Tema 1
Introducción a los servicios web REST
Parte 5 (y última)
Más allá de REST

Contenidos

1. RPC
2. GraphQL
3. APIs orientados a eventos

1. RPC

APIs RPC

Abreviatura de **Remote Procedure Call**, son APIs modelados en torno a **operaciones** en lugar de **recursos**

```
http://api-de-mentira.ua.es/buscarAlumno?dni=11222333
```

¿Qué pasa con 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**.

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

Elecciones de diseño en un API RPC

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

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

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();
```

HTML completo (ejemplo de cliente Thrift desde el navegador)

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 estilo más popular (actualmente) en APIs web públicos

2. Introducción básica a GraphQL

Un problema importante de 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)
```

GraphQL es algo así como esta idea, pero 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

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

En local

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

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: {
      ...
    }
    ...
  })
  ...
})
```

Código completo

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

- GraphQL Explorer (hace falta *estar logueado* en Github, todas las llamadas requieren autenticación)
- 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 interesante, pero **tampoco tiene por qué ser un "REST *killer*"**

"

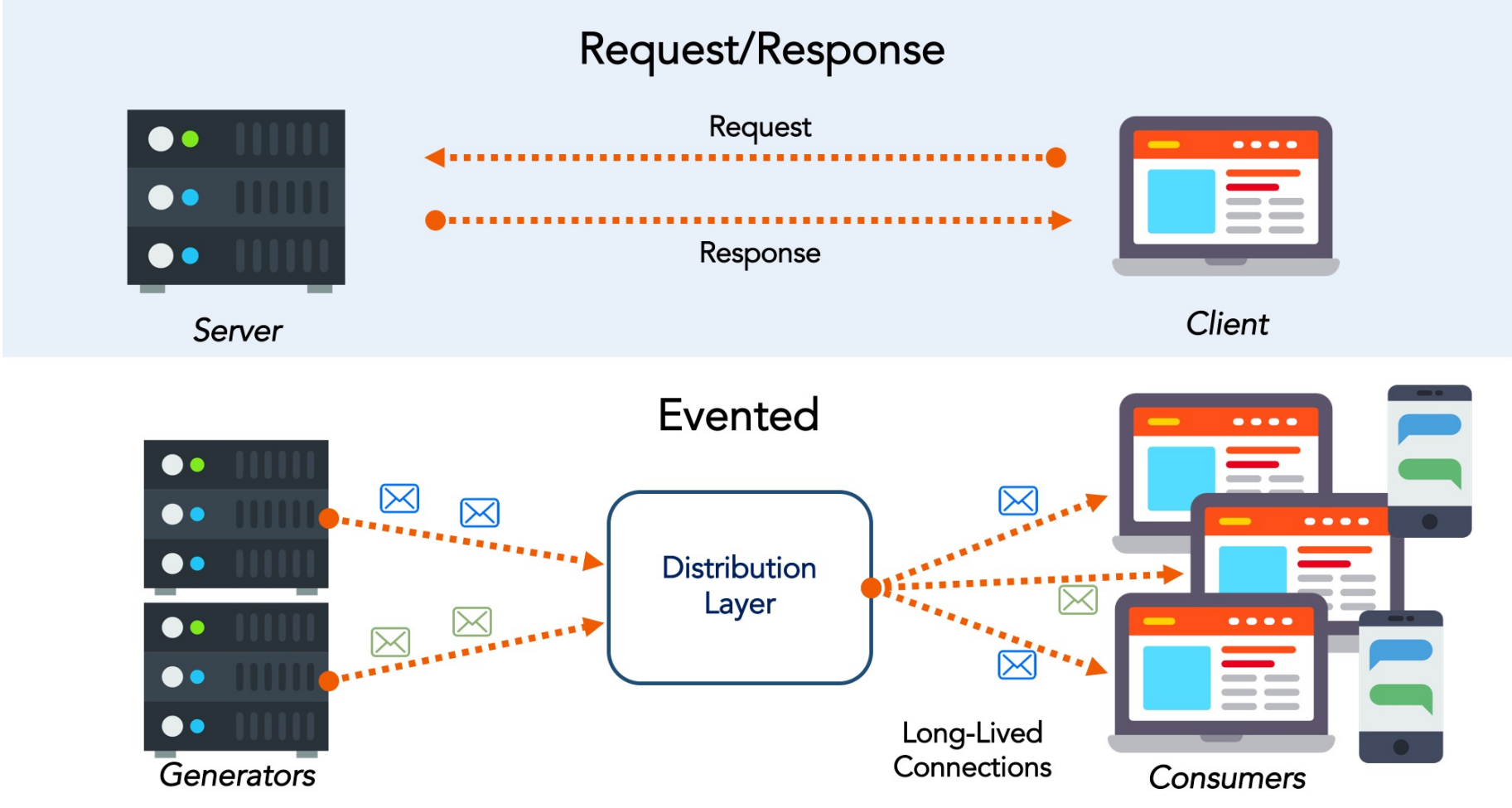
Por qué API REST está muerto y debemos usar APIs GraphQL - José María Rodríguez (Independientemente del *clickbait* del título, una charla interesante y bien explicada, para ilustrar cómo funciona GraphQL y las diferencias con REST)

3. APIs orientados a 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

de Servidor a Cliente(Navegador)

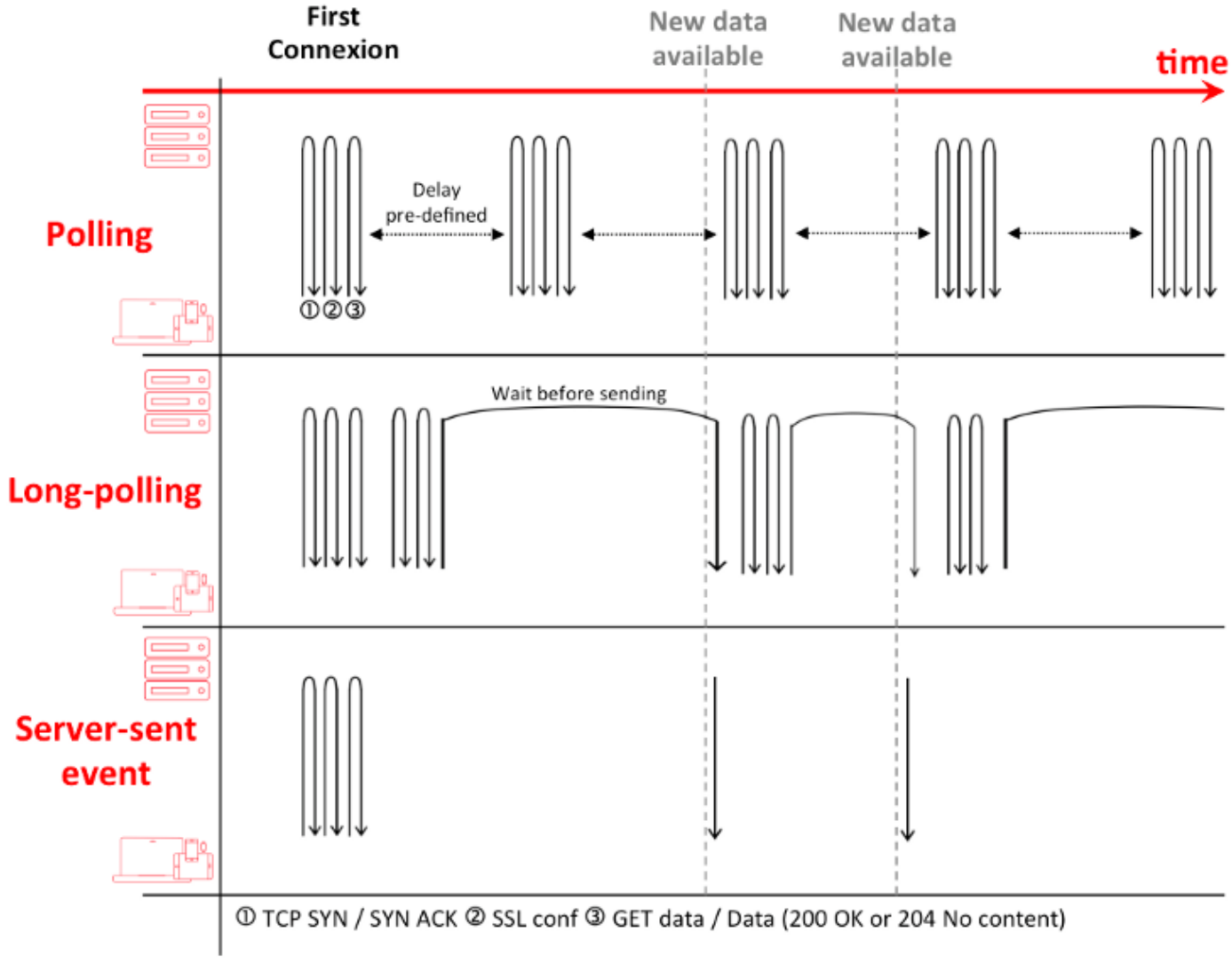
- **Long polling**: el cliente hace *polling* pero la conexión se mantiene abierta hasta que el servidor envía datos. Entonces hay que hacer *polling* de nuevo
- **Server Sent Events**: el cliente recibe de forma asíncrona mensajes y eventos del servidor
- **Websockets**: comunicación bidireccional asíncrona basada en eventos
- **Notificaciones push**: el navegador recibe notificaciones que muestra automáticamente (las veremos en la parte de móviles)

de Servidor a Servidor

- **Webhooks**: se avisa con una petición HTTP cuando hay nuevos datos

Polling vs Long polling vs. SSE

De Polling vs SSE vs WebSocket— How to choose the right one



Server Sent Events

- **Unidireccionales**, siempre desde el servidor al cliente
- Mensajes de **texto**
- Funciona sobre **HTTP**
- Amplio soporte en navegadores actuales (en Edge debe ser una versión reciente)

Formato de los eventos

- El tipo MIME debe ser **`text/event-stream`**
- Cada evento es una línea que comienza por **`data:`**. Un evento puede ocupar varias líneas

