📚 Introducción Teórica: APIs RESTful y Relaciones 1 a Muchos

Autor: Dilan Real
Fecha: 12 de diciembre de 2025
Curso: Desarrollo Web - 6to Semestre
Institución: Escuela Politécnica Nacional

🌐 ¿Qué es el estándar RESTful?

Definición

REST (Representational State Transfer) es un estilo arquitectónico para diseñar servicios web y APIs que fue introducido por Roy Fielding en su tesis doctoral en el año 2000. No es un protocolo ni un estándar, sino un conjunto de principios y restricciones que, cuando se siguen, resultan en sistemas web escalables, flexibles y fáciles de mantener.

Una **API RESTful** es una interfaz de programación de aplicaciones que implementa la arquitectura REST, permitiendo la comunicación entre diferentes sistemas a través del protocolo HTTP de manera eficiente y estandarizada.

🎯 Principios Fundamentales de REST

1. Arquitectura Cliente-Servidor

¿Qué significa?

REST separa la interfaz de usuario (cliente) de la lógica de negocio y almacenamiento de datos (servidor). Esta separación permite que:

- **El cliente** se preocupe únicamente de la presentación de datos y experiencia del usuario
- **El servidor** se enfoque en procesar lógica de negocio y gestionar datos
- Ambos puedan evolucionar independientemente

Ejemplo práctico:

```
Cliente (React App) ↔ HTTP ↔ Servidor (Node.js API) ↔ Base de Datos
```

Ventajas:

- ✅ Portabilidad: El mismo servidor puede atender múltiples clientes (web, móvil, desktop)
- ✅ Escalabilidad: Cliente y servidor pueden escalar independientemente
- ✅ Flexibilidad: Cambios en el frontend no afectan al backend y viceversa

---

2. Stateless (Sin Estado)

¿Qué significa?

Cada petición del cliente al servidor debe contener **toda la información necesaria** para entender y procesar la petición. El servidor no guarda ningún contexto del cliente entre peticiones.

Sin REST (Stateful):

```
Cliente: "Hola, soy Juan" (Login)
Servidor: "Ok, te recuerdo" (Guarda sesión)

Cliente: "Dame mis pedidos"
Servidor: "Como sé que eres Juan, aquí están" (Usa sesión guardada)
```

Con REST (Stateless):

```
Cliente: "Dame mis pedidos, aquí está mi token JWT"
Servidor: "Verifico el token, aquí están tus pedidos"

Cliente: "Dame mi perfil, aquí está mi token JWT"
Servidor: "Verifico el token, aquí está tu perfil"
```

Ventajas:

- ✅ Escalabilidad horizontal: Cualquier servidor puede atender cualquier petición
- ✅ Simplicidad: El servidor no necesita mantener estado
- ✅ Confiabilidad: Menos complejidad = menos errores
- ✅ Cache: Las respuestas son más fáciles de cachear

Implementación práctica:

```
GET /api/dishes/1
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
```

Cada petición incluye el token de autenticación.

---

3. Interfaz Uniforme

¿Qué significa?

REST define una interfaz consistente y predecible que se aplica a todos los recursos del sistema. Esta uniformidad se logra mediante:

a) Identificación de recursos mediante URLs

Cada recurso debe tener un identificador único (URI).

Ejemplos:

```
https://api.ejemplo.com/restaurants/1
https://api.ejemplo.com/dishes/25
https://api.ejemplo.com/users/admin@ejemplo.com
```

Convenciones:

- ✅ Usar sustantivos en plural: `/restaurants`, `/dishes`
- ✅ Usar IDs para recursos específicos: `/restaurants/1`
- ✅ Usar jerarquías para relaciones: `/restaurants/1/dishes`
- ❌ Evitar verbos en URLs: `/getRestaurant`, `/createDish`

b) Manipulación de recursos mediante representaciones

Los clientes interactúan con representaciones de recursos (usualmente JSON), no con los recursos directamente.

