ChatGPT

Spring Boot Payment Gateway Architecture

We use the **Strategy pattern** to encapsulate each vendor's integration logic behind a common interface. Each vendor (A, B, C, etc.) implements a `PaymentStrategy` interface so they can be interchanged easily at runtime. New vendors simply provide a new `PaymentStrategy` implementation bean (annotated with `@Component`) without changing existing code [1] [2]. This follows the Open/Closed Principle: the system is *open for extension* (add new vendors) but *closed for modification* of existing classes [2] [3]. Because each vendor class implements the same interface, they remain loosely coupled (they share no code and only interact via the interface) [3].

Spring's dependency injection automatically discovers all vendor beans. For example, each implementation is annotated with `@Component` so it's picked up as a bean by Spring [4]. A central router service can then select the correct bean at runtime (e.g. via a lookup map of bean names) based on business logic (such as comparing rates). The router remains decoupled from specific vendors: it simply looks up the `PaymentStrategy` by key (vendor ID) and invokes it. This "pluggable" strategy setup means adding a new vendor only requires adding a new class implementing the interface (no code changes elsewhere).

For communication we use Spring WebFlux's **WebClient**. WebClient is a fully non-blocking, reactive HTTP client [5], which makes our calls to the external middleware high-performance and scalable. Spring Boot auto-configures a single shared `WebClient.Builder` bean that we can inject. We use this builder to create a `WebClient` instance per vendor (setting the vendor's base URL), but under the hood Spring shares the underlying HTTP resources for all WebClients [6] [7]. This ensures efficient, non-blocking I/O while allowing different base URLs. In summary, the design uses Spring DI, interfaces, and reactive WebClient to achieve loose coupling, open-closed extensibility, and high throughput [1] [5].

Key design choices: - Strategy Pattern – one `PaymentStrategy` interface for all vendors (decouples vendor-specific code) [1].
- Open/Closed Principle – new vendors are added by creating new classes (no modifications to existing code) [2].
- Spring DI and `@Component` beans – automatically discover vendor strategies [4].
- Non-blocking WebClient calls for middleware – single shared `WebClient.Builder` to create clients [6] [7].
- **Dynamic routing** – a router service uses business logic to pick the right strategy at runtime.

