apiman - User Guide

Chapter 1. Introduction

1.1. What is API Management?

A popular trend in enterprise software development these days is to design client apps to be very decoupled and use APIs to connect them. This approach provides an excellent way to reuse functionality across various applications and business units. Another great benefit of API usage in enterprises is the ability to create those APIs using a variety of disparate technologies.

However, this approach also introduces its own pitfalls and disadvantages. Some of those disadvantages include things like:

- Difficulty discovering or sharing existing APIs
- Difficulty sharing common functionality across API implementations
- Tracking of API usage/consumption

API Management is a technology that addresses these and other issues by providing an API Manager to track APIs and configure governance policies, as well as an API Gateway that sits between the API and the client. This API Gateway is responsible for applying the policies configured during management.

Therefore an API management system tends to provide the following features:

- Centralized governance policy configuration
- Tracking of APIs and consumers of those APIs
- Easy sharing and discovery of APIs
- Leveraging common policy configuration across different APIs

1.2. Project Goals

The goals of the JBoss API management project are to provide an easy to use and powerful API Manager as well as a small, fast, low-overhead API Gateway to implement standard API management functionality.

1.3. Typical Use Cases

Some common API management use cases include:

1.3.1. Security

APIs will very often have a security requirement such that clients connecting to the API must authenticate in some fashion. Authentication can vary greatly both in the protocols used to authenticate and the identity source used for validation.

It can often be convenient to provide authentication at the API management layer to free up the back end API from having to do this work. This approach also has the side benefit of centralizing configuration of authentication for a wide array of disparate APIs.

Therefore the API management layer must provide authentication capabilities using a wide range of protocols including BASIC, digest, OAuth, etc.

1.3.2. Throttling/Rate Limiting

The API management layer is a convenient place to ensure throttling (also known as rate limiting) to your APIs. Throttling is a way to prevent individual clients from issuing too many requests to an API. Because all requests to an API go through the API Gateway it is an excellent place to do this throttling work.

1.3.3. Metering/Billing

There are a number of reasons why an API provider would be interested in the number of requests made to her API. The most common reason for public facing APIs is to implement billing based on usage per consumer. Metering is the feature that provides this API management capability.

A Crash Course in API Management with JBoss apiman

Chapter 2. Introduction

Welcome to the crash course in JBoss apiman! The goal of this crash course is to provide a short and easy to consume introduction to API Management with apiman. This crash course will explain how apiman works and will walk you through a hands-on session to demonstrate how to use it. This crash course also provides links to the extensive on-line apiman user documentation, demos and quickstarts, and other resources.

This crash course is organized into the following sections:

- How apiman Works - This section describes the data model apiman uses, and how it uses the elements created in that data model to perform its tasks.
- Policies: At the Core of API Management - In API Management, policies are where the action is. It’s the policies that define the rules that apiman uses to manage your APIs. Since policies are so important, this is the longest section of the crash course.

Each of the next four sections in the crash course describes how apiman supports the four primary groups of apiman users:

- Providing APIs - This section describes how API providers can use apiman to publish APIs and summarizes management options for those APIs.
- Consuming APIs - And this section describes how their counterparts, consumers of APIs, utilize apiman.
- Managing apiman - This section describes the major tasks that are performed by apiman administrators.
- Developing for apiman - And, this section describes how apiman supports the development of extensions and customizations.

Each of the last two sections in the crash course provides step-by-step descriptions of how to use apiman to:

- Install, configure and get apiman running for API providers and API consumers in 10 minutes.
- Utilize apiman’s REST interface to automate tasks.

Let’s get started!

Chapter 3. Section 1 - How apiman Works

The (4) primary groups of apiman users are:

- API Providers
- API Consumers
- API Management Administrators
- API Developers

Apiman fulfills these groups of users' needs by providing these subsystems:

- API Manager - The API Manager provides an easy way for API providers to use a web UI to define plans for their APIs, apply these plans across multiple APIs, and control role-based user access and API versioning. These plans can govern access to APIs and limits on the rate at which consumers can access APIs. The same UI enables API consumers to easily locate and access APIs. All features available in the web UI are also available via a REST interface, allowing full automation.
- API Gateway - The gateway applies the policies configured in the API Manager, enforcing them as runtime rules for each managed API request made. The way that the API Gateway works is that the consumer of the API accesses the API through a URL that designates the API Gateway as a proxy for the API. If the policies defined to govern access to the API permit that access, the API Gateway then proxies requests to the backend API implementation.
- An Extensible Plugin-Based Architecture - API Developers can create their own custom API management policies and install them into apiman. In addition, custom implementations of core apiman components can be created and used, without needing to rebuild the core system.

3.1. The apiman Data Model

Apiman uses a hierarchical data model that consists of the following primary elements:

- Organizations
- Plans
- APIs
- Client Apps
- Policies

- Policies - Policies are at the lowest level of the data model, but they are arguably the most important concept because they represent the unit of work done at runtime (when the API Gateway applies the policies to all API requests). Everything defined in the API Manager is there to enable apiman to apply policies to requests made to APIs. When a request to an API is made, apiman creates a chain of policies to be applied to that request. Apiman policy chains define a specific sequence order in which the policies defined in the API Manager UI are applied to API requests. You can think of a policy as a rule, or set of rules that are enforced by the API Gateway. There are multiple types of apiman policies. Some policies allow or block access to APIs based on the IP address of the client application, while others allow or restrict access to specific resources provided by an API, while still others enable you to control or “throttle” the rate at which requests made to an API.

- Plans - In apiman, a Plan is a set policies that together define the level of service that apiman provides for an API. Plans enable apiman users to define multiple different levels of service for their APIs. It’s common to define different plans for the same API, where the differences depend on configuration options. For example, an organization may offer both a “gold” and “silver” plan for the same API. The gold plan may be more expensive than the silver plan, but it may offer a higher level of API requests in a given (and configurable) time period.

- APIs - These represent real back-end APIs that are being “managed” by apiman. An API can either be Public (available to any invoker at a static endpoint) or Private (only invokable by Client Apps that are modeled within apiman). This is the primary entity that an API Provider is responsible for editing within the API Manager.

- Client Apps - A Client App represents a piece of software that needs to consume managed APIs that are offered through apiman. Each Client App can consume multiple APIs within apiman by creating a contract to the API through one of its Plans. This is the primary entity that an API Consumer is responsible for editing within the API Manager.

- Organizations - The Organization is at the top level of the apiman data model. An organization contains and manages all elements used by a company, university, group inside a company, etc. for API management with apiman. All Plans, APIs,and Client Apps are defined in an apiman organization. In this way, an organization acts as a container of other elements. Users must be associated with an organization before they can use apiman to manage elements within it. Apiman implements role-based access controls for users. The organization membership role assigned to a user defines the actions that a user can perform and the elements that a user can manage within the organization.

Tip

Key concept to remember. In the apiman data model, almost everything exists in the context of an organization.

Note that the apiman data model supports versioning for many of its data elements. Versioning enables you to retain and make use of multiple versions of data elements such as APIs and plans.

3.2. Apiman at Runtime - Proxying Requests

What happens is that when an API request is received by the API Gateway at runtime, the policy chain is applied in the order of client app, plan, and API. If no failures, such as a rate counter being exceeded, occur, the API Gateway sends the request to the API. In order to be accepted by the API Gateway, a request must include an API key. The API Gateway acts as a proxy for the API:

Next, when the API Gateway receives a response from the API's backend implementation, the policy chain is applied again, but this time in the reverse order. The API policies are applied first, then the plan policies, and finally the client app policies. If no failures occur, then the API response is sent back to the consumer of the API.

Before we move on, it's important that we're clear on some basic terminology. When we talk about an API that is *managed* by apiman (in other words, a *managed API*), we're referring to an an API where the apiman API Gateway is acting as a proxy. In order to able access a managed API, a client app must make of of an *API key* that is generated when the API is published to the API Gateway. The API key is embedded in the URL at which the managed API is published by the API Gateway. We'll see a working example of this later in the crash course.

Tip

Key concept to remember. From a client app's perspective, the only difference between accessing a managed API and another API is the format of the APIs' endpoint URL.

The sequence in which incoming API requests have policies applied is:

- First, at the client app level. In apiman, a client app is contracted to use one or more APIs.
- Second, at the plan level. In apiman, policies can be organized into groups called plans.
- Third, at the individual API level.

By applying the policy chain twice, both for the originating incoming request and the resulting response, apiman allows policy implementations two opportunities to provide management functionality during the lifecycle. The following diagram illustrates this two-way approach to applying policies:

Chapter 4. Section 2 - Policies, the Core of API Management

Policies are the most important element of API management. All the subsystems in apiman, from the Management API UI to the API Gateway, exist for one ultimate goal; to ensure that API governance is achieved by the application of policies to API requests. In apiman, policies are applied through a policy chain. Apiman is not only preconfigured with a rich set of policies that you can use, right out of the box, it also supports a mechanism that you can use to define your own custom policies.

Tip

Key concept to remember. API governance is achieved by the API Gateway applying policies to API requests.

Apiman has support for many policies, including (but not limited to):

Policy Categories	Policies
Security Policies	• BASIC Authentication - A username/password is required to access an API.
Limiting Policies	• Rate Limiting - Access to an API is limited by the number of requests in a defined time period (generally used to create a fine-grained limit).
Modification Policies	• URL Rewriting - Modify any URLs in the API response so that they direct users through the API Gateway rather than directly to the back-end API.
Other Policies	• Caching - Cache results from a backend API.

Let’s learn a little bit more about policies.

4.1. What’s in a Policy

An apiman policy consists of the following:

- Basic meta-data about the policy (name, description)
- JSON based configuration

- A Java class providing the implementation of the policy

Each policy supported by apiman performs a specific task, such as (but not limited to):

- Rate Limiting/Quotas
- Security
- Caching
- Transformation

Every API managed by apiman can be configured with zero or more policies. In addition, an API can be offered for consumption through several Plans, where each Plan can be configured with zero or more policies. Finally, a Client App can also be configured with a set of policies. Whenever the API Gateway receives a request for an API (optionally on behalf of a specific Client App), it creates a chain of policies from those configured at the three levels, and then applies that chain of policies to the request.

Most of the apiman policies work alone (e.g. caching), but some of them are used in conjunction with other policies. The next couple of sections will discuss two very common categories of policies, some of the policies found in those categories, and how they work together.

4.2. Security Policies - Authentication & Authorization

We'll start with the Authorization and Authentication policies. We'll review these policies together as the use of the Authorization type depends on the BASIC authentication type. Before we take a detailed look at the policies supported by apiman, it's important that we understand the differences between authentication and authorization:

- In authentication-based policies, access to an API is governed by the identity of the user
- In authorization-based policies, access to an API, or specific resources provided by an API is governed by the role(s) assigned to a user

In order to make use of an authorization policy, roles must be extracted during authentication. In other words, you cannot have authorization without authentication.

APIs often define security requirements to ensure that clients have to authenticate. By having apiman perform this authentication, backend APIs are freed from having to implement and perform this authentication. This also has the added benefit of centralizing the authentication for all your APIs.

In creating an Authentication policy, we define an Authentication Realm (think of this as an area to be protected, within which usernames and passwords exist) and an optional HTTP header. The HTTP header is used to optionally pass the authenticated user's principal to the back-end API through an HTTP header. This is useful if the back-end system needs to know the username of the user calling it (e.g. to perform a user-specific operation).

An apiman Authorization policy consists of a set of rules. The rules define the resources that can be accessed in terms of a regular expression and an HTTP verb (GET, PUT, etc.)

Through its authorization policies, apiman enables you to create fine-grained rules to govern access to your API's resources. For example, based on the user roles that you define, users assigned a "sales" role can access the sales related API resources, and users assigned a "marketing" role can access the marketing related API resources. Users assigned to to an "admin" role are able to access all the API's resources.

As we mentioned a moment ago, in order to make use of an authorization policy, roles must be extracted during authentication. Apiman can be configured to extract those roles from an available source; for instance, the *JSON Web Token* [http://jwt.io/] when using Keycloak, or JDBC/LDAP with the BASIC authentication policy in the API request. Remember, you cannot have authorization without authentication.

4.3. Limiting Policies - Rates and Quotas

Apiman provides (3) limiting policies:

- Rate Limiting - This policy type governs the number of times requests are made to an API within a specified time period. The requests can be filtered by user, application, or API and can set the level of granularity for the time period to second, minute, hour, day, month, or year. The intended use of this policy type is for fine grained processing (e.g., 10 requests per second).
- Quota - This policy type performs the same basic functionality as the Rate Limiting policy type., however, the intended use of this policy type is for less fine grained processing (e.g., 10,000 requests per month).
- Transfer Quota - In contrast to the other policies, Transfer Quota tracks the number of bytes transferred (either uploaded or downloaded) rather than the total number of requests made.

Each of these policies, if used singly, can be effective in throttling requests. Apiman, however, adds an additional layer of flexibility to your use of these policies by enabling you to use them in combinations. Let's look at a few examples.

Limiting the total number of API requests within a period of time is a straightforward task and can be configured in a quota policy. This policy, however, may not have the desired effect as the quota may be reached early in the defined time period. If this happens, the requests made to the API during the remainder of the (typically long) time period will be blocked by the policy. A better way to deal with a situation like this is to implement a more flexible approach where the monthly quota policy is combined with a fine grained rate limiting policy that will act as a throttle on the traffic.

To illustrate, there are about 2.5 million seconds in a month. If we want to set the API request quota for a month to 1/2 million, then we can also set a rate limit policy to a limit of 5 requests per second to ensure that API requests are throttled and the API can be accessed throughout the entire month.

Here's a visual view of a rate limiting policy based on a time period of one week. If we define a weekly quota, there is no guarantee that users will not consume that quota before the week is over. This will result in an API's requests being denied at the end of the week. In contrast, if we augment the weekly quota with a more fine grained policy, we can maintain the API's ability to respond to requests throughout the week:

Tip

Key concept to remember. Policies can be configured to work together in combinations.

The ability to throttle API requests based on API request counts and bytes transferred provides even greater flexibility in implementing policies. APIs that transfer larger amounts of data, but rely on fewer API requests can have that data transfer throttled on a per byte basis. For example, an API that is data intensive, will return a large amount of data in response to each API request. The API may only receive a request a few hundreds of times a day, but each request may result in several megabytes of data being transferred. Let's say that we want to limit the amount of data transferred to 6GB per hour. For this type of API, we could set a rate limiting policy to allow for one request per minute, and then augment that policy with a transfer quota policy of 100Mb per hour.

Before we move on, let's look at how we can combine multiple policies into a plan.

It's important to keep in mind that a plan can contain multiple policies. For our example, we'll create both a "gold" plan and a "silver" plan. In a real-world situation, gold and silver level plans might look something like this:

Gold plan	• A coarse grained Quota plan with a limit of 100,000 API requests per month, and
Silver plan	• A coarse grained Quota plan with a limit of 20,000 API requests per month, and

This diagram lets us visualize how the two policies will work together:

In this diagram, each filled in box represents one API request. The important thing to understand is how the policies work together to enable you to have flexible throttling of requests to your API:

- The fine grained rate limit is reset at the end of the time period defined for the rate limit policy
- And, the total number of API requests continue to be applied to the defined quota until the quota policy time limit is reached.

4.4. Other Policies

There are many other policies offered by apiman, each of them performing a specific task. And more policies are added with every release! Even more interesting, you can add your own custom policies using apiman's excellent plugin framework (more on that later). You can refer to the apiman User Guide for a full list of official policies, what each policy does, and how to configure it.

