Apunte de Clase 16 — Consumo de API Externa con RestClient

> *Integrar servicios externos en la lógica de negocio de un microservicio particular en el backend de una una aplicación.*

🎯 Objetivo del apunte

Comprender los fundamentos conceptuales del **consumo de servicios externos** desde un backend desarrollado con Spring Boot, analizando las diferentes formas en que los sistemas pueden **intercambiar información** entre sí y cómo incorporar esas integraciones dentro de la lógica de negocio de una aplicación.

Este bloque busca que los estudiantes adquieran una **visión clara y general** de:

- Qué es una API y cómo se comunican entre sí los sistemas.
- Qué tipos de APIs existen y en qué contextos se utilizan.
- Cuáles son los principales **desafíos de integración** entre servicios.
- Cómo evolucionó el soporte de clientes HTTP en el ecosistema de Spring.
- Qué problemas resuelve el nuevo **RestClient** introducido en Spring 6 / Boot 3.2+.

Introducción al consumo de APIs externas

¿Qué es una API?

Como ya venimos viendo el término **API (Application Programming Interface)** hace referencia a un *conjunto de reglas y contratos* que permiten que dos sistemas distintos se comuniquen entre sí. Las APIs definen cómo deben enviarse las solicitudes, cómo serán las respuestas y qué formatos se usarán para intercambiar información (por ejemplo, JSON o XML).

Podemos pensar una API como un **acuerdo entre dos aplicaciones**:

> "Si me pedís esto, de esta forma, yo te devolveré aquello, con esta estructura."

Ya hicimos revisión de las buenas prácticas de diseño, de las herramientas de diseño y documentación y de algunos patrones específicos al respecto, ahora nos toca empezar a verlas como cajas negras para analizar la interacción entre ellas.

Concepto de API pública y servicios externos

- **API interna:** solo accesible dentro de una organización (por ejemplo, entre microservicios propios).
- **API pública:** expuesta a Internet, permitiendo que aplicaciones externas la consuman (por ejemplo, la API de Google Maps o OpenWeather).
- **Servicio externo:** cualquier aplicación o sistema que nuestra aplicación necesita consultar o invocar para obtener o enviar información.

Escenarios de integración entre microservicios

En arquitecturas de microservicios, la funcionalidad total del sistema se divide en componentes pequeños y especializados. Cada microservicio ofrece una API para que los demás puedan comunicarse con él. Ejemplos típicos:

- Un servicio de **Pedidos** que consulta a otro servicio de **Proveedores**.
- Un servicio de **Usuarios** que valida credenciales ante un servicio de **Autenticación**.
- Un servicio de **Pagos** que interactúa con **pasarelas externas** como MercadoPago o Stripe.

REST vs SOAP vs gRPC (Repaso breve)

En la actualidad, la mayoría de las integraciones entre sistemas se realizan a través de **APIs REST**, usando **JSON** como formato de intercambio debido a que REST Full lo adopta como su estándar. Sin embargo, esta no es la única forma posible ni necesariamente la más adecuada en todos los contextos.

A lo largo del tiempo, las aplicaciones distribuidas han utilizado diferentes **estilos de comunicación** entre clientes y servidores. Cada uno tiene sus propios **formatos de datos**, **protocolos**, **ventajas** y **limitaciones**.

📡 REST (Representational State Transfer) Repaso

REST no es un protocolo, sino un estilo arquitectónico propuesto por Roy Fielding. Define un conjunto de principios que aprovechan las capacidades del protocolo HTTP para representar y manipular recursos.

- **Formato más usado:** JSON (aunque también puede usarse XML, YAML o texto plano).
- **Ventajas:**
  - Ligero y simple de implementar.
  - Utiliza directamente los métodos HTTP estándar (`GET`, `POST`, `PUT`, `DELETE`, etc.).
  - Fácilmente consumible desde navegadores y aplicaciones móviles.
  - Altamente interoperable.
- **Desventajas:**
  - Menor formalidad en la definición del contrato (comparado con SOAP o gRPC).
  - Sin control de versión del contrato incorporado.
  - Puede requerir convenciones manuales para manejar errores, validaciones y versionado.
- **Uso típico:** aplicaciones web, microservicios y APIs públicas.

🧩 SOAP (Simple Object Access Protocol)

SOAP es un protocolo formal basado en XML. Surgió antes de REST y fue durante mucho tiempo el estándar dominante en entornos corporativos. Define reglas estrictas sobre cómo deben estructurarse los mensajes, incluyendo cabeceras, cuerpo y metadatos.

- **Formato:** exclusivamente **XML**, con estructura definida mediante un **WSDL (Web Service Definition Language)**.
- **Ventajas:**
  - Contrato fuertemente tipado (WSDL define la interfaz y tipos de datos).
  - Estándar bien definido, con soporte para **seguridad**, **transacciones** y **mensajería confiable**.
  - Ideal para entornos donde se requiere auditoría o trazabilidad formal.
- **Desventajas:**
  - Verbosidad: los mensajes XML suelen ser muy pesados.
  - Requiere herramientas más complejas para su consumo.
  - Menor flexibilidad y mayor acoplamiento.

- **Uso típico:** sistemas bancarios, seguros, ERPs y entornos gubernamentales con requisitos formales de interoperabilidad.

⚙️ **gRPC (Google Remote Procedure Call)**

gRPC es un framework moderno de comunicación desarrollado por Google. Se basa en el concepto de **llamadas a procedimientos remotos (RPC)** y utiliza **HTTP/2** como protocolo de transporte.

- **Formato: Protobuf (Protocol Buffers)**, un formato binario ligero y eficiente.
- **Ventajas:**
  - Transmisión binaria compacta (mucho más rápida que JSON o XML).
  - Generación automática de código cliente/servidor a partir del contrato `.proto`.
  - Soporta **streaming bidireccional** y **autenticación TLS**.
  - Ideal para arquitecturas de microservicios con alto tráfico interno.
- **Desventajas:**
  - No tan legible para humanos (requiere herramientas o librerías específicas).
  - No está diseñado para ser consumido directamente desde navegadores.
  - Depende del uso de Protobuf, lo que introduce un paso de compilación adicional.
- **Uso típico:** comunicación interna entre microservicios, sistemas de tiempo real o IoT.

🔍 **Comparativa resumida**

Tecnología	Estilo	Formato	Ventajas	Uso típico
REST	Arquitectura basada en recursos y métodos HTTP	JSON / XML	Ligero, simple, ampliamente soportado	Web y microservicios modernos
SOAP	Protocolo formal basado en XML	XML	Estricto, estandarizado, ideal para entornos corporativos	Integraciones legacy, sistemas financieros
gRPC	RPC binario sobre HTTP/2	Protobuf	Rápido, eficiente, orientado a contratos	Sistemas de alto rendimiento o IoT

APIs públicas de ejemplo

Las **APIs públicas** son servicios expuestos por organizaciones o comunidades que permiten acceder a información o funcionalidades específicas sin necesidad de construir una infraestructura propia. Suelen estar disponibles mediante HTTP y devuelven datos en formatos estandarizados como JSON o XML.

A continuación, se presentan algunos ejemplos ampliamente utilizados, junto con enlaces a su documentación y ejemplos de endpoints.

🌍 Google Maps API

Ofrece funcionalidades de **geolocalización**, **rutas**, **distancias** y **búsqueda de lugares**. Es una de las APIs más utilizadas en el mundo, tanto en aplicaciones móviles como web.

- **Documentación general:** https://developers.google.com/maps/documentation
- **Distance Matrix API:** cálculo de distancias y tiempos entre coordenadas o direcciones.

    - https://developers.google.com/maps/documentation/distance-matrix

- **Ejemplo de endpoint:**

```
https://maps.googleapis.com/maps/api/distancematrix/json?
origins=Córdoba&destinations=Rosario&key=TU_API_KEY
```

Usos típicos: cálculo de rutas logísticas, estimación de tiempos de entrega, localización de sucursales, trazado de recorridos.

☁️ OpenWeather API

Proporciona información meteorológica actual, pronósticos y datos históricos de clima en formato JSON.

- **Sitio oficial:** https://openweathermap.org/api
- **Tipos de API disponibles:**
    - *Current Weather Data* (clima actual)
    - *5 Day / 3 Hour Forecast* (pronóstico a corto plazo)
    - *One Call API* (histórico, pronóstico y alertas en una sola llamada)
- **Ejemplo de endpoint:**

```
https://api.openweathermap.org/data/2.5/weather?
q=Córdoba&appid=TU_API_KEY&units=metric&lang=es
```

Usos típicos: aplicaciones de turismo, agricultura inteligente, planificación de eventos o consumo energético.

🐙 GitHub API

Permite interactuar con casi todos los recursos de GitHub: repositorios, usuarios, issues, pull requests, commits, entre otros.

- **Documentación REST:** https://docs.github.com/en/rest
- **Documentación GraphQL:** https://docs.github.com/en/graphql
- **Ejemplo de endpoint:**

```
https://api.github.com/users/octocat/repos
```

Usos típicos: automatización de tareas DevOps, obtención de métricas de repositorios, dashboards de contribuciones, análisis de proyectos open source.

💱 CurrencyLayer API

Ofrece tasas de cambio de divisas en tiempo real y datos históricos. Ideal para aplicaciones financieras o de comercio electrónico.

- **Documentación:** https://currencylayer.com/documentation
- **Ejemplo de endpoint:**

```
http://api.currencylayer.com/live?
access_key=TU_API_KEY&currencies=USD,EUR,ARS
```

Usos típicos: conversión de monedas, análisis de tendencias financieras, cotizaciones automáticas.

🧪 JSONPlaceholder API

API gratuita ideal para **pruebas y aprendizaje**, que simula un backend real con recursos comunes (usuarios, publicaciones, comentarios, etc.).

- **Sitio oficial:** https://jsonplaceholder.typicode.com
- **Ejemplo de endpoint:**

```
https://jsonplaceholder.typicode.com/posts/1
```

Usos típicos: testing de clientes HTTP, prototipos de interfaces o ejercicios de programación.

🗂️ Public APIs Directory

Catálogo colaborativo con cientos de APIs clasificadas por tema (música, transporte, educación, datos abiertos, etc.). Es una excelente fuente para explorar nuevas integraciones.

