El Contrato API REST: Pilar Estratégico en Arquitecturas de Servicios y Microservicios

Introducción: El Contrato como Acuerdo Fundamental en Ecosistemas Distribuidos

En el epicentro de las arquitecturas de software modernas, particularmente aquellas basadas en servicios y microservicios, yace un concepto fundamental que gobierna toda interacción: el contrato de la Interfaz de Programación de Aplicaciones (API). Un contrato API se define como un acuerdo formal y vinculante entre un proveedor de servicios (el servidor) y sus consumidores (los clientes).[1] Este documento, legible tanto por humanos como por máquinas, detalla de manera inequívoca la funcionalidad, el comportamiento esperado y el protocolo de comunicación de la API. Abarca desde los métodos disponibles y los formatos de datos admitidos hasta las entradas esperadas, las respuestas posibles y los códigos de error estandarizados.[1]

El rol de este contrato se vuelve críticamente importante en el contexto de las arquitecturas de microservicios. En estos ecosistemas distribuidos, múltiples equipos desarrollan y despliegan servicios de forma autónoma e independiente.[3] En este paradigma, el contrato API trasciende su función de mera documentación para convertirse en el mecanismo esencial que garantiza la interoperabilidad, el desacoplamiento y la coherencia. Sin un contrato sólido y bien definido, la autonomía de los equipos puede derivar rápidamente en un caos de integración, donde los cambios en un servicio provocan fallos en cascada en otros.[4]

Este informe se adentra en el universo de los contratos API REST para responder dos preguntas fundamentales que guían su diseño e implementación:

1. ¿Cómo se formaliza este acuerdo? Se explorarán los "tipos" de contratos, analizando los lenguajes de especificación que les dan forma.
2. ¿Cuáles son las disciplinas y principios para asegurar que este acuerdo sea de alta calidad? Se investigarán los "métodos" de construcción, abarcando desde metodologías de desarrollo hasta principios de diseño de probada eficacia.

Para comprender plenamente el estado actual de los contratos API, es crucial reconocer su evolución. La arquitectura REST, por su propia naturaleza, define un conjunto de reglas y restricciones que actúan como un *contrato implícito*.[3] Sin embargo, la industria ha transitado progresivamente hacia la creación de artefactos

explícitos y legibles por máquina, como los archivos de especificación OpenAPI. Este cambio no es una mera tendencia de documentación, sino una respuesta directa y necesaria a la complejidad operativa introducida por los microservicios. Inicialmente, seguir los principios REST era suficiente para un nivel básico de previsibilidad.[3] No obstante, con la explosión de arquitecturas compuestas por cientos de servicios desarrollados por equipos dispares, la ambigüedad se volvió intolerable.[4] Se necesitaba una "única fuente de verdad" que permitiera la coordinación sin una comunicación constante y directa. Esta necesidad de escala y automatización impulsó la adopción de contratos explícitos, que pueden ser utilizados para generar documentación, kits de desarrollo de software (SDKs) para clientes, servidores de prueba (
mock servers) y conjuntos de pruebas automatizadas, industrializando así la comunicación entre servicios.[2]

Sección 1. Fundamentos de la Arquitectura REST y su Contrato Implícito

Antes de la existencia de lenguajes de especificación formales, la propia arquitectura de Transferencia de Estado Representacional (REST), definida por Roy Fielding en su tesis doctoral en el año 2000, establecía un conjunto de principios que actuaban como un contrato fundamental e implícito.[3] Comprender estas restricciones arquitectónicas es esencial, ya que constituyen la base sobre la cual se construyen todos los contratos API RESTful, tanto implícitos como explícitos.

Análisis de las Restricciones Arquitectónicas de REST

Los seis principios de diseño de REST no son sugerencias, sino restricciones que, al ser cumplidas, garantizan las propiedades deseadas de un sistema distribuido, como la escalabilidad, la flexibilidad y la fiabilidad.[3]

- **Cliente-Servidor:** Esta restricción establece una separación clara de responsabilidades. El cliente se encarga de la interfaz de usuario y la experiencia, mientras que el servidor gestiona la lógica de negocio y el almacenamiento de datos. Esta separación permite que ambos componentes evolucionen de forma independiente, siempre que la interfaz (el contrato) no se altere.[3]
- **Sin Estado (Stateless):** Esta es una de las cláusulas más críticas del contrato REST. Dicta que cada solicitud enviada desde el cliente al servidor debe contener toda la información necesaria para que el servidor la entienda y la procese. El servidor no debe almacenar ningún contexto de sesión del cliente entre solicitudes.[3] La responsabilidad de mantener el estado de la aplicación recae enteramente en el cliente.
- **Cacheable:** Para mejorar el rendimiento y la escalabilidad, las respuestas del servidor

deben indicar explícitamente si son cacheables o no. Esto permite que los clientes o los intermediarios (como una Red de Entrega de Contenidos o CDN) almacenen y reutilicen respuestas, reduciendo la carga en el servidor.[3]

- **Interfaz Uniforme:** Considerada el pilar central del contrato REST, esta restricción estandariza la forma en que los clientes y servidores interactúan, simplificando la arquitectura general del sistema. Se desglosa en varios sub-principios que se analizarán a continuación.
- **Sistema en Capas:** La arquitectura puede estar compuesta por múltiples capas de intermediarios (por ejemplo, proxies, gateways, balanceadores de carga). El cliente no necesita saber si está comunicándose directamente con el servidor final o con un intermediario. Esto permite una mayor flexibilidad arquitectónica para la seguridad, el balanceo de carga y el almacenamiento en caché.[3]
- **Código bajo demanda (Opcional):** Esta restricción opcional permite al servidor extender la funcionalidad del cliente enviando código ejecutable, como scripts o applets de Java, que se ejecutan a petición.[3]

La restricción "sin estado" es a menudo vista como una simple limitación técnica, pero sus implicaciones son profundas y estratégicas, especialmente en la era de la computación en la nube. Un servicio que mantiene estado (*stateful*) requiere que las solicitudes sucesivas de un mismo cliente sean dirigidas al mismo servidor donde reside su sesión. Esto complica enormemente el balanceo de carga y la recuperación ante fallos; si un servidor falla, la sesión del usuario se pierde. La arquitectura sin estado elimina esta dependencia. Al contener cada solicitud todo el contexto necesario, cualquier instancia del servidor puede procesarla en cualquier momento. Este principio es la base de la escalabilidad horizontal elástica, un pilar fundamental de las arquitecturas nativas de la nube. Por lo tanto, la cláusula de "sin estado" en el contrato REST no es un capricho académico, sino una precondición arquitectónica que habilita directamente los patrones de despliegue y operación más eficientes y resilientes de la infraestructura moderna. En esencia, el contrato dicta la topología de la infraestructura.