Chapter 5. Section 3 - Providing APIs

5.1. Publishing APIs

When an API is published to the API Gateway, the API is made available to the client apps that are the consumers of APIs. There are two different ways to publish an API:

Publishing an API as Public API - Public APIs can be directly accessed by any client, without providing an API Key. This allows you to distribute the URL that is used to access the API through through the API Gateway. The URL for a managed Public API takes this form:

```
http://gatewayhostname:port/apiman-gateway/{organizationId}/{API ID}/{API
 version}/
```

Public APIs are also very flexible in that they can be updated without being re-published. Unlike APIs published through Plans, Public APS can be accessed by a client app without requiring API consumers to agree to any terms and conditions related to a contract defined in a plan for the API. It is also important to note that when an API is Public, only the policies configured on the API itself will be applied by the API Gateway.

Publishing an API through Plans - In contrast to Public APIs, these APIs, once published, must be accessed by a Client App via its API key. In order to gain access to an API, the Client App must create a contract with an API through one of the API's configured Plans. Also unlike Public APIs, APIs that are published and accessed through its Plans, once published, cannot be changed. To make changes, new versions of these APIs must be created.

5.2. Security for APIs - Policy and Endpoint Security

One important aspect of all APIs that are managed by the API Gateway is the security that the API Gateway provides. Let's next take a look at the different types of security that are available.

The authentication policy type provides username/password security for clients as they access the managed API through the API Gateway, but it does not protect the API from unauthorized access attempts that bypass the Gateway completely. To make the API secure from unauthorized client applications, endpoint level security should also be configured.

The best way to start our discussion of the different, but complementary types of security that we'll examine in this article is with a diagram. The nodes involved are the client applications that will access our APIs, the apiman API Gateway, and the servers that host our APIs:

Let's work our way through the diagram from left to right and start by taking a look at Policy Level Security.

5.2.1. Policy Level Security

Policy level security, such as that provided by an Authentication policy, secures the left side of the diagram, that is the communication channel between the applications and the API Gateway. In this communication channel, the applications play the role of the client, and the API Gateway plays the role of the server.

We also want to secure the right side of the diagram, where the API Gateway plays the role of a client, and the APIs play the role of the servers.

Note

It's worth noting that while policy security protects the managed API, it does nothing to protect the unmanaged API as this API can be reached directly, without going through the API Gateway. This is illustrated by the red line in the diagram. So, while access to the managed API through the apiman API Gateway is secure, policy security does not secure the unmanaged API endpoint.

5.2.2. Endpoint Level Security

In contrast to policy level security, with endpoint security we are securing the right side of the diagram. Current apiman supports two endpoint security options:

- BASIC Authentication
- MTLS (two-way SSL)

A recent post by Marc Savy to the apiman blog *described how to configure Mutually Authenticated TLS* [http://www.apiman.io/blog/gateway/security/mutual-auth/ssl/mtls/2015/06/16/mtls-mutual-auth.html] (Transport Layer Security) between the API Gateway and the managed APIs. With Mutual TLS, bi-direction authentication is configured so that the identities of both the client and server are verified before a connection can be made.

We should also note that, unlike policy security, endpoint security also secures the APIs from attempts to bypass the API Gateway. With Mutual TLS, a two-way trust pattern is created. The API Gateway trusts the APIs and the APIs trust the API Gateway. The APIs, however, do not trust the client applications. As is shown by the large "X" character that indicates that an application cannot bypass the API Gateway and access the APIs directly.

One last point that is important to remember is that the endpoint level of security applies to all requests made to the APIs, regardless of the policies configured.

Tip

Key concept to remember. Policy security alone does not secure an API's unmanaged endpoints.

To summarize, the differences between policy level security and endpoint level security are:

Policy Level Security	End Point Level Security
Secures communications between the applications (clients) and API Gateway (server)	Secures communications between the API Gateway (client) and APIs (servers)
Configured in an API Gateway policy	Configured for the API Gateway as a whole in apiman.properties and with key/certificates infrastructure
Applied by a policy at runtime	Enabled for all API requests, regardless of the policies configured for an API
Does not secure the unmanaged API from access by unauthorized clients	Secures the unmanaged API endpoints from access by unauthorized clients

5.3. API Metrics

After you've created and published your APIs, you will want to be able to keep track of the level of use they are receiving. To fulfill this need, apiman provides you with API metrics. The metrics track the following information:

- Request start and end times
- API start and end times (i.e. just the part of the request taken up by the back end API)
- Resource path
- Response type (success, failure, error)
- API info (org id, id, version)
- Client App info (org id, id, version)
- Bytes uploaded/downloaded

API Metrics can be accessed in the Management UI and through the REST API. The metrics are displayed visually in the Management UI, for example:

Chapter 6. Section 4 - Consuming APIs

6.1. Invoking Managed APIs

From a client app's perspective, the only difference between accessing a managed API and another API is the URL of the API's endpoint. As we mentioned earlier in this crash course, a managed apiman endpoint takes this form:

```
http://gatewayhostname:port/apiman-gateway/{organizationId}/{API ID}/{API
 version}/
```

In addition, if the API is not Public, then the managed API endpoint must include a Client App's API Key, either as a query parameter in the URL or as an HTTP header. For example:

```
http://localhost:8080/apiman-gateway/ACMEServices/echo/1.0?apikey=c374c202-
d4b3-444206e3d
```

Tip

Don't panic! You don't have to memorize the endpoint string. As we'll see in a bit, the endpoint string is provided to you by apiman.

6.2. Managing Client Applications and Contracts

Public APIs can be consumed by any client. APIs that are not public can only be consumed by client applications that exist in an apiman organization and are registered with apiman.

When you create a client app in the Management UI, you are able to perform a search through all published APIs to locate the API that you want the client app to consume. The Management UI allows you to select from all published versions of an API, and from all the defined plans for an API. (Remember that, in this context, a plan is a set of policies that the API enforces.) Note that client apps can have configured policies, the same manner as plans and APIs.

Once you find an API that you want your client app to consume, and after you select the version of the API and the plan that you want to govern how your client app will consume the API, you use the Management UI to create an API contract. The contract contains the "Terms and Conditions" defined by the API provider that govern your client app's use of the API.

Your client app can consume one or more API. Once your client app has created contracts with all of the APIs it needs to consume, it must be registered with the Gateway. This enables the Gateway to know which contracts are valid and how to create the full policy chain it will apply to the request.

Chapter 7. Section 5 - Managing apiman

7.1. Users and Roles

In the apiman data model, all data elements exist in the context of the organization. The same holds true for user memberships as users can be members of multiple organizations. Permissions in apiman are role based. The actions that a user is able to perform are dependent on the roles to which the user is assigned when she is added as a member of an organization.

Let's start by looking at the roles that are preconfigured in apiman.

7.1.1. Understanding OOTB apiman user roles

In apiman, each role defines a set of permissions granted by that role. When a user is made a member of an organization, that user must be assigned to a role. A role definition consists of a name and description, and, most importantly, a set of permissions that govern the user's ability to view, edit, and administer the organization itself, as well as the organization's plans, APIs, and applications.

Role Definitions are managed in the Roles section of the apiman System Administration section of the Management UI.

Apiman is preconfigured with the following roles:

- Organization Owner
- API Developer
- Client App Developer

These role names are self-explanatory. For example, a user assigned the Client App Developer role is able to manage the organization's client apps but is blocked from managing its APIs or plans.

The full set of permissions provided in apiman by these preconfigured roles are:

Preconfigured Role	Who Should be Assigned this Role	Permissions Granted by this Role
Client App Developer	Users responsible for creating and managing client apps.	Client App View
Organization Owner	Automatically granted to the user who creates an Organization. Can be granted to other users by an existing Organization Owner.	All permissions

Preconfigured Role	Who Should be Assigned this Role	Permissions Granted by this Role
API Developer	Users responsible for creating and managing APIs.	Plan View

Organization owners can assign roles to users through the *Manage Members* form in the apiman Management UI (found off the *Members* tab for an Organization). Each user must be assigned at least one role, but users can also be assigned multiple roles.

While apiman admin users can also modify the permissions as defined for these preconfigured roles, it is also very easy to create new custom roles.

7.1.2. Creating a New User Role/Defining the Role Permissions

Custom roles give you the ability to exercise fine-grained control over the set of permissions granted to users.

Let's look at an example of a custom role. Imagine a situation where you have API developer users and client app developer users. These sets of users can rely on apiman's preconfigured roles. Let's also imagine that you have a third set of user. You want these users to have read access to APIs and applications so that they can participate in a review/approval process. However, you do not want to give these users write access. You can create a view-only (read-only) role these users by configuring your custom Role Definition to only grant the Client App View and API View permissions.

7.2. Managing Policies and Plugins

Apiman is preconfigured with a core set of policies, but also supports adding more policies by installing one or more plugin. There are a number of official apiman plugins which will enable additional policies to be configured. Some examples of the official apiman plugin policies include (but are not limited to):

- CORS - This plugin implements CORS (Cross-origin resource sharing): A method of controlling access to resources outside of an originating domain.
- HTTP Security - Provides a policy which allows security-related HTTP headers to be set, which can help mitigate a range of common security vulnerabilities.
- JSONP - A plugin that contributes a policy that turns a standard RESTful endpoint into a JSONP compatible endpoint.
- Keycloak OAuth - This plugin offers an OAuth2 policy which leverages the Keycloak authentication platform as the identity and access provider.
- Log Headers - Offers a simple policy that allows request headers to be added or stripped from the HTTP request (outgoing) or HTTP response (incoming).

These optional plugins are accessed in the administrative page in the apiman Management UI. You can install these policies as needed, and then uninstall them when they are no longer needed.

There are a couple of caveats to keep in mind when you uninstall a policy plugin:

- First, uninstalling the plugin removes it from the apiman Management UI, but it still remains in use for all APIs in which it was previously configured.
- Second, if you want to completely remove the plugin from all APIs in which it was previously configured, you must manually click on each API, Plan, and Client App that uses the policy and remove it. Apiman does not include a single “kill” button to automatically remove all references to a policy.

In addition to enabling you to create and install your own custom policies, apiman also provides a mechanism to upgrade to new versions of those policies. This is an especially useful feature as, over time, a policy may be upgraded to include bug fixes or new features.

7.3. Managing Gateways

When you install apiman, it’s configured with one API Gateway. Apiman, however, enables you to configure and use multiple API Gateways simultaneously. There are several reasons why you might want to configure multiple API Gateways:

- It’s a good practice to maintain separate test and production environments for apiman. A test environment provides you with a safe place to experiment with the design of plans and custom policies without causing any interruption in service for APIs that are use for mission-critical production environments.
- If some APIs are used more heavily than others, you might want to group these APIs and configure an API Gateway for them on higher performance servers, or base these APIs on API Gateways located in geographic locations closer to their highest use Client apps.

Note that typically you will want to set up a single Gateway which is actually backed by multiple nodes/instances. Each instance (e.g. running on WildFly) should be configured to use the same backing storage (e.g. Elasticsearch or JDBC). This configuration results in a single “logical” gateway in apiman - so only one (1) gateway needs to be configured in the UI - when an API is published to one of the nodes, it will be available to them all.

7.4. Apiman REST API

It’s inevitable that, after you work with a product’s UI for a while that you encounter tasks that are better suited to a scripting or batch interface. For example, if you have to perform a similar task for a large number of related data items, the time that it can require to perform these tasks through an interactive UI can be prohibitive. Also, it’s easy for repetitive tasks to become error prone as you can lose focus, even if you are working in a well designed and easy to use interface such as apiman.

One solution to this problem is to augment the UI with a command line or scripting interface. This can lead to a whole separate set of issues if the new interface is built on a different set of underlying routines than the UI. A better approach is to allow access to the same routines in which the UI is constructed. This approach removes any duplication, and also enables you to replicate manual UI based tasks with automated or scripted tools.

Apiman follows this second approach with its REST interface. All of the functionality provided by apiman in its Management UI are directly supported in the API Manager REST API In fact, the UI simply makes calls to the REST layer in order to get data or make changes.

Tip

Key concept to remember. You can use the REST interface to automate any task that is performed in the UI.

The documentation for the apiman REST API is available (for free, of course), here: <u>http://www.apiman.io/latest/api-manager-restdocs.html</u>

Chapter 8. Section 6 - Getting Up and Running with apiman in 10 minutes

In this section, we'll also take a very hands-on look at apiman. In about 10 minutes, we'll get apiman installed and running, define an API policy, create and publish an API, register an application, and watch apiman enforce that policy.

Let's start by installing the prerequisite software packages that we will need.

8.1. Prerequisite Software Required

Like all JBoss middleware projects, you can run apiman on any operating system that supports Java software development. We don't need very much in the way of prerequisite software to run apiman out of the box. (Note that there really isn't a physical box as you can just download everything.)

What you will need to install to run apiman and follow all the steps in this chapter are: * Java - apiman can run Java version 1.7 or newer. You will want to install the full Java JDK. You can use either OpenJDK or Oracle's JDK. * Apache Maven - While you do not need the maven build tool to run apiman, we will use it to build an example API. You should download and install maven version 3.3 or newer.

We don't need very much to run apiman out of the box. Before we install apiman, you'll have to have Java (version 1.7 or newer, in this section we'll use Java 1.8) installed on your system. You'll also need to install git and maven installed to be able to build the example API that we'll use.

After you install the prerequisite software, the next thing we have to do is to get ourselves a copy of JBoss apiman.

8.2. Getting the Bits - Downloading apiman

To download apiman, open a browser and navigate to
<u>http://http://www.apiman.io</u>

The phrase "running an apiman server" is a bit misleading, as apiman itself is not a server. apiman is distributed in multiple forms. We'll examine and use each of these forms in this book:

- apiman WildFly Overlay - In this distribution, apiman is packaged in a zip file that is installed over a JBoss WildFly (

<u>http://wildfly.org/</u>
) server.

- Docker - In this distribution, apiman is packaged as Docker (
<u>https://www.docker.com/</u>
) images.

We'll keep things simple in this chapter and use the apiman WildFly Overlay distribution. (You can also download apiman packaged as a Docker image.) If you navigate to the "downloads" page, you'll see:

Let's take a look at the contents of the WildFly Overlay. There are three main directories in the WildFly Overlay:

The apiman directory - This directory contains configuration data specific to apiman such as the DDL (Data Description Language) files that define database schemas used by apiman, JSON files that define policy and security settings, and a quickstart example program that we will use as an example API. The apiman directory is a new directory that is created when you unzip the WildFly Overlay file. The top level directories in the apiman directory look like this:

```
### apiman
#   ### data
#   #   ### all-policyDefs.json
#   #   ### apiman-realm.json
#   ### ddls
#   #   ### apiman_mysql5.ddl
#   #   ### apiman_postgresql9.ddl
#   ### quickstarts
#   #   ### echo-service
#   #   ### LICENSE
#   #   ### pom.xml
#   #   ### README.md
#   ### sample-configs
#       ### apiman-ds_mysql.xml
#       ### apiman-ds_postgresql.xml
```

The modules directory - This directory contains configuration files, including keycloak (URL) configuration files that are added to the WildFly server for apiman. These files are added to the WildFly "standalone" server configuration . The top levels in this directory look like this:

```
### modules
#   ### system
#       ### layers
### standalone
 ### configuration
 #   ### apiman.jks
```

```
#   ### apiman.properties
#   ### keycloak-server.json
#   ### providers
#   ### standalone-apiman.xml
#   ### standalone-keycloak.xml
#   ### themes
### data
#   ### es
#   ### h2
#   ### keycloak.h2.db
```

The deployments directory - This directory contains the apiman API Gateway, back end APIs, and apiman Management UI, packaged as .war files. By unzipping the WildFly Overlay file, these .war files are deployed to the WildFly server. The top levels in this directory look like this:

```
### deployments
     ### apiman-ds.xml
     ### apiman-es.war
     ### apiman-gateway-api.war
     ### apiman-gateway.war
     ### apimanui.war
     ### apiman.war
```

Make a mental note of these apiman deployment files. We'll see them again in a few minutes.

