

Lineamientos y estándares para el desarrollo de API-REST

v1.1

OTIC

OFICINA DE TECNOLOGÍAS DE INFORMACIÓN Y COMUNICACIÓN – OTIC

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| CONTROL DE VERSIONES | | | | | |
| Versión | Elaborado por | Revisado por | Aprobado por | Fecha | Motivo |
| 1.0 | Hugo Zumaeta  Alexander Acurio  Jorge Loayza | Fernando Quispe | Francisco La Torre | 25/02/2020 | Primera versión del documento |
| 1.1 | Hugo Zumaeta | Fernando Quispe  Jhon Barrantes | Francisco La Torre | 10/09/2020 | API Management  API Gateway |

Contenido

[Contenido 2](#_Toc47018626)

[1. GENERALIDADES 3](#_Toc47018627)

[1.1 JUSTIFICACION 3](#_Toc47018628)

[1.2 ALCANCE 3](#_Toc47018629)

[1.3 DEFINICIONES Y ABREVIATURAS 4](#_Toc47018630)

[2. RESTRICCIONES DE REST (COMO ESTILO DE ARQUITECTURA) 5](#_Toc47018631)

[3. VENTAJAS DE USAR API REST 6](#_Toc47018632)

[4. RECOMENDACIONES GENERALES 6](#_Toc47018633)

[5. RECOMENDACIONES ESPECÍFICAS 8](#_Toc47018634)

[5.1 Usar sustantivos en plural para nombrar recursos 8](#_Toc47018635)

[5.2 Uso de métodos HTTP 8](#_Toc47018636)

[5.3 Evitar incluir verbos propios de HTTP en la URI 10](#_Toc47018637)

[5.4 Aplicación de verbos que denotan acciones 10](#_Toc47018638)

[5.5 Relaciones entre recursos 10](#_Toc47018639)

[5.6 Filtrado, ordenación y búsqueda 10](#_Toc47018640)

[5.7 Uso de parámetros de cabecera 11](#_Toc47018641)

[5.8 Manejo de estados 11](#_Toc47018642)

[5.9 Manejo de versiones del API. 13](#_Toc47018643)

[5.10 Estructura de las respuestas (response) JSON 13](#_Toc47018644)

[6. RESPECTO A LAS HERRAMIENTAS A UTILIZAR 16](#_Toc47018645)

[6.1 Para construir API 16](#_Toc47018646)

[6.2 Para probar y documentar API 16](#_Toc47018647)

[7. SEGURIDAD DE LAS API 16](#_Toc47018648)

[7.1 HTTPS 16](#_Toc47018649)

[7.2 Ofuscar el identificador del recurso 17](#_Toc47018650)

[7.3 JWT 17](#_Toc47018651)

[7.4 Límites de tráfico (cuotas) 17](#_Toc47018652)

[8. ANEXOS ¡Error! Marcador no definido.](#_Toc47018653)

[8.1 Anexo N° 01: Estilos para nombrar recursos. 24](#_Toc47018654)

[8.2 Anexo N° 02. Referencias 25](#_Toc47018655)

# GENERALIDADES

# JUSTIFICACION

Los sistemas de información distribuidos requieren mecanismos para poder interactuar con fuentes o sistemas externos. Uno de los mecanismos más difundidos que se tiene a disposición es la interfaz de programación de aplicaciones: API.

Las API permiten que sus recursos, productos o servicios se comuniquen con otros sin necesidad de saber cómo están implementadas, incorporando flexibilidad y escalabilidad, permitiendo ahorrar esfuerzo en el desarrollo de sistemas y expandir fácilmente las funcionalidades de los sistemas.

En el entorno actual, las aplicaciones de MINEDU deben estar construidas para proveer de información a tres tipos de cliente: otras aplicaciones de nuestra institución, aplicaciones de otras instituciones y al público en general. Este escenario exige que la creación y documentación de las API se realicen con buenas prácticas que conlleven al éxito de las implementaciones y la satisfacción de los usuarios finales.

Internet se ha convertido en el medio mundial por el cual las API intercambian información y servicios. El protocolo HTTP que es utilizado por las aplicaciones web y las API de éstas permiten compartir las peticiones y respuestas en diferentes formatos, de acuerdo a un protocolo establecido o a un estilo particular de arquitectura. Por lo general, estos formatos toman la forma de archivos XML o JSON. Con el auge del uso de las API, se desarrolló un protocolo denominado SOAP. Las API diseñadas para SOAP usan XML. Por otro lado, una especificación que viene siendo utilizada en las implementaciones modernas es REST. REST no es un protocolo, es un estilo de arquitectura que se monta también sobre el protocolo HTTP.

API REST está siendo usado a nivel mundial en todos los entornos de desarrollo de aplicaciones de todo tipo: páginas web, aplicaciones móviles, internet de las cosas (IoT), entre otras. Asimismo, el desarrollo de REST API bajo buenas prácticas permitirá una evolución hacia el desarrollo de aplicaciones altamente escalables horizontalmente, logrando satifascer a un creciente número de usuarios finales nacionales.

En la Unidad de Sistemas de Información de OTIC se requiere unificar la forma de desarrollar, declarar y documentar las API de REST a fin de que los equipos de proyectos de desarrollo de software, aplicando algunas convenciones, obtengan sistemas con orientación a servicios seguros y de fácil escalamiento. Estos lineamientos deben ser usados también por los servicios externos de desarrollo de software contratados.

Una ventaja principal en la aplicación de las buenas prácticas de desarrollo de API REST, promovidas por el presente documento, es que será posible identificar los servicios disponibles de las diferentes aplicaciones en función de las interfaces que exponen. De esta forma se promueve la reutilización de servicios, evitando volver a desarrollar funcionalidades que posiblemente ya se encuentren implementadas.

El presente documento de lineamientos, que pretende uniformizar los criterios de desarrollo de sistemas distribuidos, estará sujeto a actualizaciones permanentes por parte del equipo de Arquitectura de la Unidad de Sistemas de Información de la OTIC, en coordinación con los coordinadores y líderes técnicos de los principales proyectos de desarrollo de software.

# ALCANCE

Este documento, está dirigido a los integrantes de los equipos de desarrollo de software de la OTIC y a aquellos equipos de desarrollo de software que han sido contratados por el MINEDU y cuyo servicio viene siendo supervisado por gestores de la OTIC.

También está dirigido a los gestores de proyectos de desarrollo y mantenimiento de software en la OTIC, para que promuevan una visión de orientación a servicios en sus proyectos, con lo cual puedan estimar las actividades que correspondan y puedan gestionar los riesgos inherentes a la implementación de API-REST.

Este documento alcanza a las Unidad de Calidad y Seguridad de Sistemas de Información debido a que en su labor de acompañamiento y control de calidad, deberá hacer uso de los lineamientos y los formatos que son implementados.

Finalmente, los equipos de plataforma tecnológica podrán encontrar útil los numerales referidos al API Gateway en el cual se describen los considerandos del uso y administración.

# DEFINICIONES Y ABREVIATURAS

**API:** Interfaz de programación de aplicaciones. Es un conjunto de reglas (código) y especificaciones que las aplicaciones pueden seguir para comunicarse entre ellas: sirviendo de interfaz entre programas diferentes de la misma manera en que la interfaz de usuario facilita la interacción humano-software.

