

Guía práctica para la publicación de Datos Abiertos usando APIs

05/02/2025

Este estudio ha sido desarrollado en el marco de la Iniciativa Aporta, desarrollada por el Ministerio de Asuntos Económicos y Transformación Digital, a través de la Entidad Pública Empresarial Red.es.

El uso de este documento implica la expresa y plena aceptación de las condiciones generales de reutilización referidas en el aviso legal que se muestra en:
<http://datos.gob.es/es/aviso-legal>

Agradecimientos:

Por la autoría de una primera edición de este documento a:

- Carlos de la Fuente García.

Por sus contribuciones, sugerencias y su disponibilidad en la elaboración de esta guía a:

- Alicia Martínez Domingo (Experta en datos abiertos).
- Mariano Luis Nieves Coello (CEH, Data Scientist).
- Juan Carlos Ballesteros Hermida (Arquitecto Software y experto en datos abiertos).
- Julián Moyano Collado (Coordinador Aragón Open Data. Gobierno de Aragón).
- Jorge López Pérez (Data Scientist).
- Jorge Cimentada Báez (Research Scientist).
- Alberto Abella García (Experto en datos abiertos).

Contenido

01	Introducción.....	04
----	-------------------	----

02	Entendiendo las APIs.....	06
	• ¿Qué es una API?	06
	• ¿Para qué sirve una API?	06
	• ¿Qué se necesita para usar una API?	06
	• ¿Cómo se implementa una API?	07
	• ¿Cómo funciona una API?	09
	• ¿Cuándo es recomendable habilitar APIs para el acceso a datos abiertos	10
	• ¿Dónde se están publicando APIs?	11
	• Ecosistema FIWARE.....	12
	• ¿Dónde ampliar información sobre APIs?	16

03	Pautas de diseño e implementación de APIs	17
----	--	----

P1: Planificar el ciclo de vida.....	18	P9: Usar cabeceras HTTP para el intercambio de información.....	39
P2: Usar URIs para identificar recursos.....	19	P10: Definir interpretaciones comprensibles de códigos de estado.....	40
P3: Gestión de Resultados.....	23	P11: Incorporar una documentación completa.....	42
P4: Especificación OpenApi (OAS).....	25	P12: Evitar rupturas de servicio o cambios abruptos.....	44
P5: Recomendaciones sobre seguridad en las APIs.....	30	P13: Evitar la degradación del rendimiento del servidor.....	45
P6: Desacoplar servicios de Datos.....	35		
P7: Facilitar el acceso a Datos en tiempo real.....	36		
P8: Especificar el formato de entrega de Datos usando las cabeceras HTTP.....	37		

04	Implementación de APIs en catálogos de Datos Abiertos.....	47
----	--	----

05	APIs para el acceso a Datos Enlazados.....	49
----	--	----

06	APIs para el acceso a Servicios Web Geográficos.....	52
----	--	----

07	Referencias.....	53
----	------------------	----

01 Introducción

Las *Interfaces de Programación de Aplicaciones* (API, por sus siglas en inglés) constituyen uno de los servicios de intercambio de información y acceso a datos más comunes en la actualidad. Complementan la disponibilidad de datos en ficheros descargables y aportan una serie de ventajas que hacen que sea un medio de acceso y consumo de datos imprescindible en cualquier Iniciativa de Datos Abiertos. Además, es el mecanismo de acceso idóneo para publicar datos con alta frecuencia de actualización como los datos en tiempo real o dinámicos.

Es habitual usar **APIs** para acceder a datos meteorológicos, de transporte público o los producidos por sensores de monitorización urbanos, entre otros datos de alto valor. No obstante, las **APIs** son adecuadas para consumir todo tipo de datos de forma automática siendo posible, además, ajustar la descarga exclusivamente a los datos requeridos.

Las **APIs** son elementos software implementados para cubrir múltiples propósitos. Esta guía pone el foco en la disponibilidad de **APIs** en el contexto de los Datos Abiertos y esta orientación implica poner el énfasis en las operaciones que permiten el acceso para lectura y descarga de datos y no tanto en la capacidad de creación o modificación de datos por parte de los usuarios de la API.

Igualmente, existen diferentes alternativas en los modelos arquitectónicos que inspiran el diseño de las **APIs**. Esta guía explica el modelo API REST sobre HTTP que es el tipo de API más popular y extendido. No obstante, hay otros modelos emergentes como GraphQL que está logrando un alto nivel de popularidad por su eficiencia cuando se interactúa con abundancia de recursos de información y también merecen especial atención.

01

Un buen diseño de **APIs** es esencial para maximizar su valor, generar confianza en el sector reutilizador y potenciar la generación de innovación a partir de los datos de mayor impacto. Por esta razón, esta guía propone una serie de contenidos que ayudarán a entender el modelo de diseño REST y a promover la aplicación de un compendio de pautas cuya aplicación contribuirá a generar interfaces de mayor calidad. La guía, además, explica formas específicas de **APIs** para el consumo de datos enlazados utilizando puntos de acceso semánticos y el funcionamiento básico de servicios web que facilitan la operación de datos espaciales. Por otro lado, se hace hincapié en una tendencia esencial para el diseño estandarizado utilizando la especificación OpenAPI.

Aunque el ámbito primordial de este documento se centra en la implementación eficiente de **APIs**, no se debe olvidar que se trata de servicios de Datos y, por tanto, es fundamental la aplicación de todas las buenas prácticas vinculadas a la calidad de datos en general. En este sentido es recomendable complementar esta guía con la lectura y aplicación de otras guías que orienten sobre la aplicación de pautas para asegurar la publicación de datos estructurados de calidad.

Por último, hay que señalar que el público objetivo de esta guía es principalmente el promotor de Datos Abiertos cuyo objetivo es configurar servicios de datos a través de **APIs**, por tanto, es recomendable que el lector esté familiarizado con conocimientos básicos sobre programación – fundamentalmente porque los usuarios primordiales de las **APIs** serán desarrolladores- y la comprensión del funcionamiento de la arquitectura cliente servidor que soporta las peticiones y respuestas de información a través de Internet.

02 Entendiendo las APIs



¿Qué es una API?

Una interfaz de programación de aplicaciones, API, es un mecanismo que permite la comunicación e **intercambio de información entre sistemas**.

En el contexto de los Datos Abiertos, el término generalmente se refiere a **APIs sobre la Web**, denominadas en algunos ámbitos API Web, que es un medio habitual para soportar el intercambio de información dentro y entre organizaciones.

Esta característica implica que una API ofrece un conjunto de funcionalidades sobre un servidor en la Web para ser utilizadas por aplicaciones cliente mediante el uso de procedimientos estándar.

¿Para qué sirve una API?

Se usan para interactuar con el sistema de información de una organización sin necesidad de un conocimiento de la estructura interna o de la tecnología utilizada en su desarrollo.

Las APIs ofrecen una opción flexible de acceso a los datos respecto a la descarga de archivos, aunque ambas opciones son compatibles y una opción no excluye a la otra.

Una ventaja de las APIs frente a la descarga de archivos es que permite aplicar diferentes operaciones durante el acceso y recuperación de los datos. El filtrado de datos o la selección del formato de salida son operaciones habituales en el consumo de datos vía API.

¿Qué se necesita para usar una API?

Se necesitan conocer los mecanismos de uso de la API. Para ello, se pone a disposición de los usuarios una documentación que describe los detalles técnicos involucrados en su uso, entre otros, la forma de acceso y las operaciones necesarias para interactuar con la API.

La documentación de una API es uno de sus elementos más importantes y constituye un acuerdo entre las partes, a modo de un contrato que describe el comportamiento y las condiciones de uso de la API: especifica qué operaciones están disponibles y qué se va a obtener como respuesta al invocarlas.

¿Cómo se implementa una API?

Las APIs se implementan siguiendo algún **modelo arquitectónico o guía de diseño** que permite definir las reglas que gobiernan la interacción entre sistemas. Aunque existen diferentes modelos de implementación, el foco de esta guía es el diseño arquitectónico [REST \(Representational State Transfer\)](#). Las APIs que siguen este modelo de implementación se llaman *RESTful APIs*.

Una característica relevante de este modelo es el uso de **estándares abiertos como HTTP/HTTPS**, lo cual no vincula las implementaciones de la API en el servidor o de las aplicaciones cliente con ninguna implementación concreta, es decir, ambos componentes se pueden implementar usando lenguajes de programación diferentes, siempre que puedan formular solicitudes y entender respuestas usando el protocolo HTTP.

Una API REST se puede **implementar utilizando cualquier lenguaje de programación** siendo los más habituales: Java, .NET, Python, PHP, Ruby o el más reciente Node.js. En el apartado de referencias de esta guía se detalla una serie de herramientas de uso habitual para la implementación y documentación de APIs.

Las API REST se diseñan para exponer e interactuar con **recursos** que son objetos, datos o servicios a los que puede acceder un cliente. Conceptualmente, **un recurso no debe asociarse con una estructura física de datos**, dado que podría derivar de consultas internas a varias tablas de una base de datos relacional y presentarse al cliente como una única entidad.

API REST, por tanto, es una interfaz entre servidores y clientes para intercambiar **representaciones de datos** en la Web en diferentes formatos, sobre todo JSON y XML.

El uso del protocolo HTTP como método de acceso y de URIs para identificar unívocamente recursos de datos de forma individual, aportan una **separación fundamental entre peticiones y respuestas** de datos, logrando implementaciones de servicios y aplicaciones más eficientes.

02

Una ventaja inherente al uso de estándares de la Web para el diseño de APIs es la posibilidad de **utilizar enlaces a recursos adicionales como complemento a la salida de datos** como, por ejemplo, un enlace a un diccionario de datos o cualquier tipo de documento hipermedia relacionado con la salida de la API.

- **Ejemplo inventado de petición de información a una API sobre rutas de bus en una ciudad:**

`GET https://data.mycity.example.com/transport/api/v2/routes`

- **Ejemplo de respuesta a la petición anterior que utiliza códigos de respuesta estándar HTTP y devuelve además de datos concretos sobre la ruta, enlaces a otros recursos vinculados:**

```
{
  "code": "200",
  "text": "OK",
  "data": {
    "update_time": "2013-01-01T03:00:02Z",
    "route_id": "52",
    "route_name": "Lexington South",
    "route_description": "Lexington corridor south of Market",
    "route_type": "3"
  },
  "links": [
    {
      "href": "https://data.mycity.example.com/transport/api/v2/routes/52",
      "rel": "self",
      "type": "application/json",
      "method": "GET"
    },
    {
      "href": "https://data.mycity.example.com/transport/api/v2/routes",
      "rel": "collection",
      "type": "application/json",
      "method": "GET"
    },
    {
      "href": "https://data.mycity.example.com/transport/api/v2/schedules/52",
      "rel": "describedby",
      "type": "application/json",
      "method": "GET"
    },
    {
      "href": "https://data.mycity.example.com/transport/api/v2/maps/52",
      "rel": "describedby",
      "type": "application/json",
      "method": "GET"
    }
  ]
}
```


¿Cómo funciona una API?

La comunicación entre servidor y clientes sobre la Web se realiza mediante el intercambio de mensajes HTTP en un contexto seguro de interacción. Los mensajes son de dos tipos: **peticiones**, enviadas por el cliente al servidor para solicitar el inicio de alguna acción para interactuar con recursos de información y **respuestas** que constituyen la materialización de la acción solicitada.