La Interfaz Uniforme como Cláusula Maestra

La interfaz uniforme es lo que realmente distingue a REST de otros estilos arquitectónicos. Impone un conjunto de reglas genéricas para la interacción, desacoplando la implementación del servicio de la forma en que se expone. Sus componentes son:

- **Identificación de Recursos (URIs):** Cada recurso (un concepto abstracto como un usuario, un producto o un pedido) debe ser identificable de forma única a través de un Identificador Uniforme de Recursos (URI).[3] La URI es la "dirección" del recurso.
- **Manipulación de Recursos a través de Representaciones:** Los clientes no interactúan directamente con el recurso en el servidor, sino con una representación de su estado. Los formatos de representación más comunes son JSON y XML.[9] El cliente envía una representación para modificar el estado del recurso y recibe una representación para conocer su estado actual.

- **Mensajes Autodescriptivos:** Cada mensaje (solicitud o respuesta) contiene suficiente información para describir cómo debe ser procesado. Esto incluye el método HTTP (la acción a realizar), la URI del recurso, las cabeceras (metadatos como el tipo de contenido) y, opcionalmente, un cuerpo de mensaje.[9]
- **Hipermedia como Motor del Estado de la Aplicación (HATEOAS):** Este es el principio más maduro y a menudo el menos implementado de REST. Sugiere que una respuesta del servidor no solo debe contener los datos del recurso solicitado, sino también hipervínculos (enlaces) a otras acciones o recursos relacionados que el cliente puede explorar a continuación.[8] Esto permite que el cliente navegue por la API de forma dinámica, descubriendo funcionalidades sin necesidad de tener las URIs codificadas de antemano.

Sección 2. El Contrato API Explícito: Definición, Propósito y Componentes Clave

Si bien los principios de REST proporcionan una base sólida, la creciente complejidad de los sistemas distribuidos ha impulsado la necesidad de formalizar el contrato en un artefacto explícito. Este contrato, legible tanto por humanos como por máquinas, se convierte en la "única fuente de verdad" que gobierna la interacción entre el proveedor y el consumidor de la API.[1]

Propósito y Beneficios Estratégicos

La adopción de contratos explícitos va más allá de la simple documentación; es una herramienta estratégica que ofrece beneficios tangibles a lo largo de todo el ciclo de vida del desarrollo de software.

- **Colaboración y Entendimiento Mejorados:** Un contrato claro y conciso sirve como un lenguaje común para todos los equipos involucrados: desarrolladores de frontend, backend, ingenieros de calidad (QA), arquitectos y gerentes de producto. Esto asegura una alineación completa sobre la funcionalidad y el comportamiento esperado de la API.[2]
- **Desarrollo en Paralelo:** Con un contrato definido, los equipos de frontend y backend pueden trabajar simultáneamente. Los desarrolladores de frontend pueden construir sus interfaces utilizando servidores *mock* que se generan automáticamente a partir del contrato, sin tener que esperar a que la implementación del backend esté completa.[12]
- **Validación y Pruebas Automatizadas:** El contrato actúa como una especificación formal contra la cual se puede validar la implementación. Permite la generación automática de conjuntos de pruebas de conformidad, que verifican que la API se comporte exactamente como se describe en el contrato, detectando desviaciones de

forma temprana.[2]

- **Reducción de la Deuda Técnica:** Al forzar un diseño y un acuerdo previos, se minimizan las inconsistencias, los malentendidos y los costosos retrabajos posteriores. Cumplir con el contrato reduce la probabilidad de construir funcionalidades que no se alinean con la intención original.[2]
- **Generación de Documentación y SDKs:** El contrato es la fuente para generar automáticamente documentación interactiva y legible para los desarrolladores, así como SDKs en diversos lenguajes de programación. Esto acelera drásticamente el proceso de integración para los consumidores de la API.[6]
- **Crecimiento del Ecosistema:** Para las APIs públicas o de partners, un contrato claro, bien documentado y estable es esencial. Facilita que desarrolladores de terceros se integren con la plataforma, fomentando la innovación y la creación de un ecosistema saludable alrededor del servicio.[6]

Anatomía de un Contrato Explícito

Para entender cómo un contrato logra estos beneficios, es útil desglosar sus componentes principales, utilizando la estructura de la Especificación OpenAPI como un modelo de referencia.

- **Información General (Info Object):** Contiene metadatos esenciales sobre la API, como el título, una descripción detallada, la versión de la API (ej. "1.0.0"), términos de servicio y la información de licencia.[13]
- **Servidores (Servers):** Una lista de las URLs base donde la API está alojada. Esto permite definir diferentes entornos, como desarrollo, staging y producción.[7]
- **Rutas (Paths) y Operaciones (Operations):** Este es el corazón del contrato. Define todos los endpoints disponibles (ej. /users/{id}) y las operaciones (métodos HTTP como GET, POST, PUT, DELETE) que se pueden realizar en cada uno de ellos.[7]
- **Parámetros (Parameters):** Describe todos los parámetros que una operación puede aceptar. Esto incluye su nombre, ubicación (en la ruta, en la consulta/query, en la cabecera o en una cookie), el tipo de dato (string, integer, etc.), si es obligatorio y otras restricciones de validación.[6]
- **Cuerpos de Solicitud (Request Bodies):** Para operaciones como POST, PUT y PATCH, esta sección define la estructura y el esquema de los datos que se deben enviar en el cuerpo de la solicitud. Especifica el tipo de medio (ej. application/json) y una referencia a un esquema de datos.[6]
- **Respuestas (Responses):** Para cada operación, se deben definir todas las respuestas posibles. Cada respuesta se asocia con un código de estado HTTP (ej. 200, 404, 500). Para cada código, se describe el significado de la respuesta y, si aplica, la estructura del cuerpo de la respuesta, incluyendo su tipo de medio y esquema.[6]
- **Esquemas (Schemas):** Son definiciones reutilizables de las estructuras de datos (generalmente objetos JSON) que se utilizan en los cuerpos de solicitud y respuesta.

