PROGRAMACIÓN WEB – SEMINARIO 7

Programación basada en APIs con JS

JavaScript

Dr. José Raúl Romero Salguero
jrromero@uco.es

S7-1.
Fundamentos de las APIs

Definición de API

¿Qué es una API?

API = *Application Programming Interface*

- Conjunto de **reglas**, **contratos** y **formatos** que permiten a dos sistemas **comunicarse**.
- En el desarrollo web actual:
  - El cliente (JS en navegador) llama a un **servicio remoto** mediante **HTTP**.
  - Ese servicio devuelve **datos**, normalmente en **JSON**.

Concepto clave

- Una API define **qué** información está disponible, **cómo** solicitarla y **en qué formato** se devuelve la respuesta, **qué errores** pueden suceder.

Ejemplos habituales

- Autenticación con Google: Google OAuth API
- Acceso a mapas y tráfico: Google Maps API (*pago*)
- Una SPA pidiendo datos a su *backend*: API REST interna

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <meta charset="UTF-8"> <title>Ejemplo API</title>
  </head>
  <body>
    <div id="avatar-container"></div>
    <script>
      const nombre = "Raúl Romero";
      const urlAvatar = `https://ui-
avatars.com/api/?name=${encodeURIComponent(nombre)}`;

      const img = document.createElement('img');
      img.src = urlAvatar;
      img.alt = nombre;
      img.width = 128;
      img.height = 128;

      document.getElementById('avatar-
container').appendChild(img);
    </script>
  </body>
</html>
```

La importancia de las APIs

- Las aplicaciones modernas se basan en **arquitecturas desacopladas**:

  frontend ↔ *backend* ↔ servicios externos.

- Las **APIs** permiten integrar:
  - Bases de datos → mediante un *backend*.
  - Servicios externos → autenticación, pagos, mapas, mensajería, analíticas.
  - Microservicios dentro de la misma organización.

Hoy en día, ¡saber implementar APIs es necesario para cualquier desarrollador!

Fundamentos de API REST

Fundamentos de API REST

REST = *Representational State Transfer*

- Estilo arquitectónico para diseñar sistemas distribuidos basado en **HTTP**, **recursos**, **identificadores claros** y **operaciones estándar**.
- **Principios clave**:
  - **Cliente-Servidor**.
  - ***Stateless***: cada petición contiene toda la información necesaria (no se guarda el estado del cliente).
  - **Interfaz uniforme**: Recursos identificados por URIs, uso de HTTP, representaciones estándar para recursos con JSON.
  - **Sistema por capas**: proxys, cachés, etc.

Fundamentos de API REST

¿Es válida la siguiente invocación a API?

https://pokeapi.co/api/v2/pokemon/25

¿Cómo diseñar una API?

- Utiliza sustantivos: los verbos los debe aportar HTTP

```
GET /users
GET /orders/43
```

```
GET /getUsers
POST /createUser
```

- Rutas para colecciones de recursos:
  - **Concepto + ID**:
    https://pokeapi.co/api/v2/pokemon
    https://pokeapi.co/api/v2/pokemon/2
  - **Plural + ID**:
    https://jsonplaceholder.typicode.com/posts
    https://jsonplaceholder.typicode.com/posts/2

Buena práctica

Utiliza sustantivos en **plural** para **colecciones**, en **singular** cuando representan un **recurso individual**

¿Cómo diseñar una API?

- Casos incorrectos de uso:
  - `GET /user?id=15` ⊗ GET debería ser /users/15
  - `GET /allUsers` ⊗ nombre poco REST (no uses *camelCase*)
  - `GET /users/listAll` ⊗ la colección completa ya es /users
  - No mezcles singular y plural:
    - `/user/15/orders` → `user` debería ser `users`
    - `/users/15/order` → `order` debería ser `orders`
  - No utilices verbos, ni acciones (no se deben sustituir los métodos HTTP)
    - `POST /users/create`
    - `DELETE /users/remove/15`
  - Si necesitas representar una acción, probablemente no es un recurso REST puro

¿Cómo diseñar una API?

- Los recursos relacionados y anidados se utilizan cuando un recurso depende jerárquicamente de otro (p.ej. comentarios de un post, pedidos de un usuario)
  1. Anidar solo cuando un recurso depende conceptual del padre

     ```
     GET /posts/1/comments
     GET /users/10/orders
     ```

  2. Mantener un máximo de 2 niveles

     ```
     GET /users/{id}/orders
     ```

  3. Permitir también el acceso directo por ID

     ```
     GET /orders/{id}
     ```

- Si necesitas 3+ niveles, entonces tu modelo no está bien definido
- Considera que los *endpoints* anidados generan problemas de caché y autorización

¿Cómo diseñar una API?

- El **versionado** evita romper clientes existentes (*legacy*) cuando la API evoluciona.
  1. Si prevés escalabilidad y evolución, considera el versionado desde el principio de forma clara, explícita y estable

