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.

Chapter 2. Crash Course

This is the "tl;dr" section of the apiman User Guide. Read this section to get a decent high level overview of the concepts you need to know to get going quickly.

2.1. Organizations

Everything in apiman starts with the Organization. First you should create one of these. The Organization is where everything else will be managed. It also allows you to grant specific roles to other users you wish to collaborate with.

2.2. Providing Public APIs

The simplest way to get started with apiman is to create a "Public" API. This is simply an API that can be invoked by any client by sending HTTP requests to the appropriate Gateway endpoint. No contracts are required.

2.2.1. Public API

First, you must have an Organization created. In apiman an API is a representation of your back-end API, but managed with policies and accessible via the API Gateway instead of directly to the URL of the back-end API.

To create an API, simply navigate to the "APIs" tab for one of your Organizations and click the "New API" button. You will need to provide a Name and Initial Version for your new API and then click "Create API" at the bottom of the form.

Once you have created an API, you will need to provide some basic configuration, including:

- **API Endpoint** - the URL to your back-end API
- **API Type** - whether your API is rest or soap
- **API Content-Type** - whether your API speaks JSON or XML

Finally, on the "Plans" tab you need to check the "Make this API public" checkbox.

Once you have configured all these aspects of the API, you need to **Publish** the API to the API Gateway. This can be done by clicking the "Publish" button at the top of the page.

2.2.2. Sending Requests to your Public API

Once the API is published, you can get its URL by navigating to the "Endpoint" tab. From there you can copy the URL for this managed API. Simply send HTTP requests to this URL as you would your back-end API!

The actual managed endpoint may vary depending on the Gateway, but typically the endpoint is something like this:

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

2.3. Providing APIs`

If you want to know how to provide managed APIs via apiman, this is a fast and dirty quickstart.

2.3.1. Plans

You should already have created an Organization. Navigate to your Organization and switch to the Plans tab. From there you can create new Plans. A Plan is basically a collection of policies that will be applied to requests made to APIs being access through it. Plans will be assigned to APIs during API configuration. Plans give you the opportunity to offer access to the same API with different SLAs (e.g. different quotas).

Use the Policies tab to add zero or more Policies to the Plan. Each Policy has its own unique configuration.

Once you have created all appropriate Policies, you need to **Lock** the Plan so that it can be used by your APIs. This can be done by clicking the 'Lock' button at the top of the page. Note that a Policy must be locked for use so that API consumers can have confidence that the details of the Plan will not change after they have agreed to them.

2.3.2. APIs

After you have created at least one Plan, you can go back to the Organization page and switch to the APIs tab. From there you can create a new API. In apiman an API is a representation of your back-end API, but managed with policies and discoverable within the apiman system.

Once you have created an API, you will need to provide some basic configuration, including:

- **API Endpoint** - the URL to your back-end API
- **API Type** - whether your API is rest or soap
- **API Content-Type** - whether your API speaks JSON or XML
- **Plans** - One or more Plans

The Plans are chosen from those available within the Organization. An API can be made available through any or all of the Organization’s plans. When a client app consumes the API it will do so through one of the available plans. See the "Consuming APIs" section for details.

Once you have configured all necessary aspects of the API, you need to **Publish** the API to the API Gateway. This can be done by clicking the 'Publish' button at the top of the page.

2.4. Consuming APIs

If you're interested in consuming APIs that someone else is providing via apiman, this section should give you a good start.

2.4.1. Client Apps

In order to consume an API, you will first need to create a Client App in your Organization. The Client App **can** (but often does not) exist in the same Organization as the APIs it consumes. To create a Client App, navigate to the Client Apps tab for your Organization.

Once you have create a Client App, you can search for APIs to consume. There are a number of ways to search for APIs. Once you have located an API, you must create a Contract between your Client App and the API. A Contract is always created through one of the Plans provided by the API.

After you have created all appropriate API Contracts, you must register the Client App with the API Gateway. This can be done by clicking the 'Register' button at the top of the page.

2.4.2. Sending Requests

After the Client App is Registered, navigate to the APIs tab to see a list of all the API Contracts and to get details about each one. This tab is important because it allows you to find out the API Key assigned to the Client App. The API Key is necessary when making managed calls to the API Gateway. Additionally, this tab shows the actual API Gateway endpoint that your client must invoke in order to call the managed API.

The actual managed endpoint may vary depending on the Gateway, but typically the endpoint is something like this:

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

Requests to managed endpoints must 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**

Chapter 3. 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).

3.1. 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.

3.1.1. 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.

3.1.2. 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.

3.1.3. 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.

3.1.4. 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.

3.1.5. 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.

3.1.6. 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.

3.1.7. 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.

3.2. 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.

3.2.1. 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.

3.2.2. 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.

3.2.3. 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.

3.3. 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.

3.4. 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.

3.4.1. 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.

3.4.2. 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.

3.4.3. 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.

3.5. 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.

3.5.1. 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.

3.5.2. 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.

3.5.3. 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.

3.5.4. 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.

3.5.5. 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.

3.5.6. 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.

3.5.7. 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.

3.5.8. 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.

3.6. 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.

3.6.1. 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.

3.6.2. 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.

3.6.3. 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.

3.6.4. 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.

3.6.5. 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.

3.6.6. 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.

3.6.7. 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.

3.7. 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.

3.8. 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.

3.8.1. 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.

3.8.2. 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.

3.8.3. 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.

3.8.4. 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.

3.8.4.1. 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.

3.8.4.2. 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.

3.8.4.3. 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).

