apigee

# A Conceptual Guide to Programming Apigee Edge

*First Edition*
*Nov 2016*

# Preface

You are an awesome developer. You have a new task, that of building and exposing your company's APIs on Apigee Edge. Apigee has great documentation (http://docs.apigee.com) and tons of articles on (http://community.apigee.com ) to help you build APIs fast and help you succeed.

As with programming any new system and/or language, over time, you will build a deeper understanding. The purpose of this e-book is to help shorten that cycle, help you become an expert Edge developer, sooner than later.

Personally, having a deeper understanding of the system or the language that I am building on / in, early on, made me design and build better programs. Couldn't imagine building without books like "The Design of the UNIX Operating System by Maurice J Bach" , "The C Programming Language by Kernighan & Ritchie", "Inside the C++ Object Model by Stanley Lippman", "Expert C Programming: Deep C Secrets by Peter van der Linden" , "JavaScript The Good Parts by Douglas Crockford" etc. That's exactly what this book aspires to do, to equip you with a conceptual understanding of the Edge platform.

Part I talks about the design of the platform, specifically the core requirements and the design principles that heavily influenced the design.

Part II presents the the language, the runtime and the packaging model.

**What does this not cover?**

The goal is not to cover all features of Edge.

This is not a step by step guide or tutorial. There are tons of great samples, community articles, video tutorials and documentation out there to help you do that.

**Many Thanks**

To the engineering team for maintaining notes about the product design. Threaded conversations are much better than a static design docs.

Apigee has great documentation (http://docs.apigee.com) and tons of articles on (http://community.apigee.com ). Thanks to all of you Edge practitioners for writing those great articles on community.

Thanks to my colleagues at Apigee, Rajesh Doda and Anil Sagar for putting up with my barrage of questions and for passionate conversations. I would like to thank Senthil T, a member of the original implementation team, for going down memory lane to debate about why something has been done a particular way. Threaded conversations and blogs don't capture everything!

Many thanks to Steve Richardson, Sean Davis, Ricardo de Andrade, Nigel Walters, Will Witman, Brian McNamara, Vidya R, Terry David and Sudheendra for going through the alpha version of this e-book and providing feedback. It takes lot of effort to read alpha versions and passion to make it better.

I sincerely believe that this will help you become an expert programmer on Edge sooner than later, equip you to understand the behavior of Edge, issues or errors, and finally, a way to think about your favorite pain point about the system (you will always have one!).

Srinivasulu Grandhi
(sgrandhi@apigee.com)

# Table of Contents

# Part I – Design of the Apigee Edge Platform

*Successful digital businesses have two things in common (a) They deliver amazing digital experiences and (b) They run fast and are agile. They continuously expand digital interaction surface area for customers, employees and partners.*

*APIs power digital experiences.*

*Let us say that you are building a ride share app. Instead of building your own map service, you can, for example, integrate with google maps. Instead of building your own user authentication service, you can, for example, provide "Login with Facebook" option from within your app. To do the above, all you have to do is sign up with google and get a key to call their maps APIs from within your app. The same goes for facebook. APIs speed up delivery of digital experiences.*

*Imagine a similar (self service, frictionless) mechansim to build apps that integrate with any service or system within your enterprise. Imagine that your partners use a similar mechanism to integrate with your organization. That certainly increases the speed with which you can bring new digital experiences to life.*

*Going a step further and adapting the new ways to build and run software (e.g. CI/CD, Devops, Cloud etc – aimed at removing the friction) further speeds up your software delivery. While it is true that increasing organizational speed and agility is a much broader challenge, your ability to deliver software quickly is a critical component.*

*An API platform built to support and enhance the speed with which you can deliver digital experiences is key to your Digital transformation.*

*What exactly are the requirements of such an API platform? The chapter "Apigee Edge" walks you through a discovery process to unearth the need and requirements of an API Platform.*

*The chapter "Design of Apigee Edge" walks you through the key requirements and the design principles that influenced the design of Edge. Having such an understanding of the platform will enable you to build APIs better.*

# 1. Apigee Edge

Edge is an API platform through which you conduct all your digital interactions.

Digital interactions are the various applications (mobile or web) that your customers, partners and employees use to interact with your organization, your organization's systems of record. These interactions happen via APIs. Apigee Edge enables you to rapidly build, deploy, run and manage these APIs.

This chapter walks you through the discovery of the need for such a platform.

## Need for an API Platform

Let us say that we are part of a retail company and are tasked with building a mobile commerce application. Let us further say that we already have the following set of backend systems (also referred to as Systems of Record) that provide the core functionality.

- CRM – Uniquely identifies a customer. Has their profile information. Keep tracks of their previous purchases. Exposes a REST interface.
- Product Catalog – It is a set of systems that collectively provide information about products, prices and offers. Their functionality is exposed as a collection of SOAP Endpoints through an ESB layer.

In addition, we are going to use a partner system for processing payments.

- Payment Processor – Provided by our partner. Exposes SOAP Endpoints.

Key architectural decision

In designing this mobile application (henceforth called 'app'), one of the key decisions we have to make is whether the app directly talks to the backend systems **Or** whether it goes through an API layer.

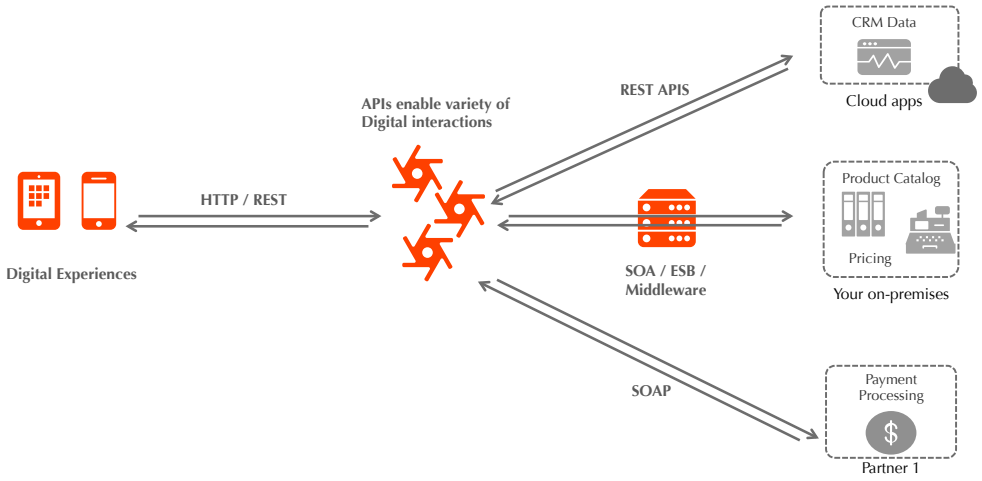
Here are a few considerations that would influence that decision

- Should a change to my backend systems also require an app upgrade?
- Should we worry about low bandwidth connections between the app and backend systems? If yes, would invoking SOAP calls directly from the app be the best approach?
- Do our existing REST and SOAP Endpoints provide the right level of abstraction for my app's needs or do we have to do additional processing on my app? If later, it only increases the data size on the wire but also may reduce the battery life of the user's mobile device.
- It is very likely that we may want to build additional applications. Would encapsulating common functionality within an API layer enable better reuse?

Logical Architecture

Let us say that the above considerations resulted in us choosing to split the app's functionality across the app and the API layer.

Specifically we would create a set of RESTful APIs that provide the right level abstraction to our application and host them on a web server or app server of our choice. These APIs in turn talk to the CRM, Product Catalog and the Payment Processor backend systems.




Reliable scaling

Let us say that our customers love the app. We expect more and more people to use it. We expect more and more transactions to come through the app.

That means we have to scale our API layer and the backend systems reliably to be able to handle the increased load. “Scaling out” the API layer (run multiple instances of our API servers) and additionally caching data at the API layer should get us pretty far in supporting the increased load.

If this still does not solve, (i.e. one of our backend systems becomes a bottleneck) before we decide to rewrite or modify our backend which typically is expensive (time and effort), we should explore whether changing the API semantics or app features gets us the scale. As an example, every time a customer orders a product, checking and reserving that quantity would be a bottleneck at scale. If we can source a subset of products just in time to deliver, we could skip this step. During peak traffic time, based on the customer’s prior history, should we accept orders without immediate payment confirmation? Some of these would help us get far without rewriting our backend systems.

Ideally, we should scale out as traffic increases and scale down as traffic comes down to save resources.

Expanding to popular Digital Channels

Its great that our customers love our app. It turns out that they also use shopping comparison apps before they decide to place an order with us. Wouldn’t it be nice for our customers to fulfil their orders right from that shopping comparison app?

We have to provide a way for those app developers to program to our APIs. If we have to do this at scale (i.e. across many 3rd party apps), we have to make this self-serviceable.

Should we be providing premium functionality to some of those 3rd party apps? Maybe build tight partnerships with a few wherein they get access to our customer's profile data to provide richer experience to our customer or offer a combined loyalty program. We have to provide a way for partners to only get access to APIs but we need to let our business teams decide the level of access.

Security

Beyond authentication and authorization of apps and users, we need to ensure that the API layer is highly secure. We need to code our API layer to protect against well known attack vectors and keep it current as new attack vectors are identified. We need to identify and block DDoS attacks. We need to identify and maybe block bots from accessing our data.

Becoming a Digital Business

With the success of this app, our company wants to expand the range of digital interactions it provides to its customers, partners and employees. A lot more teams are now building apps. How can we get those apps out quickly and of course, successfully deal with security and scale?

Some of the new capabilities that our company has to build or solve for are

- Enable multiple teams to build APIs and experiences in parallel.
- Is there a self service model for all those teams building APIs? e.g. enable API developers to do CI/CD. E.g. enable developers of one division or team to discover, signup and use the APIs of another team in a self-service way.
- Reduce the time it takes to build APIs.
- Reduce the time it takes to adapt APIs as new devices come to life.
- Learn about who is using APIs and optimize accordingly.
- Is there a self-serviceable model for our API business teams to expose some of those APIs as products for either specific partners or any outside developer?
- Charge apps for using some subset of those APIs.

API Layer to API Platform

Looking at our logical architecture diagram above, what started out as a simple API layer hosting a few APIs for the initial mobile app grew into a full fleged platform for APIs across the company. Architecturally, an API platform is a new tier within your enterprise IT architecture that sits on your trust boundary and is key to your Digital Transformation.

What is interesting to note is that both architecturally and functionally, none of the existing IT architectural layers (and systems within those layers) would fit the bill. i.e. would be fit to be an API tier.

While many enterprise middleware platforms provide RESTful Endpoints, they were never intended to solve for requirements we discovered in our journey above. They are absolutely great at what they do. For example, if your backend doesn't expose services by default, an ESB with rich array of connectors and integration capabilities may be the right choice to service-enable that backend. But they never were built to be an API tier – They don't provide for the capabilities listed above. Most likely, they were never built to be directly exposable to the internet. Most likely, not built on cloud (like) infrastructure. Most likely, don't support zero downtime deployments (which was not a concern for when they originated) etc. Their design considerations were vastly different from those of the needs of an API layer.

You need an API platform built ground up for the requirements we discovered above. That is what Apigee Edge set out to solve and solves for you.

Not just the requirements but the API platform needs to support and enhance the speed with which you build software. If you are a company looking to build software the way digital natives build (agile, small teams, self service, cloud-like infrastructure etc), Apigee Edge feels very natural to you. It brings the same coolness to building APIs and accelerates your digital journey.

## Edge Use Cases

Here are the key use cases for Edge

1. Build and launch your APIs super fast
    a. A platform that sits on your trust boundary to allow your services to be consumed through APIs.
    b. Build APIs really fast.
    c. Securely expose your services to the outside world as APIs. Control who can use it and how much. Get visibility into the usage. Automatically handle traffic increases with 99.99% availability.
    d. Support the new software development mechanisms. Small self contained teams – Devops, CI/CD, Self service etc.
2. Manage a complete ecosystem around your APIs with multiple developers, partners collaborating via your own community. Full self service to onboard new developers. Enable your API business teams to expose APIs as products etc.
3. Monetize your APIs
4. Sometimes, as part of building digital experiences, you need to build some new backend services. You could do it elsewhere or you could do it on Edge
    a. Your mobile apps may need to manage user preferences. You need a new persistence layer and a logic. You could build such a micro service on Apigee using Apigee's persistence capabilities.
    b. Your backend service may need to be enriched with some new data. Eg – your store locator database at the backend may not have lat/long stored and for various reasons it may be faster adding that data outside of that database

*Note – Edge is not a general purpose PaaS but we have found that our customers needed to quickly build some micro services) and having this capability was pretty useful and helped them deliver digital experiences super quicker.*

**Recap**

- An API Platform is key to your digital transformation.
- It needs to be built ground up for cloud scale.
- It needs to support the new ways of building software (small teams. CI/CD. Devops etc) to help you deliver digital experiences faster.

# 2. Design of Apigee Edge

This chapter lists the core requirements of Edge, the runtime architecture and the design principles that influenced the architecture.

Having this level of understanding will enable you to reason about Edge behavior and helps you become an expert Edge programmer faster.

## Core Requirements

1. Given that Edge is a “proxy” to your systems, the core functionality is to
    a. Read an incoming request (API call).
    b. Validate (e.g. Is the app allowed to call this API?)
    c. Enhance or Transform the request if required
    d. Send to a target system
    e. Read the response
    f. Transform the response if required
    g. Send back to the client who made the API call.

2. Optionally, as part of 1c, Edge should have a capability to call multiple other systems and use those responses to enhance the original request before sending to target.

3. Should enable building the API logic really fast.

    For the most common functions, Edge should provide out of the box support and a declarative programming model so that APIs can be built quickly. In addition, Edge should allow developers to write custom code using JavaScript, java or python.

    Prototyping, iterating, deploying and redeploying should be simple and quick.

4. Should be available as a cloud platform as well as deployable on-premises.

5. Support the customer’s agile mechanics of building , deploying APIs (how are their teams organized ; how do they want the isolation? What tools do they use for CI/CD? etc).

6. Provide comprehensive security capabilities (e.g. allow only authorized applications to call APIs ; protect against common threats ; support for oauth ; traffic management etc)

7. Support cloud scale – both runtime scale for APIs as well as scale in terms of #customers on the platform, scale in terms of #developers building on this platform, scale in terms of #analytics data etc.

