Thinking in Policy

Policy Development in the CA API Management Product Line

Contents

Introduction

The CA API Management Gateway product line is focused on the world of APIs and Services, and is considered by most to be part of network and security infrastructure. In technical terms, it is a reverse proxy that deeply inspects network requests and responses from external and external client software in business to business communication, business to consumer, business to employee communication. The client software ranges from mobile applications to web browsers to business systems.

This guide is meant to explain how the product works at a conceptual level, and introduce the tooling prospective policy authors and architects will work with.

The Gateway product was built to secure and protect APIs and services from attacks; hence one of the main uses of the gateway is to make an API safely and securely available for public use. There are additional uses for the gateway in orchestrating related API calls into composite functions, but unlike a workflow product or an ESB the gateway is not capable of very long lived workflow management, and therefore is not suited to some pure ESB or Workflow use cases.

This guide will introduce the core concepts of how the gateway is configured, and then how these configuration items are used, and guidance on how to use the policy language to implement common use cases.

Additionally, the usage of the gateway in an enterprise requires a strategy to move configuration between the corporate environments that mirrors the process used to deploy other IT systems to a production environment, so a discussion of best practices for enterprise deployment is included.

Understanding the way the gateway was designed to implement standard use cases provides insight into what information you'll need to have at hand during policy development.

Services and Policies

At a high level, the basic unit of configuration in the Gateway product line is the Service. For the gateway a Service is a logical construct that represents the sum of the API calls the client side can call to access the service capability the gateway is protecting.

Every service has a policy that implements an individual flow of data between the client and the back end service. Optionally policies may include other modular partial policies, referred to as policy fragments or encapsulated assertions. Typically modular policies have specific roles in authentication and authorization, routing to back end services, and orchestration of larger functions.

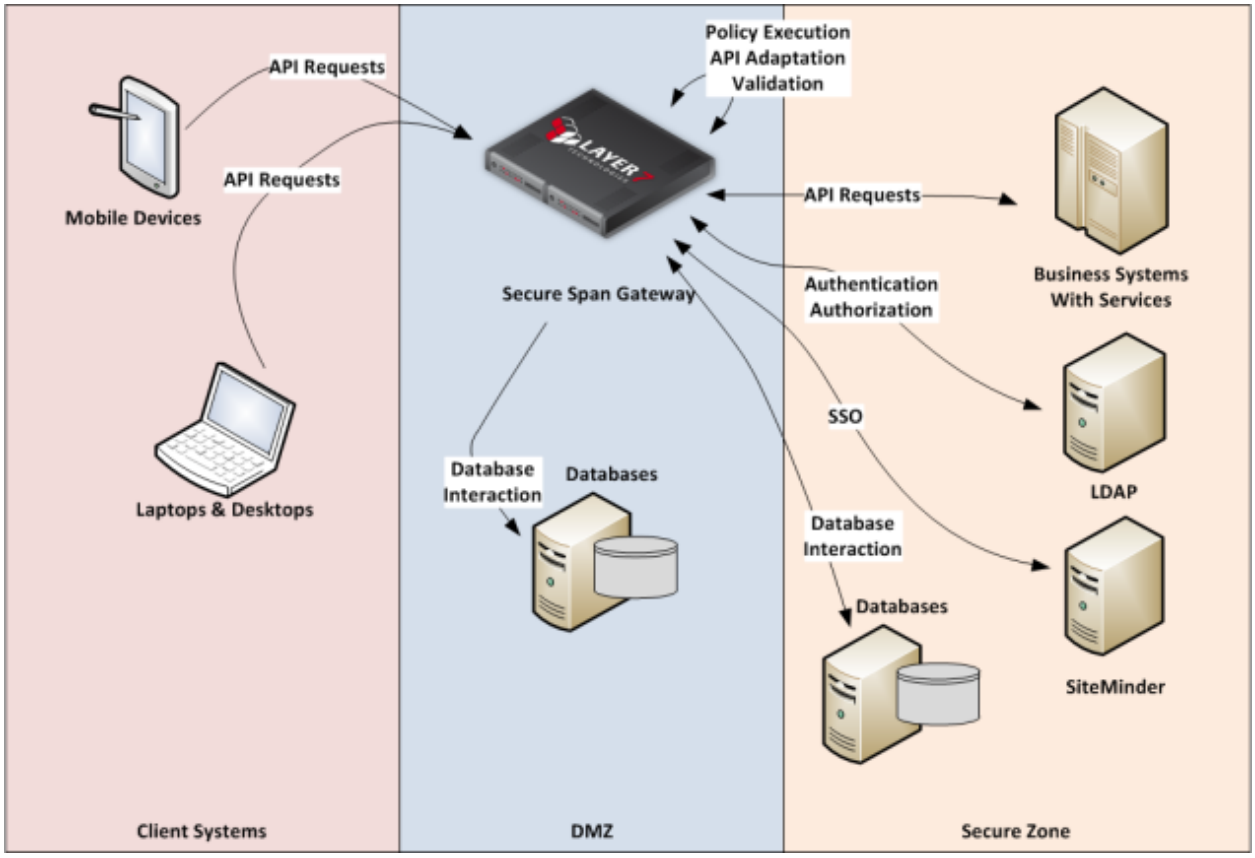
Gateway policy refers to and sometimes encompasses configuration items derived from data often held in quite disparate groups within an enterprise: The security architects, the authorization and authentication group, the network infrastructure team, the application developers and the service architects all serve roles in defining the details necessary for a policy.

As an example, the group membership strategy of a given LDAP server will be available from the authorization and authentication team. The network infrastructure team will have the IP addresses of routers and load balancers. The service architects will have access to the service URLs of back end services, and have access to XML Schema documents, WADL & WSDL documents, etc. The Security architects will decide if http basic credentials over anonymous SSL are sufficient security, or if Mutually Authenticated SSL is necessary.

Environment & Configuration

Production gateways are normally deployed in the DMZ, and provide services for mobile and client systems located in less secure zones.

Often the business services the gateway is intended to provide secure access for are located in the secure zone. In the Network diagram below, there are multiple databases; the gateway doesn't require any specific network location, but often network and security architects require that all databases are located in the secure zone. In our role as a security policy enforcement point, we are in the DMZ.



Field implementations need to cover both environment configuration and service "programming": Many of individual policy operations have configuration data needed: Query a database, an LDAP, an external system via SSL. Each has configuration, the credentials for the database, or the SSL certificate for the external system.

These parts aren't directly in policy, but are in common configuration storage. Migration between environments is deeply concerned with that.

As mentioned, services and policies have dependencies to external resources; these usually have details specific to each physical environment the Gateway is deployed in: IP addresses of back end systems and LDAP specific details are often different, and so often refer to those kinds of resources as "Environment Dependencies". Planning for correct per-environment changes to environment dependencies is critical in policy development and especially migration.

