14. RESTful API

REST

- es un estilo de arquitectura para diseñar servicios web
- significa *Re*presentational *S*tate *T*ransfer
- usa estándares existentes como HTTP
- comunica al cliente y al servidor
- se representa en 3 niveles de madurez

> *NO ES*:
>
> - Un standart
> - Un protocolo
> - un remplazo de SimpleObjectAccessProtocol (protocolo sobre http que comunica aplicaciones sobre la red)
> - una biblioteca

Características

- arquitectura cliente-servidor.
- *stateless*:
    - cada request se ejecuta de forma independiente del resto de request previas o furturas (no guarda contexto ni estados entre peticiones) -> las APIs Rest son más escalables y faciles de entender
    - cada request contiene TODA la información necesaria para completarse
- *cacheable*:
    - las respuestas se pueden almacenar (Cachear) por el cliente o intermediarios para reducir el ancho de banda, latencia y carga en los servidores.

    - el que una respuesta venga del servidor o de la cache debe ser transparente para el desarrollador
    - se deben de etiquetar con metadatos HTTP que indiquen si se puede cachear o no y durante cuánto tiempo. Se pueden usar los metadatos:
        - expires: `Expires: Fri, 19 NOv 2021 19:20:30 EST`
        - cache control: `Cache-Control: max-age=3600`
        - last-modified: `Last-Modified: Fri, 19 May 2021 09:17:49 EST`
- *compresión*:
    - los diferentes formato que pueden retornar las APIS (texto plano, XML, JSON, HTML) se pueden comprimir para ahorrar ancho de banda sobre la red.
    - normalmenet el cliente informa los mecanismos que soporta (*accept-encoding*) y el server informa qué mecanismo usó para encodear (*content-encoding*)
- *Expone recursos (URIs)*
    - Uniform Resource Identifier
    - identifica univocamente cada recurso con cadenas de carácteres
    - URIs semánticas, orgnizadas por clase/tipo de recurso. Cada recurso se accede por un tipo lógico.
        - `/users` -> clase de recurso: usuarios
        - `/products` -> clase de recurso: producto
    - Por convención: Cada recurso es un sustantivo en plural (no se admiten verbos)
    - Permite una distinción entre recursos principales y subordinados:
        - principales: tiene sentido por sí mismo (ejm: /users, /products, /articles)
        - subordinados: depende de un recurso principal para tener sentido (ejm. direcciones de un usuarios /users/{userId}/addresses)

    - algunos ejemplos:
        - `/clientes` -> todos los clientes
        - `/clientes?nombre=juan` representa a los clientes con nombre juan
        - `/torneos?state={stateTournament}` representa los torneos en un estado en particular
        - `/clientes/1/compras` representa a las compras del cliente con ID 1
        - `compras?cliente=1` las compras del cliente con ID 1 (Desde el recurso principal compras)
- *Usa explicitamente los verbos HTTP y sigue sus standares (status code)* :
    - algunos verbos HTTP
        - GET: solicita representación de un recurso en particular
        - POST: envia una entidad a un recurso específico
        - DELETE: borra un recurso en particular
        - PUT: reemplaza el valor actual de un recurso existente con el contenido de la petición.
        - PATCH: aplica modificaciones parciales a un recurso (no pisa como lo hace put)
        - OPTIONS: describe las opciones de comunicación para el recurso de destino.
    - status code:
        - informational (1xx)
        - success (2xx)
            - ok: 200
            - created: 201
        - redirection (3xx)
        - client error (4xx)
            - bad request: 400
            - autorización requerida 401

        - not found 404
        - request timeout 408
        - conflict 409
    - server error (5xx)
        - internal server error 500
        - bad gateway 502
        - gateway timeout 504
- *Navegable*: el cliente no necesita conocer de antemano las rusta de la API porque las respuesta de un recurso contienen los hipervinculos hacia recursos relacionados (principio de HATEOAS). Por ejemplo (*links*):

```
{
  "id": 42,
  "name": "Leticia",
  "email": "leti@example.com",
  "_links": {
    "self": { "href": "/users/42" },
    "orders": { "href": "/users/42/orders" },
}
```

Security Design Principles

principio/recomendación	Descripción
last privilege	exigir el menor privilegio para realizar acciones
fail-safe defaults	por defaul los recursos no son accesibles (las excepciones son los recursos públicos)
complete mediation	se deben de validar completamente llos accesos a los recursos
keep it simple	diseño y lógica de seguridad lo más simple posible
https	se puede usar enciptación en el protocolo, asi no es inseguro que viaje por internet

principio/recomendación	Descripción
password hashes	no se guardan contraseñas en texto plano, se usan algoritmos de hash seguros
never expose information on URLs	no exponer usernames, constraseñas, API keys (como acces token), ni información sensible en la URL. Algunas opciones para comunicar esta info: - header de la request (API key) - cuerpo del mensaje (username y password) - usar POST cuando se envían datos sensibles (no GET)
Timestamp en los request	para detectar y prevenir ataques de repetición
validación de parámetros de entrada	validar parámetros en la URL para evitar inyecciones y otros tipos de ataque.
monitorirear transacciones sospechosas	detectar patrones inusuales (muchas peticiones de una misma IP), ytomar acciones como limitar la velocidad o aplicar delays.