Definir esquemas promueve la consistencia y reduce la duplicación en la especificación.[10]

- **Seguridad (Security Schemes):** Declara los mecanismos de autenticación y autorización que la API utiliza. Esto puede incluir esquemas como claves API, autenticación HTTP básica, OAuth 2.0 o JSON Web Tokens (JWT). La especificación indica qué esquemas se aplican a nivel global o a operaciones específicas.[6]

Sección 3. Tipos de Contratos API REST: Un Análisis Comparativo de Lenguajes de Especificación

Aunque la arquitectura REST en sí misma no impone un Lenguaje de Definición de Interfaz (IDL) oficial, la necesidad de formalizar contratos explícitos ha llevado al surgimiento de varios estándares de facto.[14] La elección de un lenguaje de especificación es una decisión tecnológica fundamental que impacta todo el ciclo de vida de la API, desde el diseño hasta el mantenimiento.[15] En la "guerra de estándares" de la década de 2010, tres contendientes principales emergieron: OpenAPI, RAML y API Blueprint.[16]

Análisis Detallado de los Principales Contendientes

OpenAPI Specification (OAS)

- **Historia y Dominio:** Originalmente conocido como Swagger, OpenAPI es ahora un proyecto de código abierto bajo el gobierno de la Linux Foundation.[16] Se ha consolidado como el estándar de la industria por un margen abrumador, con una adopción que supera con creces a la de sus competidores.[16] Su éxito se atribuye en gran medida a su naturaleza agnóstica a las herramientas y a un ecosistema de código abierto vibrante y extenso que creció a su alrededor.[16]
- **Estructura y Sintaxis:** Utiliza formatos YAML o JSON para describir la API de una manera exhaustiva y, a veces, verbosa.[7] Su enfoque se centra en una descripción completa que puede impulsar la generación automática de documentación, SDKs de cliente, stubs de servidor y pruebas, fomentando un proceso de desarrollo iterativo e integrado.[17]

RAML (RESTful API Modeling Language)

- **Filosofía de Diseño:** Desarrollado por MuleSoft (ahora parte de Salesforce), RAML fue concebido con un fuerte enfoque en el "modelado" de la API.[16] Promueve activamente la reutilización y los principios de No Repetir (DRY) a través de conceptos poderosos como
  traits (comportamientos reutilizables como la paginación) y *resource types* (plantillas para recursos).[7] Su sintaxis, basada en YAML, está diseñada para ser concisa y altamente legible por humanos.[17]
- **Modelado vs. Descripción:** La distinción conceptual clave que sus proponentes hacían era que RAML era para *modelar* (diseñar el plano de la API), mientras que OpenAPI era para *describir* (documentar una API, posiblemente ya existente). En la práctica, sin embargo, ambas especificaciones pueden usarse para ambos propósitos, y la adopción masiva de OpenAPI ha llevado a que se utilice tanto para el diseño como para la documentación.[16]

API Blueprint

- **Enfoque en la Documentación:** Creado por Apiary (adquirida por Oracle), API Blueprint adopta un enfoque radicalmente simple.[16] Utiliza una sintaxis similar a Markdown llamada MSON (Markdown Syntax for Object Notation), lo que lo hace extremadamente accesible y fácil de aprender, especialmente para roles no puramente técnicos.[18] Su objetivo principal es la creación de documentación clara y legible por humanos.[18]
- **Limitaciones:** La simplicidad de API Blueprint tiene un costo. Su ecosistema de herramientas es considerablemente menos robusto que el de OpenAPI, especialmente para casos de uso avanzados como la generación de código complejo o la integración con gateways de API y herramientas de prueba sofisticadas.[7]

El desenlace de la "guerra de estándares" ofrece una lección crucial sobre la adopción de tecnología. Aunque RAML presentaba conceptos de diseño elegantes y API Blueprint una simplicidad atractiva, OpenAPI prevaleció de manera decisiva. Esto no se debió necesariamente a una superioridad técnica intrínseca, sino al poderoso "efecto de red" de su ecosistema de herramientas. El éxito de un estándar depende menos de sus méritos aislados y más del valor que se genera a medida que más usuarios y herramientas lo adoptan. Swagger (el precursor de OpenAPI) fue desarrollado de forma independiente de una plataforma de herramientas específica, lo que fomentó la creación de un ecosistema abierto y diverso desde sus inicios.[16] En contraste, RAML y API Blueprint estaban estrechamente ligados a sus empresas creadoras (MuleSoft y Apiary), lo que generó en la comunidad una percepción de posible "vendor lock-in" (dependencia del proveedor) y disuadió a los competidores de construir herramientas compatibles.[16] El vasto ecosistema de OpenAPI —que abarca generadores de código, gateways de API, plataformas de gestión, herramientas de prueba y más— creó un ciclo virtuoso: más herramientas atrajeron a más usuarios, lo que a su vez incentivó la creación de más herramientas, solidificando su posición como el estándar

de facto. Este fenómeno demuestra que, al elegir una tecnología de estandarización, la fortaleza y la apertura del ecosistema circundante son predictores de éxito y viabilidad a largo plazo más importantes que las características del propio estándar. La interoperabilidad y el soporte de la comunidad triunfan sobre la elegancia técnica aislada.

Tabla 1: Comparativa de Lenguajes de Especificación API

Característica	OpenAPI (OAS)	RAML	API Blueprint
Sintaxis Primaria	JSON / YAML [7]	YAML [17]	Markdown / MSON [18]
Filosofía Principal	Descripción exhaustiva e integrada [17]	Modelado y reutilización (DRY) [17]	Documentación y simplicidad [18]
Curva de Aprendizaje	Moderada a alta	Moderada	Baja [18]
Ecosistema de Herramientas	Muy Extenso (Estándar de facto) [7]	Moderado (Principalmente MuleSoft) [17]	Limitado [18]
Adopción en la Industria	Dominante (~55%) [16]	Nicho (~7%) [16]	Nicho (~7%) [16]
Caso de Uso Ideal	Proyectos que requieren un ciclo de vida completo (diseño, código, pruebas, docs), integración con múltiples herramientas.	Proyectos dentro del ecosistema MuleSoft, donde la reutilización del diseño es primordial.	Prototipado rápido, proyectos donde la documentación es el principal objetivo y la simplicidad es clave.

Sección 4. Metodologías de Construcción: El Paradigma API-First vs. Code-First

Más allá de la elección del lenguaje de especificación, existe una dicotomía fundamental en el proceso de desarrollo de APIs que define cuándo y cómo se crea el contrato: el enfoque *API-first* (o *design-first*) frente al enfoque *code-first*.[20] Esta elección tiene profundas implicaciones en la colaboración del equipo, la calidad del producto final y la agilidad del desarrollo.

