



Tema 3: REST

REST

ARQ inicial → se basaba en principios sólidos
Protocolos → Asumían una conexión directa con el cliente (correo, proxy, ...)

Problemas de la web
(HTML)

Conciencia emergente sobre web

Repositorios web y propiedades

- Barra de entrada (hypermedia)
- Sistema de distribución de hypermedia
- Múltiples límites organizacionales

ARQ de REST style



RE presentational State Transfer

- garantizado en 1.1 a 4.1 HTTP
- Interfaz del protocolo → superior WSDL
- Basado de Arquitectura → NO es un entorno (no hay APC)
- NO hay "hard rules"
- sigue el paradigma A - E

Caracter. Principales

- sin estado
- basado en capas
- Interfaz uniforme y desacoplada
- representaciones como autor
- cacheable

Orientado al recurso

Propiedades	heterogeneous	escalable	evolutividad
	visibilidad	confiable	eficiencia

Beneficio de ARQ: → una serie de restricciones que limitan y moldean la forma de ARQ y las relaciones entre los elem de cualquier ARQ y sigue a continuación

Perspectivas en el proceso de diseño de una ARQ

Emplea con viés

- construir ARQ con elem. conocidos
- Nivel a. vs. en las necesidades del sistema
- Foto en la cuestión y visión de límites

Emplea con las necesidades de un ser humano consciente

- Simplifica, sin restricciones de forma, incrementa identif. a través de los restrictores
- se enfoca en las restricciones y la comprensión del contexto del sistema

EJEMPLO:

definimos una ARQ sin restricciones

Style → Cliente-Servidor → Porabilidad en simplificación

→ Stateless → simplicidad
→ mejoría → estabilidad → confiabilidad → eficiencia

→ Caching → estabilidad → eficiencia → confiabilidad

→ Uniform Interface → evolutividad → implementación interoperable → eficiencia

REST UI

→ todos los recursos imp. son identif. por un único mecanismo de identificador de recursos (uniforme) simple visible multivisible (común, sin estado)

→ Los métodos de acceso (acciones) significan lo → para TODOS los recursos (universal)

→ Sist. por capas → almacenable en cache y cacheable

→ Los recursos se manipulan a través del intercambio de representaciones

Simple visible multivisible (común, sin estado) → almacenable en cache

→ Intercambiadores como → auto-descriptivos

→ Sist. por capas → almacenable en cache y cacheable

El protocolo: motor del estado de la ARQ

Sist. de texto entrelazado dinámicamente, donde los docum. pueden contener enlaces a otros docum.

→ Una hipertexto, enlaza cada (o contiene) una representación actual del estado del recurso identificado

→ El recurso permanece oculto dentro de la hipertexto

→ Algunas representaciones (tienen enlaces) → tienen estados / futuros de la interacción → integrando dinamicidad sobre como hacer la transición a otros estados cuando se recorre. Una transición

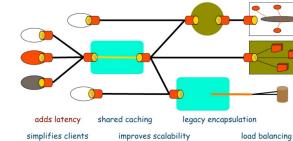
→ todo estado identificable → representa → estado actual de la aplicación

→ todo estado de la aplicación → mantiene → tiene

→ todo estado manejado → mantiene → tiene → de origen

Style → Layered System

Apply info hiding: layered system constraints



Interfaz Uniforme

la definición de los recursos consiste de

- URI
- formato → de datos (JSON, XML)
- métodos (GET, POST, PUT, DELETE)

añadir crear actualizar eliminar un recurso

- los recursos se identifican de forma única por un URI
- los servicios → revisión, leer, crear, eliminar, modificar.
- hypermedia as the engine of application state

HATEOAS

Verb	Path	Action	Used for
GET	/photos	index	display a list of all photos' information
GET	/photos/{id}	show	display specific photo
GET	/photos/new	new	return an HTML form for creating a new photo
GET	/photos/{id}/edit	edit	return an HTML form for editing a photo
POST	/photos	create	create a new photo
PUT	/photos/{id}	update	update a specific photo
DELETE	/photos/{id}	destroy	delete a specific photo