Ejemplo:

```
// Representación de un restaurante
{
  "id": 1,
  "nombre": "La Casa del Marisco",
  "ciudad": "Quito"
}
```

El cliente recibe esta representación y puede modificarla localmente. Para guardar cambios, envía la representación modificada de vuelta al servidor.

c) Mensajes auto-descriptivos

Cada mensaje HTTP incluye suficiente información para describir cómo procesarlo:

```
POST /api/restaurants HTTP/1.1
Host: api.ejemplo.com
Content-Type: application/json
Authorization: Bearer token123

{
  "nombre": "Restaurante Nuevo",
  "ciudad": "Guayaquil"
}
```

Elementos auto-descriptivos:

- `POST`: Indica que se va a crear un recurso
- `Content-Type: application/json`: Indica el formato del body
- `Authorization`: Indica cómo autenticar la petición

---

4. Sistema de Capas

REST permite una arquitectura en capas donde:

```
Cliente → Proxy/Load Balancer → API Gateway → Servidor → Base de Datos
```

El cliente no necesita saber si está hablando directamente con el servidor o con un intermediario (proxy, cache, load balancer).

Ventajas:

- ✅ Seguridad: Capas de firewall y autenticación
- ✅ Escalabilidad: Load balancers distribuyen carga
- ✅ Performance: Caches mejoran velocidad
- ✅ Flexibilidad: Cambiar infraestructura sin afectar clientes

---

5. Cacheable

Las respuestas del servidor deben indicar explícitamente si pueden ser cacheadas o no:

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: max-age=3600

{
  "id": 1,
  "nombre": "La Casa del Marisco"
}
```

Ventajas:

- ✅ Reduce latencia para el cliente
- ✅ Reduce carga en el servidor
- ✅ Mejora escalabilidad

🔧 Métodos HTTP en RESTful APIs

REST utiliza los métodos HTTP estándar para realizar operaciones sobre recursos. Cada método tiene una semántica específica:

GET - Leer/Consultar

Propósito: Obtener información de un recurso sin modificarlo.

Características:

- ✅ **Seguro**: No modifica el estado del servidor
- ✅ **Idempotente**: Llamarlo múltiples veces produce el mismo resultado
- ✅ **Cacheable**: Las respuestas pueden ser cacheadas

Ejemplos:

```
GET /restaurants
# Obtener todos los restaurantes

GET /restaurants/1
# Obtener restaurante con ID 1

GET /restaurants/1/dishes
# Obtener platos del restaurante 1
```

Respuesta exitosa:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "id": 1,
  "nombre": "La Casa del Marisco",
  "ciudad": "Quito"
}
```

POST - Crear

Propósito: Crear un nuevo recurso en el servidor.

Características:

- ❌ **No seguro**: Modifica el estado del servidor
- ❌ **No idempotente**: Llamarlo múltiples veces crea múltiples recursos
- ❌ **No cacheable**: No se debe cachear

Ejemplo:

```
POST /restaurants
Content-Type: application/json
```

```
{
  "nombre": "Restaurante Nuevo",
  "ciudad": "Guayaquil",
  "tipoCocina": "Italiana"
}
```

Respuesta exitosa:

```
HTTP/1.1 201 Created
Location: /restaurants/25
Content-Type: application/json

{
  "id": 25,
  "nombre": "Restaurante Nuevo",
  "ciudad": "Guayaquil",
  "tipoCocina": "Italiana"
}
```

Códigos de respuesta comunes:

- `201 Created`: Recurso creado exitosamente
- `400 Bad Request`: Datos inválidos
- `409 Conflict`: El recurso ya existe

PUT - Actualizar Completo

Propósito: Reemplazar completamente un recurso existente.

Características:

- ❌ **No seguro**: Modifica el estado del servidor
- ✅ **Idempotente**: Llamarlo múltiples veces con los mismos datos produce el mismo resultado
- ❌ **No cacheable**: No se debe cachear

Diferencia clave con PATCH: PUT reemplaza **TODO** el recurso. Debes enviar todos los campos, incluso los que no cambian.

