

Tema 1

Introducción a los servicios web REST

Parte 4

Más allá de REST



Contenidos

1. RPC
2. GraphQL
3. APIs en tiempo real

1. RPC

Recordar: apps web "clásicas"

- Las URL representan **operaciones**
- Los datos de entrada se envían como **parámetros HTTP**
- Las peticiones solo pueden ser **GET** o **POST**

```
//obviamos la autenticación para simplificar  
http://uacloudmentira.ua.es/obtenerAlumno?dni=11222333F
```

Esta URL devolvería una **página HTML** con los datos del alumno

Las apps webs clásicas no son **APIs**, ya que devuelven *páginas completas HTML* (==interfaz), no *datos*... pero podemos "APIficarlo" **devolviendo JSON/XML/otros**. Opcionalmente podemos enviar JSON/XML/etc en vez de usar parámetros HTTP

Petición

```
http://uaclouddementira.ua.es/obtenerAlumno?dni=11222333F
```

Respuesta

```
{  
  "dni": "11222333F"  
  "nombre": "Ernesto"  
  "apellidos": "Rito Bravo"  
  ...  
}
```

APIs RPC

Los APIs de este estilo se denominan **RPC** (Remote Procedure Call) ya que la petición HTTP es muy similar conceptualmente a la llamada a un método o función convencional

```
Alumno obtenerAlumno(String dni) {  
    ...  
}  
  
obtenerAlumno("11222333F")
```

RPC sobre HTTP

```
http://uaclouddementira.ua.es/api/obtenerAlumno?dni=11222333F
```

Recordad el API de Flickr

No es que sea un mal API, simplemente no es REST ¡¡en realidad es **RPC**!! ya que la interfaz es una lista de **operaciones** y no de **recursos**.

contacts

- flickr.contacts.getList
- flickr.contacts.getListRecentlyUploaded
- flickr.contacts.getPublicList
- flickr.contacts.getTaggingSuggestions

favorites

- flickr.favorites.add
- flickr.favorites.getContext
- flickr.favorites.getList
- flickr.favorites.getPublicList
- flickr.favorites.remove

Recordad el API de Flickr

La única diferencia de sintaxis es que la operación en lugar de estar en la URL es un parámetro HTTP (**method**)

```
https://api.flickr.com/services/rest/?method=flickr.test.echo&name=value
```


RPC no es implícitamente inferior (ni superior) a REST

- RPC puede resultar más intuitivo cuando un API se exprese mejor como un conjunto de operaciones/procesos más que de recursos
- REST es el más popular (actualmente) en APIs públicos

Elecciones de diseño en un API RPC

- ¿Qué **protocolo** de transporte se usa?: HTTP, protocolos propios
- ¿Cómo se especifica la **operación**?: URL, parámetro HTTP, dato (JSON/XML/...) en el cuerpo de la petición
- ¿Cómo se pasan los **datos de entrada**?: parámetros HTTP, datos (JSON/XML,...) en el cuerpo de la petición

Algunos *frameworks* para desarrollar APIs RPC van un paso más allá y **hacen transparente la llamada remota**. En el código no hacemos peticiones HTTP sino en apariencia solo llamadas a **métodos/funciones**. Por ejemplo, [Apache Thrift](#)

```
var transport = new Thrift.TXHRTransport("http://localhost:8585/hello");  
var protocol  = new Thrift.TJSONProtocol(transport);  
var client = new HelloSvcClient(protocol);  
var msg = client.hello_func();
```

[HMTML completo \(ejemplo de cliente Thrift desde el navegador\)](#)

Estándares y protocolos abiertos en APIs RPC

- Estándares
 - **JSON-RPC**: JSON sobre HTTP
 - **SOAP**(*): XML sobre (generalmente) HTTP
- Abiertos
 - **gRPC**(*) (Google): datos binarios sobre HTTP/2
 - **Apache Thrift**(*): JSON,XML,texto,binario sobre HTTP