Enfoque Code-First (El Contrato como un Subproducto)

Este es el enfoque más tradicional. El ciclo de desarrollo comienza directamente con la escritura del código de la aplicación y la lógica de negocio.[21] La especificación de la API y su

documentación se generan en una etapa posterior, a menudo de forma automática a partir de anotaciones o comentarios en el código fuente.[20]

- **Ventajas:** Su principal beneficio es la velocidad inicial. Permite a los desarrolladores comenzar a codificar inmediatamente, lo que es ideal para el prototipado rápido y proyectos donde los requisitos son volátiles o poco claros al principio. Ofrece una gran flexibilidad para realizar cambios sobre la marcha sin estar atado a un contrato predefinido.[12]
- **Desventajas:** Este enfoque a menudo conduce a APIs inconsistentes, ya que diferentes partes de la API pueden ser diseñadas por diferentes desarrolladores sin una guía unificada. La documentación tiende a ser un pensamiento tardío (*afterthought*), resultando incompleta o desactualizada. Además, crea un fuerte acoplamiento entre la interfaz de la API y su implementación subyacente, lo que dificulta la evolución independiente. Finalmente, obstaculiza el desarrollo en paralelo, ya que los equipos consumidores deben esperar a que la implementación esté lista para poder integrarse.[12]

Enfoque API-First (Design-First)

En este paradigma moderno, el ciclo de vida del software se invierte. El proceso comienza con el diseño y la definición colaborativa del contrato de la API antes de escribir una sola línea de código de implementación.[12] Este contrato, una vez acordado por todas las partes interesadas (desarrolladores, arquitectos, gerentes de producto), se convierte en el plano maestro que guía todo el desarrollo posterior.

- **Ventajas:** Fomenta la colaboración y el alineamiento desde las primeras etapas, asegurando que la API satisfaga las necesidades de sus consumidores. El contrato estable resultante sirve como base para la generación automática de servidores *mock*, SDKs de cliente y pruebas de conformidad, lo que permite un desarrollo verdaderamente en paralelo.[6] Este enfoque conduce a APIs de mayor calidad, más consistentes y mejor documentadas, ya que el diseño es una actividad deliberada y no un subproducto accidental.[23]
- **Desventajas:** Requiere una inversión de tiempo inicial en la fase de diseño, lo que puede percibirse como un retraso en el inicio de la codificación. Puede ser menos ágil si los requisitos son extremadamente inciertos, ya que los cambios en un contrato ya acordado pueden requerir una nueva negociación entre las partes.[21]

La adopción de un enfoque API-First representa más que un simple cambio en el flujo de trabajo técnico; es un cambio cultural profundo. El enfoque Code-First tiende a tratar la API como una característica secundaria de una aplicación específica; el objetivo principal es "hacer que la aplicación funcione". Por el contrario, el enfoque API-First eleva la API al estatus de un producto de primera clase, con su propio ciclo de vida, sus propios consumidores y su propio valor de negocio.[23] Este cambio de perspectiva obliga a una organización a pensar en sus capacidades de negocio como un conjunto de servicios componibles y reutilizables, en lugar de como aplicaciones monolíticas y aisladas. Exige una colaboración más estrecha y

temprana entre los roles de negocio y técnicos para definir qué "capacidades" debe ofrecer la API, en lugar de qué "funciones" necesita una aplicación en particular. Por lo tanto, adoptar API-First es una decisión estratégica que impulsa a la organización hacia un modelo de pensamiento de "plataforma", fomentando la reutilización, la agilidad y la creación de nuevo valor a partir de las capacidades existentes. Es un claro indicador de madurez organizacional en la era digital.

Tabla 2: Comparativa de Enfoques de Desarrollo

Criterio	Enfoque API-First (Design-First)	Enfoque Code-First
Punto de Partida	El Contrato API [23]	El Código de Implementación [20]
Colaboración	Alta, fomenta el alineamiento temprano entre equipos [12]	Baja, a menudo aislada al equipo de backend inicialmente [12]
Velocidad Inicial	Más lenta, requiere diseño previo [22]	Más rápida, ideal para prototipado [21]
Calidad y Consistencia	Alta, el contrato impone un estándar [23]	Variable, riesgo de inconsistencias [21]
Calidad de la Documentación	Alta, es un producto del diseño [12]	A menudo un *afterthought*, riesgo de estar incompleta o desactualizada [12]
Escenario Ideal	APIs públicas o para partners, sistemas complejos con múltiples equipos, arquitecturas de microservicios a escala [12]	Prototipos, APIs internas pequeñas, equipos pequeños y ágiles con requisitos poco claros [12]

Sección 5. Principios y Mejores Prácticas para el Diseño de un Contrato API REST Robusto

La creación de un contrato API de alta calidad no es un arte, sino una disciplina de ingeniería que sigue un conjunto de principios y mejores prácticas bien establecidos. Esta sección detalla las reglas y patrones esenciales para diseñar contratos que sean claros, consistentes, escalables y fáciles de usar para los desarrolladores.

5.1. Diseño de Endpoints y URIs: La Semántica de los Recursos

La URI es la dirección de un recurso y su diseño debe ser intuitivo y predecible.

- **Usar Sustantivos, no Verbos:** Las URIs deben identificar recursos, que son "cosas" (sustantivos). La acción a realizar sobre ese recurso ya está definida por el método HTTP (verbo). Por ejemplo, para obtener una lista de usuarios, la URI correcta es GET /users, no GET /getAllUsers.[11]
- **Usar Sustantivos en Plural para Colecciones:** Para mantener la consistencia y la intuición, los endpoints que representan una colección de recursos deben usar sustantivos en plural. Un recurso individual dentro de esa colección se identifica añadiendo su ID. Por ejemplo, /users representa la colección de todos los usuarios, mientras que /users/123 representa al usuario específico con el ID 123.[11]
- **Jerarquías Lógicas para Relaciones:** La estructura de la URI puede usarse para expresar relaciones jerárquicas o de anidamiento entre recursos. Por ejemplo, para obtener todos los pedidos de un cliente específico, una URI intuitiva sería /customers/123/orders. Sin embargo, es crucial no abusar de esta anidación. Una profundidad mayor a coleccion/item/coleccion puede volverse difícil de manejar y frágil ante cambios en el modelo de datos.[11]
- **Consistencia en la Nomenclatura:** Para mejorar la legibilidad, se debe adoptar una convención de nomenclatura consistente. La práctica recomendada es usar caracteres en minúscula y separar las palabras con guiones (-), conocido como *kebab-case*. Se deben evitar los guiones bajos (_) o el uso de *camelCase*, ya que pueden causar problemas de visibilidad o interpretación en algunos sistemas.[29]

5.2. Uso Semántico de Métodos y Códigos de Estado HTTP

El protocolo HTTP proporciona las herramientas semánticas para interactuar con los recursos definidos por las URIs. Su uso correcto es fundamental para un contrato claro.

