REST

REpresentational State Transfer

APIs que permitan comunicar a nuestro servidor con sus clientes usando el protocolo HTTP mediante URIs lo suficientemente inteligentes para poder satisfacer la necesidad del cliente

¿En qué consiste?

Se implementan **RECURSOS** para generar comunicación, es decir crea URIs únicas que permiten al cliente entender y utilizar lo que está exponiendo. Por ejemplo:

api.um.edu.ar/users

api.um.edu.ar/users/1405

Verbos HTTP

Cada petición responde a un verbo

POST (create)

GET (read)

PUT (update)

DELETE (delete)

Creación de recursos POST

- 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
```

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 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.

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

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)

200 (Recurso creado cuando le pasamos el id deseado al servidor)

201 (Recurso creado correctamente)

Eliminar recursos DELETE

Tras ejecutar el DELETE con éxito, las siguientes peticiones GET a la URL del recurso deberían devolver 404

DELETE mi_url/empleados/1234

Resultados posibles:

404 (Not Founf)

500 (Server ERROR)

200 (OK)

Códigos de estados HTTP

HTTP verbs CRUD

- 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 sustantivos y no verbos
- Estructura jerárquica para indicar la estructura: /v1/clientes/1234/pedidos/203

Formato de la 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

Solicitudes AJAX entre dominios

El modelo de seguridad de las aplicaciones web no permite en principio realizar peticiones Ajax entre dominios. Tiene que coincidir dirección y puerto.

Si intentamos hacer una petición AJAX de un dominio a otro dominio diferente, por norma general se obtiene un error HTTP Forbidden 403.

Evitar restricción CORS

En el **2008** se publicó la primera versión de la especificación **XMLHttpRequest Level 2**.

CORS es el acrónimo de **Cross-origin resource sharing**.

Esta nueva especificación, añade funcionalidades nuevas a las peticiones AJAX como las peticiones entre dominios (cross-site), eventos de progreso y envío de datos binarios.

CORS requiere configuraciones en el servidor.

Configuración en el servidor

Si tu aplicación está en **www.example.com** y quieres obtener datos de **www.example2.com**, el host example2 permitirá peticiones de example añadiendo una cabecera:

Access-Control-Allow-Origin: http://www.example.com