8. Provide 99.99% availability. What this means, for eg, is that Edge should support zero downtime deployments, should deploy in multiple geo-regions and should fundamentally limit or contain the impact of failures (e.g. isolation not only between customers deployments but within)

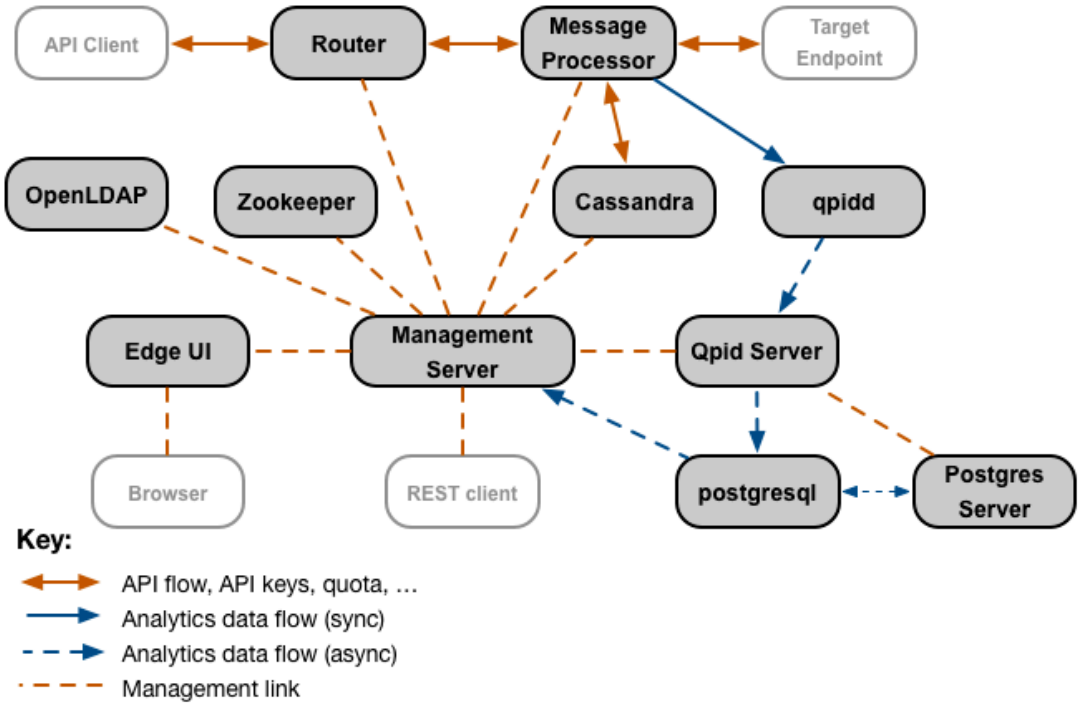
9. Support multi-geography API deployments for minimal latency.

10. Support multiple versions of the same API (maybe for things like A/B testing or to support blue-green deployments)

11. Provide developer services (e.g. billing. Self service for getting access to APIs. API documentation etc)

## Edge Runtime Architecture

Here are the key runtime components that make up Edge




| | |
|---|---|
| API runtime | APIs pass through Router and Message Processor before hitting the actual target.<br><br>Message Processor is where your API processing logic runs. Scaling out means running many instances of Router and Message Processor. Each instance of the Message Processor runs a complete copy of your API processing logic. |
| API state | The most commons state that your API processing logic needs to manage across API calls is “access tokens”. If you decide to apply an API quota across all API calls (which are processed by multiple Message Processors), of course, you have to persist it. Cassandra serves as that underlying storage system for all such persistent data.<br><br>Other than above, most of the API processing logic is typically stateless. In the event that your API processing logic needs to store API specific data across API calls, it can store it in Cassandra. |
| Config Subsystem | OpenLDAP is the authentication and authorization store for Edge (management) users<br><br>zookeeper is the global store for the runtime topology. For example, a particular customer’s APIs may be deployed to multiple regions (to satisfy both latency and availability needs) which means that each region will have a set of Routers, Message Processors and Cassandra (for state management).<br>One region may have higher number of Message Processors than other. Depending upon the load characteristics and SLAs, these |

| | |
|---|---|
| | instances maybe shared across customers within a region or be dedicated instances. All of these details are stored in Zookeeper.<br><br>Your API deployment artifacts are also stored in cassandra. Depending upon the load characteristics, this could be a different cassandra or the same as used for API statement management.<br><br>Management Server provides APIs to interact with Edge.<br><br>Edge UI is the UI interface for performing operations tasks and building and deploying API artifacts. |
| Qpid, Postgres, Spark | Provide API analytics. Depending upon the load characteristics and SLAs, these can be configured individually for each customer or shared across customers. |

## Design Principles

Given the "core requirements" for an API Platform, one could conceive many ways to architect it. What drove the above design? Why did the team built it that way?

Design principles help the team make consistent decisions and tradeoffs thereby building a coherent system. Here are the key principles that influenced the design of Edge.

1. Ensure that one customer doesn't impact other customers on the platform. Isolation trumps everything.

2. It's critical that the system adds the lowest possible latency to the API flow. What this means is that once an API call is read for processing, continue to process till you hit a wait point (I/O) at which point you pick up the next API to process. That may make an incoming API call wait a bit more to be picked up for processing, but that's a scale-out problem.

   Unless absolutely required, reduce the need to go across the network which means that a Message Processor should have the complete information in memory to finish processing an API call.

   Each processing node should utilize the resources effectively. For a given capacity, it should be able to pick up the right level of concurrency (#of API calls to process at the same time) while staying within the acceptable latency levels for each API.

3. No matter how much we think through, we will learn a lot as the scale increases. We need to be able to super quickly configure and reconfigure a particular customer's runtime topology to reliably process the API calls.

   Ideally you want the entire system to be highly available but availability of the API runtime is absolutely more critical than availability of configuration and management components. You absolutely don't want API traffic to be disrupted but it's temporarily ok that one can't deploy a new version to the cloud or to not be able to see a particular report.

   The above may mean that the runtime and config subsystems are loosely coupled with runtime pulling information as and when needed. This may mean that when a user makes a configuration change, there may be a delay before it gets picked up by relevant nodes. In some rare occurences, that configuration may not be picked up by all the nodes. As long as there is a way for a user to query the state and trigger a force-read without downtime, it's acceptable.

Operational flexibility will be key as we scale our customer base and traffic.

4. API everything. There must be an API for everything you want to do with Edge. We expect others to build IDEs or plugins to popular IDEs to build and deploy to Edge. We expect on-premises customers to integrate Edge into their monitoring and operations infrastructure through APIs.

   This flexibility might increase the configuration complexity but is crucial to gain broader developer adoption and fits the tools developers and customers use than forcing them to use just one tool.

5. If you were to struggle between “Here’s a great underlying technology to use for cloud but that may not be (yet) a preferred infrastructure for on-premises (e.g. Our customers may not know how to manage cassandra for their on-premises”, always optimize for cloud scale deployment. We deeply believe that our customers on-premises setup will eventually look like cloud.

6. In general, you want a behavior that “if writes succeed, reads should return that new value”. EG If I revoked a key successfully, every subsequent API call with that key should fail.

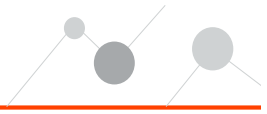
   At scale and at deployments spanning multiple geographic locations, that strong consistency may be a bottleneck. Strong consistency adds to failures when partitions occur.

   In some cases we know that eventual consistency is fine (e.g. you updated a cache in one location and it’s ok that for next few minutes that a second region’s cache is not updated!). In other cases, it may be acceptable (allowing the actual API calls slightly beyond the set quota value).

   We need a way to be able to switch between strong and weak consistency depending upon the specific use cases, system behavior and customer asks. If you have to pick up one strategy and when in doubt, choose eventual consistency model. Scales well.

7. All of the above are more important than optimizing the infrastructure for lowest possible cost. Cost is not the key design driver. Speed of delivery is.

8. Don’t invent a declarative language to specify customer’s API processing logic. Developers should be able to use the languages that they are already familiar with to code the API processing logic.



# Part II – Apigee Edge Programming Language

*Apigee Edge is an API platform through which you expose your services, as APIs, for others to consume and manage your API offerings.*

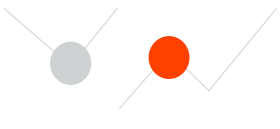
*Your API typically is a façade to a set of backend services to provide consumption specific APIs. What is the language used to build your API logic?*

*Apigee Edge provides an*

- *extensive set of predefined functions that are most commonly needed to build API facades. Using these speeds up your API delivery.*
- *an XML specification for describing your API logic using those common functions AND*
- *if required, a way for you to write custom code using Java or JavaScript or Python.*

*Your API processing logic is a set of XML files and optionally code written in Java or JavaScript or Python. To successfully build you need (a) an understanding of the XML specification and (b) the Edge runtime constructs. This is what subsequent chapters will equip you with.*

*Note – If you haven't built a "hello world" API on Edge yet, go ahead and do that before you read further. [http://docs.apigee.com/tutorials/add-and-configure-your-first-api](http://docs.apigee.com/tutorials/add-and-configure-your-first-api) helps you quickly build one. It helps to experiment with the concepts as you read them.*



# 3. Overview

Successfully programming in a new environment requires an undertanding of

a) the runtime context within which your program runs
b) the language within which you specify your logic and
c) the structure of your program (both build time and deployment format)

For example, building a successful service on Linux requires an understanding of

a) Runtime Context - How do I ensure that whenever Linux boots, this particular service needs to be run? How can I pass environment variables to my service? How can my tell service inform Linux that it failed etc.

b) Language – Let us say that I decided to build this in “C”. I need to understand the language to code. I need to understand how to group code into reusable functions. I need to know how to represent data (e.g. arrays). I need to understand the standard libraries available etc.

c) Structure – Linux would mandate a binary structure that my C compiler generates. Fortunately I don’t have to worry about except that I need to know a few things like (a) There exists a global namespace where I can store global data variables (b) That the first function to get called is main() etc.

This chapter provides you an overview of the API runtime, the structure of your API program, how control flows through your API Program and the API Context (data).

## API Runtime

Edge is a multi-tenant, cloud scale API platform that allows you to rapidly build APIs and expose, manage them. APIs typically represent a façade to one or more of your backend systems for easier consumption.

Edge hosts your API logic. It provides a language for you to declaratively define your API logic and/or code your API logic in any of JavaScript, node.js, java, python languages.

It provides a runtime container (or engine) within which your API logic runs. Container provides the runtime isolation required in a multi-tenant environment, handles a set of tasks on your behalf to help you build APIs faster.

Before you jump in and explore the language to write your API logic, its critical to understand what the Edge runtime container does , when it invokes your API logic and with what context. Best way to understand this is to look at, generically, what needs to happen for every incoming API call.

1. Read an incoming request.
2. Ensure that there is a particular program deployed on Edge to process this particular request
3. Validate (e.g. Is the app allowed to call this API? exceeded the quota limits? etc )
4. Enhance or Transform the request if required
5. Send to a target system
6. Read the response
7. Enhance or Transform the response if required
8. Send back to the client who made the API call.

As part of 4 and 6, you may want to

- Check if the cache already has a valid response to send back
- Change the message headers/content
- Collect statistics
- Perform some custom tasks like fetching authentication details from external server, etc.
- Call out to multiple systems and use those responses to enhance the original request.
- ….

**API Runtime Container vs. Your API Logic (Flows and Policies)**

Edge run time takes care of all the above orange tasks (1,2,5,8). Of course, you have to provide Edge with the right configuration (e.g. The address to reach your backend (a.k.a. target)).

Blue tasks (3,4,7) are typically your API logic. Edge provides an exhaustive set of common capabilities needed in building APIs (e.g. Quota or XMLtoJSON). Edge calls these 'Policies' or more specifically 'Policy Templates'.

Typical of a procedural language, you define your API logic as a sequence of Policies (specific instances of Policy templates) that container executes one after another (I.e. Calls the next Policy only after the prior one finishes). Edge calls such a sequence of Policies as "Flows". You can think of "flows" as "procedures". Edge allows you to tag a policy or a flow with a condition in which case, only if that condition is satisfied, then that policy or flow executes (called "conditional flows").

Tasks 3 and 4 needs to be called during the request processing after Edge reads the request (and before sending the request to the target). Task 7 needs to be called during the response processing' after receiving the response from target (and typically before sending the response back to client). Edge provides a way for you to tag your flows as either "request" or "response" flows.

API runtime Container provides a context (or namespace) for each API call that provides your flows and policies with the default variables (e.g. request object). The context also stores the variables your policies and your code create. The context gets created when an API call enters Edge and lives till the response is sent back to the client.

Each container, while it executes your API logic sequentially and synchronously for each API call, has the capability to concurrently execute multiple API calls. Of course, it is typical to r un multiple instances of container to handle more TPS and provide high availability.

## Programming Language

Policy templates are XML specifications. Creating an instance of a policy is creating an XML file of that specification.

Edge allows you to write custom code using JavaScript or Java or Python policies.

Flows, again are specfied in XML.

Essentially, your API program within Edge is a set of XML files and optionally custom code packaged into an API Proxy.

## Deployment Package (API Program Structure)

You have to package your API logic in a format that Edge container understands and deploy that package to Edge to process your APIs.

Here are the principles that influenced the design of the package format

- To ensure the lowest latency possible, Edge runs an API call within a single runtime container. Of course, Edge scales out containers and each container handles multiple API calls concurrently but a single API call gets executed by a single container to ensure lowest latency.

