ChatGPT

Payment Gateway Microservice (Spring WebFlux Skeleton)

This skeleton defines a **Spring Boot 3 / WebFlux** microservice to handle payment processing via multiple vendors, calling a middleware API and a credit-service, and includes Swagger/OpenAPI documentation. It uses **reactive, non-blocking I/O** (via Spring WebFlux and `WebClient`) for scalability [1]. Vendor routing is implemented with the **Strategy Pattern**: one service bean per gateway, injected as a map to dispatch dynamically based on business logic [2] [3]. Configuration includes proxy support for `WebClient` and an Oracle DB datasource (using Spring Data R2DBC for reactive access) [4]. Swagger documentation is enabled via `springdoc-openapi` [5].

Below is an outline of the project structure and example code files:

- `pom.xml` – project dependencies (Spring WebFlux, R2DBC, Swagger, etc).
- `src/main/java/.../PaymentApplication.java` – main class with Swagger/OpenAPI annotation.
- `config/WebClientConfig.java` – bean configurations for `WebClient` (with HTTP proxy).
- `config/SwaggerConfig.java` – (optional) Swagger configuration.
- `model/PaymentRequest.java`, `model/PaymentResponse.java`, `model/PaymentTransaction.java` – DTOs and entity.
- `repository/PaymentRepository.java` – reactive repository for Oracle DB.
- `service/PaymentProvider.java` (interface), `service/PayPalProvider.java`, `service/StripeProvider.java` – vendor-specific strategy beans.
- `service/PaymentRouter.java` – routes requests to the correct `PaymentProvider`.
- `service/CreditClientService.java` – client for the credit-customer microservice.
- `controller/PaymentController.java` – REST controller (reactive endpoints).
- `resources/application.yml` – configuration (DB, URLs, proxies, Swagger).

Each part is briefly explained below with code snippets.

1. Maven Dependencies

Include Spring WebFlux, Spring Data R2DBC (for Oracle), Springdoc OpenAPI, and any resilience libraries. For example, in `pom.xml`:

```
<dependencies>
    <!-- WebFlux (reactive web) -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-webflux</artifactId>
    </dependency>
```

```
    <!-- Spring Data R2DBC for Oracle (reactive DB) -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-r2dbc</artifactId>
    </dependency>
    <!-- Oracle R2DBC driver (example groupId/artifactId) -->
    <dependency>
        <groupId>com.oracle.database.r2dbc</groupId>
        <artifactId>oracle-r2dbc</artifactId>
        <version>0.4.0</version>
    </dependency>

    <!-- Swagger / OpenAPI (SpringDoc) -->
    <dependency>
        <groupId>org.springdoc</groupId>
        <artifactId>springdoc-openapi-starter-webflux-ui</artifactId>
        <version>2.0.4</version>
    </dependency>

    <!-- (Optional) Resilience4j for circuit-breaker, retry -->
    <dependency>
        <groupId>io.github.resilience4j</groupId>
        <artifactId>resilience4j-spring-boot3</artifactId>
        <version>2.0.2</version>
    </dependency>
</dependencies>
```

This configuration enables **non-blocking web I/O** via WebFlux [1] and reactive database access. (Spring Data R2DBC supports Oracle as a reactive DB [6].)

2. Main Application & Swagger

The main class enables Spring Boot and Swagger. We annotate it with `@OpenAPIDefinition` (from Springdoc) for API info [7]:

```
package com.example.payments;

import io.swagger.v3.oas.annotations.OpenAPIDefinition;
import io.swagger.v3.oas.annotations.info.Info;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
@OpenAPIDefinition(info = @Info(title = "Payment API", version = "v1",
    description = "Handles payments via multiple gateways"))
```

```
public class PaymentApplication {
    public static void main(String[] args) {
        SpringApplication.run(PaymentApplication.class, args);
    }
}
```

With `springdoc-openapi-starter-webflux-ui` added, Swagger UI will be available (e.g. at `/swagger-ui/index.html`) [8].

