenta arquitecturas dirigidas por eventos al transmitir mensajes en tiempo real.

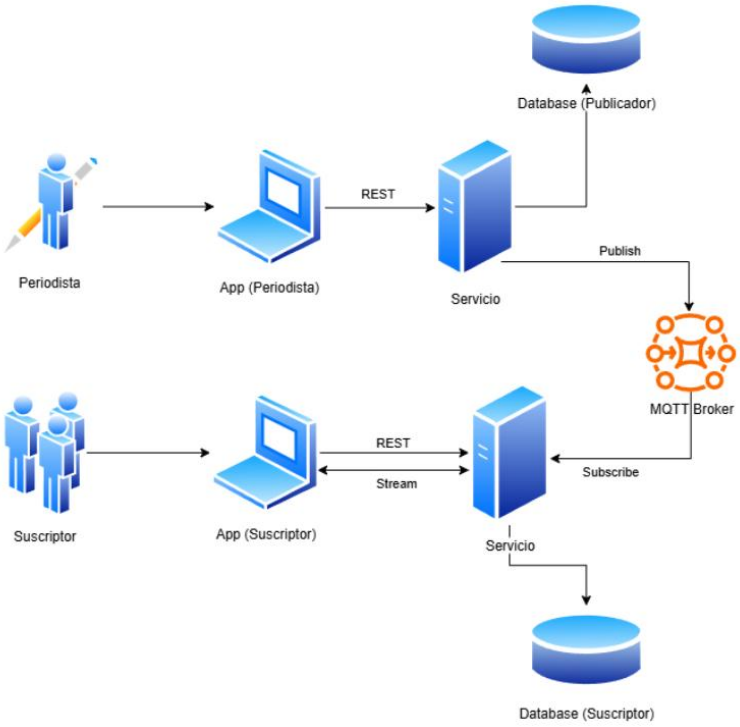
- **Matriz de análisis de Mercado Laboral vs Temas**

Tema / Tecnología	Demanda laboral actual	Fortalezas en el mercado	Desafíos / competencia	Salario estimado (anual / mensual / diario)	Nivel típico estimado
Vue.js (Frontend)	Alta. Muchas ofertas de desarrollador front-end piden Vue.	Framework conocido, buen desempeño para SPA, comunidad activa.	Alta competencia frente a React; algunos proyectos todavía usan Vue 2.	USD 114,785 / 9,565 / 441	Senior / intermedio
FastAPI (Backend, Python)	En crecimiento. Muchas ofertas	Moderno, eficiente, tipado,	Menos maduro que frameworks como Django	USD 109,905 / 9,159 / 423	Intermedio / senior

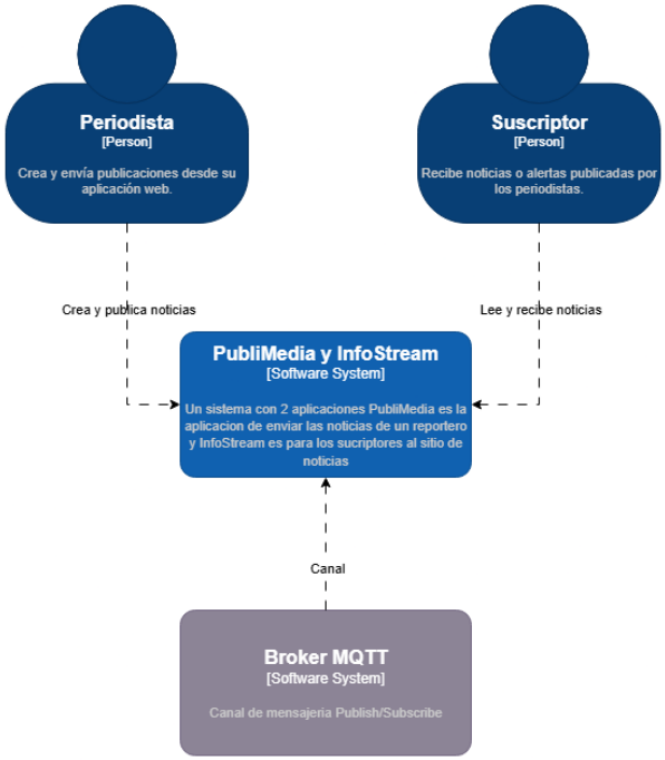
	remotas buscan desarrolladores con experiencia en FastAPI.	documentación automática.	o Flask; menos especialistas con experiencia.		
MySQL (Base de datos relacional)	Muy alta. MySQL sigue siendo usado por muchas empresas de distintos tamaños.	Conocimiento amplio, estabilidad, confiabilidad.	Competencia creciente de PostgreSQL o bases NoSQL según el caso.	USD 46,679 / 3,889 / 180	Junior / intermedio
MQTT (Integración / mensajería IoT)	Especializada, pero con crecimiento en proyectos de IoT e industria.	Muy demandado en el segmento IoT, ideal para comunicación ligera.	Menos oportunidades en proyectos no relacionados con IoT; requiere conocimientos técnicos específicos.	USD 147,514 / 12,293 / 567	Intermedio / senior

