DGTE-008. Especificación Diseño API

Versión 1.0

Área de Arquitectura

GERENCIA INFORMÁTICA
JOSEFA VALCÁRCEL, 44
28027-MADRID

Índice General

Índice de Tablas

Índice de Peticiones y respuestas HTTP

1 Introducción

Este documento normaliza el diseño de servicios conforme al estilo arquitectural Rest, conocidos comúnmente como Servicios REST.
Este estilo es descrito originariamente por Roy Thomas Fielding en su Tesis denominada "Architectural Styles and the Design of Network-based Software Architectures" (https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm).

1.1 Objetivo

El objeto de este documento es normalizar el diseño de Servicios REST. Esta normalización persigue los siguientes objetivos:

- Homogeneizar el aspecto de todos los Servicios con el objeto de que puedan formar parte del API público de la DGT. Los servicios de este API deben verse como un API único y, para ello, deben reunir un mismo conjunto de características que faciliten su consumo a los posibles usuarios.
- Facilitar el consumo interno de las distintas áreas de negocio. El aspecto homogéneo agiliza la comprensión del API y simplifica su integración.
- Agiliza su diseño al disponer de un conjunto de características predefinidas.
- Simplifica su publicación a consumidores, tanto internos como externos a la DGT.

1.2 Audiencia

Este documento está dirigido a las Áreas de Desarrollo y Negocio de la Dirección General de Tráfico.
El documento también es informativo tanto para el Departamento de Calidad como de Sistemas en la iniciativa de maduración del desarrollo de Servicios REST como mecanismo de publicación de API.

El documento presupone que el lector tiene un conocimiento acerca del diseño e implementación de un API REST.

1.3 Estructura del documento

Este documento está distribuido en 4 capítulos, con los siguientes contenidos:

- Capítulo 1: Introducción, contiene información relativa al propio documento
- Capítulo 2: Introducción y estado del arte tecnológico que rodea al ecosistema API
- Capítulo 3: Normativa DGT respecto al diseño de APIs.
- Capítulo 4: Anexo con el los aspectos más relevantes de la Normativa.
- Capítulo 5: Referencias utilizadas para la elaboración de este documento.

1.4 Glosario

1.4.1 Términos

En este apartado se incluyen, organizados por orden alfabético, los términos que se utilizan en el propio documento para facilitar su lectura y comprensión.

- **Excepción:** Cualquier error capturado y manejado por la aplicación durante su ejecución

1.4.2 Acrónimos

En este apartado se incluyen, organizados por orden alfabético, los acrónimos que se utilizan en el propio documento para facilitar su lectura y comprensión.

- **DGT**: Dirección General de Tráfico

2 Introducción a REST.

Cabe remarcar la naturaleza de REST como estilo arquitectural para la definición de APIs accesibles en red.

Si se quiere construir un API basado en red es fundamental que se respete la semántica del protocolo de transporte. En esta dirección, REST se definió pensando en respetar la semántica proporcionada por el protocolo HTTP/1.1. De esta forma, los agentes e intermediarios involucrados en una comunicación HTTP pueden entender el API sin necesidad de tener conocimiento de la aplicación.

Así, por ejemplo, si seguimos la semántica HTTP existente para un método GET dispondremos de los mecanismos de cacheado provistos por el protocolo HTTP. De esta forma, es fundamental que tengamos presente la semántica HTTP a la hora de diseñar nuestro API si queremos que éste pueda considerarse un API REST.

Esto no excluye que REST pueda ser de aplicación con otros protocolos de transporte, pero si queremos hacer un uso optimizado de las características proporcionadas por el protocolo, este último debe compartir en gran medida la semántica del protocolo HTTP/1.1.

2.1 Que caracteriza a un API REST.

El principio básico en el diseño de un API REST consiste en definir recursos, identificables mediante un nombre, que puedan ser manipulados usando un conjunto reducido de métodos.

En este sentido, la definición de una API REST implica:

- La identificación de recursos.
- La manipulación de recursos por medio de representaciones de los mismos.
- Intercambio de mensajes auto descriptivos y completos.
- El uso de hipermedia como ingeniería de estado de la aplicación.

2.1.1 Identificación de Recursos.

Cualquier información reconocible, esto es, que podamos identificar y reconocer por un nombre, puede ser un recurso.

Un recurso es un concepto que se corresponde con un conjunto de entidades. Los valores de dicho conjunto podrían ser representaciones de recursos o identificadores de recursos.

En ningún caso hay que identificar un recurso con el estado que puedan tener esas entidades en un momento dado.

REST usa un identificador de recurso para identificar un recurso involucrado en una interacción a través del API.

REST proporciona una interfaz genérica para acceder y manipular un recurso. Las acciones sobre un recurso se realizan usando una representación del mismo que captura el estado actual o deseado del recurso. Esta representación es intercambiada en la interacción con el API.

La representación es una secuencia de bytes junto con metadatos que describen dicha secuencia de bytes.

Una representación puede consistir en información, metadatos describiendo esa información, y, en ocasiones, metadatos describiendo, a su vez, a los metadatos. Estos últimos suelen tener como funcionalidad la verificación de la integridad del mensaje.

Los datos de control definen el propósito de un mensaje, tal como la acción requerida o el significado de la respuesta. Así, dependiendo de los datos de control, una representación dada podría corresponderse con el estado actual de un recurso, con el estado deseado o, incluso, el valor de algún otro recurso como sucede en una consulta con filtro.

El formato de la información en una representación es conocido como tipo de medio (***media type***).

La invocación es síncrona, pero tanto los parámetros de entrada como los de salida pueden ser proporcionados mediante '***streaming'***.

2.1.2 Manipulación de Recursos.

En la manipulación de un recurso disponemos de las acciones básicas tales como la consulta, creación, modificación, y borrado de un recurso. Estas acciones tienen una correspondencia

con los verbos propios del protocolo HTTP/1.1. Otras acciones que sean necesarias se definirán respetando la semántica proporcionada por el protocolo HTTP/1.1.

