1. Interview-Style Opening

"Integrating with external APIs is a daily task in microservices. In the Java ecosystem, the way we make these calls has evolved significantly. I've moved from using the legacy `RestTemplate` to more modern, declarative approaches like **OpenFeign** for internal microservices, and **WebClient** or the new **RestClient** for high-performance, non-blocking scenarios."

2. Problem Understanding and Clarification

The question asks for the "possible ways" to consume REST APIs and my experience with "third-party clients."

Clarification of "Third Party":

- **Native/Spring Libraries:** `HttpURLConnection` (core Java), `RestTemplate`, `WebClient`, `RestClient`.
- **Third-Party Libraries:** Apache HttpClient, OkHttp, Retrofit.
- **Declarative Clients:** Spring Cloud OpenFeign.

Assumptions:

- We are working in a **Spring Boot** environment (standard for enterprise Java).
- We care about maintainability and resilience (timeouts, retries).

3. High-Level Approach (Evolution of Clients)

1. **The Legacy Way (`RestTemplate`):** Simple, synchronous, blocking. Good for quick scripts but deprecated for new development because it blocks threads.[1]
2. **The Reactive Way (`WebClient`):** Part of Spring WebFlux. Non-blocking, asynchronous, and high-performance. It's powerful but has a steep learning curve (Mono/Flux wrappers).[2]
3. **The Modern Standard (`RestClient` - Java 21+):** Introduced in Spring Boot 3.2. It offers the **fluent API** of WebClient but runs synchronously (on the Servlet stack). This is now the recommended replacement for RestTemplate.
4. **The "Magic" Way (`OpenFeign`):** A declarative client. You just write an interface, annotate it with `@GetMapping`, and Spring generates the implementation at runtime. I use this heavily for **inter-service communication.**[3]

4. Visual Comparison

Feature	**OpenFeign**	**RestClient (Spring 3.2+)**	**WebClient**
Style	Declarative (Interface)	Fluent / Functional	Reactive / Functional
Blocking	Yes (Sync)	Yes (Sync)	No (Async)
Best For	Internal Microservices	External 3rd Party APIs	High Concurrency / Streaming
Boilerplate	Low (Annotations only)	Medium	Medium

5. Java Code (Production-Quality)

I'll show two examples: **OpenFeign** (for clean internal calls) and **RestClient** (for robust external calls).

Example A: OpenFeign (Declarative)

```
// 1. Dependency: spring-cloud-starter-openfeign
// 2. Enable in Main: @EnableFeignClients

@FeignClient(name = "inventory-service", fallback = InventoryFallback.class)
public interface InventoryClient {

    @GetMapping("/api/v1/stocks/{sku}")
    StockResponse checkStock(@PathVariable("sku") String sku);
}

// Fallback for Resilience (Circuit Breaker)
@Component
class InventoryFallback implements InventoryClient {
    @Override
    public StockResponse checkStock(String sku) {
        return new StockResponse(sku, 0, false); // Default: Out of stock
    }
}
```

Example B: RestClient (Modern Spring Boot 3)

```
@Service
public class PaymentGatewayService {

    private final RestClient restClient;

    public PaymentGatewayService(RestClient.Builder builder) {
        this.restClient = builder
                .baseUrl("https://api.stripe.com/v1")
                .defaultHeader("Authorization", "Bearer sk_test_123")
                .build();
    }

    public PaymentResponse charge(PaymentRequest request) {
        return restClient.post()
                .uri("/charges")
                .contentType(MediaType.APPLICATION_JSON)
                .body(request)
                .retrieve()
                // Robust Error Handling
                .onStatus(HttpStatusCode::is4xxClientError, (req, res) -> {
                    throw new PaymentValidationException("Invalid Card");
                })
                .onStatus(HttpStatusCode::is5xxServerError, (req, res) -> {
                    throw new PaymentGatewayDownException("Stripe is down");
                })
                .body(PaymentResponse.class);
    }
}
```

6. Code Walkthrough (Line-by-Line)

- `@FeignClient(name = "inventory-service")`: Spring Cloud LoadBalancer uses this name to look up the service IP from Eureka/Consul. I don't need to hardcode URLs.[4]
- `fallback = InventoryFallback.class`: If the inventory service is down, Feign automatically catches the exception and calls my fallback class. This prevents cascading failures.
- `restClient.post()...retrieve()`: This fluent API is readable. Unlike `RestTemplate.exchange(...)`, I don't have to wrap things in `HttpEntity` repeatedly.
- `onStatus(...)`: This is a major improvement. In `RestTemplate`, handling 404s vs 500s required a messy `ResponseErrorHandler`. Here, it's a lambda.

7. How to Explain This to the Interviewer

"I have extensive experience with third-party clients.

For **internal microservices**, I exclusively use **OpenFeign**. It keeps my codebase clean because I'm just defining interfaces. It also integrates out-of-the-box with **Resilience4j** for circuit breaking.

However, for consuming **public APIs** (like Stripe or Google Maps), I prefer the new **Spring RestClient**. Feign can be a bit 'magic' and hard to debug when dealing with complex OAuth2 flows or non-standard JSON responses. RestClient gives me fine-grained control over headers and error parsing while remaining thread-safe and cleaner than the old RestTemplate."

8. Edge Cases and Follow-Up Questions

Q: OpenFeign is synchronous. Does that hurt performance?
*A: In a standard Tomcat (Servlet) app, one blocking call is fine. But if I need high throughput, I would switch to **WebClient** (Reactive) or use Feign with its **Async** capability (though that's experimental). For 99% of CRUD apps, blocking Feign is fast enough.*

Q: Have you used OkHttp directly?
A: Yes, in legacy projects where Spring wasn't available. OkHttp is great because of its connection pooling and interceptor chain, but in a Spring Boot app, I prefer wrapping it behind `RestClient` *to keep dependency injection consistent.*

⁂

1. https://www.baeldung.com/spring-webclient-resttemplate
2. https://digma.ai/restclient-vs-webclient-vs-resttemplate/
3. https://blog.dtdl.in/simplifying-microservices-communication-with-spring-cloud-openfeign-c9f65f6c15bf
4. https://www.geeksforgeeks.org/java/spring-cloud-openfeign-with-example-project/
5. https://docs.spring.io/spring-boot/reference/io/rest-client.html
6. https://www.javaspring.net/blog/java-call-rest-api/
7. https://www.reddit.com/r/SpringBoot/comments/191l7ya/a_comparison_between_restclient_webclient_and/
8. https://stackoverflow.blog/2020/03/02/best-practices-for-rest-api-design/
9. https://www.youtube.com/watch?v=ECAhXdFrBaE
10. https://www.linkedin.com/pulse/spring-cloud-openfeign-rest-client-boot-app-coditation-systems
11. https://www.geeksforgeeks.org/advance-java/spring-webclient-vs-resttemplate/
12. https://stackoverflow.com/questions/47974757/webclient-vs-resttemplate
13. https://www.reddit.com/r/SpringBoot/comments/1ijqu9q/restclient_vs_webclient_vs_resttemplate_using_the/
14. https://www.javacodemonk.com/retrofit-vs-feign-for-server-side-d7f199c4
15. https://www.youtube.com/watch?v=9oq7Y8n1t00