The apiman download page is here:
<u>http://www.apiman.io/latest/download.html</u>

The steps you follow are:

- Download and Unzip the WildFly Server - Download
  <u>http://download.jboss.org/wildfly/10.0.0.Final/wildfly-10.0.0.Final.zip</u>
  and unzip the file into the directory in which you want to run the sever.
- Download and Unzip the apiman WildFly overlay .zip file - Download the apiman WildFly overlay zip file into the directory that was created when you unzipped the WildFly download. The apiman WildFly overlay zip file is available here:
  <u>http://downloads.jboss.org/apiman/1.2.3.Final/apiman-distro-wildfly10-1.2.3.Final-overlay.zip</u>
  After the file is downloaded, unzip it directly over the directory into which you unzipped the WildFly download. This will install apiman into the WildFly server.

Installing apiman on an WildFly Server

The commands that you will execute to install the server will look something like this:

```
mkdir ~/apiman-1.2.3.Final
cd ~/apiman-1.2.3.Final
```

```
curl http://download.jboss.org/wildfly/10.0.0.Final/wildfly-10.0.0.Final.zip
 -o wildfly-10.0.0.Final.zip
curl http://downloads.jboss.org/apiman/1.2.3.Final/apiman-distro-
wildfly10-1.2.3.Final-overlay.zip -o apiman-distro-wildfly10-1.2.3.Final-
overlay.zip
unzip wildfly-10.0.0.Final.zip
unzip -o apiman-distro-wildfly10-1.2.3.Final-overlay.zip -d
 wildfly-10.0.0.Final
```

Before we move on, we have one server administration task to perform. We have to create a server user, so that we can log onto the server administrative console. This is necessary as WildFly does not come pre-installed with any users.

To create a new server user, navigate to this directory:

```
cd apiman-1.2.3.Final/wildfly-10.0.0.Final/bin
```

And execute this script:

```
./add-user.sh
```

When you are prompted for the type of user to create, select Management User:

What type of user do you wish to add? a) Management User (mgmt-users.properties) b) Application User (application-users.properties) (a):

After you define a username and password, for the remainder of the prompts, you can safely take the default values, or select “yes” to complete the creation of a user account.

(Details on the administration of a WildFly server, including user management, are out of scope for this book as our focus is apiman. If you are interested in learning more about WildFly server administration, refer to the WildFly Server Administration Guide here:
<u>https://docs.jboss.org/author/display/WFLY10/Admin+Guide</u>

Running the WildFly Server

To start the WildFly server, you navigate back to the directory into which you installed the server, execute these commands - note that in this context, “standalone” refers to a standalone (i.e., non-clustered) WildFly server. You can learn more about WildFly server configuration options in the WildFly Server Administration Guide:

```
cd apiman-1.2.3.Final/wildfly-10.0.0.Final
./bin/standalone.sh -c standalone-apiman.xml
```

When the server starts, it will write logging messages (a lot of messages!) to the screen. The server will also create a server log file with these messages. When you see some messages that look like this, you’ll know that the server is up and running with apiman installed:

```
"apiman-gateway.war")23:28:49,091 INFO  [org.jboss.as] (Controller
 Boot Thread) WFLYSRV0060: Http management interface listening on
 http://127.0.0.1:9990/management
23:28:49,091 INFO  [org.jboss.as] (Controller Boot Thread) WFLYSRV0051:
 Admin console listening on http://127.0.0.1:9990
23:28:49,091 INFO  [org.jboss.as] (Controller Boot Thread) WFLYSRV0025:
 WildFly Full 10.0.0.Final (WildFly Core 2.0.10.Final) started in 11891ms -
 Started 1131 of 1543 services (616 services are lazy, passive or on-demand)
```

Testing the Installation

Congratulations! Your WildFly server is up and running with apiman installed!

Or is it?

Let's take a quick look at how we can be sure that the server installation was correct. To do this, we'll look in two places.

First, we'll look at the WildFly Server Administrative Console.

Remember the user account that we created? We'll use it now. To access the WildFly Server Administrative Console, open up a browser, and navigate to:
<u>http://localhost:8080</u>

This page will be displayed:

When you select the Administration Console selection, you will be prompted for the username and password:

Enter the username and password for the user that you defined (for this example, we used the very unimaginative and insecure username "admin") and you will brought to the WildFly Server Administration Console:

If you then select the "Deployments" tab at the top of the page, you'll see the applications deployed to the server. This is where you should see the apiman deployments for the APIs, Gateway, and Management UI:

If you don't see the apiman deployments, don't panic, but something went wrong with the installation. The most common reason for the apiman deployments to be missing is that you unzipped the apiman overlay .zip file into a different directory from the WildFly server. Remember, that the

reason that the overlay file is named “overlay” is that it must be unzipped over an installed WildFly server. You can confirm that this is what happened by looking in the WildFly server’s deployment directory here: wildfly-10.0.0.Final/standalone/deployments

If you look in this directory, you should see these files (the presence of files with the “.deployed” suffix indicates that the corresponding file was deployed successfully):

```
apiman-ds.xml
apiman-ds.xml.deployed
apiman-es.war
apiman-es.war.deployed
apiman-gateway-api.war
apiman-gateway-api.war.deployed
apiman-gateway.war
apiman-gateway.war.deployed
apimanui.war
apimanui.war.deployed
apiman.war
apiman.war.deployed
```

So, if you don’t see the apiman deployments, stop the server and start the installation over. Be careful to unzip the apiman overlay file directly over the directory created when you unzipped the WildFly server .zip file.

The second place we’ll look for evidence that the installation was successful is the WildFly server’s server.log file.

The WildFly server’s server.log file is created when the server is started. All the information that is displayed on the screen when you started the server is also written to the log file. (The level of detail written to the console and the log file is configurable. You can read about configuring WildFly logging here: *https://docs.jboss.org/author/display/WFLY10/Admin+Guide*)

You can find the WildFly server file here: wildfly-10.0.0.Final/standalone/log/server.log

The WildFly server log file can be quite large as the server will append more logging statements to it over time. While you can certainly read the entire file anytime you want, we’ll focus on some highlights related to ensuring that the server started cleanly. An obvious first step is to search the file for logging statements written at the ERROR level. If the file does not contain any errors, you can look for statements that look like this to confirm that the server started cleanly:

```
23:28:48,978 INFO  [org.wildfly.extension.undertow] (ServerService Thread
 Pool -- 71) WFLYUT0021: Registered web context: /apiman-es
23:28:49,000 INFO  [org.jboss.as.server] (ServerService Thread Pool -- 36)
 WFLYSRV0010: Deployed "apiman-gateway-api.war" (runtime-name : "apiman-
gateway-api.war")
23:28:48,999 INFO  [org.jboss.as.server] (ServerService Thread Pool -- 60)
 WFLYSRV0010: Deployed "keycloak-server.war" (runtime-name : "keycloak-
server.war")
```

```
23:28:49,000 INFO  [org.jboss.as.server] (ServerService Thread Pool -- 36)
 WFLYSRV0010: Deployed "apiman.war" (runtime-name : "apiman.war")
23:28:49,000 INFO  [org.jboss.as.server] (ServerService Thread Pool -- 36)
 WFLYSRV0010: Deployed "apiman-es.war" (runtime-name : "apiman-es.war")
23:28:49,001 INFO  [org.jboss.as.server] (ServerService Thread Pool -- 36)
 WFLYSRV0010: Deployed "apiman-ds.xml" (runtime-name : "apiman-ds.xml")
23:28:49,001 INFO  [org.jboss.as.server] (ServerService Thread Pool -- 36)
 WFLYSRV0010: Deployed "apimanui.war" (runtime-name : "apimanui.war")
23:28:49,001 INFO  [org.jboss.as.server] (ServerService Thread Pool -- 36)
 WFLYSRV0010: Deployed "services.war" (runtime-name : "services.war")
23:28:49,001 INFO  [org.jboss.as.server] (ServerService Thread Pool --
 36) WFLYSRV0010: Deployed "authtest-ds.xml" (runtime-name : "authtest-
ds.xml")23:28:49,001 INFO  [org.jboss.as.server] (ServerService Thread Pool
 -- 36) WFLYSRV0010: Deployed "apiman-gateway.war" (runtime-name :
```

That's right, it's the same apiman deployment files. If you see statements like these, and there are no ERROR statements, then you should be able to safely access the WildFly Administration console.

There's just more point we should cover before moving on. While the server may be up and running, it's not really configured for production use. As a convenience, when you install apiman, it is preconfigured with a default administrator account. The username for this account is "admin" and the password is "admin123!" - not exactly a mission critical level of security! If this were a production server, the first thing that we'd do is to change the default apiman admin username and password. apiman is configured by default to use JBoss KeyCloak (
<u>http://keycloak.jboss.org/</u>
) for password security. Also, the default database used by apiman to store contract and API information is the H2 in-memory database. For a production server, you'd want to reconfigure this to use a production database. We'll cover apiman server security and production configuration settings in later chapters.

The Echo API "Quickstart"

The source code for the example service is contained in a git repo (*http://git-scm.com*) hosted at github (*https://github.com/apiman*). To download a copy of the example service, navigate to the directory in which you want to build the service and execute this git command:

git clone
<u>*git@github.com* [mailto:git@github.com]</u>
:apiman/apiman-quickstarts.git

As the source code is downloading, you'll see output that looks like this:

```
git clone git@github.com:apiman/apiman-quickstarts.git
Initialized empty Git repository in apiman-quickstarts/.git/
remote: Counting objects: 104, done.
remote: Total 104 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (104/104), 18.16 KiB, done.
```

```
Resolving deltas: 100% (40/40), done.
```

The source code for the example API is provided in the wildfly-10.0.0.Final/apiman/quickstarts directory. (In JBoss software, the term “quickstart” refers to an example program.)

The echo-API quickstart includes these files:

And, after the download is complete, you’ll see a populated directory tree that looks like this:

```
### apiman-quickstarts
 ### echo-service
 #   ### pom.xml
 #   ### README.md
 #   ### src
 #    ### main
 #        ### java
 #        #   ### io
 #        #    ### apiman
 #        #        ### quickstarts
 #        #            ### echo
 #        #                ### EchoResponse.java
 #        #                ### EchoServlet.java
 #        ### webapp
 #            ### WEB-INF
 #                ### jboss-web.xml
 #                ### web.xml
 ### LICENSE
 ### pom.xml
 ### README.md
 ### release.sh
 ### src
  ### main
      ### assembly
          ### dist.xml
```

As we mentioned earlier, the example API is very simple. The only action that the API performs is to echo back in responses the meta data in the REST (*http://en.wikipedia.org/wiki/Representational_state_transfer*) requests that it receives.

Maven is used to build the API. To build the API into a deployable .war file, navigate to the directory into which you downloaded the API example:

```
cd apiman-quickstarts/echo-service
```

And then execute this maven command:

```
mvn package
```

As the API is being built into a .war file, you'll see output that looks like this:

```
[INFO] Scanning for projects...
[INFO]
[INFO]
 ------------------------------------------------------------------------
[INFO] Building apiman-quickstarts-echo-service 1.2.4-SNAPSHOT
[INFO]
 ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-resources-plugin:2.7:resources (default-resources) @
 apiman-quickstarts-echo-service ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory  local/redhat_git/apiman-
quickstarts/echo-service/src/main/resources
[INFO]
[INFO] --- maven-compiler-plugin:3.2:compile (default-compile) @ apiman-
quickstarts-echo-service ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 2 source files to  local/redhat_git/apiman-quickstarts/
echo-service/target/classes
[INFO]
[INFO] --- maven-resources-plugin:2.7:testResources (default-testResources)
 @ apiman-quickstarts-echo-service ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory  local/redhat_git/apiman-
quickstarts/echo-service/src/test/resources
[INFO]
[INFO] --- maven-compiler-plugin:3.2:testCompile (default-testCompile) @
 apiman-quickstarts-echo-service ---
[INFO] No sources to compile
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ apiman-
quickstarts-echo-service ---
[INFO] No tests to run.
[INFO]
[INFO] --- maven-war-plugin:2.5:war (default-war) @ apiman-quickstarts-echo-
service ---
[INFO] Packaging webapp
[INFO] Assembling webapp [apiman-quickstarts-echo-service] in [ local/
redhat_git/apiman-quickstarts/echo-service/target/apiman-quickstarts-echo-
service-1.2.4-SNAPSHOT]
[INFO] Processing war project
[INFO] Copying webapp resources [ local/redhat_git/apiman-quickstarts/echo-
service/src/main/webapp]
[INFO] Webapp assembled in [37 msecs]
[INFO] Building war:  local/redhat_git/apiman-quickstarts/echo-service/
target/apiman-quickstarts-echo-service-1.2.4-SNAPSHOT.war
[INFO]
```

```
[INFO] --- maven-source-plugin:2.4:jar-no-fork (attach-sources) @ apiman-
quickstarts-echo-service ---
[INFO] Building jar:  local/redhat_git/apiman-quickstarts/echo-service/
target/apiman-quickstarts-echo-service-1.2.4-SNAPSHOT-sources.jar
[INFO]
[INFO] --- maven-javadoc-plugin:2.10.1:jar (attach-javadocs) @ apiman-
quickstarts-echo-service ---
[INFO]
Loading source files for package io.apiman.quickstarts.echo...
[INFO] Building jar:  local/redhat_git/apiman-quickstarts/echo-service/
target/apiman-quickstarts-echo-service-1.2.4-SNAPSHOT-javadoc.jar
[INFO]
 ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO]
 ------------------------------------------------------------------------
[INFO] Total time: 3.061 s
[INFO] Finished at: 2016-04-16T22:13:10-04:00
[INFO] Final Memory: 26M/307M
[INFO]
 ------------------------------------------------------------------------
```

If you look closely, near the end of the output, you'll see the location of the .war file:

```
local/redhat_git/apiman-quickstarts/echo-service/target/apiman-quickstarts-
echo-service-1.2.4-SNAPSHOT.war
```

To deploy the API, we can copy the .war file to our WildFly server's deployments directory. After you copy the API's .war file to the deployments directory, you'll see output like this generated by the WildFly server:

```
22:33:59,794 INFO  [org.jboss.as.repository] (DeploymentScanner-threads
 - 1) WFLYDR0001: Content added at location local/redhat_git/apiman/
tools/server-all/target/wildfly-10.0.0.Final/standalone/data/content/31/
f9a163bd92c51daf54f70d09bff518c2aeef7e/content
22:33:59,797 INFO  [org.jboss.as.server.deployment] (MSC service thread 1-6)
 WFLYSRV0027: Starting deployment of "apiman-quickstarts-echo-service-1.2.4-
SNAPSHOT.war" (runtime-name: "apiman-quickstarts-echo-service-1.2.4-
SNAPSHOT.war")
22:33:59,907 INFO  [org.wildfly.extension.undertow] (ServerService Thread
 Pool -- 76) WFLYUT0021: Registered web context: /apiman-echo
22:33:59,960 INFO  [org.jboss.as.server] (DeploymentScanner-threads
 - 1) WFLYSRV0010: Deployed "apiman-quickstarts-echo-service-1.2.4-
SNAPSHOT.war" (runtime-name : "apiman-quickstarts-echo-service-1.2.4-
SNAPSHOT.war")
```

Make special note of this line of output:

```
22:33:59,907 INFO  [org.wildfly.extension.undertow] (ServerService Thread
 Pool -- 76) WFLYUT0021: Registered web context: /apiman-echo
```

This output indicates that the URL of the deployed example API is:

<u>http://localhost:8080/apiman-echo</u>

Remember, however, that this is the URL of the deployed example API if we access it directly. We'll refer to this as the "unmanaged API" as we are able to connect to the API directly, without going through the API Gateway. The URL to access the API through the API Gateway ("the managed API") at runtime will be different.