In most cases, environment specific values should use the cluster property feature. Cluster Properties allow you to refer to a common configuration item with a symbol instead of a literal value, providing with a very simple way to make wholesale changes to many references. Cluster properties are referred to in the policy editor as "${gateway.property_name}".

For instance, in an HTTP routing assertion, if you use the cluster property "${gateway.BusinessService}" in the host name field, you can define the BusinessService cluster property as "dev.bizsvc.company.com" for your development environment and as "qa.bizsvc.company.com" in your QA environment. When you move a group of policies that all refer to that business service from development to QA, you need change only the value of the cluster property, and the QA gateway policies will send requests to the appropriate server, in the example given, qa.bizsvc.company.com. This concept generalizes to most shared resources: app servers, database servers, LDAP services.

Most gateway subsystems rely in various ways predefined cluster properties for some details of configuration. The gateway component discussion later describes the most important of the subsystems. An easy example is the HTTP input subsystem which uses the predefined cluster property ${io.httpCoreConcurrency}, which defines the number of simultaneous http based inbound connections allowed at one time. The default of 185 reflects a conservative estimate of what kind of traffic expectations would cover the most common deployments.

Just as with user created Cluster Properties; all nodes in a cluster share exactly the same configuration, greatly easing installation of larger environments. The ability to create new cluster properties on demand extends the power of cluster wide configuration changes to our policy language. There are several hundred pre-defined cluster properties documented in Appendix D of the browser and pdf manuals.

Gateway Design

At its most basic, the gateway consists of a protocol independent message processor, network input subsystems, and assertions that provide actions in the message processor, and output capabilities as well. The gateway was, very early in its product development an Apache Tomcat application but that is no longer true, and so it is not an app server.

We describe the message processor as being protocol independent because it operates on a message, not an HTTP message or an FTP message. How that message was received is not considered. The back end system that

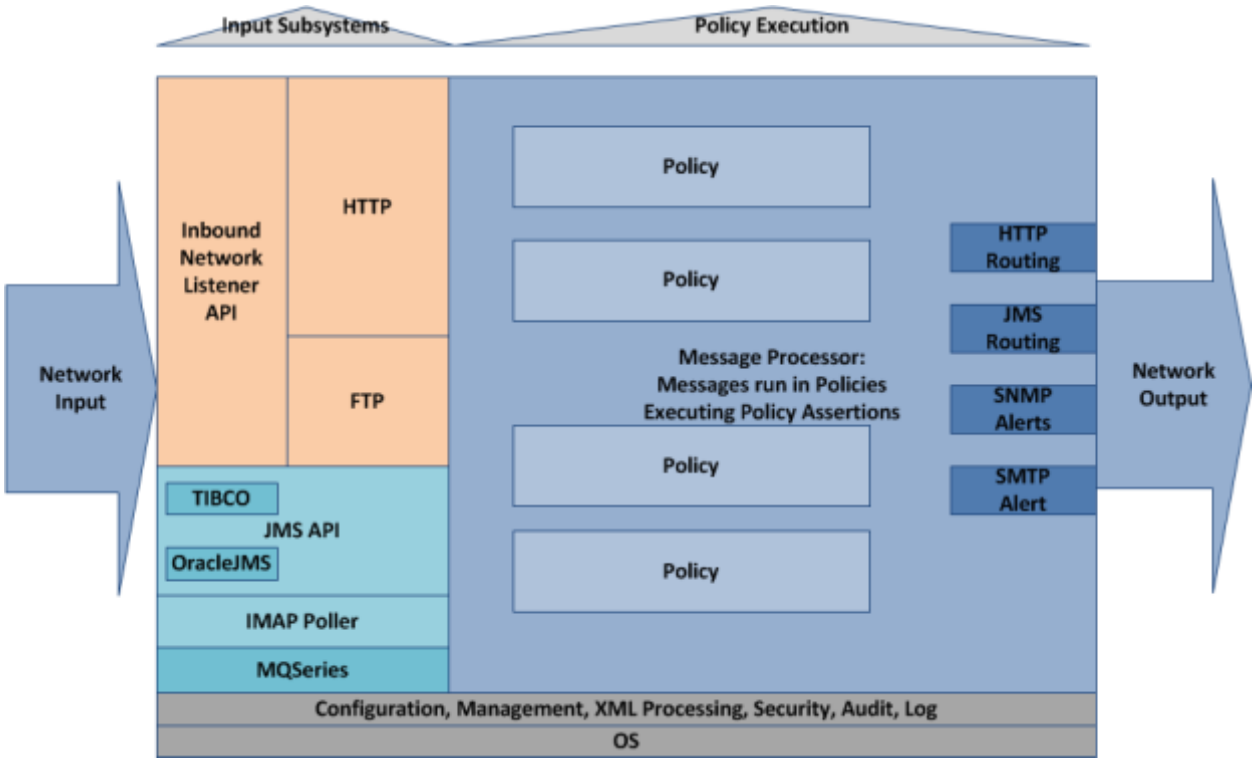
the request is eventually sent to does not have to be the same technology; it's common to accept a message on HTTP and send that to a back end using a JMS Queue.

The product allows environment dependencies to have shared configuration between different services: It is very probable, for instance, that though an environment may have multiple LDAP servers, there will be more than one Service that authenticates against each LDAP. If that definition is re-created multiple times, the chances for errors start to increase.

The original product design was partly based on the concept of "XML Firewalling", so it has some parallel concepts to an IP Firewall. The gateway does no processing without a service defined, so similarly to the best practices in IP firewalling i.e. all tcp ports are closed by default, in the gateway all messages are suspect and therefore possible attempts to bypass security until proven otherwise.

This means that the gateway does not pass any data by default. If something in an inbound request is unexpected, the gateway is designed to block that information. In detail, this means that the gateway doesn't copy most HTTP headers from request to back end unless specifically configured to do so; this is because the gateway is responsible for protecting back end systems, and if the data is not explicitly referenced, the gateway is designed to block that data. Additionally, services and APIs are usually textual, so binary data can require special handling, as interpreting binary data in policy context can require decoding and encoding steps.

Gateway Components



Message Input

Message input into the message processor is via network listeners like HTTP and FTP, or outbound connectors that initiate connections to external services like IMAP Polling, Tibco EMS or MQ Series, which function by connection to an external server, retrieving messages and putting those messages into the message processor. The complete list of connectors changes every release, and there is an SDK to add additional network output protocols. Message input connectors are more complex, so need a different, non-public SDK.

All of these input systems have two roles: Provide the network protocol as appropriate and pre-fill some network level details that are used later in policy. For instance, the HTTP connector puts the remote IP address into a special context variable; more on that later in the document.