Lógicos

200 : OK
204 : No Content
400 : Bad Request
401 : Unauthorized
403 : Forbidden
404 : Not Found
405 : Method not allowed

Restricciones

Arquitectónicas

cliente-servidor	sin Estado	cacheable
sist. en capas	code on demand	uniform interface
	opcional	

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Date: Mon, 01 Dec 2013 04:49:41 GMT
Server: Apache/2.2.22 (Ubuntu)
Vary: Accept-Encoding
Content-Length: 113
{
    "id": "1",
    "name": "Ariane 5 ECA Launch Vehicle"
}
```

Este JSON filtra los datos que están siendo devueltos en la respuesta. Los datos que no cumplen con la especificación de OpenAPI se ignoran.

```
application/json
{
    "id": "1",
    "name": "Ariane 5 ECA Launch Vehicle"
}
```

Orientado al Recurso

con HTTP → el recurso se representa → body del →

→ tipo de formato → se establece mediante el tipo de contenido

→ B. puede negociar con Accept

→ Content-Type → text/plain, text/html, application/json, application/xml

→ Contrato de servicio WSDL

equivalente a WSDL o SOAP

uso en ocasiones, re-dimensionado extensivo

utiliza generar código de cliente y de servidor

mejor integraciones con herramientas y frameworks

mejor extensibilidad mejor con Open API (swagger)

se puede usar

versión

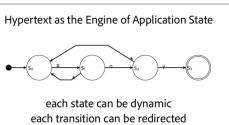
Tema 3: REST

REST

sin estado

- El **SG** no debe mantener el estado del Cliente
 - simplifica el balanceo de carga
 - permite el uso de cache
- El servidoralmacena únicamente el estado de sus recursos
- los servicios del lado del servidor NO están permitidos
 - El cliente SÍ puede gestionar el uso, actualizaciones, etc.
 - cualquier petición respuesta, donde el cliente renderiza la información → Poder procesar?

HATEOAS

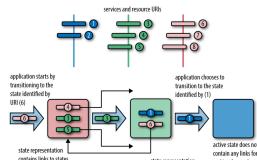


REST → Aplicación Web se puede ver como una gran máquina virtual de estados

Recursos → nodos
Entidades → transiciones de estado
siguiente estado → resultado de seguir el enlace y el resultado al que accedes

Hypermedia As The Engine Of Application State

- extensión del concepto de hipertexto
- Recursos → tienen enlaces a otra vez a otros recursos
- la autorización sale tras cada respuesta → posible continuaciones
- Mantiene el estado de la aplicación (funciona de estados dinámicos)

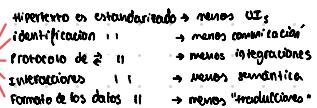


seguridad en REST

- Mecanismos disponibles en los capas de transporte y HTTP
 - [SSL/TLS]
- Mecanismos autenticación: **HTTP Basic** (más usado, más usado)
- HTTP Digest (más usado, más usado)
- SSL → permite emplear cifrado
- limitación frente a las soluciones de seguridad basadas en mensaje interoperables por WS-Security

Beneficios API basada en REST

- Reutilización
- acoplamiento → permite evolución
- elimina condiciones de fallo forzadas
 - (fallos del SG → no contienen el estado del cliente)
 - los estados comprobados → se pueden recuperar como un recurso
- Escala sin límites



- Utiliza tecnologías y plataformas bien documentadas y medidas (HTTP) → hereda SG de HTTP
- Facilidad de uso → dada una URI, todo el SG sabe cómo acceder a ella
- Nos autoriza a usar otro protocolo
- Promueve utilizar → al formato de SG
- Navegar entre → Usar lo ya está establecido para dar un paso más; es la extensión lógica de la web

NOTA:

SOAP (Simple Object Access Protocol) es un protocolo de mensajería basado en XML para intercambiar información estructurada entre aplicaciones a través de redes, como HTTP. Se utiliza principalmente en servicios web para comunicar aplicaciones distribuidas.

servicios RESTful

modelo de estructura → clasificación de servicios web

Nivel 0

- más básico, 1 sola URI
- utilizan un único método HTTP (núcleo del POST)
- Ej: servicios SOAP
- Ej: uso: HTTP, pero sin aprovechar las <operaciones>



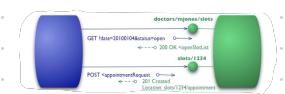
Nivel 1