Ejemplo práctico – diagramas

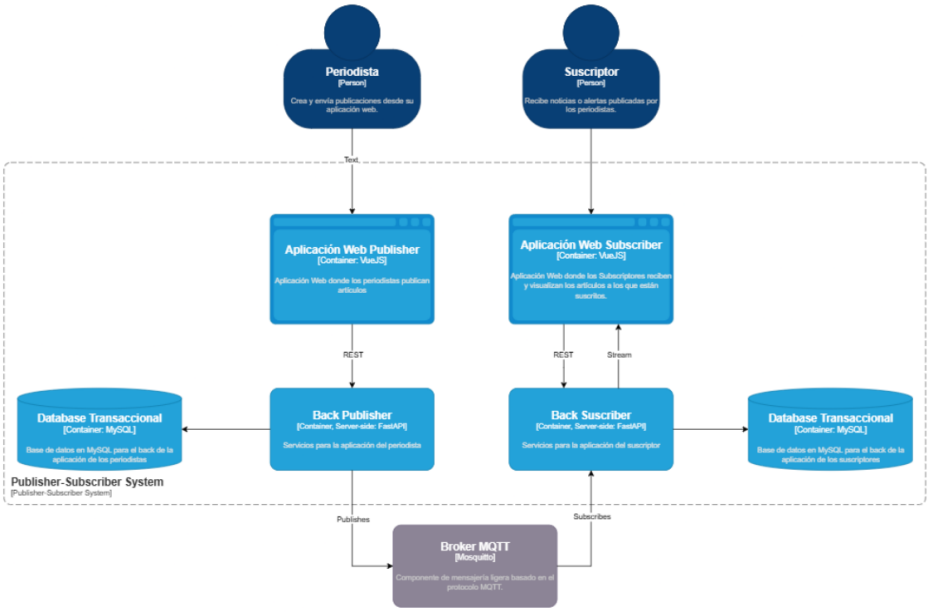
- Diagrama de alto nivel



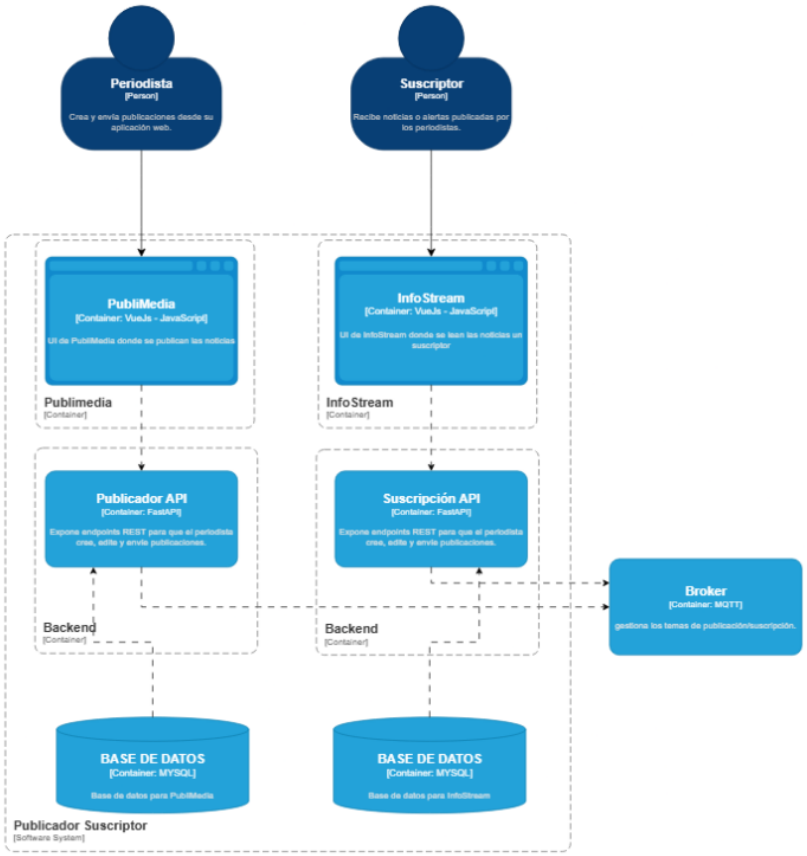
- Diagrama de contexto