Monitorireas transacciones sospechosas:

- control de cantidad de request por IP o por mismo Token/jwt/user para evitas DoS o evitar/controlar el uso excesivo que puede bajar la performance de la API
- limitar la velocidad o agregar delays entre una request y otra (dadas transacciones sopechosas), ayuda a reducir la solicitudes excesivas que realetizarían la API
- se puede configurar el límite de uso (por ejemplo APIS de paga como las de google)

Autenticación & autorización

Conceptos previos

- **AUTENTICACIÓN**: el sistema verifica tu identidad
- **AUTORIZACIÓN**: el sistema verifica (conocida tu identidad) qué estás autorizado a hacer y a qué recursos podés acceder.

Basic Auth

- Método simple de autenticación: el cliente envía su usuario y contraseña (codificados en Base64, que no cifra los datos, solo los encodea) dentro del header de Authorization.
- Fácil de implementar, pero *debe usarse con HTTPS para que los datos del header no queden expuestos si alguien los intercepta.*
  Por ejemplo, para la request:

  ```
  GET /api/data HTTP/1.1
  Host: ejemplo.com
  Authorization: Basic Zml1YmE6a0BYNFIkS0ZFYkNu
  ```

la construcción del campo Authorization se crea con los siguientes campos: user es el nombre de usuario, pass es la contraseña en ascii, plain-auth es la concatenación con ':' del user y la contraseña en ascii, authorization es el encodeo en base64 del plain-auth.

```
user: fiuba
pass: k@X4R$KFEbCn
plain-auth: fiuba:k@X4R$KFEbCn
Authorization: Zml1YmE6a0BYNFIkS0ZFYkNu
```

API Keys

- una API KEY es un *token* que el cliente provee cuando realiza su query, sirve como identificadores secretos que el *cliente incluye en cada solicitud para autenticarse*.
- Debe ser secretro (solo conocido por cliente y servidor).
- como viaja a través de mensajes, deben de usarse en conjunto con otros mecanismos de seguridad como HTTPS/SSL.
- Se puede comunicar:
  - en el mismo url (Esto no es seguro porque los browsers guardan el historial): `GET /something?api_key=123`
  - como header:
    ```
    GET something HTTP/1.1
    X-API-key: 123
    ```

- como cookie:
  ```
  GET /something HTTP/1.1
  Cookie: X-API-Key=123
  ```

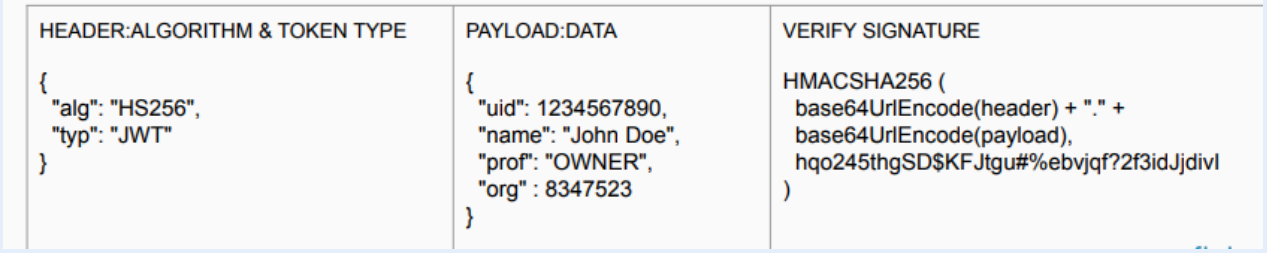
Token Auth / Bearer Authentication

- Usa tokens de seguridad llamados Bearer que representa un token de acceso.
- Bearer Autentication significa "quien posee el token puede aceder."
- El cliente envía un *header de authorization* que contiene el token, y este se envía cada solicitud.
  `Authorization: Bearer <token>`
- algunos tokens que se pueden usar son:
  - JWT: codifica info como: ID de user, Rol, fecha de expiración. se puede verificar sin consultar una base de datos
  - Opaque Token: cadane de texto aleatoria sin info legible (el servidor debe de almacenar los tokens de esta clase que están activos) en la base de datos.}
- se pueden usar tokens refresh para permitir la renovación del token de acceso.

JWT

- el token se genera en el primer paso del proceso de autenticación, el server responde con este cuando el user manda sus credenciales una única vez (las credenciales solo viajan una vez).
- el token no se almacena del lado del serv para luego validar al usuario (a diferencia de opaque token) => mejora la eficiencia porque se evitan múltiples llamados a la base de datos.

