# Gateway Aggregator pattern

The problem with calling multiple API from client

**Multiple Service Calls Issue**: In a microservices architecture, a client application (like a web browser) often needs data from multiple backend services to render a single page or complete a task. This creates several problems:

**Browser Connection Limits**: Web browsers have built-in limits on concurrent connections per domain. Chrome, for example, limits this to 6 simultaneous connections. If your application needs to call 8 different services, only 6 calls can happen at once - the remaining 2 must wait for slots to free up.

**Network Latency Multiplication**: Each service call adds network latency. If your servers are in the US and users are in Europe, every individual service call experiences that trans-Atlantic delay, multiplying the total page load time.

**Client-Side Complexity**: The client application becomes responsible for orchestrating multiple service calls, handling failures, and managing the complexity of combining responses from different services.

**The Gateway Aggregator Solution**

The Gateway Aggregator Pattern introduces a dedicated microservice that acts as an intermediary between clients and backend services

Use case

<http://localhost:7070/swagger-ui/>

As part of the aggregator pattern, we must call below APIs simultaneously. Product page has all the necessary information ready

1. <http://localhost:7070/sec01/product/1> Provides the product details for the given product id (up to 50)
2. <http://localhost:7070/sec01/promotion/1> Product id 5, 10, 15, 20, 25, 30, 35, 40, 45, 50 are not present in this service
3. <http://localhost:7070/sec01/review/1> Product id 10, 20, 30, 40, 50 are not present in this service. Product Id 7 has some weird issues

We use ProductAggregatorService to call all the API using ZIP function

*public* Mono<ProductAggregate> aggregate(Integer id){  
 *return* Mono.zip(  
 *this*.productClient.getProduct(id),  
 *this*.promotionClient.getPromotion(id),  
 *this*.reviewClient.getReviews(id)  
 )  
 .map(t -> toDto(t.getT1(), t.getT2(), t.getT3()));  
}

Error handling

Since the /product/id API is the main API and if fails there is nothing to send to user hence we return Mono.empty()

*public* Mono<ProductResponse> getProduct(Integer id){  
 *return this*.client  
 .get()  
 .uri("{id}", id)  
 .retrieve()  
 .bodyToMono(ProductResponse.*class*)  
 .onErrorResume(ex -> Mono.empty());  
}

Then we handle the empty in controller

*@RestController  
@RequestMapping*("sec01")  
*public class* ProductAggregateController {  
  
 *@Autowired  
 private* ProductAggregatorService service;  
  
 *@GetMapping*("product/{id}")  
 *public* Mono<ResponseEntity<ProductAggregate>> getProductAggregate(*@PathVariable* Integer id){  
 *return this*.service.aggregate(id)  
 .map(ResponseEntity::ok)  
 .defaultIfEmpty(ResponseEntity.notFound().build());  
 }  
  
}

For promotion if there is no promotion is present for Product id 5, 10, 15, 20, 25, 30, 35, 40, 45, 50 and it will return 404, so we need to handle it as Zip will expect tuple. So we need to handle it

*public* Mono<PromotionResponse> getPromotion(Integer id){  
 *return this*.client  
 .get()  
 .uri("{id}", id)  
 .retrieve()  
 .bodyToMono(PromotionResponse.*class*)  
 .onErrorReturn(noPromotion);  
}

For review also we have similar issues where Product id 10, 20, 30, 40, 50 are not present in this service. Product Id 7 has some weird issues. So we need to handle error.

*public* Mono<List<Review>> getReviews(Integer id){  
 *return this*.client  
 .get()  
 .uri("{id}", id)  
 .retrieve()  
 .bodyToFlux(Review.*class*)  
 .collectList() *// Will collect to List in non blocking way* .onErrorReturn(Collections.emptyList());  
}

# Scatter gather pattern

**What is the Scatter-Gather Pattern?**

The Scatter-Gather pattern involves:

1. **Scattering**: Broadcasting the same request to multiple similar services
2. **Gathering**: Collecting all the responses and either combining them or selecting the best one
3. **Responding**: Sending the compiled or filtered results back to the client

**Real-World Example: Flight Booking**

In a flight booking application like Google Flights or Kayak:

**The Process**:

1. User searches for flights from City A to City B
2. The service acts as a travel agent
3. It broadcasts the same query to all airline carriers (Frontier, Delta, American, etc.)
4. Each airline responds with their available flights and prices
5. The system gathers all responses

**The Options**:

* **Present All**: Show all available flights from all airlines
* **Filter/Select**: Choose the cheapest flight, fastest route, or best value
* **Combine**: Group results by criteria (price ranges, departure times, etc.)

