Apunte de Clase 18 — 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`).