Message Processor & Policy Execution

This is a multithreaded, very efficient, and above all relatively simple system.

At its most fundamental level, the message processor runs policies that invoke functions sequentially on an inbound message following the flow control in our policy logic. Functions are called Assertions in the product.

Assertions store state within the context of the currently executing policy via the request and response objects, and via the glue of the policy language, context variables.

Note that there is no specific message output system defined. This reflects how the gateway is protocol independent. If a use case has a back end system on JMS, then the policy uses JMS Routing assertions. If the back end system is SFTP, then the policy similarly uses the SFTP Routing assertion. This implies an important point: Back end requests are explicit policy actions.

Support Tooling

The support tooling provides services used by multiple assertions and the listeners: Certificates, caching, connection pooling, audit, logging, management, token services etc.

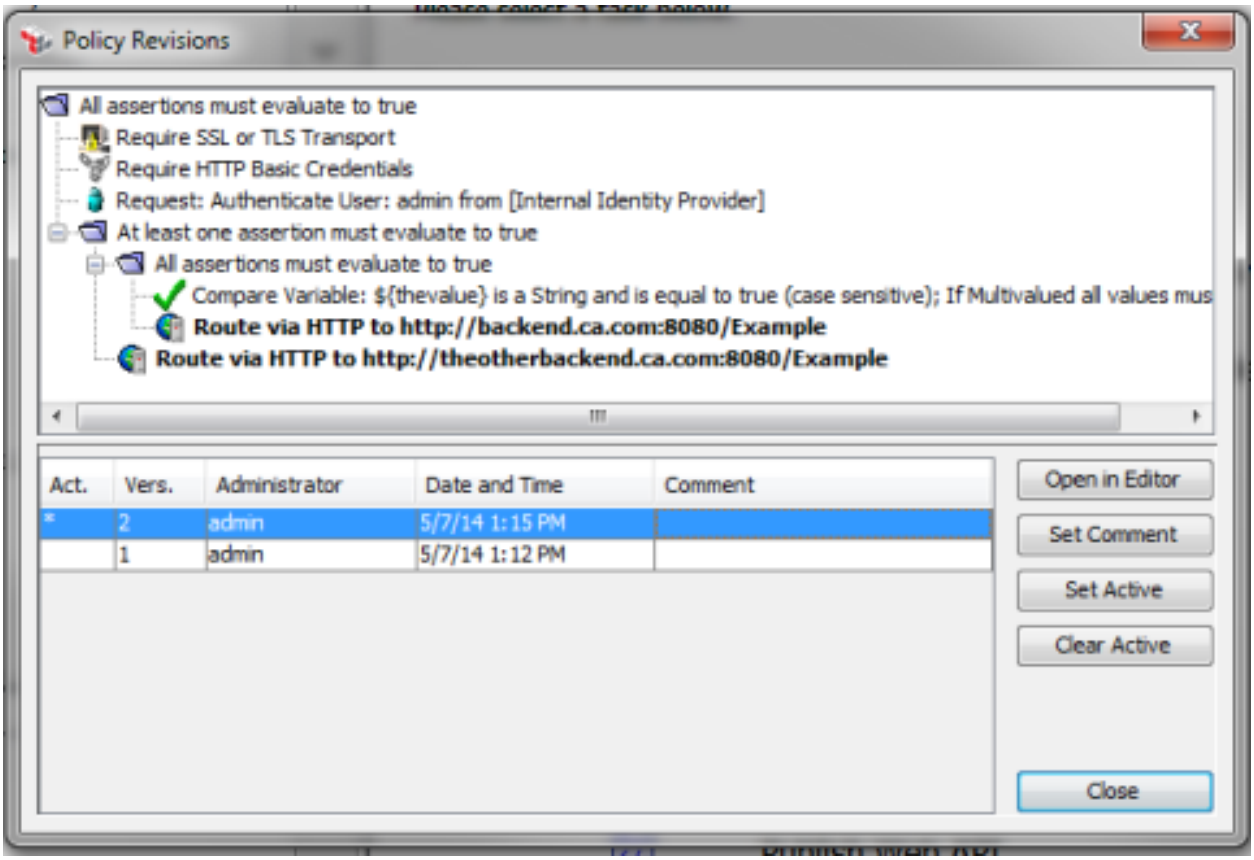
Developing Policy

At its most basic level, Gateway Policies provide APIs to client systems, while in turn utilizing back end APIs. Enterprises need to have the same kind of governance that is used in software deployment to govern policy deployment.

Policies have details that can dramatically alter the structure of API calls; as such they need to be deployed with similar kinds of rigor as the software that uses them API calls. We often refer to that kind of effort as a "Policy Development Lifecycle"; having a long list of parallels to standard software development lifecycles.

Development of Policy is Usually Iterative and has Environment Dependencies

Fully planning a service or API deployment is not particularly easy or even practical; reality forces us to make changes. This is fully supported in the gateway with version control. This is helpful, but the gateway-based version control doesn't extend to multiple environment support. As shown below, policy authors can view and edit versions, and set previous edited policy as active. This is covered in the policy manual in Chapter 6: Working with Revisions.



Expecting changes in policy is one of the reasons why certain resources are in common storage. It would be a poor use of policy development time if LDAP configurations had to be defined and managed on a per service basis.

Allowing environment dependencies to change without the structure and content of the policy changing is one of the ways the gateway enables efficient workflow. This also allows use of policies developed in development environments largely unchanged in production.

As mentioned previously, planning for environment dependencies is part of policy development.

Policy can be abstracted into Modular Pieces

Modular policies are referred to in the documentation as Policy Fragments and the newer Encapsulated Assertions. They work in much the same way as the C language "#include" works: the fragment becomes part of the policy, and executes using the same rules that any policy runs.

Modular policies allow subject matter experts and specialists to work on individual functions without disturbing other policy development in process, much like the standard practice of dividing large development projects into subsystems and assigning different teams to each subsystem.

For instance, it is very common to create a common "Authorization and Authentication" module as policy fragment. This reduces errors dramatically, and also, since the product has role based access control; this module or fragment can be protected from change by inexperienced staff by allowing policy authors read-only access and allowing your security team read-write access.

The documentation has more detail in the section named "Working with Policy Fragments"

Local Environment Dependencies are Critical

When policies move from a development environment to a test environment, most IT operations groups have different servers for various functions; these will have different host names or IP addresses.

The product supports mapping these kind of dependencies; when policies migrated a second time, the already defined local resources are usually reused, to reduce error rates.

Objects and Actions in Policy

In the policy language, there have several kinds of high level objects that we operate on with the actions in policy. Actions are called Assertions in the product and documentation, mostly because the early product planning implemented ideas from the WS-Policy working group, and the Assertion concept was very central to that effort.