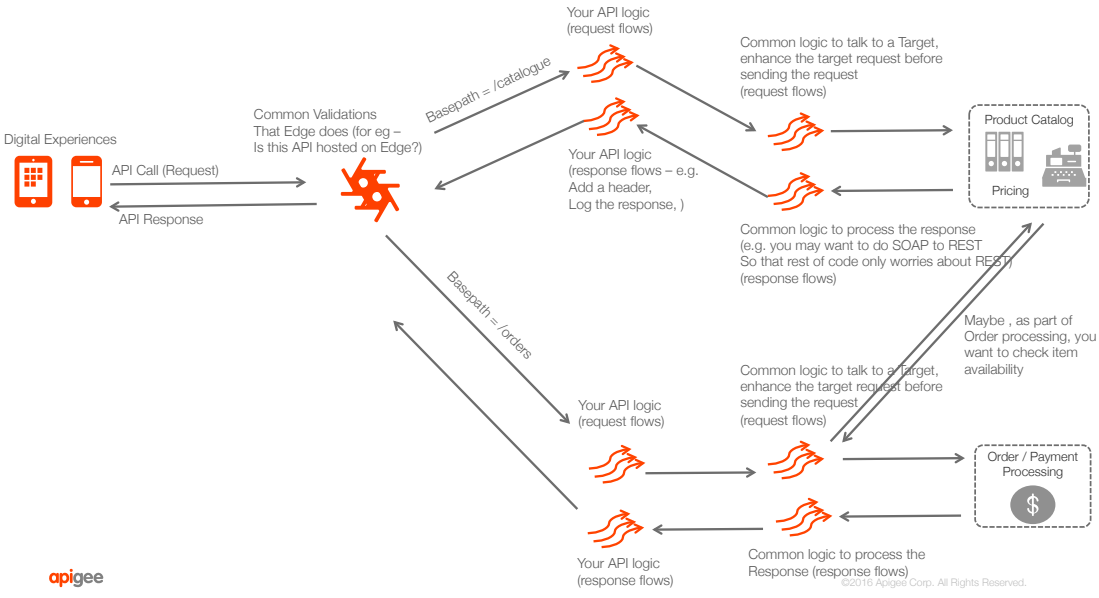
- A typical API project may have multiple APIs being built by a team. There may be common code across those APIs. You may want to partition the work across your team members in many ways (e.g. a few to write the logic to talk to a Target system). While its tempting to define a package format to have references to shared code , managing dependencies at runtime in a cloud scale environment is complex. Edge always favored runtime simplicity, sometimes at the expense of pushing complexity to the API developers.

For both of the above reasons, Edge enforces that your API logic for a particular API be a self contained package. Edge calls that self contained package a 'Proxy'.

**API Proxy**

Your API package OR executable, in Edge, is called an API PROXY (henceforth called Proxy). A Proxy encapsulates your API processing logic. For every incoming API call, Edge reads the request, parses, finds the appropriate Proxy, creates a context to hold the data variables and executes the code in that Proxy.

Lets look at an example to better understand the executable (package) structure.




In the above example, there are two distinct APIs – "…/catalogue" and "…/orders" with their own API flows. Each API talks to its own backend.

You could package the flows up as

- Two separate Proxies - 1 to process the basepath "/catalogue" and another to process the base path "/orders".
- One single Proxy to process both basepaths.

*Question – On what criteria would you choose to build above as two proxies vs one proxy? Before you build out your API project, its good to have a runtime model for your APIs (e.g. how many types of Proxies?)*

ProxyEndpoint AND TargetEndpoint

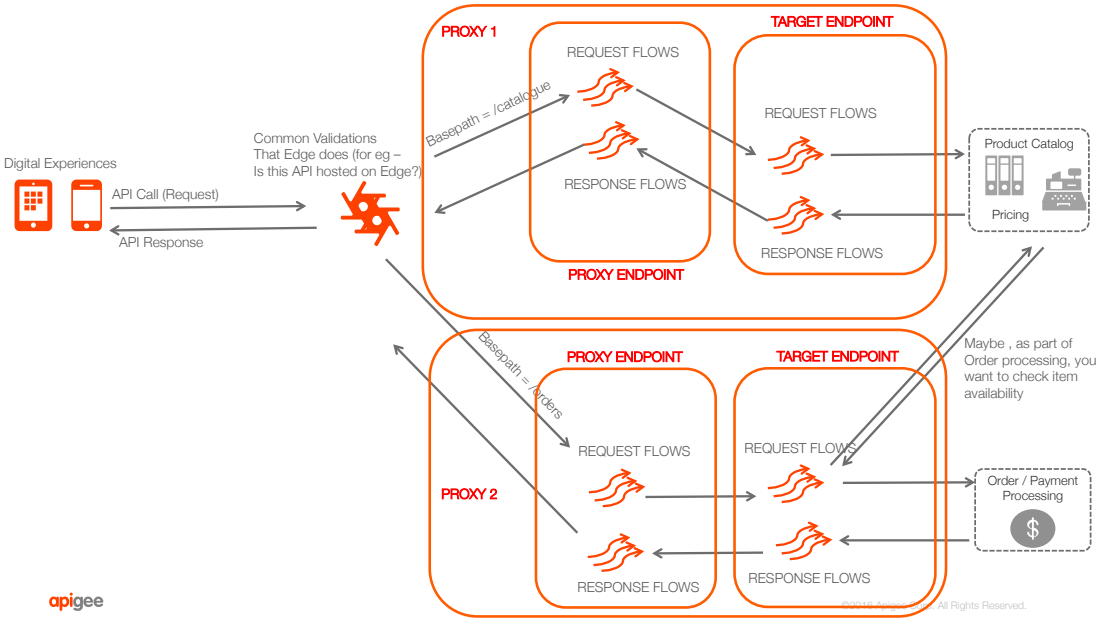
Each API Proxy has one more ProxyEndpoints and one or more TargetEndpoints. Think of a ProxyEndpoint as a code encapsulation to handle a specific API basepath. Think of a TargetEndpoint as a code encapsulation to handle a specific target. Every Endpoint has flows, fault rules and network settings.

Isn't tagging "flows" as "request" or "response" sufficient? The container executes the flows sequentially in the order they appear in the package. Why does Edge provide a way to further classify and group them within ProxyEndpoint or TargetEndpoint?

While the package format is optimized for runtime (i.e. entire logic to process an API is within the Proxy), your development teams may be structured in different ways? You may have a few developers writing "flows" to talk to your target systems. You may have few others writing common validation "flows". You may have common reusable "flows" for doing specific tasks (e.g. Build SOAP payload). Appropriately naming these flows and grouping them enables faster parallel development and easier composition of the final Proxy package.

Yes, two proxies may have the same "TargetEndpoint flows" embedded within them, as the package is self-contained, but in your development environment, you can manage that TargetEndpoint code as a single entity.

Assuming that we model the above example as two different Proxies, here's what the the runtime structure would look like



PreFlow and PostFlow

ProxyEndpoint and TargetEndpoint provide two standard flows called PreFlow and PostFlow within which you can insert your API logic.

## Deploying a Proxy

When you sign up for a trial account on https://enterprise.apigee.com, you get one organization and two environments (test and prod).

*Note – Depending upon the license that you bought, you get multiple organizations and environments within them.*

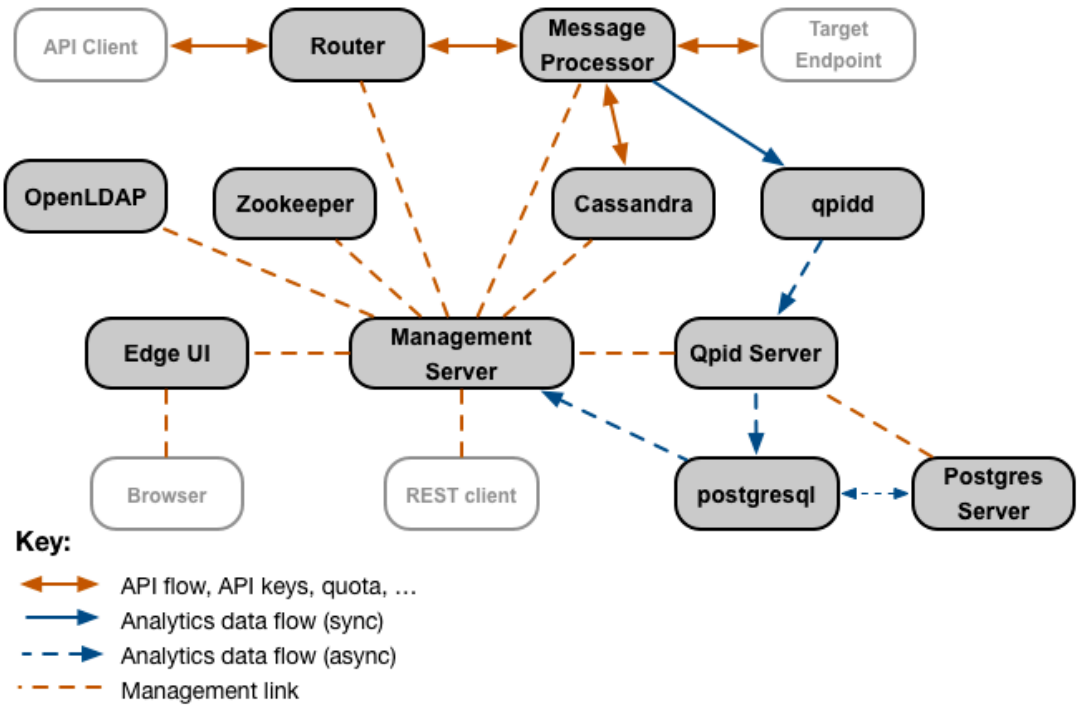
Each Proxy handles API calls for a particular base path within your API set. Edge allows you to save multiple revisions of an API Proxy as you build. You deploy a particular revision of an API proxy to an ORG-ENV.

Edge allows you to have multiple active deployments of an API Proxy but each has to be with a different base path. This capability is typically used to support multiple versions of a particular API

## Proxy Runtime

Edge has a runtime process called Message Processor (henceforth referred to as MP). MP is the API container.

It is typical that every organization has more than one MP, so when you deploy your API Proxy to an ORG-ENV, it gets deployed to all the MPs configured for that particular ORG-ENV.



## Configuration Data

**Common Entities**
Edge provides a comprehensive list of common configurable data entities that further speed up your API development and makes it easy to manage your API Program. For example

- The target URL used by your Proxy, most likely, would be different within a test environment versus a production environment. It is a common practice to read this value at runtime than to hardcode. You can accomplish this using TargetServer configuration within an environment.

- How would your developers get access to your APIs? Do you want to set different quota limits for different developer apps? Do you want to package up your APIs as products and price them differently? Edge provides a set of configurable entities for you to configure these at the organization level.

**Your own Environment Variables**
Sometimes, your API Proxy may need additional custom environment variables.
Edge provides a data store called KVM (Key Value Map) that you can use to create your own configuration data items. You can create multiple Key Value maps. A particular KVM can be scoped to either a specific proxy or an environment or to an organization (shared across all environments within that Org). For example, if you have multiple proxys that share the same config data, you create a KVM at the environment level, the environment where those proxies are deployed.

## API Runtime data

**Current API Message Context**
For every incoming API call, Edge reads the request, parses and finds the appropriate MP to execute the Proxy. The MP then creates a message context to hold the data variables and executes the flows within that Proxy.

If there are N concurrent API calls to an API that needs to be processed by a particular API Proxy, MP executes those in parallel by calling the same API Proxy, each with its own distinct message context (variable set).

*Note – Strictly speaking, the number of API calls an MP can process parallely depends upon the reources on which it is running (#Cores, #Memory) and on the number of I/O wait points within a Proxy flow (e.g. If your Proxy, as part of its processing, has made an external call and is waiting to get a response, MP may context switch and run another one rather than busy wait for the response).*

Policies and flows use this 'per API call' or message context as the namespace to read and write data. This means that all the data variables that get created as flows execute are scoped to the per-API context.

Apart from Edge container and the policies that create variables or set variables as they execute, you can also create/read/write variables as part of your flows (by using a policy like `AssignMessage` OR within custom code written in `JavaScript or Java Callout or Python` policies).

Here are some commonly used objects

- `context` wrapper for the message context and flow
- `client.request` wrapper for the client request
- `request` wrapper for the HTTP TargetRequest message. Initialized with client.request

- `response` wrapper for the HTTP Proxy Response. Initialized from the response received from the target.

**State beyond Current Message Context**
Sometimes you need state beyond a particular API call. EG. If you want to enforce a limit on number of API calls a particular developer app can make per hour, then you have to keep track of that count. Apigee Edge policies have access to inmemory storage, scoped to a Proxy within an MP where they can store such counters.

Apigee Edge also provides a global (across MP) storage where both Edge policies as well as your flow variables can be persisted if required (e.g. KVM, Cache – can be used to store information beyond the context of the current flow and available across the MPs)

## Control flow

For every incoming API call, Edge reads the request, parses, finds the appropriate Proxy, creates a context to hold the data variables (e.g. the actual request object) and executes the flows within that Proxy. Here's how the control flows through the 'flows'

Request Processing

```
ProxyEndpoint :: Request PreFlow
        ProxyEndpoint :: Request Conditional Flows
                ProxyEndpoint :: Request PostFlow
```

Depending upon the `<RouteRules>` specified within the `<ProxyEndpoint>`, the runtime selects the appropriate `<TargetEndpoint>` to execute.

```
                        TargetEndpoint :: Request PreFlow
                                TargetEndpoint :: Request Conditional Flows
                                        TargetEndpoint :: Request PostFlow
```

Response Processing

```
TargetEndpoint :: Response PreFlow
        TargetEndpoint:: Response Conditional Flows
                TargetEndpoint:: Response PostFlow
                        ProxyEndpoint :: Response PreFlow
                                ProxyEndpoint:: Response Conditional Flows
                                        ProxyEndpoint:: Response PostFlow
                                                ProxyEndpoint:: PostClientFlow
```

`ProxyEndpoint:: PostClientFlow` is executed in parallel to sending the response to the client. A common step to execute within is to log the request / response to a log system.

*Note – There is no runtime benefit by coding a flow within, say, a PreFlow vs. having it as the first flow within the "Flows". It provides better code readability and aids in organizing your development effort.*

*Question – Every API program in Edge typically is modeled like a proxy – i.e. is assumed to send the request to a target system, get the response and send it back to the client. If your API provides an abstraction over multiple distinct backend systems, you can model such a program in two ways*

- *As part of processing, your program invokes various backend systems and constructs the final response to be sent. In this case, there is no specific target!*
- *Designate one of those as a target and handle the calls to other systems during processing.*

*Is there an inherent benefit to either one of these? Why did Edge pick the second pattern?*

# 4. Organization and Environment

Where do you deploy your API Proxy? An environment within an organization! *Lovingly called org-env!*

Everything within Edge is scoped to an organization and environment. For eg, users belong to one or more organizations. You deploy your API Proxy to a particular organization and environment. Each organization and environment scales independent of others. One might be a DN (distributed network where your Proxies are deployed at multiple data centers across the world) but the other may not be.