**API Management:** Es el proceso de crear y publicar APIs web, orientado al uso de políticas, controlando el acceso, coleccionando y analizando las estadísticas de uso y reportando el performance. El administrador de API es el rol encargado en el despliegue, monitoreo, optimización, mantenimiento y baja de APIs.

**Case sensitive:** Es una expresión usada en [informática](https://es.wikipedia.org/wiki/Inform%C3%A1tica) que se aplica a los textos en los que tiene relevancia escribir un carácter en mayúsculas o minúsculas.

**Endpoint:** Es un extremo de un canal de comunicación. Cuando una API interactúa con otras API, los puntos de contacto de esta comunicación se consideran “endpoint”. Un endpoint puede incluir una URL de un servidor o servicio. Cada punto final es la ubicación desde la cual las API pueden acceder a los recursos que necesitan para llevar a cabo su función.

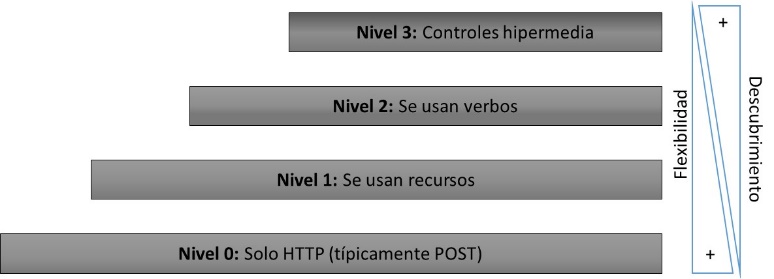
**HTTP:** Es un protocolo que funciona por peticiones y respuestas. Es un protocolo que no está basado en el estado, lo cual significa que el protocolo trata cada petición como una transacción independiente y no tiene en cuenta cualquier solicitud anterior.

**Idempotencia:** Aplicado a REST (realmente es a los métodos HTTP) significa que la ejecución repetida de una petición con los mismos parámetros sobre un mismo recurso tendrá el mismo efecto en el estado de nuestro recurso en el sistema si se ejecuta 1 o N veces.

**IoT:** Traducido al español como “Internet de las cosas” es un sistema de dispositivos de computación interrelacionados, máquinas mecánicas y digitales, objetos, animales o personas que tienen identificadores únicos y la capacidad de transferir datos a través de una red, sin requerir de interacciones humano a humano o humano a computadora.

**Métodos HTTP:** Son palabras o comandos que permiten comunicar al servidor lo que se quiere realizar con un recurso bajo una URI. Los métodos más importantes de HTTP (especialmente para hacer aplicaciones REST) son POST, GET, PUT, DELETE y HEAD.

**Modelo de madurez de Richardson:** Modelo propuesto por Leonard Richarson que permite entender el concepto REST y por tanto conseguir llevar a cabo mejores implementaciones del mismo.



**Petición (request):** Conjunto conformado por una URI, un método HTTP y un cuerpo (opcional) para acceder a un recurso.

**Recurso:** Un recurso es cualquier cosa distinguible y disponible en internet. Por ejemplo, un documento electrónico, una imagen, un servicio, etc.

**Respuesta (response):** Conjunto de datos que son emitidos por un servidor ante una petición y que está conformado por un encabezado y un cuerpo. El encabezado está conformado a su vez por un código de estado y un conjunto de parámetros.

**URI:** Son las siglas en inglés de Uniform Resource Identifier (en español identificador uniforme de recursos). La URI no se debe confundir con el URL, ya que las URI incluyen en su estructura a una URL. La diferencia fundamental es que mientras las URI identifican, las URL localizan. La estructura de un URL es como sigue:

|  |
| --- |
| {protocolo}://{dominio o hostname}[:puerto (opcional)]/{ruta del recurso}?{consulta de filtrado} |

**URL:** De las siglas en inglés Uniform Resource Locator o localizador uniforme de recursos. Es la dirección completa de una página web, mientras que la dirección del servidor se conoce como DNS (Domain Name System) o nombre de dominio. La URL es la dirección electrónica para poder acceder a un recurso en un servidor remoto (en un puerto específico). El tipo más común de URL es el de las páginas web, con la dirección http://, pero existen otras direcciones URL como ftp://, que proporciona la ubicación de red de un recurso FTP para poder transferir archivos.

**UUID:** El identificador único universal  es un número de 16 [bytes](https://es.wikipedia.org/wiki/Byte) (128 [bits](https://es.wikipedia.org/wiki/Bit)). Por tanto, el número de posibles UUID es de 1632, o unos 3,4 × 1038. En su [forma canónica](https://es.wikipedia.org/wiki/Forma_can%C3%B3nica) un UUID se expresa mediante 32 dígitos [hexadecimales](https://es.wikipedia.org/wiki/Hexadecimales) divididos en cinco grupos separados por guiones de la forma 8-4-4-4-12 lo que da un total de 36 caracteres (32 dígitos y 4 guiones). Por ejemplo:

550e8400-e29b-41d4-a716-446655440000

**Rest:** De las siglas en inglés Representational State Transfer (Transferencia de presentación de estado). Es una técnica o estilo de arquitectura de software usada para construir APIs que permitan comunicar servidores con sus máquinas clientes usando el protocolo HTTP mediante URIs definidas. REST es más simple y convencional que otras alternativas que se han usado en los últimos diez años como SOAP y XML-RPC.

# RESTRICCIONES DE REST (COMO ESTILO DE ARQUITECTURA)

Como se ha descrito en el numeral 1.1. en este documento, API REST es un estilo de arquitectura sino un conjunto de restricciones con el que se pueden crear aplicaciones distribuidas sobre HTTP. Las siguientes son las restricciones:

1. **Interface uniforme**

También conocido como “contrato uniforme” indica que un cliente antes de interactuar con un servicio REST debe acordar la identificación y la manipulación. Mediante la identificación, se especifica que debe ser posible conocer cada recurso que el servicio tiene para ofrecer. Mediante la manipulación se establece que existen un conjunto estándar de operaciones que se pueden aplicar sobre cualquier recurso con resultados predecibles.

1. **Cliente servidor**

En el marco de un desarrollo distribuido de aplicaciones, la aplicación cliente y la aplicación servidor deben evolucionar por separado y de forma independiente entre sí.

1. **Sin estado**

Todas las interacciones entre el cliente y el servidor no tiene estado. Esta es una característica del HTTP, pues todas las interacciones son independientes una de otra.

Si una aplicación requiere que el servidor “recuerde” su estado, como por ejemplo cuando un usuario inicia sesión una vez y realiza otras operaciones autorizadas después de eso, entonces; cada solicitud de la aplicación cliente debe contener toda la información necesaria para que el servidor atienda la solicitud.

1. **Caché**

El almacenamiento en caché se aplicará a los recursos cuando se requiera mejorar el rendimiento del sistema. Para esto, los recursos deben declararse almacenables en caché. El almacenamiento en caché se puede implementar en el servidor o en el lado del cliente.

1. **Sistema en capas**

Un sistema en capas exige una separación de responsabilidades, lo cual significa a su vez que cada capa interactúe con su capa adyacente sin pasar por alto alguna capa.