(*) Hace transparentes las llamadas remotas

JSON-RPC

Campos JSON estándar: `jsonrpc` (versión JSONRPC), `method` (operación), `id` (id del cliente), `params` (objeto con parámetros)

- Ejemplo: el *media center* [Kodi](#) lo usa para permitir el control remoto

```
# URL
http://<your-ip>:<your-port>/jsonrpc
```

```
// Cuerpo de la petición
// Pausar (o play si está pausado) el reproductor número 0
{
  "jsonrpc": "2.0",
  "method": "Player.PlayPause",
  "params": { "playerid": 0 },
  "id": 1
}
```

http://kodi.wiki/view/JSON-RPC_API/Examples

Otro ejemplo: el API Web de Slack

No sigue ningún estándar RPC, como [se dice en la documentación](#) simplemente es **"una colección de métodos HTTP al estilo RPC"**

- La URL siempre es de la forma `https://slack.com/api/METHOD_FAMILY.method`. Por ejemplo
 - `https://slack.com/api/channels.create`
 - `https://slack.com/api/conversations.invite`
 - `https://slack.com/api/conversations.archive`
- Usa únicamente GET y POST (por ejemplo borrar un mensaje se hace con POST)
- Según la operación se admiten los parámetros en formato `application/x-www-form-urlencoded` (o sea `nombre1=valor1&nombre2=valor2...`) o bien en JSON

Ejemplo: [documentación de cómo crear un canal](#)

2. Introducción básica a GraphQL

Un problema con los APIs REST

La **granularidad** de los recursos es **fija**. En la petición no podemos indicar que queremos solo **parte del recurso** o que queremos también **recursos relacionados**

```
http://miapiREST.com/blogs/1/posts/1
```

Queremos ver el post 1 del blog 1

El diseñador del API puede haber decidido que un post ya incluye los comentarios, o bien que no, pero es una *decisión fija*. Si a veces los necesitamos y otras no, tendremos un problema de eficiencia.

Ya vimos que ciertos APIs REST **extienden la sintaxis** para obtener solo algunos campos o para obtener recursos relacionados

```
https://graph.facebook.com/JustinBieber?fields=id,name,picture  
https://graph.facebook.com/me?fields=photos.limit(5),posts.limit(5)
```

Podríamos ver GraphQL como esta idea mejorada y ampliada

¿Qué es GraphQL?

Es un lenguaje para hacer consultas flexibles a **APIs orientados a recursos** en los que estos están relacionados entre sí formando un **grafo**

Esquema GraphQL

Además del lenguaje de consulta hay una sintaxis para definir el **esquema** de los recursos (**estructura** del grafo + **consultas** posibles)

```
type Author {  
  id: Int!  
  firstName: String  
  lastName: String  
  posts: [Post]  
}  
  
type Post {  
  id: Int!  
  title: String  
  author: Author  
  votes: Int  
}  
  
type Query {  
  posts: [Post]  
  author(id: ID!): Author  
}
```

Esta sintaxis es "abstracta". La sintaxis real dependerá del lenguaje que estemos usando para implementar el servidor GraphQL

Evolución de GraphQL

- Desarrollado en Facebook y usado internamente desde 2012. [Dado a conocer](#) en 2015
- La especificación es *open source*, aunque controlada por FB:
<https://github.com/facebook/graphql>
- Hay [multitud de implementaciones](#) de cliente y servidor en diferentes lenguajes. Las más conocidas son [Relay](#) (de FB) y [Apollo](#) (terceros)

Ejemplo sencillo

Tomado de <https://github.com/kadirahq/graphql-blog-schema>. Un API para gestionar un blog, sin BD, con datos en memoria para

- simplificar.
- Esquema
 - Recursos: Post, Category, Author, Comment
 - Relaciones: `Post->Category(1:1)`, `Post->Comment(1:N)`, `Post->Author(N:1)`, `Comment->Author(N:1)`