3. Configuration: WebClient with Proxy

Use `WebClient` for all external HTTP calls (to middleware and credit service). To support an HTTP proxy, configure a `ReactorClientHttpConnector` with proxy settings [9]. For example, in `config/WebClientConfig.java`:

```
package com.example.payments.config;

import io.netty.channel.ChannelOption;
import io.netty.handler.proxy.ProxyHandler;
import io.netty.resolver.DefaultAddressResolverGroup;
import reactor.netty.http.client.HttpClient;
import reactor.netty.transport.ProxyProvider;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.client.reactive.ClientHttpConnector;
import org.springframework.web.reactive.function.client.WebClient;
import reactor.netty.http.client.HttpClient;

@Configuration
public class WebClientConfig {

    @Value("${middleware.service.url}")
    private String middlewareBaseUrl;
    @Value("${credit.service.url}")
    private String creditServiceBaseUrl;
    @Value("${http.proxy.host}")
    private String proxyHost;
    @Value("${http.proxy.port}")
    private Integer proxyPort;

    // WebClient for payment middleware calls
    @Bean
    public WebClient paymentWebClient() {
        // Configure HTTP proxy
        HttpClient httpClient = HttpClient.create()
```

```
            .tcpConfiguration(tcpClient ->
                tcpClient
                  .resolver(DefaultAddressResolverGroup.INSTANCE)
                  .option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 5000)
                  .doOnConnected(conn -> conn
                      .addHandlerLast(new ProxyHandler(
                          ProxyProvider.Proxy.HTTP, proxyHost, proxyPort))))
            ;
        // Build WebClient with base URL of middleware service
        ClientHttpConnector connector = new
ReactorClientHttpConnector(httpClient);
        return WebClient.builder()
                .baseUrl(middlewareBaseUrl)
                .clientConnector(connector)
                .build();
    }

    // WebClient for credit microservice calls
    @Bean
    public WebClient creditWebClient() {
        return WebClient.builder()
                .baseUrl(creditServiceBaseUrl)
                .build();
    }
}
```

This sets up two `WebClient` beans: one for the payment middleware (with proxy support) and one for the credit service. The proxy is added in code because JVM proxy flags do not work with WebFlux clients [10] [11].

4. Data Model and Repository

Define a request/response DTO and an entity for Oracle persistence. For example:

```
// model/PaymentRequest.java
public class PaymentRequest {
    private String customerId;
    private String vendorId;
    private double amount;
    // getters/setters...
}

// model/PaymentResponse.java
public class PaymentResponse {
    private String transactionId;
    private String status;
    private String message;
```

```
    // getters/setters...
}

// model/PaymentTransaction.java (for DB)
import org.springframework.data.annotation.Id;
import org.springframework.data.relational.core.mapping.Table;

@Table("PAYMENT_TXN")
public class PaymentTransaction {
    @Id
    private Long id;
    private String customerId;
    private String vendorId;
    private double amount;
    private String status;
    private String details;
    // getters/setters...
}
```

And a reactive repository interface for Oracle R2DBC (assuming Spring Data R2DBC is used) in `repository/PaymentRepository.java`:

```
package com.example.payments.repository;

import com.example.payments.model.PaymentTransaction;
import org.springframework.data.repository.reactive.ReactiveCrudRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface PaymentRepository extends
ReactiveCrudRepository<PaymentTransaction, Long> {
    // Reactive CRUD methods for PaymentTransaction
}
```

This uses **Spring Data R2DBC** to handle Oracle in a reactive way [4].

5. Vendor Strategy Interfaces

Define a generic interface for payment providers (the Strategy pattern) and implementations for each vendor. Spring can autowire all implementations into a map by bean name [3]. For example:

```
// service/PaymentProvider.java
package com.example.payments.service;

import com.example.payments.model.PaymentRequest;
```

```java
import com.example.payments.model.PaymentResponse;
import reactor.core.publisher.Mono;

public interface PaymentProvider {
    String getVendorId();  // e.g. "paypal", "stripe"
    Mono<PaymentResponse> processPayment(PaymentRequest request);
}
```