```
v1/users
v2/users
```

  2. Puedes considerar versionado semántico (**SemVer**) si es necesario.

El **versionado semántico** es una convención que permite asignar números de versión a un software y que describen claramente qué tipo de cambios o evoluciones se han introducido en la nueva versión.

v1 → v2 Cambio MAJOR que podría afectar a la compatibilidad
v1.1.0 → v1.2.0 Cambio MINOR que añade funcionalidades compatibles
v1.0.0 → v1.0.1 Cambio por resolución de *bug*

SemVer permite a los clientes decidir si evolucionan a la siguiente versión en términos de los cambios realizados.

  3. (Más avanzado) Se puede considerar incluir el versionado en la cabecera.

```
GET /users
Accept: application/vnd.api.v2+json
```

¿Cómo diseñar una API?

¿Hay malas prácticas en versionado?

- No considerar versionado
- Versionar como parámetro de la *query*

```
GET /users?version=2
```

Considera que cada versión implica mantener un tiempo viva la anterior.
Nunca versiones respuestas internas, solo endpoints públicos

Mapeo CRUD - HTTP

Acción	HTTP	Ejemplo
Leer colección *(nunca modifica estado)*	GET	`/users`
Leer elemento *(nunca modifica estado)*	GET	`/users/15`
Crear	POST	`/users`
Reemplazar todo *(sobrescribe)*	PUT	`/users/15`
Modificar parcial	PATCH	`/users/15`
Borrar	DELETE	`/users/15`

```
POST /users/15
GET /users/delete/15
PUT /users
```

POST no debe usarse para modificar recursos
DELETE va en el método, no en la URL
PUT debe apuntar a un recurso individual

Códigos de estado HTTP en APIs

- Indican si la petición tuvo éxito, falló, o si la culpa es del cliente o del servidor.
- Son estándar IETF, por lo que cualquier cliente (web, móvil, IoT) los entiende.
- Sin un uso correcto de los estados, una API REST es confusa e impredecible.
- En APIs REST, se utilizan fundamentalmente 2xx, 4xx y 5xx.

Categoría	Rango	Significado
2xx	200–299	Operación correcta
4xx	400–499	Error del cliente
5xx	500–599	Error del servidor

200 OK → Petición correcta.
201 Created → Recurso creado (POST).
400 Bad Request → Parámetros inválidos.
401 Unauthorized → Falta token o API key.
404 Not Found → Recurso no existe.
500 Internal Server Error → Error servidor.

Códigos de estado HTTP en APIs

200 OK

Se usa cuando:

- GET correcto
- PUT/PATCH correcto
- DELETE correcto (si decides devolver datos)

`POST /users` → **200 OK**
(debería utilizar **201 Created**)

201 Created

Se usa cuando:

- Un nuevo recurso ha sido creado

Características:

- Cabecera `Location` con URL del nuevo recurso
- Cuerpo JSON con recurso ya creado

```
POST /users → 201 Created
Location: /users/87
```

204 No Content

Se usa cuando:

- DELETE correcto
- PUT correcto sin devolver datos
- PATCH opcionalmente sin devolver datos

Características:

- No tiene cuerpo JSON