- **Mapeo de Operaciones CRUD:** Cada método HTTP tiene un propósito específico y propiedades de seguridad e idempotencia que deben ser respetadas.
    - GET: Se utiliza exclusivamente para recuperar datos. Es una operación segura (no debe modificar el estado del servidor) e idempotente (múltiples llamadas idénticas producen el mismo resultado).[3]
    - POST: Se utiliza para crear un nuevo recurso subordinado a una colección. No es seguro ni idempotente (múltiples llamadas crearán múltiples recursos).[3]
    - PUT: Se utiliza para reemplazar completamente un recurso existente en una URI conocida. Es idempotente (múltiples llamadas con el mismo cuerpo tendrán el mismo efecto que una sola: el recurso quedará en ese estado final).[3]
    - DELETE: Se utiliza para eliminar un recurso. Es idempotente (una vez que un recurso es eliminado, las llamadas posteriores para eliminarlo seguirán resultando

en el mismo estado: el recurso no existe).[3]
    - PATCH: Se utiliza para aplicar una actualización parcial a un recurso. No es universalmente idempotente; su idempotencia depende de la naturaleza de la operación de parcheo.[5]
- **Comunicación de Resultados con Códigos de Estado:** Los códigos de estado HTTP son un mecanismo estandarizado para que el servidor comunique el resultado de la solicitud del cliente. Usar el código correcto es esencial para que los clientes puedan manejar las respuestas de forma programática y predecible.[5]

Tabla 3: Uso Semántico de Códigos de Estado HTTP Comunes

Rango	Código	Significado	Caso de Uso Común en REST
2xx (Éxito)	200 OK	La solicitud fue exitosa.	Respuesta estándar para un GET exitoso.
	201 Created	El recurso fue creado exitosamente.	Respuesta a un POST o PUT que resultó en la creación de un nuevo recurso. Debe incluir una cabecera Location con la URI del nuevo recurso.
	204 No Content	La solicitud fue exitosa pero no hay contenido que devolver.	Respuesta común para un DELETE exitoso o un PUT que no devuelve el recurso actualizado.
3xx (Redirección)	301 Moved Permanently	El recurso ha sido movido permanentemente a una nueva URI.	Para indicar que un endpoint ha cambiado de ubicación de forma definitiva.
4xx (Error del Cliente)	400 Bad Request	La solicitud es malformada o inválida.	Errores de sintaxis en el JSON, parámetros faltantes, fallos de validación de datos.
	401 Unauthorized	La solicitud requiere autenticación.	El cliente no ha proporcionado credenciales válidas o no se ha autenticado.
	403 Forbidden	El servidor entiende la solicitud, pero se niega	El cliente está autenticado pero no

		a autorizarla.	tiene los permisos necesarios para acceder al recurso.
	404 Not Found	El recurso solicitado no pudo ser encontrado.	La URI no corresponde a ningún recurso existente.
	409 Conflict	La solicitud no se pudo completar debido a un conflicto con el estado actual del recurso.	Intentar crear un recurso que ya existe (ej. con un email que debe ser único).
5xx (Error del Servidor)	500 Internal Server Error	Ocurrió un error inesperado en el servidor.	Un error genérico para problemas no controlados en el backend (ej. una excepción no capturada).

Fuentes: [5]

5.3. Estructura del Payload: Paginación y Manejo de Errores

El diseño del cuerpo (payload) de las respuestas JSON es tan importante como el de las URIs y los métodos.

- **Paginación para Grandes Conjuntos de Datos:** Cuando un endpoint puede devolver una gran cantidad de resultados, es crucial implementar la paginación para evitar respuestas masivas que degraden el rendimiento y consuman un ancho de banda excesivo.[35]
    - **Paginación por Offset (limit/offset):** Es la estrategia más simple. El cliente especifica cuántos ítems quiere (limit) y cuántos quiere saltar desde el principio (offset). Es fácil de implementar y permite al usuario saltar a cualquier página. Sin embargo, su rendimiento se degrada significativamente en conjuntos de datos muy grandes (la base de datos debe escanear y descartar todos los registros del offset), y es susceptible a inconsistencias si se añaden o eliminan datos mientras el usuario pagina.[36]
    - **Paginación por Cursor (Keyset):** Esta estrategia más avanzada utiliza un "cursor", que es un puntero opaco a un ítem específico en el conjunto de resultados. Para obtener la siguiente página, el cliente envía el cursor del último ítem que recibió. El servidor entonces busca los ítems que vienen *después* de ese cursor. Es mucho más eficiente para grandes volúmenes de datos y es inmune a los problemas de inconsistencia, lo que la hace ideal para feeds de

desplazamiento infinito y datos en tiempo real. Su principal desventaja es que no permite saltar a una página arbitraria.[35]

    - **Estructura de la Respuesta Paginada:** La respuesta no solo debe contener la lista de ítems de la página actual, sino también metadatos que guíen al cliente, como el número total de resultados, el número total de páginas (para offset), o un next_cursor y un booleano has_next_page (para cursor).[35]
- **Manejo de Errores Consistente:** Devolver respuestas de error inconsistentes o poco informativas es un antipatrón común que frustra a los desarrolladores. La mejor práctica es estandarizar el formato de las respuestas de error.
    - **RFC 7807/9457 "Problem Details for HTTP APIs":** Este estándar del IETF se ha convertido en la referencia para el manejo de errores en APIs HTTP.[38] Define un tipo de medio (application/problem+json) y un objeto JSON estándar para comunicar los detalles de un error. Los campos clave incluyen [39]:
        - type: Un URI que identifica el tipo de problema. Idealmente, este URI debería resolver a una documentación legible por humanos sobre ese error.
        - title: Un resumen corto y legible del tipo de problema.
        - status: El código de estado HTTP asociado con este error.
        - detail: Una explicación legible y específica de esta ocurrencia del problema.
        - instance: Un URI que identifica la ocurrencia específica del problema, útil para el registro y la depuración.
    - **Beneficios:** La adopción de este estándar proporciona respuestas de error predecibles y legibles por máquina, lo que permite a los clientes construir una lógica de manejo de errores genérica y robusta. Grandes proveedores como Google y Microsoft, aunque a veces usan sus propios esquemas, siguen principios similares de proporcionar errores estructurados y detallados.[41]

5.4. Estrategias de Versionado para la Evolución de la API

Las APIs no son estáticas; evolucionan con el tiempo para añadir nuevas funcionalidades o mejorar las existentes. El versionado es la estrategia para gestionar estos cambios, especialmente aquellos que rompen la compatibilidad con versiones anteriores (*breaking changes*), sin afectar a los clientes existentes.[26]