- https://public-apis.io
- https://github.com/public-apis/public-apis

Usos típicos: búsqueda de APIs abiertas para proyectos, inspiración de prácticas o actividades académicas.

> 💡 **Consejo docente:** utilizar ejemplos como `JSONPlaceholder` o `OpenWeather` en las primeras prácticas permite introducir el consumo de APIs reales sin necesidad de gestionar claves o costos, facilitando la comprensión de los conceptos de *request*, *response*, *endpoint* y *payload*.

Cliente HTTP en Spring Boot

Aunque siempre podemos realizar llamadas HTTP utilizando librerías estándar de Java (como `HttpURLConnection` o `HttpClient`), **Spring Framework** ofrece implementaciones más expresivas y declarativas que simplifican el flujo de comunicación entre aplicaciones. Estas herramientas se integran con el ecosistema Spring, ofreciendo conversión automática de objetos, manejo de errores, interceptores, seguridad y pruebas.

La evolución natural de los clientes HTTP en Spring puede resumirse así:

Etapa	Clase / Tecnología	Características	Ejemplo mínimo

Etapa	Clase / Tecnología	Características	Ejemplo mínimo
Spring 3–5	`RestTemplate`	Síncrono y bloqueante; API sencilla; ampliamente utilizado; marcado como *deprecated* desde Spring 6.	`new RestTemplate().getForObject("/api/foo/{id}", Foo.class, id);`
Spring 5+	`WebClient`	No bloqueante (reactivo); soporta back-pressure; ideal para WebFlux y arquitecturas reactivas.	`WebClient.create(base).get().uri("/api/foo/{id}", id).retrieve().bodyToMono(Foo.class).block();`
Spring 6 / Boot 3.2+	`RestClient`	Bloqueante, moderno y fluido; reemplazo natural de `RestTemplate`; mejor ergonomía y manejo de errores.	`restClient.get().uri("/api/foo/{id}", id).retrieve().body(Foo.class);`

> 💡 **Regla práctica:** En aplicaciones **Spring MVC (bloqueantes)** se recomienda usar `RestClient`. En aplicaciones **WebFlux (reactivas)**, la mejor opción sigue siendo `WebClient`.

Configuración básica y creación de beans

La forma recomendada de declarar clientes HTTP en proyectos Spring Boot es mediante **beans configurables**. Esto facilita la inyección de dependencias y la reutilización.

```
@Configuration
public class HttpClientsConfig {

  @Bean
  RestClient restClient(@Value("${api.base-url}") String baseUrl) {
      return RestClient.builder()
          .baseUrl(baseUrl)
          .build();
  }

  @Bean
  WebClient webClient(@Value("${api.base-url}") String baseUrl) {
```

```
        return WebClient.builder()
            .baseUrl(baseUrl)
            .build();
    }
}
```

Comparativa de uso

🪨 RestTemplate (histórico, bloqueante)

```
RestTemplate restTemplate = new RestTemplate();
Foo foo = restTemplate.getForObject(base + "/api/foo/{id}", Foo.class, 42);
System.out.println(foo);
```

- Bloquea el hilo hasta recibir la respuesta.
- Sencillo de usar, pero limitado en configuración.
- Su mantenimiento fue discontinuado en Spring 6.

⚙️ WebClient (reactivo, no bloqueante)

```
WebClient webClient = WebClient.builder()
    .baseUrl(base)
    .build();

Foo foo = webClient.get()
    .uri("/api/foo/{id}", 42)
    .retrieve()
    .bodyToMono(Foo.class)
    .block(); // Solo en ejemplo; evita bloquear en WebFlux real.
```

- Utiliza programación reactiva (basado en Project Reactor).
- Permite flujos asíncronos y streaming bidireccional.
- Recomendado en entornos de alto rendimiento.

🚀 RestClient (moderno, fluido, bloqueante)

```
RestClient restClient = RestClient.builder()
    .baseUrl(base)
    .build();

Foo foo = restClient.get()
    .uri("/api/foo/{id}", 42)
    .retrieve()
    .body(Foo.class);
```

- API fluida inspirada en `WebClient`, pero síncrona.
- Incluye manejo nativo de `ProblemDetail` y status codes.

- Recomendado en proyectos MVC o microservicios Spring Boot 3.2+.

En resumen

La evolución de los clientes HTTP en Spring refleja el avance hacia un modelo más **declarativo y expresivo**, con soporte nativo para buenas prácticas modernas: serialización automática, validaciones, resiliencia y compatibilidad con la programación reactiva.

> En este bloque se busca que el estudiante comprenda las diferencias conceptuales y prácticas entre estos clientes, para elegir el más adecuado según la naturaleza de la aplicación (bloqueante o reactiva). Sin embargo como no hemos planteado avanzar con aplicaciones reactivas en la asignatura Backend para los ejemplos nos vamos a enfocar en `RestClient`

Características y ventajas de `RestClient`

- Basado en **Java 21** y **Spring 6**.
- API fluida: `restClient.get().uri("/api/…").retrieve().body(Foo.class)`.
- Integración con el manejo de **status codes** y **ProblemDetail**.
- Configurable con **baseUrl, interceptores, autenticación y timeout**.
- Compatible con **inyección de dependencias** (`@Bean`, `@Component`).
- Soporta pruebas unitarias con **MockRestServiceServer**.

Como ya vimos: **la configuración básica y creación de beans es:**

```
@Configuration
public class RestClientConfig {

  @Bean
  RestClient restClient(@Value("${api.base-url}") String baseUrl) {
      return RestClient.builder()
          .baseUrl(baseUrl)
          .build();
  }
}
```

Estructura de request y response

Método	Uso	Ejemplo
`GET`	Obtener datos	`restClient.get().uri("/users").retrieve().body(...)`
`POST`	Crear recurso	`restClient.post().uri("/users").body(obj).retrieve()`
`PUT`	Actualizar recurso	`restClient.put().uri("/users/1").body(obj).retrieve()`
`DELETE`	Eliminar recurso	`restClient.delete().uri("/users/1").retrieve()`

Manejo de errores y validaciones

Tipos de errores HTTP comunes

Código	Nombre	Significado

Código	Nombre	Significado
400	Bad Request	Petición mal formada
401	Unauthorized	Falta autenticación
403	Forbidden	Acceso denegado
404	Not Found	Recurso inexistente
500	Internal Server Error	Error en el servidor remoto
502	Bad Gateway	Error en servicio intermedio (gateway/proxy)

Manejo de excepciones con `RestClientResponseException`

```
try {
    var response = restClient.get()
        .uri("/api/proveedores/99")
        .retrieve()
        .body(ProveedorDTO.class);
} catch (RestClientResponseException ex) {
    System.err.println("Error: " + ex.getStatusCode());
    System.err.println("Respuesta: " + ex.getResponseBodyAsString());
}
```

Mapeo de `ProblemDetail` en Spring Boot

Problem Detail es un formato de error estandarizado (RFC 7807) que Spring 6 soporta nativamente.

```
{
  "type": "about:blank",
  "title": "Not Found",
  "status": 404,
  "detail": "Proveedor no existente: 99"
}
```

Buenas prácticas de logging y resiliencia

- Registrar **toda solicitud saliente** y su respuesta (status, tiempo, URI).
- Evitar exponer **datos sensibles** en logs.
- Manejar **timeouts y reintentos** controlados.
- Utilizar **circuit breakers** o mecanismos de fallback.
- Monitorear la latencia y disponibilidad de los servicios consumidos.

Veamos todo de forma gráfica

C4 Model: por qué y cómo

C4 Model (de Simon Brown) es un enfoque para documentar arquitectura de software usando **cuatro niveles de diagramas** que van desde una vista muy amplia hasta el detalle del código. El objetivo es **comunicar con claridad** la estructura del sistema y las responsabilidades de cada parte, adaptando el nivel de detalle al público.

- **Nivel 1 – Contexto del Sistema**: el sistema y su entorno (usuarios, sistemas vecinos).
- **Nivel 2 – Contenedores**: los *contenedores* que ejecutan el software: aplicaciones, bases de datos, colas, microservicios, etc., y cómo **se comunican**.
- **Nivel 3 – Componentes**: componentes internos dentro de un contenedor (módulos, capas, servicios internos).
- **Nivel 4 – Código**: clases, interfaces y relaciones a muy bajo nivel (poco usado en documentación viva).

El Nivel 2 (Contenedores) es clave en este punto

En este bloque presentamos **dos microservicios** que se comunican entre sí. El **Nivel 2** es el lugar ideal para mostrar:

- Qué contenedores existen (por ejemplo, *ms-proveedores* y *ms-pedidos*).
- Qué tecnología usa cada uno (Spring Boot 3.5, Java 21, HTTP/JSON).
- **Relaciones y direccionalidad** de la comunicación (quién consume a quién).
- Dependencias externas (por ejemplo, una base de datos, un gateway o un servicio de autenticación, si aplicara).

Esta visión permite que quienes **no codifican** (o recién se inician) entiendan el flujo entre servicios, y que quienes **sí codifican** tengan un mapa claro para ubicar el código y sus responsabilidades.