Cada **petición contiene toda la información necesaria** para ejecutar la acción requerida, lo que implica que toda la interacción se realiza en un único ciclo y no es necesario recordar ningún estado previo para satisfacerla. Esta es una característica importante de la arquitectura REST. Implica que el contenido solicitado por el cliente no queda almacenado en el servidor entre solicitudes, sino que la información sobre el estado de la sesión la maneja el cliente.

Las acciones u operaciones, también llamadas métodos, se especifican mediante el uso de los denominados **verbos HTTP**.

En una petición, cada método tiene encomendada una misión. Cualquier sistema que exponga una API REST dispone de los siguientes métodos:

- **GET**: recupera una representación de un recurso de datos.
- **HEAD**: recupera la cabecera de una respuesta.
- **POST**: crea un nuevo recurso de datos.
- **PUT**: actualiza un recurso existente o lo crea si no existe previamente.
- **PATCH**: realiza una actualización parcial de un recurso.
- **DELETE**: elimina un recurso de datos existente.
- **OPTIONS**: recupera las opciones de comunicación para el recurso solicitado.

Los métodos expuestos pueden incluir un cuerpo de mensaje para especificar detalles sobre la petición.

En el contexto de los Datos Abiertos no es habitual disponer de los métodos PUT, PATCH o DELETE. Normalmente se dispone de los métodos GET, HEAD o POST, usando GET como la operación común para el acceso y descarga de recursos de datos.

¿Cuándo es recomendable habilitar APIs para el acceso a Datos Abiertos?

Siempre que sea posible. Los métodos habituales para consumir datos abiertos son la disponibilidad de archivos descargables y el acceso a datos de forma automatizada a través de APIs. Ambos métodos no son excluyentes.

Desde el punto de vista de la publicación de datos, habilitar APIs de acceso estables y bien documentadas, **requieren un mayor esfuerzo inicial, no obstante, a medio plazo resulta beneficioso** para el publicador que aumenta las capacidades de innovación permitiendo a los desarrolladores crear nuevas aplicaciones y servicios explotando datos reutilizables.

Una buena estrategia de apertura de datos tendrá disponible la última versión de los datos a través de APIs y también los históricos generados con anterioridad, adecuadamente versionados, en ficheros estáticos descargables.

La disponibilidad de APIs es el **mecanismo más eficiente para consumir datos con alta frecuencia de actualización**, es decir, cercanos al tiempo real, de tal forma que los sistemas que producen los datos puedan hacerlos disponibles de forma automática.

Es aconsejable disponer APIs si los conjuntos de datos son grandes, de alta frecuencia de actualización, o incluso de alta complejidad, la opción de implementar una API es la más adecuada ya que será posible para el diseñador modelar las representaciones de los datos adecuadamente y permitir aplicar determinadas operaciones de filtrado y selección.

Además, la implementación de APIs facilita la interoperabilidad entre diferentes sistemas y plataformas, permitiendo una integración más fluida con otras herramientas y servicios. Esto no solo mejora la accesibilidad de los datos, sino que también potencia la creación de un ecosistema digital más conectado y eficiente.

¿Dónde se están publicando APIs?

Existen Iniciativas de Datos Abiertos de la Administración Pública en España que incorporan la disponibilidad de APIs en sus estrategias de apertura de datos. Entre otras cabe mencionar:

- **Datos Abiertos de Zaragoza** <https://www.zaragoza.es/sede/portal/datos-abiertos/api>
- **Dades Obertes Manlleu**: <https://dadesobertes.diba.cat/dades-obertes/documentacio-tecnica/api>
- **Aemet OpenData API**: <https://opendata.aemet.es/>
- **Catálogo de APIs Abiertas ISTAC**: <https://www3.gobiernodecanarias.org/aplicaciones/appsistac/api>
- **EMT mobilitylabs**: <https://mobilitylabs.emtmadrid.es/>

Repositorio RapidAPI Hub:

Un espacio de referencia para observar la evolución de la denominada "economía de las APIs" es el repositorio del sitio web especializado **RapidAPI Hub**.

Además de ser un importante centro de recursos para el desarrollo de APIs, mantiene uno de los directorios más grandes y activos de APIs. Entre otras categorías, incluye un catálogo significativo de APIs disponibles en el ámbito de la Administración Pública, permitiendo explorar, probar y conectar con miles de APIs públicas y privadas:

<https://rapidapi.com/hub>

Otros repositorios de APIs públicas:

- <https://apis.guru/>
- <https://github.com/public-apis/public-apis>
- <https://smart-api.info/>

Ejemplos de APIs públicas muy populares:

Algunas APIs de notable éxito y de uso sencillo, están disponibles en empresas que gestionan volúmenes ingentes de información: Amazon, Google Maps, Twitter o Paypal, entre otras. De forma general, están muy bien documentadas y son un referente importante para comprender el alcance de una buena documentación:

- **API de Github**: <https://developer.github.com/v3/>
- **API de Google Maps**: <https://cloud.google.com/maps-platform/?hl=es>
- **API de X**: <https://developer.twitter.com/>

Ecosistema FIWARE – Implementación y arquitectura

Implementación de APIs usando FIWARE

FIWARE representa una opción consolidada para la implementación de APIs en el contexto de los datos abiertos gubernamentales. Este marco de trabajo, nacido como iniciativa europea y ahora convertido en estándar global, ofrece un conjunto de especificaciones y componentes que facilitan el desarrollo de servicios digitales interoperables.

La principal ventaja de FIWARE para los publicadores de datos abiertos reside en su **enfoque estandarizado**. Al utilizar FIWARE, las organizaciones no solo implementan una API, sino que se integran en un **ecosistema maduro** que facilita la **interoperabilidad y la reutilización de datos**.

Una de las ventajas de FIWARE es la capacidad para facilitar la interacción con información de contexto (*Context Broker*), es decir, datos de entorno originados en diferentes fuentes como, por ejemplo, redes de sensores, aplicaciones móviles o todo tipo de sistemas de información y redes sociales, para generar acciones propias del entorno en el que se implementa una solución.

Los **modelos de datos FIWARE** representan uno de los pilares fundamentales del ecosistema, proporcionando una **base estandarizada para la representación de información** en diferentes dominios. Estos modelos han sido desarrollados a través de un proceso colaborativo que involucra a expertos en diferentes campos y han sido validados en implementaciones reales.

Arquitectura FIWARE

La arquitectura FIWARE se caracteriza por su diseño modular y escalable, con el *Context Broker* como componente central que revoluciona la gestión de datos en tiempo real. Este elemento actúa como el cerebro del sistema, procesando y distribuyendo información contextual que define el estado actual del entorno a través de "entidades de contexto", las cuales representan objetos del mundo real como sensores, vehículos o edificios.

La gestión de la información se realiza mediante la interfaz NGSI (*Next Generation Service Interface*), un protocolo estandarizado que permite la interacción con el *Context Broker* a través de operaciones fundamentales como consulta de datos actuales e históricos, actualización en tiempo real, suscripción a cambios en entidades específicas, y registro y descubrimiento de fuentes de contexto. La implementación de referencia, denominada Orion, proporciona una base robusta para la gestión de información contextual mediante un modelo de publicación/suscripción.

Una de las características más destacadas de FIWARE es su **capacidad para integrar múltiples fuentes de datos heterogéneas**, permitiendo al *Context Broker* recibir información desde redes de sensores IoT, sistemas de información tradicionales, aplicaciones móviles, redes sociales, servicios web externos y sistemas de terceros, lo que lo convierte en una solución versátil y completa para la gestión de datos en tiempo real.

Modelos de datos FIWARE

La armonización de modelos de datos resuelve uno de los principales desafíos en la interoperabilidad de sistemas: la heterogeneidad en la representación de la información.

Al definir estructuras de datos comunes y estandarizadas, FIWARE facilita la integración entre diferentes sistemas y aplicaciones, la portabilidad de soluciones entre diferentes ciudades o regiones, la reutilización de componentes y aplicaciones, y la creación de un mercado de soluciones interoperables.

Una de las principales aportaciones de FIWARE es su extensa biblioteca de modelos de datos estandarizados. [Estos modelos](#) cubren múltiples dominios y han sido validados en implementaciones reales.

Smart Cities

Este dominio abarca la gestión urbana integral, incluyendo modelos para:

- Transporte y movilidad urbana (paradas de autobús, estaciones de bicicletas, parkings).
- Medio ambiente urbano (calidad del aire, ruido, zonas verdes).
- Servicios municipales (residuos, alumbrado, riego).
- Equipamientos públicos (edificios, instalaciones deportivas).

Smart Environment

Centrado en la monitorización y gestión ambiental, incluye modelos para:

- Calidad del aire y emisiones.
- Recursos hídricos.
- Condiciones meteorológicas.
- Contaminación acústica.
- Biodiversidad.

Smart Agrifood

Orientado al sector agroalimentario, proporciona modelos para:

- Explotaciones agrícolas.
- Sistemas de riego.
- Control de ganado.
- Cadena de suministro alimentaria.

Se han desarrollado además los **modelos transversales** de *Cross Sector*, los cuales pueden aplicarse en múltiples dominios, incluyendo alertas y eventos, indicadores clave de rendimiento (KPIs), gestión de usuarios, y calendarios y programaciones.

Cada modelo de datos está **completamente documentado** con una definición detallada de atributos y relaciones, restricciones y validaciones, ejemplos de uso, documentación completa y formatos de serialización estándar.

La adopción de estos modelos estandarizados aporta beneficios significativos al proceso de desarrollo, como la reducción del tiempo de implementación, una mejora notable en la calidad de los datos y la creación de un ecosistema de aplicaciones plenamente compatibles entre sí.

Ejemplo FIWARE: Sistema de calidad del aire

Para ilustrar el uso práctico de FIWARE en un caso real, se presenta a continuación un ejemplo detallado de implementación de un sistema de medición de calidad del aire. Este ejemplo está basado en el modelo de datos estandarizado [AirQualityObserved](#).

El sistema permite gestionar mediciones de calidad del aire realizadas en diferentes puntos de una ciudad. Para cada medición, se registran tanto los valores de los contaminantes como las condiciones ambientales asociadas.

1. Creación de una nueva medición

```
POST /v2/entities
Host: localhost:1026
Content-Type: application/json
fiware-service: opendata
fiware-servicepath: /environment

{
  "id": "Madrid-AmbientObserved-28079004-2016-03-15T11:00:00",
  "type": "AirQualityObserved",
  "address": {
    "addressCountry": "ES",
    "addressLocality": "Madrid",
    "streetAddress": "Plaza de España"
  },
  "dateObserved": "2016-03-15T11:00:00/2016-03-15T12:00:00",
  "location": {
    "type": "Point",
    "coordinates": [-3.71224722222222, 40.42385277777775]
  },
  "source": "http://datos.madrid.es",
  "airQualityLevel": "moderate",
  "reliability": 0.9,
  "CO": 500,
  "NO": 45,
  "NO2": 69,
  "NOx": 139,
  "SO2": 11
}
```

En este ejemplo, la estructura de datos se organiza de manera precisa y completa: el identificador único se construye mediante la combinación de la ciudad, estación y *timestamp*, mientras que la localización se detalla tanto a través de la dirección como de las coordenadas geográficas.

El modelo incorpora las mediciones de diversos contaminantes (CO, NO, NO2, NOx, SO2) y además proporciona un indicador cualitativo general sobre la calidad del aire.