Permitirá hacer peticiones a cualquier dominio con esta otra cabecera:

Access-Control-Allow-Origin: *

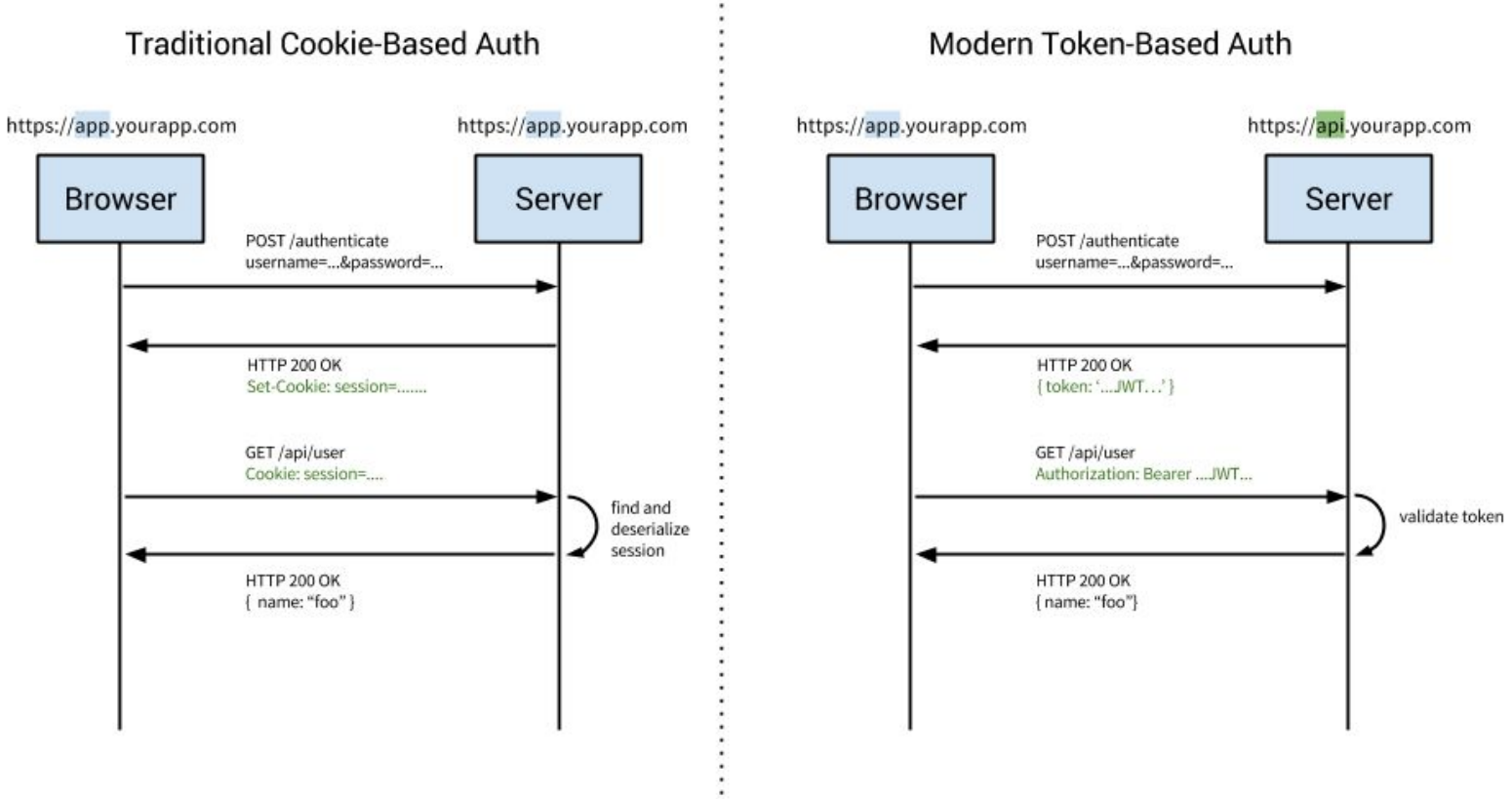
Métodos de autenticación

Basada en **cookies**, la más utilizada:

El **servidor guarda la cookie** para autenticar al usuario en cada request.

Habrá que tener un **almacén de sesiones:** en bbdd, Redis..

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


Traditional Cookie-Based Auth
https://app.yourapp.com
https://app.yourapp.com
Browser
Server
POST /authenticate
username=...&password=...
HTTP 200 OK
Set-Cookie: session=.....
GET /api/user
Cookie: session=....
find and
deserialize
session
HTTP 200 OK
{ name: "foo" }
Modern Token-Based Auth
https://app.yourapp.com
https://api.yourapp.com
Browser
Server
POST /authenticate
username=...&password=...
HTTP 200 OK
{ token: '....JWT...' }
GET /api/user
Authorization: Bearer ....JWT...
validate token
HTTP 200 OK
{ name: "foo"}

¿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
- Beneficio: uso entre dominios

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.

API REST

- **Es solo una convención:** Es una manera de comunicarse entre el servidor y cliente, cada uno tiene sus reglas.
- **El servidor expone recursos:** Los clientes se tienen que adecuarse a como están expuestos.
- **Hace overfetching:** Envía más información de la que se necesita.
- **Múltiples request por vista:** Muy costoso en performance, básicamente es una aplicación en blanco que aún no ha cargado datos o tiene custom endpoints.
- **Documentación ajena al desarrollo:** No hay un estándar por lo que depende mucho del desarrollador para mantenerla.

GraphQL

Query Language

Lenguaje de consulta

Desarrollado por Facebook en 2012

Nos permite hacer consultas y esperar una respuesta predecible.

Es para comunicar clientes y servidores.

Alternativa a REST.

La principal mejora que propone es la optimización

Para comenzar: https://www.graphql.com/tutorials/

Schemas(GQL)

- define las **entidades**
- cómo se **relacionan** entre ellas,
- cuáles son las entidades que están **disponibles** para cada cliente,
- están compuestos de **types** los cuales se conocen como scalars.

Scalars

Permiten definir las propiedades de las entidades

Los tipos que nos permite manejar son:

Int: Números enteros.

Float: Números con decimales.

String: Cadenas de texto.

Boolean: maneja los valores True o False.

ID: Identificador único(GQL se encargará de esto) este puede ser de tipo Int o String

Objects