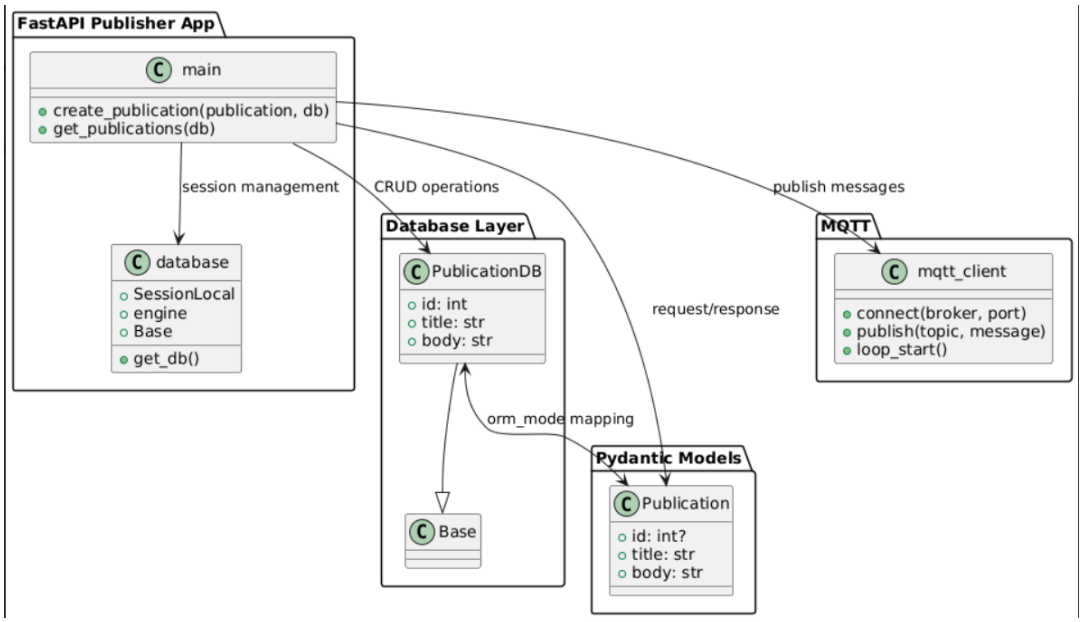
- Diagrama de contenedores



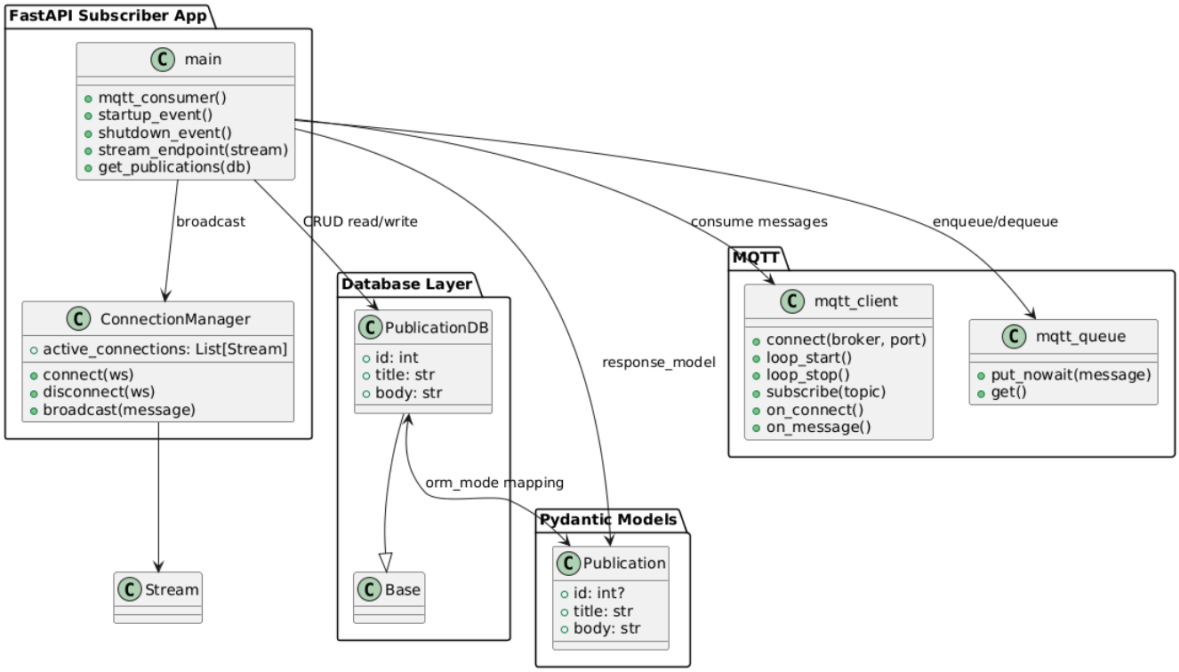
- Diagrama de componentes