Cómo probar el ejemplo

1. Clonar el [repositorio git](#)
2. Instalar dependencias con `npm install`
3. Arrancar el servidor GraphQL con `npm run start`
4. Abrir un navegador e ir a `http://localhost:3000`. Aparecerá [GraphiQL](#), que es un editor interactivo y con autocompletado para lanzar consultas a APIs GraphQL

También podéis probarla *online* en <https://radiant-atoll-63982.herokuapp.com/>

Podéis probar estas consultas, u otras similares:

Podemos obtener los campos que queramos, del objeto sobre el que hacemos la *query* o de los relacionados

```
query {  
  latestPost {  
    title  
    author {  
      name  
    }  
  }  
}
```

Las *queries* pueden tener parámetros

```
query {  
  recentPosts(count:2) {  
    title  
    category  
  }  
}
```

Ejemplo de mutación

```
mutation {  
  createAuthor(_id:"Pepito", name:"Pepito Pérez", twitterHandle:"@pepito") {  
    # la mutación devuelve el autor creado, mostramos el nombre  
    # (aunque es un poco tontería porque ya lo sabíamos :))  
    name  
  }  
}
```


El *schema*

Define la interfaz con el API: las **queries** (consultas), las **mutaciones** (modificaciones) y la **estructura de los datos**.

```
const Schema = new GraphQLSchema({
  query: Query,
  mutation: Mutation
});

//aqui se definen las queries posibles y también la estructura
const Query = new GraphQLObjectType({
  name: 'BlogSchema',
  description: 'Root of the Blog Schema',
  fields: () => ({
    posts: {
      ...
    }
    latestPost: {
      ...
    }
    ...
  })
  ...
})
```

El *schema*

Define también la estructura de los datos

```
const Author = new GraphQLObjectType({
  name: 'Author',
  description: 'Represent the type of an author of a blog post or a comment',
  fields: () => ({
    _id: {type: GraphQLString},
    name: {type: GraphQLString},
    twitterHandle: {type: GraphQLString}
  })
});
```

La *magia*, o el "**enganche**" entre GraphQL y los datos reales (típicamente en una BD, pero aquí simplemente en variables en memoria) se hace en la función **resolve()**

```
latestPost: {  
  type: Post,  
  description: 'Latest post in the blog',  
  resolve: function() {  
    PostsList.sort((a, b) => {  
      var bTime = new Date(b.date['$date']).getTime();  
      var aTime = new Date(a.date['$date']).getTime();  
  
      return bTime - aTime;  
    });  
  
    return PostsList[0];  
  }  
},
```

Demo con el API GraphQL de Github

- Prueba (hace falta *loguearse* en Github)
- Documentación
- Otros APIs GraphQL de acceso público

Más sobre GraphQL

- En un API REST hay una URL por recurso, aquí **todas las peticiones van a la misma URL**

```
http://miservidorgraphql/api?query={...}
```

- Para los **errores no se usa el código de estado HTTP**, sino campos en el JSON de la respuesta

```
#query incorrecta, ya que el campo "titulo" no existe
query {
  post(_id:"100") {
    titulo
  }
}

#respuesta del servidor
{
  "errors": [
    {
      "message": "Cannot query field \"titulo\" on type \"Post\"."
    }
  ]
}
```

GraphQL es una tecnología prometedora, pero como todas las nuevas tecnologías tiene un tiempo de vida incierto. **Tampoco hace falta matar ya a REST**

