MIRACLE
SOFTWARE SYSTEMS

APIGEE Best Practices for Leading Restaurant

Request for Proposal | January 05th, 2017

Hanu Veluri
Director - Hybrid Integration
Miracle Software Systems, Inc.

January 04th, 2017

MIRACLE
SOFTWARE SYSTEMS

Table of Contents

APIGEE Best Practices for Leading Restaurant

Document Control Section

Document History

The following table contains the revision history of the System & Feature requirements document.

Table 1 - Revision History

Version	Description of Change	Author	Date
1.0	Initial Version	Miracle Software Systems	08/29/2016

Document Approvals

Version	Owner	Approvers	Date Approved	Version Notes (Changes Made)

Document Distribution

This document is distributed to the following:

- Leading Pizza Store

1. Introduction

Our customer is a leading American Restaurant. They run the third largest take-away and pizza delivery restaurant chain in the world. The customer is in need of improving their strategy, road map and re-engineer their reference architecture to build Pizza tracking system using APIGEE and other proprietary development tools to build APIs – for serving their online consumers to monitor pizza status.

2. Purpose

The purpose of this document is to represent in detail of our best practices for Apigee development, administration and monitoring. This document is targeted for Apigee architect, developers and support persons. This best practices can be used by developers and architect as a reference for any new development project.

3. API URL Naming and Structure

REST is a software architecture style. where user progresses through different states by selecting different links. Clicking the link cause a change of state and takes a user to a different state of software. URLs are very important for any restful apes. In a typical web application, URLs references resources and should reflect the resource name so that it is easily understandable by end user. There are some general rules for naming URL's.

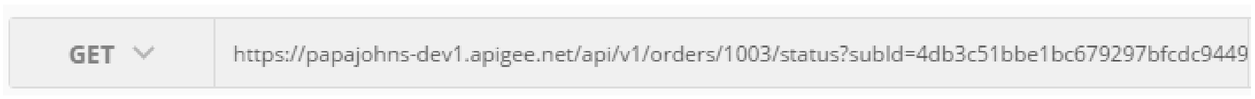
3.1 Nouns are Good and Verbs are Bad

Since HTTP protocol itself provides a way to define method calls using verbs like GET, POST, PUT and DELETE etc., it makes sense to use in-built methods as call to perform task on resources. Using specific verbs in URL complicate thing. There can be tens of hundreds of methods/verbs that can be used on particular resource. Designing APIs based on verbs will not scale well as tens of hundreds APIs can be introduced over the period of time, if verbs are used in API design.

In AIGEE, HTTP method aware rules can be configured to execute specific flows or policies.

3.2 Plural Nouns are Better

It's better to use plural nouns like /orders, /deals, /sales as it logically makes sense to put specific items, objects, images or resource under this parent noun. The below image shows the use of nouns "orders" in URL.


GET

https://papajohns-dev1.apigee.net/api/v1/orders/1003/status?subId=4db3c51bbe1bc679297bfcdc9449

The above image is showing the use of nouns "orders" in URL

3.3 Concrete Nouns are Better than Abstract

Abstraction is good in most cases but it cannot be best approach in all use cases. Developers might want more specific nouns which are more specific instead of architect recommended abstract nouns.

For example, /images, /videos and /music's makes more sense as URL paths for a content hosting application instead of /items or /contents.

3.4 Relationship Complexities and Attributes

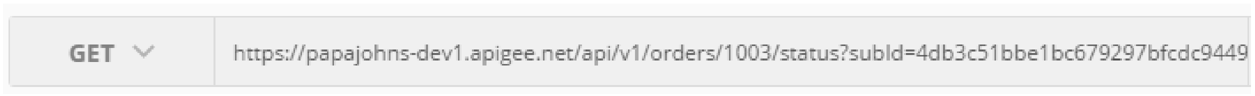
There can be many relationships between nouns for example, for an e-commerce site selling books, a book can have several attributes which can be used for searching like Author, Publisher, Title or ISBN number. These complexities can be handles by combination of two nouns in URLs;/publisher/{publisherId}/title/{title name}.

If there are some characteristics of items like color, size, create_at, published_date etc. That can be passed as query parameter in HTTP URL.