2. Consulta de mediciones

Para recuperar las mediciones existentes, se puede realizar una petición GET:

```
GET /v2/entities?type=AirQualityObserved&options=keyValues
Host: localhost:1026
fiware-service: opendata
fiware-servicepath: /environment
```

La respuesta incluirá todas las mediciones registradas. **También es posible filtrar** por diferentes criterios, por ejemplo, para obtener solo las mediciones de una estación específica:

```
GET /v2/entities?type=AirQualityObserved&q=address.streetAddress=="Plaza de España"
Host: localhost:1026
fiware-service: opendata
fiware-servicepath: /environment
```

Cuando se reciben nuevas lecturas de los sensores, se pueden **actualizar los valores** mediante una petición PATCH:

```
PATCH /v2/entities/Madrid-AmbientObserved-28079004-2016-03-15T11:00:00/attrs
Host: localhost:1026
Content-Type: application/json
fiware-service: opendata
fiware-servicepath: /environment

{
  "CO": {
    "value": 550
  },
  "NO2": {
    "value": 75
  },
  "airQualityLevel": {
    "value": "unhealthy"
  }
}
```

La **implementación completa** del sistema requiere **considerar diversos aspectos críticos**, como la validación de los valores recibidos y la gestión de errores y valores anómalos. También debe contemplar el almacenamiento histórico de las mediciones y la generación automática de alertas cuando se superan ciertos umbrales.

¿Dónde ampliar información sobre APIs?



La unidad didáctica ["Buenas prácticas en el diseño de APIs y Linked Data"](#) del catálogo de materiales formativos de la Iniciativa Aporta profundiza y amplía el contenido en esta guía. Su lectura permitirá:

Entender los principios generales de las APIs web basadas en protocolo HTML.

Ser capaz de realizar un diseño a alto nivel del esquema de direcciones (URLs) y operaciones (métodos) de una API REST para acceso a datos gubernamentales teniendo en cuenta las recomendaciones técnicas ya existentes.

Comprender qué son los formatos semánticos y la importancia de los datos enlazados como solución técnica al movimiento por los datos abiertos.

Entender la evolución y relación entre los conceptos de datos abiertos (Open Data) y datos enlazados (Linked Data).

Conocer los proyectos institucionales más relevantes que apuestan por el uso de datos enlazados y web semántica para ofrecer datos abiertos gubernamentales.

Entender los principios generales y tecnológicos de la web semántica, y su diferencia con formatos.

Entender cómo funcionan las consultas semánticas mediante el lenguaje SPARQL.

03 Pautas de diseño e implementación de APIs

En el siguiente apartado de esta guía, se recogen una serie de pautas sobre el diseño e implementación de APIs que garantizan la calidad, seguridad y usabilidad de los servicios desarrollados.

Estas recomendaciones están basadas en estándares internacionales y mejores prácticas de la industria, asegurando así la creación de APIs robustas y mantenibles.



P1 - Planificar el ciclo de vida



P2 - Usar URIs para recursos



P6 - Desacoplar servicios de Datos



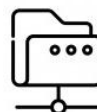
P10 - Definir interpretaciones comprensibles de códigos



P3 - Gestión de Resultados



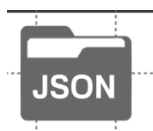
P7 - Facilitar el acceso a Datos en tiempo real



P11 - Incorporar una documentación completa



P4 - Especificación OpenApi (OAS)



P8 - Formato de entrega de Datos



P12 - Evitar rupturas de servicio



P5 - Recomendaciones sobre seguridad en las APIs



P9 - Usar cabeceras HTTP



P13 - Evitar la degradación del rendimiento del servidor

P1 – Planificar el ciclo de vida

La disponibilidad de datos a través de una API requiere una **visión holística y la aplicación de un modelo de gestión adecuado para maximizar el valor de la API**.

Entre otras cuestiones, los publicadores de datos, además de garantizar el acceso a los recursos de datos disponibles y documentar las especificaciones de la API, deben tener en cuenta parámetros relacionados con la frecuencia de actualización, la latencia introducida en el procesamiento de los recursos, la infraestructura necesaria para su disponibilidad y la respuesta de la comunidad de reutilizadores.

Estos aspectos y otros que se irán desgranando en esta guía, deben ser tenidos en cuenta en las diferentes etapas del **ciclo de vida de una API**. De manera orientativa se indican los componentes que lo configuran:



De forma resumida y no exhaustiva, estas etapas conllevan la ejecución de las siguientes tareas:

Diseño: definición de requisitos de modelado de recursos, seguridad, infraestructura, niveles de calidad de servicio, versionado y documentación.

Implementación: preparación de *backend* y de la capa de administración para implementar políticas de seguridad, operaciones, parámetros y publicación en la Web.

Despliegue: establecimiento de la puerta de enlace para atender peticiones.

Administración: parametrización y gestión de los acuerdos de nivel de servicio.

Descubrimiento: publicación de la API con sus especificaciones y documentación en catálogos públicos de acceso a APIs. Implicación de la comunidad de reutilizadores.

Monitorización: evaluación y métricas de rendimiento, niveles de servicio y seguridad.

P2 – URIs para identificar recursos

Las API REST gestionan y exponen recursos de información direccionables por medio de Identificadores de Recursos Uniformes (URI), que los diferencia y permiten el acceso a cualquiera de las representaciones disponibles.

Aunque es común encontrar que los términos URI y URL se utilizan indistintamente en muchos textos, existe una diferencia sustancial entre ambos conceptos. La **responsabilidad** de definir las URIs que se utilizarán para invocar a la API **recae en los diseñadores**, por lo que resulta fundamental incorporar una buena estrategia de nombrado durante la etapa de diseño, aspecto que se abordará en detalle en el apartado de pautas de esta guía, donde se explicará el patrón de construcción que siguen las URIs.

URI = esquema ":" // " autoridad "/" ruta ["?" consulta] ["#" fragmento]

Por ejemplo:

`https://datos.ejemplo.com/v1/licitaciones/contratos?estado=finalizado`

Cada elemento de la URI tiene una misión y su especificación debe ser adecuada para que la URI resultante sea intuitiva y fácilmente comprensible para el usuario.

Son elementos de la URI los siguientes:

- El **esquema o protocolo de acceso**, normalmente "https" o "http".
- La **autoridad que publica la API**, indicada por el nombre del servidor y opcionalmente el puerto de acceso donde está desplegada la API.
- La **versión de la API**, numerada secuencialmente desde la primera.
- La **ruta de acceso al recurso o endpoint** que se consulta y la operación que se realiza sobre el recurso.
- La **consulta sobre el recurso** con las opciones de filtrado que sean necesarias son elementos opcionales de la URI.

[1] Diferencia entre URI y URL

Las URIs actúan como identificadores de recursos (indicando su 'nombre'), mientras que las URLs son localizadores que indican su 'ubicación' en la red. Aunque toda URL es también un URI por su naturaleza identificadora, no todo URI es una URL. Esta diferencia puede ejemplificarse comparando el nombre de una persona (identificador) con una dirección postal (localizador e identificador), donde la dirección no solo ubica, sino que también puede identificar indirectamente a su único habitante.

RECOMENDACIONES PARA EL DISEÑO DE URIs

Un buen diseño de URIs en APIs REST debe fundamentarse en una adecuada **estrategia de nombrado de recursos**.

Estas APIs pueden manejar diferentes tipos de recursos, incluyendo documentos (instancias individuales), colecciones (conjuntos de recursos), almacenes o 'stores' (repositorios gestionados por clientes), y controladores (para procesos específicos). Por ejemplo, mientras una URI como <https://datos.ejemplo.com/v1/licitaciones/contratos/{id-contrato}> representa un documento individual, <https://datos.ejemplo.com/v1/licitaciones/contratos> hace referencia a una colección completa.

En el contexto de la publicación de Datos Abiertos, el enfoque principal se centra en la gestión de recursos individuales y colecciones. Para su implementación efectiva, es crucial seguir ciertas recomendaciones lingüísticas en el diseño de URIs:

- * Utilizar términos sencillos, intuitivos y auto-explicativos que no requieran conocimiento específico del contexto.
- * Mantener la coherencia en el uso del plural o singular (preferentemente plural, según la práctica más generalizada).
- Evitar ambigüedades en la denominación y términos que puedan entrar en conflicto con palabras clave de lenguajes de programación.

La implementación de estas prácticas de diseño no solo mejora la mantenibilidad del código, sino que también **facilita la integración con otros sistemas** y la **adopción por parte de desarrolladores externos**. Una URI bien diseñada actúa como una forma de documentación implícita, permitiendo a los usuarios de la API comprender intuitivamente la estructura y funcionamiento de los *endpoints* sin necesidad de consultar documentación adicional.

Además, un diseño coherente de URIs **contribuye significativamente a la escalabilidad del sistema**, permitiendo la evolución natural de la API **sin romper la compatibilidad con versiones anteriores**. Esto es particularmente importante en el contexto de servicios públicos y datos abiertos, donde la estabilidad y la previsibilidad son características esenciales para mantener la confianza de los usuarios y facilitar la adopción a largo plazo.

Uso del separador "/"

El separador "/" se utiliza para indicar relaciones jerárquicas entre recursos:

```
https://datos.ejemplo.com/v1/licitaciones
https://datos.ejemplo.com/v1/licitaciones/contratos
https://datos.ejemplo.com/v1/licitaciones/contratos/menores
https://datos.ejemplo.com/v1/licitaciones/contratos/menores/servicios
```

Omisión del separador final

El separador "/" no debe incluirse al final de la URI, ya que no aporta valor semántico:

- ✓ Correcto: `https://datos.ejemplo.com/v1/licitaciones/contratos`
- ✗ Incorrecto: `https://datos.ejemplo.com/v1/licitaciones/contratos/`

Uso del guion para legibilidad

Se recomienda usar el carácter "-" para mejorar la legibilidad de las URIs:

- ✓ Correcto: `https://datos.ejemplo.com/v1/licitaciones/contratos/{id-contrato}`
- ✗ Incorrecto: `https://datos.ejemplo.com/v1/licitaciones/contratos/{idcontrato}`

Evitar el guion bajo

El carácter "_" debe evitarse ya que puede quedar oculto en enlaces subrayados:

- ✓ Correcto: `https://datos.ejemplo.com/v1/licitaciones/contrato-menor/{id-contrato}`
- ✗ Incorrecto: `https://datos.ejemplo.com/v1/licitaciones/contrato_menor/{id_contrato}`

El diseño cuidadoso de URIs, prestando atención a los caracteres especiales y su uso apropiado, es fundamental para crear APIs intuitivas y fáciles de usar. La consistencia en el uso de separadores, la omisión del separador final, y la preferencia por guiones sobre guiones bajos contribuyen a una mejor experiencia de usuario y mantenibilidad del sistema. Estas convenciones, aunque parezcan detalles menores, tienen un impacto significativo en la calidad general de la API.

Uso de minúsculas en URIs

La normalización de URIs requiere el **uso de letras minúsculas**, según la especificación RFC 3986. Es importante notar que solo el esquema y la autoridad (host) no son sensibles a mayúsculas y minúsculas.

- ✓ Correcto: `https://datos.ejemplo.com/v1/licitaciones/contratos`
- ✗ Incorrecto: `https://DATOS.EJEMPLO.COM/v1/licitaciones/contratos`
- ✗ Incorrecto: `https://datos.ejemplo.com/v1/Licitaciones/Contratos`

Extensiones de archivo

Las **extensiones de archivo son innecesarias** y aumentan el tamaño de la URI. El formato debe especificarse mediante cabeceras HTTP.