Our Policy data is Messages, Strings, Certificates, Dates, and More

There is a long list of types of data handled by the gateway: Messages, Strings, Numbers, Certificates are some examples. Each assertion in policy will act on objects based on configuration and some will use configuration from environment dependencies.

Actions based on configuration might be a regular expression configured entirely by the text in the dialog. An example of actions based on configuration might be how the Authenticate assertion will query the configured LDAP server to see if the supplied credentials are valid.

Context Variables

No language is complete without a way to make data available in manageable units. In our policy language we call these context variables. There is a long list of automatically available policy data, and it is well covered in the documentation in a section titled "Predefined Context Variables".

For instance the request and response are the two most important predefined context variables, and a great deal of interaction with the common assertions is via the default data that represent elements that are always present during policy operation. You can also refer to attributes of that data item via an explicit variable reference like *${request.tcp.remoteip}*.

Similarly policy can specifically create new context variables to make more sophisticated policy possible. Some assertions automatically create context variables, for instance the XPath assertion saves data about what information was found when run in a list of variables.

A particularly complex type of context variable is the Message type. This embodies the ability in the gateway to consult multiple sources of data synchronously and asynchronously, and the ability to consult external systems to make decisions within policy, without disturbing the flow of the main request and response data.

Policy Elements are Program Actions

Assertions act on the policy data mentioned previously; an assertion might change the request as configured in that specific policy: an XSL Transform Assertion could modify the request message according to the specific configuration in that specific policy. All assertions return a Boolean true or false, meaning the assertion is true, in general this means success or failure.

Composing Policy Logic

Policy logic has some areas that can be confusing, but the main composition is via the composite assertions often referred to as "AND" and "OR". The Boolean return values of the assertions, combined with the composite assertions allow policy to express some very powerful decisions. The Policy Manager visually uses a folder metaphor to imply that the composite "contains" other assertions and to imply a parent-child relationship, and the gateway uses the child assertions as inputs in terms of binary logic.

Composite Assertions

In the policy palette we have wordy but more understandable names for the common composites: "All Assertions Must Evaluate to True Assertion" which operate similarly to the logical "AND", meaning all of the inputs must be true.

Similarly "At Least One Assertion Must Evaluate to True Assertion" operates similarly to the logical OR, meaning only one assertion needs to be true for the OR to be true.

These composites run their child assertions until they change state: AND runs its child assertions sequentially until one fails, and OR runs its assertions until one succeeds.

Ordering matters because we only run assertions until the condition of the parent composite is satisfied.

Since ordering matters and policy will halt execution of a path of policy execution on failure, a best practice is to put the "cheapest" assertions earlier in your policy. For example, since checking if SSL is present is a very quick assertion, and if SSL is always required, it quickly weeds out attack attempts with a Non-SSL message, for instance.

Examples of quick assertions are assertions that simply check for the existence of certain data. SSL is one example, and Require HTTP Basic Credentials is another. In comparison, XSL Transform Assertions are not as cheap because they incur CPU usage as they process a message, and of course, would require more CPU when the message is larger.

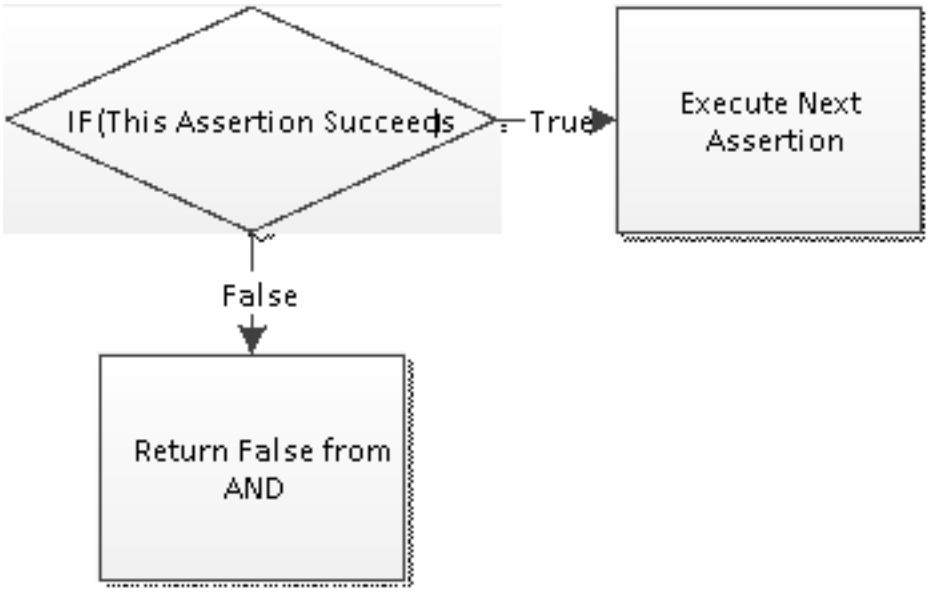
Building Common Structures

IF-THEN

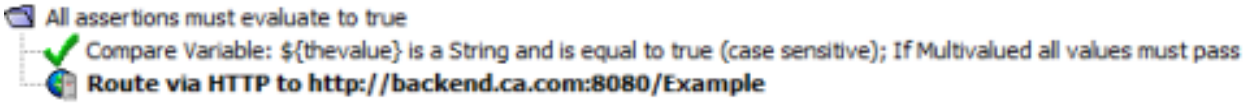
Because assertions execute in sequence, the AND assertion acts as the standard programming "if" statement. That's slightly confusing because in a strict definition of logical AND, all inputs are evaluated. Because we don't evaluate all inputs, we sometimes refer to our AND as a "short-circuited AND". This means that the first child assertion that returns false will cause the entire AND to return false immediately.

Turning that back into policy this means that If/Then is one AND with two child assertions inside it. If the first assertion succeeds, the second is attempted, giving the effect of If-Then.

In Flow charts this looks like a simple decision:



In policy this looks like the following. Visually, composites look like folders, and this is meant to imply that the "All assertions must evaluate to true" has two child assertions that are evaluated in sequence.



This could be read as "**If** the context variable *${thevalue}* is equal to the string "true", **then** attempt to route to the service".