For example

/publisher/{publisherId}/title/ {title name}? formFactor=Hardcover, publish_date=08252016

/publisher/{publisherId}/title/{title name}?formFactor=eBook


GET
https://papajohns-dev1.apigee.net/api/v1/orders/1003/status?subId=4db3c51bbe1bc679297bfcdc9449

The above image displays the use of noun "subId" as URL attribute

4. API Supported Format

Partners and external developers facing APIs should be able to support multiple data format like json, xml and in some cases even text so that it is easily consumable by developer's application. Data format can be used as query parameter.

For example

/orders/books?type=json

5. Pagination support

Any consumer facing APIs should be using pagination and partial response rules, so that user gets only the information that is needed from database.

For example

/books?limit=25&offset=8
There should also be a default limits for these parameter;
/books?limit=10&offset=0

6. API versioning

APIs must include a version in URLs . Different approaches have been used in the industry some enterprise use code compilation date or code release date as version. The most prevalent and recommended approach is to use prefix v followed by a number at the base of URL to reflect the relevant API version. For example /v1/orders/books/{bookId}.

Minor number can also be suffixed to the version number for minor bug fixes or smaller release for ex. /v1.2 but that would be confusing for end user and would reflect frequent change in APIs, so that should be avoided.

Passing version number in query parameter like books ? Version=v1 will make it optional attribute which is not recommended.

7. API Error Codes

Exposing API related error codes and messages to developers is a matter of choice of to make API more intuitive or make it more secure and simple. Exposing too many error codes might;

- Not be good for API security
- Confusing to end user
- Increase API proxies and consuming applications complexity

Following 3 http status codes are very common.

- 200 - OK
- 404 - Not Found
- 500 - Internal Server Error

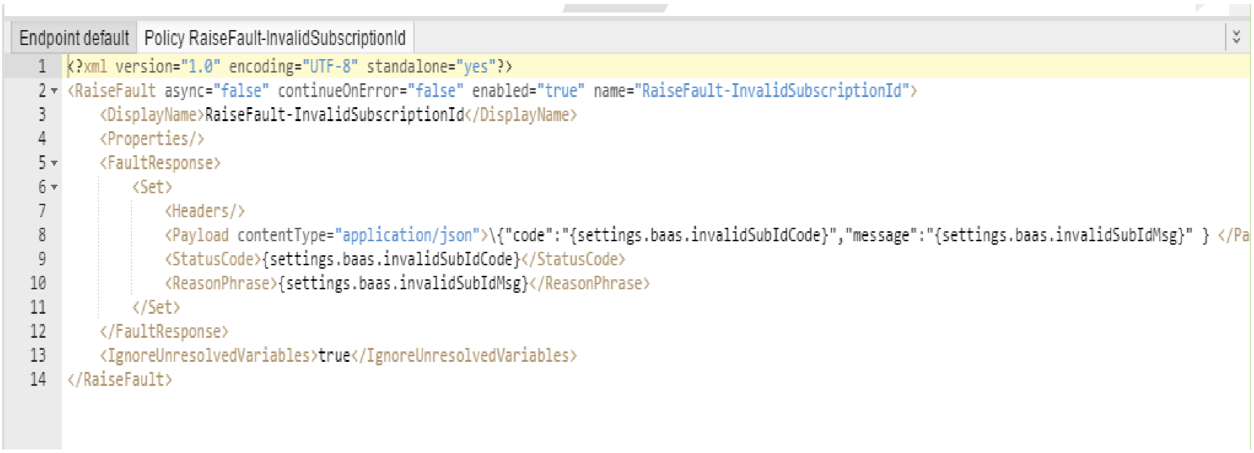
If you chose to expose more error conditions to these 3, try picking among these additional 5

- 201 - Created
- 304 - Not Modified
- 400 - Bad Request
- 401 - Unauthorized
- 403 – Forbidden