||

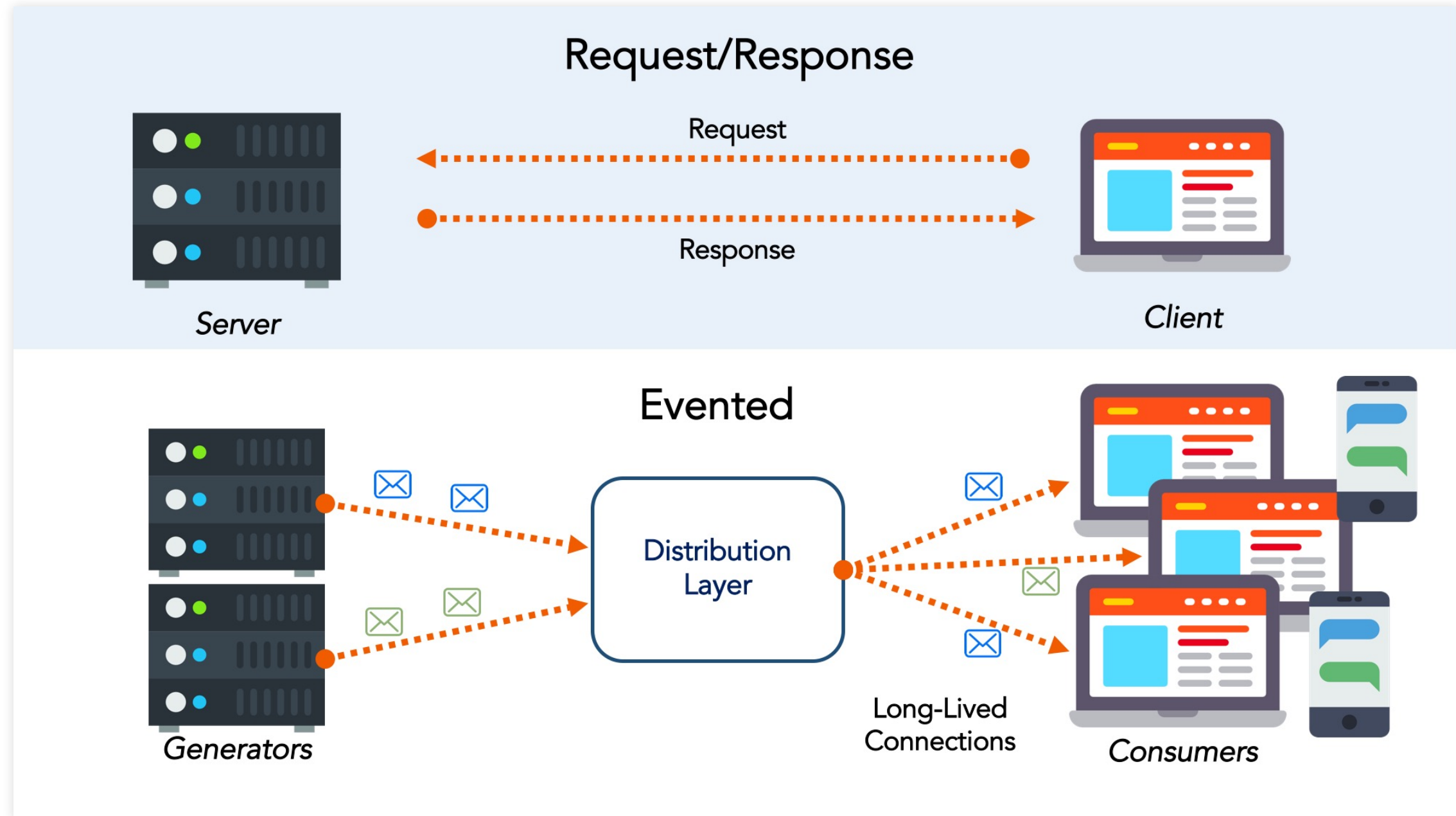
[Por qué API REST está muerto y debemos usar APIs GraphQL - José María Rodríguez](#) (Independientemente de lo tremendista del título, una charla interesante para ilustrar cómo funciona GraphQL y diferencias con REST)

3. APIs basados en eventos

- En ciertos casos queremos estar al tanto de las **actualizaciones del servidor** (p. ej. un *juego online*, un *chat*, ver los *tweets* de nuestro *timeline*, ...)
- El cliente puede hacer ***polling* periódicamente, pero es ineficiente**, es mejor **que el servidor "nos avise"** de que hay nuevos datos