- ✓ Correcto: `https://datos.ejemplo.com/v1/licitaciones/contratos`
- ✗ Incorrecto: `https://datos.ejemplo.com/v1/licitaciones/contratos.xml`

Operaciones CRUD

Las **URIs deben identificar recursos, no acciones**. Las operaciones CRUD (Create, Read, Update, Delete) se manejan a través de métodos HTTP:

- ✓ Correcto: HTTP GET `https://datos.ejemplo.com/v1/licitaciones/contratos`
- ✗ Incorrecto: HTTP GET `https://datos.ejemplo.com/v1/licitaciones/obtenercontratos`

La adherencia a estas prácticas de diseño de URIs no solo mejora la legibilidad y mantenibilidad de las APIs, sino que también garantiza una mayor consistencia y profesionalidad en el desarrollo. El uso de minúsculas, la omisión de extensiones de archivo y la correcta implementación de operaciones CRUD a través de métodos HTTP son fundamentales para crear APIs RESTful robustas y escalables. Estas pautas, aunque simples en apariencia, constituyen la base de una arquitectura API bien diseñada y orientada a futuro.

P3 – Gestión de resultados

Uno de los usos más probables de una API es el acceso a **porciones concretas de datos en base a las especificaciones que determina el usuario**. Para ello, las API REST permiten definir dichos criterios en la parte de la URI destinada a la consulta, habilitando así funcionalidades de filtrado, ordenación y paginación para recuperar exclusivamente los datos necesarios.

Filtrado: devuelve el resultado de la evaluación de una expresión lógica compuesta por tres componentes: propiedad, operador lógico y valor, que pueden ser concatenados utilizando el carácter ‘&’ aportando flexibilidad a la consulta. La sintaxis sigue la siguiente estructura:

?[campo][operador][valor]

Son operadores habituales: igual (= o ‘eq’), mayor o igual (>= o ‘gte’), menor o igual (<= o ‘lte’), contiene (~) para campos de texto. Otros operadores que también pueden ser aplicables son [exists], [regex], [before], y [after]. El operador de concatenación ‘&’ es evaluado como un OR para los valores de un mismo nombre de atributo y AND entre atributos distintos.

Por ejemplo:

<https://datos.ejemplo.com/v1/licitaciones/contratos?estado=finalizado&annio=2019>

El ejemplo anterior está recuperando de la colección de contratos, los registros de 2019 correspondientes a contratos finalizados. Existen lenguajes de consultas, como FIQL o RSQL (evolución del anterior) que facilitan el filtrado parametrizado de invocaciones a API RESTful.

- **Ordenación:** devuelve las peticiones ordenadas según un determinado criterio aplicado sobre propiedades del recurso. Se expresa mediante parámetros ‘sort’, ‘sort_by’ u ‘order’ que admiten modificadores para indicar el sentido del orden ‘ascendente’ o ‘descendente’.

Por ejemplo:

<https://datos.ejemplo.com/v1/licitaciones/contratos?estado=finalizado&annio=2019&sort=fecha-a-adjudicacion>

Al ejemplo anterior se añade el requisito de ordenación por fecha de adjudicación.

- **Paginación:** permite acotar los resultados devueltos por una petición. Se pueden utilizar diferentes parámetros:
 - paginación con 'offset'.
 - Paginación basada en tiempo con 'limit':
 - Paginación basada en cursor o identificador específico de registro.

Por ejemplo:

<https://datos.ejemplo.com/v1/licitaciones/contratos?offset=50&limit=25&estado=finalizado>

El ejemplo anterior está devolviendo de la colección de contratos, los registros entre el 51 y el 75. El siguiente ejemplo está devolviendo registro específico y situando un puntero al siguiente registro sobre el que se sitúa el comienzo de la consulta obtener la siguiente página de resultados.

<https://datos.ejemplo.com/v1/licitaciones/contratos?cursor=1234567>

Además, es posible implementar mecanismos de control de paginación utilizando parámetros como los que se indican a continuación:

- Utilizando un identificador de página específico usando 'paginationkey'.
- el número de orden de pagina que contiene resultados de una consulta, usando 'page'
- el número de elementos que debe contener una pagina de respuesta, usando 'pageSize'
- Otros indicadores de paginación para controlar movimientos entre páginas, usando 'firstLink', 'lastLink', 'nextLink', 'previousLink', 'selfLink' que permiten desplazamientos a primera, ultima, siguiente, previa o página actual.

Ejemplos:

<https://datos.ejemplo.com/v1/licitaciones/contratos?offset=50&limit=25&paginationKey=xyz>

El ejemplo está devolviendo contenido en función de una página concreta que posee un determinado identificador de página.

<https://datos.ejemplo.com/v1/licitaciones/contratos?pageSize=2&page=3>

El ejemplo está devolviendo contenido de la tercera página de resultados con dos elementos por página.

P4 – Especificación OpenApi (OAS)

Para comprender la motivación principal de la existencia de un **estándar abierto para APIs**, es fundamental analizar la perspectiva del desarrollador: una parte significativa de su tiempo y esfuerzo se dedica a integrar servicios de información proporcionados por múltiples APIs, que frecuentemente siguen especificaciones diferentes, lo que requiere crear adaptaciones personalizadas para cada servicio, resultando en mayores costes y posibles ineficiencias.

La [especificación OpenAPI \(OAS\)](#) establece una descripción estándar de interfaz independiente del lenguaje de programación para las API REST, permitiendo tanto a personas como a máquinas descubrir, entender y utilizar las capacidades de un servicio de manera efectiva. Como evolución abierta del [proyecto Swagger](#), OpenAPI, actualmente en su versión 3.0, promueve la interoperabilidad de sistemas mediante interfaces claras y duraderas.

En un **único marco de trabajo**, OpenAPI facilita el diseño y documentación de APIs Rest, asegura la compatibilidad con estándares establecidos, y permite la generación de interfaces de desarrollo (SDKs) con el nivel de abstracción necesario para que terceros puedan comunicarse eficientemente con la API, garantizando su funcionamiento en diferentes versiones de un mismo lenguaje. Además, ofrece la flexibilidad de personalizar y adaptar la interfaz según necesidades específicas, realizar pruebas de la API, y obtener métricas de uso y análisis detallados.

Una característica particularmente valiosa de OpenAPI es su capacidad de mantener la documentación sincronizada automáticamente con el código de la API: cuando se actualiza el código, la documentación de los métodos, parámetros y modelos de datos se actualiza correspondientemente, asegurando una coherencia perfecta entre implementación y documentación.

Un caso de uso destacable es la aplicación de esta especificación a APIs existentes, lo que resulta en una API completamente documentada según el estándar y la posibilidad de generar SDKs para clientes, mejorando significativamente su accesibilidad y usabilidad.

Ejemplo de uso de la especificación OpenAPI en el portal de Datos Abiertos de la Unión Europea:

https://app.swaggerhub.com/apis/EU-Open-Data-Portal/eu-open_data_portal/0.8.0

Reutilización y adaptación de interfaces existentes

En el contexto de los Datos Abiertos, el diseño e implementación de APIs no debe partir necesariamente desde cero. La experiencia demuestra que la reutilización y adaptación de interfaces existentes no solo acelera el proceso de desarrollo, sino que también favorece la adopción por parte de la comunidad de desarrolladores, quienes ya están familiarizados con patrones y estructuras similares.

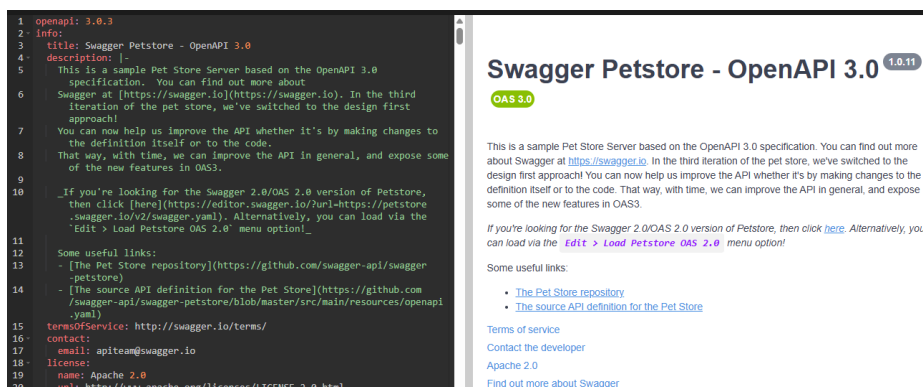
Este enfoque de reutilización resulta especialmente valioso cuando observamos el ecosistema actual de APIs en el sector público. Por ejemplo, portales como datos.gob.es, la [API de AEMET OpenData](#) o las APIs municipales de ciudades como [Madrid](#) han establecido patrones de diseño que han demostrado su efectividad en la práctica. Al analizar estas implementaciones, se encuentran estructuras comunes que se pueden aprovechar para desarrollos propios.

El proceso de adaptación de una API existente comprende varios aspectos fundamentales:

1. Análisis de interfaces de referencia

En el caso de una API para datos de movilidad urbana, por ejemplo, la [API de la EMT Madrid](#) puede servir como referencia por su robustez y efectividad probada. Su estructura ofrece insights valiosos sobre:

- La organización de *endpoints* más utilizados.
- Los modelos de datos efectivos en la práctica.
- El manejo de aspectos como la paginación y el filtrado.
- Las estrategias de versionado y documentación.



2. Proceso de adaptación

La adaptación no implica una reproducción exacta, sino una transformación que preserve las mejores prácticas mientras se ajusta a las necesidades específicas. Este proceso requiere:

- La simplificación de estructuras complejas cuando sea posible.
- La extensión de modelos de datos para incluir información adicional relevante.
- El ajuste de parámetros de consulta manteniendo la consistencia.
- La personalización de respuestas preservando formatos estándar.

3. Herramientas fundamentales para la implementación

El ecosistema OpenAPI proporciona herramientas esenciales para el proceso de adaptación y desarrollo.

[Swagger Editor](#) resulta fundamental durante la fase de diseño y adaptación, permitiendo la verificación en tiempo real de la validez de la especificación, la visualización previa de la documentación, la detección temprana de inconsistencias y la experimentación segura con diferentes estructuras.

[Swagger UI](#) es esencial para la fase de documentación y pruebas, ya que genera una interfaz de documentación interactiva automática, facilita las pruebas directas de endpoints, presenta ejemplos de uso claros y mantiene una experiencia consistente entre diferentes APIs.

Por último, [Swagger Codegen](#) es crucial para facilitar la adopción, pues genera código cliente en múltiples lenguajes de programación, minimiza errores de implementación, optimiza el proceso de integración y asegura la consistencia entre diferentes implementaciones.

Recursos actualizados para el desarrollo

La implementación exitosa de APIs basadas en OpenAPI requiere el uso de recursos actualizados y mejores prácticas. Se destacan especialmente:

El repositorio de APIs de referencia

Colección curada de especificaciones OpenAPI que sirven como referencia e inspiración

La comunidad de desarrolladores

Espacio de colaboración para la resolución de dudas y el intercambio de experiencias.

El catálogo de herramientas OpenAPI

Recopilación actualizada de utilidades para diferentes aspectos del desarrollo.

La implementación efectiva de una API de datos abiertos requiere **equilibrar la adopción de estándares probados con la adaptación a necesidades específicas**. El aprovechamiento de especificaciones existentes y la utilización de las herramientas adecuadas permite crear APIs más robustas, mantenibles y fáciles de adoptar por la comunidad de desarrolladores.