Now that our example API is installed, it's time to install and configure our client to access the server.

Accessing the Example API Through a Client

There are a lot of options available when it comes to what we can use for a client to access our API. We'll keep the client simple so that we can keep our focus on apiman and simply use a browser as the client. If you enter the API's URL into a browser, an HTTP GET command will be executed. The response will look like this:

```
{
  "method" : "GET",
  "resource" : "/apiman-echo",
  "uri" : "/apiman-echo",
  "headers" : {
    "Cookie" : "s_fid=722D028B20E49214-13EAE1456E752098;
 __utma=111872281.807845787.1452188093.1460777731.1460777731.4;
 __utmz=111872281.1452188093.1.1.utmcsr=(direct)|
utmccn=(direct)|utmcmd=(none); _ga=GA1.1.807845787.1452188093;
 __qca=P0-404983419-1452188093717; __utmc=111872281",
    "Accept" : "text/html,application/xhtml+xml,application/xml;q=0.9,*/
*;q=0.8",
    "Connection" : "keep-alive",
    "User-Agent" : "Mozilla/5.0 (X11; Linux x86_64; rv:38.0) Gecko/20100101
 Firefox/38.0",
    "Host" : "localhost:8080",
    "Accept-Language" : "en-US,en;q=0.5",
    "Accept-Encoding" : "gzip, deflate",
    "DNT" : "1"
  },
  "bodyLength" : null,
  "bodySha1" : null
}
```

Now that our example API is built, deployed and running, it's time to create the organizations for the API provider and the API consumer. The differences between the requirements of the two organizations will be evident in their apiman configuration properties.

OK, the preliminaries are over! Now, it’s time to go into the apiman Management UI and create the apiman data elements for our demonstration.

Creating Users for the API Provider and Consumer Organizations

Before we create the organizations, we have to create a user for each organization. We’ll start by creating the API provider user. To do this, logout from the admin account in the API Manager UI. The login dialog will then be displayed.

Select the "New user/Register" Option and register the API provider user:

Then, logout and repeat the process to register a new application developer user too:

Now that the new users are registered we can create the organizations.

Creating the API Provider Organization

To create the API provider organization, log back into the API Manager UI as the apiprov user and select “Create a new Organization”:

Select a name and description for the organization, and press “Create Organization”:

And, here’s our organization:

Note that in a production environment, users would request membership in an organization. The approval process for accepting new members into an organization would follow the organization’s workflow, but this would be handled outside of the API Manager API. For the purposes of our demonstration, we’ll keep things simple.

Configuring the API, its Policies, and Plans

To configure the API, we’ll first create a plan to contain the policies that we want applied by the API Gateway at runtime when requests to the API are made. To create a new plan, select the “Plans” tab. We’ll create a “gold” plan:

Once the plan is created, we will add policies to it:

apiman provides several OOTB policies/plans. Since we want to be able to demonstrate a policy being applied, we'll select a Rate Limiting Policy, and set its limit to a very low level. If our API receives more than 10 requests in a day/month, the policy should block all subsequent requests. So much for a "gold" level of API!

After we create the policy and add it to the plan, we have to lock the plan:

And, here is the finished, and locked plan:

At this point, additional plans can be defined for the API. We'll also create a "silver" plan, that will offer a lower level of API (i.e., a request rate limit lower than 10 per day/month) than the gold plan. Since the process to create this silver plan is identical to that of the gold plan, we'll skip the screenshots.

Now that the two plans are complete and locked, it's time to define the API.

We'll give the API an appropriate name, so that providers and consumers alike will be able to run a query in the API Manager to find it.

After the API is defined, we have to define its implementation. In the context of the API Manager, the API Endpoint is the API's direct URL. Remember that the API Gateway will act as a proxy for the API, so it must know the API's actual URL. In the case of our example API, the URL is: <u>http://localhost:8080/apiman-echo</u>

The plans tab shows which plans are available to be applied to the API:

Let's make our API more secure by adding an authentication policy that will require users to login before they can access the API. Select the Policies tab, and then define a simple authentication policy. Remember the user name and password that you define here as we'll need them later on when send requests to the API.

After the authentication policy is added, we can publish the API to the API Gateway:

And, here it is, the published API:

OK, that finishes the definition of the API provider organization and the publication of the API.

Next, we'll switch over to the API consumer side and create the API consumer organization and register an application to connect to the managed API through the proxy of the API Gateway.

The API Consumer Organization

We'll repeat the process that we used to create the application development organization. Log in to the API Manager UI as the "appdev" user and create the organization:

Unlike the process we used when we created the elements used by the API provider, the first step that we'll take is to create a new application and then search for the API to be used by the application:

Searching for the API is easy, as we were careful to set the API name to something memorable:

Select the API name, and then specify the plan to be used. We'll splurge and use the gold plan:

Next, select "create contract" for the plan (for this example, we'll just accept all the defaults):

The last step is to register the application with the API Gateway so that the gateway can act as a proxy for the API:

Congratulations! All the steps necessary to both provide and consume the configure the API are complete!

There's just one more step that we have to take in order for clients to be able access the API through the API Gateway.

Remember the URL that we used to access the unmanaged API directly? Well, forget it. In order to access the managed API through the API Gateway acting as a proxy for other API we have

to obtain the managed API's URL. In the API Manager UI, header over to the "APIs" tab for the application, select the API and then click select on the "i" character to the right of the API name. This will expose the API Key and the API's HTTP endpoint in the API Gateway:

In order to be able access the API through the API Gateway, we have to provide the API Key with each request. combine the API Key and HTTP endpoint. The API Key can be provided either through an HTTP Header (X-API-Key) or a URL query parameter.

In our example, the API request looks like this:

```
https://localhost:8443/apiman-gateway/ACMEAPIs/echo/1.0?
apikey=ed4564c1-2715-45f6-881e-ca8bc1168d17
```

Copy the URL into the clipboard.

Accessing the Managed API Through the apiman API Gateway, Watching the Policies at Runtime

Thanks for hanging in there! The set up is done. Now, we can fire up the client and watch the policies in action as they are applied at runtime by the API Gateway.

Open a new browser window or tab, and enter the URL for the managed API.

What happens first is that the authentication policy is applied and a login dialog is then displayed:

Enter the username and password (user1/password) that we defined when we created the authentication policy to access the API. The fact that you are seeing this dialog confirms that you are accessing the managed API and are not accessing the API directly.

When you send a GET request to the API, you should see a successful response:

```
{
 "method" : "GET",
 "resource" : "/apiman-echo",
 "uri" : "/apiman-echo",
 "headers" : {
 "Cookie" : "s_fid=722D028B20E49214-13EAE1456E752098;
 __utma=111872281.807845787.1452188093.1460777731.1460777731.4;
 __utmz=111872281.1452188093.1.1.utmcsr=(direct)|
utmccn=(direct)|utmcmd=(none); _ga=GA1.1.807845787.1452188093;
 __qca=P0-404983419-1452188093717; __utmc=111872281",
 "Accept" : "text/html,application/xhtml+xml,application/xml;q=0.9,*/
*;q=0.8",
 "User-Agent" : "Mozilla/5.0 (X11; Linux x86_64; rv:38.0) Gecko/20100101
 Firefox/38.0",
```

```
  "Connection" : "keep-alive",
  "X-Identity" : "user1",
  "Host" : "localhost:8080",
  "Accept-Language" : "en-US,en;q=0.5",
  "Accept-Encoding" : "gzip, deflate",
  "DNT" : "1"
  },
  "bodyLength" : null,
  "bodySha1" : null
}
```

So far so good. Now, send 10 more requests and you will see a response that looks like this as the gold plan rate limit is exceeded:

```
{"type":"Other","failureCode":10005,"responseCode":429,"message":"Rate
 limit exceeded.","headers":{"entries":[{"X-RateLimit-Remaining":"-1"},{"X-
RateLimit-Reset":"50904"},{"X-RateLimit-Limit":"10"}],"empty":false}}
```

And there it is. Your gold plan has been exceeded. Maybe next time you'll spend a little more and get the platinum plan! ;-)

Let's recap what we just accomplished in this demo:

- We installed apiman onto a WildFly server instance.
- We used git to download and maven to build a sample REST client.
- As an API provider, we created an organization, defined policies based on API use limit rates and user authentication, and a plan, and assigned them to an API.
- As an API consumer, we searched for and found that API, and assigned it to an application.
- As a client, we accessed the API and observed how the API Gateway managed the API.

And, if you note, in the process of doing all this, the only code that we had to write or build was for the client. We were able to fully configure the API, policies, plans, and the application in the API Manager UI.

Chapter 9. Section 7 - Apiman Resources

- Apiman site - <u>http://www.apiman.io/latest/</u>
- Apiman blog - <u>http://www.apiman.io/blog/</u>
- Apiman downloads - <u>http://www.apiman.io/latest/download.html</u>
- Apiman user guide - <u>http://www.apiman.io/latest/user-guide.html</u>
- Apiman developer guide - <u>http://www.apiman.io/latest/developer-guide.html</u>
- Apiman videos - <u>https://vimeo.com/user34396826</u>
- Apiman on github - <u>https://github.com/apiman</u>
- Apiman on JIRA - <u>https://issues.jboss.org/projects/APIMAN</u>
- Apiman chat on IRC - <u>http://www.apiman.io/latest/chat.html</u>
- Apiman on Twitter - <u>https://twitter.com/apiman_io</u>
- Apiman mailing list - <u>https://lists.jboss.org/mailman/listinfo/apiman-user</u>
- Apiman contributors - <u>http://www.apiman.io/latest/contributors.html</u>

Chapter 10. API Manager

There are two layers to the API management project. There is an API Manager which allows end users to centrally manage their APIs. Additionally there is an API Gateway which is responsible for applying those policies to API requests.

The API Manager allows end-users to track, configure, and share APIs with other users. All of this is accomplished by the end user by logging into the API Manager user interface (or using the API Manager REST API).

10.1. Data Model

It is perhaps most important to understand the various entities used by the API Manager, as well as their relationships with each other.

10.1.1. Organizations

The top level container concept within the API management project its called the organization. All other entities are managed within the scope of an organization.

When users log into the API management system they must be affiliated with one or more organization. Users can have different roles within that organization allowing them to perform different actions and manage different entities. Please see the 'User Management' section below for more details on this topic.

What an organization actually represents will depend upon who is using API management. When installed within a large enterprise, an organization may represent an internal group within IT (for example the HR group). If installed in the cloud, an organization might represent an external company or organization.

In any case, an organization is required before the end user can create or consume APIs.

10.1.2. Policies

The most important concept in API management is the policy. The policy is the unit of work executed at runtime in order to implement API governance. All other entities within the API Manager exist in support of configuring policies and sensibly applying them at runtime.

When a request for an API is made at runtime, a policy chain is created and applied to the inbound request, prior to proxying that request to the back-end API implementation. This policy chain consists of policies configured in the API Manager.

An individual policy consists of a type (e.g. authentication or rate limiting) as well as configuration details specific to the type and instance of that policy. Multiple policies can be configured per API resulting in a policy chain that is applied at runtime.

It is very important to understand that policies can be configured at three different levels within API management. Policies can be configured on an API, on a plan, or on a client app. For more details please see the sections below.

10.1.3. Plans

A plan is a set of policies that define a level of service for an API. When an API is consumed it may be consumed through a plan. Please see the section on 'API Contracts' for more information.

An organization can have multiple plans associated with it. Typically each plan within an organization consists of the same set of policies but with different configuration details. For example, an organization might have a Gold plan with a rate limiting policy that restricts consumers to 1000 requests per day. The same organization may then have a Silver plan which is also configured with a rate limiting policy, but which restricts consumers to 500 requests per day.

Once a plan has been fully configured (all desired policies added and configured) it must be locked so that it can be used by APIs. This is done so that API providers can't change the details of the plan out from underneath the client app developers who are using it.

10.1.4. APIs

An API represents an external API that is being governed by the API management system. An API consists of a set of metadata including name and description as well as an external endpoint defining the API implementation. The external API implementation endpoint includes:

- The type/protocol of the endpoint (e.g. REST or SOAP)
- The endpoint content type (e.g. XML or JSON)
- The endpoint location (URL) so that the API can be properly proxied to at runtime.

In addition, policies can be configured on an API. Typically, the policies applied to APIs are things like authentication, or caching. Any policies configured on API will be applied at runtime regardless of the client app and API contract. This is why authentication is a common policy to configure at the API level.

APIs may be offered through one or more plans configured in the same organization. When plans are used, API consumers (client apps) must consume the API through one of those plans. Please see the section on 'API Contracts' for more information. Alternatively, an API can simply be marked as "Public", in which case any client may access the API's managed endpoint without providing an API Key.

Only once an API is fully configured, including its policies, implementation, and plans can it be published to the Gateway for consumption by client apps. Once an API has been published, it can only be changed if it is a "Public" API. APIs that are offered via Plans are immutable - to change them you must create a new version. The reason for this is that API consumers may have created Contracts with your API, through your Plan. When they do this, they must agree to some terms and conditions. It is therefore understood that the terms to which they are agreeing will not

change. However, for Public APIs, there is no such agreement. For this reason, you can make changes to Public APIs and re-publish them at any time.

10.1.5. Client Apps

A client app represents a consumer of an API. Typical API consumers are things like mobile applications and B2B applications. Regardless of the actual implementation, a client app must be added to the API management system so that Contracts can be created between it and the APIs it wishes to consume.

A client app consists of basic metadata such as name and description. Policies can also be configured on a client app, but are optional.

Finally, API Contracts can be created between a client app and the API(s) it wishes to consume. Once the API Contracts are created, the client app can be registered with the runtime gateway. Policies and Contracts can be added/removed at any time. However, after any changes are made, you must re-register the client app.

10.1.6. API Contracts

An API contract is simply a link between a Client App and an API through a plan offered by that API. This is the only way that a client app can consume an API. If there are no client apps that have created API contracts with an API, that API cannot be accessed through the API management runtime gateway (unless of course the API is "Public").

When an API Contract is created, the system generates a unique API key specific to that contract. All requests made to the API by a Client App through the API Gateway must include this API key. The API key is used to create the runtime policy chain from the policies configured on the API, plan, and client app.

API Contracts can only be created between Client Apps and published APIs which are offered through at least one Plan. An API Contract cannot be created between a Client App and a Public API.

10.1.7. Policy Chain

A policy chain is an ordered sequence of policies that are applied when a request is made for an API through the API Gateway. The order that policies are applied is important and is as follows:

1. Client App
2. Plan
3. API

Within these individual sections, the end user can specify the order of the policies.

When a request for an API is received by the API Gateway the policy chain is applied to the request in the order listed above. If none of the policies fail, the API Gateway will proxy the request to the

backend API implementation. Once a response is received from the back end API implementation, the policy chain is then applied in reverse order to that response. This allows each policy to be applied twice, once to the inbound request and then again to the outbound response.

10.2. User Management

The API Manager offers user role capabilities at the organization level. Users can be members of organizations and have specific roles within those organizations. The roles themselves are configurable by an administrator, and each role provides the user with a set of permissions that determine what actions the user can take within an organization.

10.2.1. New Users

Users must self register with the management UI in order to be given access to an organization or to create their own organization. In some configurations it is possible that user self registration is unavailable and instead user information is provided by a standard source of identity such as LDAP. In either case, the actions a user can take are determined by that user's role memberships within the context of an organization.

10.2.2. Membership

Users can be members of organizations. All memberships in an organization include the specific roles the user is granted. It is typically up to the owner of an organization to grant role memberships to the members of that organization.

10.2.3. Roles

Roles determine the capabilities granted a user within the context of the organization. The roles themselves and the capabilities that those roles grant are configured by system administrators. For example, administrators would typically configure the following roles:

- Organization Owner
- API Developer
- Client App Developer

Each of these roles is configured by an administrator to provide a specific set of permissions allowing the user to perform relevant actions appropriate to that role. For example the Client App Developer role would grant an end user the ability to manage client apps and API contracts for those client apps. However that user would not be able to create or manage the organization's APIs or plans.