8. Apigee Development Standards

8.1 Policy Reusability

First principal of any development is "Don't repeat yourself" or to make your code or configuration reusable. Apigee provides several in-built policies which require only some configuration to make them usable as per your use case. These policies should be made more generic so that it can be used among different flows inside the proxies or can be stored in a source control system to be used by other API proxies.



```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<RaiseFault async="false" continueOnError="false" enabled="true" name="RaiseFault-InvalidSubscriptionId">
    <DisplayName>RaiseFault-InvalidSubscriptionId</DisplayName>
    <Properties/>
    <FaultResponse>
        <Set>
            <Headers/>
            <Payload contentType="application/json">\{"code":"{settings.baas.invalidSubIdCode}","message":"{settings.baas.invalidSubIdMsg}" } </Pa
            <StatusCode>{settings.baas.invalidSubIdCode}</StatusCode>
            <ReasonPhrase>{settings.baas.invalidSubIdMsg}</ReasonPhrase>
        </Set>
    </FaultResponse>
    <IgnoreUnresolvedVariables>true</IgnoreUnresolvedVariables>
</RaiseFault>
```

The above image shows the use of variables in policies for Error Payload Generation

8.2 Error Code Reusability

Architects and developers should conceive some standard error codes and messages that can be used across all API proxies in error scenario.

For example

For any order status inquiry related errors, a generic error message like "order not found" can be used whether order is for pizza, wings or soft drink.

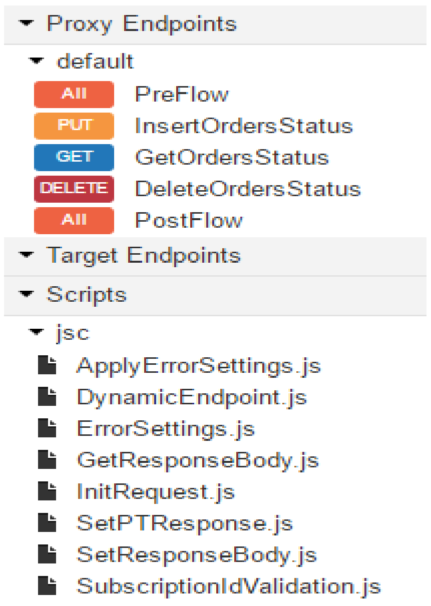
It is suggested to use a common property file for custom error messages, if error handling is performed using java, java script or python. Please refer to figure show above.

8.3 Policy and Files Naming Conventions

- The Policy name attribute and the XML policy file name must be identical
- The Script and Service Callout policy name attribute and the name of the resource file should be identical
- DisplayName should accurately describe the policy's function to someone who has never worked with that API proxy before
- Name policies according to their function, for example,

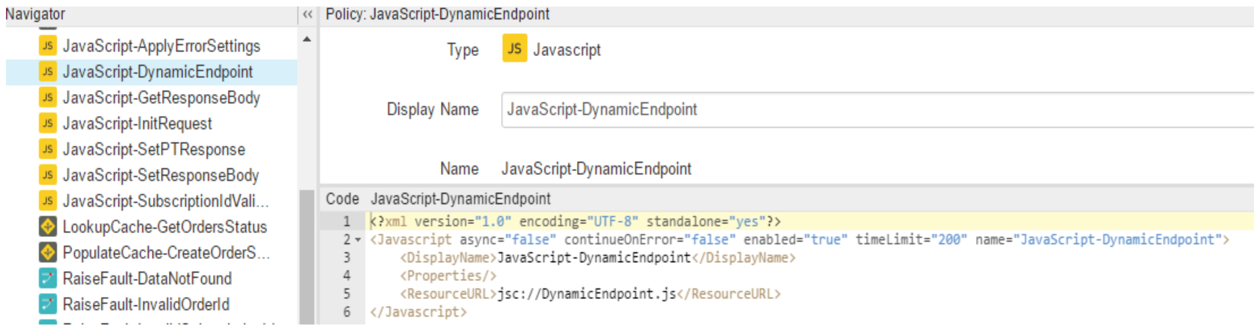
AssignTargetAuthHeader orAssignMessage.TargetAuthHeader, RaiseFault.InvalidUser

- Use proper extensions for resource files, .js for JavaScript, .py for python, and .jar for Java JAR files
- Variable names should be consistent. If you choose a style, such as camel Case or underscore, use it throughout the API proxy
- Use variable prefixes, where possible, to organize variables based on their purpose, for example Consumer username and Consumer password



```
Proxy Endpoints
  default
    All     PreFlow
    PUT     InsertOrdersStatus
    GET     GetOrdersStatus
    DELETE  DeleteOrdersStatus
    All     PostFlow
Target Endpoints
Scripts
  jsc
    ApplyErrorSettings.js
    DynamicEndpoint.js
    ErrorSettings.js
    GetResponseBody.js
    InitRequest.js
    SetPTResponse.js
    SetResponseBody.js
    SubscriptionIdValidation.js
```