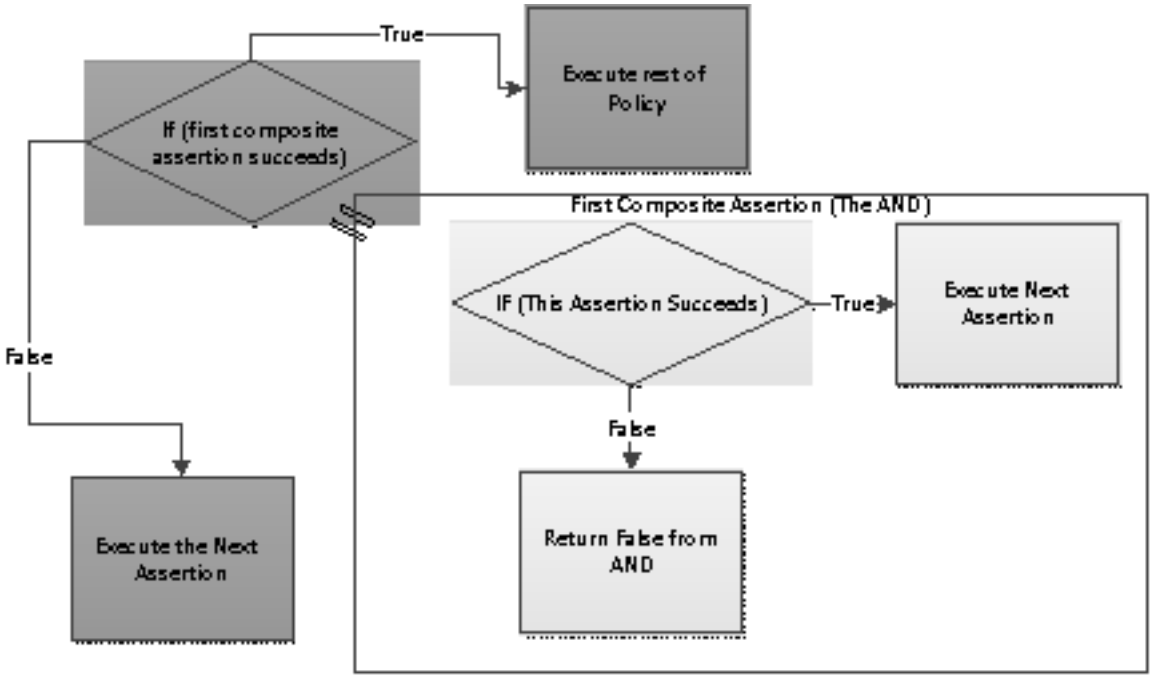
IF-THEN-ELSE

OR, in our policy language means that the next child assertion is attempted if the previous assertion returns false. Using the structure from before as 'If-Then', the OR becomes roughly equivalent to 'Else'. This is repeated if there are multiple children of the OR assertion.

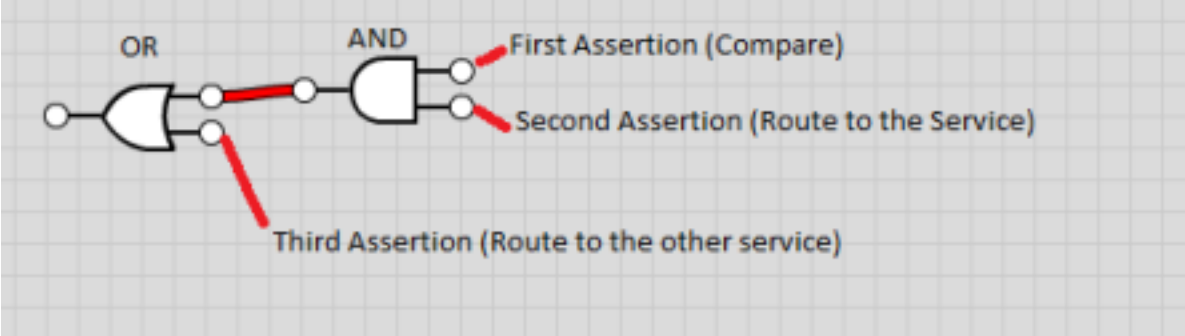
So to assemble an if-then-else capability, we nest an AND inside an OR assertion.

If the first assertion fails, in our example the "Compare Variable" assertion, the gateway doesn't attempt the second child of the "All assertions must be true" composite, the 'Route Via HTTP' assertion, because the failed compare means the AND composite assertion immediately returns false, and subsequently the OR will execute its next child assertion, providing the ELSE function.

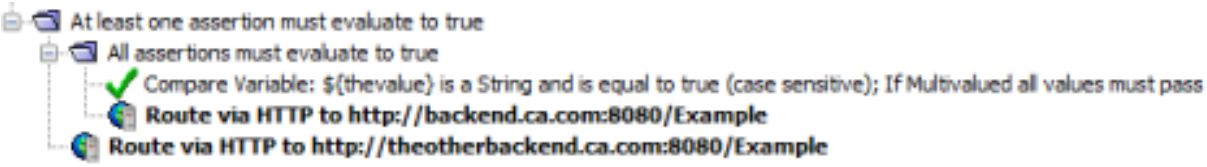
In a flow chart, it looks a bit more complicated:



But to show this in terms of formal logic gates, this looks a lot simpler



In our policy editor this simple case looks like the following. Note that the previous composite assertion containing the compare and route assertions are visually set up to be contained in the At least one assertion. Given that the return value from the composite would be false if the compare assertion fails, the gateway will attempt the other routing assertion.



This is interpreted as "**If** the context variable *${thevalue}* is equal to the string "true", **then** attempt to route to the backend service, **else** attempt to route to the other backend service".

Composite Assertions Define Policy Flow, but can Hide Errors

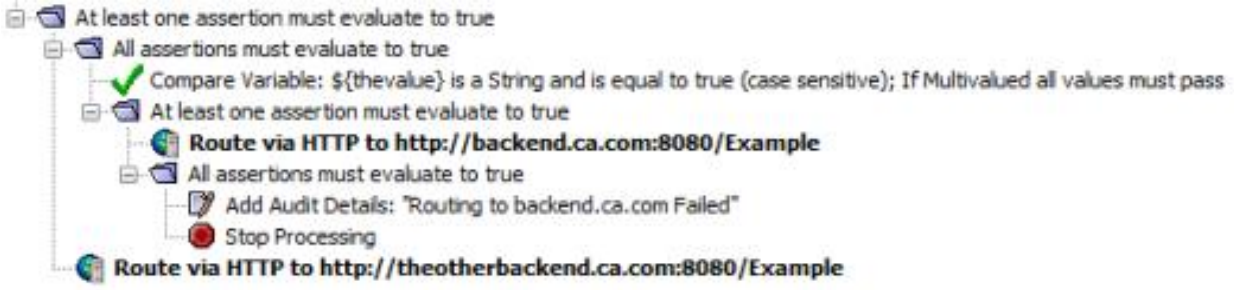
All Assertions return false if the desired condition isn't met. This can mean that a string comparison is not equal, or if an XPath didn't find an element in the document. Or it can mean that the back end system didn't successfully acknowledge our HTTP request.