Your organization may have different teams responsible for different API sets and need isolation (both at development time and runtime). Your API team needs to integrate code on a daily basis, automatically perform tests and even push new code to production on a daily / weekly basis.
Org-env combination along with powerful management APIs supports the way you are organized and the way your API teams build / deploy and manage APIs.

*Question - As you build out your APIs on Edge, you should leverage the org-env model to help speed up your API development and manage your APIs. How many organizations and environments do you need?*

*Note – Each organization has a distinct host name (e.g. api.apigee.com). Two organizations can't share the same host name. If you have two different API projects in different organzations, you may want to pick the right host naming convention for your api Endpoints.*

**Key Definitions**

| | |
|---|---|
| User | Someone who is authorized to login to Apigee system. EG An API developer is a user of Apigee system. In the above diagram, users are represented by API Team. |
| Organization | Is an entity to which your configuration, runtime and data is associated with. |
| Environment | Is an API runtime instance within a particular organization. You deploy APIs to an ORG-ENV. Your API analytics data is scoped to an ORG-ENV. A user can have permissions to work on / with multiple ORG-ENV combinations. |

## Configuration Items

Refer to the documentation for complete list of the configuration items at the organization or environment level. Here we present a few that are relevant to the purpose of this document.

**Organization Level Configuration**

| | |
|---|---|
| Developer | Developers who consume (use / call) APIs deployed within an ORG-ENV. |
| Developer App | A runtime entity built by a Developer that actually makes the API call. |
| API Product | Configuration that describes how a set of APIs are exposed. EG. Your "free" API offering may have lower quota limits than your "paid" API offering. |
| KVM | Edge provides a data store called KVM (Key Value Map) that you can use to create the environment variables required for your Proxy. |

**Environment Level Configuration**

| | |
|---|---|
| Virtual Host | You can define any number of virtual Hosts at Environment level. The URL of a A Proxy specifies the which 'Virtual Host' and the 'Base Path' that it is interested in. |
| TargetServers | A Target server is an alias for a host configuration. Its akin to setting up an environment variable to hold the actual target name / protocol etc so that its not hardcoded in your code.<br><br>of course, you can define multiple TargetServers and use them from within a TargetEndpoint. You can also specify a load balancing strategy within your TargetEndpoint. |
| KVM | Edge provides a data store called KVM (Key Value Map) that you can use to create the environment variables required for your Proxy. |

# 5. Flows

A “Flow” is a sequence of any number of “Steps”, executed in order they are specified within the “flow”. A “flow” always executes in the context of “request processing” or “response processing” or “error processing”.

A “Step” is one of the policy instances. You create policy instances, may initialize that instance with specific data values (e.g. Quota limit in a Quota Policy) and attach them to the flows. A “Step” can be a custom code, say, written in JavaScript, instead of using a predefined policy.

Edge executes the policies in order they appear within a flow and synchronously (i.e. waits for one policy to finish before executing the next one). For example, if you use a ServiceCallout policy and are interested in reading the response, Edge makes the call, waits for the response, reads and parses it before proceeding to the next policy.

## Request Processing

For every incoming API call, Edge reads the request, parses, finds the appropriate Proxy, creates a context to hold the data variables (e.g. the actual request object received and the request that would be sent to the target, if an API product exists, then those config values from the API product are used to set various paramters like Quota etc) and executes the flows within that Proxy. Here is the order in which various flows are called during request processing.

*Note – Ensure that you have mutually exclusive API Products. In case where an incoming API call matches with more than one API Product, there is no predictable way to ensure which one of those will be used.*

```
ProxyEndpoint :: Request PreFlow
        ProxyEndpoint :: Request Conditional Flows
                ProxyEndpoint :: Request PostFlow
```

Depending upon the `<RouteRules>` specified within the `<ProxyEndpoint>`, the runtime selects the appropriate `<TargetEndpoint>` to execute.

```
                        TargetEndpoint :: Request PreFlow
                                TargetEndpoint :: Request Conditional Flows
                                        TargetEndpoint :: Request PostFlow
```

Edge populates the default “request” object at the beginning (before calling `ProxyEndpoint :: Request PreFlow`) with the incoming request from the client. After executing `TargetEndpoint :: Request PostFlow` Edge picks up the default “request” object and sends to target!

Of course, you can do multiple callouts in the middle by using the same or creating new request objects but ultimately you have to ensure that the object named “request” is correctly populated to be sent to the target.

Runtime provides a default object called “message” that is mapped to “request” object so that your policy code , if need be, can be reused across both request and response flows. Remember, the fundamental assumption is that during request processing, your flows would deal with the request you got from the client to be sent to target.

## Routing to Target

Routes are a way to express where the incoming request will be directed to among one or more of the defined target Endpoints based on some aspects of the request message. This can be achieved in the following ways through <RouteRule>

Static Routing

It is expressed as a simple route in the proxy configuration specifying the name of the target Endpoint configured.

```
<RouteRule name="Route_To_Twitter">
   <URL>http://api.twitter.com/</URL>
</RouteRule>
```

Transport properties, SSL etc can be configured with in HTTPTargetConnection

```
<RouteRule name="Route_To_Twitter”>
    <HTTPTargetConnection>
       <URL>http://api.twitter.com/</URL>
    </HTTPTargetConnection>
</RouteRule>

<RouteRule name="Route_To_CatalogueTarget">
   <TargetEndpoint>CatalogueTarget</TargetEndpoint>
</RouteRule>
```

Dynamic Routing

It is expressed as a set of route rules each having a condition. The first route which evaluates to true will provide the name of target configured.

```
<RouteRule name="Route_Basedon_Header">
     <Condition>request.header.routeTo = "fooTarget"</Condition>
     <TargetEndpoint>fooTarget</TargetEndpoint>
</RouteRule>
```

Overriding Target URL

Another way of achieving dynamic target URL is to set the variable "**target.url**" to a URL. This could be any valid HTTP URL. The target.url variable is visible in the TargetEndpoint :: Request flow i.e. a TargetEndpoint should have been selected. The following points summarize the behavior when target.url is set

- The connection settings defined in the TargetEndpoint are used (timeouts, keep-alive, SSL, etc)
- If the TargetEndpoint was configured for SSL but the target.url was set with a "http" url, then the SSL settings are ignored
- If the TargetEndpoint was configured for Non-SSL but the target.url was set with a "https" url, then default SSL settings are used.

Script Target

Apigee Edge allows you to run node.js servers as targets within the API Platform. Here’s a way to define it within a TargetEndpoint

```
<TargetEndpoint>
    <ScriptTarget>
       <ResourceURL>node://hello-world.js</ResourceURL>
    </ScriptTarget>
</TargetEndpoint>
```

Routing to another Proxy Endpoint

A Proxy is a self-contained deployment and runtime entity. This means, that the entire API logic of a Proxy is embedded inside the Proxy. While it has its advantages (e.g. deployment @ scale and performance), it is not conducive to patching. For example, if you have to change some flow within a TargetEndpoint which is embedded inside five proxies, all of those five proxies have to be upgraded.

Edge now provides a way for one Proxy to call another Proxy l(ike a LRPC call). Here's an example.

```
<TargetEndpoint name="CallCatalog">

    <PreFlow name="PreFlow">
        <!-- PreFlow policies -->
    </PreFlow>

    <PostFlow name="PostFlow">
        <!-- PostFlow policies -->
    </PostFlow>

    <LocalTargetConnection>
        <APIProxy>Catalog</APIProxy>
        <ProxyEndpoint>default</ProxyEndpoint>
    </LocalTargetConnection>

</TargetEndpoint>
```

*Question – Your proxies can call other proxies but within the same organization and environment. Why? The entire message context is available for the called Proxy. Its like two functions sharing the same global data segment!*

What if your Proxy doesn't have a target defined?

What happens if I don't have a target? Or what happens if the routing rules don't result in a target? Edge enters "response processing" with a default response object (The request is echoed by the container). The request and response can be accessed/modified by attaching policies at proxy in request/response paths.

```
<RouteRule name="noneed">
</RouteRule>
```

The response can be overridden by attaching assign message policy at proxy response.

*Critique – While you can define the target host directly (within either URI tag or HTTPTargetConnection from within a RouteRule, you cannot define a ScriptTarget or a LocalTargetConnection within a RouteRule. It has to be defined within a TargetEndpoint.*

## Response Processing

For every response coming from the target, Edge reads the response, parses, finds the Proxy that originally sent the request, retrieves the saved context from the request processing (so that all your variables that were created during request processing can be accessed) , adds a new response object to the context and kicks off the response flows.

```
TargetEndpoint :: Response PreFlow
        TargetEndpoint:: Response Conditional Flows
                TargetEndpoint:: Response PostFlow
                        ProxyEndpoint :: Response PreFlow
                                ProxyEndpoint:: Response Conditional Flows
                                        ProxyEndpoint:: Response PostFlow
                                                ProxyEndpoint:: PostClientFlow
```

Edge populates the default “response” object at the beginning (before calling `TargetEndpoint :: Response PreFlow`) with the incoming response from the target. After executing `ProxyEndpoint:: Response PostFlow` Edge picks up the default “response” object and sends it to the client who made the API call in the first place.

It is the responsibility of the policies within `<Response>` flows to ensure that the “response” object correctly represents the final response to be sent back to the client.

The default object “message” is mapped to “response” object so that your policy code , if need be, can be reused across both request and response flows! Remember, the fundamental assumption is that during response processing, your flows would worry about readying the response you got from the client to be sent to target.

`ProxyEndpoint:: PostClientFlow` is executed in parallel to sending the response to the client. A common step to execute within is to log the request / response to a log system.

## Flow Syntax

Request flows appear within `<Request>` tag. Response flows appear within `<Response>` tag. As stated earlier, these flows appear with `<PreFlow>` , `<Flows>` (conditional flows) and `<PostFlow>` tags within either `ProxyEndpoint or TargetEndpoint.`

### Flow

```
<PreFlow name="ValidateKeys">
    <Request>
        <Step><Name>VerifyAPIKey</Name></Step>
        <Step><Name>BuildRequest</Name></Step>
        <Step><Name>ServiceCalloutGetMockResponse</Name></Step>
     </Request>
     <Response/>
 </PreFlow>
```

`<PostFlow>` and `<Flow>` follow the same syntax as above.

Edge calls policies in sequence. Until a policy finishes, Edge doesn’t execute the next one. There is no parallelism within an API call.

### Conditional Steps and Flows

A typical use case for conditional steps / flows is to use different processing for different requests (types , params etc).

For example, Paths are usually used to classify requests into REST "resources" wherein you want to enforce different policies on different resources. This is achieved using conditional flows with the condition being a path match condition.

Here is an example of a conditional step

```
<PreFlow name="ValidateKeys">
    <Request>
        <Step><Name>VerifyAPIKey</Name></Step>
        <Step><Name>BuildRequest</Name></Step>
        <Step>
            <Condition>(request.verb == "GET")</Condition>
            <Name>ServiceCalloutGetMockResponse</Name>
        </Step>
     </Request>
     <Response/>
 </PreFlow>
```

Think of conditional flows as “if-elseif-elseif—else” block of code.

```
<Flows>

<Flow name="GetRequests">
<Condition>request.verb="GET"</Condition>
    <Request>
        <Step>
            <Condition>request.path ~ "/statuses/**"</Condition>
            <Name>StatusesRequestPolicy</Name>
        </Step>
    </Request>

    <Response>
        <Step>
            <Condition>(response.status.code = 503) or (response.status.code =
                                                        400)</Condition>
            <Name>MaintenancePolicy</Name>
        </Step>
    </Response>
</Flow>

<Flow name=”HandleAllOtherCalls”>
    <Request>
        <Step>…</Step>
    </Request>
</Flow>

</Flows>
```

`<Flows>` is like your “if-elseif-elseif—else” block. Each block is a `<Flow>` with a `<Condition>`. If a condition is satisfied, depending upon whether the API call is in request or response processing, that conditional flow executes and other conditional flows are skipped.

As seen in the above example, you can of course use a conditional step within a conditional flow. While conditional flows are mutually exlcusive (if-elseif-elseif-else block), conditional steps within a flow are not. They are individual “if” blocks.

*Note – In the above example, if you moved the `<Flow name="HandleAllOtherCalls">`, which doesn’t have a condition to the top of the `<Flows>` block, it always gets executed. Edge doesn’t evaluate others. Of course, having two flows , both without conditions does not make sense. Edge always executes the first one as it matches.*

*Note – Using proxy.pathsuffix than request.path in our conditions is much better as request.path depends upon the deployment path which can change across environments.*

`<PreFlow>` and `<PostFlow>` don’t support conditions. They always execute. Of course, you can have conditional steps within a `<PreFlow>` or `<PostFlow>`.

*Critique – When thinking about building API processing logic, it is natural to I think about “This is what I have to do during request processing” and “This is what I have to do during response processing”. It is also natural, in building multiple APIs, to abstract out common logic into flows and reuse within request and response processing across APIs.*

*Having the top level tag as <Request> helps tremendously both in visualizing and in organizing code. Imagine*

```
<Request>
    <PreFlow name="ValidateKeys">
        <Step><Name>VerifyAPIKey</Name></Step>
        <Step><Name>BuildRequest</Name></Step>
    </PreFlow>

    <Flows>
        <Flow name="CallService">
            <Step><Name>ServiceCalloutGetMockResponse</Name></Step>
        </Flow>
    </Flows>

    <PostFlow>
    </PostFlow>

 </Request>
```

## Message Context

Policy steps may refer to data stored by Apigee about end users, applications, developers, API products etc. (at the organization and environment level). They may refer to runtime data stored either at the proxy or the environment level.

In addition, as flows execute, they need to

- read data, for eg, from a request or the response object
- create multiple outgoing request objects for various callouts and store those responses. You may need to extract data from the callout responses and later use those data to enhance the request object to be sent to the target.
- create flags or new variables to drive conditional flows

Where is all this data stored? What is its scope?

Edge creates a new context object for each API call and populates it with default variables as the message processing progresses. For example,

- it creates a client object that encapsulates the actual API request that the client made and a “request” object that encapsulates the request to be sent to the target, before kicking off the request flows.
- For example, it creates a target.response to represent the actual target response and response object that encapsulates the Proxy response to be eventually sent back to the client.

Policies typically create variables as they execute. They are available as part of the context object of the API call.