```
DELETE /users/87
            → 204 No Content
```

Códigos de estado HTTP en APIs

400 Bad Request

Se usa cuando:

- JSON mal formado
- Campos obligatorios ausentes
- Tipos de datos incorrectos
- Validación de entradas fallida

```
{
  "error": "Invalid request",
  "details": "Field 'email'
is required"
}
```

401 Unauthorized

Se usa cuando:

- El cliente no está autenticado
- Falta token o API Key

```
GET /orders
        → 401 (no autenticado)
```

403 Forbidden

Se usa cuando:

- El cliente está autenticado
- ... PERO no tiene permisos

```
GET /admin/users
        → 403 (sin permisos)
```

404 Not Found

Se usa cuando:

- El recurso solicitado NO existe

Precaución:

- Nunca devolver errores dentro de un 200

```
GET /users/999999 → 404
```

200 OK

```
{
  "error": "User not
found"
}
```

Códigos de estado HTTP en APIs

409 Conflict

Se usa cuando:

- Recurso (p.ej. email) ya existe
- Recursos duplicados
- Colisiones en operaciones concurrentes

```
POST /users → 409 Conflict
{
   "error": "Email
already registered"
}
```

422 Unprocessable Entity

Se usa cuando:

- Los datos tienen sentido sintáctico, pero no sentido lógico

```
{
  "error": "Birthdate
cannot be in the future"
}
```

Errores de servidor (5xx):

500 – Internal Server Error

- Error interno inesperado
- Excepción no controlada
- Error en BD o servidor

503 – Service Unavailable

- Caída temporal
- Mantenimiento
- Sobrecarga del sistema

Paginación, filtros y ordenación

- Permiten controlar grandes colecciones, **reducir** la carga del servidor y **facilitar** la búsqueda y listados **eficientes**
- **Paginación**: división de la colección en particiones (navegación incremental)

```
https://pokeapi.co/api/v2/pokemon?limit=5&offset=10
```

```
GET /products?page=3&limit=20
```

  - **`limit`**: cuántos elementos por página. Evita grandes respuestas y mejora tiempos.

```
GET /products?limit=20
```

→ Dame 20 productos como máximo

Paginación, filtros y ordenación

- **Paginación**: división de la colección en particiones (navegación incremental)
  - **`offset`**: desde qué posición de la colección se comienza el recorrido.

```
GET /products?limit=20&offset=0
```
→ (Página 1) Dame los 20 primeros productos como máximo

```
GET /products?limit=20&offset=20
```
→ (Página 2) Productos 21 a 40

Permite moverse por una colección usando “desplazamiento”.

  - ¿Cuándo usar **`limit`** y **`offset`**?
    - Cuando quieres paginación **rígida y clásica**, basada en posiciones.
    - Útil para administradores y listados simples.
    - Es el modelo más fácil de implementar.

Paginación, filtros y ordenación

- En **APIs orientadas al *end-user*** (no tanto a datos), se utiliza:
  - **`page`**: número de página.
  - **`per_page`**: cuántos elementos por página (equivale a **`limit`**)

```
GET /products?page=1&per_page=20
```
→ Página 1 (20 productos por página)

```
GET /products?page=3&per_page=10
```
→ Página 3 (10 elementos por página)

  - ¿Cuándo usar **`page`** y **`per_page`**?
    - Cuando el cliente necesita paginación de estilo “libro” o “UI paginada”.
    - Interfaz más intuitiva que offset/limit.
    - Usada por APIs como GitHub (v3) o muchas APIs de comercio electrónico.
    - Internamente, el servidor las convertirá en **`limit`**/**`offset`**.

Paginación, filtros y ordenación

- El **servidor** debe aceptar los parámetros correspondientes.
- El **cliente** arma la URL con los parámetros de paginación e interpreta la respuesta.
  - El cliente NO implementa la paginación, solo la “dibuja” y la solicita.
  - Es el servidor quien responde con el subconjunto de datos correcto.