Because OR runs the next assertion if the previous assertion returns false, and errors are always false, this means it is possible to have an error hidden by policy logic, because OR runs the next assertion if the previous one returns false.

An intentional effect of OR running the next assertion if the previous one returns false is that we can use it to do policy actions on failure: to have direct control of alerting, an OR containing a routing assertion could be followed with SNMP trap with the message "Http routing failure" that would only be sent if the routing attempt fails.

This pattern provides a convenient alert mechanism, but the Audit and SNMP Trap assertions always succeed, so it can be confusing to the policy author because it will cause the OR to return true, so a STOP assertion is needed after an explicit audit or trap. Note that Audit here succeeds, which, if put immediately after the routing assertion would cause the OR to succeed, and would be a logic error. Instead, the approach is to use an AND assertion with a STOP to cause a branch of policy to return false.

In the policy editor, the general structure is similar to the following.



This is interpreted as 'If the context variable *${thevalue}* is equal to the string "true", then attempt to route to the backend service. If that routing attempt fails audit a message, and stop and fail that sub branch of the policy. Finally if previous policy failed, attempt to route to the other backend service'.

Audit context always holds the assertion error data even if no Audit assertion is encountered, so it's possible to have a case where a service is working as planned, hiding an error from users, reporting errors to the monitoring tools. In most cases, this is a correct and desired outcome, but can be unexpected.

Complex Policy Structures

The above simple cases don't show more complex interactions, and properly explaining them all is beyond the scope of this document. However, like any development project, decomposing problems into modular units is the key to success. The policy fragments are a critical element of large project development.

As an example some features of the extended product family are delivered entirely as policy: The OAuth Toolkit and the Mobile SSO capabilities in the Mobile Access Gateway are examples. Both of these efforts use policy fragments extensively to boost reliability and lower maintenance.

Policy Style for Maintainability

For the most part it is about making the code readable and leveraging the comment capabilities to an extreme.

These suggestions are very general, and obviously local style guides might supersede these recommendations, but in the spirit of attempting to describe best practices, this style guide is the basis of our current internal training on policy authoring

- Always include a comment block at the top of the policy.

  It should describe the policy in general terms and more specific terms if required.
- Include versioning notes in the comment block.

  Since policy is the only artefact to survive migration through the enterprise, this is key to tracking changes
- Include information about passed arguments if policy is tied to an encapsulated assertion as well as any cluster-wide properties and global shared context variables.

- This should be in the comment block.
- Actual references are usually buried deep in policy so it makes life simpler to have them documented up front.

- Use copious amounts of right side comments to describe what is going on.

  Many assertions hide their configurations inside the dialogue so the only way to understand what is happening in them is to open them, or describe in a comment.

- Use left side comments sparingly.

  The policy language natively reads "VERB ACTION" and adding left side comments breaks up the readability. Only use left side comments to "call attention" to something. Otherwise use right side comments for everything.

- Right side comments should ALWAYS start with a comfortable delimiter, like //.
  - This mirrors a syntax that most people are familiar with and makes the policy more readable.
  - Simply "greying" out the comment often is not clear enough.
  - Also if opening policy in a fragment then all content is greyed out so there is no distinction for the comment.

- All logical folders ("all must" and "at least one must") MUST have a right side comment describing the contents within.

  One should never be required to open and analyse a folder to get some idea of what it does.

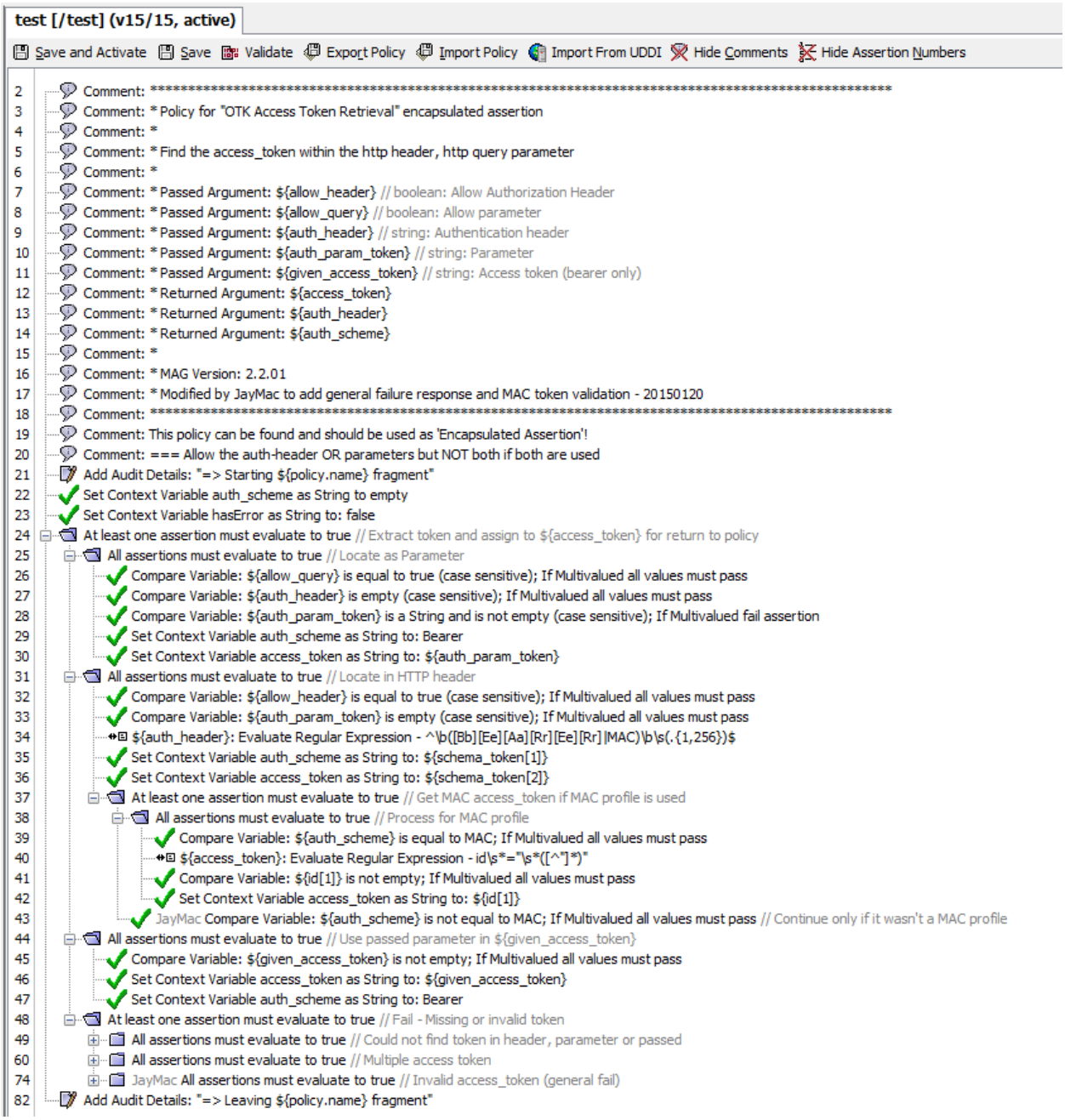
- Use "All" folders to logically group functionality.