Petición/respuesta vs. eventos

De <https://realtimeapi.io/hub/event-driven-apis/>



Algunas tecnologías web para tiempo real/eventos

Servidor-Cliente(Navegador)

- **Long polling:** el cliente hace *polling* pero la conexión se mantiene abierta hasta que el servidor envía datos. Entonces hay que hacer *polling* de nuevo
- **Server Sent Events:** el cliente recibe de forma asíncrona mensajes y eventos del servidor
- **Websockets:** comunicación bidireccional asíncrona basada en eventos

Servidor-Servidor

- **Webhooks:** se avisa con una petición HTTP cuando hay nuevos datos (no sigue exactamente el esquema de la transparencia anterior)

Polling vs Long polling vs. SSE

Server Sent Events

- **Unidireccionales**, siempre desde el servidor al cliente
- Mensajes de **texto**
- Funciona sobre **HTTP**
- El API **no está soportado** en **Explorer/Edge**, Opera

Ejemplo de SSE

Ejemplo completo en <https://glitch.com/edit/#!/peridot-coin>

```
//Servidor
app.get('/sse', function(pet, resp) {
  //El servidor de eventos debe usar el tipo MIME text/event-stream
  resp.header('Content-Type', 'text/event-stream')
  //Temporizador cada dos segundos
  setInterval(function() {
    //nombre del evento
    resp.write('event: ping\n')
    //datos del evento (texto, en nuestro caso un JSON)
    resp.write(`data: {"timestamp":"${new Date()}"}\n`)
    //Hay que acabar el mensaje con 2 retornos de carro
    resp.write('\n\n')
  }, 2000)
})
```

```
//Cliente
var evtSource = new EventSource("/sse");
evtSource.addEventListener('ping', function(evento) {
  var datos = JSON.parse(evento.data)
  console.log(datos.timestamp)
})
```

Facebook ofrece algunos *endpoints* SSE en su "graph API"

<https://developers.facebook.com/docs/graph-api/server-sent-events>

Endpoints

Endpoint	Description
<code>live_comments</code>	Allows you to subscribe to real-time comments on Live videos.
<code>live_reactions</code>	Allows you to subscribe to real-time reactions to Live videos.

Websockets

- **Bidireccionales**, tanto cliente como servidor pueden enviar mensajes
- Los mensajes pueden contener **texto** o datos **binarios**
- Usa un **protocolo propio** (no es HTTP). Podemos tener problemas para pasar algunos *firewalls*

