APIs REST

Juegos en Red - Grado en Desarrollo de Videojuegos

Ruben Rodríguez
ruben.rodriguez@urjc.es
URJC

Natalia Madrueño
natalia.madrueno@urjc.es
URJC

2025-09-09

Universidad Rey Juan Carlos

DSLAB
DATA SCIENCE LABORATORY

Tabla de contenidos

Introducción a APIs REST

Contexto en Juegos Multijugador

Después de implementar el juego con Phaser 3 y el Patrón Command

- Cliente y servidor necesitan comunicarse
- Gestionar registro y autenticación de jugadores
- Almacenar puntuaciones y estadísticas
- Gestionar partidas y salas de juego
- Consultar rankings y perfiles

> **Nota**
>
> REST para operaciones que no requieren actualización instantánea

Comunicación HTTP Tradicional vs API REST

Aplicaciones web tradicionales:

- Peticiones HTTP devuelven documentos HTML
- El navegador renderiza la página completa

Aplicaciones con AJAX y SPA:

- Peticiones HTTP intercambian información estructurada
- No devuelven HTML, sino datos (típicamente JSON)
- El cliente procesa y actualiza la interfaz

```
Petición:
GET http://www.mygame.com/players/alice

Respuesta:
{
  "id": "alice",
  "name": "Alice Smith",
  "level": 15,
  "score": 8500
}
```

¿Qué es una API REST?

REST (Representational State Transfer)

- Estilo de arquitectura de software basado en HTTP
- Acuñado en 2000 por **Roy Fielding** (coautor de HTTP)
- Cuando un servicio cumple estos principios es **RESTful**

Características clave:

- Operaciones CRUD sobre recursos del servidor
- Aprovecha URL, métodos HTTP, códigos de estado
- Intercambio de información en formato JSON
- Simplicidad y eficiencia

> **Tip**
>
> REST se ha convertido en el estándar de facto frente a otras alternativas como SOAP

Niveles de Madurez REST

Modelo de Richardson

4 niveles según conformidad con principios REST

Nivel 0 - The Swamp of POX

Plain Old XML

- HTTP solo como sistema de transporte
- Todas las peticiones POST a una única URL
- El cuerpo indica qué operación realizar
- No aprovecha características de HTTP

```
POST /api HTTP/1.1
Body: <operation>getPlayer</operation><id>alice</id>
```

> ⚠ **Advertencia**
>
> Similar a usar HTTP como túnel para otros protocolos

Nivel 1 - Resources

Introducción de recursos individuales

- Cada recurso tiene su propia URI específica
- `/players/alice` y `/matches/123` son URIs distintas
- Todavía no usa correctamente los métodos HTTP
- Típicamente solo POST para todas las operaciones

```
POST /players/alice HTTP/1.1
POST /matches/123 HTTP/1.1
```

Nivel 2 - HTTP Verbs

Uso correcto de métodos y códigos HTTP

- GET para obtener, POST para crear
- PUT para actualizar, DELETE para eliminar
- Códigos de estado apropiados (200, 404, 500, etc.)
- Aprovecha la semántica del protocolo HTTP

> **Importante**
>
> **Este es el nivel que usaremos en este curso**

```
GET /players/alice HTTP/1.1      → 200 OK
POST /matches HTTP/1.1            → 201 Created
DELETE /players/bob HTTP/1.1      → 204 No Content
```

Nivel 3 - HATEOAS

Hypermedia as the Engine of Application State

- Las respuestas incluyen enlaces hipermedia
- Guían al cliente sobre acciones disponibles
- Los clientes descubren dinámicamente la API

```json
{
  "id": "alice",
  "name": "Alice Smith",
  "links": {
    "matches": "/players/alice/matches",
    "friends": "/players/alice/friends"
  }
}
```

Nota

Pocas APIs implementan HATEOAS en la práctica debido a su complejidad

Formato JSON

JavaScript Object Notation

Formato estándar para intercambio de datos

- Ligero y fácil de leer para humanos
- Fácil de parsear para máquinas
- Basado en JavaScript pero independiente del lenguaje