Diagrama C4 — Nivel 2 (Contenedores) para el ejemplo de dos microservicios

> **Leyenda**: [Persona] – (Sistema) – [Contenedor]

```
@startuml
!include https://raw.githubusercontent.com/plantuml-stdlib/C4-
PlantUML/master/C4_Container.puml
LAYOUT_WITH_LEGEND()
Person(user, "Usuario/Cliente - Front u otra app")

System_Boundary(s1, "Sistema de Pedidos") {
  Container(ms_pedidos, "ms-pedidos", "Spring Boot 3.5, Java 21", "Exponer API
de pedidos; orquestar y enriquecer datos")
}
System_Boundary(s2, "Sistema de Proveedores") {
  Container(ms_proveedores, "ms-proveedores", "Spring Boot 3.5, Java 21",
"Exponer proveedores (leer/listar/detalle)")
}

Rel(user, ms_pedidos, "Realiza consultas/operaciones de pedidos", "HTTP/JSON")
Rel(ms_pedidos, ms_proveedores, "Consulta datos de proveedor", "HTTP/JSON
(RestClient)")
@enduml
```

```
@startuml
!include https://raw.githubusercontent.com/plantuml-stdlib/C4-
PlantUML/master/C4_Container.puml
LAYOUT_WITH_LEGEND()
Person(user, "Usuario/Cliente - Front u otra app")

System_Boundary(s1, "Sistema de Pedidos") {
```

```
  Container(ms_pedidos, "ms-pedidos", "Spring Boot 3.5, Java 21", "Exponer API
de pedidos; orquestar y enriquecer datos")
}
System_Boundary(s2, "Sistema de Proveedores") {
  Container(ms_proveedores, "ms-proveedores", "Spring Boot 3.5, Java 21",
"Exponer proveedores (leer/listar/detalle)")
}

Rel(user, ms_pedidos, "Realiza consultas/operaciones de pedidos", "HTTP/JSON")
Rel(ms_pedidos, ms_proveedores, "Consulta datos de proveedor", "HTTP/JSON
(RestClient)")
@enduml
```

> [!TIP] **Tips a tener en cuanta**
>
> - **Direccionalidad**: `ms-pedidos` **consume** a `ms-proveedores` (evitar acople circular).
> - **Contratos de API**: ambos exponen `/api/...` con JSON.
> - **Tecnología explícita**: ayuda a entender el stack y los requisitos de ejecución.
> - **Evolución natural**: este nivel admite sumar gateway, auth service, DBs, etc.

Ejemplo: Consumo de API entre microservicios propios (FoodMatch)

Escenario general

Para comprender el intercambio entre APIs antes de integrar servicios externos reales, usaremos dos microservicios simples del dominio **FoodMatch** (el ejemplo completo y funcional puede allarse en ejemplos/foodmatch):

- **Servicio A — ms-comidas (Server):** expone una API propia con un catálogo de comidas en memoria.
- **Servicio B — ms-maridaje (Client):** consume la API de ms-comidas usando `RestClient` y aplica reglas de maridaje para sugerir una bebida.

El objetivo es poder observar la **estructura** y el **flujo de comunicación** entre microservicios: el cliente orquesta y agrega valor sobre los datos del servidor.

Creación del microservicio *Server* (ms-comidas)

- Endpoints básicos:
  - `GET /api/comidas` → listado (con filtro opcional `?q=` por nombre)
  - `GET /api/comidas/{id}` → detalle por id
- Respuesta servida desde un **singleton en memoria** con varias categorías (PIZZA, SUSHI, HAMBURGUESA, TACOS, ASADO, ENSALADA, POSTRE, PASTA, RAMEN).
- Ejemplo de modelo:

```
@Data @AllArgsConstructor @NoArgsConstructor @Builder
public class Comida {
  private Integer id;
  private String nombre;      // "Pizza Muzza", "Sushi", ...
  private String tipo;        // PIZZA | SUSHI | ...
  private boolean picante;
```

```
  private String perfil;      // SALADO | DULCE | UMAMI
  private String grasa;       // BAJA | MEDIA | ALTA
}
```

Creación del microservicio *Client* o *Consumidor* (ms-maridaje)

- Configura un bean `RestClient` apuntando a la URL del servidor.
- Implementa un cliente `ComidasApiClient` con métodos para obtener una comida por id.
- Expone un endpoint de maridaje que recibe el id de la comida, consulta el servidor y devuelve una **bebida sugerida** con una **razón**.

Configuración de `application.properties` (cliente)

```
app.comidas.base-url=http://localhost:8080
springdoc.swagger-ui.path=/swagger-ui.html
```

Y en el cliente:

```
@Bean
RestClient comidasClient(@Value("${app.comidas.base-url}") String baseUrl) {
  return RestClient.builder().baseUrl(baseUrl).build();
}
```

Implementación de cliente usando `RestClient`

```
@Component
@RequiredArgsConstructor
public class ComidasApiClient {
  private final RestClient comidasClient;

  public ComidaDTO obtenerPorId(Integer id) {
    return comidasClient.get()
        .uri("/api/comidas/{id}", id)
        .retrieve()
        .body(ComidaDTO.class);
  }
}
```

Mapeo de respuesta JSON a DTOs

Spring convierte automáticamente el cuerpo JSON de la respuesta en objetos Java mediante Jackson.

```
@Data @AllArgsConstructor @NoArgsConstructor @Builder
public class ComidaDTO {
  private Integer id;
  private String nombre;
  private String tipo;      // PIZZA | SUSHI | ...
```

```
  private boolean picante;
  private String perfil;    // SALADO | DULCE | UMAMI
  private String grasa;     // BAJA | MEDIA | ALTA
}
```

```
@Data @AllArgsConstructor @NoArgsConstructor @Builder
public class BebidaDTO {
  private String nombre;    // "Cerveza IPA", "Malbec", "Agua con gas"
  private String tipo;      // CERVEZA | VINO | REFRESCO | AGUA | TÉ
  private String razon;     // breve explicación del porqué
}
```

Lógica de maridaje (reglas simples en el cliente)

```
@Component
public class MotorMaridaje {
  public BebidaDTO sugerir(ComidaDTO c) {
    switch (c.getTipo()) {
      case "PIZZA" ->  bebida("Cerveza Lager", "CERVEZA", "Limpia grasa y
acompaña masas/quesos suaves");
      case "SUSHI" ->  bebida("Sake / Té verde", "TÉ", "Perfiles limpios que no
tapan pescado/arroz");
      case "HAMBURGUESA" -> bebida("Cerveza IPA", "CERVEZA", "Amargor corta
grasa alta y resalta tostados");
      case "ASADO" ->  bebida("Malbec", "VINO", "Taninos acompañan carnes rojas
y grasas");
      case "PASTA" ->  bebida("Vino Blanco Seco", "VINO", "Acidez acompaña
salsas cremosas/queso");
      case "POSTRE" -> bebida("Café o Porto", "REFRESCO", "Dulzor del vino o
amargor del café equilibra");
      default -> {}
    }
    if (c.isPicante())
      return bebida("Cerveza de Trigo / Lassi", "CERVEZA", "Baja graduación y
cuerpo refrescante calman el picante");
    if ("ALTA".equals(c.getGrasa()))
      return bebida("Agua con gas o IPA suave", "AGUA", "Carbonatación/ligero
amargor limpian el paladar");
    if ("DULCE".equals(c.getPerfil()))
      return bebida("Infusión / Café", "TÉ", "El amargor equilibra preparaciones
dulces");
    return bebida("Agua con gas", "AGUA", "Siempre combina y limpia el
paladar");
  }
  private BebidaDTO bebida(String n, String t, String r) { return new
BebidaDTO(n, t, r); }
}
```

Exposición de endpoint en el microservicio cliente

```
@RestController
@RequestMapping("/api/maridaje")
@RequiredArgsConstructor
public class MaridajeController {
  private final ComidasApiClient comidas;
  private final MotorMaridaje motor;

  @GetMapping("/{idComida}")
  public ResponseEntity<BebidaDTO> sugerirPorId(@PathVariable Integer idComida)
{
    ComidaDTO comida = comidas.obtenerPorId(idComida);
    return ResponseEntity.ok(motor.sugerir(comida));
  }

  @PostMapping
  public ResponseEntity<BebidaDTO> sugerirPorBody(@RequestBody ComidaDTO comida)
{
    return ResponseEntity.ok(motor.sugerir(comida));
  }
}
```

Pruebas rápidas

- `GET http://localhost:8081/api/comidas` → lista de comidas del servidor
- `GET http://localhost:8081/api/comidas/1` → detalle de comida
- `GET http://localhost:8082/api/maridaje/1` → sugerencia de bebida por id (cliente → servidor)
- `POST http://localhost:8082/api/maridaje` con body `ComidaDTO` → sugerencia evaluando directamente en el cliente

Extensiones y temas avanzados

En el próximo bloque al aplicar conceptos de seguridad vamos a avanzar sobre más herramientas de RestClient que aquí solo mencionamos.

Uso de interceptores para logging y autenticación (builder)

> `RestClient` permite interceptar solicitudes y respuestas para registrar o agregar cabeceras (por ejemplo, tokens).

```
@Bean
RestClient restClientWithLogging(@Value("${api.base-url}") String baseUrl) {
  return RestClient.builder()
      .baseUrl(baseUrl)
      .requestInterceptor((request, body, execution) -> {
          long t0 = System.currentTimeMillis();
          var response = execution.execute(request, body);
          long dt = System.currentTimeMillis() - t0;
          System.out.println("→ " + request.getMethod() + " " + request.getURI()
+ " | ⬅️ " + response.getStatusCode() + " | " + dt + "ms");
          return response;
      })
      .build();
}
```

Circuit Breaker (concepto introductorio)

Un *circuit breaker* evita que un servicio saturado o caído siga recibiendo llamadas inútiles. Si el cliente detecta varios fallos consecutivos, "abre el circuito" y responde con un fallback local hasta que el servicio remoto se recupere.

Herramientas populares:

- **Resilience4j** (para Spring Boot 3.x)
- **Spring Cloud Circuit Breaker**

Testeo de clientes REST con `MockRestServiceServer`

Permite simular un servidor remoto sin necesidad de conexión real y validar el comportamiento del cliente frente a distintas respuestas.

```
MockRestServiceServer server = MockRestServiceServer.bindTo(restClient).build();
server.expect(requestTo("/api/proveedores/1"))
      .andRespond(withSuccess("{\"id\":1,\"nombre\":\"Test\"}",
MediaType.APPLICATION_JSON));
```

Seguridad y manejo de claves de API

Cuando se consumen APIs externas (Google, OpenWeather, etc.), suele requerirse una **API Key**. Buenas prácticas:

- No guardar claves en el código fuente.
- Usar variables de entorno o `application.properties` con `${API_KEY}`.
- En producción, usar un *Secret Manager* o configuración cifrada.
- Transmitir siempre las solicitudes mediante **HTTPS**.

🧠 Resumen

El consumo de servicios externos es una capacidad esencial en los sistemas modernos. Toda aplicación backend debe estar preparada para:

- **Interactuar con otros servicios** (propios o de terceros),
- **Manejar errores y latencias** de red,
- **Asegurar la integridad y seguridad** de los datos intercambiados, y
- **Registrar y monitorear** sus integraciones.