The above image is displaying the flow and Java script resource names



Navigator

- JavaScript-ApplyErrorSettings
- JavaScript-DynamicEndpoint
- JavaScript-GetResponseBody
- JavaScript-InitRequest
- JavaScript-SetPTResponse
- JavaScript-SetResponseBody
- JavaScript-SubscriptionIdVali...
- LookupCache-GetOrdersStatus
- PopulateCache-CreateOrderS...
- RaiseFault-DataNotFound
- RaiseFault-InvalidOrderId

Policy: JavaScript-DynamicEndpoint

Type: Javascript

Display Name: JavaScript-DynamicEndpoint

Name: JavaScript-DynamicEndpoint

Code JavaScript-DynamicEndpoint

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Javascript async="false" continueOnError="false" enabled="true" timeLimit="200" name="JavaScript-DynamicEndpoint">
    <DisplayName>JavaScript-DynamicEndpoint</DisplayName>
    <Properties/>
    <ResourceURL>jsc://DynamicEndpoint.js</ResourceURL>
</Javascript>
```

The above image displays the policy names

8.4 API Proxy Design Configuration

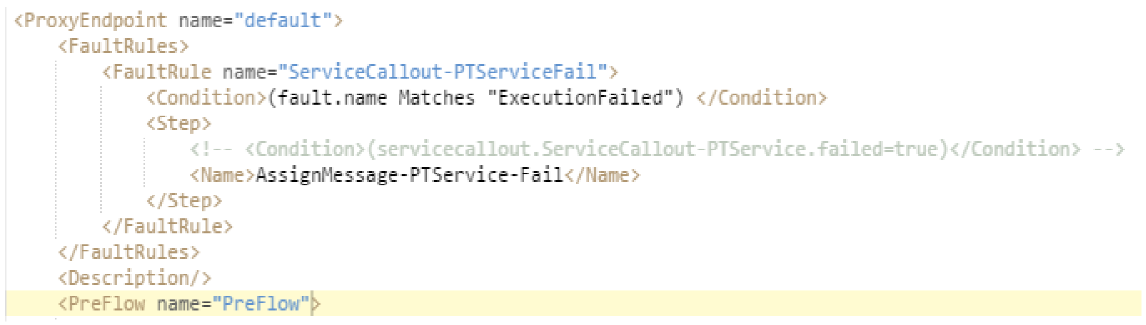
- Leverage Apigee Edge policies and functionality wherever possible to build API proxies. Avoid coding all proxy logic in JavaScript, Java, or Python resources
- Do not call the Edge management API from inside API proxies. The management API is used for administrative management purposes, not API flow logic. Policies are provided for interaction with API Services entities, such as developers, apps, access tokens and so on
- Construct Flows in an organized manner. Multiple Flows, each with a single condition, are preferable to multiple conditional attachments to the same PreFlow and Postflow
- As a 'failsafe', create a default API proxy with a proxy endpoint base path of /. This can be used to redirect base API requests to a developer site, to return a custom response, or perform another action more useful than returning the default CLASSIFICATION_ERROR
- Use Target Server resources to decouple Target Endpoint configurations from concrete URLs, supporting promotion across environments
- If you have multiple routerules, create one as the 'default', that is, as a routerule with no condition
- Maximum message payload size in edge cloud -> 10 Mb
- API proxy bundle zip file size shouldn’t exceed -> 15 Mb
- Stream requests and responses. Note that when you stream, policies no longer have access to the message content

8.5 API Proxy Fault Handling

- Leverage Fault Rules to handle all fault handling. (Raise Fault policies are used to stop message Flow and send processing to the Fault Rules Flow.)
- Within the Fault Rules Flow, use AssignMessage policies to build the fault response, not Raise Fault policies. Conditionally execute AssignMessage policies based on the fault type that occurs
- Always includes a default 'catch-all' fault handler so that system-generated faults can be mapped to customer-defined fault response formats

- If possible, always make fault responses match any standard formats available in your company or project

Use meaningful, human-readable error messages that suggest a solution to the error condition