3.8.5. 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 4. 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.

4.1. 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.

4.2. 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.

4.3. 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 5. 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.

5.1. Security Policies

5.1.1. BASIC Authentication Policy

5.1.1.1. 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.

5.1.1.2. 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!)

5.1.1.3. 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"
   }
}
```

5.1.1.4. 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 = ?"
```

```
    }
```

5.1.2. Authorization Policy

5.1.2.1. 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.

5.1.2.2. 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.

5.1.2.3. Sample Configuration

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

5.1.3. SOAP Authorization Policy

5.1.3.1. 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.

5.1.3.2. Plugin

```
{
 "groupId": "io.apiman.plugins",
 "artifactId": "apiman-plugins-soap-authorization-policy",
 "version": "1.2.3.Final" // Please check for the latest version, this is just
 an example.
}
```

5.1.3.3. 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:
  - **pathPattern** (string) : Defines the path you'd like the policy to be applicable to.
  - **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.

5.1.3.4. Sample Configuration

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

5.1.4. IP Whitelist Policy

5.1.4.1. 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.

5.1.4.2. 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'.

5.1.4.3. Sample Configuration

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

5.1.5. IP Blacklist Policy

5.1.5.1. 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.

5.1.5.2. 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'.

5.1.5.3. Sample Configuration

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

5.1.6. Ignored Resources Policy

5.1.6.1. 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

This Path	Results in these Resources Being Ignored
/customers	Denies access to all customer information
/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.

5.1.6.2. 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.

5.1.6.3. Sample Configuration

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

5.1.7. Time Restricted Access Policy

5.1.7.1. 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.

5.1.7.2. 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.

5.1.7.3. 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/.*"
        }
    ]
}```


==== CORS Policy
[[policy-cors]]
===== Description
DESCRIPTION TBD HERE

===== Plugin
PLUGIN COORDINATES GO HERE

===== Configuration
CONFIGURATION TBD HERE

===== Sample Configuration
SAMPLE CONFIG TBD
```json
{
   "TBD" : "TBD"
}
```

5.1.8. HTTP Security Policy

5.1.8.1. Description

DESCRIPTION TBD HERE

5.1.8.2. Plugin

PLUGIN COORDINATES GO HERE

5.1.8.3. Configuration

CONFIGURATION TBD HERE

5.1.8.4. Sample Configuration

SAMPLE CONFIG TBD

```
{
   "TBD" : "TBD"
}
```
```

5.1.9. OAuth Policy (Keycloak)

5.1.9.1. Description

DESCRIPTION TBD HERE

5.1.9.2. Plugin

PLUGIN COORDINATES GO HERE

5.1.9.3. Configuration

CONFIGURATION TBD HERE

5.1.9.4. Sample Configuration

SAMPLE CONFIG TBD

```
{
   "TBD" : "TBD"
}
```

5.1.10. URL Whitelist Policy

5.1.10.1. 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.

5.1.10.2. Plugin

```
{
 "groupId": "io.apiman.plugins",
 "artifactId": "apiman-plugins-url-whitelist-policy",
 "version": "1.2.3.Final" // Please check for the latest version, this is just
 an example.
}
```

5.1.10.3. 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**).

5.1.10.4. Sample Configuration

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

5.2. 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

5.2.1. Rate Limiting Policy

5.2.1.1. 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.

5.2.1.2. 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.

5.2.1.3. Sample Configuration

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

5.2.2. Quota Policy

5.2.2.1. 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.

5.2.2.2. 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.

5.2.2.3. Sample Configuration

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

5.2.3. Transfer Quota Policy

5.2.3.1. 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.

5.2.3.2. 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.

5.2.3.3. Sample Configuration

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

5.3. Modification Policies

5.3.1. URL Rewriting Policy

5.3.1.1. 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.

5.3.1.2. 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.

5.3.1.3. Sample Configuration

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

}
```

5.3.2. Transformation Policy

5.3.2.1. 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.

5.3.2.2. Plugin

```
{
    "groupId": "io.apiman.plugins",
    "artifactId": "apiman-plugins-transformation-policy",
     "version": "1.2.3.Final" // Please check for the latest version, this is
 just an example.
}
```

5.3.2.3. 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'

5.3.2.4. Sample Configuration

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

5.3.3. JSONP Policy

5.3.3.1. 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.

5.3.3.2. Plugin

```
{
    "groupId": "io.apiman.plugins",
    "artifactId": "apiman-plugins-jsonp-policy",
     "version": "1.2.3.Final" // Please check for the latest version, this is
 just an example.
}
```

5.3.3.3. 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.

5.3.3.4. 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"
```

5.3.4. Simple Header Policy

5.3.4.1. Description

DESCRIPTION TBD HERE

5.3.4.2. Plugin

PLUGIN COORDINATES GO HERE

5.3.4.3. Configuration

CONFIGURATION TBD HERE

5.3.4.4. Sample Configuration

SAMPLE CONFIG TBD

```
{
   "TBD" : "TBD"
}
```

5.4. Other Policies

5.4.1. Caching Policy

5.4.1.1. 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.

5.4.1.2. 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.

5.4.1.3. Sample Configuration

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

5.4.2. Log Policy

5.4.2.1. 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.

5.4.2.2. Plugin

```
{
    "groupId": "io.apiman.plugins",
    "artifactId": "apiman-plugins-log-policy",
     "version": "1.2.3.Final" // Please check for the latest version, this is
 just an example.
}
```

5.4.2.3. 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'

5.4.2.4. Sample Configuration

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