10.3. Managing Organizations

Before any other actions can be taken an organization must exist. All other operations take place within the context of an organization.

In order to create an organization click the 'Create a New Organization' link found on the dashboard page that appears when you first login. Simply provide an organization name and description and then click the 'Create Organization' button. If successful you will be taken to the organization details page.

If you create multiple organizations, you can see the list of those organizations on your home page. For example, you may click the 'Go to My Organizations' link from the dashboard page.

10.4. Managing Plans

Plans must be managed within the scope of an organization. Once created, plans can be used for any API defined within that same organization. To see a list of existing plans for an organization, navigate to the 'Plans' tab for that organization on its details page.

10.4.1. Creating a Plan

Plans can be created easily from the 'Plans' tab of the organization details page. Simply click the 'New Plan' button and then provide a plan name, version, and description. Once that information is provided, click the 'Create Plan' button. If successfully created, you'll be taken to the plan details page.

10.4.2. Plan Policies

If you switch to the 'Policies' tab on the plan details page you can configure the list of policies for the plan. Please note that the order of the policies can be changed and is important. The order that the policies appear in the user interface determines the order they will be applied at runtime. You can drag a policy up and down the list to change the order.

To add a policy to the plan click the 'Add Policy' button. On the resulting page choose the type of policy you wish to create and then configure the details for that policy. Once you have configured the details click the 'Add Policy' button to add the policy to the plan.

10.4.3. Locking the Plan

Once all your plan policies are added and configured the way you want them, you will need to Lock the plan. This can be done from any tab of the Plan UI page. Locking the plan will prevent all future policy changes, and make the plan available for use by APIs.

10.5. Providing APIs

A core capability of API management is for end users to create, manage, and configure APIs they wish to provide. This section explains the steps necessary for users to provide those APIs.

10.5.1. Creating an API

First the user must create an API within an organization. If an organization does not yet exist one can easily be created. See the 'Managing Organizations' section for details.

From the organization details page, navigate to the 'APIs' tab and click on the 'New API' button. You will be asked to provide an API name, version number, and description.

If successfully created, you will be taken to the API details page. From here you can configure the details of the API.

10.5.2. API Implementation

Every API must be configured with an API implementation. The implementation indicates the external API that the API Gateway will proxy to if all the policies are successfully applied. Click the 'Implementation' tab to configure the API endpoint and API type details on your API.

The 'Implementation' tab is primarily used to configure the details of the back-end API that apiman will proxy to at runtime. You must configure the following:

- **Endpoint URL** - The URL that apiman will use to proxy a request made for this API.
- **Endpoint Type** - Currently either REST or SOAP (not presently used, future information)
- **Endpoint Content Type** - Choose between JSON and XML, information primarily used to respond with a policy failure or error in the appropriate format.

Additionally, the 'Implementation' tab allows you to configure any security options that might be required when proxying requests to the back-end API. For example, if you are using two-way SSL to ensure full security between the API Gateway and your back-end API, you may configure that here. We also support simple BASIC authentication between the gateway and your back end API. Please note that BASIC authentication is not ideal, and especially insecure if not using SSL/HTTPS to connect to the back end API.

If the apiman administrator has configured multiple Gateways (see the "System Administration / Gateways" section below), then the 'Implementation' tab will also include an option that will let you choose which Gateway(s) to use when publishing. You may select one or more Gateway in this case. If you choose multiple Gateways, then when you click the 'Publish' button, apiman will publish the API to **all** of the selected Gateways.

Tip

If a single Gateway has been configured, then you don't have a choice, and so the UI will hide the Gateway selector entirely and simply pick the default Gateway for you.

Do not forget to click the Save button when you are done making changes.

10.5.3. API Definition

As a provider of an API, it is best to include as much information about the API as possible, so that consumers can not only create contracts, but also learn how to make calls. For this purpose, you

can optionally include an API Definition document by adding it to your API on the Definition tab. Currently the only supported type of definition file is Swagger. Include a swagger spec document here so that consumers of your API can browse information about your API directly in the API Manager UI.

10.5.4. Available Plans

Before an API can be consumed by a client app, it must make itself available through at least one of the organization's plans (or it must be marked as "Public"). Marking an API as public or making an API available through one or more plan can be done by navigating to the 'Plans' tab on the API details page. The 'Plans' tab will list all of the available plans defined by the organization. Simply choose one or more plan from this list. If no plans are needed, you can instead mark the API as "Public", making it available to be consumed anonymously by any client. Although an API can be **both** Public and available through one or more plan, it is unusual to do so.

Tip

After you have either marked the API as "public" or selected at least one plan, make sure to click the Save button.

10.5.5. Managing Policies

API policies can be added and configured by navigating to the 'Policies' tab on the API details page. The 'Policies' tab presents a list of all the policies configured for this API. To add another policy to the API click the 'Add Policy' button. On the resulting page choose the type of policy you wish to create and then configure the details for that policy. Once you have configured the details click the 'Add Policy' button to add the policy to the API.

10.5.6. Publishing in the Gateway

After all of the configuration is complete for an API, it is time to publish the API to the runtime gateway. This can be done from any tab on the API details page by clicking the 'Publish' button in the top section of the UI. If successful, the status of the API will change to "Published" and the 'Publish' button will disappear.

Tip

If the API cannot yet be published (the 'Publish' button is disabled) then a notification will appear near the button and will read "Why Can't I publish?" Clicking this notification will provide details about what information is still required before the API can be published to the Gateway.

Once the API has been published, it may or may not be editable depending on whether it is a "Public" API or not. For "Public" APIs, you will be able to continue making changes. After at least

one change is made, you will have the option to "Re-Publish" the API to the Gateway. Doing so will update all information about the API in the Gateway. However, if the API is **not** Public, then the API will be immutable - therefore in order to make any changes you will need to create a new version of the API.

10.5.7. API Metrics

Once an API is published and is being consumed at runtime, metrics information about that usage is recorded in a metrics storage system. See the Metrics section of the API Gateway documentation for more about how and when metrics data is recorded.

If an API has been used by at least once, then it will have metrics information available. This information can be viewed in the 'Metrics' tab on the API's details page. On this page you can choose the type of metric you wish to see (e.g. Usage metrics and Response Type metrics) as well as a pre-defined time range (e.g. Last 30 Days, Last Week, etc…).

The API Metrics page is a great way to figure out how often your API is used, and in what ways.

10.5.8. Importing API(s)

As an alternative to manually creating and configuring an API, apiman also supports importing an API from a globally configured API Catalog.

Tip

The API Catalog is configured by the apiman system administrator/installer. See the installation guide for more information about how to configure a custom API Catalog.

An API can be imported into apiman in one of two ways. First, from the Organization's "APIs" tab you can click the down-arrow next to the "New API" button and choose the "Import API(s)" option. This results in a wizard that will guide you through importing one or more API from the catalog into the Organization. This wizard will allow you to search for, find, and select multiple APIs. It will then walk you through choosing your Plans or making the APIs "Public". Once all the wizard pages are completed, you can then import the API(s).

Tip

The Import API(s) wizard above is the only way to import multiple APIs at the same time.

Another option for importing an API from the catalog is to use the API Catalog Browser UI, which can be found by clicking the "Browse available/importable APIs" link on the API Manager Dashboard. This link will open the catalog browser, allowing you to search for APIs to import. The catalog browser is a friendlier interface, but only allows you to import a single API at a time.

10.6. Consuming APIs

After the API providers have added a number of APIs to the API management system, those APIs can be consumed by Client Apps. This section explains how to consume APIs.

10.6.1. Consuming Public APIs

If you have marked an API as "Public", then consuming it is a simple matter of sending a request to the appropriate API Gateway endpoint. The managed API endpoint may vary depending on the Gateway being used, but it typically of the following form:

- *http://gatewayhost:port/apiman-gateway/{organizationId}/{apiId}/{version}/*

Simply send requests to the managed API endpoint, and do not include an API Key.

Tip

The managed endpoint URL can be easily determined in the UI by navigating to the "Endpoint" tab on the API details UI page.

10.6.2. Creating a Client App

In order to consume an API that is not "Public" you must first create a client app. Client Apps must exist within the context of an organization. If an organization does not yet exist for this purpose, simply create a new organization. See the section above on 'Managing Organizations' for more information.

To create a new Client App click the 'Create a New Client App' link on the dashboard page. On the resulting page provide a client app name, version, and description and then click the 'Create Client App' button. If the client app is successfully created, you will be taken to the client app details page.

Tip

You can also create a Client App within an Organization by going to the Organization's "Client Apps" tab and clicking the "New Client App" button.

10.6.3. Creating API Contracts

The primary action taken when configuring a client app is the creation of Contracts to APIs. This is what we mean when we say "consuming an API". There are a number of ways to create API contracts. This section will describe the most useful of these options.

From the Client App details page, you can find an API to consume by clicking on the 'Search for APIs to consume' link in the top section of the page. You will be taken to a page that will help you search for and find the API you wish to consume.

Use the controls on this page to search for an API. Once you have found the API you are interested in, click on its name in the search results area. This will take you to the API details page for API consumers. The consumer-oriented API details page presents you with all of the information necessary to make a decision about how to consume the API. It includes a list of all the API versions and a list of all of the available plans the API can be consumed through.

Note that you can click on an individual plan to see the details of the policies that will be enforced should that plan be chosen. Click on the 'Create Contract' button next to the plan you wish to use when consuming this API. You will be taken to the new contract page to confirm that you want to create an API contract to this API through the selected plan. If you are sure this is the API contract you wish to create, click the 'Create Contract' button and then agree to the terms and conditions. If successful, you will be taken to the 'Contracts' tab on the client app details page.

From the 'Contracts' tab on the client app details page you can see the list of API contracts already created for this client app. It is also possible to break API contracts from this same list by clicking an appropriate 'Break Contract' button.

10.6.4. API Definition Information

If An API provider has included An API Definition for the API they are providing, you will be presented with an additional link on the consumer-oriented API details page labeled "API Definition". This link will take you to a page where you can browse the detailed documentation for the API. The detailed documentation should be very helpful in learning what resources and operations are supported by the API, which will aid in figuring out how precisely to consume the API.

10.6.5. Managing Policies

Just like plans and APIs, client apps can have configured policies. The 'Policies' tab will present a list of all the policies configured for this client app. To add another policy to the client app click the 'Add Policy' button. On the resulting page choose the type of policy you wish to create and then configure the details for that policy. Once you have configured the details click the 'Add Policy' button to add the policy to the client app.

Of course, just like for Plans and APIs, you can manage the Client App policies from the 'Policies' tab. This allows you to not only add new policies but also edit, remove, and reorder them.

10.6.6. Registering in the Gateway

After at least one API contract has been created for the client app, it is possible to register the client app with the runtime gateway. Until the client app is registered with the runtime gateway, it is not possible to make requests to back-end APIs on behalf of that client app.

To register the client app with the gateway, simply click the "Register" button at the top of the Client App details UI page (any tab). If the status of the client app is "Ready", then the 'Register' button should be enabled. If successful, the client app status will change to "Registered", and the 'Register' button will disappear.

Once the client app is registered, you can continue to make changes to it (such as modify its policies or create/break API Contracts). If you do make any changes, then the 'Re-Register' button will become enabled. Whenever you make changes to your Client App, you **must** Re-Register it before those changes will show up in the Gateway.

10.6.7. Live API Endpoints

After a client app has been registered with the runtime gateway, it is possible to send requests to the back-end APIs on behalf of that client app (through the client app's API contracts). To do this you must know the URL of the managed API. This URL 'optionally' includes the API Key generated for the Client App.

To view a list of all of these managed endpoints, navigate to the 'APIs' tab on the API detail page. Each API contract is represented in the list of managed endpoints. You can expand an entry in the managed API endpoints table by clicking the '>' icon in the first column. The resulting details will help you figure out the appropriate endpoint to use for a particular managed API.

Tip

There are two ways to pass the API Key to the Gateway when you make a request for a Managed Endpoint. You can either include the API Key in the URL as a query parameter, or you can pass it via the **X-API-Key** HTTP header.

10.7. Versioning

Many of the entities in the API Manager support multiple simultaneous versions. These include the following:

- Plans
- APIs
- Client Apps

Typically once an entity is frozen (e.g. Locked or Published) it can no longer be modified. But often as things change, modifications to the API Management configuration are necessary. For example, as an API implementation evolves, the policies associated with it in the API Manager may need to change. Versioning allows this to happen, by providing a way for a user to create a new version of a particular API (or Client App or Plan) and then making changes to it.

To create a new version of an entity, view the details of the entity and click the "New Version" button in the UI. This will allow you to make a new version of the entity. You can either make a simple, empty new version or you can make a clone of an existing version. The latter is typically more convenient when making incremental changes.

Tip

"Public" APIs and Client Apps can be modified and re-published (or re-registered) in the Gateway without the need to create a new version.

10.8. System Administration

There are several "global" settings that must be configured/managed by an apiman administrator. These global settings are managed by navigating to the **System Administration** section of the API Manager UI.

10.8.1. Roles

Users must become a member of an organization before being allowed to manage any of the plans, APIs, or client apps contained within it. When a user is made a member of an organization, they are granted a specific role within that organization. Typical examples of roles are **Organization Owner**, **API Provider**, and **Client App Developer**. These roles each grant different specific privileges to the user. For example, a user with the **Client App Developer** role will be able to manage the organization's client apps but not its APIs or plans.

The roles that are available when adding a member to an organization are managed in the **Roles** section of the **System Administration** UI. The apiman admin can create as many roles as she wishes, giving each one a name, description, and the set of permissions it grants. Additionally, certain roles may be automatically granted to users who create new organizations. At least one such role must be present, otherwise organizations cannot be created.

10.8.2. Policy Definitions

The policies available when configuring APIs, Plans, and Client Apps are controlled by the **Policy Definitions** known to apiman. These definitions are stored in the API Manager and are added by the apiman admin. Typically these are added once and rarely changed. But as new versions of apiman are released, additional policies will be made available. For each policy, a policy definition must be configured in the **System Administration** UI.

Additionally, it is possible for a plugin, when installed, to contribute one or more policy definitions to the list. This is a very common way for new policy definitions to be added to apiman.

10.8.3. Gateways

Apiman allows multiple logical Gateways to be configured. The Gateway is the server that actually applies the policies configured in the API Manager to live requests to managed APIs. When using apiman, at least one Gateway must be running and configured in the API Manager. However, there is no limit to the total number of Gateways that may be running. The typical reason to have multiple Gateways is when some APIs are very high volume and others are not. In this case,

the high volume APIs could be published to a Gateway that can handle such load, while the low volume APIs could be published to another (perhaps cheaper) Gateway.

Another reason you may want multiple Gateways if if you need some of your APIs to be provided in a particular physical region and others in a different one. In this case, you may have a Gateway (perhaps clustered) running in a US data center, while another Gateway (different cluster) is running separately in a data center in Europe.

In all cases, the apiman admin must configure these Gateways in the **System Administration** UI. Each Gateway has a name, description, and configuration endpoint. The configuration endpoint is what the API Manager will use when publishing APIs and client apps into the Gateway.

When configuring an API Gateway you will need to include the authentication credentials required to invoke the API Gateway configuration REST API. Typically this user must have the 'apipublisher' role in order to successfully talk to the API Gateway. The Gateway UI includes a **Test Gateway** button which will attempt to contact the Gateway API with the credentials included. If successful, the test button will turn green. If unsuccessful, details about the failure will be displayed and the test button will turn red.