Ejemplo de websockets

Ejemplo completo en <https://glitch.com/edit/#!/sugar-property>

```
//SERVIDOR
var express = require('express');
var app = express();
app.use(express.static('public'));
var expressWs = require('express-ws')(app);

app.ws('/', function(ws, pet) {
  ws.on('message', function(data) {
    console.log("Mensaje del cliente: " + data)
  })

  setInterval(
    () => ws.send(new Date().toLocaleTimeString()),
    2000
  )
}))
```

```
//CLIENTE
var ws = new WebSocket('wss://' + window.location.hostname)

ws.onmessage = function(evento) {
  console.log("El servidor dice: " + evento.data)
}

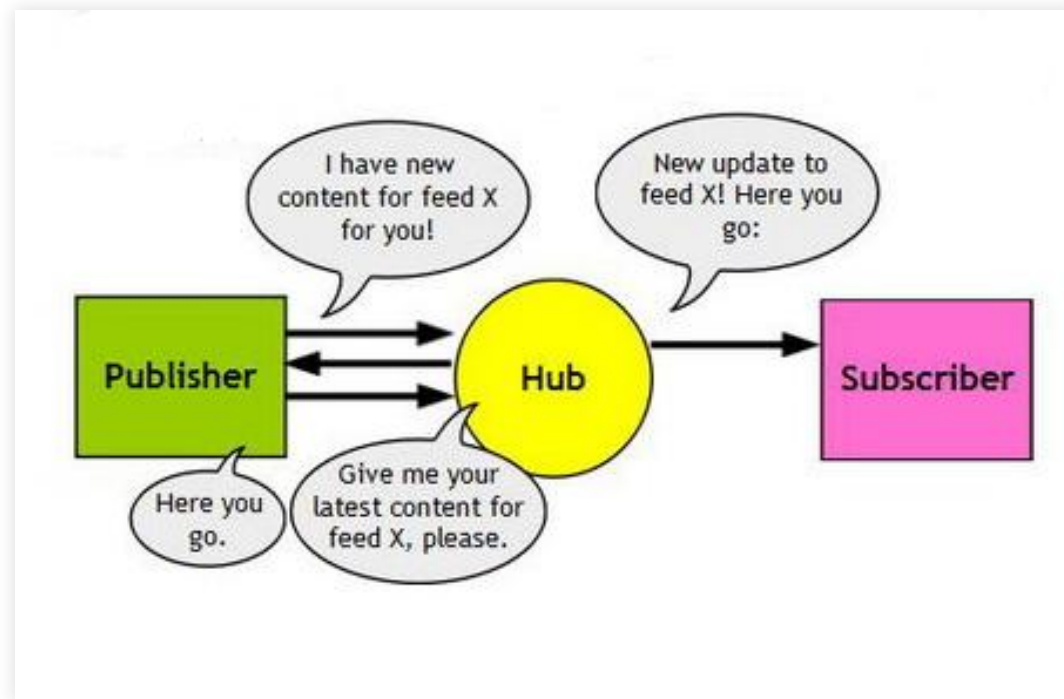
document.getElementById('botonMensaje').addEventListener('click', function() {
```


Webhooks

- Cuando nuestro servidor accede a un servicio de un tercero y queremos que ese tercero nos avise de actualizaciones
- Unir/modificar ideas que ya conocéis de otras asignaturas
 - "Patrón de diseño" **publicar/suscribir**
 - ***Callbacks***
- Cuando hay algún evento importante, el servidor del Webhook lanza una petición POST a una URL de nuestro servidor (*callback*)

PubSubHubbub

- "Extensión" de *webhooks* en el que las suscripciones las gestiona un tercero (*hub*)



- Añade funcionalidades como la **validación de suscripciones** (para evitar que alguien nos suscriba sin nosotros desearlo): cuando nos suscribimos, el servidor envía un GET al que hay que responder de la forma adecuada para confirmar suscripción

Webhooks "ahí afuera"

- Algunos APIs REST públicos que usan *webhooks* y/o PubSubHubbub: Facebook, Instagram, Github, Paypal, Foursquare, algunos de Google (p.ej. Calendar), ...
- Documentación y ejemplos
 - [Facebook real-time updates](#)
 - [Github webhooks](#)

A background image showing several Minions from the 'Despicable Me' franchise sitting in a theater. They are wearing their signature blue overalls and goggles. The Minions in the foreground are looking towards the camera with various expressions, while those in the background are looking forward. The theater seats are red.