```
<ProxyEndpoint name="default">
    <FaultRules>
        <FaultRule name="ServiceCallout-PTServiceFail">
            <Condition>(fault.name Matches "ExecutionFailed") </Condition>
            <Step>
                <!-- <Condition>(servicecallout.ServiceCallout-PTService.failed=true)</Condition> -->
                <Name>AssignMessage-PTService-Fail</Name>
            </Step>
        </FaultRule>
    </FaultRules>
    <Description/>
    <PreFlow name="PreFlow">
```

The above image shows the use of fault rule to handle exception scenario in policy

9. APIGEE Caching

- Set appropriate cache expiration intervals to avoid dirty reads
- Whenever possible, try to have the response cache policy that populates the cache execute at the ProxyEndpoint response PostFlow as late as possible
- Use key/value maps only for limited data sets. They are not designed to be a long-term data store
- Consider performance when using key/value maps as this information is stored in the Cassandra database
- See Key Value Map Operations policy
- Do not populate the response cache if the response is not successful or if the request is not a GET. Creates, updates, and deletes should not be cached
- Set appropriate cache expiration intervals to avoid dirty reads
- Whenever possible, try to have the response cache policy that populates the cache execute at the Proxy Endpoint response Post-Flow as late as possible
- The response cache policy to lookup the cache entry should occur in the Proxy Endpoint request Pre-flow

- In general, you should always keep the response cache lookup as close to the client request as possible. Conversely, you should keep the response cache population as close to the client response as possible
- JavaScript is preferred over Python and Java. However, if performance is the primary requirement, Java should be used over JavaScript

10. API Proxy and Versioning

There can be multiple scenarios to handle versioning at Apigee's API proxy layer based on how backend APIs handle the version and functionality of API proxy layer. This scenario as re follows

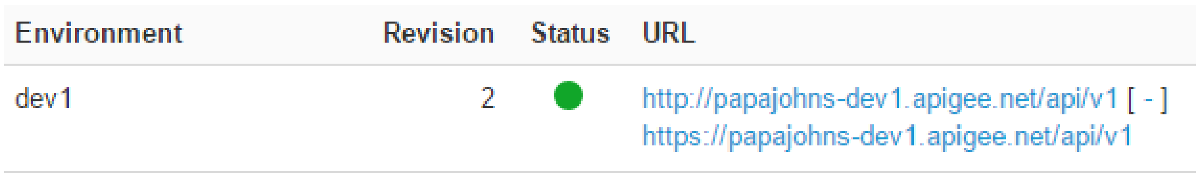
- Backend Supports multiple versions, Apigee does a pass through
- Only advantage of this is visibility and analytics
- Backend Supports multiple versions, Apigee routes to the appropriate version
- Apigee does content/context based routing base on incoming request based on various parameters like URL/Query Parameters/headers/payload elements/client ip/authentication information etc. This will give flexibility on how you manage the versions at the backend (like changing the URL paths, moving them to new hardware etc.)
- Backend supports one version, Apigee does the mediation
- In cases where it is expensive to manage multiple backend versions, Apigee can mediate the request/responses. This is recommended as long as the mediation rules are straightforward (and simple)
- Apigee gives an error for API requests for older versions

Sun-setting can be done in phased manner as follows

- **Phase 1** - Apigee adds a header or some other non-intrusive way- to tell the developer that the API will be depreciated
- **Phase 2** - Apigee can introduce stringent rate limits on older version
- **Phase 3** - Apigee can block the requests and send the appropriate error message

11. Security

By default Apigee proxy endpoint creates with both HTTP and HTTPS virtual endpoint. It is good practices to disable HTTP endpoint if not needed



Environment	Revision	Status	URL
dev1	2	●	http://papajohns-dev1.apigee.net/api/v1 [-] https://papajohns-dev1.apigee.net/api/v1

The above image gives a better understanding on the default virtual hosts

<VirtualHost>secure</VirtualHost>
<VirtualHost>default</VirtualHost>   //Can be removed, if not in use

- It is good practice to attach spike arrest and quota policy if traffic pattern to API proxy is predictable. It protects APIs from traffic spikes and DDoS
- For partner oriented APIs, it’s a good security practice to do ip whitelisting and access control policy to allow only certain ip/subnet range to your APIs
- Any external developer facing API should have verifyapikey policy, as its first policy when it is exposed on developer portal
- Connection to API proxies &/ API proxies to backend connection should be protected with TLS based security