10.8.4. Plugins

Apiman supports contributing additional functionality via a powerful plugin mechanism. Plugins can be managed by an administrator within the API Manager UI. The plugin management administration page allows an admin to install and uninstall plugins.

10.8.4.1. Adding Plugins

The Plugin admin page has two tabs - one shows the list of plugins currently installed, and the other shows a list of "Available Plugins". The list of available plugins comes from a plugin registry that is configured when apiman is installed (see the Installation Guide for details on how to configure a custom plugin registry). By default, the "official" apiman plugins will show up in the list.

A custom plugin is typically added by clicking on the 'Add Custom Plugin' button found on the "Available Plugins" tab. This allows you to install a plugin that is not found in the configured plugin registry. When installing a custom plugin, you must provide the "coordinates" of the plugin. All plugins are actually maven artifacts, and as such their coordinates consist of the following maven properties:

- Group ID
- Artifact ID
- Version
- Classifier (optional)
- Type (optional, defaults to 'war')

When installing a plugin from the plugin registry, simply locate it in the list shown on the "Available Plugins" tab and then click the "Install" action. This will again take you to the Add Plugin page, but with all of the appropriate information already filled in. At this point you should only need to click the "Add Plugin" button.

Plugins primarily are used to contribute custom policies to apiman. These policies are automatically discovered (if they exist in the plugin) when a plugin is added to the API Manager. Policies that have been contributed via a plugin will appear in the Policy Definitions admin page along with the built-in policies.

10.8.4.2. Uninstalling Plugins

At any time you may choose to uninstall a plugin. Note that if the plugin was contributing one or more policies to apiman, then the policy will no longer be available for use when configuring your Plans, APIs, and Client Apps. However, if the policy is already in use by one of these entities, it will continue to work. In other words, uninstalling a plugin only removes the policy for use by new entities, it does **not** break existing usages.

To uninstall a plugin, simply click the "Uninstall" action for the appropriate plugin on the "Installed Plugins" tab (it is likely represented as a button with a little X). After confirming the action, the plugin should disappear from the list.

10.8.4.3. Upgrading Plugins

If apiman determines that a plugin can be upgraded, then an "Upgrade Plugin" action button will show up for the plugin in the "Installed Plugins" tab. This action will be represented as an up arrow icon button. When clicked, you will be prompted for the version of the plugin you wish to upgrade **to**. The result will be that a new version of the plugin will be downloaded and installed, replacing the older version you had before. Note that any Plans, APIs, or Client Apps that were using the old version of the plugin's policies will **continue** to use the older version. However, any new policies from the plugin added to entities will use the new version. In order to upgrade an existing entity to a newer policy, you will need to remove the old policy from that entity and re-add it. We recommend that you only do this if there is a compelling reason (e.g. a bug is fixed or a new feature added).

10.8.5. Export/Import Data

Apiman has a feature that allows an admin user to Export and/or Import data. You can access this feature by clicking the "Export/Import Data" link on the API Manager Dashboard page (admin only). This feature is useful for the following use-cases:

- Backing up data
- Migrating data between environments (e.g. Test#Production)
- Upgrading between apiman versions

From the Export/Import UI page, simply click the "Export All" button if you wish to export all of the data in the API Manager. The result will be a downloaded JSON file containing all of your

apiman data. This file can then be optionally post-processed (perhaps you want to migrate only a single Organization from your Test environment to your Prod environment). At some later time, you can import this file (typically into a different installation of apiman) by selecting it and choosing "Upload File".

Chapter 11. API Gateway

The runtime layer of apiman consists of a small, lightweight and embeddable API Gateway, which is responsible for applying the policies configured in the API Manager to all requests to managed APIs. By default apiman comes with a WAR version of the API Gateway. Additionally, there is an asynchronous version of the API Gateway that runs on the vert.x platform.

11.1. Configuration

The API Gateway is a completely separate component from the API Manager, and can therefore be used completely standalone if desired. However, the API Manager provides a great deal of management functionality (along with a user interface) that is quite useful. The API Gateway has a simple REST API that is used to configure it. The API provides the following basic capabilities:

- Publish an API
- Register a Client App (with API Contracts)
- Retire an API
- Unregister a Client App

Typically the API Manager is used to manage the configuration of various APIs and client apps within the scope of one or more Organizations. At various times during the management of these entities, the user of the API Manager will 'Publish' an API or 'Register' a Client App. When this action occurs, the API Manager invokes one of the relevant API Gateway configuration endpoints listed above.

11.2. Invoking Managed APIs

Once appropriate configuration has been published/registered with the API Gateway (see the Configuration section above), the API Gateway can be used to make managed calls to the APIs it knows about. A managed API can be invoked as though the back-end API were being invoked directly, with the exception that the endpoint is obviously different. The specific endpoint to use in order to invoke a particular API can be different based on the Gateway implementation. However, typically the endpoint format is:

- *http://gatewayhost:port/apiman-gateway/{organizationId}/{apiId}/{version}/*

Note that all path segments beyond the {version} segment will be proxied on to the back-end API endpoint. Additionally, all HTTP headers and all query parameters (except for the API Key) will also be proxied to the back-end API.

Requests to managed endpoints may include the API Key so that the Gateway knows which Client App is being used to invoke the API. The API Key can be sent in one of the following ways:

- As an HTTP Header named **X-API-Key**
- As a URL query parameter named **apikey**

If the API being invoked is a "Public" API, then no API Key should be sent. However, the request should still be sent to the same endpoint as described above. The endpoint itself contains enough information to let the Gateway know what API is being invoked.

If an API is not "Public" and you ommit the API Key, then the request will fail.

11.3. Recording Metrics

The API Gateway is typically configured to record each request made to it into a metrics storage system of some kind. By default apiman will use an included elasticsearch instance to store this information. Various pieces of information about each request is included in the record, including but not necessarily limited to the following:

- Request start and end times
- API start and end times (i.e. just the part of the request taken up by the back end API)
- Resource path
- Response type (success, failure, error)
- API info (org id, id, version)
- Client App info (org id, id, version)

This information is then available for analysis and reporting. The data can be accessed in a number of ways, including:

- Through the API Manager UI
- Through the API Manager REST API
- Directly from the metrics system

Chapter 12. Policies

The most important runtime concept in apiman is the policy. Policies are configured in the API Manager and then applied at runtime by the API Gateway. This section of the guide provides more information about each of the policies available in apiman, what they do, and how they can be configured.

12.1. Security Policies

12.1.1. BASIC Authentication Policy

12.1.1.1. Description

This policy enables HTTP BASIC Authentication on an API. In other words, you can use this policy to require clients to provide HTTP BASIC authentication credentials when making requests to the managed API.

Tip

This is a built-in policy and therefore no plugins need to be installed prior to using it.

12.1.1.2. Configuration

The BASIC Authentication policy has a number of configuration options. There are several top level configuration properties:

- **realm** (string) : defines the BASIC Auth realm that will be used when responding with an auth challenge (when authentication is missing or fails)
- **forwardIdentityHttpHeader** (string) : if authentication succeeds, indicates the name of an HTTP header to send with the principal/identity of the authenticated user (useful when the back-end API needs to know the identify of the authenticated user)
- **requireTransportSecurity** (boolean) : set to true if this policy should fail when receiving a message over an unsecured communication channel (in other words, enabling this will require clients to use **https**)
- **requireBasicAuth** (boolean) : set to true if BASIC authentication credentials are **required** (set to false if alternative authentication mechanisms, such as OAuth, are also supported)

Additionally, one of the following complex properties must be included in the configuration, indicating whether apiman should use JDBC, LDAP, or Static (not recommended for production) information as the source of identity used to validate provided user credentials.

- **jdbcIdentity** (object) : included when you wish to use JDBC to connect to a database containing user and password information

- **type** (enum) : what type of JDBC connection to use - options are 'datasource', 'url'
- **datasourcePath** (string) : the JNDI path of the datasource to use (only when type is 'datasource')
- **jdbcUrl** (string) : the URL to the JDBC database (only when type is 'url')
- **username** (string) : the Username to use when connecting to the JDBC database (only when type is 'url')
- **password** (string) : the Passowrd to use when connecting to the JDBC database (only when type is 'url')
- **query** (string) : the SQL query to use when searching for a user record - the first parameter passed to the query will be the username, the second parameter will be the (optionally hashed) password
- **hashAlgorithm** (enum) : the hashing algorithm used when storing the password data in the database
- **extractRoles** (boolean) : set to true if you also want to extract role information from the database
- **roleQuery** (string) : a SQL query to use when extracting role information - the first parameter passed to the query will be the username

- **ldapIdentity** (object) : included when you wish to connect to LDAP when validating user credentials
  - **url** (string) : the URL to the LDAP server
  - **dnPattern** (string) : the pattern to use when binding to the LDAP server (you can use ${username} in this pattern)
  - **bindAs** (enum) : whether to bind directly to LDAP as the authenticating user (UserAccount), or instead to bind as a service account and then search LDAP for the user's record (ServiceAccount)
  - **credentials** (object) : an object with two properties: 'username' and 'password' - credentials used when initially binding to LDAP as a service account
  - **userSearch** (object) : an object with two properties: 'baseDn' and 'expression' - used to search for the user's LDAP record so that it can be used to re-bind to LDAP with the appropriate password
  - **extractRoles** (boolean) : set to true if you wish to extract role information from LDAP
  - **membershipAttribute** (string) : the attribute representing the user's membership in a group - each value should be a reference to another LDAP node

- **rolenameAttribute** (string) : the attribute on a role LDAP node that represents the name of the role

- **staticIdentity** (object) : used mostly for testing purposes - allows you to provide a static set of user names and passwords (do not use in production!)

12.1.1.3. Sample Configuration (LDAP)

Here is an example of the JSON configuration you might use when configuring a BASIC Authentication policy that uses LDAP to validate the inbound credentials:

```
{
   "realm" : "Example",
   "forwardIdentityHttpHeader" : "X-Identity",
   "requireTransportSecurity" : true,
   "requireBasicAuth" : true,
   "ldapIdentity" : {
      "url" : "ldap://example.org",
      "dnPattern" : "cn=${username},dc=example,dc=org",
      "bindAs" : "UserAccount",
      "extractRoles" : true,
      "membershipAttribute" : "memberOf",
      "rolenameAttribute" : "objectGUID"
   }
}
```

12.1.1.4. Sample Configuration (JDBC)

Here is an example of the JSON configuration you might use when configuring a BASIC Authentication policy that uses JDBC to validate the inbound credentials:

```
{
   "realm" : "Example",
   "forwardIdentityHttpHeader" : "X-Identity",
   "requireTransportSecurity" : true,
   "requireBasicAuth" : true,
   "jdbcIdentity" : {
      "type" : "url",
      "jdbcUrl" : "jdbc:h2:mem:UserDB",
      "username" : "dbuser",
      "password" : "dbpass123#",
      "query" : "SELECT * FROM users WHERE userid = ? AND pass = ?",
      "hashAlgorithm" : "SHA1",
      "extractRoles" : true,
      "roleQuery" : "SELECT r.rolename FROM roles r WHERE r.user = ?"
   }
```

```
}
```

12.1.2. Authorization Policy

12.1.2.1. Description

This policy enables fine grained authorization to API resources based on authenticated user roles. This policy can be used to control precisely who (authenticated users) are allowed to access the API, at an arbitrarily fine-grained level.

Tip

This is a built-in policy and therefore no plugins need to be installed prior to using it.

12.1.2.2. Configuration

The configuration of this policy consists of a number of rules that are applied to any inbound request to the API. Each rule consists of a regular expression pattern, an HTTP verb, and the role that an authenticated user must possess in order for access to be granted.

Tip

It's **very** important to note that this policy must be configured **after** one of the standard apiman authentication policies (e.g. the BASIC Authentication policy or the Keycloak OAuth Policy). The reason is that an Authentication policy is responsible for extracting the authenticated user's roles, which is data that is required for the Authorization Policy to do its work.

- **rules** (array) : Array of rules - each rule is applied only if it matches the current request.
  - **pathPattern** (string regexp) : Pattern that must match the request resource path you'd like the policy to be applicable to.
  - **verb** (string) : The HTTP verb that must match the request you'd like the policy to be applicable to.
  - **role** (string) : The role the user must have if this pattern matches the request.
- **multimatch** (boolean) : Should the request pass when any or all of the authorization rules pass? Set to true if all rules must match, false if only one rule must match.
- **requestUnmatched** (boolean) : If the request does not match any of the authorization rules, should it pass or fail? Set to true if you want the policy to **pass** when no rules are matched.

12.1.2.3. Sample Configuration

```
{
   "rules" : [
    {
     "pathPattern": "/admin/.*",
     "verb": "*",
     "role": "admin"
    },
    {
     "pathPattern": "/.*",
     "verb": "GET",
     "role": "user"
    }
   ],
   "multiMatch": true,
   "requestUnmatched": false
}
```

12.1.3. SOAP Authorization Policy

12.1.3.1. Description

This policy is nearly identical to our Authorization Policy, with the exception that it accepts a SOAPAction in the HTTP header. Please note that this policy will only accept a single SOAPAction header, and will not extract the operation name from the SOAP body.

12.1.3.2. Plugin

```
{
 "groupId": "io.apiman.plugins",
 "artifactId": "apiman-plugins-soap-authorization-policy",
 "version": "1.2.5.Final" // Please check for the latest version, this is just
 an example.
}
```

12.1.3.3. Configuration

Just as with the Authorization policy, you can define any number of rules you'd like.

- **rules** (array) : A single rule that your policy will apply if each of the following properties match:
  - **action** (string) : Defines the SOAPAction you'd like the policy to be applicable to.
  - **role** (string) : The role the user must have if this pattern matches the request.

- **multiMatch** (boolean) : Should the request pass when any or all of the authorization rules pass? Set to true if all rules must match, false if only one rule must match.
- **requestUnmatched** (boolean) : If the request does not match any of the authorization rules, should it pass or fail? Set to true if you want the policy to **pass** when no rules are matched.

12.1.3.4. Sample Configuration

```
{
   "rules" : [
    {
     "action": "hello",
     "role": "admin"
    },
    {
     "action": "goodbye",
     "role": "user"
    }
   ],
   "multiMatch": true,
   "requestUnmatched": false
}
```

12.1.4. IP Whitelist Policy

12.1.4.1. Description

The IP Whitelist Policy Type is the counterpart to the IP Blacklist Policy type. In the IP Whitelist policy, only inbound API requests from Client Apps, policies, or APIs that satisfy the policy are accepted.

The IP Blacklist and IP Whitelist policies are complementary, but different, approaches to limiting access to an API: * The IP Blacklist policy type is exclusive in that you must specify the IP address ranges to be excluded from being able to access the API. Any addresses that you do not explicitly exclude from the policy are able to access the API. * The IP Whitelist policy type is inclusive in that you must specify the IP address ranges to be included to be able to access the API. Any addresses that you do not explicitly include are not able to access the API.

Tip

This is a built-in policy and therefore no plugins need to be installed prior to using it.

12.1.4.2. Configuration

The configuration parameters for an IP Whitelist Policy are:

- **ipList** (array) : The IP address(es), and/or ranges of addresses that will be allowed to access the API.
- **responseCode** (int) : The server response code. The possible values for the return code are:
  - 500 - Server error
  - 404 - Not found
  - 403 - Authentication failure
- **httpHeader** (string) [optional] : Tells apiman to use the IP address found in the given HTTP request header **instead** of the one associated with the incoming TCP socket. Useful when going through a proxy, often the value of this is 'X-Forwarded-For'.

12.1.4.3. Sample Configuration

```
{
  "ipList" : ["192.168.3.*", "192.168.4.*"],
  "responseCode" : 403,
  "httpHeader" : "X-Forwarded-For"
}
```

12.1.5. IP Blacklist Policy

12.1.5.1. Description

As its name indicates, the IP blacklist policy type blocks access to an API's resources based on the IP address of the client application. The apiman Management UI form used to create an IP blacklist policy enables you to use wildcard characters in specifying the IP addresses to be blocked. In addition, apiman gives you the option of specifying the return error code sent in the response to the client if a request is denied. Note that an IP Blacklist policy in a plan overrides the an IP Whitelist policy in the same plan.