```
const url = "/products?page=2&per_page=10";
const res = await fetch(url);
```

```
GET /movies?page=3&per_page=20
```

```
offset = (3 - 1) * 20 = 40
limit = 20
```

Paginación, filtros y ordenación

- **Filtros**: permiten que el cliente solicite solo los elementos que cumplen ciertas condiciones.
- Reducen el tamaño de la respuesta y aceleran las búsquedas

```
GET /products?category=books
GET /users?role=admin
GET /orders?from=2024-01-01&to=2024-12-31
```

```
GET https://restcountries.com/v3.1/region/europe
```

```
GET https://jsonplaceholder.typicode.com/comments?postId=1
```

Paginación, filtros y ordenación

- **Ordenación**: permite al cliente especificar **cómo** quiere que se ordenen los resultados.
- La ordenación NO cambia los datos, solo la forma en que se presentan.

```
GET /products?sort=price&order=asc
GET /users?sort=name&order=desc
```

- Ordenar por **campos sin índice** puede degradar rendimiento.
- La **ordenación múltiple** debe ser coherente (sort=name,-age).
- Los clientes deben **saber si hay orden por defecto**.
- Filtros y ordenación puede ir juntos en la misma *query*.

```
GET /products?category=books&sort=price&order=asc
```

Ejemplo con Postman:
https://shorturl.at/sXyGP

S7-1.
Consumo de APIs con JS

fetch()

- Método nativo moderno para hacer peticiones HTTP.
- Devuelve una **`Promise`**.
- Soporta todos los métodos: GET, POST, PUT, PATCH, DELETE.
- Reemplaza a `XMLHttpRequest (XHR)`.

```
fetch(url, options) → Promise<Response>
```

`options` puede incluir:

- method
- headers
- body
- mode (CORS)
- credentials (cookies, tokens)

La respuesta se procesa con:

- `response.ok`
- `response.status`
- `response.json()`
- `response.text()`

fetch() – Ejemplo de GET

- Método nativo moderno para hacer peticiones HTTP.
- Devuelve una **`Promise`**.
- Soporta todos los métodos: GET, POST, PUT, PATCH, DELETE.
- Reemplaza a `XMLHttpRequest (XHR)`.

```
async function getPokemonCollection() {
  const res = await fetch("https://pokeapi.co/api/v2/pokemon");
  const data = await res.json();
  console.log(data);
}
```

petición HTTP

Convierte el cuerpo a objeto JS

Sin hacer **res.json()** se estaría leyendo un stream sin parsear

```
if (!res.ok) {
  console.error("Error HTTP:", res.status);
}
```

Promises

- Una **`Promise`** es un objeto que representa el **resultado futuro** de una operación asíncrona
- No devuelve el valor ya, sino una "promesa" de que en algún momento:
  - Se resolverá correctamente (*fulfilled*)
  - Fallará (*rejected*)
- Ciclo de vida de una Promise:
  1. *pending* (pendiente) → la operación sigue en curso
  2. *fulfilled* (resuelta) → terminó bien → devuelve un valor
  3. *rejected* (rechazada) → terminó mal → devuelve un error

Una **Promise** es un contrato:

"Te prometo que cuando acabe esta operación asíncrona, te diré si salió bien (*resolve*) o mal (*reject*). Hasta entonces, estoy en estado *pending*."

Promises

`.then()` le dice a JavaScript: “cuando tengas el resultado asíncrono, ejecuta esta función”.

`promesa.then(resultado => { ... })`
Devuelve una nueva Promise (permite encadenación)

```
fetch("https://api.example.com/data")
  .then(response => response.json())
  .then(data => console.log("Datos:", data))
  .catch(error => console.error("Error:", error));
```

```
async function loadData() {
  try {
    const res = await fetch("https://api.example.com/data");
    const data = await res.json();
    console.log("Datos:", data);
  } catch (err) {
    console.error("Error:", err);
  }
}
```

`async/await` es *syntactic sugar* de `fetch/then`

Cadena *query*

- Usar `encodeURIComponent` para evitar errores.
- Construir URLs de forma segura utilizando JS, no cadenas.