The following skeleton code illustrates these ideas in a Maven/Java17 Spring Boot project:

```
<!-- pom.xml -->
<project ...>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example.gateway</groupId>
  <artifactId>payment-gateway</artifactId>
  <version>0.0.1-SNAPSHOT</version>
```

```
  <properties>
    <java.version>17</java.version>
    <spring.boot.version>3.2.0</spring.boot.version>
  </properties>
  <dependencyManagement>
    <dependencies>
      <!-- Use Spring Boot parent for dependency management -->
      <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-dependencies</artifactId>
        <version>${spring.boot.version}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
  <dependencies>
    <!-- Use WebFlux starter for reactive WebClient -->
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-webflux</artifactId>
    </dependency>
    <!-- (Other dependencies like Lombok, etc. can be added as needed) -->
  </dependencies>
  <build>
    <plugins>
      <!-- Spring Boot Maven plugin to package the app -->
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>
```

```
// src/main/java/com/example/gateway/PaymentGatewayApplication.java
package com.example.gateway;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class PaymentGatewayApplication {
    public static void main(String[] args) {
        SpringApplication.run(PaymentGatewayApplication.class, args);
```

```
    }
}
```

```
// src/main/java/com/example/gateway/model/PaymentRequest.java
package com.example.gateway.model;

public class PaymentRequest {
    private String vendorId;  // e.g. "vendorA", "vendorB"
    private double amount;
    private String currency;
    // other payment details...

    // Getters and setters (or use Lombok @Data)
    public String getVendorId() { return vendorId; }
    public void setVendorId(String vendorId) { this.vendorId = vendorId; }
    public double getAmount() { return amount; }
    public void setAmount(double amount) { this.amount = amount; }
    public String getCurrency() { return currency; }
    public void setCurrency(String currency) { this.currency = currency; }
}
```

```
// src/main/java/com/example/gateway/model/PaymentResponse.java
package com.example.gateway.model;

public class PaymentResponse {
    private String transactionId;
    private String status;
    // other response fields...

    // Constructors, getters, setters
    public String getTransactionId() { return transactionId; }
    public void setTransactionId(String transactionId) { this.transactionId =
transactionId; }
    public String getStatus() { return status; }
    public void setStatus(String status) { this.status = status; }
}
```

```
// src/main/java/com/example/gateway/strategy/PaymentStrategy.java
package com.example.gateway.strategy;

import reactor.core.publisher.Mono;
import com.example.gateway.model.PaymentRequest;
import com.example.gateway.model.PaymentResponse;
```

```java
/**
 * Common interface for all payment vendor strategies.
 * Each implementation handles a specific vendor's API.
 */
public interface PaymentStrategy {
    Mono<PaymentResponse> pay(PaymentRequest request);
}
```

```java
// src/main/java/com/example/gateway/strategy/VendorAService.java
package com.example.gateway.strategy;

import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.client.WebClient;
import reactor.core.publisher.Mono;
import com.example.gateway.model.PaymentRequest;
import com.example.gateway.model.PaymentResponse;

/**
 * Vendor A integration (example). Uses a reactive WebClient call.
 */
@Component("vendorA")  // Bean name "vendorA" for lookup
public class VendorAService implements PaymentStrategy {

    private final WebClient webClient;

    // Inject the shared WebClient.Builder and set Vendor A's base URL
    public VendorAService(WebClient.Builder webClientBuilder) {
        this.webClient = webClientBuilder
            .baseUrl("https://api.vendorA.com")  // example base URL
            .build();
    }

    @Override
    public Mono<PaymentResponse> pay(PaymentRequest request) {
        // Call Vendor A's API (POST /pay endpoint, for example)
        return webClient.post()
                .uri("/pay")
                .bodyValue(request)
                .retrieve()
                .bodyToMono(PaymentResponse.class);
    }
}
```

```java
// src/main/java/com/example/gateway/strategy/VendorBService.java
package com.example.gateway.strategy;
```

```java
import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.client.WebClient;
import reactor.core.publisher.Mono;
import com.example.gateway.model.PaymentRequest;
import com.example.gateway.model.PaymentResponse;

/**
 * Vendor B integration (example). Different URL or API format.
 */
@Component("vendorB")
public class VendorBService implements PaymentStrategy {

    private final WebClient webClient;

    public VendorBService(WebClient.Builder webClientBuilder) {
        this.webClient = webClientBuilder
            .baseUrl("https://api.vendorB.com")
            .build();
    }

    @Override
    public Mono<PaymentResponse> pay(PaymentRequest request) {
        // Different API format can be handled here if needed
        return webClient.post()
                .uri("/processPayment")
                .bodyValue(request)
                .retrieve()
                .bodyToMono(PaymentResponse.class);
    }
}
```

```java
// src/main/java/com/example/gateway/strategy/VendorCService.java
package com.example.gateway.strategy;

import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.client.WebClient;
import reactor.core.publisher.Mono;
import com.example.gateway.model.PaymentRequest;
import com.example.gateway.model.PaymentResponse;

/**
 * Vendor C integration (example).
 */
@Component("vendorC")
public class VendorCService implements PaymentStrategy {
```

```
    private final WebClient webClient;

    public VendorCService(WebClient.Builder webClientBuilder) {
        this.webClient = webClientBuilder
            .baseUrl("https://api.vendorC.com")
            .build();
    }

    @Override
    public Mono<PaymentResponse> pay(PaymentRequest request) {
        return webClient.post()
                .uri("/execute")
                .bodyValue(request)
                .retrieve()
                .bodyToMono(PaymentResponse.class);
    }
}
```

```
// src/main/java/com/example/gateway/service/PaymentRouter.java
package com.example.gateway.service;

import org.springframework.stereotype.Component;
import reactor.core.publisher.Mono;
import com.example.gateway.strategy.PaymentStrategy;
import com.example.gateway.model.PaymentRequest;
import com.example.gateway.model.PaymentResponse;
import java.util.Map;

/**
 * Routes payment requests to the correct vendor strategy.
 */
@Component
public class PaymentRouter {

    private final Map<String, PaymentStrategy> strategies;

    // Spring injects a map of all PaymentStrategy beans (keys are bean names,
e.g. "vendorA")
    public PaymentRouter(Map<String, PaymentStrategy> strategies) {
        this.strategies = strategies;
    }

    public Mono<PaymentResponse> route(PaymentRequest request) {
        String vendorId = request.getVendorId();
        PaymentStrategy strategy = strategies.get(vendorId);
```

```
        if (strategy == null) {
            return Mono.error(new IllegalArgumentException("Unknown vendor: " +
vendorId));
        }
        // Call the chosen vendor's pay() method
        return strategy.pay(request);
    }
}
```

```
// src/main/java/com/example/gateway/controller/PaymentController.java
package com.example.gateway.controller;

import org.springframework.web.bind.annotation.*;
import reactor.core.publisher.Mono;
import com.example.gateway.service.PaymentRouter;
import com.example.gateway.model.PaymentRequest;
import com.example.gateway.model.PaymentResponse;

/**
 * Exposes a REST endpoint for processing payments.
 */
@RestController
@RequestMapping("/payments")
public class PaymentController {

    private final PaymentRouter router;

    public PaymentController(PaymentRouter router) {
        this.router = router;
    }

    @PostMapping
    public Mono<PaymentResponse> processPayment(@RequestBody PaymentRequest
request) {
        // Delegate to the router which selects the vendor
        return router.route(request);
    }
}
```

```
// src/main/java/com/example/gateway/config/WebClientConfig.java
package com.example.gateway.config;

import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Bean;
import org.springframework.web.reactive.function.client.WebClient;
```

```java
/**
 * Configuration for WebClient. Spring Boot auto-configures a WebClient.Builder bean,
 * but we define it here explicitly for clarity.
 */
@Configuration
public class WebClientConfig {

    @Bean
    public WebClient.Builder webClientBuilder() {
        return WebClient.builder();
    }
}
```