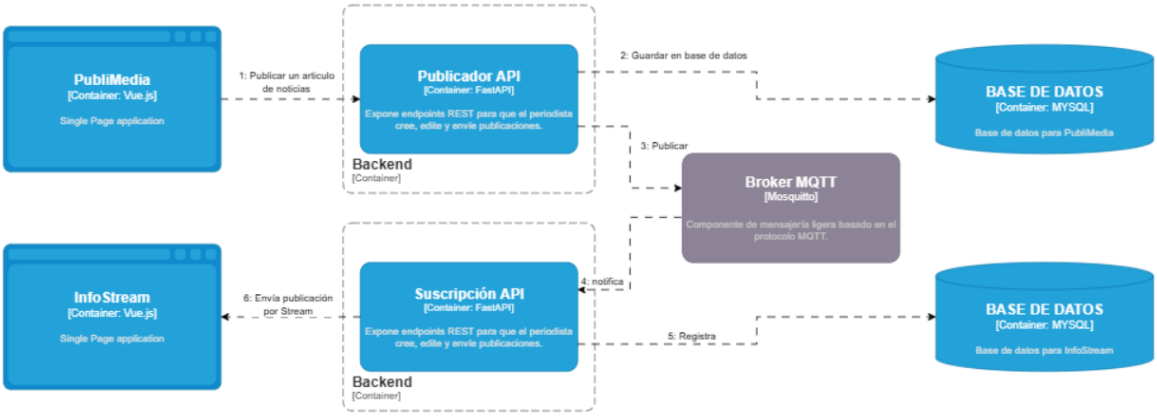
- Diagrama de código – App publicador



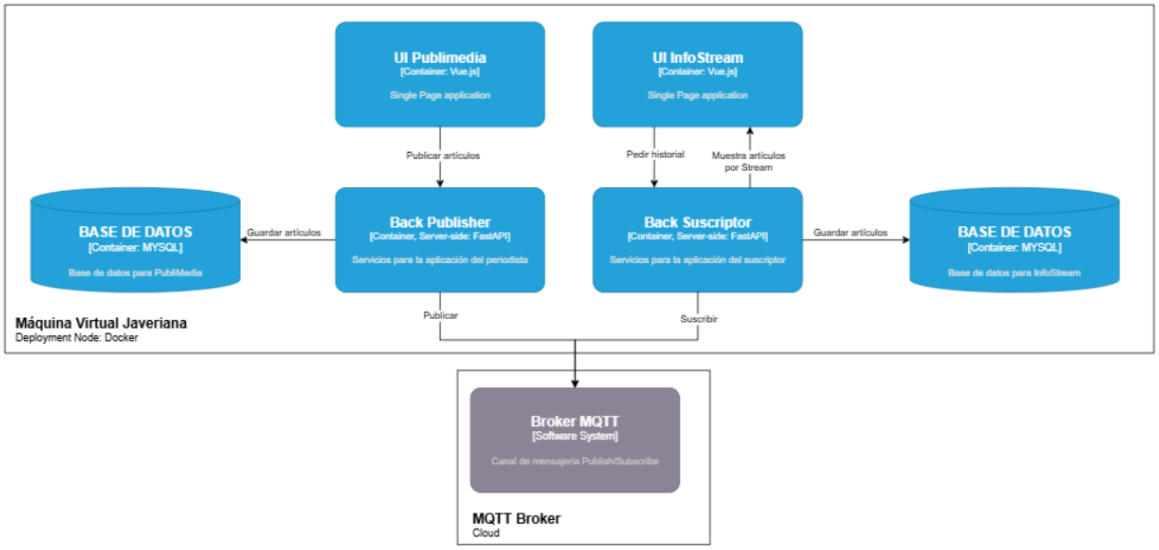
- Diagrama de código – App suscriptor