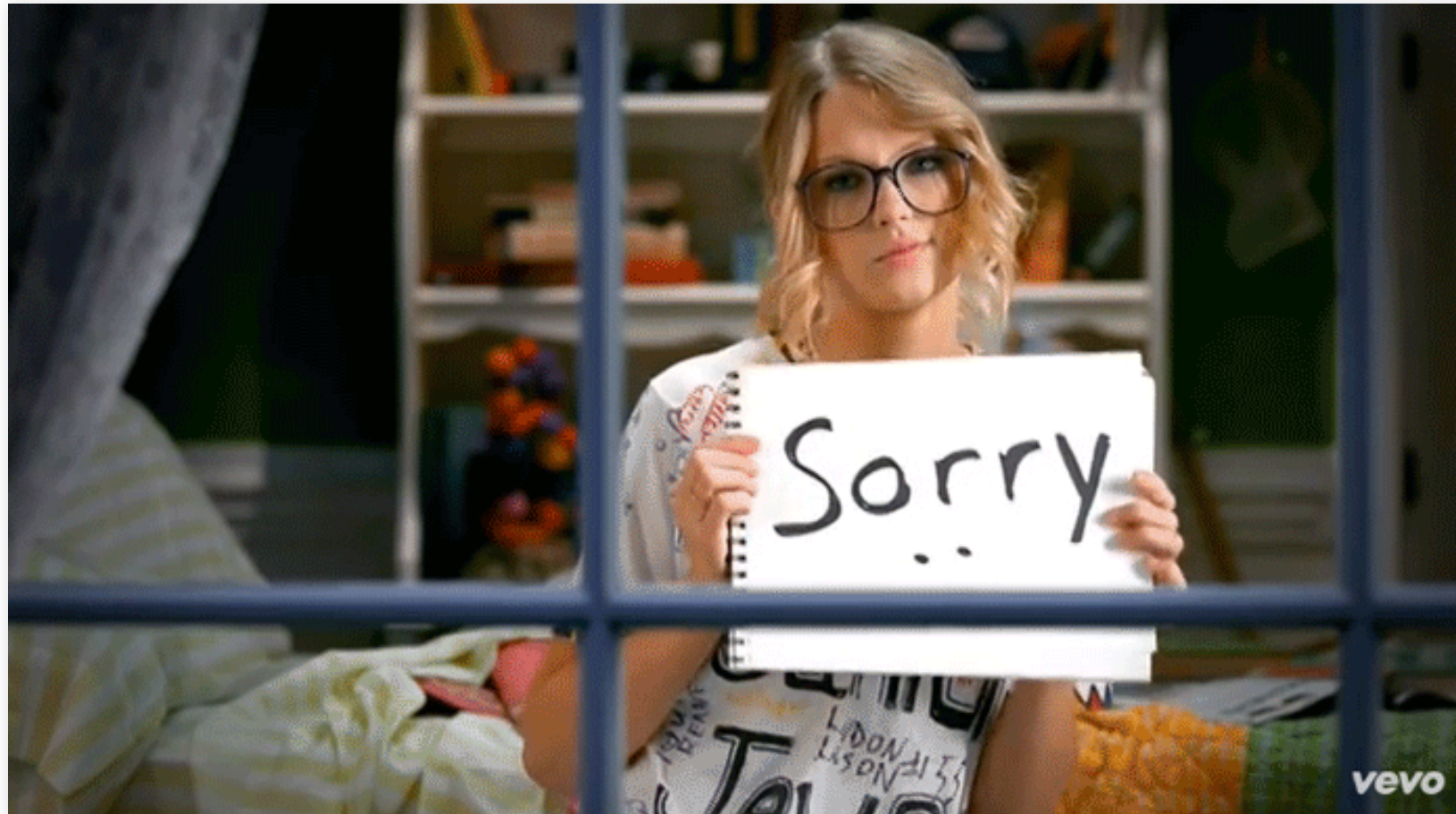
Demo: ***webhooks* en Github**

1. Necesitamos un servidor que pueda recibir peticiones del exterior. Podemos p.ej. [Beeceptor](#) para obtener una URL pública *temporal* a la que hacer peticiones.
2. En el API REST de github, dada la URL de un repo, sus *webhooks* están en `https://api.github.com/repos/:nombre_usuario/:nombre_repo`. Como queremos *crear* uno nuevo, hay que enviar un POST con datos JSON, ejemplo

```
{
  "name": "web",
  "active": true,
  "events": [
    "push",
    "pull_request"
  ],
  "config": {
    "url": URL pública que nos ha dado RequestBin,
    "content_type": "json"
  }
}
```


Limitación importante de los webhooks

El suscriptor al *webhook* debe poder responder a peticiones HTTP. **No nos vale** por ejemplo para avisar a nuestra app cliente **en el navegador**



¿Alguna duda?