```
data: esto es un mensaje

data: este es otro, y tiene
data: dos líneas
```

- Se puede especificar un tipo de evento con una línea que comienza por **`event:`** y una etiqueta arbitraria

```
event: login
data: usuario533
```

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()}"}`)
     //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
`live_comments`	Allows you to subscribe to real-time comments on Live videos.
`live_reactions`	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
  )
}))
```

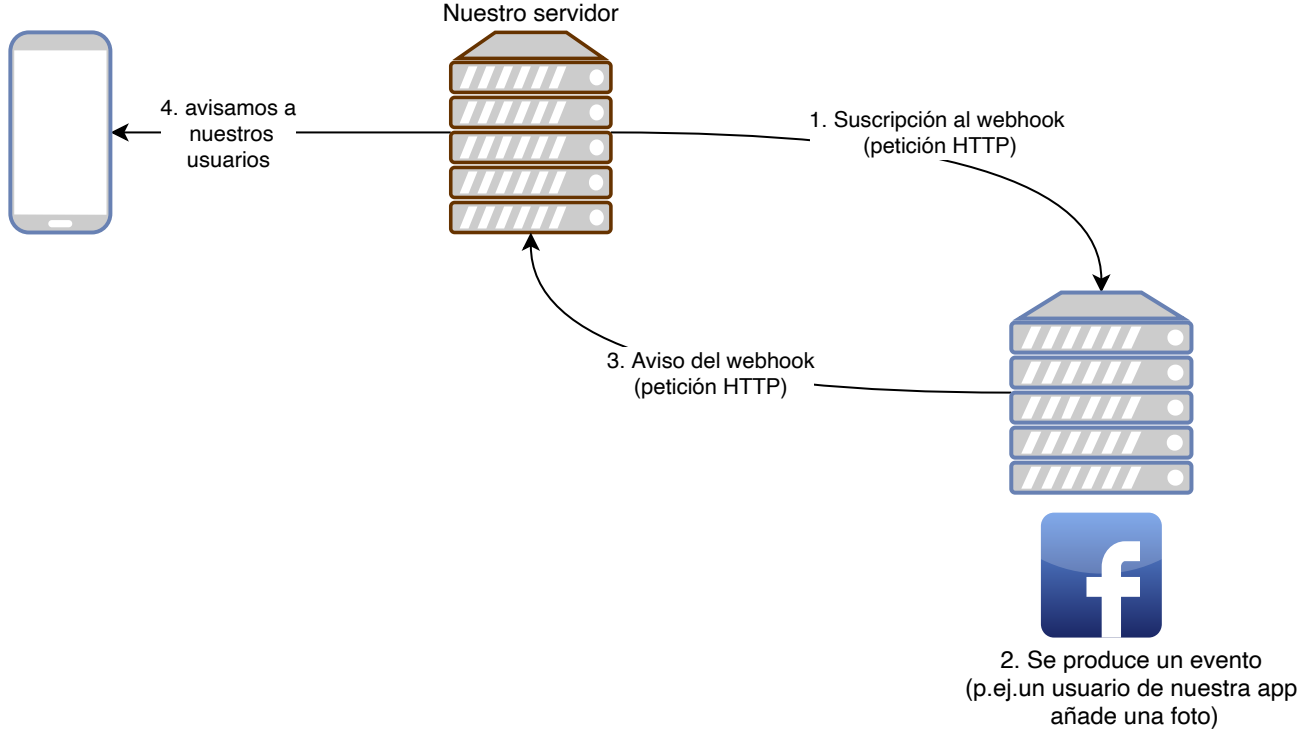
```
//CLIENTE
var ws = new WebSocket('wss://' + window.location.hostname)

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

document.getElementById('botonMensaje').addEventListener('click', function() {
  ws.send(document.getElementById('mensaje').value)
})
```

Webhooks

- Caso de uso típico: nuestros usuarios lo son también de un servicio de un tercero y queremos que ese tercero nos avise de actualizaciones
- Ante un evento al que estamos suscritos, el servidor del Webhook lanza una petición POST a una URL de nuestro servidor (*callback*)



Ejemplos reales de webhooks

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

¿Alguna duda?