**Best Practices for Microservice APIs with Apigee**

This document elaborates on best practices in designing, securing, and maintaining Microservice APIs using Apigee. It focuses on four critical areas: naming conventions, versioning, security layers, and performance monitoring.

**1. Naming Conventions**

**1.1 Importance**

Naming conventions bring clarity, maintainability, and traceability to your API ecosystem, especially in microservice architectures where multiple small APIs are developed and deployed independently.

**1.2 Best Practices**

* **Use hyphen-separated lowercase names** for readability.
* Follow a format like: business-domain-api-version or service-functionality-v1.
* Align proxy names with internal service or business function names.
* Prefix internal services differently if needed (e.g., int-, svc-, etc.).
* Example: order-confirmation-v1, inventory-query-v2, customer-profile-v1

**1.3 Example Use Case**

For an e-commerce platform:

* order-service handles orders
* order-service-v1 becomes the Apigee proxy name
* Developers instantly recognize this proxy’s function and version

**2. API Versioning**

**2.1 Importance**

APIs evolve. To prevent breaking changes from affecting consumers, you must version them effectively.

**2.2 Best Practices**

* **Use URI versioning**: /v1/products, /v2/products.
* Avoid using query params or headers for versioning.
* Keep breaking and non-breaking changes separate:
  + Minor updates: v1.1
  + Breaking changes: increment major version: v2
* Deprecate old versions gracefully with announcements.
* Document each version clearly.

**2.3 Example Use Case**

/v1/users returns basic user data. /v2/users adds detailed address and preferences. Consumers with legacy systems remain on v1 while new systems adopt v2.

**3. Security Layers**

**3.1 Importance**

Microservices often expose sensitive data and must be secured at multiple layers to prevent abuse, data leaks, and unauthorized access.

**3.2 Best Practices**

* Use **OAuth 2.0** or **API Key** authentication depending on client type.
* Use **Spike Arrest** to rate-limit requests and prevent sudden spikes.
* Use **Quota** to limit requests per client/app per time window.
* Block known malicious IPs or use **IP Whitelisting**.
* Always use HTTPS and disable TLS versions < 1.2.
* Use VerifyJWT for internal service-to-service authentication.
* Never expose stack traces or internal error details.
* Define **custom fault handling** using FaultRules and AssignMessage for clean error messages.

**3.3 Example Use Case**

In a payment system:

* External users must pass an OAuth 2.0 access token.
* Rate limit to 5 rps using Spike Arrest.
* Limit to 1000 calls/day using Quota.
* Internally, Apigee uses JWT to validate backend microservices’ identity.

**4. Monitoring Performance Bottlenecks**

**4.1 Importance**

Even well-designed APIs can degrade in performance. Apigee’s tools allow proactive detection and root cause analysis of such issues.

**4.2 Best Practices**

* Use **Apigee Analytics** to monitor:
  + Average response time
  + Backend latency
  + Error rates by status code
* Use **custom dimensions** to log user types, app IDs, or transaction IDs.
* Utilize the **Trace tool** for real-time debugging.
* Implement **MessageLogging** policies to log to Pub/Sub or Stackdriver.
* Identify and isolate **backend delays** using TargetLatency metrics.

**4.3 Example Use Case**

If /checkout endpoint latency spikes:

* Use Trace tool: find delays caused by ServiceCallout to fraud detection API.
* Logs show fraud service took 3s+.
* Introduce fallback logic to bypass fraud checks during high traffic, ensuring user checkout completes within SLA.

**Conclusion**

Following these best practices in Apigee ensures that microservice APIs are consistent, secure, scalable, and observable. Teams can manage APIs confidently across their lifecycle while meeting enterprise-grade SLAs and user expectations.

Let us know if you need checklists, templates, or implementation examples for each policy.