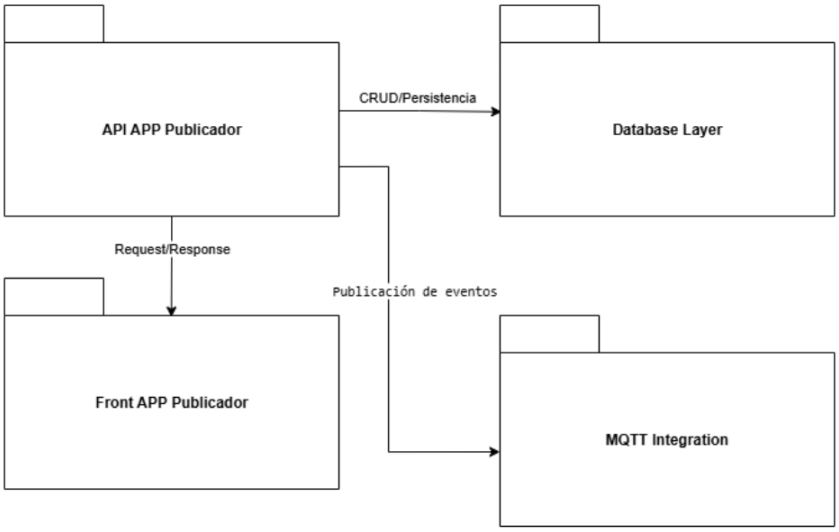
- Diagrama dinámico



- Diagrama de despliegue



- Diagramas de paquetes – Publicador



- Diagramas de paquetes – Suscriptor