2.1.3 Intercambio de mensajes auto descriptivos y completos.

La naturaleza sin estado propia de REST nos obliga a que los mensajes contengan toda la información requerida para su interpretación.
El servidor, al recibir una petición, debe poder dar respuesta sin necesidad de disponer de información recopilada en peticiones previas El servidor no conserva estado de la interacción (‘***stateless’***).

2.1.4 Uso de hipermedia como ingeniería de estado de la aplicación.

Hipermedia nos facilita el descubrimiento de recursos a partir de recursos raíz o relaciones entre recursos facilitando la navegación entre los mismos.
Junto con la información propia del Recurso se proporciona información que facilita el descubrimiento de otros recursos, así como, los posibles subrecursos (recursos contenidos en otro recurso) y operaciones disponibles.

3 Normativa sobre Servicios REST.

Esta normativa establece una serie de reglas restrictivas a la hora de aplicar el estilo arquitectural REST. Estas reglas cubren aspectos tan diversos como la denominación de recursos, la definición de esquemas para crear las representaciones de recursos o los verbos HTTP a emplear según la semántica.

3.1 Niveles de Madurez.

En 2008, Leonard Richardson propuso el siguiente modelo de madurez para un API REST:

- **Nivel 0:** Define una única URI y todas las operaciones se implementan con el verbo HTTP POST.
- **Nivel 1:** Define URI independientes para cada recurso.
- **Nivel 2:** Uso de verbos HTTP para definir operaciones sobre recursos.
- **Nivel 3:** Uso de hipermedia (HATEOAS).

Este último nivel 3 se corresponde con un API Restful, tal y como lo define *Fielding* en su tesis.

Todo API que se publique en la plataforma de API Management de la DGT, deben cumplir al menos el Nivel 2 de madurez.

3.2 API y versionado.

Los API correspondientes a la organización deben ser agrupados en Productos para que sus recursos puedan ser expuestos a consumidores, tanto para el acceso desde aplicaciones cliente a los servicios Rest a través de un Gateway, como para su descubrimiento por la comunidad de desarrolladores y potenciales consumidores a través de un portal web de la organización.

Un Producto engloba un conjunto de recursos gestionados de forma conjunta. En este sentido, es el Producto el que está sujeto a versionado.

Un Producto será accesible mediante uno o varios puntos de acceso (‘***endpoint’***). Estos puntos de acceso tendrán información acerca de la versión del Producto publicado.
Puesto que el nombre completo de un Recurso contiene la denominación del Producto al que pertenece, los nombres de Producto deben ser definidos conforme a las reglas definidas para un segmento de URI (https://tools.ietf.org/html/rfc3986#appendix-A). Al mismo tiempo, el nombre debe componerse con el prefijo ‘api-’, como, por ejemplo, ‘api-sanciones’. Se recomienda no emplear nombres que hagan referencia a la estructura organizativa de la DGT sino nombres descriptivos de la funcionalidad proporcionada.
Criterios a la hora de definir Productos:

- Afinidad funcional. Es recomendable que las entidades de negocio referenciadas en un mismo contexto funcional estén accesibles a través de un mismo punto de acceso.
- Gestión de cambios. Es recomendable que las entidades fuertemente acopladas compartan un mismo ciclo de vida que facilite su versionado.

Como ejemplo de un Producto podríamos considerar el ámbito de las sanciones en el que nos encontraremos con recursos tales como infracciones o reclamaciones. El Producto podríamos llamarlo ‘api-sanciones’ y un hipotético recurso ‘reclamaciones’ se denominaría ‘api-sanciones/v1.0/reclamaciones’.

3.2.1 Versionado.

El versionado de un API se realizará mediante el versionado del Producto que lo engloba. El versionado se realizará conforme a la recomendación ‘Versionado Semántico 2.0.0’ (https://semver.org/lang/es/). En lo que respecta a esta recomendación, será de obligado cumplimiento todo lo referido a la versión ‘Major’, dejando al propietario del Producto la gestión de las versiones ‘Minor’ y ‘Patch’, de carácter opcional y como un recurso de información a los clientes del Producto de los cambios a ese nivel de detalle.

3.2.2 Recursos.

El API debe estar organizado entorno a recursos. Los recursos son entidades del negocio. La representación de un recurso debe ser independiente de su implementación interna.

Las entidades son agrupadas en colecciones identificables como recursos. Los recursos pueden contener subrecursos. Un subrecurso puede ser un recurso o una colección. Así, dispondremos de tres tipos de recursos, colecciones, recursos (entidades concretas) y subrecursos (entidades contenidas en otra entidad), todos ellos organizados de forma jerárquica.

3.2.2.1 Identificadores de Recursos.

Un recurso debe ser identificado mediante URI (Identificador de Recurso Uniforme).
La URI debe contener el nombre del Producto y la versión del mismo en distintos segmentos de la URI. Por ejemplo, 'api-examenes/v1.0/carnet-por-puntos'
Distinguiremos entre nombre de recurso cuando nos referimos a aquel segmento de la URI que identifica a un Recurso, y el identificador del Recurso que se corresponde con la URI completa que identifica al Recurso.

- Un nombre de recurso debe construirse en base a **nombres** y no verbos. Los nombres deben ir en **minúsculas**. Los nombres compuestos deben separarse mediante guiones (Kebab case - https://en.wikipedia.org/wiki/Letter_case#Special_case_styles).
- Los nombres deben ir en español.
- Los nombres deben ser definidos conforme a las reglas definidas para un segmento de URI (https://tools.ietf.org/html/rfc3986#appendix-A).
- Los nombres de colecciones deben ser nombres en plural.
- El nombre del Producto se denominará mediante el prefijo **"api-"**, seguida de un sufijo que indicará el nombre de la api que deberá estar asociado a la funcionalidad genérica de la misma.
- Un identificador de recurso debe construirse en base a la estructura jerárquica en la que se engloba un recurso. Se recomienda que un identificador de recurso este compuesto a lo sumo de tres niveles de anidamiento en la estructura jerárquica (colección/recurso/subrecurso). Un nivel de anidamiento mayor genera a futuro problemas para mantener el API. Sería preferible el uso de referencias a recursos, aunque esto implique una petición extra para recuperar el recurso.