```
// al usar Type indica que esto es un objeto
    type Curso {
        // al usar el signo ! indica que
        // el valor es obligatrio
        id: ID!
        descripcion: String
        // la utilizar [] indica que es una lista
        // los que nos indica que puedes tener 1 o mas
        // profesores
        profesores: [Profesor]
    }
```

```
type Profesor {
        id: ID!
        nombre: String
        edad: Int
        tieneCurso: Boolean
    }
```

Interface

```
/// Al usar interface se indica el uso de interface
  interface Perfil {
    // para este ejemplo se setea
    // el uso de campos obligados
    nombre: String!
    email: String!
    edad: Int!
  }
```

```
// Al usar implements se indica que usar al interface
Perfil
  type Alumno implements Perfil {
    // De igual manera tenemos que declarar
    // los campos que se utilizan en la interface
    nombre: String!
    email: String!
    edad: Int!
    cruso: String
  }
```

Unión

Permite agrupar diferentes tipos en los cuales se puede realizar una búsqueda siempre dentro de los tipos agrupados, ejemplo:

union Busqueda = Profesor | Curso

```
// Objecto que muestra el resultado de la búsqueda
    type QueryBusqueda {
        // Al usar [] indica que puede ser 1 o más valores
        // prácticamente puede ser una lista
        results: [Busqueda]
    }
```

```
// Objecto en el que se realiza la busqueda
 type Curso {
    id: ID!
    descripcion: String
    profesores: [Profesor]
 }


    type Profesor {
        id: ID!
        nombre: String
        edad: Int
        tieneCurso: Boolean
    }
```

Root Type: Query

Es el punto de entrada para realizar las consultas

```
// Esta declaracion es epsecial ya que
// Dicta el unicio punto de entrada
Type Query {
    // Dentro se colocan los puntos de entrada  y se le asignan las
       entidades a las cuales puedes tener acceso
     cursos: [Curso]
     profesores: [Profesores]
    // También dentro de los puntos de entrada se le pueden
    // mandar parametros para resolver querys es importante
    // mencionar que al resolver un query se tiene que indicar
    // en que entidad se resuelve dicho query.
     curso(id: String!): Curso
     profesor(id: String!, limite: init): Profesores
}
```

Root Type: Mutation

GQL no solo permite realizar consultas sino que también tiene la capacidad de insertar, borrar y editar elementos

Mutation

```
// punto de entrada especial que permite agregar, modificar y borrar contenido
type Mutation {
  // Declaración del punto de  entrada
    agregarCurso {
     // Declaración de los campos a modificar o agregar
        descripcion: String
        profesorId: String
    } : Curso // identidad a la que se le agregaran los nuevos campos
}
```

Conclusión

Como hemos podido ver, GQL nos da herramientas para desarrollar un API de forma rápida, natural e independiente del acceso a base de datos o en algunos casos conectarnos a un CMS.

Además soporta una gran cantidad de lenguajes y clientes.

GraphQL

Lenguaje tipado y validable: Le damos una forma de lo que recibe y lo que devolvemos, además de agregarle seguridad.

El Cliente define que recibe: Haciendo una consulta, de la estructura que se define como respuesta.

Envía lo necesario: Se tiene control total de las respuestas que se esperan del servidor.

Hace un solo request por vista: Se maneja un solo row, prácticamente en solo request puedes mandar todo lo que necesitas.

GraphQL

Problema de seguridad: sobre todo es difícil de controlar que los usuarios no realicen consultas que sobrecarguen el sistema.

Problema de caché: el aplicado de sistemas de caché es mucho más complicado que en los servicios REST.

REST vs GQL

Con un ejemplo

Ejemplo

En base a la siguiente API de Star Wars debemos de obtener:

- nombre del personaje,
- fecha de nacimiento,
- lugar de nacimiento,
- películas en las que aparece el personaje

El personaje seleccionado será Luke Skywalker

Si lo hacemos con servicios REST

1. https://swapi.dev/
2. https://swapi.dev/api/planets/1/
3. https://swapi.dev/api/films/2/
4. https://swapi.dev/api/films/6/
5. https://swapi.dev/api/films/3/
6. https://swapi.dev/api/films/1/
7. https://swapi.dev/api/films/7/

Ahora vamos a resolverlo en GQL

https://swapi.apis.guru/

```
{
    person(personID:1)
    {
        name,
        homeworld{
            name
        }
        filmConnection
        {
            films{
                title
            }
        }
    }
}
```