Ejemplo:

```
PUT /restaurants/1
Content-Type: application/json

{
  "id": 1,
  "nombre": "La Casa del Marisco Premium",
  "direccion": "Av. Amazonas N24-03",
  "ciudad": "Quito",
```

```
  "tipoCocina": "Mariscos",
  "telefono": "02-234-5678"
}
```

Respuesta:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "id": 1,
  "nombre": "La Casa del Marisco Premium",
  ...
}
```

PATCH - Actualizar Parcial

Propósito: Modificar parcialmente un recurso existente.

Características:

- ❌ **No seguro**: Modifica el estado del servidor
- ✅ **Idempotente**: En la mayoría de casos
- ❌ **No cacheable**: No se debe cachear

Diferencia con PUT: PATCH modifica **solo los campos enviados**, no reemplaza todo el recurso.

Ejemplo:

```
PATCH /restaurants/1
Content-Type: application/json

{
  "telefono": "02-999-8888"
}
```

Solo actualiza el teléfono, el resto de campos permanecen igual.

Cuándo usar PUT vs PATCH:

Escenario	Método
Actualizar todos los campos del recurso	PUT
Cambiar solo el nombre	PATCH
Actualizar dirección y teléfono	PATCH

Escenario	Método
Reemplazar completamente un restaurante	PUT

DELETE - Eliminar

Propósito: Eliminar un recurso del servidor.

Características:

- ❌ **No seguro**: Modifica el estado del servidor
- ✅ **Idempotente**: Eliminar un recurso ya eliminado no cambia nada
- ❌ **No cacheable**: No se debe cachear

Ejemplo:

```
DELETE /restaurants/1
```

Respuesta:

```
HTTP/1.1 204 No Content
```

Códigos de respuesta comunes:

- `204 No Content`: Eliminado exitosamente (sin cuerpo de respuesta)
- `200 OK`: Eliminado exitosamente (con confirmación en el cuerpo)
- `404 Not Found`: El recurso no existe
- `409 Conflict`: No se puede eliminar (tiene dependencias)

🔗 Recursos y URLs

¿Qué es un Recurso?

En REST, un **recurso** es cualquier cosa que pueda ser nombrada y manipulada:

- Un objeto (restaurante, plato, usuario)
- Una colección (lista de restaurantes)
- Una relación (platos de un restaurante)
- Un servicio (cálculo de precio con descuento)

Diseño de URLs RESTful

Principios:

1. **Usar sustantivos, no verbos**

✅ **Correcto:**

```
GET /restaurants
POST /restaurants
DELETE /restaurants/1
```

❌ **Incorrecto:**

```
GET /getRestaurants
POST /createRestaurant
GET /deleteRestaurant?id=1
```

2. **Usar plurales para colecciones**

✅ **Correcto:**

```
GET /restaurants      # Colección
GET /restaurants/1    # Elemento específico
```

❌ **Incorrecto:**

```
GET /restaurant       # Singular para colección
```

3. **Usar jerarquías para relaciones**

✅ **Correcto:**

```
GET /restaurants/1/dishes        # Platos del restaurante 1
GET /authors/5/books             # Libros del autor 5
GET /universities/2/students     # Estudiantes de la universidad 2
```

4. **Usar guiones para legibilidad**

✅ **Correcto:**

```
GET /user-profiles
GET /order-items
```

❌ **Incorrecto:**

```
GET /user_profiles   # Guión bajo
GET /userProfiles    # CamelCase
```

5. **Usar minúsculas**

✅ **Correcto:**

```
GET /restaurants/1/dishes
```

❌ **Incorrecto:**

```
GET /Restaurants/1/Dishes
```

Ejemplos de URLs bien diseñadas:

```
# Colecciones
GET /restaurants
GET /dishes
GET /users

# Elementos específicos
GET /restaurants/1
GET /dishes/25
GET /users/admin

# Relaciones
GET /restaurants/1/dishes
GET /universities/3/students
GET /teams/5/players

# Filtros (query parameters)
GET /dishes?categoria=Entrada
GET /restaurants?ciudad=Quito&tipoCocina=Italiana
GET /users?activo=true&rol=admin

# Ordenamiento
GET /dishes?ordenarPor=precio&orden=asc

# Paginación
GET /restaurants?pagina=2&limite=10
```

📄 Formatos de Datos

JSON - El estándar de facto

¿Por qué JSON?

- ✅ Fácil de leer para humanos
- ✅ Fácil de parsear para máquinas
- ✅ Soportado nativamente en JavaScript
- ✅ Ligero (menos bytes que XML)
- ✅ Tipos de datos claros (strings, números, booleanos, arrays, objetos)

Estructura JSON para un restaurante:

```
{
  "id": 1,
  "nombre": "La Casa del Marisco",
  "direccion": "Av. Amazonas N24-03",
  "ciudad": "Quito",
  "tipoCocina": "Mariscos",
  "telefono": "02-234-5678",
  "platos": [
    {
      "id": 1,
      "nombre": "Ceviche de Camarón",
      "precio": 12.50,
      "disponible": true
    },
    {
      "id": 2,
      "nombre": "Arroz Marinero",
      "precio": 18.00,
      "disponible": true
    }
  ]
}
```

Content-Type Headers

Siempre especificar el formato:

```
Content-Type: application/json
Accept: application/json
```

🔄 Relaciones en APIs RESTful

Tipos de Relaciones

1. Uno a Uno (1:1)

Ejemplo: Usuario → Perfil

```
Usuario (id=1) ↔ Perfil (usuario_id=1)
```

Endpoints:

```
GET /users/1
GET /users/1/profile
```

2. Uno a Muchos (1:N) ☆

Ejemplo: Restaurante → Platos

```
Restaurante (id=1) ↔ Plato (restaurante_id=1)
                   ↔ Plato (restaurante_id=1)
                   ↔ Plato (restaurante_id=1)
```

Endpoints:

```
GET /restaurants/1
GET /restaurants/1/dishes
GET /dishes/1
```

Características:

- El lado "uno" (restaurante) no referencia a los "muchos"
- El lado "muchos" (platos) referencia al "uno" mediante Foreign Key
- Endpoint especial para obtener los "muchos": `/restaurants/{id}/dishes`

3. Muchos a Muchos (N:N)

Ejemplo: Estudiantes ↔ Cursos

```
Estudiante ↔ Inscripción ↔ Curso
```

Endpoints:

```
GET /students/1/courses
GET /courses/5/students
```

📊 Ejemplo de Relación 1 a Muchos

Caso General: Equipo de Fútbol y Jugadores

Descripción:

- Un equipo de fútbol tiene muchos jugadores
- Cada jugador pertenece a un solo equipo

Modelo de datos:

```
Equipo:
- id: integer
- nombre: string
- pais: string

Jugador:
- id: integer
- nombre: string
- posicion: string
- equipo_id: integer (FK → Equipo.id)
```

Relación:

```
Equipo "Barcelona FC" (id=1)
    ├─→ Jugador "Lionel Messi" (equipo_id=1)
    ├─→ Jugador "Gerard Piqué" (equipo_id=1)
    └─→ Jugador "Sergio Busquets" (equipo_id=1)

Equipo "Real Madrid" (id=2)
    ├─→ Jugador "Karim Benzema" (equipo_id=2)
    └─→ Jugador "Luka Modrić" (equipo_id=2)
```

Endpoints RESTful:

```
# Equipos
GET /teams                    # Todos los equipos
GET /teams/1                  # Equipo específico
POST /teams                   # Crear equipo
PUT /teams/1                  # Actualizar equipo
DELETE /teams/1               # Eliminar equipo

# Jugadores
GET /players                  # Todos los jugadores
GET /players/1                # Jugador específico
POST /players                 # Crear jugador
PUT /players/1                # Actualizar jugador
DELETE /players/1             # Eliminar jugador
```

```
# Relación (endpoint especial)
GET /teams/1/players          # Jugadores del equipo 1
```

Ejemplo de petición - Crear jugador:

```
POST /players
Content-Type: application/json

{
  "nombre": "Lionel Messi",
  "posicion": "Delantero",
  "equipo_id": 1
}
```

Respuesta:

```
HTTP/1.1 201 Created
Content-Type: application/json

{
  "id": 10,
  "nombre": "Lionel Messi",
  "posicion": "Delantero",
  "equipo_id": 1
}
```

Ejemplo de petición - Obtener jugadores de un equipo:

```
GET /teams/1/players
```

Respuesta:

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  {
    "id": 10,
    "nombre": "Lionel Messi",
    "posicion": "Delantero",
    "equipo_id": 1
  },
  {
    "id": 11,
    "nombre": "Gerard Piqué",
```

```
    "posicion": "Defensa",
    "equipo_id": 1
  }
]
```

🍽️ Ejemplo Aplicado en este Proyecto: Restaurantes y Platos

Descripción de la Relación

En este proyecto implementamos una relación 1 a muchos entre:

- **Entidad Padre:** Restaurante
- **Entidad Hija:** Plato

Regla de negocio:

- Un restaurante puede tener múltiples platos en su menú
- Cada plato pertenece a un único restaurante

Modelo de Datos Implementado

Restaurante (Entidad Padre)

```
{
  "id": 1,
  "nombre": "La Casa del Marisco",
  "direccion": "Av. Amazonas N24-03",
  "ciudad": "Quito",
  "tipoCocina": "Mariscos",
  "telefono": "02-234-5678"
}
```

Campos:

- `id`: Identificador único del restaurante
- `nombre`: Nombre comercial
- `direccion`: Ubicación física
- `ciudad`: Ciudad donde opera
- `tipoCocina`: Especialidad culinaria (Mariscos, Italiana, Mexicana, etc.)
- `telefono`: Contacto

Plato (Entidad Hija)

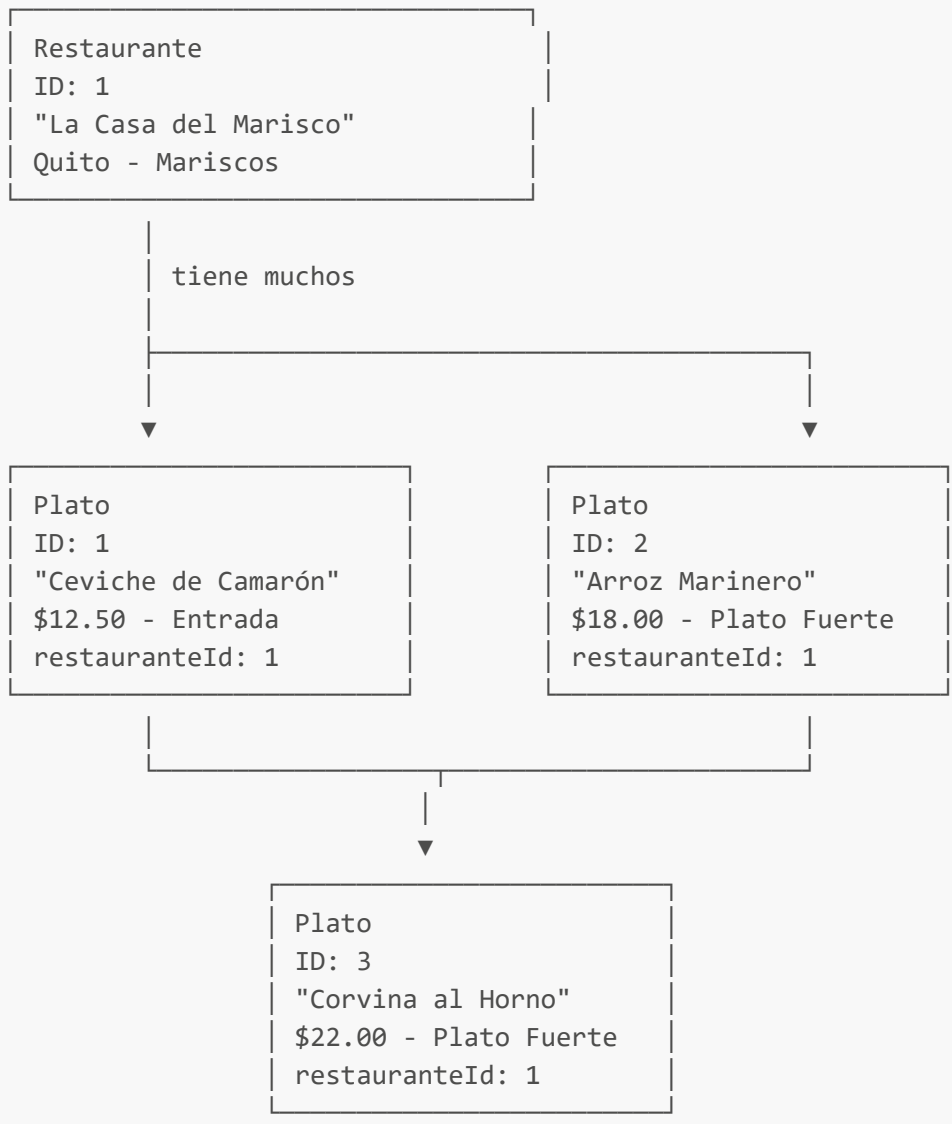
```
{
  "id": 1,
  "nombre": "Ceviche de Camarón",
  "descripcion": "Camarones frescos marinados en limón con cebolla morada y
```

```
cilantro",
  "precio": 12.50,
  "categoria": "Entrada",
  "disponible": true,
  "restauranteId": 1
}
```