All the variables that you create are stored within this context object as well.

The message context object and hence the data is available till the API call finishes (it creates before the `ProxyEndpoint :: Request PreFlow` and the object is deleted after `ProxyEndpoint:: Post Response Flow`)

*Note – What this means is that, If there are 100 concurrent API calls to the same API, Edge processes those in parallel , each with its own context (variable set).*

**Mapping between client request and target request AND target response and Proxy response**

By default, the request object is an exact copy of the request sent by the client and the response object is a copy of the response received from the target.

The default behavior can be changed for a specific proxy/target using following transport properties.

```
request.retain.headers.enabled    <!-- true or false -->
request.retain.headers            <!-- comma separated list of headers to be -->
                                  <!-- retained. Valid if above is false -->


response.retain.headers.enabled   <!-- true or false -->
response.retain.headers           <!-- comma separated list of headers to be -->
                                  <!-- retained. Valid if above is false -->
```

`ProxyEndpoint` and `TargetEndpoint` support Properties within `HTTPProxyConnection`. Here's an example that turns off the "copying the headers".

```
<HTTPProxyConnection>

    <Properties>
        <Property name="request.retain.headers.enabled">true</Property>
    </Properties>

    <VirtualHost>default</VirtualHost>
    <VirtualHost>secure</VirtualHost>

</HTTPProxyConnection>
```

What is the difference in turning off at proxy versus target?

Let us say proxy p1 has routes to targets t1, t2, t3.

If you turned off on proxy p1, incoming headers (from the client) will not be retained.

If you turned off on target t2, headers will not be retained when sending to target t2, but target t1 and t3 will see all headers that client sent.

Let us take an example. Let us say client is sending header h1 with value v1.

If this is turned off on proxy, any step accessing the variable "`request.header.h1`" will get null. In order to get your flow work, you should access this header using variable, "`client.request.header.h1`"

If this is turned off on target, steps attached to request path can get the value using "request.header.h1", but steps attached on response path will get "request.header.h1" as null. As said before, still you can access this header using variable, "client.request.header.h1"

If my step explicitly sets a header value, does it reach the target when retain headers is turned off?

Edge doesn't distinguish between the headers received from client and which are explicitly set by your flow.

Let us say your step is attached on request path and it is setting a particular header in request. If the setting is turned off on proxy, then target will receive the header that your step set. If you turned off on target, then target will not receive the header that your step set, because just before sending to target, fountainhead will remove all headers.

## Streaming

By default streaming is turned off. It means the entire payload of http message is read and buffered in memory for processing.

```
<Property name="request.streaming.enabled">true</Property>
```

If the above transport property is added to proxy, then the http request's payload is not read into buffer. The request's payload is streamed as it is to the target. The request payload will not be available to any policy. But a policy has a chance to set its own payload, in such case the payload set will be sent to the target.

```
<Property name="response.streaming.enabled">true</Property>
```

If the above transport property is added to target, then the http response's payload is not read into buffer. The response's payload streamed as it is to the proxy. The response payload will not be available to any policy. But a policy has a chance to set its own payload, in such case the payload set will be sent to the proxy.

# 6. Variables

Runtime Container populates the context with a set of variables as it executes the flows. Policies create or set specific variables. We typically refers to these as framework variables.

Your flows and code, through policies like AssignMessage or JavaScript for example, can create and set variables. We refer to these as user defined variables.

## Scope

Unless set , a variable is “undefined”. Here’s how you check whether a particular variable, in this case a header value, is defined or not

```
request.header.myvalue == null        // true if undefined or nonexistent
                                      // false if defined or exists
```

Edge provides a literal called `null` to mean “undefined”.

*note: request.header.myvalue == “null” is a wrong check for ascertaining whether something is undefined or not. Why?*

Once set, all the variables (both framework or user defined) are available till the API call finishes (i.e. the response sent back to client).

## Naming Convention

All variables start with a letter and can be any combination of [a-z] [A-Z] [0-9] [_-].

All framework variables use a dotted convention and are all in lower case. E.g. `target.url` or `request.uri` or `target.sent.end.timestamp`.

## Types

All framework variables have a well defined data type and is one of boolean, int, long, float, double,  String, Collection, Map, Date, String[] or Message.

User defined variables have no definitive data type. If the variable was set by Jython or JavaScript, they may be of the type specified by those languages.

Since Edge does not define types for variables, it adapts values as needed for comparison. See the section on “Conditions” below.

## Literals

Edge provides the following literals

- `null` `// variable undefined or not set`
- `true` `// Boolean. True.`
- `false` `// Boolean. False.`

For numeric values, their type is interpreted as Integer unless the value is terminated by

- "f" or "F" which implies float (e.g: `3.142f, 91.1F`)
- "d" or "D" which implies double (`e.g: 3.142d, 100.123D`)
- "l" or "L" which implies long (`e.g: 12321421312L`)

## Framework Variables

For a full list of the variables, see this reference http://docs.apigee.com/api-services/reference/variables-reference

Here's a few of them to give you a quick flavor of the power of Edge.

| | |
|---|---|
| System | Information about the system like host name, date and time etc. |
| Configuration | Key Value Maps → stores runtime configuration.<br>Caches → can be not only used for response caching but to also store data beyond the context of the current API call.<br>Proxy → information about this proxy like proxy.name, proxy.basepath |
| Current API call | current.flow.name<br><br>request → wrapper for the HTTP target request message. You can access different parts of the current request message like request.header, request.querystring, etc<br><br>response → wrapper for the HTTP proxy response message. You can access different parts of the current response message like response.statuscode, response.reasonphrase, response.body etc.<br><br>error → Edge creates a default error response object whenever an API processing results in an error. Available within an error flow.<br>You can set / access different parts of the error response object like error.statuscode<br><br>message → It either refers to request or response object depending upon whether the API is in request flow or response flow. Useful if your code can be used across request or response flows.<br><br>target → Available after the route rules are evaluated.<br><br>Policy Properties →<br>Policy Variables → |
| Metrics | metrics about each message flow like start timestamp, end timestamp etc |
| Context | If you are writing a JavaScript policy, use context.getVariable to get and context.SetVariable to set flow variables. During setVariable, always check for the result. false implies that the set failed and the value is not set. |

## Creating and Referencing Variables

See section 10 "Variable Assignment"

# 7. Regular Expressions

One of the most common flow fragments you use in building API logic is

- to extract specific parts of a request or a response. While Edge provides variables like request.uri, you may be interested in more fine grained parts of the URI.
- to branch out to conditional flows based on the incoming API call (URI or query params for eg)

Edge provides a rich set of regular expressions to match and extract as well as for use in conditional expressions.

## Regular Expressions for String Matching

| | |
|---|---|
| Foo*Bar | Matches any of these fooBar or fooaaBar or fooabcBar. Default is Case-sensitive matching. Yiou can set flags to ignore case |
| Foo*Bar with Ignore wildcard flag | Matches Foo*Bar |
| Foo*Bar%*World | % is used to escape the following wildcard. This for eg matches FooABCBar*World |
| Foo*Bar%%World | %% is used to escape the % itself. So it matches FooBar%World |
| Foo{var}Bar | {var} refers to the variable called var. for example, if var = abc, matches FooabcBar |

## Regular Expressions for Path Matching

| | |
|---|---|
| Foo/*/Bar | * matches a sequence of characters upto /. So this expression matches Foo/world/Bar but not Foo/Bar nor Foo/hello/world/Bar |
| Foo/**/Bar | ** matches a sequence of characters beyond /. So this expression. Matches Foo/hello/world/Bar but does not match Foo/Bar |
| Foo/*/Bar Prefix pattern flag | prefix pattern implies that there can be any number of characters beyond the matching pattern. So for eg, it matches Foo/123/Bar/abc/xyz |
| Foo/%*/Bar | Matches Foo/*/Bar literally |
| Foo/{var}/Bar | Matches Foo/World/Bar if var = World |

# 8. Conditional Expressions

You can set conditions under which you want Edge to execute a step or a flow.

## Operators

| | |
|---|---|
| ! | unary operator |
| = or == | equals |
| := | equals but case insensitive |
| != | not equals |
| >, <, >=, <= | greater than, less than, greater than or equal to , less than or equal to |
| && | condition1 AND condition2 |
| \|\| | condition1 OR condition2 |
| ~ | Matches like expression, case insensitive |
| :~ | Matches like expression, case insensitive |
| ~/ | Matches a path expression |
| ~~ | Matches a java regular expression |
| =\| | Starts with |

Most of the above operators also have aliases. EG you can use the world Equals instead of =. Or MatchesPath instead of ~/.

The precendence of operators is similar to the precedence defined in Java language.

*Note - In order to have the operator characters included into the variable name like request.header.help!me it needs to be enclosed in single quotes like 'request.header.help!me'*

## Type Conversion

Since Edge does not define types for variables, it becomes necessary to adapt values as needed for comparison.

For example, response.status.code is an integer.
`<Condition> (response.status.code == 400) </Condition>` does not require any adaptation.

Whereas
`<Condition> (response.status.code == “400”) </Condition>` requires the integer to be converted to string before the comparison can happen.

*Question - The result is the same in both cases but any guesses as to which would be faster at runtime?*

Here is how Edge adapts the values before comparison. If any of the LHS (Left Hand Side of the operator) or RHS (Rigth Hand Side of the operator) is a string, the entire comparision becomes a string comparison. Otherwise, the lower type gets converted to higher one in the following order order – boolean, Integer, Long, Float, Double.

Here are a few examples

```
<Condition>(request.header.content-type = "text/xml")</Condition>
<Condition>(request.header.content-length < 4096 && request.verb = "PUT")</Condition>
<Condition>(response.status.code = 404 || response.status.code = 500)</Condition>
<Condition>(request.uri MatchesPath "/*/statuses/**")</Condition>
<Condition>(request.queryparam.q0 NotEquals 10)</Condition>
```

# 9. Policies

Edge provides a set of predefined functions to speed up your API development. These are called Policy Templates. To use them within a flow, you have to create an instance of a 'Policy Template' and use that Policy instance within your flows.

In addition to predefined functionality, Edge also provides "Extension" policies for you to write your custom API logic in either JavaScript, java, python or node.js.

Here are the categories of policies that Edge provies

- Traffic Management. For example
  - <Quota> - Enforce a limit on how many API calls a particular developer app can make, say in an hour
  - <PopulateCache> - Store the response in cache so that subsequent calls be processed quickly

- Security. For example
  - <VerifyAPIKey> - Check whether the developer app calling the API is allowed to call that API
  - <XMLThreatProtection>, <JSONThreatProtection> etc
  - <OauthV2>

- Mediation. For example
  - <JSONToXML> and <XSLTransform> - To easily transform a JSON message to XML format
  - <ExtractVariables> and <AssignMessage> - To extract from incoming messages and to set the appropriate fields in the outgoing messages
  - <KeyValueMapOperations> - To store values beyond the lifetime of the current API call.

- Extensions. For example
  - <JavaScript> - to write your custom code in JavaScript.
  - <StatisticsCollector> - to collect custom fields / values from messages and store them within Apigee Edge Analytics server.

*note – The above is not the exhaustive list of policies. For a complete list of policies in Edge, refer to [http://docs.apigee.com/api-services/reference/reference-overview-policy](http://docs.apigee.com/api-services/reference/reference-overview-policy)*

## Invoking a Policy

By including a instance of a policy template within a step in a flow, Edge executes that policy as part of executing that flow.

How do you specifiy what you want the policy to do? For example

- How do you tell the `<Quota>` policy the specific quota limit to check against?
- `<AssignMessage>` can assign headers, payload and can even create new user defined variables (also called as flow variables) to hold some value. How do you tell what you want the policy to do?
- `<ServiceCallout>` needs to store the response in an object that you can access it elsewhere in the flow. How do you pass pass this object?

You specifiy those by creating and configuring an instance of a policy template within a proxy..

Lets look at this simple example. This example uses VerifyAPIKey policy to check for the key during the request flow and assigns custom headers during the response flow using the `AssignMessage` policy.

Here's an instance of the `AssignMessage` policy.

```
<AssignMessage async="false" continueOnError="false" enabled="true"
name="Assign_Custom_Headers">
    <DisplayName>Assign Custom Headers</DisplayName>
    <Properties/>
    <Set>
        <Headers>      <!-- Specify the headers to set -->
            <Header name="x-firstName">{firstName}</Header>
            <Header name="x-lastName">{lastName}</Header>
        </Headers>
    </Set>
    <IgnoreUnresolvedVariables>true</IgnoreUnresolvedVariables>

    <!-- set the headers on the existing response object -->
    <AssignTo type="response" transport="http" createNew="false"/>
</AssignMessage>
```

Here's an instance of the `VerifyAPIKey` policy.

```
<VerifyAPIKey async="false" continueOnError="false" enabled="true"
name="VerifyAPIKey">
    <DisplayName>VerifyAPIKey</DisplayName>
    <FaultRules/>
    <Properties/>

    <!-- Specify the Key to check -->
    <APIKey ref="request.queryparam.apikey"/>
</VerifyAPIKey>
```

Here is how it is included in a flow.

```
<ProxyEndpoint name="Endpoint_For_BasePath_ALL">
    <Description/>
    <FaultRules/>
    <PreFlow name="PreFlow">
        <Request>
            <Step>
                <!—Specify the policy to invoke. You refer to a policy instance by its
                name -->
                <Name>Verify_APIKey</Name>
            </Step>
        </Request>
        <Response>
            <Step>
                <!—Specify the policy to invoke. You refer to a policy instance by
                its name -->
                <Name>Assign_Custom_Headers</Name>
            </Step>
        </Response>
    </PreFlow>
</ProxyEndpoint>
```

## Structure of a Policy

All policies have a pretty standard structure (XML schema). Lets look at those common elements.

