Tema 1: APIs web

Parte I: Introducción a los APIs REST

Aplicaciones Distribuídas en Internet, curso 2023-24

Contenidos:

1. Tipos de APIs web: RPC vs REST
2. Introducción a los APIs REST
3. Convenciones básicas en CRUD

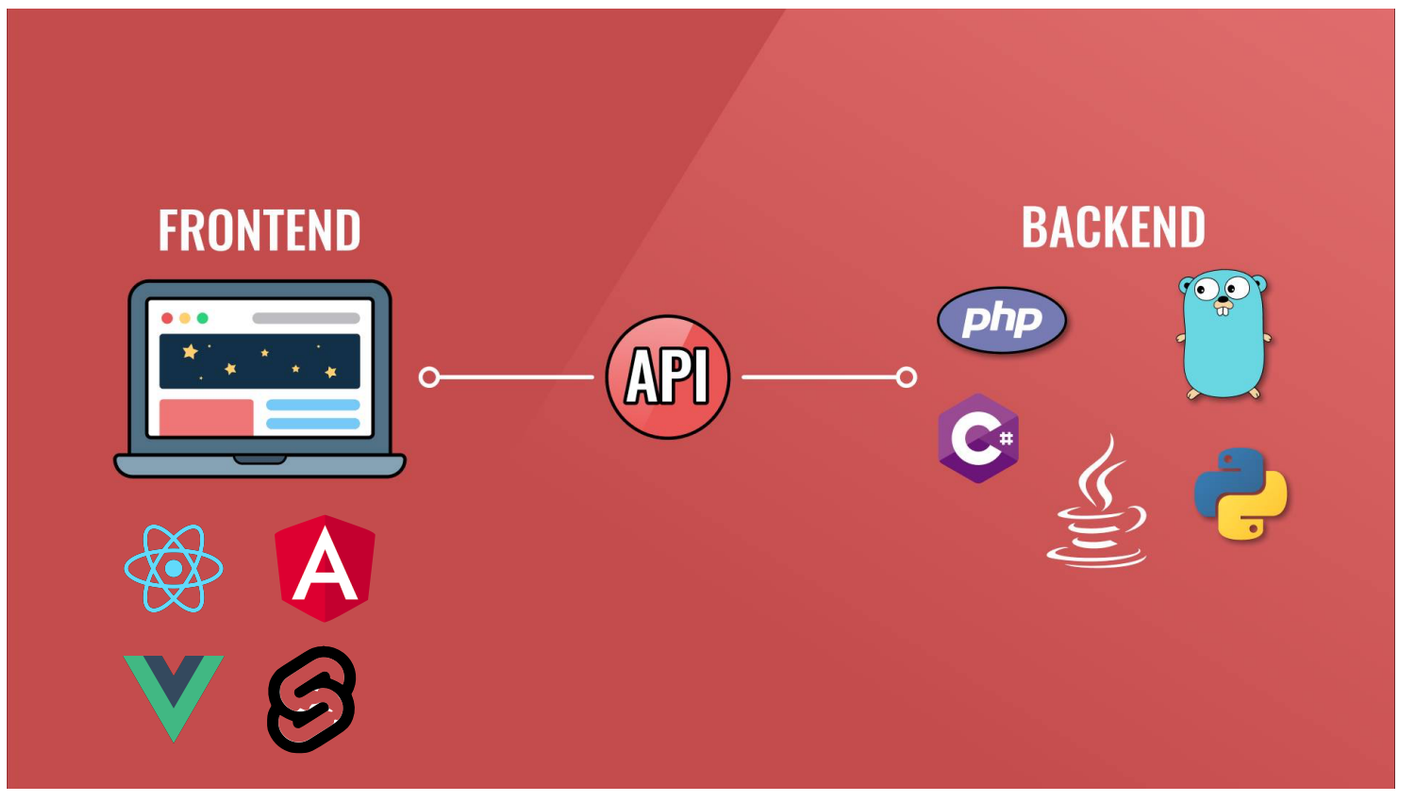
API

- La **interfaz** de un sistema destinada al **desarrollador**, no al usuario final
- Ejemplos: Librería estándar de C++, API Collections de Java
- Aquí nos interesan las **APIs Web** ya que son las que necesitamos para construir aplicaciones Web: **APIs remotas accesibles mediante el protocolo HTTP**

¿Para qué se puede usar un API Web?