A modo de ejemplo, se detalla de forma muy resumida algunos pasos constructivos que ilustran **cómo se crea una operación utilizando Swagger**:

Después de ajustar determinados valores que constituyen la URI del acceso base a la API, como es el nombre de host y el número de puerto, se ajustan los elementos que definan operaciones sobre recursos. Para cada operación será necesario:

Asignar un nombre al controlador que se encargará de procesar la petición.

Asignar el método a la operación que se implementa al invocar este controlador. Por ejemplo, GET.

Definir el **conjunto de parámetros de entrada**. Por ejemplo: coordenadas de un punto en forma de Latitud, Longitud para que la petición devuelva paradas de bus en el entorno de 1 km a la redonda.

Definir **la forma que tendrá la respuesta**, es decir, se devolverá un código http 200 y la estructura de los datos devueltos denominada carga útil en el formato de salida previsto, por ejemplo, en formato JSON.

Por último, se ajustarán **otros valores de respuesta para indicar posibles errores y evitar mensajes por defecto**.

El resultado final de este proceso será similar al que se muestra en la siguiente imagen en la que se observa la documentación de la Empresa Municipal de Transportes (EMT) de Madrid.

La especificación completa se encuentra disponible en:

<https://apidocs.emtmadrid.es/>

Block_3_TRANSPORT_BUSEMTMAD - Stops Around Stop 0.0.0

This webmethod shows details of the stop request from EMTMADRID around one stop in a specific radius.

GET

`https://openapi.emtmadrid.es/v2/transport/busentmad/stops/aroundstop/<stopId>/<radius>/`

Header

Campo	Tipo	Descripción
accessToken	String	Current token generated from login

Parámetro

Campo	Tipo	Descripción
stopId	String	Stop number.
radius	String	meters around the stop.

Success 200

Campo	Tipo	Descripción
code	String	Result of operation (00=OK)
description	String	Description of success
datetime	String	Instant of current operation in server side
data	Array	Main structure of array values (If operation did well or empty array) contains (in distance order) below: (Object) geometry GEOJSON coordinates of stop (String) stopId Stop number (Integer) metersToPoint Distance on meters from point to stop (String) stopName Name of stop (Array) lines array with lines belong to stop, contains below: (String) nameA Name or Header A of line (String) nameB Name or Header B of line (Integer) metersFromHeader Distance of referred stop from the header of line (String) label public code of line (String) to Position into itinerary (header A to header B is to "B" and viceversa) (String) line internal code of line

```
{
  "code": "00",
  "description": "Success",
  "datetime": "2018-08-08T10:00:00",
  "data": [
    {
      "geometry": {
        "type": "Point",
        "coordinates": [
          -3.71793952686863,
          40.4384746558832
        ]
      },
      "stopId": "1",
      "metersToPoint": 100,
      "stopName": "A",
      "lines": [
        {
          "nameA": "A",
          "nameB": "B",
          "metersFromHeader": 100,
          "label": "1",
          "to": "B",
          "line": "1"
        }
      ]
    }
  ]
}
```

P5 – Recomendaciones sobre seguridad en las APIs

La seguridad de las API y la forma en la que ésta debe implementarse depende en cierta medida del tipo de datos que se transfiere y de la sensibilidad de los mismos. Es razonable pensar que **el nivel de protección y autenticación de usuarios necesario para una API transaccional (de lectura / escritura) debe ser mayor que el requerido para una API de solo lectura.**

En el contexto de los **Datos Abiertos**, el foco está en la **lectura de datos** y en cierta medida el método HTTP que se utiliza fundamentalmente es GET. No obstante, hay una serie de pautas que de forma general deben tenerse en cuenta a la hora de disponer recursos a través de la Web para su consumo por parte de terceros.

Como se ha comentado las API REST exponen recursos que son accedidos y manipulados a través del protocolo HTTP y cuyos **intercambios de información pueden ser cifrados** de tal forma que mantengan privadas las interconexiones entre sistemas.

De forma paralela es fundamental **preservar la integridad de las APIs** de usos abusivos o malintencionados de los servicios expuestos.

Por otro lado, hay que valorar el **nivel de autenticación requerido** a los usuarios de la API con el objetivo de mantener un equilibrio adecuado entre seguridad y maximización del uso de la API.

Antes de abordar la descripción de posibles formas de interacción es necesario recomendar una serie de pautas que siempre deben ser tenidas en cuenta independientemente del nivel de autenticación requerida a los usuarios de la API. Estas recomendaciones se exponen a continuación.

API Security es una iniciativa que promueve la organización de referencia a nivel internacional **Open Web Application Security Project (OWASP)** que se centra en la recomendación de estrategias y soluciones para entender y mitigar vulnerabilidades y riesgos de seguridad. La iniciativa propone 10 directrices fundamentales para la implementación de las API

Recomendaciones sobre el cifrado de los intercambios de información

Es aconsejable configurar de forma segura los servicios que reciben conexiones entrantes de clientes desconocidos, como es el caso de peticiones a una API. Para ello es aconsejable, entre otras cuestiones

- 1.** **Publicar servicios usando solo HTTPS (HTTP sobre TLS).** De esta forma la información viajaría encriptada. *Transport Layer Security* (TLS) es un protocolo que proporciona privacidad entre las aplicaciones y usuarios, o entre servicios de intercambio de información. Cuando un servidor y un cliente se comunican, la configuración adecuada de TLS garantiza que ningún tercero pueda interferir o alterar ningún mensaje.
- 2.** Usar las últimas **versiones estables de las librerías** (TLS y componentes software desplegados para la disponibilidad del servicio) evitando siempre la existencia de vulnerabilidades conocidas.
- 3.** Redirigir automáticamente a los usuarios que visitan la versión HTTP de un servicio en la Web a la **versión HTTPS**.
- 4.** **Habilitar política de seguridad HTTP Strict Transport Security (HSTS)** para establecer que todo el tráfico entre cliente y servidor debe realizarse sobre HTTPS.
- 5.** Adquirir los **certificados de servidor** en autoridades de certificación confiables.

Al margen del cifrado de datos, es recomendable prestar atención al problema conocido como **JSON injection** que es necesario prevenir. Como ya se ha señalado, es común el uso del formato de intercambio JSON en API RESTful. El problema está relacionado con el potencial para la inyección de código malicioso JavaScript en cadenas JSON. La vulnerabilidad se puede dar en dos vertientes: del lado del servidor, cuando los datos de un origen no confiable no son 'desinfectados' por el servidor o del lado del cliente cuando los datos de una fuente JSON no confiable no se desinfectan y analizan directamente mediante la función 'eval' de JavaScript. Por tanto, es altamente recomendable adoptar un *sanitizer* como **json-sanitizer de OWASP**. Igualmente, resulta conveniente hacer comprobaciones de sistemas con **pruebas de penetración específica**, como 'Freddy the Serial(isation) Killer'.

Autenticación de usuarios

A continuación, se describen tres enfoques de requisitos de autenticación de usuarios para la interacción con APIs. Hay otros enfoques como el uso de usuario y contraseña que no se está considerando en el detalle siguiente por ser menos eficiente. Los enfoques siguientes se muestran ordenados de menor a mayor nivel de seguridad:

1. Sin autenticación

En el contexto de los Datos Abiertos puede ser factible no requerir ningún tipo de autenticación de usuarios. Este enfoque se puede usar cuando el riesgo asociado con la API es poco significativo, por ejemplo, si **solo está permitido el uso del método GET para el acceso a los datos**. La desventaja de este modelo es que hace que sea difícil recopilar análisis efectivos y realizar un monitoreo para detectar actividades maliciosas.

Ejemplo de uso de API del Registro del Reino Unido que no requiere autenticación de usuario para su invocación:

<https://www.api.gov.uk/>

2. Autenticación por medio de claves API (API Key)

Una API Key es una cadena larga de caracteres que funciona como una especie de contraseña que tiene una caducidad. La autenticación mediante **API Key** requiere que la clave generada tenga que ser utilizada en cada interacción entre sistemas. La práctica habitual es que el desarrollador de aplicaciones obtenga una clave para su aplicación del proveedor de la API y la utilice dentro de su aplicación. Para obtener la clave. El desarrollador previamente realiza un proceso de registro. Las claves API son pares: **clave API pública e identificador API privado**, utilizado para validar la primera. La caducidad de la clave determina el periodo de tiempo consecutivo sin que el usuario tenga que volver a acreditarse.

Ejemplo de documentación de la API de la Empresa Municipal de Transportes (EMT) de Madrid en la que se explica el método de autenticación requerido:

https://apidocs.emtmadrid.es/#api-Block_1_User_identity

3. Autenticación utilizando OAuth (Open Authorization)

OAuth es un estándar abierto de autorización basado en la concesión de token de acceso que se implementa usando diferentes flujos de concesión de autorización que se desencadenan en función de determinadas situaciones o patrones. Estos patrones definen los diferentes tipos de interacción que una aplicación cliente puede realizar para obtener un "token de acceso" y, por lo tanto, acceder al recurso protegido de la API.

Hay cuatro tipos de flujos de autorización soportados por OAuth 2.0, denominados: código de autorización, autorización implícita, credenciales de contraseña del propietario del recurso y credenciales de cliente.

Los flujos de autorización que propone OAuth 2.0 se plantean para proporcionar a las organizaciones la flexibilidad de soportar diferentes escenarios de interacción de aplicaciones de clientes, por lo que es importante conocer el detalle sobre lo que ofrece cada flujo de autorización y que supone su implementación.

El "Código de autorización" es el modelo más utilizado y es el modelo más seguro. Por otro lado, la autorización implícita es el flujo menos seguro y la recomendación para su implementación está ligada a la autorización de acceso solo a información pública y solo cuando se usa el método GET.



Recomendaciones sobre limitación de uso

Las directivas de limitación de uso se realizan mediante ajustes de la tasa de peticiones o estableciendo cuotas por petición para una clave dada, donde la clave puede ser la dirección IP del cliente que realiza la petición o la clave API otorgada en el proceso de autenticación:

1. Tasa de peticiones por clave

Sirve para evitar picos de uso de la API limitando la tasa de peticiones a un número especificado por un período de tiempo establecido. Por ejemplo: 10 peticiones por segundo para una misma IP.

2. Cuota de uso por clave

En términos de volumen de llamadas y/o una cuota de ancho de banda, es decir, cantidad máxima de kilobytes permitidos durante el intervalo de tiempo.

Por ejemplo, 10000 peticiones y 40MB de datos en el periodo de 3600 segundos para una misma IP.

Cuando se desencadena alguna de estas directivas, el autor de la llamada recibe un código de estado de respuesta '429 Too Many Requests'.

Es importante tener en cuenta que la **monitorización de las métricas de servicio de la API** y en particular de consultas y limitaciones, es esencial para un preciso dimensionamiento este tipo de limitaciones y en general de todas las directivas de seguridad aplicadas.

P6 - Desacoplar servicios de Datos

A diferencia de otras APIs de uso interno o de uso externo de propósito especial para usuarios con los que se mantiene una relación contractual, conocidos como *partner APIs*, las **APIs públicas o APIs abiertas**, son las que se utilizan en el contexto de los Datos Abiertos, donde cualquier usuario, autenticado o no, puede hacer el uso que estime oportuno tantas veces como lo requiera.

Durante la etapa de diseño de la API es difícil conocer con anticipación y suficiente detalle las necesidades de los usuarios de un servicio de datos.