- utilizamos varias URIs
- 1 solo método HTTP (GET o POST fundamentalmente)
- se expusen a varios recursos
- muchos de los servicios REST actuales se quedan a este nivel



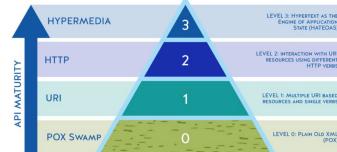
Nivel 2

- varios URIs → soportan N métodos HTTP
- servicios q pueden manejar op. CRUD
- se utilizan métodos de estado HTTP
- ya no usamos HTTP solo como transporte



Nivel 3

- se añade la noción de HATEOAS
- las representaciones incluyen URIs
- la transición de un recurso a otro genera el estado de la aplicación
- otros recursos



Buenas Prácticas

- Nombrado: sustantivos, no verbos
<http://example.com/api/getBooks> → <http://example.com/api/books>
- Múltiples URLs por recurso
<http://example.com/api/books> o <http://example.com/api/books/10>
[GET http://example.com/api/books/10](http://example.com/api/books/10) → Obtener todos los libros
[GET http://example.com/api/books/10](http://example.com/api/books/10) → Obtener el libro con ID 10
[POST http://example.com/api/books/10](http://example.com/api/books/10) → Crear un nuevo libro
[PUT http://example.com/api/books/10](http://example.com/api/books/10) → Actualizar libro con ID 10
[DELETE http://example.com/api/books/10](http://example.com/api/books/10) → Borrar libro con ID 10
- Asociaciones: La API debe ser intuitiva a la hora de definir las asociaciones
<http://example.com/api/user/123/books/10> → Obtener el libro con ID 10 para el usuario con ID 23
- No sobrecargar al cliente, si el contenido que se va a servir es muy grande usar paginación (no devolver todos los resultados)
- Recordar que las peticiones GET tienen un tamaño máximo
- Si el servicio tiene muchos argumentos o con valores de tamaño variable se debe utilizar POST o PUT
- Usar correctamente los métodos y códigos de error HTTP
- Para buscar, ordenar, filtrar, pagination, etc. No se crearán nuevos recursos, sino que se añadirán al existente del GET
[GET http://example.com/api/books?rank_asc](http://example.com/api/books?rank_asc)
[GET http://example.com/api/books?category_id=10](http://example.com/api/books?category_id=10)
- En una API siempre se debe mantener la compatibilidad hacia atrás, si no se hace, se debe cambiar el nombre
- Usar un mapa de URLs lo más intuitivo posible y no cambiarlo
- Pensar muy bien cómo va a realizar el consumidor todas las operaciones necesarias
- Aplicar HATEOAS

Tema 3: REST

Estandarización

Lenguajes de Descripción   

Conclusión de REST

- no es una idea fija, sino un principio de diseño
- Requerimientos (comunicantes) → codificar en diseño
- estilo o lenguaje de diseño
- El cambio es inevitable, usa el principio de diseño
 - identifica las novedades API y guíate
 - selecc. entres. de API q sean apropiadas
 - probadas
 - restringe el comportamiento para introducir mejoras
 - compensa los desventajas del diseño.

--- T3A ---

YAML : YAML Ain't Markup Language

- Data serialization language
 - ↳ representación de objeto o estructura de datos
 - ↳ formato para almacenar y enviar
- Machine & human readable
- usado principalmente como lenguaje de config.

• comentarios → # (no documentados) multilínea
Puede haber varios elem. vacíos
• formato → convenio del documento → ...
final del documento → ...
Surgiría → más + espacio (creando todo el nivel)

• tipos de datos de los nodos
Escalares (conmutativos) 
mapping → pares clave-valor
sequencia → lista de nodos

mapping

- pares clave-valor separados por ":" y un espacio
- cada par en una línea
- si no se indica un valor → se considera null
- las claves NO se pueden repetir
- las claves → cualquier tipo de dato

Ejemplo:
street: El Playa de San Juan
zip code: 12345
city: North Pole

Secuencia

- lista de nodos
 - cada uno empieza por coma y espacio (cuando, como longitud)
 - cada uno en su propia línea
- Ejemplo:
- ```