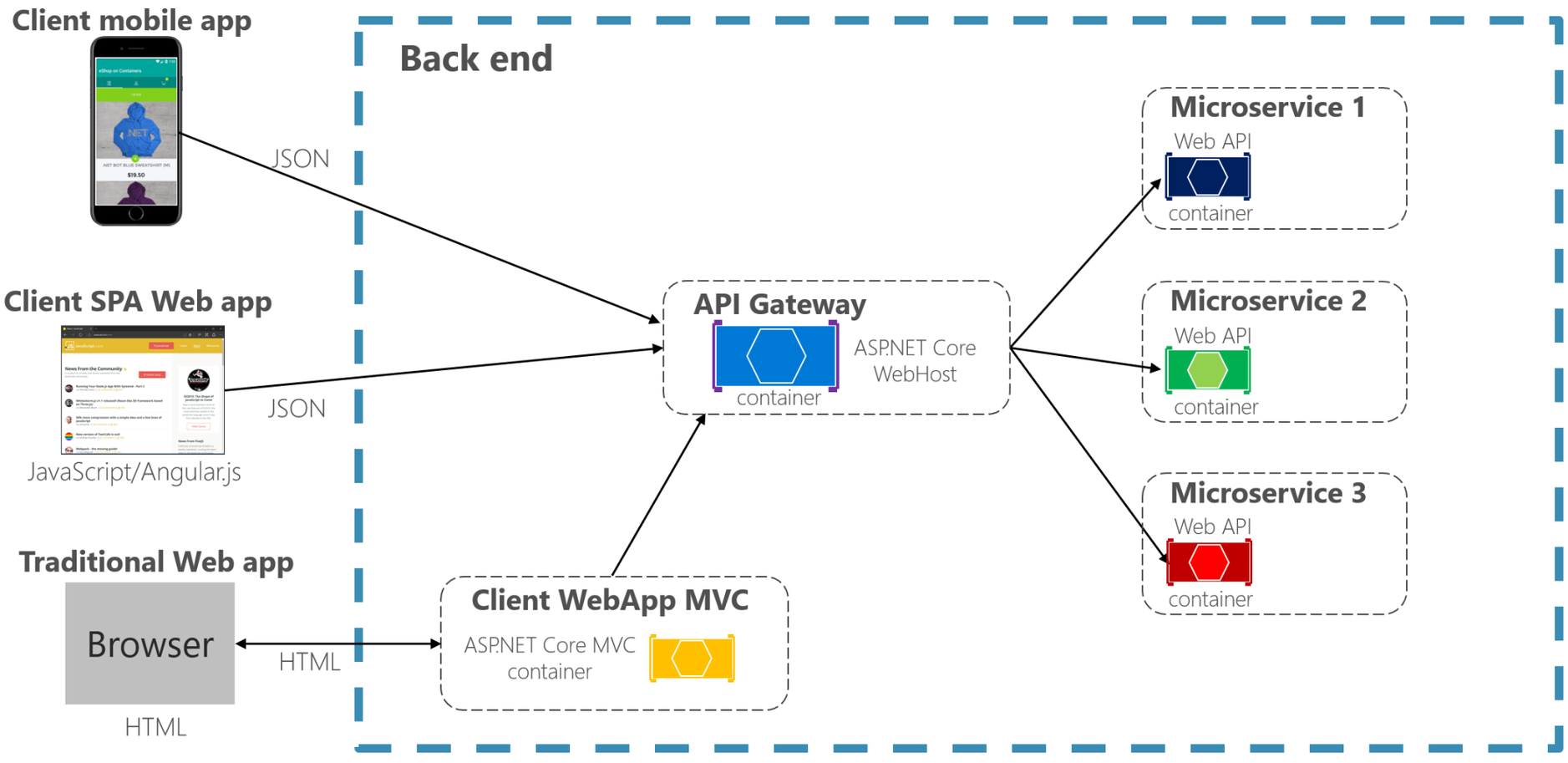
- Integrar sistemas heterogéneos
- Dar acceso a sistemas “legacy”
- Ofrecer un servicio, bien para dar valor añadido a nuestro negocio (ej. Github) o bien para monetizarlo directamente (ej. Stripe)
- En nuestro caso, **nos interesa para comunicar *frontend* y *backend***

Ejemplo: https://github.com/gothinkster/realworld, una app (*clon de Medium*) hecha con múltiples frontends y backends, para dejar clara la idea de que **si se comunican usando determinada especificación de un API, back y front son independientes**