- user: fiuba / pass: k@X4R$KFEbCn
- Authorization: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1aWQiOjEyMzQ1Njc4OTAsIm5hbWUiOiJKb2huIERvZSIsInByb2YiOiJPV05FUiIsIm9yZyI6ODM0NzUyM30.2PWhCiy6sgDeBhGFbC1Ws1wIoGgy7eY-44uey_aR0eo
- Authorization - Desencriptado: https://jwt.io/#debugger-io



FORMA DEL TOKEN JWT:
`Header.payload.signature`
donde:

- header y payload están encodeados (no cifrados)
  - *header* indica el algoritmo empleado para la firma y el tipo de token usandose. Es decir:
    ```
    "{alg" = "HS256",
    "type" = "JWT"}
    ```
  - *payload* indica la metadata del usuario en cuestión. Es decir:
    ```
    {}"uid": 1234567890,
    "name": "John Doe",
    "role": "OWNER",
    "org": 1729317}
    ```
- signature (está cifrado): se concatena el header + '.' + el payload y la clave privada del servidor y a esto se le aplica el algoritmo de hash indicado en el header.

GENERACIÓN DE UN TOKEN JWT:

1. El cliente provee de sus credenciales:
   ```
   "{email": "leticia@example.com",
   "password": "contrasenia123"}
   ```
2. El servidor busca en su base de datos (donde guarda el mapeo del email con el hash de la contraseña provista en el registro), compara si la contraseña es la esperada y entonces genera el token (que no usa la contraseña).
3. Para generar el token encodea en base64 el header y el payload. luego aplica el hash a *header + '.' + payload* con la clave privada del serv: Este es el token de acceso, se lo provee en una respuesta al cliente.

4. Cuando el cliente queire acceder a un recurso, agrega este token en el header de la request
5. Cuando el srv recibe la request, desencodea el token y obtiene el header, el payload y la firma. Reaplica el hash al *header + '.' + payload* con su *clave privada* y verifica si el resultado obtenido es equivalente a la firma provista en el token.

> *TOKENS: firmas & stateless*:
> Los JWT pueden ser mensajes o firmados o encriptados o ambos.
>
> - Si un token solo es firmado (no encriptado): el contenido es accesible para cualquiera (campos del header y del payload) pero nadie más puede generar una nueva firma válidad cambiando el payload o el header o alterar el token sin invalidar la firma
> - La API sigue siendo stateless sin necesidad de acceder a la base de datos para buscar un token asociado.

Refresh token

Ya que los access token tienen un tiempo de vida, los refresh token sirven solo para obtener un nuevo access token. Es decir, es una credencial que permite a un usuario obtener nuevos tokens sin necesidad de volver a pedir las credenciales típicas: "user y contraseña".

> *Versionado*:
> rest no privee mecanismos de versionado pero se suelen ver estrategias como:
>
> - usar la URI:
>   - `http://api.fi.uba.ar/v1`
>   - `http://apiv1.fi.uba.ar`
>   - `https://api.fi.uba.ar/20211101/`
> - usar un header personalizado:
>   - `Accept-version: v1`
> - usar un header accept:
>   - `Accept: application/vnd.example.v1+json`
>   - `Accept: application/vnd.example+json;version=1.0`

hateos

hypermedio as the engine of application state. es un principio de REST en el que un cliente interactúa con una app solo a través de hipervínculos proporcionados dinámicamente por las respuestas del servidor.
Es decir, el servidor guía al cliente a través de los recursos disponibles sin que el cliente necesite conocer las URLs de antemano
Esto hace que:

- la app sea navegable
- que el cliente descubra recursos dinpamicamente
- el sistema sea más desacoplado porque el cliente no necesita conocer las rutas de antemano.

Repuestas del servidor

- manetener los más estandarizadas a las mismas
- reducir el tamaño de la respuesta a solo lo necesario
- usar código de errores HTTP
  por ejemplo:
- caso de falla:

```
HTTP CODE: 401 {
"success": false, // solo informativo, el error se define por el HTTP CODE
"message": "Invalid email or password",
"error_code": 1308,
"data": {}
}
```

- caso de éxito:

```
HTTP CODE: 200 {
"success": true,
"message": "User logged in successfully", // optional in
success responses
"data": { }
}
```

acerca de la respuesta

Data es un dta trasnfer object que contiene un mapa de otros objetos. Por ejemplo:

```
{ "success": true,
"message": "User found",
 "data": {
    "user": {
       "id": 2,
        "name": "Juan",
         "email": " juan@fi.uba.ar ",
         "city": {
            "id": 3,
            "name": "Buenos Aires",
            "country": {
               "id": 2,
               "name": "Argentina",
               "code_country": "AR",
               "avatar": "
//localhost:3000/api/v1/country_AR.png "
            }
          }
       },
      "role": "client",
       "favorites": ["blue", "red", "white"]
```

```
        }
    }
```