# Orchestrator Pattern

**Orchestrator Pattern**, is a specialized form of the aggregator pattern that includes business logic for managing complex workflows and transactions.

**What is the Orchestrator Pattern?**

The Orchestrator Pattern is an aggregator with additional capabilities:

* **Workflow Management**: Controls the sequence and flow of service calls
* **Business Logic**: Contains rules about how services should interact
* **Transaction Coordination**: Manages success/failure scenarios across multiple services
* **Compensation Logic**: Handles rollback operations when things go wrong

**The E-commerce Order Example**

The explanation uses an order processing system with these services:

* **Product Service**: Manages product information and pricing
* **Payment Service**: Handles payment processing
* **Inventory Service**: Manages stock availability
* **Shipping Service**: Coordinates delivery

**The Challenge Without Orchestration**

**Complex Dependencies**: For a successful order, ALL services must succeed:

1. Product must exist and have valid pricing
2. User must have sufficient funds
3. Item must be available in inventory
4. Shipping must be available for the destination

**Compensation Required**: If any step fails after others have succeeded, you need to undo previous actions. For example:

* Payment was charged ✓
* But inventory check failed ✗
* Now you must refund the payment

**Service Coupling**: Without orchestration, the order service would need to handle all this coordination logic, making it complex and tightly coupled to other services.

**How the Orchestrator Pattern Solves This**

**Separation of Concerns**:

1. **Order Service**: Handles basic validation and database operations
2. **Orchestrator Service**: Manages the complex workflow and coordination

**Workflow Steps**:

1. Order service receives request and does basic validation
2. Order service creates database record with status "CREATED" (not success/fail)
3. Request is passed to the orchestrator
4. Orchestrator manages the complex workflow:
   * Calls product service for pricing information
   * Makes parallel calls to payment, inventory, and shipping services
   * Handles success/failure scenarios
   * Manages compensation if needed

**Transaction Management**:

* **Success Case**: All services succeed → Order status updated to "SUCCESS"
* **Failure Case**: Any service fails → Orchestrator handles rollback:
  + Refunds payment if already charged
  + Cancels shipping if already scheduled
  + Updates order status to "FAILED"

**Key Benefits**

**Centralized Business Logic**: All workflow rules and coordination logic live in one place rather than being scattered across services.

**Clean Service Boundaries**: Individual services (payment, inventory, shipping) remain focused on their core responsibilities without worrying about coordination.

**Reliable Transactions**: The orchestrator ensures that either all operations succeed or all are properly rolled back.

**Maintainable Workflows**: Changes to business rules or workflow logic only require updates to the orchestrator, not multiple services.

**Error Handling**: Centralized handling of failure scenarios and compensation logic.

**When to Use the Orchestrator Pattern**

This pattern is ideal when you have:

* Complex business workflows spanning multiple services
* Strong consistency requirements (all-or-nothing transactions)
* Need for compensation/rollback logic
* Business rules that govern how services interact
* Requirements for centralized monitoring and control of workflows

This explanation outlines the **implementation strategy for the Orchestrator Pattern** and provides a comprehensive matrix of success/failure scenarios. Let me break down the key concepts:

**Implementation Approach**

**Service Boundaries**: The orchestrator service will consume existing APIs from:

* Product Service
* User Service (handles payment/balance checks)
* Inventory Service
* Shipping Service

**Orchestrator Responsibility**: The orchestrator doesn't implement the business logic of individual services - it coordinates their interactions and handles the workflow complexity.

**Execution Flow**

**Step 1 - Product Validation**:

* Make initial call to Product Service
* If product doesn't exist, fail immediately without calling other services

**Step 2 - Parallel Service Calls**:

* Simultaneously call User Service (payment), Inventory Service, and Shipping Service
* This parallelization improves performance since these operations are independent

**Step 3 - Success/Failure Handling**:

* If all succeed → Order successful
* If any fail → Execute compensation logic for successful operations



External services

1. <http://localhost:7070/sec03/product/1> return product result with price
2. <http://localhost:7070/sec03/user/1> return user details with balance
3. <http://localhost:7070/sec03/user/deduct> POST API where we deduct the amount from user balance