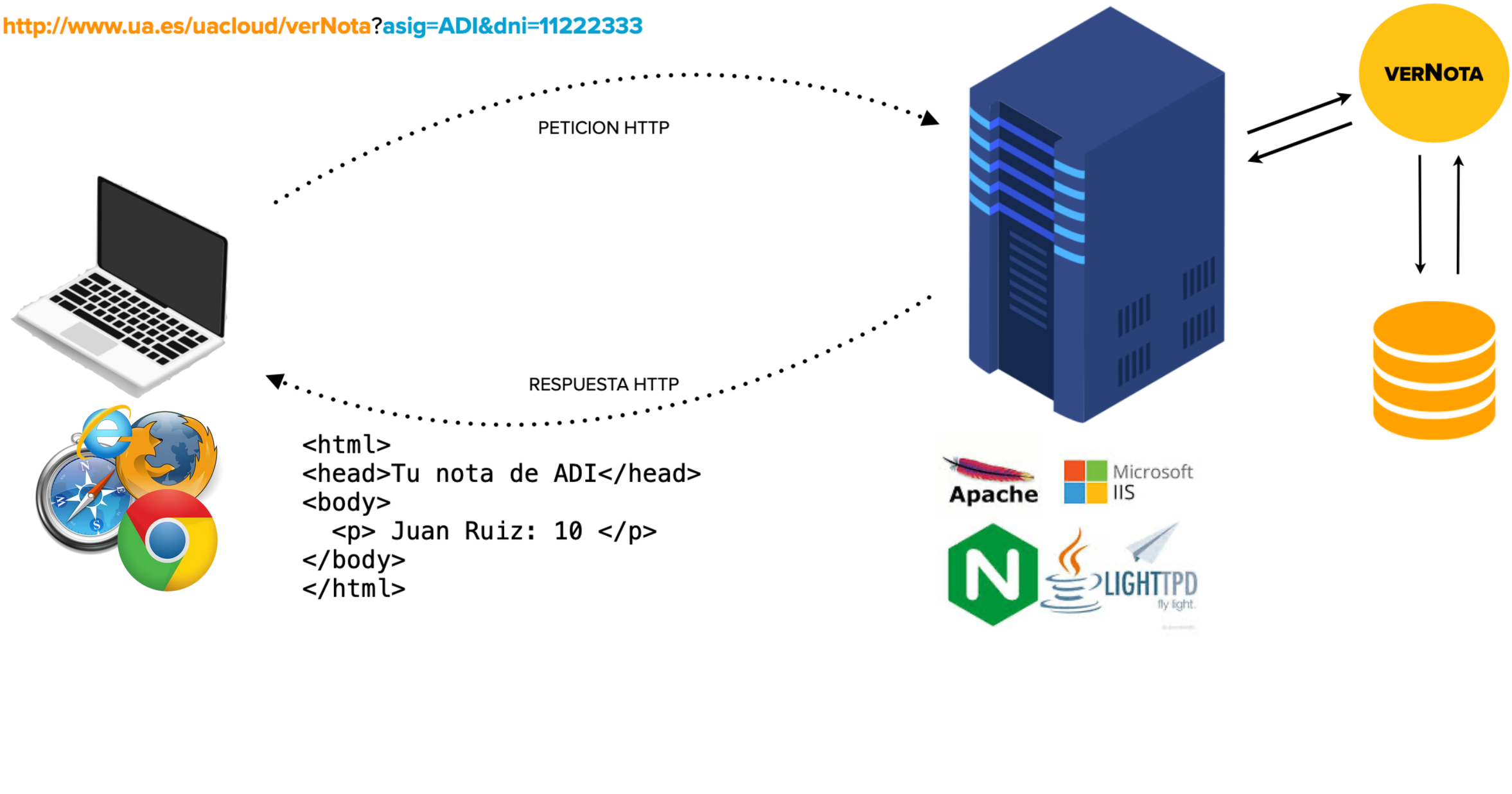
¿Para qué se puede usar un API Web?