Campos:

- `id`: Identificador único del plato
- `nombre`: Nombre del plato
- `descripcion`: Detalles e ingredientes
- `precio`: Precio en dólares
- `categoria`: Tipo (Entrada, Plato Fuerte, Postre, Bebida)
- `disponible`: Si está disponible para ordenar
- `restauranteId`: **Foreign Key** → referencia al restaurante

Visualización de la Relación



Endpoints Implementados

Restaurantes (5 endpoints)

```
# 1. Obtener todos los restaurantes
GET /restaurants
Respuesta: Array de restaurantes

# 2. Obtener restaurante específico
GET /restaurants/1
Respuesta: Objeto restaurante

# 3. Crear nuevo restaurante
POST /restaurants
Body: { "nombre": "...", "ciudad": "...", ... }
Respuesta: Restaurante creado (201)

# 4. Actualizar restaurante
PUT /restaurants/1
Body: { "id": 1, "nombre": "...", ... }
Respuesta: Restaurante actualizado (200)

# 5. Eliminar restaurante
DELETE /restaurants/1
Respuesta: 204 No Content
```

Platos (6 endpoints)

```
# 1. Obtener todos los platos
GET /dishes
Respuesta: Array de todos los platos

# 2. Obtener plato específico
GET /dishes/1
Respuesta: Objeto plato

# 3. Obtener platos de un restaurante (⭐ ENDPOINT DE RELACIÓN)
GET /restaurants/1/dishes
Respuesta: Array de platos del restaurante 1

# 4. Crear nuevo plato
POST /dishes
Body: { "nombre": "...", "precio": 15.00, "restauranteId": 1 }
Respuesta: Plato creado (201)

# 5. Actualizar plato
PUT /dishes/1
Body: { "id": 1, "nombre": "...", "restauranteId": 1 }
Respuesta: Plato actualizado (200)
```

```
# 6. Eliminar plato
DELETE /dishes/1
Respuesta: 204 No Content
```

Ejemplo Completo de Flujo

Paso 1: Crear un restaurante

```
POST /restaurants
Content-Type: application/json

{
  "nombre": "Pizzería Napolitana",
  "direccion": "Calle La Ronda 123",
  "ciudad": "Quito",
  "tipoCocina": "Italiana",
  "telefono": "02-345-6789"
}
```

Respuesta:

```
{
  "id": 2,
  "nombre": "Pizzería Napolitana",
  "direccion": "Calle La Ronda 123",
  "ciudad": "Quito",
  "tipoCocina": "Italiana",
  "telefono": "02-345-6789"
}
```

Paso 2: Agregar platos al restaurante

```
POST /dishes
Content-Type: application/json

{
  "nombre": "Pizza Margarita",
  "descripcion": "Pizza clásica con tomate, mozzarella y albahaca",
  "precio": 10.00,
  "categoria": "Plato Fuerte",
  "disponible": true,
  "restauranteId": 2
}
```

```
POST /dishes
Content-Type: application/json

{
  "nombre": "Lasagna Bolognese",
  "descripcion": "Lasagna tradicional con salsa boloñesa",
  "precio": 14.00,
  "categoria": "Plato Fuerte",
  "disponible": true,
  "restauranteId": 2
}
```