# VENTAJAS DE USAR API REST

1. **Separación entre el back-end y front-end.**

Entre las ventajas en la adopción de API REST se encuentra la posibilidad de separar y especializar los equipos de desarrollo en diferentes grupos que utilicen como mecanismo de interacción a las API. El equipo de back-end construye, orquesta y despliega las API y el equipo de front-end las consume.

1. **Agnóstico a la tecnología.**

Es posible implementar API REST utilizando diferentes herramientas, frameworks y lenguajes de programación y bases de datos. Es responsabilidad del equipo de arquitectura la identificación de la tecnología a ser utilizada en el desarrollo de API.

1. **Mejora en la performance.**

REST no debe usar sesiones, por lo que se dispone de mayor cantidad de memoria RAM. Asimismo, el uso de JSON, en lugar de HTML, por lo que las respuestas al cliente tienen menor volumen.

1. **Escalabilidad.**

La escalabilidad es un atributo de calidad muy buscado en la toma de decisiones arquitectónicas para productos de software. La forma más sólida de aumentar la probabilidad de que su producto continúe escalando a las necesidades del usuario final es a través de las API.

# RECOMENDACIONES GENERALES

Las siguientes son las recomendaciones generales para la asignación de nombres en el desarrollo de las API REST.

* API simple.

Cada API debe ser implementada para que pueda ser utilizada con facilidad por otros analistas y desarrolladores, por lo que debe ser un API simple y que el usuario deba recurrir a la documentación lo menos posible.

* Usar un mismo idioma.

A menos que sea requerido de otra forma, todas las API deben estar implementadas utilizando el idioma español.

* Evitar usar abreviaturas, en los nombres de los recursos y los parámetros. Por ejemplo, no debe usarse en el api: “facts” (por indicar facturas), “recingr” (por indicar recibos de ingreso), entre otros.
* Case sensitive y convención de estilos.

Para nombrar los recursos y sus atributos se debe elegir el estilo spinal-case. Una breve descripción de los diferentes estilos se encuentra en el anexo N° 01.

Asimismo, acorde a lo indicado en el RFC 3986 Uniform Resource Identifier (URI): Generic Syntax, las URL de las API son case sensitive, excepto el esquema y el host.

* Nomenclatura de la URI base.

Es como sigue:

**https://<ambiente>.api.<producto>.<dominio>/<version>/<sub-dominio>/<recursos>**

|  |  |
| --- | --- |
| **Etiquetas** | **Descripción** |
| <ambiente> | Indica los ambientes en los que están desplegados los servicios:  Desarrollo: “dev”  Control de calidad (certificación): “qa”  Producción: “prd” |
| <producto> | Indica el producto o sistema, generalmente siglas o acrónimos; al que se encuentra asociado el servicio, por ejemplo: “siagie”, “sisda”, “sinad”, “sigas”, “planin”, entre otros. |
| <dominio> | digital.minedu.gob.pe |
| <version> | v1, v2, v3, v4 |
| <sub-dominio> | Es el nombre del servicio implementado, por ejemplo: “identidad”, “evaluacion”, “capacitacion”, “gestion”, “contratacion” |
| <recursos> | Son los recursos, por ejemplo: “estudiantes”, “docentes”, “equipos”, “instalaciones”, “materiales”, |

Por ejemplo:

* + dev.api.sisda.digital.minedu.gob.pe/v1/pagos/planillas
  + dev.api.co.digital.minedu.gob.pe/v1/capacitacion/docentes
  + qa.api.cc.digital.minedu.gob.pe/v2/gestion/materiales
  + prd.api.sisda.digital.minedu.gob.pe/v1/pagos/planillas
* Nomenclatura en el cuerpo (body) del request y en el response.

Se debe utilizar lowerCamelCase para las etiquetas del JSON en el que se utilizan, por ejemplo:

{

"conceptoId": "34",

"conceptoDescripcion": "Pago de matrícula"

}

* No utilizar extensiones de archivos, en su lugar utilizar la cabecera "Content-Type" y "Accept". Por ejemplo, la URI /boleta-notas/234.pdf no es correcta.
* Utilizar la cabecera "X-Total-Count" para devolver el total de los recursos almacenados.

# RECOMENDACIONES ESPECÍFICAS

# USAR SUSTANTIVOS EN PLURAL PARA NOMBRAR RECURSOS

Se debe usar el sustantivo en plural que describe cada recurso:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Recurso** | **Verbos HTTP** | | | |
| **POST** | **GET** | **PUT** | **DELETE** |
| /cursos | Crear un curso | Mostrar todos los cursos | Actualizar los datos de cursos | Borra todos los cursos |
| /cursos/a182h1a1 | No aplica | Mostrar el curso con el uuid a182h1a1. | Actualizar el curso con el uuid a182h1a1. | Borrar el curso con el uuid a182h1a1. |
| /docentes | Dar de alta a un docente | Mostrar todos los docentes. | Actualizar los datos de docentes. | No aplica. |
| /docentes/134 | No aplica | Mostrar el curso con el id 134. | Actualizar el docente con el id 134. | Borrar el docente con el id 134. |

El criterio para elegir el sustantivo correcto para el API, es el lenguaje ubicuo del proyecto construido con una orientación al dominio.

El uso correcto de recursos en API REST nos posiciona en el Nivel 1 del Modelo de Madurez de Richardson.

# USO DE MÉTODOS HTTP

Los métodos HTTP proporcionan la contraparte de la acción del recurso basado en sustantivos. HTTP define un conjunto de métodos de petición para indicar acciones a realizar para determinados recursos.

El uso de métodos HTTP requiere manejar dos conceptos importantes:

* Métodos seguros.
* Métodos idempotentes.

Un método HTTP es considerado seguro si no altera el estado de la aplicación. En otras palabras, un método es seguro si conduce a una operación de ‘sólo lectura’, por ejemplo: OPTIONS, GET o HEAD. Todos los métodos seguros son también a su vez idempotentes.

Por otro lado, un método es idempotente si al realizar una operación muchas veces da el mismo resultado cual si se hubiese realizado la operación una sola vez, en otras palabras; si una solicitud HTTP idéntica puede realizarse una o demasiadas veces consecutivamente obteniendo el mismo resultado dejando al servidor en el mismo estado. Es muy importante que el desarrollador de API mantenga este criterio dado que se orienta a la implementación de modernos, seguros y escalables esquemas de arquitectura de aplicaciones.