Estructuras principales:

Objetos:

- Colección de pares clave-valor
- Encerrados en `{}`

```
{
  "nombre": "Alice",
  "nivel": 15
}
```

Arrays:

- Lista ordenada de valores
- Encerrados en `[]`

```
[
  "item1",
  "item2",
  "item3"
]
```

Tipos de Valores JSON

Los valores pueden ser:

- **Cadenas de texto:** `"Hola mundo"`
- **Números:** `42`, `3.14`, `-10`
- **Booleanos:** `true`, `false`
- **Null:** `null`
- **Objetos:** `{ "key": "value" }`
- **Arrays:** `[1, 2, 3]`

Ejemplo Completo: Estado de Partida

```json
{
  "game": {
    "id": "match_12345",
    "type": "pong",
    "status": "active",
    "players": [
      {
        "id": "alice",
        "name": "Alice Smith",
        "score": 5,
        "ready": true
      },
      {
        "id": "bob",
        "name": "Bob Johnson",
        "score": 3,
        "ready": true
      }
    ],
    "settings": {
      "max_score": 11,
      "ball_speed": 300
    }
  }
}
```

Usos de JSON

Más allá de APIs REST:

- Ficheros de configuración
- Almacenamiento de datos en disco
- Bases de datos NoSQL (MongoDB)
- Comunicación entre servicios
- WebSockets y tiempo real

Principios de Diseño REST

Todo es un Recurso

Recurso: ítem de información identificado por URI única

```
http://api.game.com/players/alice
http://api.game.com/matches/12345
http://api.game.com/players/alice/scores
http://api.game.com/leaderboard
```

Estructura de URI:

- Parte fija: dominio y ruta base
- Parte variable: identifica el recurso específico

Principios de Diseño de URIs

Reglas de buenas prácticas:

Hacer:

- Usar sustantivos, no verbos
  - `/players/alice` ✓
- Usar plurales para colecciones
  - `/players` para lista
  - `/players/alice` para uno
- Crear jerarquías lógicas
  - `/players/alice/matches`

Evitar:

- Verbos en la URI
  - `/getPlayer?id=alice` ✗
- Nombres inconsistentes
  - Mezclar singular/plural
- URIs planas sin relación
- Mayúsculas y guiones bajos

Tip

Usar minúsculas y guiones: `/game-sessions/active`

Métodos HTTP

Los Cuatro Métodos Principales

Operaciones CRUD sobre recursos

Método	Operación	Seguro	Idempotente	Cacheable
GET	Obtener	✓	✓	✓
POST	Crear	✗	✗	✗
PUT	Actualizar	✗	✓	✗
DELETE	Eliminar	✗	✓	✗

Nota

Seguro: no modifica el servidor **Idempotente:** múltiples peticiones = mismo resultado

GET - Obtener Información

Características:

- Obtiene información sin modificar el servidor
- Seguro e idempotente
- Cacheable por navegadores y proxies

Ejemplos:

```
GET /players/alice HTTP/1.1
→ Devuelve información del jugador alice

GET /matches HTTP/1.1
→ Devuelve lista de todas las partidas

GET /leaderboard?limit=10 HTTP/1.1
→ Devuelve top 10 del ranking
```

POST - Crear Recursos

Características:

- Crea nuevos recursos en el servidor
- El servidor decide el ID del recurso
- Devuelve el recurso creado en la respuesta
- No es seguro ni idempotente

Ejemplo:

```
POST /matches HTTP/1.1
Content-Type: application/json

{
  "type": "pong",
  "mode": "ranked",
  "player_id": "alice"
}

→ 201 Created
Location: /matches/67890
```

PUT - Actualizar Recursos

Características:

- Actualiza un recurso existente
- Envía el recurso completo (reemplazo)
- No es seguro pero sí idempotente
- Múltiples PUT producen el mismo resultado