Comprender la **comunicación entre APIs** no es solo una cuestión técnica, sino también de diseño arquitectónico y de responsabilidad entre componentes.

📚 Bibliografía y recursos recomendados

- Spring Framework Reference: Rest Clients (Spring 6)
- Baeldung: *Guide to RestClient in Spring Boot 3.2*
- Simon Brown – *The C4 Model for Software Architecture*
- Newman, S. (2015). *Building Microservices* – O'Reilly.
- Martin, R. (2008). *Clean Code* – Pearson.

- Packt. *Mastering Microservices with Java* – Cap. 7 “Inter-Service Communication”.

> **Próximo paso**: guía práctica con los dos microservicios Spring Boot 3.5 que se consumen entre sí (cliente con `RestClient`).

Apunte 17: API Gateway

Fundamentos

Es importante tener presente que si se decidió desarrollar el backend con una arquitectura de microservicios, cada uno de estos servicios se despliega para ejecutarse como un proceso separado y que para comunicarse con ellos el cliente debe conocer los detalles de conexión de cada uno de estos servicios. Si se considera el caso más frecuente, donde los servicios brindan una API sobre HTTP, esto significa que el cliente debe conocer la IP y el puerto correspondiente a la API de cada microservicio. Este esquema puede ser manejable si son pocos microservicios, pero es una verdadera complicación cuando esta cantidad crece.

Adicionalmente a lo ya descripto, la estrategia de realizar una conexión directa entre el cliente y cada uno de los microservicios presenta una serie de inconvenientes y desafíos adicionales, entre ellos:

- *Acoplamiento*

El hecho de realizar una conexión directa entre el cliente y cada microservicio implica que el cliente conoce cuál es la estructura interna del backend, algo que no es recomendable. Se espera que el cliente se conecte a una API unificada y que los detalles de implementación internos del backend no sean expuestos al cliente.

También debe considerarse que, si la conexión es directa hacia los microservicios, los cambios en la interfaz de los mismos afectarán al cliente. Podría pasar que una interfaz se modifique, o que se decida (algo no tan infrecuente) separar las funcionalidades que antes estaban en un microservicio en más de uno, y también lo contrario: unificar en un microservicio las funcionalidades de maś de uno de ellos.

- *Aspectos que atraviesan toda la aplicación*

Existen algunos aspectos, como autenticación, autorización, captura de logs y trazas, etc., que deben ser aplicados en todos los microservicios. Un caso típico tiene que ver con la seguridad. A menos que se estuviera tratando de datos o acciones no protegidas, para que un microservicio responda a una petición, debe tener dos garantías básicas:

1. Que quien realiza la petición sea quién dice ser (Autenticación)
2. Que quien realiza la petición tenga permisos para realizar la acción que pretende (Autorización).

Si cada microservicio es un proceso separado y, a su vez, los clientes se conectan directamente con ellos, entonces cada microservicio debería encargarse de cada uno de los aspectos mencionados. Esto significa programar (o incorporar) en el código de cada microservicio el manejo, por ejemplo, de la autorización y la autenticación. Se podría pensar que no es algo tan problemático, en definitiva se puede hacer una librería compartida e incorporarlo en todos los servicios y problema resuelto. Lamentablemente no es tan simple: si hubiera una actualización que realizar sobre esa librería común, tendrían que re-compilarse/re-empaquetarse y actualizarse todos los servicios; por otro lado, ¿qué pasa si no todos los servicios están

desarrollados en el mismo lenguaje?. Otro aspecto negativo de esta estrategia es que suele haber distintos equipos para los distintos microservicios y podría pasar que un equipo se olvidara de incorporar, por ejemplo, el aspecto de seguridad a su microservicio; o bien, el equipo podría realizar su propia implementación de las políticas de seguridad que no sea consistente con el resto de los servicios.

- *Posibilidad de tener protocolos no web-friendly*

Nada obliga a que un microservicio tenga que exponer su API mediante HTTP (y, por supuesto, nada obliga a utilizar un estilo REST ni el formato JSON). Por distintos motivos podría querer diseñarse un microservicio cuya interfaz se brinde mediante otro protocolo, por ejemplo para soportar plataformas de streaming o colas de mensajería, entre algunas opciones.

Si este fuera el caso y se estuviera utilizando algún protocolo que no es *web-friendly*, el cliente tendría que lidiar con la complejidad de manejar esta comunicación. En principio esto no parece muy preocupante, pero hay que considerar que las tecnologías más típicas de desarrollo para los clientes del backend tienen incorporadas todas las capacidades para realizar invocaciones de estilo REST, pero no suelen tener integradas capacidades de comunicación con otros protocolos (como podría ser AMPQ, por ejemplo).

- *Mayor superficie de exposición*

En relación a la seguridad, hay que tener en cuenta que cuantos más servicios estén publicados al exterior, mayor será la "superficie" de exposición y por lo tanto, una mayor cantidad de puntos por donde se podría recibir un ataque.

Introducción al API Gateway

Por todo lo expresado anteriormente, queda claro que la comunicación directa desde los clientes hacia los microservicios presenta una serie de inconvenientes y desafíos que quisieran resolverse. Una opción posible es implementar un *API Gateway*, este componente de software actúa como una fachada centralizando las peticiones de los clientes y sirviendo para los mismos una interfaz unificada. Implementando este gateway, que se ubica entre los clientes y los microservicios, a los clientes se les brinda un único punto para conectarse (ya no pueden conectarse más a los microservicios directamente, sino únicamente mediante el API Gateway).

La interacción entre el cliente y los microservicios ocurrirá en la siguiente secuencia:

1. El cliente (una aplicación web, por ejemplo) envía la petición a la URL del API Gateway
2. El API Gateway analiza la petición recibida de acuerdo a su configuración. En base a su configuración decidirá si debe aplicarse alguna acción sobre la petición o no y procederá a *rutear* la petición hacia la URL del microservicio correspondiente.
3. El API Gateway recibe la respuesta del microservicio y la analiza de acuerdo a su configuración. Decide si debe realizar alguna acción sobre la respuesta (por ejemplo, agregar algún header) y retorna la respuesta al cliente.

Queda claro, entonces, que el API Gateway trabaja como un intermediario que presenta una fachada uniforme para los clientes y resuelve hacia que microservicio(s) debe enviarse la petición sin necesidad de que el cliente conozca la estructura interna del backend.

Pero, además de realizar el ruteo y presentar una interfaz uniforme, ¿qué más puede hacer el API Gateway?, ¿Qué otro inconveniente, de los mencionados, permite resolver?. En la secuencia de interacción

previamente descrita se puede leer que el gateway *decide, en base a su configuración, si hay alguna acción que aplicar sobre la petición*, una acción posible sobre la petición podría ser, por ejemplo, revisar si la misma contiene un header de autorización e invocar al servicio de autorización antes de rutear la petición al microservicio de destino. Otro ejemplo podría ser revisar la dirección IP del cliente para determinar si la misma viene de un país habilitado para utilizar los servicios del backend. También pueden recolectarse aquí logs y trazas, convertir algún protocolo o implementar un caché de peticiones. En síntesis, al ser un punto que centraliza todas las peticiones de los clientes, aquellos aspectos y políticas que atraviesan a los distintos microservicios pueden ser aplicados en este componente.

Responsabilidades típicas

Ruteo de peticiones

Se debe recordar que el API Gateway es el punto de entrada de la aplicación y, por lo tanto, su responsabilidad principal es poder "rutear" o redirigir las peticiones que recibe hacia el microservicio correcto. Para poder realizar su tarea, el gateway debe tener configurado un "mapa de rutas" que explique cuáles son las rutas que debe seguir para redireccionar las peticiones. Para entender que significa esta configuración es de utilidad revisar el siguiente ejemplo:

URL a la que accede el cliente	URL a la que se reenvía la petición
https://url_api_gw/api/v1/personas/123	http://url_servicio_personas:8080/v1/personas/123
https://url_api_gw/api/v1/articulos/7588	http://url_servicio_articulos:8081/v1/articulos/7588

El cliente tiene un único punto de acceso, que en el ejemplo es *https://url_api_gw/api/v1* y el gateway definirá en base a su configuración a qué microservicio reenviar la petición. En el ejemplo, las peticiones a */personas* se reenvían al microservicio de personas y las peticiones a */articulos* se reenvían al microservicio de artículos. Lo que responda el microservicio es lo que responderá el GW a su cliente.

Ruteo dinámico

Hay situaciones donde se requiere realizar un ruteo más inteligente que únicamente asociar una URL con otra. Podría pensarse, por ejemplo, en que si la petición contiene un cierto header entonces realizar una redirección particular. En estos casos se dice que el ruteo es dinámico, en contraposición al ruteo estático ejemplificado anteriormente.

Agregación / Composición

Otra de las responsabilidades principales de un API Gateway es liberar al cliente de la necesidad de realizar múltiples invocaciones para obtener datos relacionados entre sí.

Pensando en la aplicación web de una clínica que brinda de manera on-line los resultados de los estudios de sus pacientes, esta podría requerir mostrar (en la misma página) los datos del paciente y el listado de los resultados disponibles. En este caso, si los datos del paciente son administrados por un microservicio y los resultados de los estudios por otro, la aplicación web debe realizar dos llamadas para poder mostrar la información requerida.

En base al ejemplo anterior, ¿cómo podría hacer el API Gateway para evitar que el cliente requiera dos llamadas al backend?. La respuesta es que sea el gateway quién reciba una única petición y se encargue de realizar las dos peticiones a los servicios correspondientes, para luego agregar los resultados en una única respuesta para su cliente.

Cabe preguntarse cuál es el beneficio de esta solución si, en definitiva, igual se hacen dos llamadas a los dos microservicios. Una ventaja de esta solución (que no es menor), es que el API Gateway se encuentra en la misma infraestructura que los microservicios y conectado a los mismos mediante una red interna de alta velocidad, a diferencia del cliente del backend (por ejemplo, un web browser) que se comunica con el mismo, típicamente, a través de internet donde las demoras son mayores.