  It makes policy more readable visually, and allows the collapse function to hide logical structures, and shows the comments

- **NEVER** use a template response to return an error message.

  ALWAYS use the Customise Error Response assertion and fail the policy.
  This ensures that the Gateway knows and records policy failure properly.

An example of the style:



Decomposing Use Cases into Service Policies and Fragments

Creating a Service and Service Policy

Our product was built with SOAP services in mind, so a common beginning task is to find the WSDL associated with a service and "publish" it. This is increasingly less common now with REST and JSON services being more common. As there's a distinct lack of metadata in non-SOAP services, it becomes more effort on the policy

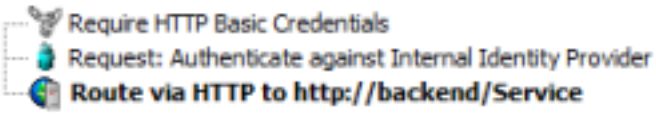
architect to pull in the various metadata needed. In some ways, publishing a simple service is easier in JSON than any other kind, and publishing a complex service with strong validation is much harder.

The Simplest Possible Case: Authentication and HTTP Routing Assertion

By far this is the simplest and a very common use case. At its trivial simplest, this policy consists of just three policy elements: Gather credentials, validate the credentials, and finally send the original request to a back end server. In this case, we're allowing all valid users to access the back end resource, so Authorization is implied by Authentication.

Though not mentioned elsewhere specifically, in the gateway, returning the back end server's response content to the user is implied by a policy finishing successfully.

As there's no optional events and no decisions that result in anything other than "This is not a valid request", there's no need for any composite assertions, no branches into conditional states. The policy looks like this:



Gathering Credentials

We support quite a few kinds of credentials: e.g. as shown above, usernames & passwords both from the http network transport level, and also message and from the transport level, various single sign on tokens, X.509 certificates, SAML tokens, JWT, and other more obscure ones. You need credentials available in policy before you validate them. That further underscores the sequential nature of policy execution, because you can't validate credentials if you haven't extracted them from the inbound message yet.

Tokens

VALIDATING CREDENTIALS/AUTHENTICATION (AUTHN)

Validation implies mostly the authentication step, but some tooling, like RADIUS implies both AuthN and AuthZ. The policy manager will warn you that you extracted credentials without validating them, and we consider it best practice to validate credentials. However, there are use cases that gathering credentials and validating them with custom tooling would cause the warning to be shown, for instance if the policy validates credentials via direct database lookup. For this reason, the policy manager will not prevent you from saving the policy. Note that gathering and validating credentials is done via always present data structures called the authentication context. This is largely automatic but there are cases where specifically referring to this context is done via context variables. See the section in the manual beginning with "Predefined Context Variables" in the subsection "Authentication Variables"

AUTHORIZATION (AUTHZ)

We often talk about authorization in terms of explicit and implicit authorizations: The explicit authorization of examining the results of authentication for attributes, and the implicit authorization of using external decision tooling like Single Sign On tooling, or purpose built local authorization tooling

LDAP

We normally use LDAP in a large portion of customer deployments. We organize them into "Identity Providers", which provide a convenient way of authenticating and extracting attributes to base later decisions on.

WORKING WITH SITEMINDER AND OTHER SSO PRODUCTS

SSO tooling often combines AuthN and AuthZ into a single call, leading to simpler looking policies, at the expense of putting the entire effective policy in two areas: partially in the Gateway as a policy enforcement point, and partially in the SSO tooling as a policy decision point. That separation can cause headaches, though there are good reasons based in corporate governance to keep the SSO tooling based decision point.

ROUTING

Sending requests to a real service once validated is by far the most common deployment of the gateway. There's a ton of detail in the routing assertion, so it's not something a quick overview can cover. In the common case, normally the gateway has some kind of authorization to the back end; this can be passing the original request credentials, or a role account, or mutually authenticated SSL.

Mediation Cases

One of the unique things about the design of the gateway is that it is not exclusively built as an HTTP gateway. It is implemented as a message processing engine, where a large portion of the available tooling is built for HTTP based use cases. In a lot of ways, the Gateway message processor is unconcerned about the way the message entered the flow, it is tasked with running the policy until it exits. All processing steps are processed with those clear AND/OR logic rules so almost any combination of input and output can be created.

Transport Mediation

A relatively common use case is to receive a message on HTTP and send the message to a queue like MQ Series or Tibco EMS. This consists of a policy nearly identical to the common case shown above, but instead of an HTTP routing assertion, it uses a JMS routing assertion or an MQ Native assertion.

Token Type Mediation

Also similar to the simple case, but this time once credentials are validated, there's a policy step to create or obtain a different kind of credential or token; this can be as simple as HTTP basic credentials inbound, and SOAP with WS-Basic Authentication to the back end. Token creation assertions and SSO authorization assertions are part of the authorization palette. Subsequently we add the credential tokens to either the response message, or to the response HTTP context in the case of browser style interactions, or the request message or context in the case of SOAP or REST style interactions.

Note that the gateway doesn't require redirects to add tokens to the back end request context, because our gateway technology does all this work with direct requests, not via involving the browser.

Federation

In the Gateway, federation can be simply a kind of token mediation: Instead of a simple token, this case uses a cryptographic token generated in policy, like a SAML token or JWT.

Other cases have the gateway send a request to an external token service. As an HTTP routing assertion is "Just An Assertion", a policy flow first validates credentials, then does an HTTP post, for instance, of a WS-Trust message to the SAML STS, receives the response message, parses the SAML token from the message and places that token into the WS-Security header of the SOAP message, and then in turn sends that combined message to the real back end service.

In the world of JSON, the flow is conceptually the same; replacing the WS Trust with whatever the JWT STS needs as a request, and JWT and structure of the message being manipulated as JSON.

In comparison to other products, our Federation is strictly claims based; we don't need a backing identity. We process the decision to issue a token with information from policy; it can be derived from identity, or it can simply be from the credential presented.

Policy Fragments & Encapsulated Assertions

Policy fragments are standard policy logic that has been designed to work with more than one service. Sometimes this is referred to as an "included policy" or as just an "include". In many ways, they should be thought of in similar ways to library routines. They have strictly defined inputs and outputs, and are meant to have limited side effects.