Ejemplo:

```
PUT /players/alice HTTP/1.1
Content-Type: application/json

{
  "name": "Alice Smith",
  "level": 16,
  "score": 9000
}

→ 200 OK
```

Nota

PATCH se usa para actualizaciones parciales (no cubierto aquí)

DELETE - Eliminar Recursos

Características:

- Elimina un recurso del servidor
- No es seguro pero sí idempotente
- Eliminar varias veces = mismo resultado (no existe)
- Responde con 204 (sin contenido) o 200

Ejemplo:

```
DELETE /matches/12345 HTTP/1.1

→ 204 No Content
```

Anatomía de Peticiones y Respuestas

Petición HTTP REST Completa

```
POST /matches HTTP/1.1
Host: api.game.com
Content-Type: application/json
Accept: application/json
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
Content-Length: 89

{
  "type": "pong",
  "mode": "ranked",
  "max_score": 11,
  "player_id": "alice"
}
```

Tres partes:

1. **Línea de petición:** Método, ruta y versión HTTP
2. **Headers:** Metadatos (tipo contenido, auth, etc.)
3. **Body:** Datos JSON enviados

Headers Comunes en Peticiones

Cabeceras importantes:

- `Content-Type: application/json`
  - Indica que enviamos JSON
- `Accept: application/json`
  - Indica que queremos recibir JSON
- `Authorization: Bearer <token>`
  - Token de autenticación
- `Content-Length: 89`
  - Tamaño del cuerpo en bytes

Respuesta HTTP REST Completa

```
HTTP/1.1 201 Created
Content-Type: application/json
Location: http://api.game.com/matches/67890
Content-Length: 156

{
  "id": "67890",
  "type": "pong",
  "mode": "ranked",
  "max_score": 11,
  "status": "waiting",
  "created_at": "2025-01-15T14:30:00Z",
  "players": ["alice"]
}
```

Tres partes:

1. **Línea de estado:** Versión HTTP, código y mensaje
2. **Headers:** Metadatos de la respuesta
3. **Body:** Datos JSON del recurso

Headers Comunes en Respuestas

Cabeceras importantes:

- `Content-Type: application/json`
  - La respuesta es JSON
- `Location: http://api.game.com/matches/67890`
  - URI del recurso creado (en POST)
- `Content-Length: 156`
  - Tamaño de la respuesta en bytes

Códigos de Estado HTTP

Categorías de Códigos

Números de tres dígitos que comunican el resultado

Categoría	Significado	Uso
1xx	Informativas	Raramente usados
2xx	Éxito	Operación exitosa
3xx	Redirecciones	Acciones adicionales
4xx	Error del cliente	Error en la petición
5xx	Error del servidor	Fallo del servidor

1xx - Respuestas Informativas

Indican que la petición fue recibida y continúa

- Raramente usados en REST
- Más relevante:
  - **101 Switching Protocols**
    - Cambio de HTTP a WebSockets

> **Nota**
>
> Veremos el código 101 cuando estudiemos WebSockets

2xx - Respuestas Exitosas

La petición fue procesada correctamente

- **200 OK**
  - Éxito general en GET, PUT
  - Respuesta con contenido
- **201 Created**
  - Recurso creado tras POST
  - Debe incluir header `Location`
- **204 No Content**
  - Éxito sin contenido
  - Típico en DELETE

3xx - Redirecciones

El cliente debe tomar acciones adicionales

- **301 Moved Permanently**
  - Recurso movido permanentemente
  - Nueva ubicación en header `Location`
- **304 Not Modified**
  - Usado con caché
  - El recurso no ha cambiado desde última petición

> **Tip**
>
> Las redirecciones ayudan a mantener APIs retrocompatibles

4xx - Errores del Cliente

Error en la petición del cliente

- **400 Bad Request:** Petición mal formada
- **401 Unauthorized:** Autenticación requerida
- **403 Forbidden:** Sin permisos (aunque esté autenticado)
- **404 Not Found:** Recurso no existe
- **409 Conflict:** Conflicto con estado actual (ej: usuario ya existe)
- **422 Unprocessable Entity:** Formato correcto pero errores de validación