*Note: The below <AssignMessage> is not the complete policy template but used to highlight the common elements across all policies.*

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<AssignMessage async="false" continueOnError="true" enabled="true" name="set_headers">

    <DisplayName>"Set Headers"</DisplayName>
    <Properties/>

    <IgnoreUnresolvedVariables>true</IgnoreUnresolvedVariables>

</AssignMessage>
```

### Naming Convention

| | |
|---|---|
| Tags | Start with uppercase letter. For a multiword tag, each word starts with an uppercase letter (e.g. `AssignMessage`) |
| Attributes | Start with lowercase letter. For a multiword tag, each word, subsequent words start with an uppercase letter (e.g. `continueOnError`) |

### Common Tags and Attributes

| | |
|---|---|
| `async` | This flag is currently not supported. |
| `continueOnError` | If this policy results in an error, do you want Edge to continue the flow or stop and jump to error flow?<br>*Question - What could some of those errors be that you want to continue?* |
| `enabled` | Edge executes the policy only if this is set to true. Why would you want this optionality? Well, if you want to have extra logging during debugging, you can have those policies turned on but off otherwise. During debugging, if you want to turn a buggy one off and insert another new one, you can. How? By using the Edge Management APIs (http://docs.apigee.com/management/apis) |
| `name` | Uniquely identifies the instance of this policy. Its very similar to creating an object (instance of a class). You could use it in any flows but they all refer to the same instance.<br><br>Please please please have a naming convention for your API project! Don't leave it as "default". Keep it separate from `DisplayName`. |
| `DisplayName` | Edge UI uses this to show the policies. Not used by the runtime. |
| `Properties` | Very few policies use this tag. Mostly the Java / JavaScript policies. It is a mechanism to pass a set of name-value pairs to the policy execution. Same thing like a config file. |

`IgnoreUnresolvedVariables`

Many policies end up referencing variables OR extracting some data from the request or response. How do you want Edge to deal with undefined variables (those that are not yet set)? What if Edge can't find the data you are looking for in the request or response?

Setting this to "false" results in a "fault" and execution jumps to error flow.

If you set `IgnoreUnresolvedVariables to true` and if the variable is not resolved , the flow continues. A value of "" is used whereever this variable is used within the text substitutions.

## Common XML Patterns (or styles ) Used Across Policies

clearPayload

```
<ExtractPolicy>
    <Source clearPayload="true|false">response_from_SystemA</Source>
</ExtractPolicy>
```

`clearPayload=true` → then payload of source is cleared after information is extracted. Useful , for eg, if you have already extracted values from a ServiceCallout response , then do you need to keep that response object till the flow finishes? Helps reduce the runtime memory footprint!

Specifying the source

```
<ServiceCallout>
    <Request clearPayload="false" variable="myrequest">
    <Response>response_from_systemA</Response>
<ServiceCallout>
```

`variable="myrequest"` → Tells the ServiceCallout policy to not use the default request object but instead to use the object `myrequest`.

*Critique - In the above 2 examples, the input variable (request or response object) is sometimes an attribute and sometimes a tag value. Sometimes it is called "source" and some times it is called "variable"*

Creating a new request object

```
<AssignMessage>
    <AssignTo type="request" transport="http" createNew="true">myrequest</AssignTo>
</AssignMessage>
```

In the above example, you are explicitly stating that this policy should create a new request named myrequest and should work on that new request. If you are modifying a to send to the target, set createNew to false `<AssignTo type="request" transport="http" createNew="false"/>`

Referencing Variables

```
VerifyAPIKey      <APIKey ref="request.queryparam.apikey"/>

ResponseCache     <CacheKey><KeyFragment ref="request.uri" type="string"/></CacheKey>
PopulateCache     <TimeoutInSec ref="expirytime_for_cache">7200</TimeoutInSec>

Quota             <Allow count = "2000" countRef = "request.header.allowed_quota" />
Quota            <Interval
ref="verifyapikey.VerifyKey.apiproduct.developer.quota.interval">1</Interval>
```

"ref" implies that you want to set the value of the variable on LHS to the value found on the right hand side variable. Last two examples set a default value in case the ref doesn't exist.

| | |
|---|---|
| `ExtractVariable` | |
| `{my_variable}` | Used within the regular expressions (string and path expressions) to hold the equivalent value if the overall expression matches. Here the enclosing tag like <URIPath> provides the source to match against. |
| `<Variable name = "my_variable">` | Used to determine the source of the match. |
| `<QueryParam name = "key">` | Use the Query Param "key" as the source |

| | |
|---|---|
| `AssignMessage` | |
| `<Name>my_variable</Name>` | Typically as a destination to set |
| `<Ref>my_variable</Ref>` | Use the value from my_variable to set. |
| `{my_variable}` | Substitute with the value. |

*Critique – Some policies use the generic name "Ref" or "ref". Some others use a specific name (e.g. countRef). In some other cases, we use {}. Sometimes it is a tag value and sometimes it is an attribute (e.g. In PopulateCache, you see the default value within the "value" of the tag but in Quota you see it added as an attribute)*

## Escaping special characters

Normal way to reference a variable is {…} but if you are assigning JSON, JSON itself treats { as special. Here's a way to escape { and } characters

```
<AssignMessage>
    <Payload contentType="application/json" variablePrefix="@" variableSuffix="#">
        {"name":"foo", "type":"@variable_name#"}
    </Payload>
</AssignMessage>
```

## Conditional Assignment

If you want to assign a value based on the value of some other element, here is an example of it

```
<Quota>
    <Allow>
        <Class ref = "request.header.clientid">
            <Allow class="X" count=100"/>
            <Allow class="Y" count="75" />
            <Allow class="_default" count="50" />
        </Class>
    </Allow>
</Quota>
```

The above is a way to specify the Quota count based on the value of a variable `request.header.clientid`. If request.header.clientid is "X" , the value of the count is set to 100. If request.header.clientid is "Y", then count is set to 75. Else it is set to 50.

*Critique – Elsewhere, the tag* <Condition> *is used for conditional execution. Maybe using the same style would have been better. For eg*
*<Conditon>(request.header.clientid = "X")</Condition>*
*<Set>….</Set>*

## Policy State Management

**Most Policies are stateless**
As specified above, you create instances of policies and use them in flows. Most policies are stateless. i.e. They don't have their own state that is carried forward to subsequent invocations. They typically work on the inputs you provide when you created the policy instance (e.g. Cache Key to search for) and read/write variables from the message context.

Almost every policy creates a set of variables when it executes. These variables are stored in the message context. The variables all follow a standard naming convention – `[Name of the Policy Template in all small letters].[Name of that policy instance you created].[variables in dotted notion].`

As an example, an instance of <ConcurrentRatelimit> policy with a name "Throttle_Catalog_Target" creates the following variables.

- `concurrentratelimit.Throttle_Catalog_Target.allowed.count`
- `concurrentratelimit.Throttle_Catalog_Target.used.count`
- `concurrentratelimit.Throttle_Catalog_Target.available.count`
- `concurrentratelimit.Throttle_Catalog_Target.identifier`

*note – You can refer to any of these variables in your API logic (policies or custom code) just like any other variable. Refer to the section on Variables.*

**Some policies have to maintain state across API calls**
A few policies, though, have to maintain state across API calls. For example, a <Quota> policy included in a Proxy has to keep track of the number of API calls processed by that Proxy to be able to enforce Quota.

Depending upon where they store their state, such policies come in two flavors. For example, <Quota> maintains its state inmemory (local to proxy within a MP instance) whereas <DistributedQuota> maintains its state in a persistent storage (accessible across proxies, across MPs, across regions in a DN depoyment)

Given that a Proxy is THE API RUNTIME UNIT, a policy instance is always scoped to a Proxy. i.e. A Quota policy instance within Proxy A is different from that of a Quota Policy instance within Proxy B even though you may name them the same.

*Question – So then, if you included the same Quota Policy instance within multiple flows within a particular Proxy , would they all refer to the same counter? For example, you create a Quota policy named MyQuotaPolicy with a limit of 5 requests and place it on multiple flows (Flow A, B, and C) in the API proxy. Even though it is used in multiple flows within the API Proxy, it maintains a single counter that is updated everytime that policy instance is invoked.*

*Note – Caching is another capability that comes in both "Local-To-Proxy" mode as well as "Common-Across-Proxies" mode (i.e. Distributed)*

## Local-To-Proxy vs Distributed: Setting tresholds

From a performance standpoint, using, for eg a <Quota> policy is faster than using a <DistributedQuota> or using <ConcurrentRatelimit> with <Distributed> = flase is faster than if <Distrubuted> set to true. Maintaining counters inmemory is faster as compared to going over network to read / write persistent counters.

If it is critical for your API to enforce EXACT limits, then choose the distributed versions of these policies. If not, then "local-to-proxy" would be a much better. But then, in that case, you have to know how many Proxy instances are deployed (a.k.a. how many MPs) to be able to compute per-proxy counter.

*Critique – It would have been wonderful if Edge auto computed the per-proxy limit as its runtime inherently has the knowledge of the number of MPs than for the developer to specify this value. Plus, as the runtime configuration changes (e.g. more capacity added), this limit would be auto adjusted as opposed to today requiring a change to the Proxy.*

## Some Policies Need to be Attached to Multiple Flows

**Caching**
Let us say that you are using "caching" to speed up your API responses and to reduce burden to your backend systems. What this means is that your code has to

- store the responses (with a key) when it reads a response from target.
- For every incoming API call, check the cache and if it contains matching response read from cache and send it back to the client. If not, send it to target.

How are you going to model the above logic within Edge? By

- Including <ResponseCache> policy in your request flow, ideally in ProxyEndpoint :: Request Preflow
- Including <ResponseCache> policy in your response flow, ideally in TargetEndpoint :: Response Preflow

**ConcurrentRatelimit**
Let us say that you want to limit the number of API calls that can be sent to a particular target. What this means is your code has to

- Check whether to send the new API call to the target or not.
- If ok to send, send the request.
- Based on the response code (success or failure), the time the backend took, and whether you want to strictly enforce the limit or adapt if the backend is able to handle, decrement the counter.

How are you going to model the above logic within Edge? By

- Including <ConcurrentRateLimit> policy in the TargetEndpoint :: Request Preflow
- Including it also in the TargetEndpoint :: Response PostFlow and DefaultFaultRule.

*Question – Why include in DefaultFaultRule? Certain target response codes automatically result in Edge jumping to error flow. If the target is overloaded, it would throw certain error codes back which means that subsequent requests should not be made.*

# 10. Variable Assignment

As part of processing, one of the most common capabilities you need is to be able to extract data from messages, set request data and create / modify data variables along the way.

## ExtractVariables

ExtractVariables policy is typically used to extract various parts of a message (request or a response). Refer to the policy documentation http://docs.apigee.com/api-services/reference/extract-variables-policy for the complete policy description.

Given that Edge primarily deals with HTTP, JSON and XML it provides a powerful set of extraction primitives to easily read and assign values.

**Examples**

**To extract a portion of the URI,** you have to construct a path expression that matches what you are looking for. Note that the variable name is part of the overall expression that gets populated if the expression matches.

```
<ExtractVariables name="...">          <!-- ExtractVariables policy. We are only
                                       looking at extraction code snippets and not the
                                       whole policy definition here -->


                                       <!-- extracting the actual itemid from within
                                       the URI and assigning to variable called
                                       item_searched -->
   <URIPath>
      <Pattern ignoreCase="true">/catalogue/{item_searched}</Pattern>
   </URIPath>

</ExtractVariables>
```

### Pattern should match full URI

```
<Pattern ignoreCase="true">/a/**/c/{path_seg_last}</Pattern>
```

This one matches /a/b/z/c/value and sets pathSegLast to “value”. Note that this does not match /a/b/z/c/value/xyz. Your pattern has to absolutely match the full URI.

### You can extract multiple variables at the same time

```
<Pattern ignoreCase="true">/a/{path_seg_b}/c/{path_seg_last}</Pattern>
```

### Extract a part of a value from a query param

```
<QueryParam name="apikey">
    <Pattern>*QVj{akey}</Pattern>          <!-- from apikey query parameter, extract
                                           part of the value and store in aKey -->
</QueryParam>


<QueryParam name="greeting">
    <Pattern>hello {user}</Pattern>
    <Pattern>hi {user}</Pattern>
</QueryParam>
```

The above example sets the variable “user” to the string following hello or hi. Here are the rules that govern the extraction.

- If source message resolves to a message type of response, then this does nothing.
- The specified patterns are matched in the order given, and the first pattern matched is used for extraction.
- Each pattern is matched against each value of the query param “greeting”
- If “greeting” queryparam has two values “value1” and “hi value2”, then second pattern matches the second value of the queryparam “greeting”
- if <VariablePrefix> is not specified, then variable “user” is assgned “value2”
- if <VariablePrefix> is defined as “extracts”, then “extracts.user” is assigned “value2”

*note - To extract only from second value of greeting, specify QueryParam@name as “greeting.2”. These rules apply to all extraction patterns.*

### Extract value(s) from a variable

```
<Variable name=”request.content”>
    <Pattern>hello {to_whom_all}</Pattern>
</Variable>

<Variable name="request.content">
    <Pattern>you have {amount} due on {date}</Pattern>
</Variable>
```

### Examples of string patterns

| Pattern | Input | values extracted |
|---|---|---|
| {host}:{port} | apigee:1234<br>apigee | host=apigee port=1234<br>no value extracted |
| *;charset={encoding} | text/xml;charset=UTF-16<br>application/xml;charset=ASCII<br>application/soap+xml | encoding=UTF-16<br>encoding=ASCII<br>no value extracted |

### some example of path patterns

| Pattern | URIPath | values extracted |
|---|---|---|
| /*/a/{v1} | /x/y/z<br>/x/a/b<br>/x/a/b/c/d<br>/x/a/b;jsessionid=123456 | no value extracted<br>v1=b<br>v1=b<br>v1=b |
| /a/**/feed/{v1}/{v2} | /a/b/feed/rss/1234<br>/a/b/c/d/feed/rss/5678<br>/a/b/c/feed/rss/5678/d/feed/atom/9876 | v1=rss v2=1234<br>v1=rss v2=5678<br>v1=rss v2=9876 |

### Extraction from JSON Payload

```
<JSONPayload>
    <Variable name="first_name">
        <JSONPath>$.firstName</JSONPath>     <!-- you are extracting firstName from
                                             the and setting the variable first_name
                                             -->
    </Variable>
    <Variable name="last_name">
        <JSONPath>$.lastName</JSONPath>
    </Variable>
 <JSONPayload>
```

### Multiple JSONPaths within a Variable Tag

```
<Variable name="v1">
    <JSONPath>$.store.book[*].author</JSONPath>
    <JSONPath>$.store.book[?(@.category = 'reference')]</JSONPath>
```

```
</Variable>
```

Here the jsonpaths specified are evaluated in specified order, and the value of first jsonpath expression that is hit is assigned to the variable. If none of jsonpaths is hit, then value of the variable is not modified.

### Extraction from XML Payload

You can use XPath to extract and assign to variables.