Como ejemplos de identificadores de recurso tendríamos:

- api-sanciones/v1.0/infracciones (colección)

- api-examenes/v1.0/carnet-por-puntos (colección)
- api-sanciones/v1.0/infracciones/ref000010 (recurso)
- api-sanciones/v1.0/infracciones/ref000010/conductor (subrecurso)
- api-vehiculos/v1.0.0/vehiculos/657899/matricula (subrecurso)
- api-sanciones/v1.0.1/infracciones (colección)

3.2.2.2 Excepciones a la orientación a Recurso.

En casos muy excepcionales, se podría considerar el diseño de determinadas operaciones que difícilmente puedan ser asociadas a recursos como verbos HTTP sin la semántica propia de un Recurso. Así, por ejemplo, una operación para suma dos números podría implementarse como un método HTTP GET con la siguiente URI: /suma?operando1=99&operando2=1

Referencia:

no-recurso: (https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design)

3.2.3 Operaciones sobre Recursos.

Las operaciones sobre recursos se deben expresar mediante alguno de los verbos HTTP disponibles. En función de la semántica asociada a la operación haremos uso de uno u otro verbo.

Cabe mencionar dos características importantes a la hora de clasificar nuestras operaciones:

- **Segura**: se dice que una operación es segura si su ejecución no altera el estado del servidor.
- **Idempotente**: se dice que una operación es idempotente si produce los mismos resultados tanto si se ejecuta una como si se ejecuta varias veces.

3.2.3.1 Consulta de un Recurso.

Operaciones de consulta de un recurso deben implementarse con el verbo HTTP GET.

Esta operación es segura e idempotente.

El cuerpo de la respuesta contiene la representación del recurso.

En el caso de recursos de tipo colección, la petición normalmente incluirá un filtro de consulta.

Los atributos usados para componer el filtro de la consulta deben ser proporcionados por estos medios:

- Como parte del PATH de la URL.
- Como Parámetros de Query de la URL.

Códigos de respuesta:

- El código HTTP de respuesta correcta será un 200.
- Si el Recurso no existe el servidor devolverá un código HTTP 404(Not Found).

3.2.3.1.1 Consulta mediante el verbo HTTP POST.

Como excepción a la regla de usar el verbo GET para realizar una consulta, nos encontramos, en ocasiones, con consultas complejas que requieren de muchos atributos en la definición del filtro de consulta. La limitación en el tamaño de la URL del protocolo HTTP provoca que no podamos usar el verbo GET para implementar este tipo de consultas. En estos casos excepcionales, se puede hacer uso del método POST con el objeto de incluir el filtro de la consulta en el cuerpo de la petición HTTP.

Esta circunstancia se considera excepcional y debe ser consultada al Departamento de Arquitectura.

3.2.3.2 Creación de un Recurso.

3.2.3.2.1 Creación de un Recurso con Identificador proporcionado por Servidor.

Las operaciones de creación de un Recurso deben implementarse mediante el verbo POST.

Esta operación NO es ni segura NI idempotente.

El servidor asigna una URI al nuevo recurso y retorna esa URI al cliente. Esta operación es aplicada sobre la colección a contener el recurso.

El cuerpo de la petición contiene una representación completa del recurso a dar de alta.

El código HTTP de respuesta correcta será un 201(created).

Junto con el código se debe devolver un identificador(URI) del Recurso creado. Este identificador se devuelve en la cabecera HTTP "Location".

3.2.3.2.2 Creación de un Recurso con Identificador proporcionado por Cliente.

Las operaciones de creación de un Recurso, cuando el identificador del Recurso (URI) es proporcionado por el Cliente en la petición HTTP, deben implementarse mediante el método HTTP PUT. Estas peticiones se realizan sobre recursos individuales (no colecciones).
El cuerpo de la petición contiene una representación completa del Recurso. Si un recurso con ese identificador(URI) ya existe, éste es actualizado. De no existir el recurso, éste es creado si el servidor admite esta forma de crear recursos.
El código HTTP de respuesta correcta será un 201(Created).
Junto con el código se debe devolver un identificador(URI) del Recurso creado. Este identificador se devuelve en la cabecera HTTP "Location".
Esta implementación de la operación habilita la semántica de creación o actualización en una misma petición (***upsert***). El código HTTP de retorno nos permite conocer si el recurso ha sido creado o actualizado.
Si no deseamos habilitar tal semántica, deberíamos implementar la creación mediante el verbo POST incluyendo el identificador(URI) del recurso como un atributo de la representación enviada en el cuerpo de la petición POST. De esta forma, habilitamos que el cliente pueda evitar la creación de recursos donde sólo se desea actualizar los mismos.

3.2.3.3 Actualización de un Recurso.

En este contexto cabe distinguir entre la actualización completa de un Recurso y la actualización de una parte del mismo.

3.2.3.3.1 Actualización completa de un Recurso.

Las operaciones de actualización de todo el contenido de un Recurso deben implementarse mediante el verbo HTTP PUT.
Esta operación NO es segura, pero SI idempotente.
Estas peticiones se realizan, salvo excepciones, sobre un recurso individual a actualizar (no colecciones). Como excepción, cabría pensar, por ejemplo, de operación por lote que actualizan un conjunto de recursos con una única operación.
Como resultado, el Recurso es sustituido enteramente por el contenido nuevo proporcionado en la petición.
Códigos de respuesta:

El código HTTP de respuesta correcta será un 200 si la respuesta incluye una representación del Recurso en su cuerpo, o bien, 204(no-content) si la respuesta no incluye representación alguna en su cuerpo (respuesta sin cuerpo).

Si no es posible realizar la actualización del Recurso, el código HTTP de respuesta debe ser 409(Conflict).

3.2.3.3.2 Actualización parcial de un Recurso.

Las operaciones de actualización parcial de un Recurso deben implementarse mediante el verbo HTTP POST . Esta operación es de aplicación sobre recursos que no sean de tipo colección.

La limitación existente en el Gateway Corporativo hace inviable el uso del verbo PATCH para implementar este tipo de operaciones.

Esta operación NO es ni segura NI idempotente(atributos no atómicos) .

Códigos de respuesta:

El código HTTP de respuesta correcta será un 200 si la respuesta incluye una representación del Recurso en su cuerpo, o bien, 204(no-content) si la respuesta no incluye representación alguna en su cuerpo (respuesta sin cuerpo).