 - Orin Bum 444
 - Henry Kensington
 - - 2
 - - 3
 - - 4
 - - 5
 - - 6

```
- se pueden anidar

### Alias y Anchor

- permite reutilizar la info
- anchor: #nombreAlias
- alias: + nombreAlias

Ejemplo:  
Shipping address: Address  
Street: Santa Claus Lane  
ZIP: 12345  
City: North Pole  
Billing address: Address

• [ ] (literal block scalar)  
Repite linea a los siguientes  
"líneas" con la = interacción  
Todas las líneas son un bloque

Ejemplo:  
run: |  
 ./configure  
make  
make test|  
run: " ./configure&make &make test"

un salto de linea implica un salto de linea  
• [ ] (folded block scalar)  
Repite linea a los siguientes  
"líneas" con la = interacción  
Todas las líneas son un bloque  
Un salto de linea → un espacio  
2 saltos de linea → 2 saltos de linea  
cada salto de linea → otro salto de linea

### String

- plain 
- sin comillas
- no usar caracteres de escape
- no se permiten ciertas combinaciones de caracteres

:<FOO>  
<FOO>#

No se permiten ciertos caracteres al principio

### 1. Comillas simples ('')

- Todo lo que está entre comillas simples se interpreta literalmente.
- No se permite el uso de caracteres especiales como \n, \t, etc.
- Para poner una comilla simple dentro del texto, debes duplicarla: ''''

#### Ejemplo:

yml:  
text0: "Este es un texto literal: \n no hace salto de linea."  
text02: 'Comilla simple dentro del texto: isn\'t'

#### Resultado:

text0: "Este es un texto literal: \n no hace salto de linea."  
text02: 'Comilla simple dentro del texto: isn\'t'

### 2. Comillas dobles (")

- Permiten caracteres especiales como \n, \t, \v, etc.
- Son útiles siquieres incluir símbolos que de otro modo YAML interpretaba como : o #.

#### Ejemplo:

yml:  
text0: "Este si hace salto de linea:\ndependiendo linea"  
text02: "Comillas dobles permiten el uso de \'citas\'"

#### Resultado:

text0: Este si hace salto de linea:  
Segundo linea  
  
text02: "Comillas dobles permiten el uso de "citas"

## Flow Style

- se pueden usar {} para secuencias
- cada entrada → separada por comas
- caracteres extra → no válidos en plain strings: ; , ; ;

per: {  
- 5.3  
- 5.10  
- 6.0}

- se puede usar ; para los mapping (también con comas y ;)

= x:3  
y:4 → - b:3,y:4,z:2  
z:2 → - b:5,y:5,z:5 t  
= x:5  
y:5  
z:5

## ¿A que equivalen?

- data: → lista de 2 elementos
- step: → un string
- -- level 2 → [una sublista]

rolls: lista q contiene una lista.  
- - 2 [ [2,5] ]

sequence:  
- a: b  
c: d → [ a:b , c:d ]

data:  
- a: b } = - a:b } [ b:a,b:  
- c: d ] } c:d ]

## Es inválido!

Features:  
