Arquitectura API REST

Creación de una API

Qué es una API

- Es una forma de describir la forma en que los programas o los 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 que se apoya totalmente en el estándar HTTP.
- REST se compone de una lista de reglas que se deben cumplir en el diseño de la arquitectura de una 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, en las que:
  - La URL representa el **recurso**

```
<http://www.formandome.es/api/cursos/1>
```

- El método (HTTP Verbs) representa la **operación**:

```
GET <http://www.formandome.es/api/cursos/1>
```

- El código de estado HTTP representa el **resultado**:

```
200 OK HTTP/1.1
404 NOT FOUND HTTP/1.1
```

Creación de recursos

- La URL estará “abierta” (el recurso todavía no existe y por tanto no tiene id)
- El método debe ser POST

```
<http://eventos.com/api/eventos/3/comentarios>
```

Respuesta a la creación de recursos

- Resultados posibles:
  - 403 (Acceso prohibido)
  - 400 (petición incorrecta, p.ej. falta un campo o su valor no es válido)

    - 500 (Error del lado del servidor al intentar crear el recurso, p.ej. se ha caído la BD)
    - 201 (Recurso creado correctamente)
- ¿Qué URL tiene el recurso recién creado?
    - La convención en REST es devolverla en la respuesta como valor de la cabecera HTTP Location

Actualización de recursos

- Método PUT
    - Según la ortodoxia REST, actualizar significaría cambiar TODOS los datos
    - PATCH es un nuevo método estándar HTTP (2010) pensado para cambiar solo ciertos datos. Muchos frameworks de programación REST todavía no lo soportan
- Resultados posibles
    - Errores ya vistos con POST
    - 201 (Recurso creado, cuando le pasamos el id deseado al servidor)
    - 200 (Recurso modificado correctamente)

Eliminar recursos

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

Arquitectura REST

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 ejemplo el recurso Empleado (Id, Nombre, Apellido, Puesto, Sueldo)
- El servidor mandará los datos (vía html, json, xml...) pero lo que tenga en su interior (BBDD por ejemplo) para el cliente es transparente
- La representación del recurso que le llega al cliente, será suficiente para poder cambiar/borrar el recurso:
  - Suponiendo que tenga permisos
  - Por eso en el recurso solicitado se suele enviar un parámetro Id

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 la segunda petición hemos tenido que incidar el identificador del recurso que queremos borrar
- El servidor no guardaba los datos de la consulta previa que tenía el cliente en partícular.
- Una petición del tipo *DELETE mi_url/empleado* debe dar error, ¡falta el id y el servidor no lo conoce!

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.

- En futuras peticiones, el cliente sabrá si puede reutilizar o no los datos que ya ha obtenido.
- Si ahorramos peticiones, mejoraremos la escabilidad de la aplicación y el rendimiento en cliente (evitamos principalmente la latencia).

Separación de cliente y servidor

- El cliente y servidor están separados, su unión es mediante la interfaz uniforme
- Los desarrollos en frontend y backend se hacen por separado, teniendo en cuenta la API.
- Mientras la interfaz no cambie, podremos cambiar el cliente o el servidor sin problemas.

Sistema de capas

- El cliente puede estar conectado mediante la interfaz al servidor o a un intermediario, para el es irrelevante y desconocido.
- Al cliente solo le preocupa que la API REST funcione como debe: no importa el COMO sino el QUE
- El uso de capas o servidores intermedios puede servir para aumentar la escalabilidad (sistemas de balanceo de carga, cachés) o para implementar políticas de seguridad

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

- Los cambios en el código no deberían afectar al 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
- Url's lo más cortas posibles
- Evita guiones y guiones bajos
- Deben ser semánticas para el cliente
- Utiliza nombres y no verbos
- Estructura jerárquica para indicar la estructura: /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.
- CORS es el acrónimo de Cross-origin resource sharing.
- Debemos tener presente, que los navegadores antiguos no soportan CORS.
- Esta nueva especificación, añade funcionalidades nuevas a las peticiones AJAX como las peticiones entre dominios (cross-site), eventos de progreso y envio de datos binarios.
- CORS requiere configuraciones en el servidor.

Autenticación y validación en API REST

Métodos de autenticación

- Básicamente tenemos dos formas de implementar la autenticación en servidor:
  - **Basada en cookies**, la más utilizada:
    - 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
```

https://juanda.gitbooks.io/webapps/content/api/cookie-token-auth.png

¿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.

Revision #1
Created 2023-04-19 18:05:37 UTC by molombo
Updated 2023-04-19 18:06:36 UTC by molombo