Si la petición, aun siendo una petición válida, no puede realizarse se debe devolver el código HTTP 422 (Unprocessable Entity). Este sería el caso, por ejemplo, en el que la aplicación del cambio supusiera dejar el Recurso en un estado incorrecto.

3.2.3.4 Borrado de un Recurso.

Las operaciones de borrado de un Recurso deben implementarse mediante el verbo HTTP DELETE.

Esta operación NO es segura, pero SI idempotente.

Estas peticiones se realizan sobre el recurso individual (recurso o subrecurso) a borrar. Esta operación no aplicará, salvo en casos excepcionales, sobre un recurso de tipo Colección.

El código HTTP de respuesta correcta será 204(no-content). Si no existe el Recurso, el código HTTP de respuesta debe ser 404(Not Found).

Sería recomendable devolver, junto con el código, el identificador(URI) del Recurso. Este identificador se devuelve en la cabecera HTTP "Location".

3.2.3.5 Operaciones asíncronas.

Cualquiera de las operaciones antes mencionadas podría implementarse de forma asíncrona si se considera procedente. En estos casos, es la respuesta la que difiere de lo descrito para cada una de estas operaciones.

El código HTTP de respuesta para estas peticiones asíncronas es 202 (Accepted).

Junto con el código se debe devolver un identificador(URI) de un Recurso que permita consultar el estado de la petición asíncrona. Este identificador se devuelve en la cabecera HTTP "Location".

Por ejemplo, el resultado de una petición de borrado podría ser este:

```
HTTP/1.1 202 Accepted
Location: /api/status/12345
```

Tráfico HTTP Ejemplo 1. Respuesta DELETE

y una consulta a ese recurso(/api/status/12345):

```
HTTP/1.1 200 OK
Content-Type: application/json
{
    "status":"In progress",
    "link": { "rel":"cancel",
"method":"delete",
"href":"/api/status/12345" }
}
```

Tráfico HTTP Ejemplo 2. Respuesta Estado de la petición

3.2.3.6 Otras operaciones.

Aquellas operaciones cuya semántica no encaje con la casuística previamente descrita se recomienda seguir los siguientes principios para determinar el verbo HTTP a usar:

- Operaciones que no impliquen un cambio de estado en el servidor se implementan mediante GET.
- Operaciones que impliquen un cambio, si son operaciones idempotentes se implementan mediante PUT. En caso contrario, implementarla mediante POST.

3.2.3.7 Negociación de Operaciones

El protocolo HTTP habilita un mecanismo para que un cliente pueda determinar los verbos HTTP disponibles para un recurso determinado. Para ello, habilita el verbo HTTP OPTIONS.

El comportamiento habitual de este método consiste en devolver una cabecera 'Allow' con la lista de verbos HTTP disponibles para el recurso especificado en la petición.

Petición:

```
OPTIONS /api/user HTTP/1.1
Host: example.org
```

Tráfico HTTP Ejemplo 3. Petición de negociación de operaciones

Respuesta:

```
HTTP/1.1 200 OK
Allow: GET, POST
```

Tráfico HTTP Ejemplo 4. Respuesta negociación de operaciones

3.2.3.7.1 CORS (Cross-Origin Resource Sharing)

Siempre que se habilite CORS para una aplicación, será requerido que se implemente esta operación ya que es invocada por los navegadores como paso previo a invocar un recurso en un dominio distinto al original.

La recomendación de seguridad es no habilitar CORS salvo que sea estrictamente necesario.

La mayor parte de los frameworks de desarrollo habilitan esta operación por configuración sin necesidad de implementarla.

3.2.4 Representación de Recursos.

La representación de un recurso puede contener el contenido, metadatos y metadatos sobre los metadatos. Desde una misma URI se pueden devolver distintas representaciones de un recurso.

Salvo para determinados recursos que contienen contenido, la mayor parte de ellos sólo contienen metadatos y metadatos sobre los metadatos.

En aquellos casos que contienen contenido, se puede usar el formato o tipo media que sea recomendable cuando se opere sobre una representación que contenga el contenido.

Aquellos recursos cuya representación sólo contiene metadatos y metadatos sobre metadatos, se debe usar JSON (*application/json*) como formato de representación.

El tipo media debe ser informado mediante la cabecera HTTP 'Content-Type' (Content-Type: application/json). Esto aplica tanto para el formato de las peticiones del cliente como para las respuestas del servidor.

Si el servidor no admite el tipo media usado en la petición, debe devolver el código HTTP 415 (Unsupported Media Type).

3.2.4.1 Negociación de Contenido.

El protocolo habilita un mecanismo de negociación de formato de contenido que permite a un cliente indicar los formatos de contenido que es capaz de interpretar.
Cuando el cliente envía una petición proporciona una lista priorizada de tipos media como los formatos de contenidos que puede interpretar. El servidor responde usando el primer formato de esa lista que se encuentre en la lista de formatos que el servidor es capaz de generar. Si el servidor no puede generar ninguno de esos formatos, debe devolver el código HTTP 406 Not Acceptable. La lista priorizada es proporcionada por el cliente mediante la cabecera HTTP 'Accept' (Accept: application/json).
La ingeniería REST es la encargada de realizar esta negociación de contenidos. En base a los métodos definidos donde se especifican los formatos de entrada y salida, la ingeniería determina el formato a proporcionar e invoca al método apropiado para generar dicho formato.

3.2.5 Códigos de Estado de Respuesta.

Toda respuesta debe proporcionar un código de Estado HTTP informativo del resultado de la petición.
El protocolo HTTP define un catálogo de posibles códigos de estado. Estos códigos se engloban en distintas categorías identificadas por su primer dígito. Así, tenemos categorías de códigos de respuesta correcta (**2xx**), redireccionamiento (**3xx**), error de cliente (**4xx**) y error del servidor (**5xx**).
Al describir las operaciones sobre recursos se especificaron algunas reglas que definían cuando utilizar algunos de estos códigos. No obstante, para una descripción detallada de todos los códigos de estado HTTP disponibles y su significado, se dispone del siguiente enlace: https://httpstatuses.com/

3.2.6 Tratamiento de Errores.

Cuando se produce un error de los englobados en las categorías 4xx o 5xx, el cuerpo de la respuesta debe consistir en un documento de tipo application/problem+json (https://tools.ietf.org/html/rfc7807).