Notes:
- Each vendor service class is annotated with `@Component("vendorX")`, making its bean name equal to the vendor ID. The router injects a `Map<String, PaymentStrategy>` where the key is the bean name. This allows simple lookup by vendor ID.
- We use **reactive return types** (`Mono<PaymentResponse>`) so the Spring controller is non-blocking.
- The `WebClient.Builder` is a singleton bean; we call `.baseUrl(...)` and `.build()` in each service. According to Spring Boot docs, the builder shares its underlying HTTP resources across instances [7]. Because `WebClient` is immutable [6], we create one client per vendor, but they all reuse the same connection pool.

With this structure, adding a new vendor (say Vendor D) just requires creating `VendorDService` implementing `PaymentStrategy` and annotating it (e.g. `@Component("vendorD")`). No changes to the router or controller are needed. The reactive WebClient ensures the calls are asynchronous and efficient [5]. This design is **flexible**, **loosely coupled**, and **scalable** for future extensions.

Sources: We applied the Strategy pattern for interchangeable behavior [1] and adhered to the Open/Closed Principle [2] [3]. Spring's WebClient is fully non-blocking and high-performance [5], and Spring Boot's pre-configured `WebClient.Builder` allows sharing HTTP resources [7].

---

[1] [4] The Strategy Design Pattern With Spring Plugin | by Milena Lazarevic | Javarevisited | Medium
https://medium.com/javarevisited/the-strategy-design-pattern-with-spring-plugin-e99021c8f6eb

[2] [3] The Open/Closed Principle with Code Examples - Stackify
https://stackify.com/solid-design-open-closed-principle/

[5] WebClient :: Spring Framework
https://docs.spring.io/spring-framework/reference/web/webflux-webclient.html

[6] [7] java - Spring webclient, how many instances? - Stack Overflow
https://stackoverflow.com/questions/53197085/spring-webclient-how-many-instances