- Más concretamente, en nuestro caso usaremos el API para servir de **interfaz entre el *backend* y múltiples *frontends*** (escritorio, móvil, …)



The API gateway pattern versus the Direct client-to-microservice communication. Net microservices Architecture e-book, Microsoft

Aplicaciones web “clásicas”



APIs web de tipo RPC

- Podemos considerar la petición/respuesta HTTP en una app web "clásica" como una llamada a un API remoto
- Por eso estos APIs se denominan **RPCs** (*Remote Procedure Calls*)
- El **concepto central** es la **operación a realizar** (`verNota`, `ponerNota`, `listarAlumnos`,...).
- Nótese que **en cada aplicación las operaciones serán distintas** (`recogerCoche`, `devolverCoche` en una agencia de alquiler de coches,...)
- Prácticamente todos los APIs de bibliotecas de funciones para lenguajes de programación siguen este enfoque (Librería estándar de C++, API Collections de Java,...)

¡ Pero en HTTP ya existe un conjunto de **operaciones estándar**, correspondiente a ciertos **métodos HTTP** (`GET`, `POST`, `PATCH/PUT`, `DELETE`), **que "valen para todo"** (*leer, crear, actualizar, borrar*), no necesitamos operaciones propias!

Con estas primitivas se puede representar cualquier* operación de un API

* casi

Contenidos:

1. Tipos de APIs web: RPC vs REST
2. Introducción a los APIs REST
3. Convenciones básicas en CRUD

Idea central de los APIs REST

Si tenemos operaciones “universales” (leer, crear, actualizar, borrar) que valen para cualquier dominio, **lo que diferencia un API de otro no son las operaciones sino los objetos* con los que operamos**

(*) Aclaración: El término común en REST no es “objetos" sino **recursos**

Elementos importantes en un API REST

- Las **URLs** (== los recursos)
- Los **métodos HTTP** (==las operaciones)
- El **código de estado** (==el resultado de la operación)
- El **formato de los datos** que intercambian servidor y cliente (típicamente JSON)

Las URLs

- Cada **recurso individual** debe tener una **URL única** que lo identifique

```
//esta URL es de un API ficticio
http://api.ua.es/asignaturas/34039
//Esta no, es del API de Github
https://api.github.com/users/ottocol
```

- Todos los recursos de una clase (**colección**) deberían estar en una URL que acaba con la *"clase del recurso en plural"*

```
//TODOS los usuarios de Github
https://api.github.com/users
```

Por motivos prácticos si probamos esta petición no los obtendremos todos

Los métodos

- Una URL por sí sola no sirve para mucho, salvo que digamos qué hacer con el recurso, lo que se consigue **haciendo una petición HTTP a la URL con el método HTTP apropiado** (GET, POST, PATCH, PUT, DELETE)

En la web "clásica" solamente se usaban GET y POST, ya que aunque en HTTP también existían PUT y DELETE, era imposible lanzar una petición de estos tipos desde el navegador *(hasta que apareció AJAX)*

El código de estado

- En REST el código de estado HTTP devuelto por el servidor es importante, ya que indica qué ha pasado con la operación.
- https://httpstatuses.com/

```
//Convenciones similares se usan en otras plataformas
int main() {
	...
	return 0; //En web esto es 200 OK
}
```

El formato de los datos

- **HTML** no es muy apropiado para datos, ya que está diseñado para representar documentos en general

```
<html>
    <head>Tu nota de ADI</head>
    <body>
      <h1>Tu nota:</h1>
      <p>Juan Ruiz:10</p>
    </body>
</html>
```

El formato de los datos (II)

- Una alternativa usada inicialmente en APIs web fue **XML**, es apropiado para datos, pero es “tedioso”… 😑

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<nota>
  <alumno>
     Juan Ruiz
  </alumno>
  <valor>
     10
  </valor>
</nota>
```

El formato de los datos (III)

- Actualmente se tiende a usar JSON en todos los APIs

```
{
   "alumno": "Juan Ruiz",
   "nota": 10
}
```

REST formalmente

Tiene su origen en la tesis doctoral de Roy Fielding,e implica 6 condiciones:

- Cliente-servidor
- Interfaz uniforme
  - Identificación de los recursos
  - Representaciones estándar
  - Mensajes auto-descriptivos
  - Hypermedia as The Engine of The Application State (HATEOAS)
- Sin estado
- Cacheable
- Capas (proxys de modo transparente)
- Código "bajo demanda" (opcional)

Contenidos:

1. Tipos de APIs web: RPC vs REST
2. Introducción a los APIs REST
3. **Convenciones básicas en CRUD**

Leer recurso: petición

- **Método GET**
- Normalmente no hacen falta cabeceras HTTP

Para probar peticiones GET simplemente podemos escribir la URL del recurso en la barra de direcciones del navegador. En línea de comandos podemos usar por ejemplo la herramienta `curl` (Unix/Linux)