```
const name = "bulbasaur";
const url = "https://pokeapi.co/api/v2/pokemon/${encodeURIComponent(name)}";
const res = await fetch(url);
```

```
const url = new URL("https://jsonplaceholder.typicode.com/comments");
url.searchParams.set("postId", 1);
url.searchParams.set("email", "user@example.com");

const res = await fetch(url);
```

POST con JSON

```
async function createUser() {
  const res = await fetch("https://reqres.in/api/users", {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({
      name: "Ada Lovelace",
      job: "Developer"
    })
  });

  const data = await res.json();
  console.log(data);
}
```

PUT y PATCH, DELETE

- PUT reemplaza un recurso completo.
- PATCH modifica solo algunos campos.

```
await fetch("/users/15", {
  method: "PATCH",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({ email: "nuevo@mail.com" })
});
```

- DELETE permite borrar un recurso (“204 No Content”, no tiene JSON de respuesta)

```
await fetch("https://reqres.in/api/users/15", {
  method: "DELETE"
});
```

Validación

Solo los **errores de red** lanzan excepción. Los **errores HTTP** requieren comprobar **`res.ok`**

```
async function getPokemon(id) {
  const url =
`https://pokeapi.co/api/v2/pokemon/${id}`;

  try {
    const res = await fetch(url);

    if (!res.ok) {
      console.error("HTTP error:", res.status);
      return;
    }

    const data = await res.json();
    console.log("Pokémon:", data);

  } catch (err) {
    console.error("Error de red:", err);
  }
}
```

Validación

```
async function addPet() {
  const res = await fetch("https://petstore.swagger.io/v2/pet", {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({
      id: 123,
      name: "Luna",
      status: "available"
    })
  });

  if (!res.ok) {
    console.log("Error:", res.status);
    return;
  }

  const data = await res.json();
  console.log("Mascota creada:", data);
}
```

CORS (*Cross-Origin Resource Sharing*)

- **CORS** es un mecanismo de seguridad del navegador que controla qué orígenes (dominios, puertos o protocolos) pueden acceder a un servidor.
- El navegador bloquea automáticamente peticiones a servidores que no permiten explícitamente el acceso desde tu origen.
- En este caso, el servidor no enviará alguna de las siguientes cabeceras:

```
Access-Control-Allow-Origin: http://miweb.com
Access-Control-Allow-Origin: *
```

- **IMPORTANTE**: CORS no es un error de tu código JS, sino de un servidor bloqueando la petición.

Buenas y malas prácticas en JS

- Comprobar siempre `res.ok`
- Usar `try/catch` para errores de red
- Enviar JSON siempre con cabecera correcta
- Usar `await res.json()` (o `text` si no es JSON)
- Validar datos antes de enviarlos
- Construir URLs con `URL()` para evitar errores
- Nombrar funciones con semántica clara (`fetchUsers`, `createPost`)

- Usar `fetch` sin manejar errores
- Asumir que todas las respuestas vienen en JSON
- Usar GET para operaciones que modifican recursos
- Construir URLs manualmente con *strings* peligrosas
- Ignorar los códigos de estado
- Duplicar código en lugar de utilizar funciones reutilizables

Recursos y lecturas

- Google OAuth API

  https://developers.google.com/identity/protocols/oauth2?hl=es-419
- CORS (Cross-Origin Resource Sharing)

  https://developer.mozilla.org/es/docs/Web/HTTP/Guides/CORS
- Recomendaciones de diseño para API REST

  https://learn.microsoft.com/es-es/azure/architecture/best-practices/api-design
- Guía de diseño de API REST de GitHub

https://microsoft.github.io/code-with-engineering-playbook/design/design-patterns/rest-api-design-guidance/

- Introducción a OpenAPI

  https://learn.openapis.org/
- Uso de Fetch y Promises, async/await

  https://developer.mozilla.org/es/docs/Web/API/Fetch_API/Using_Fetch

  https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/Scripting/Network_requests