CONCEPTOS GENERALES SOBRE API REST

QUÉ ES UNA API

- Forma de describir la forma en que los programas/sitios webs intercambian datos.
- El formato de intercambio de datos normalmente es **JSON** o XML.

¿PARA QUÉ NECESITAMOS UNA API?

- Ofrecer datos a aplicaciones que se ejecutan en un movil
- Ofrecer datos a otros desarrolladores con un formato más o menos estándar.
- Ofrecer datos a nuestra propia web/aplicación
- **Consumir datos** de otras aplicaciones o sitios Web

PROVEDORES DE APIS

- Algunos ejemplos de sitios web que proveen de APIS son:
  - Twitter: acceso a datos de usuarios, estado
  - Google: por ejemplo para consumir un mapa de Google
- Pero hay muchos más: Facebook, YouTube, Amazon, foursquare...
- Pero todavía hay muchos más: directorio de proveedores de APIs

QUÉ SIGNIFICA API REST

- REST viene de, **RE**presentational **S**tate **T**ransfer
- Es un tipo de arquitectura de desarrollo web basada en el estándar HTTP
- Se compone de una lista de reglas que debemos seguir al construir la API
- Hablaremos de **servicios web restful** si cumplen la arquitectura REST.
- Restful = adjetivo, Rest = Nombre

COMO FUNCIONA REST

LLAMADAS AL API

- Las llamadas al API se implementan como peticiones HTTP
  - La URL representa el **recurso**
  - El método (HTTP Verbs) representa la **operación**:
  - El código de estado HTTP representa el **resultado**:

```
GET http://www.formandome.es/api/cursos/1
200 OK HTTP/1.1
404 NOT FOUND HTTP/1.1
```

CREACIÓN DE RECURSOS

- La URL estará “abierta”
  - El recurso todavía no existe
  - No tiene id
- El método debe ser POST

```
POST http://eventos.com/api/eventos/3/comentarios
```

CREACIÓN DE RECURSOS: RESPUESTAS

- 403 (Acceso prohibido)
- 400 (petición incorrecta, p.ej. falta un campo o su valor no es válido)
- 500 (Error del lado del servidor, p.ej. caida de BBDD)
- 201 (Recurso creado correctamente)
- ¿Qué URL tiene el recurso recién creado?
  - Se devuelve en el header Location

ACTUALIZACIÓN DE RECURSOS

- Método **PUT**: se actualizan todos los datos
- Método **PATCH** (2010): pensado para cambiar solo ciertos datos.
- Resultados posibles
  - Errores ya vistos con POST
  - 200 (Recurso modificado correctamente)

ELIMINAR RECURSOS

- Método DELETE
- Algunos resultados posibles:
  - 200 OK
  - 404 Not found
- Tras ejecutar el DELETE con éxito, las siguientes peticiones GET a la URL del recurso deberían devolver 404

REGLAS DE UNA ARQUITECTURA REST

- Interfaz uniforme
- Peticiones sin estado
- Cacheable
- Separación de cliente y servidor
- Sistema de Capas
- Código bajo demanda (opcional)

INTERFAZ UNIFORME

- La interfaz de basa en recursos, por ej:

```
Empleado {
  id,
  nombre,
  apellido,
  puesto,
  sueldo
}
```

- Se envía un id para su gestión (DELETE, PUT...) por el consumidor de la API
- Uso de formatos estándar, normalmente JSON
- La arquitectura interior (BD..) es transparente

INTERFAZ UNIFORME: MENSAJES DESCRIPTIVOS

- Mensajes descriptivos:
  - Usar las características del protocolo http para mejorar la semántica:
    - HTTP Verbs
    - HTTP Status Codes
    - HTTP Authentication
  - Procurar una API sencilla y jerárquica y con ciertas reglas: **uso de nombres en plural**

PETICIONES SIN ESTADO

- http es un protocolo sin estado ---> mayor rendimiento

```
GET mi_url/empleados/1234
DELETE mi_url/empleados/1234
```

- En el DELETE hemos tenido que volver a indicar el identificador del recurso

CACHEABLE

- En la web los clientes pueden cachear las respuestas del servidor
- Las respuestas se deben marcar de forma implícita o explícita como cacheables o no
- Mejoraremos la escabilidad de la aplicación (menos peticiones)
- Mejora el rendimiento en cliente (evitamos la latencia)

SEPARACIÓN DE CLIENTE Y SERVIDOR

- El cliente y servidor están separados mediante la interfaz uniforme
- Los desarrollos en frontend y backend se hacen por separado
- La interfaz de la API es un contrato que deben cumplir frontend y backend

SISTEMA DE CAPAS

- Al cliente le preocupa QUE la API REST funcione
- Al cliente NO le preocupa COMO funciona la API
- El uso de otras capas:
  - Aumentar la escalabilidad (sistemas de balanceo de carga, cachés)
  - Implementar políticas de seguridad
  - Es transparente para el cliente

CÓDIGO BAJO DEMANDA (OPCIONAL)

- Los servidores pueden ser capaces de aumentar o definir cierta funcionalidad en el cliente transfiriéndole cierta lógica que pueda ejecutar:
  - Componentes compilados como applets de Java
  - JavaScript en cliente.