```
<Variable name="v1">
    <XPath>/root/child[5]/@name</XPath>
    <XPath>/company/employee[5]/@name</XPath>
</Variable>
```

Variables can be used within XPath expressions

```
<Variable name="experiencedEmployeed" type="nodeset">
    <XPath>/company/employee[@age>=$request.header.age]</XPath>
</Variable>
```

The above example is used to extract details of employees whose age is greater than or equal to the value specified in request header named age.

*Note – Variable@type specifies the datatype, the xpath expressions to evaluated to. if Variable@type is not specified it is treated as string.*

For a deeper understanding of all the extraction capabilities that Edge provides with `ExtractVariables` policy, refer to the policy documentation.

## AssignMessage

AssignMessage policy is typically used to set variables, enhance the request or response and to create new request objects for callouts. Refer to the policy documentation http://docs.apigee.com/api-services/reference/assign-message-policy for complete policy description.

**Examples**

**Set timeout to 10**

```
<AssignMessage name="...">

    <AssignVariable>
        <Name>timeout</Name>                <!-- you are setting timeout to 10 -->
        <Value>10</Value>
    </AssignVariable>

 </AssignMessage>
```

### Assign with a default value

```
<AssignVariable>

    <Name>timeout</Name>
    <Value>10</Value>
    <Ref>t_value</Ref>                <!-- Use the variable t_value. If undefined
                                      set it to 10 -->
</AssignVariable>
```

*note – Can you assign a JSON data to a variable? Well, you can assign a string (which happens to be of the JSON format). Edge treats it like a string than a JSON object.*

*Question – Is there a way to use ExtractVariables policy to work on such a string and read back individual elements?*

### Assign an existing variable to another

```
<AssignVariable>
    <Name>queryparam.A</Name>
    <Ref>request.queryparam.A</Ref>
</AssignVariable>


<AssignVariable>

    <Name>queryparam.A</Name>
    <Ref>request.queryparam.A.1</Ref>        <!--  .A.values will give you collection.
                                             A is same as A.1.
                                             Index in collection starts with 1
                                             .values.count gives you num elements
                                             -->
</AssignVariable>
```

### Assign response headers

```
<Set>
    <Headers>
        <Header name="x-firstName">{callout_response_firstName}</Header>
        <Header name="x-lastName">{callout_response_lastName}</Header>
    </Headers>
</Set>
```

### Copy one message object to another

```
<AssignMessage async="false" continueOnError="false" enabled="true"
name="CopyFromRequest">

    <AssignTo createNew="true" transport="http" type="request" name="newRequest"/>
    <Copy source="request">   <!-- .create a new req “newRequest” and copy
                              whatever is needed from request object  -->
        ….
        ….
    </Copy>

</AssignMessage>
```

# 11. Service Callout

A common requirement within an API Proxy is to be able to call multiple backend services and return a mashed up response to the client. Edge allows you to model this two ways

- Define one of them as a target. Use <ServiceCallout> policy or any custom policies to talk to others. Use <ExtractVariables> and <AssignMessage> to build the final response.

  *note – Of course, you can also build the final response using custom code in JavaScript or Java or python but why not take advantage of what Edge has to offer?*

- Do not define any target. Use <ServiceCallout> or any custom policies to talk to backend services and construct the appropriate response. Use <ExtractVariables> and <AssignMessage> to build the final response.

*Question – What are the advantages of modeling it one way or other?*

## ServiceCallout Policy

A ServiceCallout policy is used to call an external service and read / store the response in a variable for further processing. Given that Edge executes flows and policies within flows synchronously and in sequence, this implies that until the response is received and stored, the next policy will not be executed.

Here's an instance of a ServiceCallout policy.

```
<ServiceCallout async="false" continueOnError="false" enabled="true"
name="get_response_from_mockserver">

    <DisplayName>Get Mock Response</DisplayName>

    <!-- myrequest object should have been built earlier, say using AssignMessage.
    Alternatively, from within this policy, we can build the request to be sent -->
    <Request clearPayload="false" variable="myrequest">
        <IgnoreUnresolvedVariables>false</IgnoreUnresolvedVariables>
    </Request>

    <!-- This is where Edge will store the resopnse -->
    <Response>mockresponse</Response>

    <HTTPTargetConnection>
        <Properties/>
        <URL>http://mocktarget.apigee.net</URL>
    </HTTPTargetConnection>
    <Timeout>30000</Timeout>

</ServiceCallout>
```

Here is how it is included in a flow

```
<ProxyEndpoint name="CatalogAPI">
    <Description/>

    <PreFlow name="PreFlow">
        <Request>
            <Step><Name>VerifyAPIKey</Name></Step>
            <Step><Name>build_myrequest</Name></Step>
            <Step><Name>get_response_from_mockserver</Name></Step>
           <!-- call other services just like above -->
        </Request>
        <Response/>
    </PreFlow>

</ProxyEndpoint>
```

Note – When execution hits an I/O point, e.g. ServiceCallout policy, the flow gets halted and the container executes the network I/O asynchronously. Only when it receives the compete response does it resume execution of the flow.

*Note - If you used a <ServiceCallout> policy to call out to a service and do not specify a response object to store the response, you are hinting Edge that you don't care to read the response. In this scenario, Edge does not even wait to read the response. It is akin to a "fire and forget" call.*

*Question – What if the service you are calling is too slow? It eventually exhausts the thread pool within the container processing I/O at which point this particular container instance won't be able to process further incoming API calls. Left unchecked, eventually all the containers processing that API call will become unresponsive. What design practices should you employ to guard against that back pressure?*

## Custom code

Alternatively, you could use, for eg, JavaScript to call out to multiple services, wait for responses and store them for extraction by other policies.

```
// call multiple services. The below call within JavaScript is async.
var mockresponse = httpClient.get(' http://mocktarget.apigee.net ');

// wait for responses to arrive as the above call is async. If you don't wait
// the execution continues, this policy may finish and control moves to next policy
mockresponse.waitForComplete(1000); //timeout in milliseconds

// extract values and store them in context
```

*Question – Can you store the entire response object into the context? See section on Scripts.*

# 12. Scripts

Script policies allow you to write custom code using Java, JavaScript or Python policies.

## JavaScript

The runtime container compiles your JavaScript code into native Java and runs. Please note that the runtime container loads the compiled JavaScript in a sandbox and has an upper limit on how long it can run to ensure that a misbehaved or erroneous JavaScript cannot bring down the container. JavaScript executes in a different context from that of the regular policies.

```
<JavaScript timelimit="200" enabled=true name="myStep">

    <Properties/>

    <ResourceURL>jsc://script</ResourceURL>
    <IncludeURL>jsc://include</IncludeURL>   <!-- Zero or more IncludeURLs allowed -->

</JavaScript>
```

request, response and context objects are all available within the sandbox for your script to use. Here are some example code snippets.

- Set content — `response.content='Hello'`
- Get the body of the request — `c = request.body`
- Get a header in the request — `h = request.headers['via']`
- Set a header in the response — `response.headers['X-Api'] = 'Foo'`
- Get two query params in request —
```
first = request.queryParams[v] OR
First = request.queryParams[v][0]
Second = request.queryParams[v][1]
```

*Note – Index starts from 0 within JavaScript. Whereas within XML policies, they start from 1.*

**Headers, Query Parameters and Form Variables**
all have a common format, name-value pair. The name will always be a string. The value may be a single string or a list of strings in case the value is a list of members. The list can be accessed by the array operator.

For example, If the following headers are set in a HTTP response

```
Content-Type: application/json
Via: 1.0 fred
Via: 1.1 example.com
```

Here's a way to access

```
response.headers['Content-Type']      // "application/json"
response.headers['Content-Type][0]    // "application/json"
response.headers['Content-Type][1]    // undefined

response.headers['Via']               // "1.0 fred"
response.headers['Via'][1]            // "1.1 example.com"
```

Some more examples. Let us say that the API call contains the following query paramaters

```
A=hello&A=world&B=true&C=false
```

Then

```
request.queryParams['A']              // returns "hello"
```

```
request.queryParams['A'].length()     // returns 2. Then you iterate over to
                                      // get the full list of values
```

If you want to get the list of values in one go, then here's the code

```
context.getVariable("request.queryparam.A.values") ;        // returns a list
                                                            // of two values
```

*Critique – Notice that the standard variables are accessed with different names within JavaScript policy and other policies (e.g. `request.queryParams` in JavaScript versus `request.queryparam` in `AssignMessage` or `ExtractMessage`)*

*note – Within JavaScript, the index starts with 0 versus 1 within other policies. For example*

```
context.getVariable("request.queryparam.A.1")          // to get "hello", first value
request.queryParams['A'][0]                            // to get "hello", first value
```

**Content**

`response.content` has two members – `asXML` and `asJSON`.

For example, if content is

```
"<customer number='1'><name>Fred<name/><customer/>"
```

here is how you can extract

```
var number = response.content.asXML.@number;
 var name = response.content.asXML.name;
```

For example, if content is

```
"{ "a":1 , "b":2 }"
```

here is how you can extract

```
var a = response.content.asJSON.a;    // == 1
var b = response.content.asJSON.b;    // == 2
```

**Message Context**

The entire context of the API being processed (e.g. flow variables, user defined variables etc) are avaiable within the `context` object.

| | |
|---|---|
| To set variables for other policies to access | `context.setVariable()` |
| To get variables previously set | `context.getVariable()` |
| To delete a variable | `context.removeVariable()` |

**Undefined**

```
if (context.getVariable('A') === undefined)        // true if it is not set
```

**Session**

context.session is a map of objects that your JavaScript steps can use to pass data across steps. For example,

In one JavaScript step, you can initiate a http call

```
var calloutResponse = httpClient.get('http://httpbin.org/get');
context.session['calloutResponse'] = calloutResponse;
```

and in another JavaScript step, you can process the response

```
var exchange = context.session['calloutResponse'];
exchange.waitForComplete(1000); //timeout in milliseconds
var responsePayload = exchange.getResponse().content;
```

*note – The same can be accomplished by storing calloutResponse within context. For example*

```
context.setVariable("calloutResponse", calloutResponse) ;
```

and can retrieve it in another JavaScript step as

```
var exchange = context.getVariable("calloutResponse") ;
```

*Critique – Why do we have a session map when we can already store it in the context?*

*Question – Can you store the complete response object into context and use it from within an ExtractVariable policy? Why and Why not? Go ahead and test it out.*

## Node.js

Apigee Edge allows you to build and run services within the API Platform using node.js. Apigee Edge supports node.js scripts as a "target" within a proxy application. In this model, the "target" is not an HTTP URL, but instead is a script. . Instead of routing an incoming API call to a target, you can route it to this service.

Here's an example service hello-world.js

```
var http = require('http');

console.log('node.js application starting...');

var myServer = http.createServer(function(req, resp) {
        resp.end('Hello, World!');
});

myServer.listen(9000, function() {
        console.log('Node HTTP server is listening');
});
```

Here's how you specify that within a TargetEndpoint

```
<TargetEndpoint name="CatalogTarget">

    <Description/>

    <FaultRules/>

    <PreFlow name="PreFlow">
        <Request/>
        <Response/>
    </PreFlow>

    <PostFlow name="PostFlow">
        <Request/>
        <Response/>
    </PostFlow>

    <Flows/>

    <ScriptTarget>
        <ResourceURL>node://hello-world.js</ResourceURL>
    </ScriptTarget>

</TargetEndpoint>
```

This service is treated like a target except that its lifecycle is closely tied to the API Proxy lifecycle. EG When you deploy an API Proxy, the node.js server gets started. When you undeploy, the node.js server stops.

From within the node.js server, you can access all the API context variables.

*Note – Instead of writing your API logic within a Proxy, it is tempting to write it in node.js server. Please note that, in this case, the incoming API call from the client is going through two hops (Proxy and then the node.js server) before it hits the real target.*

*Question – If multiple API Proxies want to talk to the same service (implemented as node.js service within Edge), how are you going to model it? Would including the same <TargetEndpoint> definition in both API Proxies work?*

*Alternatively build one API Proxy (let us call it as Proxy A) that includes the TargetEndpoint that includes the node.js service and rest of the API Proxies that need to talk to that node.js service should refer to "Proxy A" as their target.*

# 13. Fault Handling

If an error occurs while executing the proxy code, Edge exits the current flow and enters the error flow - creates a default `error` response object that captures the "Fault" (to be sent back to the client) and executes your "catch" block.

If you don't have a "catch" block, Edge automatically sends the error response back to the client. The specific error response is normally set by the code that raised the fault.

*Note - Within the error flow, the flow variable `message` is aliased to `error`. `request` is available. If the error occurred in the response processing, `response` is also available.*

## Fault Categories

What kinds of faults should you expect?

Coding Errors

Classification Failures – Edge couldn't find a matching Proxy to execute (you can't catch these as Edge didn't find a proxy in the first place)

Routing Failures – Edge couldn't match or find a target

You have set `IgnoreUnresolvedVariables=false and continueonError=false` and a variable you referenced hasn't been set yet

Target Error Responses

By default, Edge raises a Fault if it receives 3xx, 4xx, 5xx responses from the target. You can define or change this behavior from within your TargetEndpoint definition by defining what your success codes are.

Policy Errors

eg – `<VerifyAPIKey>` policy raises a fault because the key didn't match.

http://docs.apigee.com/api-services/content/error-code-reference for a complete list of policy errors.

Custom Errors

Faults raised explicitly by your code. You can raise a fault using `<RaiseFault>` policy

System Errors

Transport errors
Out of memory

## Catching Faults

Essentially, handling a fault typically involves setting an appropriate error message for the client (typically one or more of `<AssignMessage>` policies), logging for later debug purposes and maybe for complex scenarios, doing some "backtrack" stuff (EG your API executed a payment request but failed on the response path. What's your backtrack logic?).

You should have a pretty robust "catch" block to handle errors appropriately. `<FaultRules>` is your "catch" block. If "Fault" occurred in the `ProxyEndpoint` flows, then `<FaultRules>` within `ProxyEndpoint` get executed. If the "Fault" occurred in the `TargetEndpoint` flows, then the `<FaultRules>` within `TargetEndpoint` get executed.

Note: There is no option to resume the original flow from the point of failure after handling the Fault.