- **Versionado en la URI:** (ej. /api/v1/users). Este es el enfoque más común, explícito y fácil de entender para los consumidores. Es sencillo de implementar en enrutadores y gateways, y es amigable con el almacenamiento en caché del navegador. Su principal crítica desde una perspectiva purista de REST es que viola el principio de que una URI debe identificar un recurso de forma atemporal; con este método, /v1/users/123 y /v2/users/123 son vistas como dos recursos diferentes en lugar de dos representaciones del mismo recurso.[45]
- **Versionado en Cabeceras HTTP:** (ej. Accept-Version: v1). En este enfoque, la URI del

recurso permanece constante (ej. /api/users/123), y el cliente solicita una versión específica a través de una cabecera HTTP personalizada. Es considerado más "puro" desde el punto de vista de REST. Sin embargo, hace que la API sea menos explorable directamente desde un navegador, puede complicar el almacenamiento en caché y requiere que los clientes estén programados para enviar la cabecera correcta.[45]

- **Versionado por Tipo de Medio (Content Negotiation):** (ej. Accept: application/vnd.myapi.v1+json). Este método utiliza la cabecera Accept estándar de HTTP para que el cliente negocie la versión de la representación que desea recibir. Es el enfoque más académicamente correcto según los principios de HATEOAS, pero también es el más complejo de implementar tanto para el proveedor como para el consumidor, y su adopción en la práctica es menos común.[45]

5.5. Seguridad del Contrato: Autenticación y Autorización

La seguridad debe ser una parte integral del diseño del contrato, no una ocurrencia tardía. El contrato debe especificar claramente cómo los clientes deben proteger sus solicitudes.

- **Diferenciación Clave:** Es crucial distinguir entre dos conceptos de seguridad:
  - **Autenticación:** El proceso de verificar la identidad de un cliente ("¿quién eres?").[49]
  - **Autorización:** El proceso de determinar si un cliente autenticado tiene los permisos necesarios para realizar una acción específica ("¿qué tienes permitido hacer?").[49]
- **Estándares de la Industria:** Los contratos modernos deben basarse en estándares abiertos y probados para la seguridad.
  - **OAuth 2.0:** Es el framework estándar de la industria para la autorización delegada. Permite que una aplicación obtenga acceso a los recursos de un usuario sin necesidad de manejar sus credenciales directamente. El flujo authorization_code es uno de los más comunes y seguros para aplicaciones web y móviles.[50]
  - **JSON Web Tokens (JWT):** Es un estándar para crear tokens de acceso que contienen *claims* (afirmaciones) de forma compacta y autocontenida. Un JWT puede ser firmado digitalmente para verificar su autenticidad e integridad. Es comúnmente utilizado como el formato del token de acceso en flujos de OAuth 2.0, permitiendo una autenticación y autorización sin estado.[50]
- **Definición en el Contrato:** Un contrato OpenAPI bien definido incluye una sección de securitySchemes donde se declaran los métodos de seguridad utilizados (ej. apiKey, http, oauth2). Luego, las operaciones individuales o la API en su totalidad pueden especificar qué esquemas de seguridad se les aplican, informando a los clientes sobre cómo deben autenticar y autorizar sus solicitudes.

Sección 6. Análisis de Caso Práctico: La API de Stripe como Modelo de Excelencia

Para ilustrar cómo se aplican estos principios en el mundo real, se analizará la API de Stripe, ampliamente reconocida en la industria como un estándar de oro en diseño de API por su claridad, consistencia y excelente experiencia para el desarrollador (DX).[54]

Deconstrucción de un Diseño de Clase Mundial

El diseño de la API de Stripe demuestra una aplicación pragmática y efectiva de las mejores prácticas discutidas.[54]

- **Estructura de URI:** Las URIs de Stripe son un ejemplo de libro de texto de diseño orientado a recursos. Son predecibles, jerárquicas y utilizan sustantivos en plural. Por ejemplo, https://api.stripe.com/v1/customers representa la colección de clientes, y https://api.stripe.com/v1/customers/{customer_id} se refiere a un cliente específico.[54]
- **Uso de Métodos HTTP y Códigos de Estado:** Stripe mapea rigurosamente las operaciones CRUD a los verbos HTTP correspondientes. Su uso de los códigos de estado HTTP es exhaustivo y semánticamente correcto, proporcionando una retroalimentación clara e inequívoca para cada solicitud, ya sea exitosa o fallida.[54]
- **Formato de Respuestas de Error:** Aunque no implementa RFC 7807 de forma literal, el formato de sus objetos de error JSON comparte la misma filosofía. Cada respuesta de error es un objeto estructurado que contiene campos útiles para la depuración, como type (ej. card_error), code (un código de error específico como card_declined), param (el parámetro de la solicitud que causó el error) y un message legible para el desarrollador. Esto permite un manejo de errores programático y robusto.[54]
- **Estrategia de Versionado:** Stripe utiliza el enfoque de versionado en la URI (/v1/). Esta elección pragmática demuestra la prevalencia de este método en la industria, a pesar de las críticas teóricas. Prioriza la claridad, la facilidad de uso para los desarrolladores y la simplicidad del almacenamiento en caché sobre la pureza académica.[54]
- **Autenticación:** El mecanismo de autenticación es simple y seguro, basado en claves API secretas que se proporcionan en la cabecera de la solicitud. Distinguen claramente entre claves de prueba (sk_test_...) y claves de producción (sk_live_...), lo que facilita un ciclo de desarrollo seguro.[54]

El análisis de la API de Stripe revela una lección importante: el pragmatismo a menudo triunfa sobre la pureza académica en el diseño de APIs exitosas. La elección de Stripe de usar versionado en la URI, una práctica que algunos puristas de REST critican, es un claro ejemplo. Teóricamente, el versionado en cabeceras podría ser más "correcto".[46] Sin embargo, la versión en la URI es innegablemente más fácil de explorar, depurar, compartir y cachear para la gran mayoría de los desarrolladores.[46] Stripe priorizó la experiencia del desarrollador (DX) y

la practicidad sobre la adherencia dogmática a un principio. De manera similar, su modelo de error, aunque propietario, cumple todos los objetivos de un buen sistema de errores: es estructurado, predecible y sumamente útil. El éxito de una API en el mundo real no se mide por su conformidad absoluta con los ideales teóricos, sino por su usabilidad, claridad y la facilidad con la que los desarrolladores pueden integrarla para resolver problemas de negocio. Las mejores APIs son aquellas que toman decisiones de diseño deliberadas y pragmáticas, equilibrando los principios arquitectónicos con las necesidades prácticas de sus consumidores.