La cabecera HTTP 'Content-Type' debe contener el valor indicado: application/problem+json.

El documento de respuesta debe incluir:

- **tipo**: referencia URI que identifica el tipo de problema. Podría ser publicada como un recurso que describa el tipo de error.
- **titulo**: un sumario del tipo de error. No debería cambiar entre ocurrencias de un mismo tipo de error salvo los cambios derivados de aplicar la adaptación local (localization) al texto.
- **status**: código de Estado HTTP (número).

Opcionalmente, este documento también puede incluir:

- ***detalle***: una descripción detallada y legible por el usuario del error específico.
- ***instancia***: referencia URI que identifica la ocurrencia específica del error.

El documento puede incluir otros metadatos que proporcionen información añadida acerca del error.

Por ejemplo:

```
{
  "tipo":
"https://example.net/validation-
error",
  "titulo": "Parámetros
incorrectos.",
  "status": 422,
  "detalle": {
    "invalid-params": [
      {
        "name": "edad",
        "reason": "debe ser un
entero positivo"
      },
      {
        "name": "color",
        "reason": "debe ser rojo
o negro"
      }
    ]
  }
}
```

Tráfico HTTP Ejemplo 5. Tratamiento de errores. Validación de campos

3.2.7 Paso de información a través de cabeceras HTTP

Se permitirá el uso de custom header en las peticiones y/o respuestas HTTP siempre y cuando la información a trasmitir no tenga relación directa con el recurso en cuestión. En caso de que la información a trasmitir si esté relacionada con el recurso, está información podrá introducirse dentro del body (ej: operaciones de tipo POST o PUT) o como parámetro in query (ej: operaciones de tipo GET) siempre teniendo en cuenta la semántica planteada en Rest.

Únicamente se podrán utilizar las custom headers definidas en el siguiente listado y para los fines para los que cada una de ellas se ha definido:

NOMBRE	Petición HTTP	Respuesta HTTP	Objetivo
X-DGT-RequesterData	X		Deberá contener información o metadatos relacionados con el peticionario de la solicitud (ej: uidUsuario, aplicacionID, serverName, etc). Ninguno de los datos transportados en esta cabecera podrá ser utilizado con fines de

			autenticación o control de acceso.

Tabla 1. Listado de custom headers permitidos

Nota: La columna “Petición HTTP” y “Respuesta HTTP” identifican en que pasos del flujo HTTP se permite el uso de cada cabecera.

En caso de que sea necesario utilizar una custom header que no se encuentre en la lista anterior ya que la información a trasmitir no concuerde semánticamente con el objetivo de ninguna de las cabeceras del listado, deberá realizarse una solicitud al departamento de Arquitectura Software para consensuar el nuevo nombre de cabecera en caso de que proceda.

3.2.8 Esquemas asociados a la Representación de un Recurso.

3.2.8.1 Nombres de Campos.

Los nombres de campos deben cumplir las siguientes reglas:

- Se debe usar minúsculas.
- Palabras compuestas o frases deben codificarse en formato 'snake case'. Esto es, debe separarse cada palabra simple por un guion bajo ('_'). Por ejemplo, "nombre_completo", "dia_semana".

3.2.8.2 Tipos de Campos.

Valores de tipo primitivo deben ser serializados siguiendo las reglas del estándar RFC4627 (https://tools.ietf.org/html/rfc4627).

3.2.8.2.1 Campos Nulos.

Campos nulos deberían ser excluidos del documento salvo que la semántica del campo requiera distinguir entre un campo no informado y un campo con un valor nulo.
Cadenas vacías no deberían considerarse como campos nulos. Su valor sería "".

En una actualización parcial(‘*patch’*) podría ser necesario indicar el campo con valor nulo para indicar que se desea borrar el contenido de dicho campo.

3.2.8.3 Fechas e Instantes de Tiempo.

Las fechas e instantes de tiempo deben definirse conforme a RFC3339 (http://xml2rfc.ietf.org/public/rfc/html/rfc3339).

Los formatos se corresponden con el estándar ISO 8601.

Para las fechas se debe usar el formato ‘ISO8601 full-date’ y para instantes de tiempo se debe usar el formato ‘ISO8601 date-time’ (https://xml2rfc.tools.ietf.org/public/rfc/html/rfc3339.html#anchor14).

	Representación
Fecha	YYYY-MM-DD
Instante de Tiempo	YYYY-MM-DDTHH:MM:SSZ

Tabla 2. Representación de fechas e instantes de tiempo

En casos excepcionales, se podrá hace uso de la representación con fracción decimal: ‘HH:MM:SS,ssZ’

Ejemplos de fechas e instantes de tiempo serían:

2018-08-23

2018-08-22T12:00:00Z (Hora UTC)

2018-08-24T23:30:59+0002

3.2.8.4 Duración e intervalos de Tiempo.

Para describir la duración de un evento se recomienda el uso del estándar ISO 8601. Este estándar establece el siguiente formato:

P[n]Y[n]M[n]DT[n]H[n]M[n]S donde cada elemento tiene la siguiente significación:

P: indicador de duración. Permite identificar el campo como una duración y, así, diferenciarlo de una fecha, instante de tiempo o intervalo.

Y: indicador de Año.

M: indicador de Mes.

W: indicador de Semana.

D: indicador de Día.

T: indicador de Tiempo.

H: indicador de Hora.

M: indicador de Minutos.

S: indicador de Segundos.

Así, por ejemplo, "P3Y6M4DT12H30M5S" representa una duración de 3 años, 6 meses, 4 días, 20 horas, 30 minutos y 5 segundos.