```
# si no tienes curl puedes probarlo en
https://replit.com/@ottocol/GET-Github-API
# el -i nos muestra el status code y las
cabeceras del servidor
curl -i https://api.github.com/users/octocat
```

Leer recurso: respuesta

- Algunos estados posibles: **200** (OK, se devuelve el recurso), **404** (el recurso con dicho `id` no existe), **401** (credenciales incorrectas), **403** (no tienes permiso para esta operación), **500** (Error del servidor, p.ej. se ha caído la BD)
- La cabecera `Content-Type` especifica el tipo MIME del formato de los datos

```
200 OK
Content-Type:application/json

{
  "login": "octocat",
  "id": 583231,
  "avatar_url": "https://avatars.githubusercontent.com/u/583231?v=3",
  ...
}
```

Crear recurso: petición

- Típicamente **la URL es la de la colección**, ya que el nuevo recurso todavía no tiene un `id` (normalmente lo asigna el servidor)

```
# "user" se pone literalmente, representa el
usuario autentificado en la llamada al API
# luego veremos cómo nos autentificamos
https://api.github.com/user/repos
```

- El método debe ser **POST**
- Se debe enviar el nuevo recurso en el cuerpo de la petición
- Se debe enviar la cabecera `Content-Type` con el tipo de datos
- En alguna parte de la petición habrá que enviar las credenciales de **autentificacion**

Crear recurso: respuesta

- Algunos estados posibles: **201** (OK, recurso creado), **404** (el recurso con dicho `id` no existe), **401** (credenciales incorrectas), **403** (no tienes permiso para esta operación), **500** (Error del servidor, p.ej. se ha caído la BD)
- En caso de **201** Lo más RESTful es **devolver la URL del recurso creado** en la cabecera HTTP `Location` de la respuesta

```
# esto es ficticio, salvo que pudiéramos identificarnos
como octocat
201 CREATED HTTP/1.1
Location: https://api.github.com/repos/octocat/Hello-World
```

Crear recurso: ejemplo con Github

- Normalmente para crear recursos hay que autentificarse, ya veremos con más detalle métodos de autentificación en APIs REST. En el caso del API de Github hace falta un token de acceso personal

```
# Cómo enviar la petición usando la herramienta "curl"
# CAMBIAR "mi_usuario_de_github" por el vuestro
curl -v -H "Content-Type: application/json" \
  -X POST \
  -u "mi_usuario_de_github"  \
  -d '{"name":"NuevoRepo","description":"Repo de prueba"}'
\
  https://api.github.com/user/repos
```

Podéis probarlo en https://replit.com/@ottocol/pruebaPOSTgithub#main.sh (cambiar el usuario por el vuestro de github y darle al botón "run", os pedirá un password que en realidad será el *personal access token*)

Modificar recurso: petición

- **URL:** la del recurso ya existente
- Métodos HTTP **PUT o PATCH**

> Según la ortodoxia REST, **PUT significaría enviar TODOS los datos** del recurso, incluso los que no cambian. **PATCH: enviar solo los que cambian**. No es tan conocido como PUT al ser una adición más reciente a HTTP.

- Los datos se suelen enviar en JSON en el cuerpo de la petición
- Típicamente la petición necesita autenticación

Modificar recurso: respuesta

- **Resultados posibles**: **204** (Recurso modificado correctamente, no hay nada que añadir :)), **404** (recurso no existente), Errores ya vistos con POST (**400**, **401**, **403** **500**, ...)

Eliminar recurso: petición/respuesta

- **URL:** la del recurso a eliminar
- Método HTTP **DELETE**
- Típicamente la petición necesita **autenticación**
- **Resultados posibles**: **204** (Recurso eliminado correctamente, nada que añadir :)), **404** (recurso no existente), Errores ya vistos (**400**, **401**, **403 500**, …)

Todo esto no son más que convenciones, que podrían ser de otra forma...

> "Para diseñar un buen API para los servicios necesitamos usar algo que la gente conozca. Así que, **aunque no hay nada superior desde el punto de vista técnico en REST y JSON** con respecto a usar RPC con un protocolo de más bajo nivel, **usar algo que la gente comprenda bien** [...] **ayuda mucho** en el diseño del API"

Jay Kreps, Lessons from Building and Scaling LinkedIn, QCon NY 2013