```
<FaultRules>
    <FaultRule name="name your rule">
    <Condition>(execute the rule if this matches)</Condition>
            <Step><Name>Your policy to execute</Name></Step>
            <Step><Name>Your policy to execute</Name></Step>
    </FaultRule>

    <FaultRule name="name your rule">
    <Condition>(execute the rule if this matches)</Condition>
        <Step>
        <Condition>(execute the step if this matches)</Condition>
            <Name>Your policy to execute</Name>
        </Step>
    </FaultRule>

</FaultRules>
```

`<FaultRules>` is pretty much like an "if-else if-else if-else" block. Once a particular `<FaultRule>` is matched and executed, others are skipped.

*Critique – For unexplainable reasons (mostly an implementation side effect!) Edge seems to evaluate the Fault conditions bottom to top (in the way they defined within the `<FaultRules>` block) within the `ProxyEndpoint` but top to bottom within the `TargetEndpoint`.*
https://community.apigee.com/articles/23724/an-error-handling-pattern-for-apigee-proxies.html

A `<FaultRule>` without a condition will always match, so ensure you place at the end of the FaultRules block within a TargetEndpoint and at the beginning of the FaultRules block within an ProxyEndpoint.

## Fault Conditions

To catch and execute your fault logic, you need to know what to catch.

`X-Apigee.fault-policy` is set to the actual name of the policy that generated the fault. You could use this flag within <FaultRules> to execute the appropriate catch block.

If you raised a "Fault", you can always set your custom flow variables to denote the specific error that occurred. You can use those flow variables to execute appropriate "catch" block.

Two additional context variables are available within the error flow - `fault.name` AND `fault.category`. For all policy errors, Policies set

- `fault.name` to the specific error that occurred. Example, `VerifyAPIKey` sets `fault.name` to `InvalidAPIKey` if the app key doesn't match. Refer to http://docs.apigee.com/api-services/content/error-code-reference for the complete list
- `fault.category = Step` and
- `TypeofPolicy.SpecificPolicyThatFailed.failed = true` (e.g. `verifyapikey.VerifyAppKey.failed = true`) where `VerifyAppKey` is the policy type of `VerifyAPIKey` policy that raised this fault

**Default Fault Rule**

In addition, Edge supports a `<DefaultFaultRule>`.

```
<DefaultFaultRule name="all">
       <AlwaysEnforce>false</AlwaysEnforce>
              <Step><Name>CatchBadParam-B</Name></Step>
</DefaultFaultRule>
```

By setting <AlwaysEnforce> to "true", even if Edge already executed one of the <FaultRules>, it always executes this <DefaultFaultRule>. By setting this to "false", Edge executes <DefaultFaultRule> only if none of the <FaultRules> were executed.

## Raising Faults

Can you raise a Fault? Pretty much like "throw" in java?

Absolutely! By calling <RaiseFault> policy. RaiseFault policy allows you to set your specific error response (headers, body, status code, …) and once Edge executes this policy, it enters the error flow and executes appropriate <FaultRules>

Post the execution of <RaiseFault> policy, Edge sets the following variables

- fault.name = raisefault
- fault.category = Step
- raisefault.*SpecificRaiseFaultPolicy*.failed = true (i.e. the specific <RaiseFault> policy that executed / resulted in this fault gets set to "true".

**What if you fail in the error flow?**

note that if either <RaiseFault> fails (errors out) OR if a <Step> within a FaultRule fails, Edge stops the entire API flow and immediately sends an error response to the client. The error response contains the name of the Policy that failed (or raised fault).

*Critique – I would imagine that a failure while raising a fault should jump straight to fault rules but it doesn't!*

## How Can I Denote an Error from Within my Script Policy?

```
try {

    ..JS CODE..
    ..JS CODE..

} catch (err) {
    // set appropriate information so that your FaultRule can
    // catch it and send back an appropriate response
    context.SetVariable(reasoncode, 25) ;

    // To trigger the error flow, you have to throw!
    throw 'Error in JavaScript';
}
```

*Note – using JavaScript throw results in Edge logging a message and eventually uncleared logs may fill up the log system Alternatively, from within the catch, you can set some flow variables to denote that an error occurred and then have a conditional step to check and raise fault.*

*Question – If you set error object from within your JavaScript, would that state be available from within the Fault flows?*

# 14. Putting it all Together: API Proxy

Your API procesing logic is bundled up as one or more Proxies and deployed to Edge. A Proxy is a collection of XML specifications and optionally custom logic written in any of JavaScript or java or python.

There are three ways in which you can develop and deploy Proxies.

1. Use Edge UI (https://enterprise.apigee.com) to build and deploy a Proxy. This is a great way to get started.

   If your API project is structured in a way that one developer owns or builds one proxy (i.e. each API Proxy is fairly independent of others), then go ahead and use the UI! If you have multiple developers working on the same proxy, then, the default UI isn't the way to go.

2. Once you have designed the runtime (i.e. how many proxies? what would each proxy do?) and have a high level design of the flows (i.e. think of flows as reusable code fragments or functions) that suits your API project / team composition , your developers can use any of their favorite tools to build those XMLs.

   For API reference, check out http://docs.apigee.com/management/apis

   Of course, you need to have a mechanism of putting together all those code fragments into flows within endpoints within proxies. You can accomplish that by directly calling Edge APIs (http://docs.apigee.com/management/apis) or by using a build / deploy tool like https://github.com/apigee/api-platform-tools.

3. Of course, you could build your own build and deploy tools.

## Proxy Directory Structure

Apigee Deploy tool uses the following directory sturcture. For example, if your build system can create such a structure, you can use the deploy tool to deploy to Apigee.

Note – A word within [] is the name of such a file.

```
[ProxyDirectory]
        | [Proxy].xml                  // Name of your proxy. This XML contains metadata about
        |                              // this proxy.
        | policies                     // Directory that holds all the policies used by this Proxy
                | [Policy].xml         // An instance of a Particular Policy used within flows within
                |                      // this proxy.
                | [Policy].xml         // Another policy instance used within this proxy.
                | [Policy].xml         // Another policy
        | proxies                      // Directory that holds all you ProxyEndpoints.
                | [ProxyEndpoint].xml  // Name of your proxy end pont. XML defines the flows
                |                      // to execute for each basepath of a particular API call
                | [ProxyEndpoint].xml  // You can have multiple end points handling different basepaths
        | targets                      // Directory that holds all the TargetEndpoints
                | [TargetEndpoint].xml // TargetEndpoint represents a particular Target. In addition to
                |                      // defining the target, you can optionally define flows to
                |                      // execute
                | [TargetEndpoint].xml // you can have multiple TargetEndpoints for different scenarios
        | resources
                | [Resource file]      // eg - JavaScript code you want to execute within a flow
                | [Resource file]
```

## Proxy

The top level XML file defines all the code and resources that are part of this Proxy.

```
<APIProxy revision="4" name="HandleRetailInformation">
    <ConfigurationVersion majorVersion="4" minorVersion="0"/>
    <CreatedAt>1475579629929</CreatedAt>
    <CreatedBy>srinivasulu.grandhi@gmail.com</CreatedBy>
    <Description/>
    <DisplayName>simple</DisplayName>
    <LastModifiedAt>1475579629929</LastModifiedAt>
    <LastModifiedBy>srinivasulu.grandhi@gmail.com</LastModifiedBy>

    <Policies>
        <Policy>VerifyAPIKey<Policy/>
    <Policies>

    <ProxyEndpoints>
        <ProxyEndpoint>HandleCatalogue</ProxyEndpoint>
    </ProxyEndpoints>

    <Resources/>
    <TargetServers/>
    <TargetEndpoints/>
</APIProxy>
```

Policies — List of all policies referenced within the ProxyEndpoints within this Proxy. A Policy is a predefined functionality or a custom function. Your API processing logic is written as a sequence of policies, called flows

ProxyEndpoints

List of all ProxyEndpoints within this Proxy. A Proxy end point is an entry point to your code, called when an incoming API call matches the basepath as defined within the ProxyEndpoint. You can have multiple proxy end points within a Proxy, each with its distinct basepath.

In the above example, there is one proxy end point by name HandleCatalogue.

Resources — Scripts, JAR files and XSLT files that are referenced by any policies within this Proxy to execute custom logic.

TargetEndpoints

List of all TargetEndpoints referenced within the ProxyEndpoints within this Proxy. Think of a TargetEndpoint as a code encapsulation to handle a specific target. i.e. if your API logic is specific to handling a particular target, it's a good idea to embed those flows within the TargetEndpoint so that it can be reused.

## Proxy Endpoint

Each ProxyEndpoint handles a distinct basepath. You can't have two Proxies or two ProxyEndpoints handling the same basepath!

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ProxyEndpoint name="HandleCatalogue">

    <Description/>

    <PreFlow name="PreFlow">
        <Request>
            <Step>
            <Name>VerifyAPIKey</Name>
            </Step>

            <Step>
            <Name>RemoveAPIKey</Name>
            </Step>
        </Request>

        <Response/>

    </PreFlow>

    <Flows>


    </PostFlows>

    <HTTPProxyConnection>
        <BasePath>/catalogue</BasePath>
        <Properties/>
        <VirtualHost>default</VirtualHost>
        <VirtualHost>secure</VirtualHost>
    </HTTPProxyConnection>

    <RouteRule name="CatalogueBackend">
       <URL>https://retail.mybackend.com</URL>
    </RouteRule>

</ProxyEndpoint>
```

You are essentially telling Edge container to call the flows within the ProxyEndpoint for the basepath defined within <HTTPProxyConnection>

Once ProxyEndpoint :: Request PostFlow finishes, <RouteRule> defines the target to which Edge should send this API call to. In the above example, the URL is directly specified within the <RouteRule>. Alternatively you could specify a TargetEndpoint.

**Routing Rules**
Routing rules are a specific instance of conditional flows (i.e. "if-elseif-elseif—else" block of code)

```
<RouteRule name="GetStatus">
<Condition>request.verb="GET"</Condition>
    <TargetEndpoint>StatusTarget</TargetEndpoint>
</RouteRule>

<RouteRule>
      <URL>http://api.usergrid.com</URL>
</RouteRule>
```

*Critique – Notice that there is no <RouteRules> tag that encompasses all RouteRules. On the other hand, <Flows> embed all your flows. The probability of having multiple flows is much higher than having multiple target routing rules. Personally, I love consistency, makes it easy to learn a new skill.*

## Target Endpoint

Each <TargetEndpoint> is a code encapsulation for talking to a specific target. A Proxy can have zero or more TargetEndpoints. <RouteRule> will determine, at runtime, which one to pickup and execute.

```
<TargetEndpoint name="Usergrid_Target">
    <Description/>
    <FaultRules/>
    <PreFlow name="PreFlow">
        <Request/>
        <Response/>
    </PreFlow>
    <PostFlow name="PostFlow">
        <Request/>
        <Response/>
    </PostFlow>
    <Flows/>
    <HTTPTargetConnection>
    </HTTPTargetConnection>
</TargetEndpoint>
```

# 15. Zero Downtime Deployment

Your APIs are mission critical, needs to be up 24x7. As you enhance and upgrade APIs, it is critical that the deployment doesn't require a downtime.

**Versioning**
Its normally a best practice to include your API version number within your basepath. If you have a breaking change, you can then increase the version number (which results in a new basepath) and deploy the proxy. This ensures that both the old version of the proxy (with the old basepath) and a new version of the proxy (with a new basepath) are active at the same time.

**Zero downtime deployment**
In general, any configuration or code change that you make to your organization or environment or proxy gets immediately reflected at runtime. Edge updates the configuration and triggers a notification to respective runtime components to refresh their inmemory copy.

*Note – There is, of course going to be a slight delay from the time the notifications are sent to the time the notifications are received and processed by the runtime components*.

**You don't necessarily need to deploy the whole proxy every time you want to update an existing version.**

Here are a few examples

You can disable a step at runtime without requiring redploying. For example

```
http://{ManagementIp}:8080/v1/organizations/{organizationName}/apis/{apiName}/revisions/{revision}/stepdefinitions/{stepName}?enabled={enabled}
```

```
enabled = true | false
```

You can also use the APIs to modify existing flows without redployment. Note that the APIs allow you to create a new flow or append to an existing flow but does not support insertion within an existing flow.

# 16. Before You Start Your API Implementation

## API Design

Your API layer is typically a façade to your backend systems. The APIs you build and deploy on Apigee Edge is primarily geared towards enabling consumption of your services by others (your own team, other groups or divisions within your company OR external partners OR developers).

Design your APIs, with an outside-in perspective. Brian Mulloy's excellent whitepaper is a must read - http://apigee.com/about/resources/ebooks/web-api-design

## Leverage Apigee Edge for Speed and Agility

A key aspect of your transformation to Digital is "speed and agility". From an API standpoint, it translates to (a) build APIs quickly (b) evangelize (c) learn about what's working and what's not and (d) change quickly (e.g. maybe the APIs need to be optimized for performance. Maybe new API SKUs are needed for different developer audiences. Maybe pricing needs to be tweaked. Support multiple API versions to get real world usage feedback etc)

Understanding how Edge enables you to build quickly and be agile helps you to take full advantage of Edge.

**Build quickly, your way**

If you have multiple developers building your APIs, you need a code structure and an API development process / life cycle that doesn't hamper your speed. You need to organize your API logic for reuse, change management, parallel development by a team etc. While there is no one way that works for every API project and team, Edge provides a flexible way for you to organize to suit your specific needs.

For example, Edge provides

1. Powerful Management APIs so that your developers use their favourite IDEs to build and deploy APIs.
   *note – Apigee Edge provides rich APIs to build your API Proxy step by step.*

2. ORG-Env model to support flexbile deployment models and "path to production".

3. Flexible way to structure your code for parallel development and enabling reuse
   a. How many Proxies? What would each be responsible for?
   b. Flow segments (reusable flows)
   c. ProxyEndpoints.
   d. TargetEndpoints.

4. Tools to quickly publish documentation.

**Change Quickly**

Edge provides a set of mechanisms to help you iterate quickly. For example,

1. Versioning allows multiple versions to be deployed seamlessly.
2. Out of the box analytics helps you identify the bottlenecks.
3. API Products to quickly tweak and release new API SKUs.

**Edge is built to enable speed and agility. Leverage those capabilities to accelerate your journey to DIGITAL.**