Los métodos HTTP utilizados con más frecuencia son:

| **Método HTTP** | **Operación** | **Es idempotente** | **Es seguro** |
| --- | --- | --- | --- |
| POST | Creación | No | No |
| GET | Lectura | Si | Si |
| PUT | Actualización/Reemplazar | Si | No |
| PATCH | Actualización/Modificar | No | No |
| DELETE | Eliminación | Si | No |
| HEAD | GET sin cuerpo de respuesta | Si | Si |

Mensajes de retorno recomendados por cada método HTTP en combinación con la URI del recurso:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Método HTTP** | **CRUD** | **Colección completa** | | | **Ítem específico** | | |
| **(ejm. /estudiantes)** | | | **(ejm. /estudiantes/{id})** | | |
| **Código** | **Significado** | **Respuesta** | **Código** | **Significado** | **Respuesta** |
| **POST** | Crear | 201 | Creado | Cabecera ‘Location’ con link a /estudiantes/{id} conteniendo el nuevo ID | 404 | No encontrado |  |
| 409 | Conflicto | Si recurso ya existe |
| **GET** | Leer | 200 | Correcto | Lista de estudiantes. Usa paginación, ordenamiento y filtrado para navegar listas muy grandes | 200 | Correcto | Estudiante único |
| 404 | No encontrado | Si ID no es encontrado o no es válido |
| **PUT** | Actualizar/Reemplazar | 405 | Método no permitido | A menos que se quiera actualizar/reemplazar cada recursos de la colección entera | 200 | Correcto |  |
| 204 | Sin contenido |  |
| 404 | No encontrado | Si ID no es encontrado o no es válido |
| **PATCH** | Actualizar/Modificar | 405 | Método no permitido | A menos que se quiera actualizar/modificar la colección entera | 200 | Correcto |  |
| 204 | Sin contenido |  |
| 404 | No encontrado | Si ID no es encontrado o no es válido |
| **DELETE** | Eliminar | 405 | Método no permitido | A menos que se quiera eliminar toda la colección - no recomendable | 200 | Correcto |  |
| 404 | No encontrado | Si ID no es encontrado o no es válido |

**NOTA:**

Lo más usual es que un POST cree un recurso cuyo identificador es una valor que es asignado por el motor de persistencia (este valor es comúnmente conocido como un “correlativo”, “identity” o “autonumérico”). Sin embargo, cuando el identificador del recurso a ser creado es asignado por el cliente, es posible realizar la creación de dicho recurso a través un PUT.

El uso correcto de métodos en API REST nos posiciona en el Nivel 2 del Modelo de Madurez de Richardson.

# EVITAR INCLUIR VERBOS PROPIOS DE HTTP EN LA URI

Se debe **evitar** usar verbos en la URI tales como “get”, “save”, “obtiene”, “lista”, etc, por ejemplo, no se debe usar:

|  |  |  |
| --- | --- | --- |
| /getUsers | /borrarCurso | /setNewUsuario |
| /aplicarCambio | /conseguir | /updateEstudiante |

Sin embargo, como se verá más adelante, los únicos verbos permitidos son: “buscar” u “ordenar” o sus equivalentes en inglés, en el ámbito del uso de filtros u ordenadores.

# APLICACIÓN DE VERBOS QUE DENOTAN ACCIONES

Es posible utilizar verbos que denotan acciones, siempre que no denoten operaciones CRUD, por ejemplo: activar, desactivar, matricular, confirmar, firmar, priorizar, cerrar, entre otros.

Estos verbos, son siempre utilizados al final de la URI y aplica para un único recurso o para la colección. Por ejemplo:

|  |  |
| --- | --- |
| **/estudiantes/as3jas123/matricular** | Matricula al estudiante con el uuid **as3jas123** |
| /colegios/812as13/cerrar | Cierra el colegio con el uuid 812as13 |
| /colegios/2020/iniciar | Inicia todo el periodo anual de los colegios. |

# RELACIONES ENTRE RECURSOS

Es posible realizar operaciones sobre recursos relacionados, por ejemplo cuando requerimos construir una API que relacione dos recursos como por ejemplo cursos y estudiantes. Por ejemplo:

|  |
| --- |
| …/cursos/{id}/estudiantes |
| Obtiene todos los estudiantes que se han matriculado en el curso con identificación “:id”. |

El slash (“/”) debe usarse para indicar un relación jerárquica, al final de una URI no aporta nada y no se debe incluir.

|  |
| --- |
| /cuadernodecontrol.minedu.gob.pe/cuadernos  /cuadernodecontrol.minedu.gob.pe/cuadernos/{id}/dias/{id} |

Si bien es cierto, se pueden relacionar varios recursos en la URL, se deberá utilizar como máximo dos, ya que de esta forma se mantiene la característica de simplicidad en la API. Para la implementación de relaciones con un mayor número de recursos, se debe implementar un nuevo URI o utilizar el pase de parámetros por el operador “?”.

# FILTRADO, ORDENACIÓN Y BÚSQUEDA

Estas operaciones se utilizan con el método GET

* **Filtrado**:

Se basa en el uso de parámetros por cada campo que implemente un filtro siempre cuando el retorno es una colección. Por ejemplo:

|  |
| --- |
| /cuadernodecontrol.minedu.gob.pe/.../cuadernos?seccion=S3A |
| Filtra solo los cuadernos de control de los estudiantes de la sección de tercero A de secundaria. |

* **Búsqueda:**

Es posible implementar las búsquedas como si fuera un recurso. Ya se ha mencionado que no se puede utilizar verbos en la URI, pero este lineamiento es una excepción.

|  |
| --- |
| /cuadernodecontrol.minedu.gob.pe/.../cuadernos?seccion=S3A |
| Filtra solo los cuadernos de control de los estudiantes de la sección de tercero A de secundaria. |

* **Ordenación**:

De forma similar al filtrado, el parámetro genérico “sort” se debe usar para describir reglas de ordenamiento. Organiza el ordenamiento según la lista de campos separados por comas, aplicando el signo negativo (-) indicando orden descendente de ser necesario.

|  |
| --- |
| GET /avisos?sort=-prioridad |
| Ordena los avisos según la prioridad de forma descendente. |

|  |
| --- |
| GET /avisos?sort=-prioridad, fecha-creacion |
| Ordena los avisos según la prioridad de forma descendente y por fecha de creación |

# USO DE PARÁMETROS DE CABECERA

Los parámetros son opciones que se pasan a través del end-point (como por ejemplo especificando el formato o la cantidad de recursos a ser retornados) para influenciar en la respuesta. API REST maneja cuatro tipos de parámetros:

* De cabecera
* De Ruta
* Cadena de consulta (query string)
* Cuerpo de pedido (request body)

Los encabezados HTTP son una parte importante de la petición y de la respuesta de la API ya que llevan la información siguiente:

* Sobre el cuerpo de la petición y la respuesta
* La autorización de la petición.
* El manejo en caché de la respuesta.
* El manejo de cookies en la respuesta.

Además de las categorías anteriores, los encabezados HTTP también contienen mucha otra información sobre los tipos de conexión HTTP, servidores proxy, etc. La mayoría de estos encabezados son para la administración de conexiones entre cliente, servidor y servidores proxy y no requieren validación explícita mediante pruebas.

# MANEJO DE ESTADOS

Como nuestras API son utilizadas por otros programadores se enfrentan ante una caja negra, por lo que es indispensable suministrar la información suficiente referida al manejo de los estados, en especial de los estados que expresan que ocurrió un error específico.