Paso 3: Consultar el menú del restaurante

```
GET /restaurants/2/dishes
```

Respuesta:

```
[
  {
    "id": 10,
    "nombre": "Pizza Margarita",
    "descripcion": "Pizza clásica con tomate, mozzarella y albahaca",
    "precio": 10.00,
    "categoria": "Plato Fuerte",
    "disponible": true,
    "restauranteId": 2
  },
  {
    "id": 11,
    "nombre": "Lasagna Bolognese",
    "descripcion": "Lasagna tradicional con salsa boloñesa",
    "precio": 14.00,
    "categoria": "Plato Fuerte",
    "disponible": true,
    "restauranteId": 2
  }
]
```

Validaciones Implementadas

Al crear un plato:

1. **Validar que el restaurante existe:**

```
Si restauranteId = 999 y no existe
→ 400 Bad Request: "El restaurante con ID 999 no existe"
```

2. **Validar campos obligatorios:**

```
Si falta "nombre"
→ 400 Bad Request: "El nombre del plato es obligatorio"
```

3. **Validar tipos de datos:**

```
Si precio = "diez"
→ 400 Bad Request: "El precio debe ser un número"
```

Al eliminar un restaurante:

```
Si el restaurante tiene platos asociados:
→ 409 Conflict: "No se puede eliminar el restaurante porque tiene 5 platos
asociados"

O bien, eliminar en cascada (eliminar también todos los platos)
```

☑ Ventajas de usar REST

1. Escalabilidad

Por ser stateless, es fácil escalar horizontalmente:

```
Cliente → Load Balancer → Servidor 1
                        → Servidor 2
                        → Servidor 3
```

Cualquier servidor puede atender cualquier petición.

2. Flexibilidad

Múltiples clientes pueden consumir la misma API:

```
API RESTful
    ├→ Aplicación Web (React)
    ├→ Aplicación Móvil (React Native)
```

```
      ├─→ Aplicación Desktop (Electron)
      └─→ Otro servicio backend (Node.js)
```

3. Mantenibilidad

- URLs intuitivas y predecibles
- Separación clara entre recursos
- Documentación estandarizada (Swagger/OpenAPI)

4. Performance

- Uso eficiente de cache HTTP
- Respuestas ligeras (JSON)
- Posibilidad de comprimir respuestas (gzip)

5. Independencia de tecnología

- Cliente puede estar en cualquier lenguaje
- Servidor puede estar en cualquier framework
- Comunicación mediante HTTP estándar

🎯 Conclusiones

Principios clave de REST:

1. ✅ **Usar métodos HTTP correctamente**
   - GET para leer
   - POST para crear
   - PUT para actualizar todo
   - PATCH para actualizar parcialmente
   - DELETE para eliminar
2. ✅ **Diseñar URLs jerárquicas y semánticas**
   - `/restaurants` para colecciones
   - `/restaurants/1` para elementos
   - `/restaurants/1/dishes` para relaciones
3. ✅ **Ser stateless**
   - Cada petición es independiente
   - Incluir toda la información necesaria
4. ✅ **Usar JSON como formato**
   - Estándar de la industria
   - Fácil de leer y parsear
5. ✅ **Implementar relaciones correctamente**

    - Foreign Keys en la entidad hija
    - Endpoints especiales para acceder a relaciones
    - Validaciones apropiadas

En este proyecto aplicamos:

- ✅ Diseño RESTful completo
- ✅ Relación 1:N (Restaurantes → Platos)
- ✅ 11 endpoints documentados
- ✅ Validaciones y manejo de errores
- ✅ Documentación con Swagger/OpenAPI
- ✅ Testing con Bruno

📚 Referencias

- Fielding, Roy Thomas. "Architectural Styles and the Design of Network-based Software Architectures." Doctoral dissertation, University of California, Irvine, 2000.
- REST API Tutorial
- OpenAPI Specification
- HTTP Methods - MDN
- Richardson Maturity Model

Proyecto: API RESTful de Restaurantes y Platos
Estudiante: Dilan Real
Institución: Escuela Politécnica Nacional
Fecha: 12 de diciembre de 2025