Por otro lado, para describir un intervalo de tiempo se recomienda también el uso del estándar ISO 8601

Este estándar contempla varias opciones:

- Fecha/Instante inicial y Fecha/Instante Final: "2007-03-01T13:00:00Z/2008-05-11T15:30:00Z"

- Fecha/Instante inicial y Duración: "2007-03-01T13:00:00Z/P1Y2M10DT2H30M"

- Duración y Fecha/Instante Final: "P1Y2M10DT2H30M/2008-05-11T15:30:00Z"

- Duración solamente. El contexto proporciona la información adicional para determinar el intervalo: "P1Y2M10DT2H30M,"

3.2.8.5 Otras Recomendaciones.

Evitar el uso de identificadores fuertemente acoplados a la implementación, como claves primarias de tablas de BBDD.

Evitar el uso de abreviaturas salvo que estas sean de uso muy común y, en consecuencia, sean fácilmente reconocibles.

3.2.8.6 Denominación de campos más comunes.

Esta lista pretende ser una recopilación de nombres de campos a emplear para referenciar conceptos comunes que suelen aparecer de forma recurrente en la definición de los esquemas de un API.

El objeto de recopilar estas denominaciones comunes es contribuir a una homogeneización de los nombres de campos empleados en el diseño de APIs.

Se recomienda el empleo de estos nombres en aquellos esquemas que incorporen el concepto correspondiente.

NOMBRE	NOMBRE DEL RECURSO. Palabra única.
titulo	Nombre del recurso. Puede contener espacios.
descripcion	Descripción breve del Recurso.
fecha_creacion	Fecha de Creación del Recurso. Es una fecha.
instante_creacion	Instante en el que se creó el Recurso. Incluye fecha y hora.
fecha_modificacion	Fecha de última modificación del Recurso. Es una fecha.
instante_modificacion	Instante en el que se modificó el Recurso por última vez. Incluye fecha y hora.
fecha_baja	Fecha de eliminación o borrado del Recurso. Es una fecha.
instante_baja	Instante en el que se eliminó o borró el Recurso. Incluye fecha y hora.

Tabla 3. Denominación de campos más comunes

3.2.9 Búsquedas, Filtros y Paginación.

Las consultas sobre colecciones suelen incluir mecanismos de selección de recursos mediante filtros de consulta y mecanismos de paginación y ordenación del conjunto resultante.

3.2.9.1 Filtros.

Las operaciones de consulta definen un filtro a aplicar sobre el conjunto total de recursos que forman parte de la colección.

Este filtro se puede define de dos formas:

- **Mediante segmentos añadidos al '*Path*' de la URL.**

  Por ejemplo, esta URI: *api-xxx/V1/sanciones/tipo/pendientes* implementaría una consulta sobre la colección sanciones aplicando un filtro de forma que la respuesta sólo contendría las sanciones de tipo 'pendientes'.

  El formato de la URL debe cumplir los requisitos descritos para la definición de un Recurso.

- **Mediante *parámetros de query* de la URL.**

  Por ejemplo, api-xxx/V1/sanciones?tipo=pendientes se comportaría como en el ejemplo definido mediante segmentos del 'Path'. Los campos definidos como parámetros deben seguir el mismo formato que se aplica en la definición de campos de un esquema (Ver apartado 0).

3.2.9.1.1 Consultas más comunes mediante alias.

Aquellas consultas de uso frecuente, es recomendable sean definidas mediante alias accesibles como un Recurso.

Así, por ejemplo, la consulta descrita anteriormente para obtener las sanciones pendientes podría ser implementada definiendo el siguiente recurso 'api-xxx/V1/sanciones/pendientes'

3.2.9.2 Paginación.

Las consultas sobre colecciones deben incluir mecanismos de paginación. La única excepción a la paginación sería para colecciones con un número reducido de elementos durante todo su ciclo de vida.

La implementación debe garantizar que cada respuesta contenga como máximo un número de elementos. La implementación debe proporcionar el siguiente mecanismo de configuración de la Paginación:

- Disponer del parámetro de query cuyo nombre es **'limite'**. Este parámetro permitirá, a la hora de realizar una petición, definir el tamaño de página a usar. Este parámetro no podrá tener un valor superior al máximo previamente descrito. Sería recomendable que su valor tuviese también un límite inferior.
- Disponer de un tamaño de página por defecto que aplicará cuando no se informe en la petición el parámetro 'limite'.
- Unido al tamaño de página, la implementación debe proporcionar un mecanismo para solicitar una determinada página del conjunto resultante de la consulta. Para ello, se dispone de dos mecanismos mutuamente excluyentes:
  - Un parámetro de query cuyo nombre es **'pagina'** que permite indicar la página solicitada.

- Un parámetro de query cuyo nombre es **'avance'** que permite indicar un número de recursos, tomados desde el inicio, a ignorar previamente a componer la respuesta. Este atributo es conocido normalmente como 'offset'.

3.2.9.3 Ordenación.

En consultas sobre colecciones no es obligatorio implementar mecanismos que permitan definir una ordenación específica para el conjunto resultante.
Siendo recomendación, en caso de implementarse este mecanismo, deberían seguirse las siguientes consideraciones:

- Emplear un parámetro de query con nombre **'orden'** para definir en la petición la ordenación deseada.
- Emplear un DSL simple que pueda ser codificado en el parámetro antes referenciado.

El *framework Spring*, por ejemplo, facilita un mecanismo para proporcionar una lista ordenada de campos por los que ordenar, así como, si el orden requerido para cada campo es ascendente o descendente. Un ejemplo de definición de una ordenación sería:

```
sort=nombre,asc&sort=edad,desc
```

De esta forma se ordenaría por nombre y edad, los nombres de forma ascendente y la edad descendentemente.

3.3 Documentación de API

La documentación debe ser conforme a la especificación OpenAPI 2.0 (https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.1.md).
Aunque la actual versión de dicha especificación es la 3.0.1, el reducido número de herramientas que implementan dicha especificación y las limitaciones impuestas por los productos de publicación de API hacen recomendable el uso de la versión 2.0.
La versión OpenAPI 2.0 se corresponde con la versión 2.0 de Swagger.
La documentación debe incluir los siguientes:

- Información acerca del API (***Info Object***).

- Recursos y Operaciones (Path Objects, Parameters Definitions Object).
- Códigos de Respuesta, cabeceras y esquemas usados para las Respuestas (***Responses Definitions Object***).
- Esquemas usados en el API (***Definition Objects***).
- Requisitos de Seguridad (Security Requirement Object, Security Definitions Object)
- Ejemplos de los valores que puedan tomar los parámetros.
- Ejemplos de los esquemas usados, en peticiones y respuestas.