Each vendor has its own bean. For example, `service/PayPalProvider.java`:

```java
package com.example.payments.service;

import com.example.payments.model.PaymentRequest;
import com.example.payments.model.PaymentResponse;
import org.springframework.stereotype.Component;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.reactive.function.client.WebClient;
import reactor.core.publisher.Mono;

@Component
public class PayPalProvider implements PaymentProvider {
    @Autowired
    private WebClient paymentWebClient; // injected from WebClientConfig

    @Override
    public String getVendorId() { return "paypal"; }

    @Override
    public Mono<PaymentResponse> processPayment(PaymentRequest request) {
        // Call middleware endpoint for PayPal (as an example)
        return paymentWebClient.post()
                .uri(uriBuilder -> uriBuilder.path("/pay/paypal").build())
                .bodyValue(request)
                .retrieve()
                .bodyToMono(PaymentResponse.class);
    }
}
```

And `service/StripeProvider.java` similarly:

```java
@Component
public class StripeProvider implements PaymentProvider {
    @Autowired
    private WebClient paymentWebClient;
```

```
    @Override
    public String getVendorId() { return "stripe"; }

    @Override
    public Mono<PaymentResponse> processPayment(PaymentRequest request) {
        return paymentWebClient.post()
                .uri(uriBuilder -> uriBuilder.path("/pay/stripe").build())
                .bodyValue(request)
                .retrieve()
                .bodyToMono(PaymentResponse.class);
    }
}
```

Using **Strategy Pattern** (one implementation per gateway) makes it easy to add new vendors later by adding a new `PaymentProvider` bean with `getVendorId()` and `processPayment()` logic [2].

6. Routing Logic

A router/service selects the appropriate `PaymentProvider` based on `vendorId`. We inject all `PaymentProvider` beans into a map (`vendorId` → bean) for dynamic dispatch [3]. For example, `service/PaymentRouter.java`:

```
package com.example.payments.service;

import com.example.payments.model.PaymentRequest;
import com.example.payments.model.PaymentResponse;
import reactor.core.publisher.Mono;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import java.util.Map;

@Service
public class PaymentRouter {

    private final Map<String, PaymentProvider> providers;
    private final CreditClientService creditService;
    private final PaymentRepository paymentRepo;

    @Autowired
    public PaymentRouter(Map<String, PaymentProvider> providers,
                         CreditClientService creditService,
                         PaymentRepository paymentRepo) {
        this.providers = providers;
        this.creditService = creditService;
        this.paymentRepo = paymentRepo;
    }
```

```
    public Mono<PaymentResponse> routePayment(PaymentRequest request) {
        // Example business logic: check credit before payment
        return creditService.verifyCredit(request.getCustomerId(),
request.getAmount())
            .flatMap(creditOk -> {
                if (!creditOk) {
                    return Mono.just(new PaymentResponse(/* ... insufficient
credit ... */));
                }
                // Determine provider by vendorId
                PaymentProvider provider = providers.get(request.getVendorId());
                if (provider == null) {
                    return Mono.error(new IllegalArgumentException("Unknown
vendor: " + request.getVendorId()));
                }
                // Process payment through vendor
                return provider.processPayment(request);
            })
            .flatMap(response -> {
                // Optionally save transaction in DB (reactively)
                PaymentTransaction txn = new PaymentTransaction();
                txn.setCustomerId(request.getCustomerId());
                txn.setVendorId(request.getVendorId());
                txn.setAmount(request.getAmount());
                txn.setStatus(response.getStatus());
                txn.setDetails(response.getMessage());
                return paymentRepo.save(txn).thenReturn(response);
            });
    }
}
```

In this example, `providers.get(request.getVendorId())` fetches the correct bean (e.g. `"paypal"` maps to `PayPalProvider`). This map-based dispatch implements the **Strategy pattern** [2] and makes adding new vendors easy (just add another bean and its ID). Spring automatically collects all `PaymentProvider` beans into the map [3].

7. Credit Service Client

The router calls a credit-check microservice before charging. The `CreditClientService` can use `WebClient` to call the credit microservice's API, returning a `Mono<Boolean>` or credit details. For example:

```
package com.example.payments.service;

import org.springframework.beans.factory.annotation.Autowired;
```

```
import org.springframework.stereotype.Service;
import org.springframework.web.reactive.function.client.WebClient;
import reactor.core.publisher.Mono;

@Service
public class CreditClientService {
    @Autowired
    private WebClient creditWebClient;

    // Verify if customer has sufficient credit
    public Mono<Boolean> verifyCredit(String customerId, double amount) {
        return creditWebClient.get()
                .uri(uriBuilder -> uriBuilder.path("/credit/check")
                                             .queryParam("customerId",
customerId)
                                             .queryParam("amount", amount)
                                             .build())
                .retrieve()
                .bodyToMono(Boolean.class);
    }
}
```

This is a simple reactive client; in a real system you might handle errors, timeouts, etc. The microservice URL is configured in `application.yml`.