Policy Fragments, like any policy can have role based access control rules; this allows them to be authored by domain experts, and still used in a read-only way by other policy authors. This has the same benefits as in standard software developments: domain experts define common policy elements, like Authentication, Authorization, Audit, and policy authors simply pull into their policies the appropriate fragments. This reduces errors and simplifies testing, validation and governance.

There's a variant of a Fragment, called an "Encapsulated Assertion" that has roughly the same rules, and it also has an automatically generated User Interface that allows you to very strictly define your inputs and outputs. This can be a critical tool to create strong locally relevant best practices.

Auditing

Auditing is a critical question to ask when developing a policy. Often there are business concerns that require very complete auditing. Auditing is not free, and auditing every request has serious implications: Disk space and time are used in the default audit setup, and both of those are in limited supply on the appliance based deployment of the gateways, whether virtual or physical. Planning audit should be very early in a service deployment plan.
There are several core concepts that matter:

- Auditing of request and response content will use a great deal of disk space – a bad idea for high volume services. A simple audit without the request and response body is a lot less disk intensive.
- For correctness, the gateway message processor waits for audit to flush before returning a response to the client software, so heavy audit can add latency to an otherwise fast service.
- We can make audit be more asynchronous with the addition of queuing, but certain queues are also synchronous to disk by default.

Implementing a Policy Development Lifecycle

Introduction

As a preface, the focus of software development lifecycle (abbreviated as SDLC) management often really is just problem of configuration management. Good Configuration management is essentially being able to predict reliably what versions of what artifacts are included in a production build.

For a Policy Development Lifecycle (abbreviated as PDLC), the tooling has similar requirements to an SDLC, meaning that you need to be able to roll back a change, if needed, and it presumes a development phase, a testing phase, and a deployment phase. This is further complicated in the case of Gateway policy by the necessity of differences between environments: you can't reliably test a user acceptance environment gateway policy by attempting to connect to the development LDAP; you must be able to reliably connect to an LDAP that is configured like production.

Often that requires parallels to the standard SDLC method's we're all used to; in our case this means treating our configuration artifacts (policies and their dependencies) as source code, specifically XML. There is a specific phase in using our tooling that is concerned with resolving environmental dependencies.

Terminology

In general, as the SSG works with standard SOAP services, with REST services, JSON, and various kinds of networked XML and non-xml APIs, we refer to the core business value being deployed, the API the Gateway is presenting to the users as a "Service".

The specific components that a service depends on are things like Public Keys, Private Keys, LDAP configurations, JMS configurations, JDBC configurations, MQ Series configurations, etc. Those we'll refer to as "Environment Dependencies" in general, and by name when there are details needing to be discussed.

Also, since terminology varies on environment names, initial policy development environments, often hand in hand with software development will be referred to as "DEV", initial quality assurance as "QA", acceptance testing and/or performance testing as "UAT" and finally deployment in front of end user applications as "PROD".

Policies are Code and Deserve Similar Attention

As infrastructure components like the Layer7 Secure Span Gateway are deployed, it quickly becomes apparent that for enterprise deployment, the management of configuration artifacts in a repeatable and reliable way is critical.

Most organizations know how to deploy software to application servers, and this document seeks to show how to deploy policies to infrastructure.

We provide tools and interfaces for an organization to plan a lifecycle around for their environment. As every organization has its own way of deploying production environments, we expect parts of this process will already be in place, and so what is important is the effect of these processes, not the exact steps.

Best Practices

- Develop a policy and its environmental dependencies in a non-production environment
- Copy the complete version of these artifacts to a test environment with no shared infrastructure
  - Make the local changes needed to be compatible with the test environment.
  - Local changes should be environment specifics like IP addresses or host names of Application Servers or LDAP servers, and the like.
- Test the system to be sure your components are working.
- Copy the same components to the next step in the deployment pipeline.
- The original developer should not be responsible for moving artifacts to production, nor be responsible for testing.
  - Part of good development practice is to have repeatable creation of your systems.
  - If documentation isn't sufficient to do full deployment, fix the documentation, don't assign the original developer.
- Certain kinds of environment specific artifacts, like host names of back end servers, should use cluster properties instead of literals.
- Naming conventions for objects can ease resolution of environment dependencies.
- For predictability, configuration must be pushed as a unit, and the tested versions of each artifact be used.
- Local dependencies are worked out by moving from development to testing or user acceptance.

  Moving directly to production is risky, as testing the dependency discovery is critical.
- Do not share environment items like LDAP between DEV and Test/UAT. This will almost certainly cause a failure moving to production.

GUI-Based Policy Migration (Enterprise Service Manager and Policy Manager)

Our UI tools provide an easy to use method of migrating policies between environments. ESM is an optional separate product, and is capable of maintaining mapping between environments and provides ways to keep the details saved so later migrations are predictable in terms of artifacts.

The Policy Manager also enables migration via the import and export functions documented in the manuals, but does not preserve mapping between multiple environments.

Automated Policy Migration (Management API, GMC, CMT)

Some environments don't allow GUI tooling for managing production systems, for that, scripted build systems, continuous integration and complete hands free deployment are a must. In our product the command line capability is provided in several ways: There is an API for remote management (the management API), and a command line client for it, the Gateway Management Client (GMC).

CMT

There is an optional scripting system built to cover only the policy migration case called Command Line Policy Migration Toolkit (CMT).

For CMT, version control is critical. As locally preferred tooling (Subversion, ClearCase, Polytron Version Control, Git, etc.) is widely varied, this doesn't prescribe a specific tool, but rather expects three main capabilities:

- Can check in and check out based on a "tag" or "release name" that encompasses many files and many directories
- Can deal with XML as text and also check in binary files
- Can be run using command line tooling

Internally, the Gateway uses Object Identifiers (OIDs) and Global Object Identifiers (GOIDs) to identify various configuration items. When dealing with multiple environments and multiple staff members being involved in the process, using name based identifiers can be beneficial. The CMT tooling deals with resolving names to local objects.

Naming conventions are often used to identify the back end App Servers and so naming convention can often be used to good effect in identifying artifacts.

See Migrate Gateways in the CA API Gateway wiki for supplemental reading about policy migration specifically.

Policy Development Life Cycle Process

With a Policy Development Lifecycle that mirrors software development lifecycle at your organization, you can build a reliable process. We provide the tools to do so, and a set of capabilities that can be adapted to any environment. Planning is essential, but once the process is in place, it can and should be fully automated.

Conclusion

Our policy language reflects the diverse nature of the use cases we can support. Planning for environment dependencies, choosing good modularity and above all thinking of policy in the same way that you do source code is a good choice.