Autenticación / Autorización

Se estudió en los apuntes 16 y 17 al respecto de la seguridad en el Backend. Siendo el *API Gateway* un punto de acceso a los distintos microservicios, es posible configurar el gateway para que las peticiones se ruteen únicamente si se cumple con los requisitos de autenticación y autorización.

Si, por ejemplo, se decidió OAuth2 o algún tipo de autorización basada en tokens, el *API Gateway* podría actuar como primer punto de control, revisar que el encabezado correcto esté presente en las peticiones y realizar la validación del token.

Rate limiting

Significa limitar la tasa de peticiones permitida para los clientes del API Gateway (por ejemplo, no se permiten más de 1000 peticiones/segundo). Si bien esto pareciera contrario a la expectativa de que el backend responda a tantas peticiones como sea posible, se debe tener en cuenta que hay situaciones donde este límite puede ser deseable, por ejemplo:

- *Protegerse de ataques de denegación de servicio*

Este tipo de ataques consiste en enviar la mayor cantidad de peticiones posibles hacia el backend de manera de agotar sus recursos e impedir que brinde servicio a los clientes del mismo. Limitar la cantidad de peticiones que se aceptan es favorable en este contexto.

- *Evitar problemas de performance*

Independientemente de que se haya elegido una arquitectura basada en microservicios o no, se debe tener en cuenta que los servicios que conforman el backend están desplegados sobre una infraestructura que cuenta con ciertas capacidades. Cuanto mayor sea la cantidad de peticiones que el backend deba atender en un momento dado, mayores serán los recursos a utilizar y, si estos resultan excesivos, la performance general disminuirá: cada petición demorará más tiempo en ser respondida (en el mejor de los casos).

Si no existe un límite en la tasa de peticiones que puede realizar un cliente podría pasar que uno de los clientes invoque los servicios del backend a una tasa muy elevada impactando en la respuesta del backend a los demás clientes.

Si bien este punto parece similar al anterior, debe quedar claro que el hecho de que un cliente realice peticiones a una tasa elevada no significa que esté tratando de realizar un ataque sobre el backend.

Caché

Algo que debe tenerse en cuenta es que el backend (compuesto por microservicios, o no) no es una pieza de software aislada que no tiene relación con ningún otro componente. Por el contrario, lo más común es que exista un front-end (o varios) que se comunican con el backend para conformar una aplicación y brindar funcionalidades al usuario. Por este motivo, cuanto más demore el backend en responder a las peticiones peor será la experiencia del usuario final de la aplicación y esto es algo que debe evitarse.

Una estrategia es utilizar lo que se conoce como un *caché*. Un caché es una copia de los datos que se acceden más frecuentemente, realizada en una memoria de (muy) rápido acceso.

Un ejemplo que puede resultar de utilidad para entender este concepto podría ser el de un API que brinde información sobre las estaciones de servicio que se encuentran en una ciudad (Nombre, Ubicación, bandera, combustibles disponibles, etc.). Sin un caché, cada vez que se le consulta al API sobre una estación, los datos deben ir a buscarse (probablemente) a la base de datos, lo que es una operación costosa en tiempo y recursos. Si, en cambio, existiera un caché con los datos de las estaciones, ante la primera consulta de una estación, los datos se buscarían en la base de datos (pagando el costo en tiempo y recursos) y se dejaría una copia de los mismos en memoria RAM, y las próximas veces que se consulten los datos de la misma estación, los mismos se buscarían directamente de la memoria RAM cuyo acceso es *mucho* más rápido que el acceso a una base de datos.

Una responsabilidad típica de un API Gateway es la de realizar un caché con los datos respondidos por ciertas APIs. En general, lo que hará el gateway es revisar si respondió previamente a una petición que haya sido realizada a una url y con exactamente los mismos parámetros y responderá directamente los datos del caché en lugar de invocar al microservicio correspondiente.

Debe quedar claro que esta explicación no es exhaustiva en lo más mínimo, sino una explicación muy general sobre el funcionamiento de un caché. Para implementar correctamente un caché hay muchos aspectos a considerar, por ejemplo: ¿qué datos van a ser "cacheados"?, ¿cuál es la frecuencia de actualización de esos datos?, ¿cómo se actualiza el caché?, ¿se utilizará algún producto para implementar el caché (redis, memcached, infinispan, etc.)?, ¿cuánta memoria ocupará este caché?.

Recolección de Métricas

Para poder monitorear correctamente la solución del backend es crucial contar con métricas que puedan brindar una idea de su comportamiento. Estas métricas pueden ser muy variadas (y no es el objetivo de estas notas una explicación muy detallada) pero algunos ejemplos incluyen: Cantidad de peticiones, Tiempos de respuesta, *Caché misses", Status HTTP devueltos, etc.

Siendo el gateway el punto de acceso de los clientes al backend, es razonable que esta recolección se realice en este punto y no en cada uno de los microservicios que componen al backend.

Algunas posibles desventajas

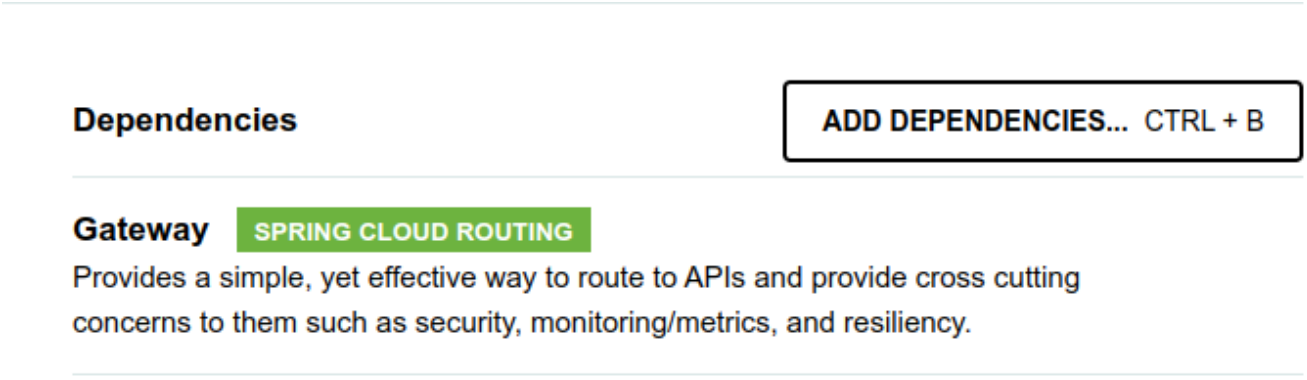
- Posible introducción de un SPoF
- Es otro componente a programar/configurar/mantener
- Introduce mayores tiempos de repuesta
- Si su desarrollo/configuración depende de un único equipo, puede introducir cuellos de botella en el desarrollo del backend como conjunto

Implementación del API Gateway

En línea con el resto de los apuntes de la materia, en este se utilizará SpringBoot para la implementación del API Gateway. Como es frecuente, dentro del ecosistema de Spring existe un proyecto que simplifica la construcción de un API Gateway. Este proyecto se llama *Spring Cloud Gateway*.

Creación del proyecto

Si se utiliza *Spring Initializer*, se debe buscar la dependencia **Gateway**, como se muestra a continuación:



La dependencia que agregará initializer será:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
```

Configuración

Al igual que en otros proyectos que utilizan SpringBoot, *Spring Cloud Gateway* tiene autoconfiguración (con configuraciones por defecto) y esta configuración puede personalizarse en base al archivo de configuración o proveyendo una nueva implementación de un bean en particular. En este apunte, esta configuración se hará mediante la creación de un bean.

Ruteo

Para la configuración del ruteo que va a realizar el *API Gateaway* el bean a proveer es una instancia que implemente *RouteLocator*. *Spring* inyectará (si así se solicita) un objeto *builder* para hacer esta tarea más sencilla. La clase de configuración puede iniciar de esta manera:

```
@Configuration
public class GWConfig {

    @Bean
    public RouteLocator configurarRutas(RouteLocatorBuilder builder){
        return builder.routes().build();
    }

}
```

Tendría poco valor construir un objeto *RouteLocator* sin rutas, por lo que será necesario agregar definiciones de rutas a este objeto. Para esto debe comprenderse que el método *routes()* de *RouteLocatorBuilder* devuelve un objeto builder también; este builder provee métodos para construír rutas (y estos métodos también devuelven al builder), haciendo que las llamdas puedan encadenarse y hacer más compacto y legible el código.

El objeto builder que permite crear una ruta provee los siguientes métodos:

```
public Builder route(Function<PredicateSpec, Buildable<Route>> fn);
public Builder route(String id, Function<PredicateSpec, Buildable<Route>>
fn);
```

Entre ambos métodos la diferencia es que en el segundo, se puede asignar un *id* a la ruta pero, dejando de lado este detalle, se puede ver que para construír una ruta es necesario pasar como argumento una Función (la interfaz funcional *Function*) cuyo parámetro es un *PredicateSpec* y que debe retornar una instancia *Buildable<Route>*. No es necesario detenerse en la estructura de estas clases e interfaces, pero sí es útil notar lo siguiente: para crear una ruta se debe especificar un predicado que permita determinar si la ruta debe activarse o no. Una ruta muy simple podría configurarse de la siguiente manera:

```
@Configuration
public class GWConfig {

    @Bean
    public RouteLocator configurarRutas(RouteLocatorBuilder builder) {
        return builder.routes()
                .route(p -> p
                        .path("/get")
                        .uri("https://postman-echo.com")
                )
                .build();
    }

}
```

Nota: en https://postman-echo.com se encuentra una serie de endpoints que sirven para realizar pruebas de clientes REST. Simplemente recibe peticiones y devuelve detalles de la misma (como si fuera un eco). De esta manera, se puede revisar si se está realizando correctamente la invocación, si llegan los parámetros, etc.

Cuando el GW reciba una petición a *http://localhost:8082/get*, la misma se reenviará a *https://postman-echo.com* y lo que responda este último, es lo que responderá el gateway. A continuación se puede ver un ejemplo de esto:

Petición

```
curl -v "http://localhost:8082/get?param1=prueba&param2=gateway"
*   Trying 127.0.0.1:8082...
* Connected to localhost (127.0.0.1) port 8082 (#0)
> GET /get?param1=prueba&param2=gateway HTTP/1.1
> Host: localhost:8082
> User-Agent: curl/7.81.0
> Accept: */*
```

Respuesta

