**Service Mediation and Composition in Apigee**

In microservices architecture, different services perform specific business functions. A business process might involve calling multiple microservices in sequence or in parallel, transforming requests and aggregating responses. **Apigee** facilitates this using policies, flow conditions, and JavaScript/JavaCallout components, serving as a powerful layer for **Service Mediation** and **Composition**.

Apigee enables orchestration without modifying backend services, making it ideal for rapid integration, modernization, and API abstraction.

**1. Service Chaining and Orchestration via Apigee**

**Definition:**

**Service chaining** refers to sequentially calling two or more microservices where the output of one may be used as input to the next. **Orchestration** is managing the logic of these chained services centrally — often at the API layer.

**Use Case:**

In a travel booking system:

* First service checks flight availability
* Second service calculates price with discounts
* Third service finalizes booking

All three can be chained in Apigee without exposing their individual complexities to the client.

**How Apigee Supports It:**

Apigee enables chaining using the <ServiceCallout> policy and conditional flows. Each callout stores its result in a variable, which can be passed to the next service.

**Example (Chaining Two Services):**

<ServiceCallout name="CallService1">

<Request variable="service1Request">

<Set>

<Verb>GET</Verb>

<URL>http://service1.com/api/check</URL>

</Set>

</Request>

<Response>service1Response</Response>

</ServiceCallout>

<ServiceCallout name="CallService2">

<Request variable="service2Request">

<Set>

<Verb>POST</Verb>

<URL>http://service2.com/api/price</URL>

<Payload contentType="application/json">

{"input": "{service1Response.content}"}

</Payload>

</Set>

</Request>

<Response>service2Response</Response>

</ServiceCallout>

**2. Calling Multiple Microservices in a Single Proxy**

**Scenario:**

Sometimes a single API request needs to trigger multiple independent microservices simultaneously — for example, a dashboard request fetching user info, orders, and notifications.

**Implementation Approaches:**

* Sequential calls using <ServiceCallout> policies
* Parallel execution using JavaScript (async) or custom JavaCallout logic (not natively parallel in Apigee)
* Store results in different variables
* Merge or conditionally select outputs

**Use Case:**

A banking app dashboard requires user profile, balance info, and recent transactions:

* /user/profile
* /user/balance
* /user/transactions

All are invoked from a single proxy.

**3. Aggregating Responses**

**Definition:**

**Response aggregation** means collecting data from multiple microservices and combining them into a unified response structure for the client.

**Use Case:**

E-commerce order summary:

* Order Service → order ID, date
* Payment Service → payment status
* Shipment Service → tracking info

**Implementation in Apigee:**

Use <ServiceCallout> to fetch responses and a JavaScript policy to merge them.

**JavaScript Aggregation Example:**

var order = JSON.parse(context.getVariable("orderResponse.content"));

var payment = JSON.parse(context.getVariable("paymentResponse.content"));

var shipment = JSON.parse(context.getVariable("shipmentResponse.content"));

var aggregated = {

"order": order,

"payment": payment,

"shipment": shipment

};

context.setVariable("aggregatedResponse", JSON.stringify(aggregated));

Then attach a AssignMessage policy to set this as the final response:

<AssignMessage name="SetResponse">

<Set>

<Payload contentType="application/json">{aggregatedResponse}</Payload>

</Set>

<AssignTo createNew="false" type="response"/>

</AssignMessage>

**Comparison Table: Orchestration in Apigee**

| **Feature** | **Description** | **Ideal Use Case** |
| --- | --- | --- |
| Service Callout | Calls external/internal services from within proxy | Sequential API interactions |
| JavaScript Aggregation | Combines multiple JSON payloads into a unified response | Dashboards, Reporting APIs |
| Conditional Flows | Routes execution based on path, method, or custom logic | API gateway logic segmentation |
| JavaCallout | For advanced orchestration logic, retries, async operations | High-performance backend logic |

**Real-World Case Study: Telecom API Orchestration**

A telecom company exposes a public API /getCustomerInfo. Internally, this involves:

* Fetching customer profile
* Fetching active service plans
* Fetching last 5 complaints

Using Apigee:

* Each service is invoked via ServiceCallout
* Results are parsed and aggregated via JavaScript
* Final response is returned to the app

This helped the company reduce client-side complexity and centralized all orchestration securely in the API layer.

**Conclusion**

Apigee’s service mediation capabilities allow businesses to decouple clients from backend complexities. Whether chaining microservices, orchestrating backend processes, or aggregating responses, Apigee makes it possible through simple, declarative policies and scripts. This promotes reuse, maintainability, and centralized control in microservices-based systems.