De la misma forma como utilizamos los verbos HTTP para manejar operaciones, también se deben utilizar los códigos HTTP para manejar los estados. Los códigos a ser manejados para las implementaciones de API son[[1]](#footnote-1):

| **Código de respuesta** | **Descripción** |
| --- | --- |
| 200 OK | Solicitud aceptada; la respuesta contiene el resultado. Este es un código de respuesta general a cualquier solicitud. En las solicitudes GET, el recurso o datos solicitados están en el cuerpo de la respuesta. En las solicitudes PUT o DELETE, la solicitud fue satisfactoria y la información acerca del resultado (como los identificadores de recursos nuevos o los cambios en el estado del recurso) se puede encontrar en el cuerpo de la respuesta. |
| 201 CREATED | Las operaciones PUT o POST devuelven este código de respuesta e indica que se ha creado un recurso de forma satisfactoria. El cuerpo de la respuesta podría, por ejemplo, contener información acerca de un nuevo recurso o información de validación (por ejemplo, cuándo se actualiza un activo). |
| 400 BAD REQUEST | La solicitud no fue válida. Este código se devuelve cuando el servidor ha intentado procesar la solicitud, pero algún aspecto de la solicitud no es válido; por ejemplo, un recurso formateado de forma incorrecta o un intento de despliegue de un proyecto de sucesos no válido en el tiempo de ejecución de sucesos. La información acerca de la solicitud se proporciona en el cuerpo de la respuesta e incluye un código de error y un mensaje de error. |
| 401 UNAUTHORIZED | El servidor de aplicaciones devuelve este código de respuesta cuando está habilitada la seguridad y faltaba la información de autorización en la solicitud. |
| 403 FORBIDDEN | Indica que el cliente ha intentado acceder a un recurso al que no tiene acceso. Podría producirse si el usuario que accede al recurso remoto no tiene privilegios suficientes. |
| 404 NOT FOUND | Indica que el recurso de destino no existe. Esto podría deberse a que el URI no está bien formado o a que se ha suprimido el recurso. |
| 405 METHOD NOT ALLOWED | Se produce cuando el recurso de destino no admite el método HTTP solicitado; por ejemplo, el recurso de funciones solo permite operaciones GET. |
| 409 CONFLICT | Indica que se ha detectado un cambio conflictivo durante un intento de modificación de un recurso. El cuerpo de la respuesta contiene más información. |
| 415 UNSUPPORTED MEDIA TYPE | El recurso de destino no admite el formato de datos del cuerpo de la solicitud especificado en la cabecera de Content-Type. |
| 500 INTERNAL SERVER ERROR | Se ha producido un error interno en el servidor. Esto podría indicar un problema con la solicitud o un problema en el código del lado del servidor. |

Debemos tener siempre en cuenta que es nuestra obligación desarrollar nuestras API de la misma forma como esperamos que otros desarrollen las API que nosotros vamos a utilizar.

Para un mayor detalle, se debe remitir a:

|  |
| --- |
| <https://en.wikipedia.org/wiki/List_of_HTTP_status_codes>  <https://www.restapitutorial.com/httpstatuscodes.html> |

# MANEJO DE VERSIONES DEL API.

El manejo de versiones se puede implementar a través de dos grupos de implementaciones:

* Por URL
* Por Header o Media Types.

El esquema de versionado que se adoptará para las API es el versionado por URL. Aunque existen diferentes formas de utilizar este esquema de versionado, el siguiente es el lineamiento a aplicar:

|  |  |
| --- | --- |
| .../mapas/version/2 | No utilizar |
| .../mapas/version/2/escuelas/version/3 | No utilizar |
| .../mapas/v2/escuelas | Utilizar |

# ESTRUCTURA DE LAS RESPUESTAS (RESPONSE) JSON

Es importante uniformizar la estructura de los JSON de respuesta (response), a fin de poder disminuir el esfuerzo en el entendimiento de los desarrolladores de back-end y front-end.

La selección inicial es una estructura basada en la distinción de las ***respuestas exitosas*** de las ***respuestas de error***. En esta línea lo primero que debemos poder determinar es, entre otras:

**Alternativa 1:** Utilizamos los códigos de estado HTTP conteniendo el estado acompañando al cuerpo (body) en el response.

**Alternativa 2:** Siempre retornamos el estado 200 y en el cuerpo (body) van el estado y los datos.

Las dos alternativas son válidas, sin embargo; por defecto vamos a adoptar la alternativa 1. En tal sentido, los lineamientos son los siguientes:

1. Los objetos requeridos únicos deben ser retornados en el nivel superior del JSON de respuesta.

|  |  |
| --- | --- |
| Para adoptar | Para no adoptar |
| Status: 200  Content-Type: application/json  {  “success”: true,  “code”: ”0000”,  “messages”: [],  “data”:  {  "employeeId": 123456,  "firstName": "Eliana",  "lastName": "Quezada",  "gender": "Female",  "hireDate": "2012-01-01",  "birthDate": "1990-01-01",  "office": "OTIC"  }  } | Status: 200  Content-Type: application/json  {  "employeeId":123456,  "firstName": "Eliana",  "lastName": "Quezada",  "gender": "Female",  "hireDate": "2012-01-01",  "birthDate": "1990-01-01",  "office": "OTIC"  } |

1. Los errores deben estar referidos a los estados HTTP y se deben manejar con etiquetas “code” (con valor siempre en mayúsculas), “message” (con valor indicando la causa del error) y “errors” (que es opcional, contiene una lista de errores”). Por ejemplo:

|  |
| --- |
| En el caso de consultas:  Status: 404  Content-Type: application/json  {  “success”: false  "code": "9000",  “error”: “Elemento no encontrado”  "messages": ["El empleado 1234456 no se encuentra registrado"]  } |
| En el caso de validaciones:  Status: 400  Content-Type: application/json  {  “success”: false,  "code": "9000",  "messages": ["La información suministrada no es válida."],  "validations": [  {  "message": "El apellido es requerido.",  "field": "lastName"  },  {  "message": "La edad del empleado debe ser mayor que 18 años",  "field": "dateOfBirth"  }  ]  } |
| Status: 401  Content-Type: application/json  {  “success”: false,  "code": "9000",  "messages": ["El acceso a la información de empleados no ha sido otorgado. "]  } |

1. Para respuestas que contienen listas como resultados de búsquedas también existen más de una alternativa con sus respectivas combinaciones:

**Alternativa 1:** Usar un estilo envolvente de respuesta que incluya un contador y los registros devueltos acompañados con el correspondiente código de estado HTTP.

**Alternativa 2:** Retornar la data, los códigos de estado y los contadores en la cabecera de la respuesta.

**Alternativa 3:** Retornar siempre código de estado 200 en cuerpo de la respuesta HTTP.

Por defecto, vamos a utilizar la alternativa 2.

|  |
| --- |
| Status: 200  Total-Count: 100  Content-Type: application/json  {  “success”: true,  “code”: ”0000”,  “messages”: [],  “data”:  [  {  "employeeId":123456,  "firstName": "Francisco",  "lastName": "Medina",  "gender": "Male",  "hireDate": "2012-01-01",  "birthDate": "1990-01-01",  "office": "OTIC"  },  {  "employeeId":123457,  "firstName": "Mariela",  "lastName": "Rojas",  "gender": "Female",  "hireDate": "2012-01-01",  "birthDate": "1990-01-01",  "office": "OTIC"  }  ]  } |
| Status: 200  Total-Count: 0  Content-Type: application/json  [] |

1. Para el manejo de las respuestas de nulos o vacíos.

En este caso, hay dos alterativas posibles;

**Alternativa 1:** No se retorna valores nulos o vacíos.

**Alternativa 2:** Sí se retorna valores nulos o vacíos.

Por defecto, vamos a adoptar la primera alternativa.