CONSEJOS PARA ELABORAR UNA API REST

VERSIONES DEL API

- Cambios en el código no deberían afectar a la API
- Si hay cambios en el API se deben usar versiones para no fustrar a los desarrolladores
- La mejor opción es añadir un prefijo a las URLS:

```
GET /v1/geocode HTTP/1.1
Host: api.geocod.io

GET /v2/geocode HTTP/1.1
Host: api.geocod.io
```

HTTP VERBS

- Si realizamos CRUD, debemos utilizar los HTTP verbs de forma adecuada para cuidar la semántica.
  - *GET*: Obtener datos. Ej: GET /v1/empleados/1234
  - *PUT*: Actualizar datos. Ej: PUT /v1/empleados/1234
  - *POST*: Crear un nuevo recurso. Ej: POST /v1/empleados
  - *DELETE*: Borrar el recurso. Ej: DELETE /v1/empleados/1234
  - *PATCH*: Para actualizar ciertos datos

NOMBRE DE LOS RECURSOS

- Plural mejor que singular, para lograr uniformidad:
  - Obtenemos un listado de clientes: *GET /v1/clientes*
  - Obtenemos un cliente en partícular: *GET /v1/clientes/1234*
- Evita guiones y guiones bajos
- Utiliza nombres y no verbos
- URL's cortas y jerárquicas y semánticas
  /v1/clientes/1234/pedidos/203

CÓDIGOS DE ESTADO

- Se utilizan los códigos de estado de http
- Si realizamos un request de POST deberemos devolver un 201.
- Se pueden producir múltiples errores en la llamada al API:
  - falta de permisos
  - errores de validación
  - O incluso un error interno de servidor.
- Siempre se debe devolver un código de estado HTTP con los requests.
- Añadir un mensaje de error si es necesario.

FORMATO DE SALIDA

- En función de la petición nuestra API podría devolver uno u otro formato.
- Nos fijaremos en el ACCEPT HEADER

```
GET /v1/geocode HTTP/1.1
Host: api.geocod.io
Accept: application/json

*GET /v1/geocode HTTP/1.1
Host: api.geocod.io
Accept: application/xml
```

- En principio utilizaremos JSON: sencillo y simple
- XML no es nuestro amigo: schemas, namespaces...
- Si no es un requerimiento, evitaremos XML

SOLICITUDES AJAX ENTRE DOMINIOS

PROBLEMÁTICA

- El modelo de seguridad de las aplicaciones web no permite en principio realizar peticiones Ajax entre dominios
- En general esto es OK, a nadie le gustaría que código malicioso de una URL estuviera accediendo a ningún dato de la solapa del navegador que está abierta con nuestra cuenta bancaria

- Pero también es un problema, por ejemplo con AJAX: una página de http://localhost en principio no puede hacer una petición AJAX a Google
- Como consecuencia, se han tenido que idear diversos “trucos”/técnicas para intentar sobrepasar este límite

TÉCNICAS PARA EVITAR LAS RESTRICCIONES DE SEGURIDAD

- Lo más habitual es el uso de JSONP (JSON con Padding)
- Se basa en que aunque no podemos consumir datos de otro dominio vía XHR, si podemos cargar un script de dicho dominio