- name: lorem ipsum  
bullets: D  
test:  
- name: >  
  lorem  
  ipsum 2  
bullets:  
test:  
  acl: loopback  
  acl: admin

bloque de texto literal  
Este no tiene ningún texto en texto en formato de bloque literal.

Este es un ejemplo de texto en formato de bloque literal.

test:  
  acl:  
    loopback  
  admin

clave "ext" duplicada

# Tema 3: REST

## Open API

Estandar de descripción de APIs, (2010)

openapi: 3.1.0

- info:
  - title: A minimal OpenAPI document
  - version: 0.0.1
  - description: OpenAPI document example
- paths: {} # No endpoints defined

### Open API Object

openapi (string) versión de OAS, REQ.

info (Object) información general, REQ.

- title (string): nombre de la API, REQ

- version (string) versión de la API, REQ

- description (string)

paths (Paths Object):

- descripción de los endpoints

- información de parámetros y respuestas

podemos definir:

- servers: Array de Server Objects que indican la ruta

- webhooks: Información de los webhooks

- security: Información sobre los mecanismos de seguridad

- components: Definir schemas, parámetros y otros elementos para reutilizar en el documento

• 3 más

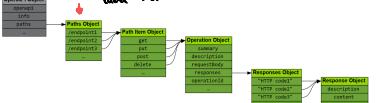
```
openapi: 3.1.0
info:
 title: API de Ejemplo
 description: API con definición de servidor y webhook.
 version: 1.0.0
```

servers:
 - url: https://api.ejemplo.com/v1
 description: Servidor de ejecución
 - url: https://testing.ejemplo.com/v1
 description: Servidor de pruebas

paths:
 /users:
 get:
 summary: Obtener lista de usuarios
 operationId: listarUsuarios
 responses:
 200:
 description: Lista de usuarios recuperada con éxito
 content:
 application/json:

## Path Object

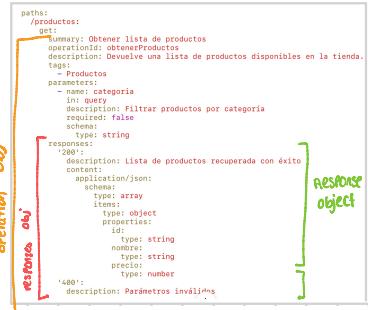
cada una empieza por "/"



- Puede contener un template entre {y}
- Tiene que estar definido después
- /books/{id}
- No puede haber dos templates path con la misma jerarquía, aunque tengan nombres diferentes
- /books/{id}/books/{ref}

## Path Item object

- Listado de los métodos soportados:
  - get
  - put
  - post
  - delete
  - Y más...
- Contiene un Operation Object
- summary
- description
- parameters (Parameter Object): lista de parámetros
- requestBody (Request Body Object)
- responses (Responses Object):
  - lista de posibles respuestas
  - Al menos tiene que haber una y debería ser la de éxito

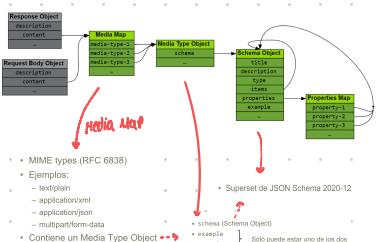


## Responses object (todos los resultados)

- Código de estado HTTP
- Entre comillas
- Se pueden usar placeholders: "1XX", "2XX", "3XX", "4XX" y "5XX"