|  |  |  |
| --- | --- | --- |
| No adoptar, solo aplica para propósitos representacionales | Para adoptar: No aparecen los valores nulos, pero sí aparecen los valores vacíos. | Para adoptar: No aparecen ni nulos ni vacios. |
| {  “employeeId”: 12345,  “firstName”: “Mónica”,  “lastName”: “Rojas”,  **“mobile”: null,**  **“addresses”: []**  } | {  “employeeId”: 12345,  “firstName”: “ Mónica “,  “lastName”: “ Rojas “,  **“addresses”: []**  } | {  "employeeId": 12345,  "firstName": "Mónica",  "lastName": "Rojas"  } |

1. Para respuestas con Date y DateTime, se debe usar el formato ISO 8601[[2]](#footnote-2). Por ejemplo:

|  |
| --- |
| {  "createdBy": "123456",  "createdAt": "2012-01-01T18:25:43.511Z",  "createdBy": "123456",  "createdAt": "2012-01-01T18:25:43.511Z",  } |

# RESPECTO A LAS HERRAMIENTAS A UTILIZAR

# PARA CONSTRUIR API

**Net Core 3.0:** Versión del framework .Net de desarrollo de aplicaciones multipltaforma web, servicios y aplicaciones de consola. Net Core no es exclusivo de Windows y sus productos pueden ser desplegados en Dockey y en la Nube, pudiendo desarrollar aplicaciones en Linux.

**NodeJS:** Es un entorno JavaScript de lado de servidor que utiliza un modelo asíncrono y dirigido por eventos.

**Laravel:** Es un framework de código abierto para desarrollar aplicaciones y servicios web con PHP 5 y PHP 7. Su filosofía es desarrollar código PHP de forma simple evitando el "código espagueti". Fue creado en 2011 y tiene una gran influencia de frameworks como Ruby on Rails, Sinatra y ASP.NET MVC.

**Ballerina:** Es un lenguaje de programación de código abierto y una plataforma para que la programación de aplicaciones en la nube.

# PARA PROBAR Y DOCUMENTAR APIS

**La especificación Open API:** Conocido también como especificación Swagger. Define una interfaz estándar independiente del lenguaje para las API REST que permite que tanto humanos como computadoras descubran y comprendan las capacidades del servicio sin tener que acceder al código fuente, la documentación o mediante la inspección del tráfico de la red.

**Postman:** Es una plataforma colaborativa para el desarrollo de API. Postman permite diseñar, construir y testear API.

**JMeter:** Software de Apache de código abierto que permite simular cargas desde varios escenarios y generar información del rendimiento de las API.

# SEGURIDAD DE LAS API

La adopción de mecanismos de seguridad va a proteger las API que sean desarrolladas y posteriormente implementadas. Existe gran cantidad de información sensible y dependiendo de su tipo y criticidad, se deben implementar diferentes mecanismos de seguridad.

# HTTPS

Los servicios API REST en ambientes productivos deben ser expuestos solo a través de endpoints HTTPS. Con este mecanismo se protegen las credenciales de autenticación en tránsito, por ejemplo contraseñas, identificadores de recursos o JSON Web Tokens. También permite que los clientes autentiquen los servicios y se garantice la integridad de la información transmitida.

En los ambientes de desarrollo de MINEDU es posible implementar HTTPS a fin de poder realizar las pruebas correspondientes a la seguridad de las API.

# OFUSCAR EL IDENTIFICADOR DEL RECURSO

Para la RAE ofuscar significa “oscurecer, turbar la vista”.

Uno de los aspectos a tener en cuenta en la construcción de API es la de mantener segura la información que se expone. En muchos de los casos, las Id de los recursos, se corresponde con las claves primarias de los registros en tablas de bases de datos. En este escenario, un atacante podría conocer cuál es el siguiente número de identificador de un recurso y recuperar u obtener toda la información asociada a dichos recursos.

El ofuscador puede implementarse como un servicio, como una función en la base de datos o una clase o como un helper.

# JWT

Es un estándar [RFC 7519](https://tools.ietf.org/html/rfc7519) para transmitir información con la identidad y “[claims](https://en.wikipedia.org/wiki/Claims-based_identity)” de un usuario de forma segura entre un cliente/servidor. Dicha información puede ser verificada y confiable porque está firmada digitalmente.

Un claim es una declaración que un sujeto, como una persona u organización, hace sobre sí mismo u otro sujeto. Por ejemplo, la declaración puede ser sobre un nombre, grupo, preferencia de compra, origen étnico, privilegio, asociación o capacidad.

En concreto un JWT es una cadena de texto que tiene 3 partes codificadas en Base64, separadas por un punto (cabecera.cuerpo.firma) que generamos y es entregada a los clientes de una API.

La secuencia de creación y uso de JWT entre API es la siguiente.



# LÍMITES DE TRÁFICO (CUOTAS)

El acceso irrestricto no es una buena práctica para la gestión de APIs en los ambientes productivos, pues se contrapone a su escalabilidad y seguridad. Los límites de procesamiento a través de API se mide en Transacciones por segundo (TPS). Si se prevé que una API recibirá muchas solicitudes en algún momento de su uso, es posible aplicar un patrón de arquitectura denominado “Throttling” (estrangulamiento).

El patrón Throttling permite controlar la cantidad de recursos (servicios) disponibles hasta cierto nivel y después limitar (o estrangular) servicios no priorizados para prevenir el incumplimiento de los SLA o evitar el colapso del sistema. Presentamos cuatro estrategias a seguir:

1. **Limitar el número de peticiones.**

Consiste en limitar el número de peticiones que un cliente puede realizar al servicio en un tiempo determinado, lo que obliga al sistema a medir el número de solicitudes por cliente, para finalmente denegar el servicio cuando el umbral ha sido alcanzado.

1. **Priorizar servicios.**

Consiste en limitar o denegar los servicios no esenciales hasta que el sistema alcance su punto de normalización. Para implementar esta estrategia es importante poder identificar cuáles son los procesos prioritarios y cuáles son prescindibles, de tal forma que podamos apagarlos o degradarlos de forma rápida, evitando que el sistema llegue a un estado de saturación.

1. **Distribución priorizada de solicitudes**

Esta estrategia se utiliza el patrón multiinquilino (multitenancy). A cada inquilino se le diferencia de otros según su importancia y se priorizan los mensajes mediante colas con diferentes prioridades. Esta estrategia puede ser usada en aplicaciones en la nube donde podemos distinguir inquilinos VIP de los no VIP.

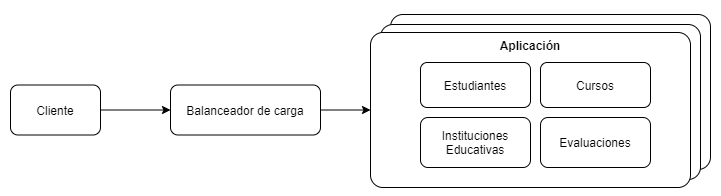
Las siguientes son las consideraciones al implementar este patrón:

* La solución debe haber sido diseñada para soportar este patrón, pues resulta importante establecer la priorización de los servicios.
* La solución debe poder diferenciar entre un error de la aplicación y uno de la aplicación de este patrón, de esta forma el cliente podrá saber el motivo por el cual su solicitud está siendo rechazada.
* Este patrón se puede utilizar como una estrategia temporal mientras el escalamiento horizontal tiene lugar.