Tip

This is a built-in policy and therefore no plugins need to be installed prior to using it.

12.1.5.2. Configuration

The configuration parameters for an IP Blacklist Policy are:

- **ipList** (array) : The IP address(es), and/or ranges of addresses that will be blocked from accessing the API.

- **responseCode** (int) : The server response code. The possible values for the return code are:
  - 500 - Server error
  - 404 - Not found
  - 403 - Authentication failure
- **httpHeader** (string) [optional] : Tells apiman to use the IP address found in the given HTTP request header **instead** of the one associated with the incoming TCP socket. Useful when going through a proxy, often the value of this is 'X-Forwarded-For'.

12.1.5.3. Sample Configuration

```
{
  "ipList" : ["192.168.7.*"],
  "responseCode" : 500,
  "httpHeader" : "X-Forwarded-For"
}
```

12.1.6. Ignored Resources Policy

12.1.6.1. Description

The ignored resources policy type enables you to shield some of an API's resources from being accessed, without blocking access to all the API's resources. Requests made to access to API resources designated as "ignored" result in an HTTP 404 ("not found") error code. By defining ignored resource policies, apiman enables you to have fine-grained control over which of an API's resources are accessible.

For example, let's say that you have an apiman managed API that provides information to remote staff. The REST resources provided by this API are structured as follows:

/customers /customers/{customer id}/orders /customers/{customer id}/orders/bad_debts

By setting up multiple ignored resource policies, these policies can work together to give you more flexibility in how you govern access to to your API's resources. What you do is to define multiple plans, and in each plan, allow differing levels of access, based on the paths (expressed as regular expressions)defined, for resources to be ignored. To illustrate, using the above examples:

This Path	Results in these Resources Being Ignored
(empty)	Access to all resources is allowed
/customers	Denies access to all customer information

This Path	Results in these Resources Being Ignored
/customers/.*/orders	Denies access to all customer order information
/customers/.*/orders/bad_debts	Denies access to all customer bad debt order information

What happens when the policy is applied to an API request is that the apiman Gateway matches the configured paths to the requested API resources. If any of the exclusion paths match, the policy triggers a failure with an HTTP return code of 404.

The IP-related policy types are less fine-grained in that they allow or block access to all of an API's resources based on the IP address of the client application. We'll look at these policy types next.

Tip

This is a built-in policy and therefore no plugins need to be installed prior to using it.

12.1.6.2. Configuration

The configuration parameters for an Ignored Resources Policy are: * **rules** (array of objects) : The list of matching rules representing the resources to be ignored. **verb (enum) : The HTTP verb to be controlled by the rule. Valid values are:** * * (matches all verbs) *** GET** * POST *** PUT** * DELETE *** OPTIONS** * HEAD *** TRACE** * CONNECT *** *pathPattern** (string regexp) : A regular expression used to match the REST resource being hidden.

12.1.6.3. Sample Configuration

```
{
  "rules" : [
    { "verb" : "GET",  "pathPattern" : "/customers" },
    { "verb" : "POST", "pathPattern" : "/customers/.*/orders" },
    { "verb" : "*",    "pathPattern" : "/customers/.*/orders/bad_debts"}
  ]
}
```

12.1.7. Time Restricted Access Policy

12.1.7.1. Description

This policy is used to only allow access to an API during certain times. In fact, the policy can be configured to apply different time restrictions to different API resources (matched via regular expressions). This allows you to control **when** client and users are allowed to access your API.

Tip

This is a built-in policy and therefore no plugins need to be installed prior to using it.

12.1.7.2. Configuration

The configuration parameters for a Time Restricted Access Policy are:

- **rules** (array of objects) : The list of matching rules representing the resources being controlled and the time ranges they are allowed to be accessed.
  - **timeStart** (time) : Indicates the time of day (UTC) to begin allowing access.
  - **timeEnd** (time) : Indicates the time of day (UTC) to stop allowing access.
  - **dayStart** (integer) : Indicates the day of week (1=Monday, 2=Tuesday, etc) to begin allowing access.
  - **dayEnd** (integer) : Indicates the day of week (1=Monday, 2=Tuesday, etc) to stop allowing access.
  - **pathPattern** (string regexp) : A regular expression used to match the request's resource path/destination. The time restriction will be applied only when the request's resource matches this pattern.

Tip

If none of the configured rules matches the request resource path/destination, then no rules will be applied and the request will succeed.

12.1.7.3. Sample Configuration

```
{
    "rules": [
        {
            "timeStart": "12:00:00",
            "timeEnd": "20:00:00",
            "dayStart": 1,
            "dayEnd": 5,
            "pathPattern": "/path/to/.*"
        },
        {
            "timeStart": "10:00:00.000Z",
            "timeEnd": "18:00:00.000Z",
            "dayStart": 1,
```

```
            "dayEnd": 7,
            "pathPattern": "/other/path/.*"
        }
    ]
}
```

12.1.8. CORS Policy

12.1.8.1. Description

A policy implementing CORS (Cross-origin resource sharing): a method of defining access to resources outside of the originating domain. It is principally a security mechanism to prevent the loading of resources from unexpected domains, for instance via XSS injection attacks.

For further references, see *CORS W3C Recommendation 16 January 2014* [http://www.w3.org/TR/2014/REC-cors-20140116/] and *MDN's articles* [https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS#Access-Control-Allow-Origin].

12.1.8.2. Plugin

```
{
    "groupId": "io.apiman.plugins",
    "artifactId": "apiman-plugins-cors-policy",
     "version": "1.2.5.Final" // Please check for the latest version, this is
 just an example.
}
```

12.1.8.3. Configuration

The configuration options available, are:

Table 12.1. CORS policy configuration

Option	Type	Description	Default
errorOnCorsFailure	Boolean	**Error on CORS failure.** When true, any request that fails CORS validation will be terminated with an appropriate error. When false, the request will still be sent to the backend API, but the browser will be left to enforce the CORS failure. In both cases valid CORS headers will be set	true
allowOrigin	Set<String>	Access-Control-Allow-Origin List of origins permitted to make CORS requests through the gateway. By default same-origin is permit-	Empty

Option	Type	Description	Default
		ted, and cross-origin is forbidden. An entry of * permits all CORS requests	
allowCredentials	Boolean	**Access-Control-Allow-Credentials.** Whether response may be exposed when the `credentials` flag is set to true on the request	false
exposeHeaders	Set<String>	**Access-Control-Expose-Headers.** Which non-simple headers the browser may expose during CORS	Empty
allowHeaders	Set<String>	**Access-Control-Allow-Headers.** In response to preflight request, which *headers* can be used during actual request	Empty
allowMethods	Set<String>	**Access-Control-Allow-Methods.** In response to preflight request, which *methods* can be used during actual request	Empty
maxAge	Integer	**Access-Control-Max-Age.** How long preflight request can be cached in delta seconds	Not included

12.1.8.4. Sample Configuration

```
{
   "exposeHeaders" : [
      "X-REQUESTS-REMAINING"
   ],
   "maxAge" : 9001,
   "allowOrigin" : [
      "https://foo.example",
      "https://bar.example"
   ],
   "errorOnCorsFailure" : true,
   "allowCredentials" : false,
   "allowMethods" : [
      "POST"
   ],
   "allowHeaders" : [
      "X-CUSTOM-HEADER"
   ]
}
```

12.1.9. HTTP Security Policy

12.1.9.1. Description

Security-related HTTP headers can be set, such as HSTS, CSP and XSS protection.

12.1.9.2. Plugin

```
{
    "groupId": "io.apiman.plugins",
    "artifactId": "apiman-plugins-http-security-policy",
     "version": "1.2.5.Final" // Please check for the latest version, this is
 just an example.
}
```

12.1.9.3. Configuration

Table 12.2. HTTP security policy configuration

Option	Type	Description	Default
frameOptions	Enum [DENY, SAME-ORIGIN, DISABLED]	**Frame Options.** Defines if, or how, a resource should be displayed in a frame, iframe or object.	DISABLED
xssProtection	Enum [OFF, ON, BLOCK, DISABLED]	**XSS Protection.** Enable or disable XSS filtering in the UA.	DISABLED
contentTypeOptions	Boolean	**X-Content-Type-Options.** Prevent MIME-sniffing to any type other than the declared Content-Type.	false
hsts	*Section 12.1.9.3.1, "hsts"*	**HTTP Strict Transport Security.** Configure HSTS.	None
contentSecurityPolicy	*Section 12.1.9.3.2, "contentSecurityPolicy"*	**Content Security Policy.** CSP definition.	None

12.1.9.3.1. hsts

Table 12.3. HTTP Strict Transport Security (hsts): Enforce transport security when using HTTP to mitigate a range of common web vulnerabilities.

Option	Type	Description	Default
enabled	Boolean	**HSTS.** Enable HTTP Strict Transport	false
includeSubdomains	Boolean	Include subdomains	false
maxAge	Integer	**Maximum age.** Delta seconds user agents should cache HSTS status for	0
preload	Boolean	**Enable HSTS preloading.** Flag to verify HSTS preload status. Popular browsers contain a hard-coded (pinned) list of domains and certificates, which they always connect securely with. This mitigates a wide range of identity and MIITM attacks, and is particularly useful for high-profile domains. Users must submit a request for their domain to be included in the scheme.	false

12.1.9.3.2. contentSecurityPolicy

Table 12.4. CSP (contentSecurityPolicy): A sophisticated mechanism to precisely define the types and sources of content that may be loaded, with violation reporting and the ability to restrict the availability and scope of many security-sensitive features

Option	Type	Description	Default
mode	Enum [ENABLED, REPORT_ONLY, DISABLED]	**CSP Mode.** Which content security policy mode to use.	DISABLED
csp	String	**Content Security Policy.** A valid CSP definition to apply	Empty string

12.1.9.4. Sample Configuration

```
{
   "contentSecurityPolicy" : {
      "mode" : "REPORT_ONLY",
       "csp" : "default-src none; script-src self; connect-src self; img-src
 self; style-src self;"
```

```
   },
   "frameOptions" : "SAMEORIGIN",
   "contentTypeOptions" : true,
   "hsts" : {
      "includeSubdomains" : true,
      "preload" : false,
      "enabled" : true,
      "maxAge" : 9001
   },
   "xssProtection" : "ON"
}
```

12.1.10. OAuth Policy (Keycloak)

12.1.10.1. Description

A *Keycloak* [http://www.keycloak.org]-specific OAuth2 policy to regulate access to APIs. This plugin enables a wide range of sophisticated auth facilities in combination with, for instance, Keycloak's federation, brokering and user management capabilities. An exploration of the basics can be found *in our blog* [http://www.apiman.io/blog/gateway/security/oauth2/keycloak/authentication/authorization/1.2.x/2016/01/22/keycloak-oauth2-redux.html], but we encourage users to explore the *project documentation* [http://keycloak.jboss.org/docs.html], as there is a tremendous depth and breadth of functionality, most of which work extremely well with apiman.

Keycloak's token format and auth mechanism facilitate excellent performance characteristics, with users able to easily tune the setup to meet their security requirements. In general, this is one of the best approaches for achieving security without greatly impacting performance.

12.1.10.2. Plugin

```
{
    "groupId": "io.apiman.plugins",
    "artifactId": "apiman-plugins-keycloak-oauth-policy",
     "version": "1.2.5.Final" // Please check for the latest version, this is
 just an example.
}
```

12.1.10.3. Configuration

Table 12.5. Keycloak oauth2 policy configuration

Option	Type	Description	Default
requireOauth	Boolean	**Require auth token.** Terminate request if no OAuth token is provided.	true

Option	Type	Description	Default
requireTransportSecurity	Boolean	**Require transport security.** Any request used without transport security will be rejected. OAuth2 requires transport security (e.g. TLS, SSL) to provide protection against replay attacks. It is strongly advised for this option to be switched on	true
blacklistUnsafeTokens	Boolean	**Blacklist unsafe tokens.** Any tokens used without transport security will be blackedlisted in all gateways to mitigate associated security risks. Uses distributed data store to share blacklist	true
stripTokens	Boolean	**Strip tokens.** Remove any Authorization header or token query parameter before forwarding traffic to the API	true
realm	String	**Realm name.** If you are using KeyCloak 1.2.0x or later this must be a full iss domain path (e.g. `https://mykeycloak.local/auth/realms/apimanrealm`); pre-1.2.0x simply use the realm name (e.g. `apimanrealm`).	Empty
realmCertificateString	String	**Keycloak Realm Certificate.** To validate OAuth2 requests. Must be a PEM-encoded X.509 certificate. This can be copied from the Keycloak console.	Empty
delegateKerberosTicket	Boolean	**Delegate Kerberos Ticket.** Delegate any Kerberos Ticket embedded in the Keycloak token to the API (via the Authorization header).	false
forwardRoles	*Section 12.1.10.31, "forwardRoles"* []	**Forward Keycloak roles.** Set whether to forward roles to an authorization policy.	None
forwardAuthInfo	*Section 12.1.10.32, "forwardAuthInfo"* []	**Forward auth information.** Set auth information from the token into header(s).	None

12.1.10.3.1. forwardRoles

Table 12.6. Forward Keycloak roles to the Authorization policy. You should specify your required role(s) in the Authorization policy's configuration.

Option	Type	Description	Default
active	Boolean	**Forward roles.** Opt whether to forward any type of roles. By default these will be **realm roles** unless the `applicationName` option is also provided.	false
applicationName (optional)	String	**Application Name.** Which application roles to forward. Note that you cannot presently forward realm and application roles, only one or the other.	Empty

12.1.10.3.2. forwardAuthInfo

Tip

Fields from the token can be set as headers and forwarded to the API. All *standard claims* [https://openid.net/specs/openid-connect-basic-1_0.html#StandardClaims], custom claims and *ID token fields* [https://openid.net/specs/openid-connect-basic-1_0.html#IDToken] are available (case sensitive). A special value of **access_token** will forward the entire encoded token. Nested claims can be accessed by using javascript dot syntax (e.g: `address.country`, `address.formatted`).

Table 12.7. Forward Keycloak token information

Option	Type	Description	Default
headers	String	**Header.** The header value to set (to paired field).	None
field	String	**Field.** The token field name.	None

12.1.10.4. Sample Configuration

```
{
  "requireOauth": true,
  "requireTransportSecurity": true,
  "blacklistUnsafeTokens": false,
  "stripTokens": false,
  "realm": "apiman-is-cool",
```

```
                                                                "realmCertificateString":
 "Y29uZ3JhdHVsYXRpb25zLCB5b3UgZm91bmQgdGhlIHNlY3JldCByb29tLiB5b3VyIHByaXplIGlzIGEgZnJlZSBkb3dub
  "forwardRoles": {
    "active": true
  },
  "delegateKerberosTicket": false,
  "forwardAuthInfo": [
    {
      "headers": "X-COUNTRY",
      "field": "address.country"
    },
    {
      "headers": "X-USERNAME",
      "field": "preferred_username"
    }
  ]
}
```

12.1.11. URL Whitelist Policy

12.1.11.1. Description

This policy allows users to explicitly allow only certain API subpaths to be accessed. It's particularly useful when only a small subset of resources from a back-end API should be exposed through the managed endpoint.

12.1.11.2. Plugin

```
{
 "groupId": "io.apiman.plugins",
 "artifactId": "apiman-plugins-url-whitelist-policy",
 "version": "1.2.5.Final" // Please check for the latest version, this is just
 an example.
}
```

12.1.11.3. Configuration

Configuration of the URL Whitelist Policy consists of a property to control the stripping of the managed endpoint prefix, and then a list of items representing the endpoint paths that are allowed.