- Tiene precedencia un método específico a un placeholder
- Contiene un Response Object

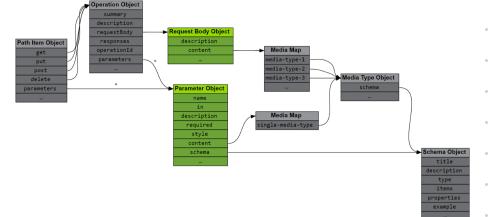
### Response Object

- description (string): REQ
- content (Media Map): lista de Media Types



- MIME types (RFC 6838)
  - Ejemplos:
    - text/plain
    - application/xml
    - application/json
    - multipart/form-data
- Superset of JSON Schema 2020-12
- Contiene un Media Type Object
- examples
- schema (Schema Object)
- example
- examples
- schema
- example

## Parámetros



## Parameter Object

- name: único en cada localización, REQ
- in: Localización del parámetro (path, query ...), REQ → Query, Path, Header o Cookie
- description
- required
- style (boolean):
  - false por defecto
  - REO para los path parameters y tiene que ser true
- explode (boolean)

| #Ejemplo 1     | #Ejemplo 2    |
|----------------|---------------|
| (...)          | (...)         |
| parameters:    | parameters:   |
| - name: id     | - name: id    |
| in: path       | in: query     |
| required: true | schema:       |
| schema:        | type: integer |
| type: integer  | minimum: 1    |
| maximum: 100   |               |

# Ejemplo 1  
(...)

parameters:  
- name: id

in: path  
 required: true

schema:  
 type: integer

minimum: 1

maximum: 100

# Ejemplo 2  
(...)

parameters:  
- name: id

in: query  
 required: true

schema:  
 type: integer

minimum: 1

maximum: 100

## Request Body Object

- content (Media Map): REQ.



Define el contenido que debe enviar en el cuerpo de una solicitud tipo

## Ventajas

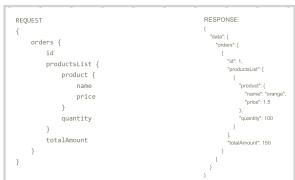
- Validación de descripción y linter
- Archivo de descripción es sintácticamente correcto
- Se cultiva a una versión específica de la especificación
- Sigue las pautas de formato del equipo
- Validación de datos
- Los datos que fluyen a través de la API son correctos
- Durante el desarrollo y uso no se desatrapagan
- Generación de info
- Generación de código
- Edificios gráficos
- Generación de documentos de descripción utilizando una interfaz gráfica
- Servidores IAC
- Análisis de seguridad

# Tema 3: REST

## GraphQL

Permite solicitar exactamente los recursos necesarios

Beneficios:   
 - comprueba el cacheado  
 - compuesto para servicios backend

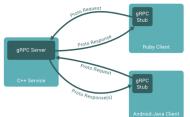


## gRPC

Google Remote Procedure Call

Open Source + HTTP/2 para transporte  
imposible crear un cliente para el navegador

Invierte métodos remotos como si estuvieran en local:



Por defecto → Protocol Buffer (Protocol Buffers) como formato de datos (→ JSON)