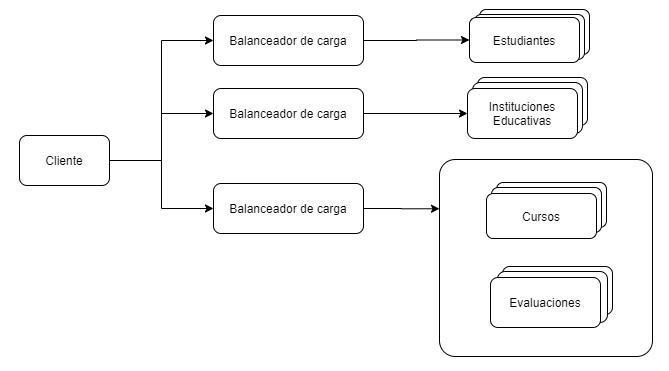
Una de las herramientas a utilizar para implementar este patrón se denomina API Gateway.

# PATRON API GATEWAY

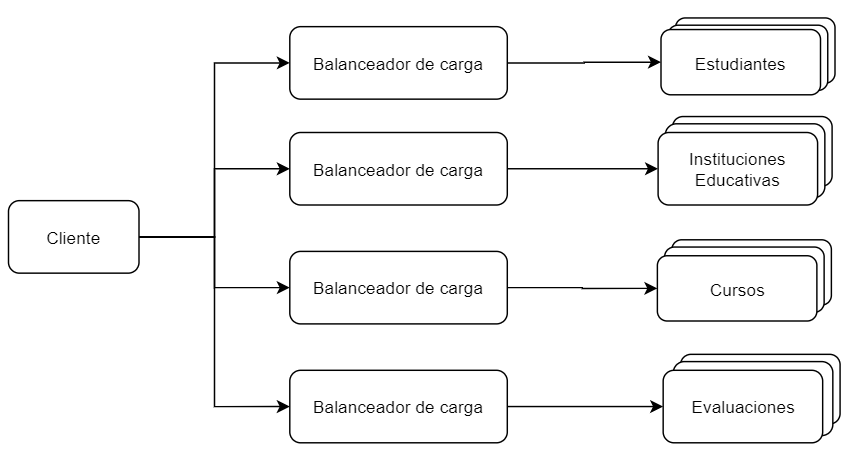
Pasar de una arquitectura monolítica de sistemas de información hacia una basada en microservicios, es un objetivo que se ha convertido en una necesidad importante a adoptar para disponer de sistemas de información escalables. Por ejemplo, analicemos el sistema cuya arquitectura monolítica se muestra a continuación:



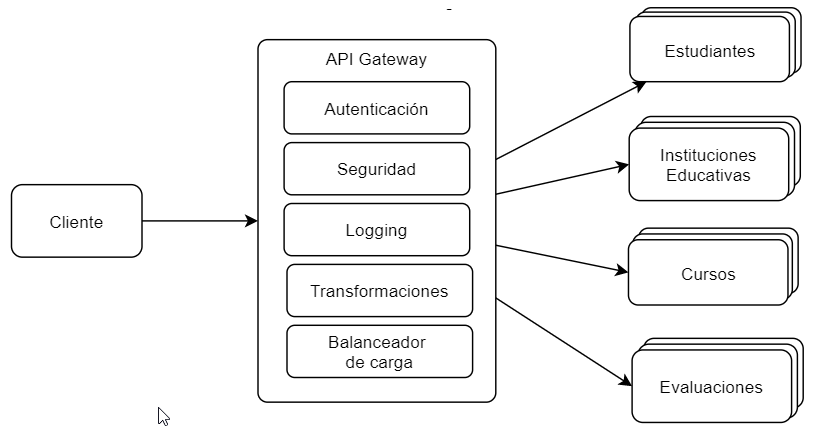
Es posible aplicar una estrategia que permita ir estrangulando la aplicación con el fin de generar microservicios progresivamente.



Al finalizar el estrangulamiento, o al construir microservicio, es una alternativa que para cada uno de estos microservicios se deba desarrollar las funcionalidades de autenticación, seguridad, logging (tracking), gestión de variables de entorno, entre otros.



Es aquí donde se manifiesta el patrón API Gateway, encargándose de factorizar las funcionalidades mencionadas en un componente intermedio entre el cliente y las APIs.

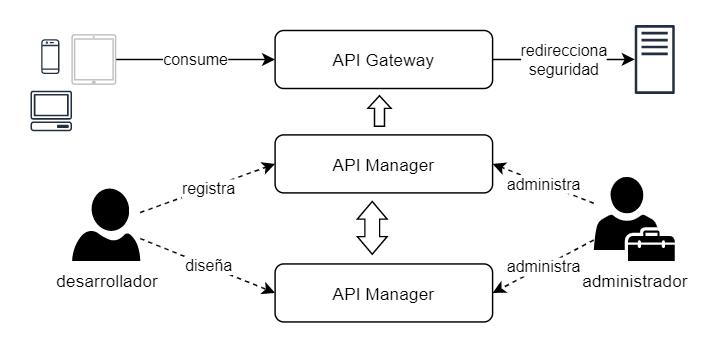


De esta forma, ya que la arquitectura exige que las funcionalidades se encuentren segregadas, se necesita presentar las API de una forma homogénea a los clientes y al mismo tiempo se evita exponer cómo está construida la estructura interna de servicios.

Como se aprecia en la gráfica, el componente API Gateway se debe encarga de las siguientes funciones:

* Asegurar las APIs para que cada una de ellas no deba ocuparse de ello de forma independiente.
* Obtener métricas de uso de las APIs.
* Obtener los logs (loggings) en un repositorio común.
* Transformar peticiones y respuestas.
* Almacenar variables de entorno.

EL API Gateway es un componente de un sistema de gestión de API:



# INSTALACIÓN DE API GATEWAY KONG

La instalación de Kong en entornos de desarrollo se puede realizar a través del uso de contenedores. Kong puede utilizarse con Cassandra o PostgreSQL como soporte de persistencia de datos, el uso de Cassandra está recomendado para entornos de elevado I/O de ambientes productivos.

Punto aparte es la instalación de la herramienta de instalación Konga, que puede instalarse también de forma contenerizada.

La administración de la instalación de Kong y Konga estará a cargo de: dos especialistas

1. Un especialista en administración de API Gateway de ambientes de desarrollo.
2. Uno o más especialistas en gobierno los API Gateway de los ambientes productivos.

# CONFIGURACIÓN Y ACTIVACIÓN DE PLUGINS EN EL API GATEWAY

La configuración y la activación de plugins (adicionales) en el API Gateway estará supeditada a las decisiones de arquitectura que deben quedar plasmadas en el documento de arquitectura del proyecto en el capítulo: Configuración de API Gateway y adicionales.