Es posible, además, que los reutilizadores implementen soluciones exitosas que requieran un volumen significativo de datos en momentos puntuales del día, por lo que es importante evitar implementar modelos de integración directos sobre los sistemas de información de las organizaciones publicadoras. En cualquier caso, a la hora de implementar una API, el arquitecto debe tener presente el grado de exposición de sus sistemas a los usuarios de la API.

Es recomendable, por tanto, **desacoplar accesos a datos públicos y privados** para evitar que el impacto del consumo externo de datos tenga repercusión sobre las operaciones internas de la organización sobre los mismos datos.

Este modelo de implementación permite, además, **implementar mejoras específicas centradas exclusivamente en la oferta de datos**.

En este sentido es fundamental **habilitar un canal de comunicación** con el sector reutilizador para identificar patrones de consumo y casos de uso implementados con el objetivo de anticipar potenciales riesgos derivados de sobrecarga o uso inadecuado de la API.

P7 - Facilitar el acceso a Datos en tiempo real

Los datos en tiempo real o cercanos al tiempo real, también denominados **datos sensibles al tiempo**, son los que están disponibles para su reutilización transcurridos, normalmente, del orden de milisegundos o unos pocos segundos desde su creación. La disponibilidad de este tipo de datos es fundamental para impulsar el desarrollo de aplicaciones y servicios en tiempo real.

Hay dos enfoques fundamentales que el publicador debe tener en cuenta sobre la forma en que los usuarios necesitan consumir los datos en tiempo real publicados vía API:

1. La necesidad de la última versión de un determinado dato que se actualiza periódicamente, por ejemplo, el tiempo estimado de llegada de un bus urbano a una parada.

2. La necesidad de acceder de forma permanente a un flujo de datos, por ejemplo, los datos de consumo eléctrico de una determinada instalación.

En el primer caso, denominado **pooling**, el dato se consume a partir de peticiones puntuales a demanda de los reutilizadores de forma infrecuente con el objetivo de obtener el dato más reciente disponible.

En el segundo caso, conocido como **streaming**, se requiere una implementación de transmisión continua de datos de tal forma que las aplicaciones cliente que desarrollan los reutilizadores reciban actualizaciones automáticas del servidor.

En cierto modo, el **streaming** supone invertir la naturaleza conversacional del modelo REST, donde se desencadena una respuesta a partir de una petición puntual, como es el caso de **pooling** cuyo requisito es mantener una frecuencia optima de refresco del dato. En un escenario de **streaming**, no hay una conversación propiamente dicha y lo que el protocolo necesita soportar es un **método para mantener el estado de una conexión abierta durante periodos largos de tiempo** a petición de varios clientes que pueden hacer la misma solicitud y luego enviar la misma información de respuesta de forma continua y globalmente a dicho grupo de usuarios.

Por esta razón, para implementar un servicio de **streaming** API se deben tener en cuenta otro tipo de modelos que se engloban en las **arquitecturas basadas en eventos**.

P8 - Especificar el formato de entrega de Datos usando las cabeceras HTTP

Como se ha comentado con anterioridad, los recursos pueden tener múltiples representaciones de los datos en función de las expectativas de los clientes. La solicitud de una determinada representación por un cliente, se denomina **negociación de contenido**.

De forma predeterminada, es recomendable **devolver contenido usando JSON o XML**.

La ventaja de usar **JSON** es que es una representación adecuada para una rápida serialización y deserialización de objetos de datos minimizando la transferencia de datos requerida, al tiempo que ofrece un mejor soporte que XML para representar estructuras de datos jerárquicas.

Además, es preferible que las respuestas de datos se estructuren como **objetos JSON** y no como *arrays* (los objetos JSON pueden contener arrays JSON), dado que los *arrays* pueden limitar la capacidad de incluir metadatos sobre la respuesta o añadir nuevas claves y valores en el futuro.

La forma de seleccionar la representación para una respuesta es **ajustando las cabeceras de petición y respuesta HTTP de servidor y cliente** de la forma adecuada. De esta manera, la API responde en el formato de datos que el cliente necesita y las aplicaciones cliente interpretan la salida correctamente.

Las **representaciones de los recursos** se especifican usando parámetros para indicar el *tipo de medios* o tipo MIME. Por ejemplo, se usarán los siguientes tipos para especificar los formatos más habituales: JSON (*application/json*); XM (*application/xml*) o CSV (*text/csv*).

- Del lado del servidor se utiliza el parámetro **Content-Type** para determinar la representación de la respuesta.
- Del lado del cliente, para determinar el tipo de representación que se requiere se utiliza el parámetro **Accept**.
- Del lado del cliente, también es posible especificar otros elementos esenciales relacionados con la codificación de la respuesta. Entre otros: el juego de caracteres: **Accept-charset**; el tipo de codificación: **Accept-encoding**; o el idioma esperado: **Accept-language**.

Si en la cabecera de petición del cliente no se especifica el parámetro *Accept*, el servidor puede responder con un tipo de representación preconfigurado por defecto.

Para ajustar con mayor precisión aspectos relacionados con la negociación de contenido es posible **expresar el nivel de preferencia entre diferentes opciones** utilizando el modificador 'q', que toma valores entre 0 y 1, en la petición como se observa en el siguiente ejemplo.

Para establecer JSON como formato preferente de intercambio, la interacción entre cliente y el servidor de la API es la siguiente: el cliente incluye en el cuerpo de la petición que realiza a la API, diferentes parámetros que personalizan la petición:

```
GET
https://datos.ejemplo.com/v1/licitaciones/contratos
Accept: application/json; q=1.0,
application/xml; q=0.8,
text/csv; q=0.2
Accept-Language: es; q=1.0,
en; q=0.7
```

En el ejemplo anterior se solicita los recursos completos de la colección de contratos y se está utilizando el modificador 'q' para determinar la preferencia de un formato y de un idioma respecto a otros. La cabecera puede contener otros elementos.

Por su parte, el servidor, entre otra metainformación, indica el formato de datos y la codificación que utiliza para responder a la petición:

```
HTTP/1.1 200 OK
Content-type: application/json;
charset=utf-8
```

Si el servidor no puede responder con la representación solicitada responderá con el código de estado HTTP 406 (No aceptable) que indica que el recurso solicitado no tiene una representación que sea aceptable para el cliente.

P9 - Usar cabeceras HTTP para el intercambio de información

Como se ha visto en la pauta anterior las cabeceras HTTP son el medio fundamental para establecer una adecuada negociación de contenido, pero además tiene otras funciones relevantes para enviar información adicional en el cuerpo de una petición o respuesta.

Las cabeceras tienen la forma de pares clave-valor y están formadas por su nombre (no sensible a las mayúsculas) seguido de dos puntos ':', y su valor (sin saltos de línea).

En este ejemplo se observa una cabecera de contexto de respuesta que indica la lista de métodos de peticiones aceptadas por un servidor. Esta cabecera la envía el servidor si éste responde con un código de estado 405 (Method Not Allowed):

Sintaxis: Allow: <http-methods>
Ejemplo de uso: Allow: GET, HEAD

Las Cabeceras pueden ser agrupadas de acuerdo a sus contextos:

- 1. Cabecera general:** Cabeceras que se aplican tanto a las peticiones como a las respuestas, pero sin relación con los datos que finalmente se transmiten en el cuerpo. Por ejemplo: "Connection", "Cache-control", etc.
- 2. Cabecera de consulta:** Cabeceras que contienen más información sobre el recurso que se solicita. Por ejemplo: "Accept", "Accept-charset", etc.
- 3. Cabecera de respuesta:** Cabeceras que contienen más información sobre el contenido, como su origen o el servidor (nombre, versión, etc.). Por ejemplo: "Content-length", "Content-encoding", etc.
- 4. Cabecera de entidad:** Cabeceras que contienen más información sobre el cuerpo de los recursos, como el tamaño del contenido o su tipo MIME. Por ejemplo: "Content-type", "Content-Language", etc.

El listado de elementos de cabecera es extenso e incluso personalizable. Para más información sobre la descripción de cada uno es recomendable consultar: <https://www.rfc-editor.org/rfc/rfc9110.html>

P10 - Definir interpretaciones comprensibles de códigos de estado

Los códigos de estado devueltos por el protocolo HTTP constituyen una información esencial para el desarrollador dado que sirven para distinguir entre respuestas satisfactorias e insatisfactorias a las invocaciones realizadas a la API.

Por lo tanto, las respuestas derivadas de las peticiones que realizan los clientes de la API deben ser **informativas, comprensibles por las personas y legibles por las máquinas**.

Es importante utilizar los códigos de estado del estándar HTTP en la implementación de la **gestión de respuesta de la API** ya que además son reconocidos por los marcos de desarrollo de aplicaciones habituales.

Los valores de los códigos de estado se agrupan en torno a cinco categorías principales de información. A continuación, se reproducen los más habituales y su interpretación:

1xx	Informativo	Solicitud recibida, el proceso de respuesta está en marcha.
2xx	Éxito	La petición del cliente se ha recibido, entendido y aceptado con éxito.
3xx	Redirección	Se deben tomar medidas adicionales para completar la solicitud .
4xx	Error del cliente	La solicitud contiene una sintaxis incorrecta o no se puede cumplir.
5xx	Error del servidor	El servidor no pudo resolver una solicitud aparentemente válida

Cuando ocurre un error, el cuerpo de la respuesta debe contener al menos los siguientes elementos:

- Código de estado estándar HTTP.
- Un código de error específico de la API, que servirá al cliente para reaccionar apropiadamente ante el error y al equipo de soporte de la API para identificar su origen. De esta forma cuando el cliente informe sobre el error al equipo de soporte usará este código específico y el equipo de soporte podrá trazar el problema y determinar exactamente qué salió mal y quién debe solucionarlo.
- Un mensaje de error legible por las personas, informativo y útil para el cliente de la API, sin ofrecer demasiados detalles técnicos. Es importante evitar revelar en la respuesta información del sistema, por ejemplo, referencias a componentes internos del sistema de información, ya que esto podría estar informando sobre posibles vulnerabilidades.
- Enlaces a detalles que complementen el mensaje y ayuden al cliente a encontrar más información.

Ejemplo de respuesta de manejo de errores devuelto por una API ante una determinada petición:

```
{  
  "Code": "400",  
  "errorCode": "12345",  
  "developerMessage": "Descripción detallada y simple del problema para orientar  
a los desarrolladores sobre cómo resolver el problema (por ejemplo: petición  
incorrecta: falta un campo o el valor incluido no es válido",  
  "userMessage": "Mensaje que puede transmitirse a los usuarios finales, si fuese  
necesario",  
  "moreInfo": "https://datos.ejemplo.com/v1/developer/path/to/help/for/12345"  
}
```


P11 - Incorporar una documentación completa

Los principales usuarios de las APIs son desarrolladores de aplicaciones y servicios y la **documentación de la API es el primer punto de contacto** para conocer la calidad y utilidad de la misma. Si la documentación de la API está completa, actualizada y resulta fácil de entender, favorecerá un uso más eficiente.

El **contenido de la documentación** debe reflejar inequívocamente cómo se realizan llamadas a la API, cuáles son los parámetros necesarios y cuál será la salida esperada. Asimismo, es fundamental reflejar con claridad que nuevas funcionalidades o resolución de problemas incorpora cada nueva versión disponible de la API.

Igualmente, la documentación de la API debe detallar sin ambigüedad qué recursos estará el usuario autorizado a utilizar y el método de autenticación de usuario utilizado para garantizar un acceso seguro, si éste es requerido.