Conclusión y Recomendaciones Estratégicas

El recorrido desde el contrato implícito de los principios REST hasta los contratos explícitos y legibles por máquina definidos por especificaciones como OpenAPI ha sido una evolución necesaria, impulsada por la complejidad de las arquitecturas de software modernas. El contrato API ha dejado de ser un mero documento para convertirse en un artefacto de diseño estratégico, un activo fundamental que habilita la agilidad organizacional, la colaboración efectiva entre equipos y la escalabilidad técnica de los sistemas distribuidos.
La investigación ha demostrado que no existe un único "tipo" de contrato, sino un estándar dominante, OpenAPI, cuyo éxito se debe más a la fortaleza de su ecosistema de herramientas que a una superioridad intrínseca. Del mismo modo, la "mejor" manera de construir un contrato no reside en una única metodología, sino en la adopción de un enfoque API-First y la aplicación disciplinada de un conjunto de mejores prácticas de diseño.
Basado en este análisis exhaustivo, se proponen las siguientes recomendaciones estratégicas para cualquier organización que busque construir y mantener un ecosistema de APIs robusto y sostenible:

1. **Adoptar una Cultura API-First:** El cambio más impactante que una organización puede hacer es tratar sus APIs como productos de primera clase, no como subproductos de aplicaciones. Esto implica comenzar el ciclo de desarrollo con el diseño deliberado y colaborativo del contrato API, involucrando a todas las partes interesadas desde el principio. Esta mentalidad fomenta la reutilización, la consistencia y alinea las capacidades técnicas con los objetivos de negocio.
2. **Estandarizar con OpenAPI:** Dada su posición dominante en la industria, su vasto ecosistema de herramientas y el amplio soporte de la comunidad, se recomienda encarecidamente adoptar la Especificación OpenAPI como el lenguaje estándar para formalizar todos los contratos API. Esta estandarización reduce la fricción, acelera la integración y permite aprovechar un rico conjunto de herramientas para la generación de código, pruebas y documentación.
3. **Implementar una Guía de Estilo de Diseño de API:** Para garantizar la coherencia y la calidad en toda la organización, es crucial establecer y hacer cumplir una guía de estilo interna para el diseño de APIs. Esta guía debe codificar las mejores prácticas analizadas en este informe, cubriendo la nomenclatura de URIs, el uso semántico de métodos y

códigos HTTP, formatos de payload consistentes (incluyendo paginación y un modelo de error estandarizado como RFC 7807), una estrategia de versionado clara y requisitos de seguridad. Un conjunto de reglas compartidas y aplicadas de forma consistente es la clave para crear una experiencia de desarrollador predecible y de alta calidad a escala.

Fuentes citadas

1. appmaster.io, acceso: agosto 22, 2025, https://appmaster.io/es/glossary/contrato-api#:~:text=Un%20contrato%20API%20funciona%20como,error%20y%20otras%20instrucciones%20esenciales.
2. Contrato API | AppMaster, acceso: agosto 22, 2025, https://appmaster.io/es/glossary/contrato-api
3. ¿Qué es una API REST (API RESTful)? - IBM, acceso: agosto 22, 2025, https://www.ibm.com/es-es/think/topics/rest-apis
4. What is a REST API and how does it work? - YouTube, acceso: agosto 22, 2025, https://www.youtube.com/watch?v=98b8DazRDdo
5. Arquitectura de una API REST · Desarrollo de aplicaciones web - juanda, acceso: agosto 22, 2025, https://juanda.gitbooks.io/webapps/content/api/arquitectura-api-rest.html
6. Guía Definitiva de Contratos API - Apidog, acceso: agosto 22, 2025, https://apidog.com/es/blog/api-contract/
7. OpenAPI Specification Guide: Structure Implementation & Best Practices - Ambassador Labs, acceso: agosto 22, 2025, https://www.getambassador.io/blog/openapi-specification-structure-best-practices
8. ¿Qué es una API de REST? - Red Hat, acceso: agosto 22, 2025, https://www.redhat.com/es/topics/api/what-is-a-rest-api
9. ¿Qué es una API de RESTful? - Explicación de API de RESTful - AWS, acceso: agosto 22, 2025, https://aws.amazon.com/es/what-is/restful-api/
10. Tipos de contenido soportados por las API REST de IBM BPM, acceso: agosto 22, 2025, https://www.ibm.com/docs/es/bpm/8.5.6?topic=apis-content-types
11. Buenas Prácticas para el Buen Diseño de una API RESTful | Mascando Bits, acceso: agosto 22, 2025, https://mascandobits.es/programacion/buenas-practicas-para-el-buen-diseno-de-una-api-restful/
12. Is API design first or code first? - Design Gurus, acceso: agosto 22, 2025, https://www.designgurus.io/answers/detail/is-api-design-first-or-code-first
13. Buenas prácticas en el diseño de APIs - En Mi Local Funciona, acceso: agosto 22, 2025, https://www.enmilocalfunciona.io/buenas-practicas-en-el-diseno-de-apis/
14. Propuesta de una nueva norma técnica sobre API para servicios web - WIPO, acceso: agosto 22, 2025, https://www.wipo.int/edocs/mdocs/cws/es/cws_8/cws_8_2-annex1.docx
15. Diseño de API - Azure Architecture Center | Microsoft Learn, acceso: agosto 22, 2025, https://learn.microsoft.com/es-es/azure/architecture/microservices/design/api-de