5xx - Errores del Servidor

El servidor no pudo completar la petición

- **500 Internal Server Error**
  - Error genérico del servidor
  - Algo falló en el procesamiento
- **503 Service Unavailable**
  - Servidor temporalmente no disponible
  - Debe incluir `Retry-After`
- **504 Gateway Timeout**
  - Timeout esperando respuesta de otro servidor

> ⚠ **Advertencia**
>
> Los errores 5xx indican problemas del servidor, no del cliente

Cliente JavaScript con fetch()

La API fetch()

API estándar moderna para peticiones HTTP

- Integrada en todos los navegadores modernos
- Utiliza Promises para operaciones asincrónicas
- Sintaxis limpia y potente

Dos formas de manejar respuestas:

Callbacks con .then()

```
fetch(url)
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

async/await (recomendado)

```
async function getData() {
  const response = await fetch(url);
  const data = await response.json();
  console.log(data);
}
```

async/await: Sintaxis Recomendada

Código más legible y natural

```
async function getPlayer() {
    try {
        const response = await fetch('https://api.game.com/players/alice');

        if (!response.ok) {
            throw new Error('HTTP error ' + response.status);
        }

        const data = await response.json();
        console.log(data);

    } catch (error) {
        console.error('Error:', error);
    }
}
```

Tip

Siempre usar try/catch para manejar errores

GET - Obtener Datos

Método por defecto de fetch()

```
// Obtener un jugador específico
async function getPlayer(id) {
    const response = await fetch(`https://api.game.com/players/${id}`);
    return await response.json();
}

// Uso
const player = await getPlayer('alice');
console.log(player.name);
console.log(player.level);
```

Nota

Template literals (`) permiten insertar variables en strings

POST - Crear Recurso

Especificar método, headers y body

```javascript
async function createPlayer(data) {
    const response = await fetch('https://api.game.com/players', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify(data)
    });

    return await response.json();
}

// Uso
const newPlayer = await createPlayer({
    username: 'charlie',
    email: 'charlie@example.com'
});
console.log('Creado con ID:', newPlayer.id);
```

PUT - Actualizar Recurso

Reemplazar completamente el recurso

```
async function updatePlayer(id, updates) {
    const response = await fetch(`https://api.game.com/players/${id}`, {
        method: 'PUT',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify(updates)
    });

    return await response.json();
}

// Uso
await updatePlayer('charlie', {
    email: 'newemail@example.com',
    level: 5
});
```

Nota

PUT reemplaza todo el recurso; PATCH actualiza solo campos específicos

DELETE - Eliminar Recurso

Método más simple

```
async function deletePlayer(id) {
    const response = await fetch(`https://api.game.com/players/${id}`, {
        method: 'DELETE'
    });

    return response.ok;
}

// Uso
const eliminado = await deletePlayer('charlie');
if (eliminado) {
    console.log('Jugador eliminado correctamente');
}
```

Tip

`response.ok` es `true` para códigos 2xx

Manejo de Errores y Códigos de Estado

Manejo robusto con switch

```javascript
async function fetchWithErrorHandling(url) {
    try {
        const response = await fetch(url);

        switch (response.status) {
            case 200:
                return await response.json();
            case 401:
                throw new Error('No autenticado');
            case 403:
                throw new Error('Sin permisos');
            case 404:
                throw new Error('Recurso no encontrado');
            case 500:
                throw new Error('Error del servidor');
            default:
                throw new Error(`Error: ${response.status}`);
        }
    } catch (error) {
        console.error('Error en petición:', error);
        throw error;
    }
}
```

Servidor con Node.js y Express

Express.js

Framework web minimalista para Node.js

Características:

- Sistema de enrutamiento robusto
- Soporte para middlewares
- Manejo de errores integrado
- Compatibilidad con todos los métodos HTTP
- Estándar de la industria

Instalación:

```
npm init -y
npm install express
```

Configuración de Proyecto

Habilitar módulos ES6:

```
{
  "name": "mi-api-juego",
  "version": "1.0.0",
  "type": "module",
  "dependencies": {
    "express": "^4.18.0"
  }
}
```

Importante

`"type": "module"` permite usar `import` en lugar de `require`

Estructura de Proyecto

```
mi-api-juego/
├── node_modules/        # Dependencias (no subir a git)
├── src/
│   ├── controllers/     # Lógica de negocio
│   ├── routes/          # Definición de rutas
│   └── app.js           # Punto de entrada
├── package.json
└── package-lock.json
```

Servidor Básico con Express

```
1  import express from 'express';
2
3  const app = express();
4
5  // Middleware para parsear JSON
6  app.use(express.json());
7
8  // Iniciar servidor
9  app.listen(8080, () => {
10   console.log('Servidor ejecutándose en http://localhost:8080');
11 });
```

Importante

`express.json()` es fundamental para leer el body de las peticiones

Ejecutar:

```
1  node src/app.js
```

Controladores

Funciones con lógica de negocio

- Procesan las peticiones HTTP
- Manipulan datos recibidos
- Devuelven respuestas apropiadas
- Separan lógica del enrutamiento

Patrón con closures para estado privado:

```
1  const createAnunciosController = () => {
2    // Estado privado
3    const anuncios = [];
4    let nextId = 1;
5
6    const getAll = (req, res) => {
7      res.json(anuncios);
8    };
9
10   return { getAll };
11 };
12
13 export default createAnunciosController;
```

Controlador Completo - Ejemplo

```
// src/controllers/anunciosController.js
const createAnunciosController = () => {
  const anuncios = [];
  let nextId = 1;

  const getAll = (req, res) => {
    res.json(anuncios);
  };

  const create = (req, res) => {
    const { nombre, asunto, comentario } = req.body;

    const nuevoAnuncio = {
      id: nextId++,
      nombre,
      asunto,
      comentario
    };

    anuncios.push(nuevoAnuncio);
    res.status(201).json(nuevoAnuncio);
  };

  return { getAll, create };
};
```

Rutas

Conectan URLs con controladores

```
1  // src/routes/anunciosRoutes.js
2  import express from 'express';
3  import createAnunciosController from '../controllers/anunciosController.js';
4
5  const router = express.Router();
6  const controller = createAnunciosController();
7
8  router.get('/', controller.getAll);
9  router.post('/', controller.create);
10
11 export default router;
```

Nota

`router.get('/')` se convertirá en `GET /anuncios` con el prefijo

Integrar Rutas en la Aplicación

```
// src/app.js
import express from 'express';
import anunciosRoutes from './routes/anunciosRoutes.js';

const app = express();
app.use(express.json());

// Registrar rutas con prefijo
app.use('/anuncios', anunciosRoutes);

app.listen(8080);
```

Resultado:

- `GET /anuncios` → obtener todos
- `POST /anuncios` → crear nuevo

Operación GET - Recurso Específico

```
app.get('/anuncios/:id', (req, res) => {
  const { id } = req.params;
  const anuncio = anuncios.find(a => a.id === parseInt(id));

  if (!anuncio) {
    return res.status(404).json({ error: 'No encontrado' });
  }

  res.json(anuncio);
});
```

Tip

`:id` es un parámetro dinámico disponible en `req.params.id`

Operación POST - Validación

```
app.post('/anuncios', (req, res) => {
  const { nombre, asunto, comentario } = req.body;

  // Validar datos
  if (!nombre || !asunto) {
    return res.status(400).json({
      error: 'Nombre y asunto son requeridos'
    });
  }

  const nuevoAnuncio = {
    id: Date.now(),
    nombre,
    asunto,
    comentario
  };

  anuncios.push(nuevoAnuncio);
  res.status(201).json(nuevoAnuncio);
});
```

Operación PUT