Los plugins (adicionales) que se pueden considerar en toda arquitectura de sistemas son:

|  |  |  |
| --- | --- | --- |
| **Tipo** | **Adicional (plugins)** | **Lineamiento** |
| Autenticación | JWT | Verifica peticiones conteniendo Jason Web Tokens firmados |
| OAuth2 | Adiciona una capa con los flujos de OAuth 2.0. Ya que requiere el plugin de SSL, solo funciona con la versión empresarial de Kong. |
| Seguridad | Restricciones de IP | Maneja listas blancas y listas negras de direcciones IP. |
| Detecciones de bot | Porteje las API de los bots más comunes. |
| ACL | Restringe el acceso a las API a través de las listas de control de acceso. |
| Control de tráfico | Ratios límite | Limita el número de peticiones HTTP por unidad de tiempo. |
| Caché en proxy | Memoria intermedia para mejorar performance. |
| Analítica y monitoreo | Datadog | Instala agente Datadog |
| Logging | Http Log | Registra el tracking de las peticiones http configuradas. |

# LINEAMIENTOS DE NOMENCLATURA DE ELEMENTOS DEL API GATEWAY

|  |  |  |
| --- | --- | --- |
| **Elemento** | **Lineamiento** | **Ejemplo** |
| Clientes | Antecede al nombre de la entidad el prefijo “client”.  Todas las letras en minúsculas. | “client-aprendo-en-casa” |
| Servicios | Antecede al nombre de la entidad el prefijo “service”.  Todas las letras en minúsculas. | “service-generador-boletas” |
| Rutas | Antecede al nombre de la entidad el prefijo “route”.  Todas las letras en minúsculas. | “route-boletas” |
| Upstreams | Antecede al nombre de la entidad el prefijo “upstream”.  Todas las letras en minúsculas. | “upstream-siagie" |

# BALANCEO DE CARGA Y MONITOREO DE SERVICIOS

El desarrollo orientado a microservicios representa desafíos para los desarrolladores y los administradores de la plataforma de despliegue, ya que se tienen más servicios que monitorear en disponibilidad y rendimiento. También, los desarrolladores deben tener en cuenta que el balanceo de carga es una estrategia a utilizar cuando los sistemas se desplieguen en los ambientes productivos.

El balanceo de carga consiste en la práctica de distribuir las peticiones del cliente a través de múltiples instancias de la aplicación, siempre que éstas se encuentren operativas y evitando de esta forma que una simple instancia se vea sobrecargada.

El balanceo de carga también permite escalar aplicaciones y actualizarlas sin necesidad de tener tiempos de inactividad total de la aplicación. Esto se logra habilitando un despliegue continuado en una pequeña cantidad de instancias antes de implementarlo totalmente.

Asimismo, en la línea de conservar la salud de los servicios desplegados, es necesaria la implementación de estrategias de monitoreo de la salud de los servicios y aislamiento de fallas.

A continuación se describen las estrategias a considerar por los desarrolladores y operadores para el despliegue de servicios.

|  |  |  |
| --- | --- | --- |
| **Estrategia** | **Tipo** | **Observación** |
| Balance de carga | Por DNS, consiste en la configuración un dominio en el servidor DNS con direcciones IP de múltiples hosts, así las solicitudes de los clientes se distribuyen | Se debe usar cuando los hosts sean de alto rendimiento o en la red interna de MINEDU. |
| Por round robin, consiste en el envío de solicitudes a un servidor centralizado, que a su vez lo distribuye a los hosts en forma cíclica. | No se debe usar cuando se necesite sesiones consistentes. |
| Por anillo, se mantienen sesiones consistentes entre clientes y hosts a través de un hash y cuando un host es removido, las sesiones se transfieren a un nuevo host. | El cliente también puede reiniciar la sesión cuando un host es removido. |
| Monitoreo de salud | Por monitoreo activo, el balanceador envía periódicamente una petición de chequeo a cada host. El host puede generar un timeout o un código de error 500. | Su uso requiere elevado nivel de configuración (uno por cada comportamiento) y por lo tanto se introduce tráfico extra. |
| Por monitoreo pasivo, el balanceador monitorea los errores mientras van ocurriendo, si éstos pasan un límite entonces se marca el host como no saludable. | La ventaja es que no requiere mayor configuración. La desventaja es la real afectación al flujo del trabajo del usuario. |
| Corto circuitos | Cuando un host no está en buen estado, lo mejor es hacer que el tráfico fluya a otro host en buen estado. | Su uso aplica cuando un host viene recibiendo demasiadas solicitudes o deba reiniciarse |

De la aplicación correcta de estas estrategias combinadas, los desarrolladores podrán concentrarse cada vez más en implementar las funcionalidades propias del negocio.

# ANEXOS

# ANEXO N° 01: ESTILOS PARA NOMBRAR RECURSOS.

En el marco del manejo de recursos existen tres principalmente estilos (o convenciones) para nombrarlos: CamelCase, snake\_case y spinal-case. En todos los casos, el objetivo es nombrar recursos que para su mejor entendimiento para los analistas y desarrolladores, más aún si en el lenguaje natural se requiere más de una palabra para dicho nombramiento.

1. CamelCase

En este estilo las letras mayúsculas aparecen a lo largo de una palabra, siendo semejantes a las jorobas de un camello.

Existen dos tipos de CamelCase:

* UpperCamelCase, cuando la primera letra de cada una de las palabras es mayúscula. Ejemplo: MinisterioDeEducacion. También se le reconoce como PascalCase.
* lowerCamelCase, igual que la anterior con la excepción de que la primera letra es minúscula. Ejemplo: entregarDocumento.

Esta convención es muy usada, se usa en #hashTags, en nombres de empresa como LaLiga y para variables en muchos lenguajes de programación PHP, Java, C#.

1. **snake\_case**

Con este estilo las palabras son separadas por barra baja (underscore) en vez de espacios y con la primera letra de cada palabra en minúscula. Por ejemplo ministerio\_de\_educacion.

Este tipo de convención se utiliza en nombres de variables y funciones de lenguajes de programación basados en C. También lenguajes como Ruby y Python lo han adoptado. Igual que el CamelCase existen variedades, por ejemplo todas las letras en mayúsculas de denomina SCREAMING\_SNAKE\_CASE utilizado para definir constantes.

1. **spinal-case**

Es una variante del snake-case que utiliza guiones "-" para separar las palabras. Es la forma tradicional de nombrar carpetas y archivos en sistemas UNIX y Linux. Ejemplos: usuario-actual, correo-electronico, etc. También es conocido como kebab-case.

# ANEXO N° 02. REFERENCIAS

Algunas páginas de referencia que han servido para elaborar el documento de lineamiento:

1. Contiene algunas recomendaciones importantes:

<https://platzi.com/blog/como-crear-apis/>

<https://dzone.com/articles/7-tips-for-building-an-api?fromrel=true>

https://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api

1. Contiene conceptos que han sido incorporados al documento.

<https://anexsoft.com/que-es-rest>

<https://tools.ietf.org/html/rfc3986>

1. Contiene conceptos importantes para la seguridad de API REST.

<https://restfulapi.net/security-essentials/>

<https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html>

<https://www.redhat.com/es/topics/security/api-security>

<https://medium.com/@icanhazstring/rest-api-security-through-obscurity-d7c2ad21f810>

1. Contiene ejemplos de aplicaciones de métodos HTTP

<https://blog.michelletorres.mx/peticiones-http-get-post-put-delete-etc/>

<http://restcookbook.com/>

1. Contiene ejemplos de uso de parámetros

<https://idratherbewriting.com/learnapidoc/docapis_doc_parameters.html>

1. https://www.ibm.com/support/knowledgecenter/es/SSQP76\_8.5.1/com.ibm.odm.dserver.events.ref/topics/ref\_dse\_restapi\_responsecodes\_errormsgs.html [↑](#footnote-ref-1)
2. https://www.w3.org/TR/NOTE-datetime [↑](#footnote-ref-2)