CORS

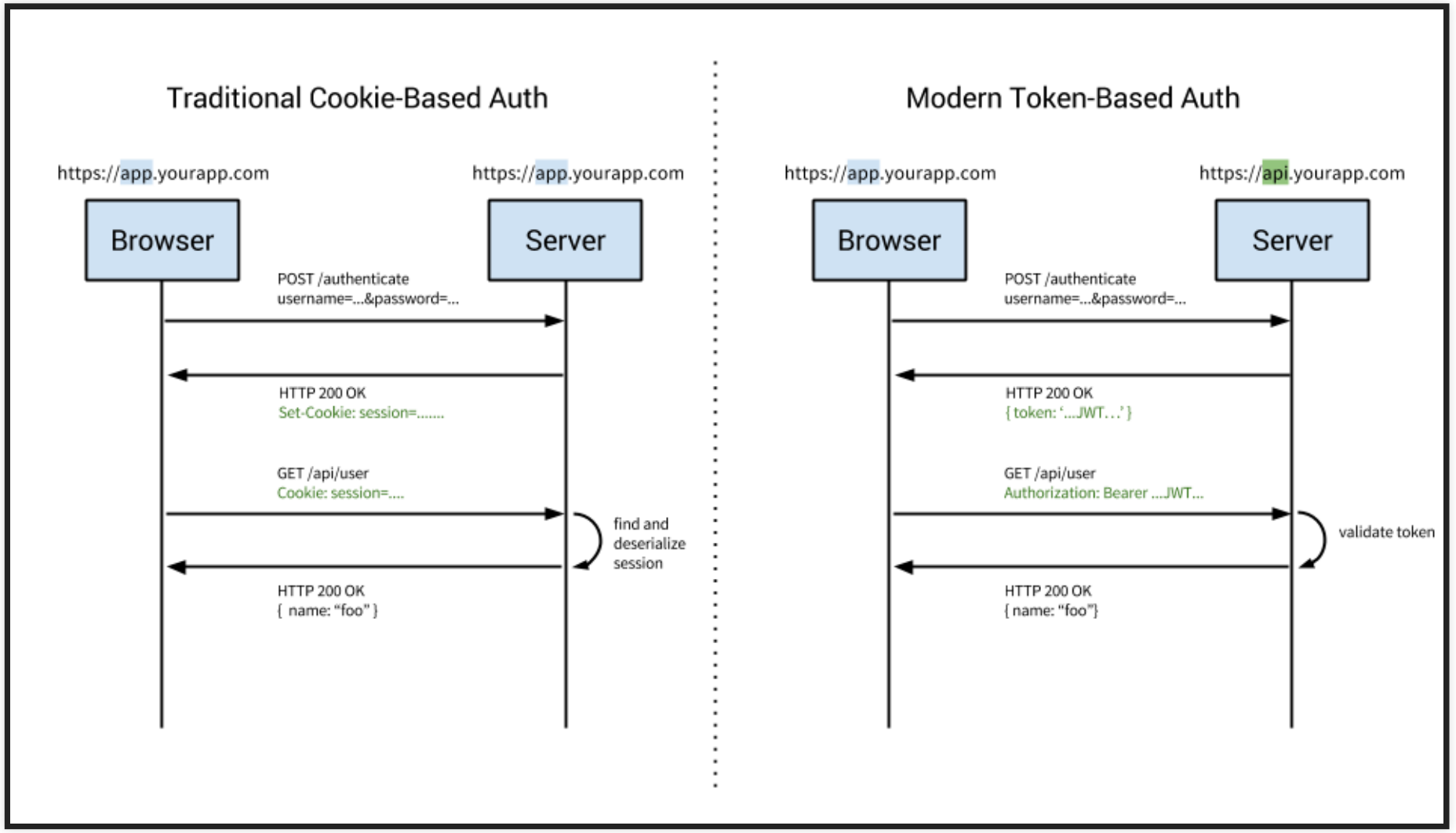
- En el 2008 se publicó la primera versión de la especificación XMLHttpRequest Level 2:
  - Peticiones AJAX entre dominios
  - Eventos de progreso
  - Envio de datos binarios.
- CORS es el acrónimo de Cross-origin resource sharing.
- CORS requiere configuraciones en el servidor

AUTENTICACIÓN Y VALIDACIÓN EN API REST

MÉTODOS DE AUTENTICACIÓN

- Hay dos formas de implementar la autenticación en servidor:
  - **Basada en cookies**, la más sencilla:
    - El servidor guarda la cookie para autenticar al usuario en cada request.
    - Habrá que tener un almacen de sesiones: en bbdd, Redis...

- **Basada en tokens,** se confía en un token firmado que se envía al servidor en cada petición



¿QUÉ ES UN TOKEN?

- Un token es un valor que nos autentica en el servidor
  - Normalmente se consigue después de hacer login mediante usuario/contraseña
- ¿Cómo se genera el token?
  - Normalmente un hash calculado con algún dato (p.ej. login del usuario + clave secreta)
  - Además el token puede llevar datos adicionales como el login

- ¿Cómo comprueba el servidor que es válido?
  - Generando de nuevo el Hash y comprobando si es igual que el que envía el usuario (100% stateless)
  - O bien habiendo almacenado el Hash en una B.D. asociado al usuario y simplemente comprobando que coincide

BENEFICIOS DE USAR TOKENS

- Uso entre dominios
  - cookies + CORS no se llevan bien

OAUTH

- Para que una app pueda acceder a servicios de terceros sin que el usuario tenga que darle a la app sus credenciales del servicio
  - Ejemplo: una app que permite publicar en tu muro de FB, pero en la que no confías lo suficiente como para meter tu login y password de FB
- Es el estándar en APIs REST abiertos a terceros
- Se basa en el uso de un token de sesión

TERMINOLOGÍA DE OAUTH

- En un proceso AUTH intervienen 3 actores:
  - **Consumer:** El servicio al que el usuario quiere acceder usando una cuenta externa.
  - **Service Provider:** Al servicio de autenticación externo se le llama Service Provider.
  - **El usuario final**

FLUJO EN OAUTH

- El consumer pide un token al service provider, esto es transparente para el usuario.
- El consumer redirige al usuario a una página segura en el service provider, pasándole el token como parámetro.
- El usuario se autentica en la página del service provider, validando el token.
- El service provider envía al usuario de vuelta a la página del consumer especificada en el parámetro oauth_callback.
- El consumer recoge al usuario en la callback URL junto con el token de confirmación de identidad.