```

message Point {
 required int32 x = 1;
 required int32 y = 2;
 optional string label = 3;
}

```

## Comparativa

|                       | REST                                                                                          | RPC                                                                                                  | GraphQL                                                                                |
|-----------------------|-----------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------|
| ¿Qué es?              | Expone datos como recursos y utiliza métodos HTTP estándar para representar operaciones CRUD. | Expone métodos de API basados en acciones. Los clientes pasan al nombre del método y los argumentos. | Un lenguaje de consulta para APIs. Los clientes definen la estructura de la respuesta. |
| Ejemplos de servicios | Stripe, GitHub, Twitter, Google                                                               | Facebook, GitHub, Yelp                                                                               | Facebook, GitHub, Yelp                                                                 |
| Ejemplo de uso        | GET /users/<id>                                                                               | GET /users/getId</id>                                                                                | query (\$id: String!) { user(id: \$id) { name company createdat } }                    |
| Verbos HTTP usados    | GET, POST, PUT, PATCH, DELETE                                                                 | GET, POST                                                                                            | GET, POST                                                                              |

|         | REST                                                                                                                                                               | RPC                                                                                                                                                                                                                                                                      | GraphQL                                                                                                                                                                                                                       |
|---------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Pros    | <ul style="list-style-type: none"> <li>• Fácil de entender y leer.</li> <li>• Formato de entorno estable.</li> <li>• Utiliza ligeras - Alto rendimiento</li> </ul> | <ul style="list-style-type: none"> <li>• Fácil de entender - Cargas útiles ligeras</li> <li>• Utiliza características de HTTP - Alta versatilidad</li> <li>• Tamaño de carga más pequeño</li> <li>• Fuertemente tipado</li> <li>• Interoperabilidad integrada</li> </ul> | <ul style="list-style-type: none"> <li>• Poco código</li> <li>• Añadir recursos</li> <li>• Indicar las DB</li> <li>• Colocar la info más frecuente</li> <li>• Realizando operaciones más ágiles de forma asíncrona</li> </ul> |
| Contras | <ul style="list-style-type: none"> <li>• Cargas útiles grandes</li> <li>• Múltiples viajes HTTP</li> </ul>                                                         | <ul style="list-style-type: none"> <li>• Difícil de desplegar</li> <li>• Experiencia limitada</li> <li>• Puede llevar a explosión de funciones</li> </ul>                                                                                                                | <ul style="list-style-type: none"> <li>• Requerir una estrategia de consumo</li> <li>• Dificultad en el rendimiento en backend</li> <li>• Demasiado complicado para una sola simulación</li> </ul>                            |

## Webhooks

### Reverse API

- ① hace petición → Al Cliente
- HTTP Post
- Notificación de eventos → NO deberás incluir payload
- Enta q el Cliente tenga q hacer polling constante para ver si hay cambios:

solo 15% de los llamados de polling dejan nuevos datos

- ② necesitamos un endpoint para recibir las llamadas
- ③ Enviamos el endpoint al ☐
- ④ ☐ POST a nuevo endpoint
- ⑤ Se genera nuevo info en el ☐

Más: GitHub > repos > settings > Webhooks

- introduce la URL
- "use me select-indiv events" > "fuentes diferentes"
- Agrega un comentario
- comprueba q aparece en la interfaz

## Escalando la API

### Scaling Throughput

- nº de llamadas / segundo
- encontrar cuotas de banda
- Soluciones:
  - Análisis de código
  - Añadir recursos
  - Indicar las DB
  - Colocar la info más frecuente
  - Realizando operaciones más ágiles de forma asíncrona

### Evolving API Backend

- Introducir nuevos patrones de acceso de datos
- Añadir nuevos métodos a la API
  - Ajustar las cuotas de uso de los clientes
  - A veces se necesitan nuevas APIs
- Soporte para endpoints monovisión
- Añadir nuevos endpoints para filtrar los resultados
  - filtro ID
  - filtro fecha
  - filtro orden
  - en cursor de polling & no

### Paginación

- dividir grandes cantidades de datos → en partes más pequeñas
- evitando devolver demasiados datos
- offset-based pagination

Es una técnica común de paginación donde se indican:

- límite: la cantidad de elementos que puedes recibir.
- offset: el número de elementos que se deben saltar antes de empezar a devolver resultados.

Ejemplo:

Imagina que tienes una lista de 1000 productos. Para ver los productos 11 al 20:

```

GET /products?limit=10&offset=10

```

### Cursor-Based Pagination

Es una técnica de paginación que usa un identificador único (cursor) de un registro (índice) o de una marca de tiempo para comenzar a devolver los siguientes elementos.

El lugar de usar offset, el cliente dice:

"Dame los siguientes 5 elementos después de este cursor".

Ejemplo:

Supón que los productos tienen un campo id. Podrás hacer:

```

GET /products?limit=10&cursor=<cursor>

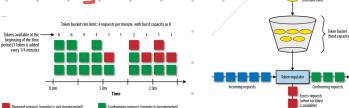
```

Esto devolverá los 10 productos siguientes al que tiene id = 10.

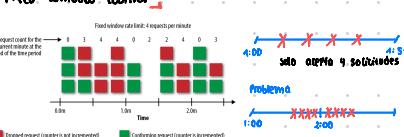
## Rate-Limiting APIs

- Limitar el nº de llamadas a nuestra API
- Evitamos ataques DOS y spam
- Podemos permitir límites globales o específicos por endpoints
- Tenemos q mgt el tráfico
- Nada q nos interese permitir más ríos de tráfico
- Cuantas mejores ejecuciones

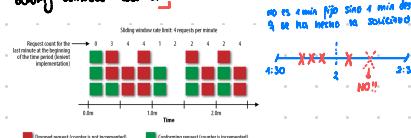
### TOKEN - Bucket



### Fixed-window Counter



### Sliding-window Counter



• Hay q indicar al cliente el nº de retenciones restantes

• Indicarlo en los cabeceras

- X-RateLimit-Limit
- X-RateLimit-Remaining
- X-RateLimit-Peace
- X-RateLimit-Reset
- Tiene q estar en la docum.

Código estado 429: too many requests

## 1. Code on Demand (Código bajo demanda)

Es una **restricción opcional** de REST que permite que los servidores proporcionen **funcionalidad adicional al cliente** en forma de **código ejecutable**, como por ejemplo:

- JavaScript
- Applets
- Scripts de configuración

**Ejemplo:** Un servidor web RESTful puede enviar un archivo JavaScript al navegador del cliente para que ejecute cierta lógica (como validación de formularios o renderización dinámica).

**Ventaja:** Mejora la flexibilidad del cliente, ya que puede recibir nuevas funcionalidades sin necesidad de actualizaciones.

**Desventaja:** Rompe un poco la "visibilidad" del sistema, ya que parte del comportamiento está encapsulado en el código que viaja del servidor al cliente.

## 2. Uniform Interface (Interfaz uniforme)

Es una **restricción fundamental** de REST. Establece una interfaz común para todas las interacciones entre cliente y servidor, independientemente de los recursos que se gestionen.

Sus principios clave son:

- **Identificación de recursos:** Cada recurso tiene una URL única (por ejemplo, /users/123).
- **Manipulación de recursos a través de representaciones:** Los recursos se modifican o consultan usando representaciones (como JSON o XML).
- **Mensajes autodescriptivos:** Cada mensaje HTTP debe contener toda la información necesaria (headers, métodos, etc.).
- **Hipermedios como el motor del estado de la aplicación (HATEOAS):** Las respuestas pueden contener enlaces que guían al cliente sobre qué acciones puede realizar después.

**Ejemplo:** Usar métodos HTTP de forma coherente:

- GET /products → Obtener productos
- POST /products → Crear un producto
- PUT /products/1 → Actualizar el producto con ID 1
- DELETE /products/1 → Eliminar el producto con ID 1

**Ventaja:** Simplifica y estandariza la comunicación, facilita la interoperabilidad entre distintos sistemas.

## 1. Fixed Window Counter

### Cómo funciona:

- Se divide el tiempo en intervalos fijos (ventanas), por ejemplo, cada **1 minuto**.
- Se mantiene un contador por cliente (o IP, o token...) que se reinicia al comenzar una nueva ventana.
- Si un cliente excede el límite dentro de la ventana, se bloquean más solicitudes hasta que empieza la siguiente ventana.

### Ejemplo:

- Límite: 100 solicitudes por minuto.
- Ventana actual: 12:00:00 a 12:00:59.
- Si haces 100 solicitudes a las 12:00:10, ya no puedes hacer más hasta las 12:01:00.

### Ventajas:

- Fácil de implementar.
- Muy eficiente en cuanto a memoria y CPU.

### Desventajas:

- No es justa: puedes hacer 100 solicitudes al final de una ventana y 100 justo al comienzo de la siguiente → 200 solicitudes casi seguidas.

## 2. Sliding Window Counter (o Sliding Log Window)

### Cómo funciona:

- Mantiene un registro de la **hora exacta de cada solicitud** o un contador ponderado que se mueve continuamente.
- Calcula el número de solicitudes en los últimos N segundos, sin importar la frontera fija de tiempo.

### Ejemplo:

- Límite: 100 solicitudes por minuto.
- Siempre se calcula cuántas solicitudes hiciste en los últimos 60 segundos desde el momento actual (por ejemplo, de 12:00:35 a 12:01:35).
- Si ya hiciste 100, tienes que esperar a que algunas "expiren" (hayan pasado más de 60 segundos) para hacer nuevas.

### Ventajas:

- Más justa: evita ráfagas intensas justo entre ventanas.
- Distribuye el tráfico de manera más uniforme.

### Desventajas:

- Más complejo de implementar (requiere almacenar timestamps o usar estructuras como colas o buckets con decaimiento).
- Mayor consumo de memoria y CPU.