Se recomienda el uso de elementos reusables para definir parámetros, esquemas de objetos y respuestas.

3.3.1 Etiquetado de operaciones

Las operaciones deben disponer de al menos una agrupación funcional que permita la identificación de todas las operaciones pertenecientes a un recurso. El mecanismo proporcionado para este cometido, son las etiquetas (“tags”) definidas a nivel de operación.
Referencia: https://swagger.io/docs/specification/grouping-operations-with-tags/

4 Seguridad.

4.1 Niveles de Seguridad y Criterios a evaluar.

Con el objeto de definir la seguridad a aplicar en el acceso a un API se han de evaluar distintos criterios que nos permitirán determinar el Nivel de Seguridad requerido para nuestro API. El Nivel de Seguridad es una categorización de los requerimientos de securización del acceso a nuestra API. Cada Nivel de Seguridad se implementará mediante una Política de Seguridad predefinida. Los criterios que a continuación se describen nos permitirán establecer el Nivel de Seguridad a aplicar en el acceso a nuestro API.

4.1.1 Clientes Confidenciales o Públicos.

El tipo de Cliente de nuestros APIs determinará, junto con otros criterios, la seguridad a aplicar a la hora de publicar nuestro API. En este sentido, cabe distinguir aquellos clientes considerados confidenciales de aquellos que son considerados públicos.

Un **Cliente Confidencial**: es aquel capaz de mantener las claves del Cliente de forma segura. Estos clientes disponen, junto con el identificador de cliente (Client ID) una clave (Client Secret) que les permite autenticarse de cara al proveedor del API. En esta categoría se engloban aquellos clientes que **residen** en un servidor protegido.

Aquellos clientes sin garantía de poder custodiar de forma segura una clave son considerados **clientes públicos**. En esta categoría se engloban aquellos clientes que residen en un entorno no securizado como las aplicaciones de móviles o las aplicaciones que se ejecutan en un navegador (javascript).

4.1.2 Aplicaciones propias y bajo control vs aplicaciones de terceros.

La naturaleza de las aplicaciones clientes de nuestro API determinará la interacción de éstas con nuestro API. Para aplicaciones propias y de confianza podría habilitarse un modelo delegado de autorización. En este modelo, la autenticación de la aplicación sería suficiente para

facilitar el acceso al API. La aplicación nos garantiza la autenticación y autorización del usuario final.
Con las vistas puestas a futuro, nuevos casos de uso en el que intervengan aplicaciones de terceros no confiables requerirán de una autorización de usuario no delegada sino controlada por los propios mecanismos de seguridad del API.

4.1.3 Tipo de Aplicación Cliente.

El tipo de implementación del Cliente condiciona el grado de seguridad que este puede ofrecer en su interacción con el servidor del API. Así, disponemos de aplicaciones WEB, aplicaciones móviles nativas, aplicaciones de escritorio o aplicaciones basadas en agente usuario (browser). Aquellos clientes pensados para ejecutarse en entornos más vulnerables como pueden ser las aplicaciones móviles deben estar sujetos a mecanismos de autenticación y autorización más seguros que aquellos ubicados en entornos protegidos.

4.1.4 Identidad de Usuario final o de la Aplicación.

En función de caso de uso será necesario disponer de la identidad del usuario final a la hora de acceder al API o, en algunos casos, tan sólo disponer de la identidad de la aplicación.
La naturaleza de la información accesible o, el uso o no de un modelo de autorización delegado determinará la necesidad de identificación del usuario final.

4.1.5 Naturaleza de la Información.

En este sentido, dispondremos de:

- Información pública accesible a cualquier Usuario.
- Información accesible por organismos de confianza.
- Información accesible de forma particularizada para cada Usuario.

En función de la naturaleza de información que maneje el API, se deberá optar por el mecanismo de seguridad apropiado.

4.2 Mecanismos de Seguridad soportados por API Manager DGT.

A continuación se describen los mecanismos de seguridad soportados por el API Manager instalado en la DGT y los escenarios en los que se son preceptivos su uso.
Independientemente de estos mecanismos, es obligatorio la protección de la capa de transporte mediante certificados (TLS) y opcionalmente, dependiendo de la sensibilidad de la información, se puede añadir un mecanismo de protección a la API mediante autenticación con certificado cliente.

4.2.1 User / Password

Este mecanismo soportado por el API Manager de DGT, no estará permitido salvo petición por necesidades de Negocio, y deberá ser previamente valorado por el Departamento de Arquitectura.

4.2.2 API Keys

El uso de API Keys estará habilitado para los siguientes escenarios:

- Cliente Confidencial
- Aplicaciones Propias o de terceros cuya naturaleza de la información sea pública y accesible a cualquier usuario o accesible por organismos de confianza (B2B).
- No es necesario conocer con seguridad la identidad del usuario final.

4.2.3 oAuth

El uso del protocolo oAuth 2.0 será de uso obligado en los siguientes escenarios:

- Cliente Público
- Aplicaciones de terceros cuya naturaleza de la información sea accesible de forma particularizada para cada usuario.
- Es necesario conocer con seguridad la identidad del usuario final.

5 Roles en Gestión de APIs (API Management).

Una plataforma de Gestión de APIs (API Management) abarca aspectos relativos a la administración de la infraestructura, la gestión de Productos y la gestión de consumidores de Productos. Esta diversidad de aspectos implica la intervención de múltiples actores con perfiles claramente diferenciados. A continuación, se describen los roles desempeñados por cada uno de los actores involucrados en la gestión de APIs.

5.1 Administrador de Topología.

Es el responsable de la infraestructura que da soporte a la plataforma de Gestión de APIs.
Sus funciones son:

- Instalación y administración de entornos operativos. Esto abarca elementos tales como gateways y hosting de portales.
- Instalación y administración de la Plataforma de Gestión de APIs. Esta administración abarca los aspectos relativos a la infraestructura.
- Configuración de las políticas de Seguridad en los distintos entornos. Estas políticas estarán disponibles durante el desarrollo de Productos para su selección y configuración.