8. REST Controller

Expose an endpoint to initiate payments. Using WebFlux, controller methods return reactive types (`Mono<...>`). For example, `controller/PaymentController.java`:

```
package com.example.payments.controller;

import com.example.payments.model.PaymentRequest;
import com.example.payments.model.PaymentResponse;
import com.example.payments.service.PaymentRouter;
import io.swagger.v3.oas.annotations.Operation;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.*;
import reactor.core.publisher.Mono;

@RestController
@RequestMapping("/api/payments")
public class PaymentController {

    @Autowired
```

```
    private PaymentRouter paymentRouter;

    @Operation(summary = "Process a payment")
    @PostMapping
    @ResponseStatus(HttpStatus.ACCEPTED)
    public Mono<PaymentResponse> processPayment(@RequestBody PaymentRequest
request) {
        return paymentRouter.routePayment(request);
    }
}
```

With Springdoc annotations (`@Operation`) this endpoint will be documented automatically. The controller delegates to the `PaymentRouter` service and returns the reactive `Mono<PaymentResponse>`.

9. Proxy and Resilience (Additional Considerations)

For resilience, one can integrate **Resilience4j** or Spring Cloud Circuit Breaker annotations on the service methods. For example, annotating `routePayment` with `@CircuitBreaker` or using Reactor's `retryWhen()` could protect against flaky external calls [12]. In this skeleton, you can add such annotations to the `PaymentRouter` or `PaymentProvider` methods if desired.

10. Configuration (application.yml)

Finally, configure connection details in `src/main/resources/application.yml`:

```
server:
  port: 8080

spring:
  r2dbc:
    url: r2dbc:oracle:thin:@//<HOST>:1521/ORCL
    username: your_db_user
    password: your_db_pass

middleware:
  service:
    url: http://payment-middleware-service/api

credit:
  service:
    url: http://credit-service/api

http:
  proxy:
    host: proxy.example.com
```

```
    port: 8888

springdoc:
  swagger-ui:
    path: /swagger-ui.html
```

Adjust the URLs and credentials for your environment. The `springdoc.swagger-ui.path` sets the path for the Swagger UI as needed.

Summary: This skeleton outlines a reactive Spring Boot microservice that routes payments to different vendors via a middleware API, using WebFlux and WebClient (with proxy), and organizes vendor logic via the Strategy pattern for easy extensibility [2] [3]. It also connects to an Oracle database (reactively via R2DBC [4]) and to another "credit" microservice for customer operations. Swagger (OpenAPI) documentation is enabled with Springdoc [5] so that all APIs are self-documented and testable. Other developers can start by filling in the business logic and vendor details in this structure.

---

[1] Spring WebClient | Baeldung
https://www.baeldung.com/spring-5-webclient

[2] Typical Pattern for multiple providers/vendors? - Stack Overflow
https://stackoverflow.com/questions/2155206/typical-pattern-for-multiple-providers-vendors

[3] Autowiring an Interface With Multiple Implementations | Baeldung
https://www.baeldung.com/spring-boot-autowire-multiple-implementations

[4] [6] Spring Data R2DBC
https://spring.io/projects/spring-data-r2dbc/

[5] [7] [8] Swagger-Open API to Spring Boot 3 Application(Web Flux) | by Pranav S Khodanpur (anushku) | Medium
https://pranavkhodanpur.medium.com/swagger-open-api-to-spring-boot-3-application-web-flux-2e99bb112151

[9] [10] [11] Adding HTTP Proxy to Spring WebFlux Client – Madhur Ahuja
https://www.madhur.co.in/blog/2022/09/03/adding-http-proxy-spring-webflux-client.html

[12] Webflux WebClient retry and Spring Cloud Circuit Breaker Resilience4J Retry pattern walk into a bar - Stack Overflow
https://stackoverflow.com/questions/64051144/webflux-webclient-retry-and-spring-cloud-circuit-breaker-resilience4j-retry-patt