Al menos, forman parte de la estructura de contenidos de la documentación de referencia de la API, los siguientes epígrafes:

1

Se es requerido, **método de autenticación** utilizado para autorizar el acceso y utilización de los recursos provistos.

2

Listado completo de las **peticiones que la API puede manejar, incluyendo el propósito de cada una, los parámetros permitidos, y la salida esperada.**

3

Ejemplos de uso de cada una de las peticiones posibles escritos en diferentes lenguajes de programación preferentes por la comunidad de desarrolladores en cada momento.

Relación de versiones de la API y las características incorporadas en cada una.

4

Información de **contacto con los promotores de la API** y mecanismo de contacto para proporcionar *feedback* sobre errores, sugerencias o preguntas sobre cualquier aspecto

5

Es recomendable acompañar la documentación de algún mecanismo *online* que permita **testar las peticiones y comprobar la respuesta de la API.**

6

Es recomendable utilizar **herramientas de edición** para confeccionar la documentación de la API. El uso de este tipo de herramientas facilita la generación de resultados como el que se muestra a continuación

The screenshot displays the OpenAQ API documentation interface. On the left is a sidebar with navigation links: 'About the API' (with sub-links 'About the API' and 'Terms of use'), 'Using the API' (with sub-links 'Quick start', 'API key', 'Rate limits', 'Pagination', 'Dates, datetimes and timezones', 'Geospatial queries', 'Errors', and 'Client libraries'), 'Examples' (with a highlighted 'Examples' button), and 'Resources' (with sub-links 'Countries', 'Instruments', 'Latest', and 'Licenses').

The main content area features two sections:

- Filtering locations by parameter**: Shows a terminal window with the following curl command:


```
curl --request GET \
  --url "https://api.openaq.org/v3/locations?parameters_id=2&limit=1000" \
  --header "X-API-Key: YOUR-OPENAQ-API-KEY"
```

 Below the code, it states: "Where parameters_id 2 is the ID for PM_{2.5} and limits the number results to 1,000 per page."
- Finding locations near a point**: Shows a terminal window with the following curl command:


```
curl --request GET \
  --url "https://api.openaq.org/v3/locations?coordinates=136.90610,35.14942&radius=12000" \
  --header "X-API-Key: YOUR-OPENAQ-API-KEY"
```

 Below the code, it states: "Where 136.90610,35.14942 is the central point in latitude,longitude and the radius is 12,000"

Ejemplo de documentación de la API de datos de calidad del aire de la organización OpenAQ: <https://docs.openaq.org/>

En el apartado 'Utilidades' de esta guía se detallan varias herramientas para confeccionar documentación y realizar pruebas de peticiones a las APIs.

Es relevante tener en cuenta que la calidad de la documentación de la API irá mejorando a medida que se incorpora el **feedback de los desarrolladores**, de ahí la importancia de mantener canales activos de comunicación con la comunidad reutilizadora.

P12 – Evitar rupturas de servicio o cambios abruptos

Al desarrollar aplicaciones o servicios que consumen APIs, los programadores dependen de las especificaciones definidas en la interfaz de la API, incluyendo su estructura de datos y formato de respuesta.

Es fundamental evitar modificaciones drásticas sin previo aviso. Cualquier actualización o mejora, por mínima que sea, debe ser notificada anticipadamente y con suficiente detalle para que los equipos de desarrollo puedan incorporar estos avances en futuras actualizaciones de sus aplicaciones.

La evolución de una API debería centrarse en **expandir su funcionalidad mediante nuevos métodos de invocación y opciones adicionales**, en lugar de alterar los comportamientos existentes. Las aplicaciones que no incorporen estas nuevas características deberían seguir funcionando sin alteraciones.

Cuando sea inevitable realizar un cambio radical, por ejemplo, debido a una modificación en la estructura de datos que resulte incompatible con el diseño original y pueda causar fallos en las aplicaciones cliente, se recomienda **desarrollar una API completamente nueva con una versión diferente**, utilizando URIs distintas para los recursos.

Es importante mantener activas las versiones anteriores para asegurar la continuidad de las aplicaciones que aún no han migrado a la última versión.

Ejemplos de cambios abruptos en una API

- Eliminar una petición
- Cambiar el método utilizado para hacer una petición
- Cambiar la URI de un recurso utilizado en una petición
- Agregar un parámetro requerido para una petición
- Cambiar el tipo de datos de un parámetro
- Cambiar el nombre de una clave en una respuesta del tipo clave-valor
- Cambiar la estructura de una respuesta XML
- Cambiar el tipo de datos de un valor en una respuesta

P13 – Evitar la degradación del rendimiento del servidor

Como se ha indicado anteriormente es difícil predecir a priori el uso que tendrá el servicio de datos a través de la API y es posible que se produzcan picos de demanda puntuales que pongan en riesgo el rendimiento de la API.

A continuación, se detallan dos aspectos importantes que impactan en el rendimiento de las APIs:

L A T E N C I A

El factor crítico de la latencia, entendida como el **tiempo transcurrido desde que se realiza una petición hasta recibir la respuesta de la API**, requiere un análisis cuidadoso. Su optimización debe evaluarse según cada escenario específico, considerando el tipo de datos y el uso previsto del servicio.

Es fundamental distinguir los requisitos de latencia según el caso de uso. Por ejemplo, los servicios que proveen datos en tiempo real, como la localización de un autobús, exigen tiempos de respuesta mínimos. En contraste, consultas a datos históricos, como documentos presupuestarios de ejercicios anteriores, pueden tolerar latencias mayores sin comprometer la calidad del servicio.

Para mejorar el rendimiento utilizando la infraestructura existente, se pueden implementar diversas estrategias de optimización:

- Diseñar una arquitectura que equilibre eficientemente las consultas pesadas (menos frecuentes) con las ligeras (más habituales).
- Implementar sistemas de caché para las solicitudes GET recurrentes.
- Reducir el tráfico de red y las consultas innecesarias al sistema.

C A C H I N G

Un elemento clave para la optimización es la implementación de técnicas de *caching*. Este enfoque busca **minimizar las consultas repetitivas mediante el almacenamiento temporal de datos**, siendo especialmente efectivo para información que se consulta frecuentemente pero se actualiza con poca regularidad. La implementación adecuada del caché requiere definir **políticas de expiración apropiadas** y mecanismos de invalidación cuando los datos cambien, pudiendo utilizar *headers* HTTP como Cache-Control y ETag para una gestión eficiente.

Esta práctica no solo acelera los tiempos de respuesta sino que también reduce significativamente la carga del servidor, optimizando el uso del ancho de banda y mejorando la escalabilidad general del sistema. La elección del tipo de caché (cliente, servidor o CDN) y su configuración debe alinearse con los **requisitos específicos de la API** y los patrones de acceso de los usuarios, permitiendo mantener un equilibrio óptimo entre rendimiento y frescura de los datos.

De cara a optimizar el rendimiento de la API, es crucial implementar **estrategias de control y optimización** que gestionen eficientemente las peticiones de gran volumen y establezcan límites apropiados en las solicitudes de los usuarios, garantizando así la disponibilidad y estabilidad del servicio.

Grandes volúmenes de datos

Al manejar volúmenes significativos de datos, se recomienda adoptar diversas técnicas de optimización. Es fundamental **estructurar los recursos** considerando la segregación estratégica de campos en recursos 'hijos', lo que facilita un acceso más fluido y eficiente a los datos. En situaciones donde sea beneficioso, se puede implementar la **compresión de archivos** mediante herramientas como GZIP, asegurando que los clientes estén preparados para procesar respuestas comprimidas. La **paginación de colecciones** resulta esencial para los recursos con fragmentos manejables, permitiendo a los usuarios realizar consultas más eficientes y controladas sin comprometer el rendimiento del servidor.

El **almacenamiento en caché de análisis** en formatos CSV puede parecer una solución atractiva para evitar que los usuarios procesen directamente los datos. Sin embargo, esta práctica debe evaluarse cuidadosamente considerando los casos de uso específicos y el perfil de consumo de la API. La implementación de esta estrategia debe balancear el uso de recursos del servidor con la mejora real en la experiencia del usuario.

Limitación de registros

En cuanto a la **limitación de registros**, es recomendable establecer un límite por defecto para cualquier consulta a la API, típicamente entre 10 y 50 registros por página. Si bien los usuarios pueden necesitar acceder a todos los datos, esto debe gestionarse mediante un parámetro específico en la URI de invocación o implementando paginación robusta. Esta limitación proporciona beneficios al no sobrecargar innecesariamente al usuario con todos los registros del dataset cuando simplemente está explorando los datos, además de prevenir problemas de rendimiento y *timeout* en las conexiones.

04 Implementación de APIs en catálogos de Datos Abiertos

Es recomendable que los catálogos de Datos Abiertos describan con precisión los servicios de Datos disponibles vía API y que además permitan el acceso a los conjuntos de datos que publican mediante su uso.

A continuación, se detalla la forma de abordar este doble objetivo:

Metadatos para describir APIs

La especificación del vocabulario para la catalogación de Datos Abiertos, [DCAT 2.0](#), incorpora una clase para definir propiedades útiles para describir puntos de acceso a APIs en el conjunto de metadatos de un catálogo de Datos Abiertos.

Aunque el uso de estos metadatos es opcional, es recomendable usarlos ya que **facilitan la interpretación y el consumo de los Datos Abiertos** a la vez que contribuyen a mejorar calidad general de los metadatos de un catálogo de Datos.

La clase [dcat:DataService](#) define una colección de operaciones que proporcionan acceso a uno o más conjuntos de datos o funciones de procesamiento de datos.

Las propiedades de esta clase son:

- [dcat:endpointURL](#), que se usa para indicar la localización del servicio de datos
- [dcat:endpointDescription](#), que permite aportar una descripción del servicio procesable por máquinas siendo posible expresarla siguiendo especificaciones estándares. En esta descripción se pueden incluir el detalle sobre operaciones y parámetros entre otra información esencial del servicio.
- [dcat:servesDataset](#), define la colección de datos que el servicio puede entregar.

Igualmente, la versión del perfil de aplicación para portales de datos en Europa, [DCAT-AP 2.0](#), se alinea con la actualización de DCAT 2.0 e introduce la capacidad de compartir servicios de datos añadiendo esta misma clase con sus propiedades como metadato opcional del catálogo de datos.

APIs para el acceso a Datos en catálogos de Datos Abiertos

Un requisito que debe satisfacer cualquier plataforma de Datos Abiertos es la disponibilidad de puntos de acceso que faciliten el consumo de los datos abiertos catalogados de forma automática.

Normalmente, estos puntos de acceso implementan **API Rest**, **SPARQL EndPoints** o **Servicios Web geográficos**. En los siguientes apartados se detallan los dos últimos.

Las plataformas de Datos Abiertos más comunes cuentan con APIs que facilitan la invocación de consultas parametrizadas que permiten diversas **operaciones de filtrado en el acceso a los datos disponibles**. Entre las operaciones habituales que se pueden utilizar usando la API de un catálogo de datos abiertos, es posible:

- Obtener el conjunto de recursos vinculados a un catálogo de datos.
- Obtener los metadatos vinculados a los datasets del catálogo.
- Exportar datasets aplicando diferentes filtros en formatos alternativos.
- Agregar datasets a los ya existentes en el catálogo.