```
< HTTP/1.1 200 OK
< Date: Fri, 20 Oct 2023 23:04:37 GMT
< Content-Type: application/json; charset=utf-8
< Content-Length: 522
< ETag: W/"20a-qseG9I9t14PBzPl3n1dZsCBgC14"
< set-cookie:
sails.sid=s%3A3sdLDoVxPdVfIqlAoRWCr3uVDRgMVcxJ.BsPJGUJNuqUhG9R9fLRL0gZh8Pl
IlxyyMvlDSi47S6o; Path=/; HttpOnly
<
{
  "args": {
    "param1": "prueba",
    "param2": "gateway"
  },
  "headers": {
    "x-forwarded-proto": "https",
    "x-forwarded-port": "443",
    "host": "postman-echo.com",
    "x-amzn-trace-id": "Root=1-65330785-79250f6525522e521763f4d3",
    "content-length": "0",
    "user-agent": "curl/7.81.0",
    "accept": "*/*",
    "forwarded":
"proto=http;host=\"localhost:8082\";for=\"127.0.0.1:46534\"",
    "x-forwarded-host": "localhost:8082"
  },
  "url": "https://postman-echo.com/get?param1=prueba&param2=gateway"
* Connection #0 to host localhost left intact
}
```

Aunque muy básica, esta primera configuración de ruteo permite ver en acción a una de las principales funcionalidades de un *API Gateway*.

Predicados

Los predicados, entonces, definirán si la petición que recibe el *API Gateway* activará o no un ruteo. Si bien el predicado del ejemplo anterior se realizó en base al path de la petición, no es la única posibilidad: Se

puede analizar coincidencia con cualquier elemento de la petición HTTP original (el método HTTP, el host al que se envía, headers, etc.).

Algunos métodos disponibles en los predicados de *Spring Cloud Gateway* son:

- **path():** Revisa la coincidencia con el path al que se realizó la petición
- **method():** Revisa la coincidencia del método HTTP invocado
- **host():** Revisa la coincidencia del host al que se realiza la petición
- **before() / after():** Revisa si la petición se hizo antes/después de un cierto momento.
- **query():** Revisa la coincidencia con un parámetro de la url.
- **header():** Revisa la coincidencia con algún encabezado de la petición.

Adicionalmente pueden combinarse entre ellos mediante el uso de **and()**/**or()**, como se muestra a continuación:

```
@Configuration
public class GWConfig {

    @Bean
    public RouteLocator configurarRutas(RouteLocatorBuilder builder) {
        return builder.routes()
                // Si llega una petición a http://localhost:8082/get, la
misma se
                // envía a https://postman-echo.com/get
                .route(p -> p
                        .host("localhost:8082")
                        .and()
                        .path("/get")
                        .and()
                        .before(ZonedDateTime.of(2023, 12, 31, 23, 59, 59,
0, ZoneId.systemDefault()))
                        .uri("https://postman-echo.com")
                )
                .build();
    }

}
```

Aquí, para que la petición sea ruteada, debe suceder que el host sea *localhost:8082*, además el path */get* y finalmente que la petición se realice antes del final del año 2023.

De acuerdo a lo explicado en las primeras secciones, construír la configuración de ruteo permitirá, entre otras cosas, que el *API Gateway* actúe como un único punto de entrada y reenvíe las peticiones al microservicio correcto.

Filtros

Los filtros permiten realizar modificaciones sobre la petición que se está reenviando al microservicio, o bien a la respuesta de este último. Si el filtro actúa *antes* de reenviar la petición del *API Gateway* al microservicio,

se trata de un *Pre-Filtro*; si, en cambio actúa sobre la respuesta del microservicio (*después* de que se envió la petición al microservicio y *antes* de retornar la respuesta al cliente) se trata de un *Post-Filtro*.

Los filtros son de gran utilidad ya que permiten, por ejemplo, agregar encabezados a la petición y/o a la respuesta. En una arquitectura de microservicios esto puede ser especialmente valioso.

Si bien los filtros se pueden programar, *Spring Cloud Gateway* cuenta con una serie de filtros ya programados para los usos más comunes como: Agregar/Modificar encabezados en las peticiones y en las respuestas, generar redirecciones, modificar la ruta de la petición, setear el código de repuesta HTTP, etc.

Utilizando la clase de configuración, un filtro se puede agregar a una ruta de la siguiente manera:

```
@Configuration
public class GWConfig {

    @Bean
    public RouteLocator configurarRutas(RouteLocatorBuilder builder) {
        return builder.routes()
                // Si llega una petición a http://localhost:8082/echo, la
misma se
                // envía a https://postman-echo.com/get
                .route(p -> p
                        .path("/echo")
                        .filters(f -> f.rewritePath("/echo", "/get"))
                        .uri("https://postman-echo.com")
                )
                .build();
    }

}
```

En este caso, se reemplaza el path original */echo* por */get*. De esta manera, si se invoca al *API Gateway* utilizando *http://localhost:8082/echo* la petición se reenviará a *https://postman-echo.com/get*.

Desde luego que se pueden agregar tantos filtros como se desee. El código a continuación muestra un ejemplo donde se agregan dos filtros (*RewritePath* y *SetRequestHeader*):

```
@Configuration
public class GWConfig {

    @Bean
    public RouteLocator configurarRutas(RouteLocatorBuilder builder) {
        return builder.routes()
                // Si llega una petición a http://localhost:8082/echo, la
misma se
                // envía a https://postman-echo.com/get
                .route(p -> p
                        .path("/echo")
                        .filters(f -> f
                                .rewritePath("/echo", "/get")
```

```
                                .setRequestHeader("User-Agent",
"SpringCloudGateway")
                        )
                        .uri("https://postman-echo.com")
                )
                .build();
    }
}
```

Uniendo las partes

En esta sección se procederá a implementar un *API Gateway* utilizando como ejemplo concreto dos microservicios que se presentaron en el **Apunte 14: Spring Data**, y que se puede encontrar en:

```
https://labsys.frc.utn.edu.ar/gitlab/backend-app/alumnos/contenido/semana-11.git
```

Bajo la ruta */ejemplos/entradas-kempes*. Se trata de dos microservicios qeu trabajan en el contexto de una aplicación que permite emitir entradas nominadas para eventos que se realizan en el Estadio Kempes. Estos dos microservicios son:

- **entradas**: Servicio que se encarga de la emisión de las entradas
- **personas**: Servicio que maneja los datos de las personas registradas en el sistema

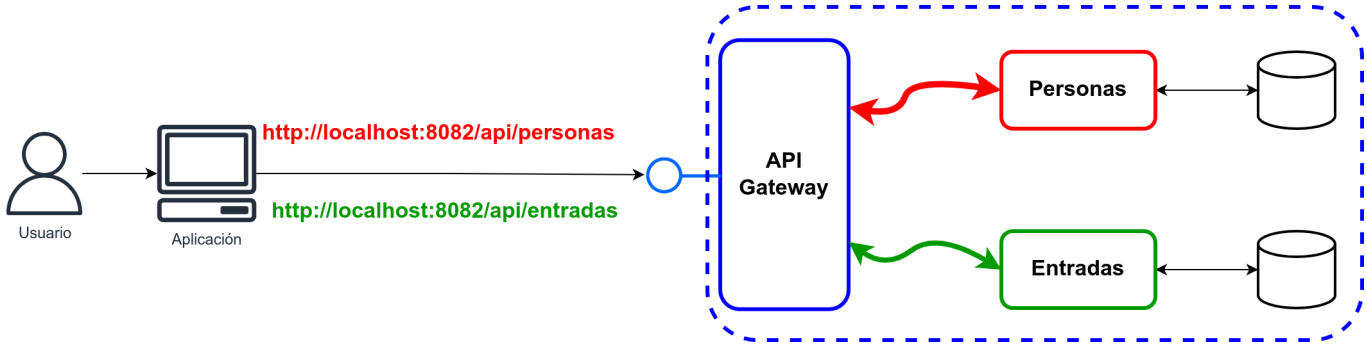
En la implementación de esta sección, se desarrollará un *API Gateway* que hará de punto de entrada para la aplicación. A partir de este punto de entrada y, en base al path al que se desee acceder, ruteará las peticiones hacia uno u otro microservicio. Adicionalmente, se implementará el control de autorización para que los endpoints puedan ser accedidos únicamente por quienes están autorizados para hacerlo.

Los valores de configuración para esta implementación serán los siguientes:

Para el proyecto *API Gateway*:

- **apunte-api-gw-kempes.url-microservicio-personas** = http://localhost:8083
- **apunte-api-gw-kempes.url-microservicio-entradas** = http://localhost:8084

La estructura será la siguiente:



Ejecución de los microservicios

Se mencionó que hay dos microservicios, el de **entradas** y el de **personas**. Estos dos microservicios son autónomos y cada uno de ellos es un programa/proceso distinto. En esta implementación, se asume que los servicios están configurados de la siguiente manera:

- **Personas:** Escucha en el puerto 8083 y brinda los siguientes endpoints:
  - **GET** */api/personas*. Devuelve todas las personas registradas
  - **GET** */api/personas/{id}*. Devuelve una persona buscada por ID
  - **GET** */api/personas/nombre/{nombre}*. Busca personas por nombre.
  - **POST** */api/personas*. Crea una nueva persona
  - **PUT** */api/personas/{id}*. Actualiza una persona
  - **DELETE** */api/personas/{id}*. Elimina a una persona
- **Entradas:** Escucha en el puerto 8084 y brinda los siguientes endpoints:
  - **POST** /api/entradas. Crea una nueva entrada.

Para comunicarse con esos servicios, hasta este momento, habría que conocer la uri de dos servicios de manera separada.

Configuración del API Gateway

Se deberá crear un proyecto con las características y dependencias estudiadas previamente. Para esto se puede utilizar *Spring Initializer*, y el archivo **pom.xml** de esta implementación podría ser como el siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.1.5</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>ar.edu.utn.bda</groupId>
    <artifactId>apunte-api-gw-kempes</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>API GW Kempes</name>
    <description>Ejemplo de API GW Entradas Kempes</description>
    <properties>
        <java.version>17</java.version>
        <spring-cloud.version>2022.0.4</spring-cloud.version>
    </properties>
    <dependencies>

        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-gateway</artifactId>
        </dependency>

        <dependency>
```

```
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <optional>true</optional>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>

    </dependencies>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>${spring-cloud.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

Luego es necesario crear las configuraciones del gateway como se estudió en la sección anterior. Para el ejemplo de esta aplicación, una configuración probable para las rutas es como la siguiente:

```
@Configuration
@EnableWebFluxSecurity
public class GWConfig {

    @Bean
    public RouteLocator configurarRutas(RouteLocatorBuilder builder,
                                        @Value("${apunte-api-gw-
kempes.url-microservicio-personas}") String uriPersonas,
                                        @Value("${apunte-api-gw-
kempes.url-microservicio-entradas}") String uriEntradas) {
        return builder.routes()
```

```
                // Ruteo al Microservicio de Personas
                .route(p -> p.path("/api/personas/**").uri(uriPersonas))
                // Ruteo al Microservicio de Entradas
                .route(p -> p.path("/api/entradas/**").uri(uriEntradas))
                .build();

    }
}
```

Esta porción de la configuración genera las rutas necesarias para ambos microservicios. Cuando el gateway reciba una petición a *http://ruta_gateway/api/personas*, la misma se reenviará a *http://ruta_microservicio_personas/api/personas*. En términos concretos de este ejemplo:

URL a la que accede el cliente	URL a la que se reenvía la petición
GET https://localhost:8082/api/personas/123	**GET** http://localhost:8083/personas/123
POST https://localhost:8082/api/personas	**POST** http://localhost:8083/personas
POST https://localhost:8082/api/entradas	**POST** http://localhost:8084/api/entradas

Nota: Se destaca que para acceder al GW se lo hace conectándose al puerto 8082 y luego el GW puede reenviar esas peticiones a los puertos 8083 y 8084 dependiendo de qué servicio deba accederse. Aquí se ve claramente lo que significa que el GW sea un único punto de entrada para la aplicación.

Apéndice 1: Invocación a un servicio

En muchos casos será necesario que una aplicación (un microservicio, por ejemplo) se comunique con otra y, posiblemente, lo haga mediante una invocación a una API ReST. A continuación se indica como esto puede ser realizado utilizando herramientas de *Spring*. Debe aclararse que la alternativa que se estudiará a continuación no es, de ninguna manera, la única; existen clientes HTTP incluídos en Java desde Java 11 y muchas librerías como *Apache HTTP Client* o, incluso, otros proyectos de *Spring* como *WebFlux*.

RestTemplate

GET

Una de las varias maneras que existen para realizar una petición REST es el uso de la clase **RestTemplate** provista por Spring. Un ejemplo se puede observar a continuación:

```
public void invocarServicio() {
      // Creación de una instancia de RestTemplate
      try {
          // Creación de la instancia de RequestTemplate
          RestTemplate template = new RestTemplate();
          // Se realiza una petición a
http://localhost:8082/api/personas/{id}, indicando que id vale 1 y que la
          // respuesta de la petición tendrá en su cuerpo a un objeto del
tipo Persona.
```

```
            ResponseEntity<Persona> res = template.getForEntity(
                    "http://localhost:8082/api/personas/{id}",
    Persona.class, 1
            );

            // Se comprueba si el código de repuesta es de la familia 200
            if (res.getStatusCode().is2xxSuccessful()) {
                log.debug("Persona obtenida correctamente: {}",
    res.getBody());
            } else {
                log.warn("Respuesta no exitosa: {}", res.getStatusCode());
            }

        } catch (HttpClientErrorException ex) {
            // La repuesta no es exitosa.
            log.error("Error en la petición", ex);
        }
    }
```

Al analizar el código se puede ver lo siguiente:

- Se construye una instancia de *RestTemplate*
- El método *getForEntity()* permite realizar una petición get esperando una entidad como respuesta (en otras palabras, se espera que la respuesta sea interpretada y devuelva un objeto).
- *getForEntity()* Espera: 1) la ruta donde se hará la petición, que puede incluír variables de path y query parameters; 2) El tipo al que pertenece la respuesta (Persona, en este caso); 3) Opcionalmente los valores de las variables de la uri (el id, en este caso).
- La respuesta no es directamente una persona, sino un objeto del tipo *ResponseEntity<Persona>*. Esto permite acceder, por ejemplo, al código de repuesta y otros detalles de la respuesta y, además, al objeto contenido en la misma si existiera.
- En caso de que la repuesta no fuera exitosa, *getForEntity()* arroja una excepción.
- Existe un método llamado *getForObject()* Similar a *getForEntity()* pero devuelve directamente el objeto y no el código de respuesta. En el ejemplo, devolvería una *Persona* y no *RequestEntity<Persona>*.

POST

También pueden realizarse peticiones **POST**. En este caso se puede enviar una entidad en el body de la petición de la siguiente manera:

```
public void crearPersona(Persona p) {
    // Creación de una instancia de RestTemplate
    try {
        // Creación de la instancia de RequestTemplate
        RestTemplate template = new RestTemplate();

        // Creación de la entidad a enviar
        HttpEntity<Persona> entity = new HttpEntity<>(p);
```

```
        // respuesta de la petición tendrá en su cuerpo a un objeto del
tipo Persona.
        ResponseEntity<Persona> res = template.postForEntity(
                "http://localhost:8082/api/personas", entity,
Persona.class
        );
    // Se comprueba si el código de repuesta es de la familia 200
        if (res.getStatusCode().is2xxSuccessful()) {
            log.debug("Persona creada correctamente: {}", res.getBody());
        } else {
            log.warn("Respuesta no exitosa: {}", res.getStatusCode());
        }

    } catch (HttpClientErrorException ex) {
        // La repuesta no es exitosa.
        log.error("Error en la petición", ex);
    }
}
```

El método utilizado aquí es *postForEntity()* cuyos parámetros son: 1) La uri a invocar; 2) La entidad que se quiere enviar; 3) El tipo que se espera como respuesta.

Al igual que existía *getForEntity()* y *getForObject()*, con iguales comentarios existe *postForObject()*.

PUT/PATCH/DELETE

RestTemplate incluye, también, los métodos *put()*, *patchForObject()* y *delete()* que permiten invocar estos métodos HTTP. Un ejemplo se muestra a continuación:

```
String uri = "http://localhost:8082/api/personas/{id}";
// PUT - Actualizar la persona con ID = 1
template.put(uri, p, 1);
// PATCH - Actualizar la persona con ID = 1 (La petición devuelve una
Persona)
template.patchForObject(uri, p, Persona.class,1);
// DELETE - Eliminar a la persona con ID = 1
template.delete(uri, 1);
```

Otras Soluciones de API Gateway fuera del ecosistema Spring

En el ecosistema de microservicios, los **API Gateway dedicados** se convirtieron en componentes fundamentales para gestionar de forma centralizada las responsabilidades comunes del backend: ruteo, seguridad, control de tráfico, observabilidad y transformación de peticiones. A diferencia de un gateway embebido en una aplicación (como *Spring Cloud Gateway*), los productos dedicados se diseñan para actuar como **puerta de entrada general** al conjunto de microservicios, incluso cuando estos están implementados en distintos lenguajes o frameworks.

Rol y beneficios de los API Gateway dedicados

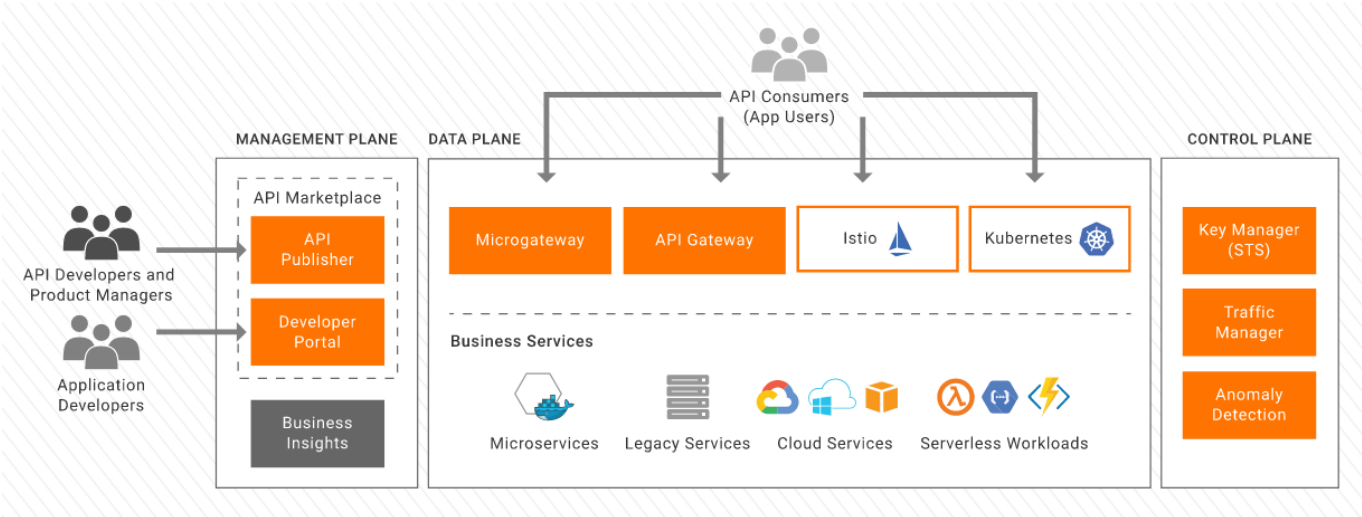
Los gateways dedicados resuelven de manera homogénea muchos de los problemas típicos en arquitecturas distribuidas:

- **Ruteo dinámico y balanceo de carga:** permiten definir rutas hacia múltiples microservicios y aplicar estrategias inteligentes de distribución de tráfico.
- **Autenticación y autorización centralizadas:** integran mecanismos de seguridad (OAuth2, JWT, API keys, OpenID Connect) sin replicar lógica en cada servicio.
- **Control de acceso y rate limiting:** previenen abusos o sobrecargas, aplicando límites de consumo por usuario o aplicación.
- **Transformación y agregación de respuestas:** facilitan la interoperabilidad entre APIs heterogéneas, transformando formatos o combinando resultados.
- **Monitoreo y observabilidad:** exponen métricas y logs centralizados para analizar el rendimiento y comportamiento global de las APIs.

En resumen, estos productos permiten **abstraer las preocupaciones transversales del backend** y ofrecer una vista unificada hacia el exterior, reduciendo el acoplamiento entre servicios y clientes.

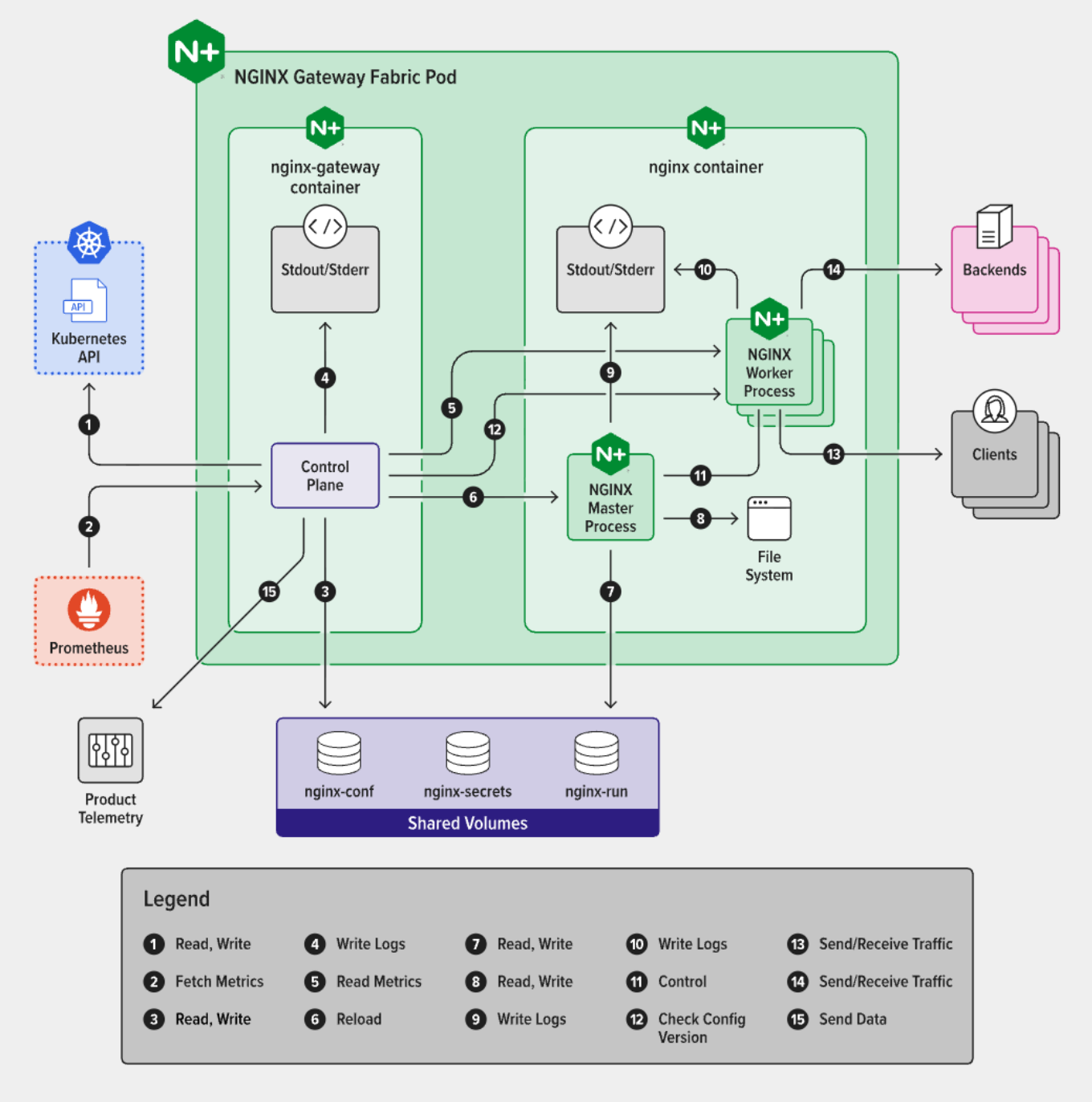
Principales alternativas Open Source

⚙️ WSO2 API Manager



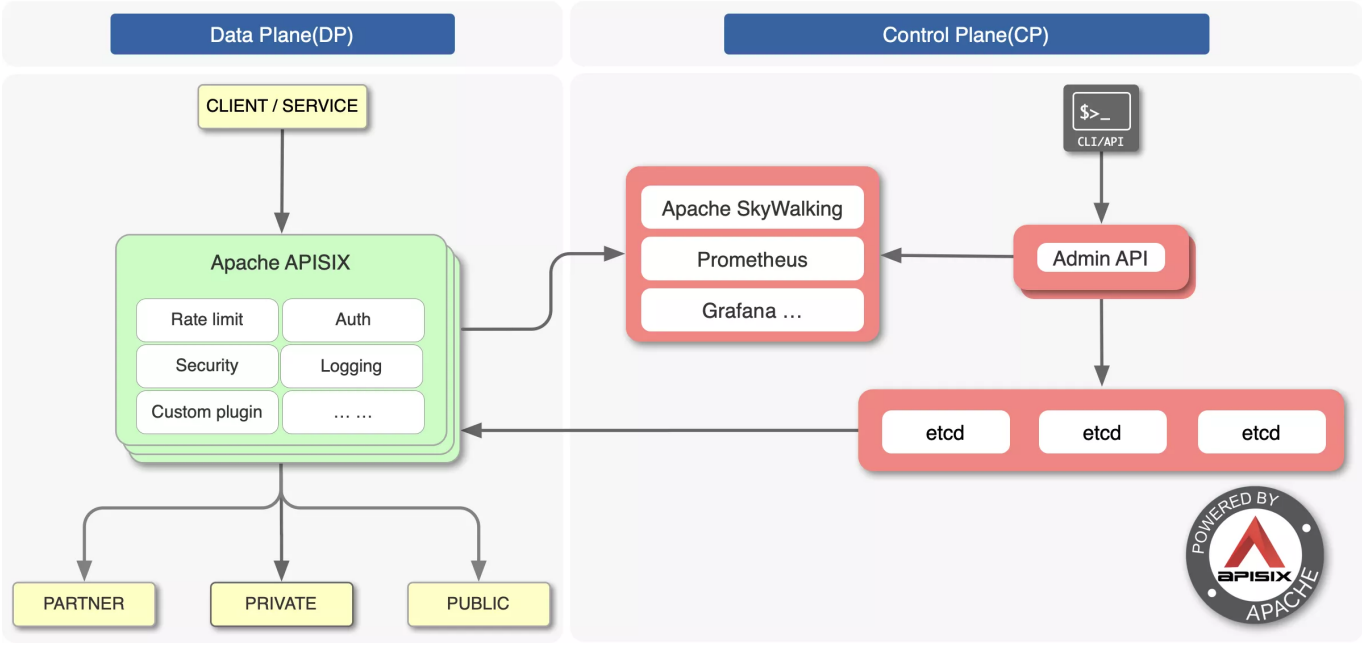
- **Lenguaje base:** Java
- **Características:** combina API Gateway, portal de desarrolladores y consola de administración en una sola plataforma.
- **Ventajas:** interfaz gráfica completa, soporte de versiones y políticas de seguridad; integra analítica avanzada y monetización.
- **Uso típico:** organizaciones que requieren gobierno de APIs con soporte multiambiente.
- **Sitio oficial:** https://wso2.com/api-management/

🌐 NGINX as API Gateway



- **Lenguaje base:** C
- **Características:** construido sobre el conocido servidor NGINX, soporta ruteo, balanceo, autenticación, caching y políticas configurables mediante archivos declarativos.
- **Ventajas:** altísimo rendimiento, footprint reducido y gran estabilidad; ampliamente adoptado por su facilidad de integración con entornos existentes.
- **Sitio oficial:** https://docs.nginx.com/nginx-gateway-fabric

🧩 Apache APISIX



- **Lenguaje base:** Lua + Nginx
- **Características:** arquitectura extensible por plugins, configuración dinámica mediante API REST o etcd, soporte para protocolos HTTP(S), gRPC y WebSocket.
- **Ventajas:** rendimiento muy alto, integración nativa con OpenTelemetry y múltiples mecanismos de autenticación. Ideal para entornos cloud nativos y despliegues híbridos.
- **Sitio oficial:** https://apisix.apache.org

Propuestas comerciales más difundidas

Además de las alternativas open source, existen productos administrados o de nivel empresarial que amplían las funcionalidades hacia la analítica, el control de políticas y la monetización de APIs:

- **Apigee (Google Cloud):** gateway corporativo con panel de monitoreo, analítica, control de cuota y monetización.
- **Amazon API Gateway (AWS):** servicio totalmente administrado con integración nativa a Lambda, EC2 y Cognito.
- **Azure API Management (Microsoft):** solución híbrida con soporte para caching, control de acceso y portal de desarrolladores.

Estos productos destacan por su escalabilidad y facilidad de integración con los ecosistemas cloud de sus proveedores, aunque suelen implicar costos asociados y menor flexibilidad de personalización frente a las soluciones open source.

Bibliografía y referencias

- **Spring Cloud Gateway Reference Documentation.** https://docs.spring.io/spring-cloud-gateway/reference/
- **Spring Cloud Project – GitHub Repository.** https://github.com/spring-cloud/spring-cloud-gateway
- **Apache APISIX Official Documentation.** https://apisix.apache.org/docs/
- **WSO2 API Manager Documentation.** https://wso2.com/api-management/documentation/
- **NGINX Gateway Fabric Documentation.** https://docs.nginx.com/nginx-gateway-fabric/
- **AWS API Gateway Developer Guide.** https://docs.aws.amazon.com/apigateway/

- **Apigee Documentation (Google Cloud).** https://cloud.google.com/apigee/docs
- **Microsoft Azure API Management Documentation.** https://learn.microsoft.com/en-us/azure/api-management/