- **removePathPrefix** (boolean) : Set to true if you want the managed endpoint prefix to be stripped out before trying to match the request path to the whitelisted items (this is typically set to 'true').
- **whitelist** (array of objects) : A list of items, where each item represents an API sub-resource that should be allowed.

- **regex** (string) : Regular expression to match the API sub-resource path (e.g. /foo/[0-9]/bar)
- **methodGet** (boolean) : True if http GET should be allowed (default **false**).
- **methodPost** (boolean) :True if http POST should be allowed (default **false**).
- **methodPut** (boolean) : True if http PUT should be allowed (default **false**).
- **methodPatch** (boolean) : True if http PATCH should be allowed (default **false**).
- **methodDelete** (boolean) : True if http DELETE should be allowed (default **false**).
- **methodHead** (boolean) : True if http HEAD should be allowed (default **false**).
- **methodOptions** (boolean) : True if http OPTIONS should be allowed (default **false**).
- **methodTrace** (boolean) : True if http TRACE should be allowed (default **false**).

12.1.11.4. Sample Configuration

```
{
    "removePathPrefix" : true,
    "whitelist" : [
        {
            "regex" : "/admin/.*",
            "methodGet" : true,
            "methodPost" : true
        },
        {
            "regex" : "/users/.*",
            "methodGet" : true,
            "methodPost" : true,
            "methodPut" : true,
            "methodDelete" : true
        }
    ]
}
```

12.2. Limiting Policies

Some apiman policies provide an all-or-nothing level of control over access to managed APIs. For example, IP Blacklist or Whitelist policies either block or enable all access to a managed API, based on the IP address of the client. Rate limiting and quota policies provide you with more flexible ways to govern access to managed APIs. With rate limiting and quota policies, you can place limits on either the number of requests an API will accept over a specified period of time, or the total number of of bytes in the API requests. In addition, you can use combinations of fine-

grained and coarse-grained rate limiting policies together to give you more flexibility in governing access to your managed API.

The ability to throttle API requests based on request counts and bytes transferred provides even greater flexibility in implementing policies. APIs that transfer larger amounts of data, but rely on fewer API requests can have that data transfer throttled on a per byte basis. For example, an API that is data intensive, will return a large amount of data in response to each API request. The API may only receive a request a few hundreds of times a day, but each request may result in several megabytes of data being transferred. Let's say that we want to limit the amount of data transferred to 6GB per hour. For this type of API, we could set a rate limiting policy to allow for one request per minute, and then augment that policy with a transfer quota policy of 100Mb per hour.

Each of these policies, if used singly, can be effective in throttling requests. apiman, however, adds an additional layer of flexibility to your use of these policy types by enabling you to use them in combinations.

apiman supports these types of limiting policies:

- Rate Limiting Policy
- Quota Policy
- Transfer Quota Policy

12.2.1. Rate Limiting Policy

12.2.1.1. Description

The Rate Limiting Policy type governs the number of times requests are made to an API within a specified time period. The requests can be filtered by user, application, or API and can set the level of granularity for the time period to second, minute, hour, day, month, or year. The intended use of this policy type is for fine grained processing (e.g., 10 requests per second).

Tip

This is a built-in policy and therefore no plugins need to be installed prior to using it.

12.2.1.2. Configuration

The configuration parameters for a Rate Limiting Policy are:

- **limit** (integer) : This is the number of requests that must be received before the policy will trigger.
- **granularity** (enum) : The apiman element for which the requests are counted. Valid values are:

- User
- Api
- Client

- **period** : The time period over which the policy is applied. Valid values are:
  - Second
  - Minute
  - Hour
  - Day
  - Month
  - Year
- **headerLimit** (string) [optional] : HTTP response header that apiman will use to store the limit being applied.
- **headerRemaining** (string) [optional] : HTTP response header that apiman will use to store how many requests remain before the limit is reached.
- **headerReset** (string) [optional] : HTTP response header that apiman will use to store the number of seconds until the limit is reset.

12.2.1.3. Sample Configuration

```
{
  "limit" : 100,
  "granularity" : "Api",
  "period" : "Minute",
  "headerLimit" : "X-Limit",
  "headerRemaining" : "X-Limit-Remaining",
  "headerReset" : "X-Limit-Reset"
}
```

12.2.2. Quota Policy

12.2.2.1. Description

The Quota Policy type performs the same basic functionality as the Rate Limiting policy type., however, the intended use of this policy type is for less fine grained processing (e.g., 10,000 requests per month).

Tip

This is a built-in policy and therefore no plugins need to be installed prior to using it.

12.2.2.2. Configuration

The configuration parameters for a Quota Policy are:

- **limit** (integer) : This is the number of requests that must be received before the policy will trigger.
- **granularity** (enum) : The apiman element for which the requests are counted. Valid values are:
  - User
  - Api
  - Client
- **period** : The time period over which the policy is applied. Valid values are:
  - Hour
  - Day
  - Month
  - Year
- **headerLimit** (string) [optional] : HTTP response header that apiman will use to store the limit being applied.
- **headerRemaining** (string) [optional] : HTTP response header that apiman will use to store how many requests remain before the limit is reached.
- **headerReset** (string) [optional] : HTTP response header that apiman will use to store the number of seconds until the limit is reset.

12.2.2.3. Sample Configuration

```
{
  "limit" : 100000,
  "granularity" : "Client",
  "period" : "Month",
  "headerLimit" : "X-Quota-Limit",
  "headerRemaining" : "X-Quota-Limit-Remaining",
  "headerReset" : "X-Quota-Limit-Reset"
}
```

12.2.3. Transfer Quota Policy

12.2.3.1. Description

In contrast to the other policy types, Transfer Quota tracks the number of bytes transferred (either uploaded or downloaded) rather than the total number of requests made.

Tip

This is a built-in policy and therefore no plugins need to be installed prior to using it.

12.2.3.2. Configuration

The configuration parameters for a Quota Policy are:

- **direction** (enum) : Indicates whether uploads, downloads, or both directions should count against the limit. Value values are:
  - upload
  - download
  - both
- **limit** (integer) : This is the number of requests that must be received before the policy will trigger.
- **granularity** (enum) : The apiman element for which the transmitted bytes are counted. Valid values are:
  - User
  - Api
  - Client
- **period** : The time period over which the policy is applied. Valid values are:
  - Hour
  - Day
  - Month
  - Year
- **headerLimit** (string) [optional] : HTTP response header that apiman will use to store the limit being applied.

- **headerRemaining** (string) [optional] : HTTP response header that apiman will use to store how many requests remain before the limit is reached.
- **headerReset** (string) [optional] : HTTP response header that apiman will use to store the number of seconds until the limit is reset.

12.2.3.3. Sample Configuration

```
{
  "direction" : "download",
  "limit" : 1024000,
  "granularity" : "Client",
  "period" : "Day",
  "headerLimit" : "X-XferQuota-Limit",
  "headerRemaining" : "X-XferQuota-Limit-Remaining",
  "headerReset" : "X-XferQuota-Limit-Reset"
}
```

12.3. Modification Policies

12.3.1. URL Rewriting Policy

12.3.1.1. Description

This policy is used to re-write responses from the back-end API such that they will be modified by fixing up any incorrect URLs found with modified ones. This is useful because apiman works through an API Gateway, and in some cases an API might return URLs to followup action or data endpoints. In these cases the back-end API will likely be configured to return a URL pointing to the unmanaged API endpoint. This policy can fix up those URL references so that they point to the managed API endpoint (the API Gateway endpoint) instead.

Tip

This is a built-in policy and therefore no plugins need to be installed prior to using it.

12.3.1.2. Configuration

This policy requires some basic configuration, including a regular expression used to match the URL, as well as a replacement value.

- **fromRegex** (string regex) : A regular expression used to identify a matching URL found in the response.
- **toReplacement** (string) : The replacement URL - regular expression groups identified in the **fromRegex** can be used.

- **processBody** (boolean) : Set to true if URLs should be replaced in the response body.
- **processHeaders** (boolean) : Set to true if URLs should be replaced in the response headers.

Tip

This policy **cannot** be used for any other replacements besides URLs - the policy is implemented specifically to find and replace valid URLs. As a result, arbitrary regular expression matching will not work (the policy scans for URLs and then matches those URLs against the configured regex). This is done for performance reasons.

12.3.1.3. Sample Configuration

```
{
    "fromRegex" : "https?://[^\/]*\/([.\/]*)",
    "toReplacement" : "https://apiman.example.com/$1",
    "processBody" : true,
    "processHeaders" : true

}
```

12.3.2. Transformation Policy

12.3.2.1. Description

This policy converts an API format between JSON and XML. If an API is implemented to return XML, but a client would prefer to receive JSON data, this policy can be used to automatically convert both the request and response bodies. In this way, the client can work with JSON data even though the back-end API requires XML (and responds with XML).

Note that this policy is very generic, and does an automatic conversion between XML and JSON. For more control over the specifics of the format conversion, a custom policy may be a better choice.

12.3.2.2. Plugin

```
{
    "groupId": "io.apiman.plugins",
    "artifactId": "apiman-plugins-transformation-policy",
     "version": "1.2.5.Final" // Please check for the latest version, this is
 just an example.
}
```

12.3.2.3. Configuration

The configuration of this policy consists of two properties which indicate:

1. the format required by the client
2. the format required by the back-end server

From these two properties, the policy can decide how (and if) to convert the data.

- **clientFormat** (enum) : The format required by the client, possible values are: 'XML', 'JSON'
- **serverFormat** (enum) : The format required by the server, possible values are: 'XML', 'JSON'

12.3.2.4. Sample Configuration

```
{
   "clientFormat" : "JSON",
   "serverFormat" : "XML"
}
```

12.3.3. JSONP Policy

12.3.3.1. Description

This policy turns a standard REST endpoint into a *JSONP* [https://en.wikipedia.org/wiki/JSONP] compatible endpoint. For example, a REST endpoint may typically return the following JSON data:

```
{
  "foo" : "bar",
  "baz" : 17
}
```

If the JSONP policy is applied to this API, then the caller must provide a JSONP callback function name via the URL (for details on this, see the **Configuration** section below). When this is done, the API might respond with this instead:

```
callbackFunction({  "foo" : "bar",  "baz" : 17})
backFunction({  "foo"
: "bar",  "baz"
:
```

Tip

If the API client does not send the JSONP callback function name in the URL (via the configured query parameter name), this policy will do nothing. This allows managed endpoints to support both standard REST **and** JSONP at the same time.

12.3.3.2. Plugin

```
{
    "groupId": "io.apiman.plugins",
    "artifactId": "apiman-plugins-jsonp-policy",
     "version": "1.2.5.Final" // Please check for the latest version, this is
 just an example.
}
```

12.3.3.3. Configuration

The JSONP policy has a single configuration property, which can be used to specify the name of the HTTP query parameter that the caller must use to pass the name of the JSONP callback function.

- **callbackParamName** (string) : Name of the HTTP query parameter that should contain the JSONP callback function name.

12.3.3.4. Sample Configuration

```
{
   "callbackParamName" : "callback"
}
```

If the above configuration were to be used, the API client (caller) must send the JSONP callback function name in the URL of the request as a query parameter named **callback**. For example:

```
GET /path/to/resource?callback=myCallbackFunction HTTP/1.1
Host: www.example.org
Accept: application/json
```

In this example, the response might look like this:

```
myCallbackFunction({  "property1" : "value1",  "property2" : "value2"})
CallbackFunction({  "property1"
 : "value1",  "property2"
```

12.3.4. Simple Header Policy

12.3.4.1. Description

Set and remove headers on request, response or both. The values can be literal strings, environment or system properties. Headers can be removed by simple string equality or regular expression.

12.3.4.2. Plugin

```
{
    "groupId": "io.apiman.plugins",
    "artifactId": "apiman-plugins-simple-header-policy",
     "version": "1.2.5.Final" // Please check for the latest version, this is
 just an example.
}
```

12.3.4.3. Configuration

Option	Type	Description	Default
addHeaders	*Section 12.3.4.3.1, "addHeaders"*[]	**Add and overwrite headers.** Add headers to a request, response or both.	None
stripHeaders	*Section 12.3.4.3.2, "stripHeaders"*[]	**Strip headers.** Remove headers from a request, response or both when patterns match.	None

12.3.4.3.1. addHeaders

Table 12.8. Add headers

Option	Type	Description	Default
headerName	String	**Header Name.** The name of the header to set.	Empty
headerValue	String	**Header Value.** The value of the header to set, or key into the environment or system properties, depending upon the value of `valueType`.	Empty

Option	Type	Description	Default
valueType	Enum [String, Env, "System Properties"]	**Value Type** String Treat as a literal value. Env Treat as a key into the environment `Env[valueType]`, and set the returned value. System Properties Treat as a key into the JVM's System Properties, and set the returned value.	None
applyTo	Enum [Request, Response, Both]	**Where to apply rule** Request Request only. Response Response only. Both Both request and response.	None
overwrite	Boolean	**Overwrite.** Overwrite any existing header with same name.	false

12.3.4.3.2. stripHeaders

Table 12.9. Strip headers

Option	Type	Description	Default
stripType	Enum[Key, Value]	**Strip when** Key `pattern` matches key. Value `pattern` matches value.	None
with	Enum[String, Regex]	**With matcher type** String Case-insensitive string equality.	Empty

Option	Type	Description	Default
		Regex Case-insensitive regular expression.	
pattern	String	**Using pattern.** String to match or compile into a regex, depending on the value of `with`.	Empty

12.3.4.4. Sample Configuration

```
{
  "addHeaders": [
    {
      "headerName": "X-APIMAN-IS",
      "headerValue": "free-and-open-source",
      "valueType": "String",
      "applyTo": "Response",
      "overwrite": false
    },
    {
      "headerName": "X-LANG-FROM-ENV",
      "headerValue": "LANG",
      "valueType": "Env",
      "applyTo": "Both",
      "overwrite": true
    },
    {
      "headerName": "X-JAVA-VERSION-FROM-PROPS",
      "headerValue": "java.version",
      "valueType": "System Properties",
      "applyTo": "Request",
      "overwrite": false
    }
  ],
  "stripHeaders": [
    {
      "stripType": "Key",
      "with": "String",
      "pattern": "Authorization"
    },
    {
      "stripType": "Key",
      "with": "Regex",
      "pattern": "^password=.*$"
    }
  ]
}
```

12.4. Other Policies

12.4.1. Caching Policy

12.4.1.1. Description

Allows caching of API responses in the Gateway to reduce overall traffic to the back-end API. The caching policy is currently very naive and only supports a simple "time-to-live" approach to caching.

Tip

This is a built-in policy and therefore no plugins need to be installed prior to using it.

12.4.1.2. Configuration

The caching policy only supports a single configuration options, which is the time to live in seconds.

- **ttl** (long) : Number of seconds to cache the response.

12.4.1.3. Sample Configuration

```
{
   "ttl" : 60
}
```

12.4.2. Log Policy

12.4.2.1. Description

A policy that logs the headers to standard out. Useful to analyse inbound HTTP traffic to the gateway when added as the first policy in the chain or to analyse outbound HTTP traffic from the gateway when added as the last policy in the chain.

12.4.2.2. Plugin

```
{
    "groupId": "io.apiman.plugins",
    "artifactId": "apiman-plugins-log-policy",
     "version": "1.2.5.Final" // Please check for the latest version, this is
 just an example.
}
```

12.4.2.3. Configuration

The Log Policy can be configured to output the request headers, the response headers, or both. When configuring this policy via the apiman REST API, there is only property:

- **direction** (enum) : Which direction you wish to log, options are: 'request', 'response', 'both'

12.4.2.4. Sample Configuration

```
{
   "direction" : "both"
}
```