Ejemplo de invocación a la API de CKAN del portal de Datos Abiertos de Madrid para obtener el dataset 'bicimad' en formato JSON:

<https://datos.madrid.es/egob/catalogo/300261-0-bicimad-disponibilidad.json>

Las plataformas de Datos Abiertos más comunes incorporan funcionalidades de API en sus versiones estándar:

- CKAN: el [DataStore de CKAN](#) provee una interfaz para almacenar y hacer invocaciones a los datos tabulares estructurados que alberga. Cada uno de los datasets tiene un punto de acceso dentro del DataStore. Soporta consultas tipo SQL.
- Socrata: [SODA API](#) permite la exportación de datos en varios formatos (JSON, XML y CSV, entre otros), y la parametrización de consultas a través de un dialecto denominado "*Socrata Query Language*" o "*SoQL*", inspirado en el lenguaje de consulta SQL.
- OpenDataSoft: Permite el acceso a todos los datos disponibles en la plataforma y retorna la información en formato JSON. Esta [API](#) incorpora especificaciones para soportar servicios web geográficos WFS y CSW que permiten la interrelación de esta plataforma con diferente software GIS. Soporta el dialecto ODSQL para consultas y filtrado de datos.

05 APIs para el acceso a Datos Enlazados

En el contexto de los datos enlazados (Linked Data) es posible recuperar datos modelados semánticamente utilizando un punto de acceso a los mismos y un lenguaje de consulta específico para este tipo de datos.

El lenguaje de consulta sobre la Web semántica y de los datos enlazados es SPARQL.

Dicho lenguaje cumple un rol similar a los lenguajes de consulta SQL en el contexto de las bases de datos relacionales, aportando una capa que **permite consultar datos que están en RDF, RDFs y otros estándares de la Web semántica.**

SPARQL está soportado por la mayoría de las APIs disponibles en el mundo de la Web semántica, como JENA o SESAME y por los motores de almacenamiento de datos semántico o *tripleStores* más conocidos, como Virtuoso o Fuseki.

Mediante estas soluciones, se pueden construir **puntos de acceso SPARQL que permiten consultar datos mediante APIs, protocolos web HTTP o interfaces gráficas de usuario.**

Ejemplos de puntos de acceso SPARQL implementados a partir de la tripleStore Virtuoso en diferentes portales de Datos usando interfaces de usuario:

<https://datos.gob.es/es/sparql>

<https://data.europa.eu/euodp/en/linked-data>

<https://dbpedia.org/sparql>

Alternativamente a los puntos de acceso disponibles mediante una interfaz de usuario como los citados en los ejemplos anteriores, es posible invocar peticiones de datos desde un navegador incluyendo los detalles de la consulta en la URL de llamada, aunque este método tiene algunas limitaciones. A continuación, se describe su utilización.

Una forma de hacer este tipo de llamadas y obtener los resultados tal como los devolvería un navegador usando la barra de direcciones, es utilizar el [comando curl](#) que permite la **transferencia de archivos de datos usando diferentes protocolos, entre ellos HTTP**. Este comando se puede utilizar para invocar una petición a cualquier API REST.

Por ejemplo, el siguiente comando puede ser ejecutado desde una ventana de terminal o de comandos y devuelve el listado de todas las URIs de todos los datasets contenidos en el catálogo datos.gob.es en un archivo CSV.

```
curl -k -o datasets.csv -G "https://datos.gob.es/virtuoso/sparql" --header "Accept:text/csv" --data-urlencode query="select distinct ?dataset where {?dataset a <http://www.w3.org/ns/dcat#Dataset>}"
```

En el comando curl es posible incluir **cualquier tipo de consulta SPARQL**, entrecomillada, a continuación de la cláusula *query=*. El parámetro *--data-urlencode* se encarga de codificarla.

Con el parámetro *-o* se indica el nombre del archivo devuelto por el comando curl. Es posible indicar el tipo de formato devuelto usando el parámetro *--header*.

Por ejemplo:

--header "Accept: text/csv" devuelve un archivo en formato CSV.

--header "Accept: application/sparql-results+json" devuelve un archivo en formato JSON.

La salida del comando anterior contenida en el archivo datasets.csv es:

"dataset"

"http://datos.gob.es/catalogo/a02002834-relacion-de-codigos-de-municipios-de-aragon1"

"http://datos.gob.es/catalogo/l02000011-centros-culturales"

"http://datos.gob.es/catalogo/l02000011-centros-sanitarios"

"http://datos.gob.es/catalogo/l02000011-municipios-de-cadiz"

"http://datos.gob.es/catalogo/l02000011-padron-de-espanoles-residentes-en-el-extranjero"

"http://datos.gob.es/catalogo/l02000011-patrimonio-de-diputacion-de-cadiz"

"http://datos.gob.es/catalogo/l02000011-retribuciones-personal-eventual-de-la-diputacion-de-cadiz"

"http://datos.gob.es/catalogo/l02000011-vertederos"

...

El comando curl no es la única forma de realizar una consulta como la indicada en el ejemplo anterior. También es posible realizar una petición SPARQL utilizando directamente el protocolo HTTP sobre un navegador.

Por ejemplo, la siguiente petición hace uso del [punto SPARQL de DBpedia](#) para recuperar la relación de grupos musicales.

```
https://dbpedia.org/sparql?default-graph-
uri=&query=PREFIX+rdf%3A+%3Chttp%3A%2F%2Fwww.w3.org%2F1999%2F02%2F22-rdf-syntax-
ns%23%3E%0D%0APREFIX+dbo%3A+%3Chttp%3A%2F%2Fdbpedia.org%2Fontology%2F%3E%0D%0APREFIX
+dbp%3A+%3Chttp%3A%2F%2Fdbpedia.org%2Fproperty%2F%3E%0D%0APREFIX+dbr%3A+%3Chttp%3A%2F
%2Fdbpedia.org%2Fresource%2F%3E%0D%0ASELECT+%3Fgrupo+WHERE%7B%0D%0A++%3Fgrupo+rdf%3At
ype+dbo%3AMusicalArtist+. %0D%0A%7D&format=text%2Fhtml&timeout=0&debug=on
```

Es conveniente señalar que los navegadores web utilizan ‘URL encoding’ para convertir los caracteres que componen una URL a un conjunto de caracteres ASCII válidos para ser transmitidos a través de la red. El encoding de URLs reemplaza caracteres ASCII no válidos, por ejemplo, espacios en blanco, por cadenas que contienen un carácter “%” seguido de dos dígitos hexadecimales. Aunque los navegadores realizan esta operación automáticamente, existen múltiples herramientas para hacer esta conversión.

La URL anterior es equivalente a la siguiente sentencia en SPARQL:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

```
PREFIX dbo: <http://dbpedia.org/ontology/>
```

```
PREFIX dbp: <http://dbpedia.org/property/>
```

```
PREFIX dbr: <http://dbpedia.org/resource/>
```

```
SELECT ?grupo WHERE {
```

```
    ?grupo rdf:type dbo:MusicalArtist .
```

```
}
```

La salida de la sentencia anterior obtenida como página web de un navegador es:

https://dbpedia.org/resource/The_Beatles

https://dbpedia.org/resource/Pink_Floyd

[https://dbpedia.org/resource/Queen_\(band\)](https://dbpedia.org/resource/Queen_(band))

https://dbpedia.org/resource/Led_Zeppelin

https://dbpedia.org/resource/The_Rolling_Stones

06 APIs para el acceso a Servicios Web Geográficos

Los estándares promovidos por el [Open Geospatial Consortium \(OGC\)](#) permiten la interoperabilidad entre sistemas de geoprocésamiento y el intercambio de la información geográfica, facilitando así la colaboración global entre organizaciones y la integración de datos espaciales de múltiples fuentes.

Dichos estándares dan soporte a servicios Web geográficos que son invocados por las Infraestructuras de Datos Espaciales o catálogos de Datos Abiertos, permitiendo el acceso uniforme y estandarizado a información geoespacial crítica para la toma de decisiones.

Entre otros, son servicios web geográficos relevantes los siguientes:

- **Los catálogos de servicios para la Web (CSW)** que implementan servicios de catálogo de datos y servicios geográficos basados en metadatos. Es la norma de referencia para exponer catálogos de registros de datos geoespaciales en XML. Estos catálogos son fundamentales para el descubrimiento y acceso a datos espaciales, permitiendo búsquedas avanzadas por ubicación, tema, fecha y otros criterios relevantes.

- **Los servicios de fenómenos geográficos (WFS)**, implementan los servicios de acceso a datos vectoriales en bruto, permitiendo acceder y consultar todos los atributos de un fenómeno o *feature* geográfico. Además, las versiones más recientes (WFS 3.0/OGC API Features) incorporan capacidades RESTful y formatos modernos como GeoJSON, facilitando la integración con aplicaciones web modernas.

- **Los servicios de acceso a cartografía (WMS)**, definen operaciones para la obtención de mapas como imágenes, la obtención de capacidades del servicio y la obtención de información sobre puntos del mapa. Se complementan con estándares más recientes como WMTS (Web Map Tile Service) para mapas en teselas, optimizando el rendimiento en aplicaciones web y móviles.

OGC está desarrollando OGCAPI para facilitar que cualquier organización pueda proporcionar datos geoespaciales a través de la web. Esta nueva implementación se está construyendo en base a la especificación OpenAPI, a partir de los estándares de servicio web OGC mencionados.

Ejemplo de invocación de un servicio web geográficos:

Catálogo de servicios Web del Instituto Geográfico Nacional:

<http://www.ign.es/web/ign/portal/ide-area-nodo-ide-ign>

Mapa de cultivos y suelos naturales de Castilla y León:

<https://datosabiertos.jcyl.es/web/jcyl/set/es/medio-rural-pesca/mapas-cultivos/1284807624344>

Referencias

En esta sección se muestra una relación no exhaustiva de algunas especificaciones, herramientas, comunidades de prácticas y medios de divulgación relacionados directamente con la implementación de APIs. Se trata de una **relación orientativa de referencia en el momento de la redacción de esta guía**, no obstante, la disponibilidad de información de carácter técnico y divulgativo sobre APIs en la Web es muy abundante por lo que se recomienda al lector orientar las búsquedas de documentación a temas específicos para ampliar el conocimiento sobre el diseño e implementación de APIs.

Especificaciones:

- OpenApis: <https://www.openapis.org/>
- GraphQL: <https://graphql.org/>
- Odata: <https://www.odata.org/>
- OGC: <https://www.ogc.org/>
- SPARQL: <https://www.w3.org/TR/sparql11-protocol/>
- DCAT 2: <https://www.w3.org/TR/vocab-dcat-2>
- DCAT 3: <https://www.w3.org/TR/vocab-dcat-3>

Herramientas implementar y generar documentación:

- Swagger: <https://swagger.io/>
- API blueprint: <https://apibuildprint.org/>
- RAML: <https://raml.org/>
- APIDocs: <https://apidocs.com/>
- io-docs: <https://github.com/mashery/iodocs>
- Apiary: <https://apiary.io/>

07

Referencias

Herramientas para testar APIs:

- SoapUI: <https://www.soapui.org/>
- Mockable: <https://www.mockable.io/>
- Runscope: <https://www.runscope.com/>
- Postman: <https://www.postman.com/>
- RESTClient: <http://restclient.net/>

Divulgación sobre APIs:

- ProgrammableWeb: <https://www.programmableweb.com/category/all/apis>
- Moesif: <https://www.moesif.com/>
- APIEvangelist: <http://apievangelist.com/>
- APIScene: <https://www.apiscene.io/>

Comunidades de prácticas sobre APIs en Gobiernos digitales:

- USA: <https://digital.gov/communities/apis/>
- Australia: <https://api.gov.au/>
- UK: <https://technology.blog.gov.uk/tag/api-design/>