5.2 Administrador de la Plataforma de Gestión de APIs.

Es el responsable de la definición y configuración de los Espacios de Publicación.
Sus funciones son:

- Definición de Espacios de Publicación.
- Gestión de usuarios administradores de Espacios de Publicación.
- Configuración de la Plataforma para cada Espacio de Publicación.

5.3 Administrador de Espacio de Publicación.

Es el responsable de un Espacio de Publicación específico.

Sus funciones son:

- Gestión de Usuarios con acceso al Espacio de Publicación administrado.
- Gestión de Contenidos publicados en el Espacio de Publicación administrado.

5.4 Propietario de Producto (Product Owner).

Es el responsable de un Producto desde su concepción hasta que éste deja de tener vigencia.

Sus funciones son:

- Definición del Producto y los APIs que lo componen.
- Gestión del Ciclo de Vida del Producto.
- Define el tipo de accesibilidad al Producto otorgado a las Comunidades de Desarrolladores, así como las condiciones aplicables a dicho acceso.

5.5 Desarrollador de API.

Es el responsable de implementar los distintos APIs que componen un *Producto*.

Implementa cada API conforme a la definición proporcionado por el *Propietario del Producto*.

Es responsable de garantizar que el API se corresponde con la definición proporcionada.

5.6 Administrador de Portal.

Es el responsable del Portal donde se publican las APIs para su consumo y está asociado a un *"Espacio de Publicación"*.

Sus funciones son:

- Gestión de Usuarios con acceso al Portal.
- Gestión de los contenidos estáticos que proporcionan el aspecto visual del Portal.

5.7 Administrador de Comunidades de Desarrolladores.

Los consumidores de APIs se organizan en Comunidades de Desarrolladores. Esto facilita la gestión de permisos de acceso a los Productos disponibles.
El administrador de Comunidades de Desarrolladores es responsable de gestionar estas comunidades y la asociación de los usuarios a las mismas.

5.8 Consumidor de API.

Es el cliente de un Producto. Es responsable de registrarse como consumidor de Producto y suscribirse a aquellos Productos que va a consumir.

5.9 Gestor de Suscripciones.

Es responsable de gestionar el acceso a los distintos Productos publicados. Esta gestión se realiza mediante la habilitación de suscripciones disponibles a los consumidores.
Sus funciones son:

- Gestión de derechos de acceso.
- Monitorización de los accesos a Productos.

6 Anexos.

6.1 Nomenclatura de Recursos.

ELEMENTO	NOMENCLATURA
recurso-completo	'/' producto '/' version '/' recurso
producto	'api-' Segmento de URI
versión	major.minor.patch (https://semver.org/lang/es/)
recurso	colección | recurso-simple | subrecurso
colección	Segmento de URI[1]
recurso-simple	colección '/' nombre-recurso
subrecurso	colección '/' nombre-recurso '/' nombre-subrecurso
nombre-recurso	Segmento de URI
nombre-subrecurso	Segmento de URI

Tabla 4. Cuadro resumen de la nomenclatura de recursos

Cabe mencionar dos características importantes a la hora de clasificar nuestras operaciones:

- **Segura**: se dice que una operación es segura si su ejecución no altera el estado del servidor.
- **Idempotente**: se dice que una operación es idempotente si produce los mismos resultados tanto si se ejecuta una como si se ejecuta varias veces.

VERBO HTTP	SEGURO	IDEMPOTENTE
GET	SI	SI
POST	NO	NO
PUT	NO	SI

[1] Segmento de URI: definido en RFC3986 (https://tools.ietf.org/html/rfc3986#appendix-A)

DELETE	NO	SI

Tabla 5. Características de los verbos http

6.2 Códigos de Respuesta http comunes.

CÓDIGO	DESCRIPCIÓN	OBSERVACIONES
200	Ok (CORRECTO)	
201	*Created* (CREADO)	Operaciones de creación de recursos
202	*Acepted* (Aceptada)	Operaciones asíncronas.
204	*No Conten*t (Sin Contenido)	Respuesta correcta pero no devuelve contenido. Operaciones ejecutadas correctamente y que no devuelven ninguna representación.
400	*Bad Request* (petición incorrecta)	La petición es incorrecta: json malformado, objeto no es correcto, parámetros incorrectos, etc.
401	*Unauthorized* (No autorizado)	No dispone de credenciales válidas para ejecutar la operación. Por ejemplo, no está autenticado o credenciales caducadas.
403	*Forbidden* (Prohibido)	No tiene acceso al recurso o no tiene permisos para realizar la operación sobre el recurso.
404	Not found (No encontrado)	Recurso no existe.
405	*Method Not Allowed* (Método no permitido)	El recurso no implementa dicho método o operación.
406	*NOT ACCEPTABLE* (no aceptable)	El recurso no puede proporcionar una representación válida para el cliente (punto 3.2.4.1 Negociación de contenido)
409	*Conflict* (conflicto)	El estado actual del recurso impide ejecutar la operación.
500	*Internal Server Error* (Error interno del servidor)	Servidor no disponible, error no recuperable.

Tabla 6. Códigos de respuesta http más comunes

6.3 Cabeceras http comunes.

NOMBRE	VERBOS HTTP	PETICIÓN/RESPUESTA
Location	POST, DELETE	Respuesta
Allow	OPTIONS	Respuesta
Content-Type	Todos	Petición y Respuesta
Accept	GET	Petición

Tabla 7. Cabeceras http más comunes

7 Referencias

- https://www.restapitutorial.com/
- https://martinfowler.com/articles/richardsonMaturityModel.html
- https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design
- https://apigility.org/documentation/api-primer/content-negotiation
- https://tools.ietf.org/html/rfc7231#section-4.3.7
- https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS
- https://apigility.org/documentation/api-primer/content-negotiation
- http://xml2rfc.ietf.org/public/rfc/html/rfc3339
- http://apiux.com/2013/03/20/5-laws-api-dates-and-times/
- http://restcookbook.com/Resources/pagination/
- https://tools.ietf.org/html/rfc5988#section-5.1
- https://swagger.io/docs/specification/2-0
- https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md
- https://semver.org/lang/es/