sign
16. RAML and API Blueprint: where are they now? | Postman Blog, acceso: agosto 22, 2025, https://blog.postman.com/raml-and-api-blueprint-where-are-they-now/
17. RAML vs OpenAPI (OAS) Blueprint: A Guide to the best API ... - Knowl, acceso: agosto 22, 2025, https://www.knowl.ai/blog/raml-vs-openapi-oas-blueprint-a-guide-to-the-best-api-specification-clt622oxh002tj7j03eykij3g
18. Choosing Between API Blueprint and Open API Specification: A Comprehensive Guide, acceso: agosto 22, 2025, https://testfully.io/blog/api-blueprint-vs-openapi-guide/
19. Swagger vs RAML vs API Blueprint - Medium, acceso: agosto 22, 2025, https://medium.com/@clsource/swagger-vs-raml-vs-api-blueprint-daccab31f0f2
20. Design First or Code First API Development - SmartBear Support, acceso: agosto 22, 2025, https://support.smartbear.com/swaggerhub/getting-started/introductory-topics/design-first-or-code-first-approach/
21. API First vs Code First: The Right Approach to Building Products - Astera Software, acceso: agosto 22, 2025, https://www.astera.com/type/blog/api-first-vs-code-first/
22. Design-First vs Develop-First: Navigating the Choices in REST API Development, acceso: agosto 22, 2025, https://dev.to/copyleftdev/design-first-vs-develop-first-navigating-the-choices-in-rest-api-development-5c87
23. What is API-first? The API-first Approach Explained | Postman, acceso: agosto 22, 2025, https://www.postman.com/api-first/
24. ¿Qué es el diseño API? Definición, proceso y mejores prácticas - Astera Software, acceso: agosto 22, 2025, https://www.astera.com/es/type/blog/api-design-best-practices/
25. API REST - Buenas prácticas (teoría y práctica) - YouTube, acceso: agosto 22, 2025, https://www.youtube.com/watch?v=uy_xd3Xwt5M
26. ¿Construyendo algo? Capítulo 6.3: [REST] Mejores prácticas | by Felipe Mantilla - Medium, acceso: agosto 22, 2025, https://medium.com/@felipemantillagomez/construyendo-algo-cap%C3%ADtulo-6-3-rest-mejores-pr%C3%A1cticas-81c901083f8f
27. Web API Design Best Practices - Azure Architecture Center | Microsoft Learn, acceso: agosto 22, 2025, https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design
28. Las 10 mejores prácticas para nombrar API REST Endpoints | by Vicente Aguilera - Medium, acceso: agosto 22, 2025, https://vicente-aguilera-perez.medium.com/las-10-mejores-pr%C3%A1cticas-para-nombrar-api-rest-endpoints-48b8bcbd1397
29. Las Mejores Prácticas de Endpoints REST que Todo Desarrollador Debe Conocer, acceso: agosto 22, 2025, https://codigomovil.mx/blog/las-mejores-practicas-de-endpoints-rest-que-todo-desarrollador-debe-conocer

30. APIs RESTful: Principios y Mejores Prácticas - API7.ai, acceso: agosto 22, 2025, https://api7.ai/es/learning-center/api-101/restful-api-best-practices
31. Procedimientos recomendados para el diseño de LA API web RESTful, acceso: agosto 22, 2025, https://learn.microsoft.com/es-es/azure/architecture/best-practices/api-design
32. Métodos HTTP - POST, GET, PUT, DELETE - Estilo Web, acceso: agosto 22, 2025, https://estilow3b.com/metodos-http-post-get-put-delete/
33. Códigos de estado de respuesta HTTP - MDN - Mozilla, acceso: agosto 22, 2025, https://developer.mozilla.org/es/docs/Web/HTTP/Reference/Status
34. Códigos de estado devueltos por las API REST de IBM BPM, acceso: agosto 22, 2025, https://www.ibm.com/docs/es/bpm/8.5.6?topic=apis-status-codes
35. Cómo Implementar Paginación en APIs REST (Guía Paso a Paso) - Apidog, acceso: agosto 22, 2025, https://apidog.com/es/blog/pagination-in-rest-apis/
36. Paginación de API REST: Guía Detallada - Apidog, acceso: agosto 22, 2025, https://apidog.com/es/blog/rest-api-pagination-es/
37. JSON API y Arquitecturas REST, acceso: agosto 22, 2025, https://www.arquitecturajava.com/json-api-y-arquitecturas-rest/
38. Erik Wilde explains Problem Details for HTTP APIs - YouTube, acceso: agosto 22, 2025, https://www.youtube.com/watch?v=OOVUxzoWkhQ
39. Charge your APIs Volume 19: Understanding Problem Details for HTTP APIs - A Deep Dive into RFC 7807 and RFC 9457 - codecentric AG, acceso: agosto 22, 2025, https://www.codecentric.de/en/knowledge-hub/blog/charge-your-apis-volume-19-understanding-problem-details-for-http-apis-a-deep-dive-into-rfc-7807-and-rfc-9457
40. The Power of Problem Details for HTTP APIs | Zuplo Learning Center, acceso: agosto 22, 2025, https://zuplo.com/learning-center/the-power-of-problem-details
41. General error handling rules | Technical Writing - Google for Developers, acceso: agosto 22, 2025, https://developers.google.com/tech-writing/error-messages/error-handling
42. Status and error codes (REST API) - Azure Storage | Microsoft Learn, acceso: agosto 22, 2025, https://learn.microsoft.com/en-us/rest/api/storageservices/status-and-error-codes2
43. Microsoft Graph error responses and resource types, acceso: agosto 22, 2025, https://learn.microsoft.com/en-us/graph/errors
44. Contratos, direccionamiento y API para microservicios | App Engine standard environment for Python 2 | Google Cloud, acceso: agosto 22, 2025, https://cloud.google.com/appengine/docs/legacy/standard/python/designing-microservice-api?hl=es-419
45. API versioning: URL vs Header vs Media Type versioning - Lonti, acceso: agosto 22, 2025, https://www.lonti.com/blog/api-versioning-url-vs-header-vs-media-type-versioning

46. REST versioning - URL vs. header - Stack Overflow, acceso: agosto 22, 2025, https://stackoverflow.com/questions/18905335/rest-versioning-url-vs-header
47. REST API Versioning: How to Version a REST API?, acceso: agosto 22, 2025, https://restfulapi.net/versioning/
48. API design: Which version of versioning is right for you? | Google Cloud Blog, acceso: agosto 22, 2025, https://cloud.google.com/blog/products/api-management/api-design-which-version-of-versioning-is-right-for-you
49. ¿Cómo securizar tus APIs con OAuth? - Paradigma Digital, acceso: agosto 22, 2025, https://www.paradigmadigital.com/dev/oauth-2-0-equilibrio-y-usabilidad-en-la-securizacion-de-apis/
50. Securing REST APIs with OAuth2 and JWT: A Comprehensive ..., acceso: agosto 22, 2025, https://igventurelli.io/securing-rest-apis-with-oauth2-and-jwt-a-comprehensive-guide/
51. Secure API Development Best Practices - OAuth2 and JWT - Conviso AppSec, acceso: agosto 22, 2025, https://blog.convisoappsec.com/en/secure-api-development-best-practices-oauth2-and-jwt/
52. OAuth2 with Password (and hashing), Bearer with JWT tokens - FastAPI, acceso: agosto 22, 2025, https://fastapi.tiangolo.com/tutorial/security/oauth2-jwt/
53. S02L03 – Introducción a Spring Boot OAuth2 JWT - Studyeasy, acceso: agosto 22, 2025, https://studyeasy.org/es/course-articles/spring-boot-restful-api-articles-es/s02l03-introduccion-a-spring-boot-oauth2-jwt/
54. Stripe API Reference, acceso: agosto 22, 2025, https://docs.stripe.com/api
55. Stripe Documentation, acceso: agosto 22, 2025, https://docs.stripe.com/
56. Developer resources - Stripe Documentation, acceso: agosto 22, 2025, https://docs.stripe.com/development