```
app.put('/anuncios/:id', (req, res) => {
  const { id } = req.params;
  const { nombre, asunto, comentario } = req.body;

  const anuncio = anuncios.find(a => a.id === parseInt(id));

  if (!anuncio) {
    return res.status(404).json({ error: 'No encontrado' });
  }

  // Actualizar propiedades
  anuncio.nombre = nombre;
  anuncio.asunto = asunto;
  anuncio.comentario = comentario;

  res.json(anuncio);
});
```

Operación DELETE

```
app.delete('/anuncios/:id', (req, res) => {
  const { id } = req.params;
  const index = anuncios.findIndex(a => a.id === parseInt(id));

  if (index === -1) {
    return res.status(404).json({ error: 'No encontrado' });
  }

  const anuncioEliminado = anuncios.splice(index, 1)[0];
  res.json(anuncioEliminado);
});
```

Nota

También se puede devolver 204 (No Content) sin body

Middlewares

Funciones que procesan peticiones antes del controlador

Usos comunes:

- Logging de peticiones
- Autenticación y autorización
- Validación de datos
- Manejo de errores

Firma:

```
(req, res, next) => {
  // Procesar
  next();  // Pasar al siguiente middleware/ruta
}
```

Middleware de Logging

```
app.use((req, res, next) => {
  console.log(`${req.method} ${req.path}`);
  next();  // Importante: pasar control
});
```

⚠ Advertencia

Olvidar llamar a `next()` bloquea la petición

Middleware de Manejo de Errores

Firma especial con 4 parámetros

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).json({
    error: 'Error interno del servidor'
  });
});
```

Importante

Debe colocarse **después** de todas las rutas

Servir Archivos Estáticos

Express puede servir HTML, CSS, JS del cliente

```
1  app.use(express.static('public'));
```

Estructura:

```
mi-api-juego/
├── public/
│   ├── index.html
│   ├── script.js
│   └── style.css
└── src/
    └── app.js
```

Acceso:

- http://localhost:8080/index.html
- http://localhost:8080/script.js

Ventajas de la Arquitectura REST

Separación de Responsabilidades

Componentes independientes:

API REST:

- Operaciones de gestión
- Registro y autenticación
- Puntuaciones y estadísticas
- Perfiles y rankings

Sistema de Comandos:

- Lógica del juego
- Acciones de jugadores
- Estado del juego

> **Tip**
>
> Cada componente puede evolucionar independientemente

Escalabilidad

Escalar servicios según necesidades

- Operaciones de gestión (REST) tienen diferentes cargas
- Operaciones de tiempo real (WebSockets) requieren más recursos
- Podemos escalar cada servicio independientemente
- Optimización específica para cada tipo de comunicación

Testabilidad

Pruebas independientes:

- Probar API REST sin necesidad del juego completo
- Probar sistema de comandos sin conexión a red
- Herramientas específicas para cada tipo de prueba
- Tests automatizados más simples

Flexibilidad

La misma API sirve a múltiples clientes:

- Juego web (navegador)
- Aplicación móvil (iOS/Android)
- Herramientas de administración
- Servicios backend

> **Nota**
>
> Arquitectura preparada para futuras expansiones

Resumen

Conceptos Clave

API REST:

- Estilo arquitectónico basado en HTTP
- Operaciones CRUD sobre recursos
- URIs bien diseñadas (sustantivos, jerarquías)
- Métodos HTTP correctos (GET, POST, PUT, DELETE)
- Códigos de estado apropiados (2xx, 4xx, 5xx)

JSON:

- Formato estándar de intercambio
- Ligero y legible
- Objetos y arrays anidados

Implementación

Cliente JavaScript:

- API `fetch()` con async/await
- Manejo de errores robusto
- Validación de códigos de estado

Servidor Node.js:

- Express.js como framework
- Controladores con lógica de negocio
- Rutas para organizar endpoints
- Middlewares para funcionalidad transversal