apiman - Developer Guide

Chapter 1. Introduction

Are you interested in contributing to the development of apiman? Maybe you want to embed the project in your own solution? In either case this is the guide for you.

Chapter 2. Developer Resources

This section describes a number of resources that are useful if you wish to contribute code to apiman. It is likely also a good starting point for those wishing to provide functionality by implementing a plugin, although more information about plugins can be found in the *Plugins* section.

2.1. Source Code

The apiman source code is located in github here:

https://github.com/apiman/apiman

Source code for the apiman policies can be found here:

https://github.com/apiman/apiman-plugins

Source code for the apiman project web site is here:

https://github.com/apiman/apiman.github.io

The official apiman docker files are currently here:

https://github.com/jboss-dockerfiles/apiman

2.2. Issue Tracking

The apiman project uses JIRA to track issues such as bugs and feature requests. It's a good place to start if you're trying to figure out how to get involved!

https://issues.jboss.org/browse/APIMAN

2.3. Development Tools

We're rather IDE agnostic, so contributors should feel free to use whatever tools they feel most comfortable with. At the time of this writing, the core apiman developers primarily use Eclipse Kepler.

http://www.eclipse.org/downloads/

The core apiman project is built using maven. Currently we recommend at least version 3.0.3.

http://maven.apache.org/download.cgi

And of course you'll need to have git installed if you want to check out the code from github.

http://git-scm.com/

2.4. Building the Project

Building apiman should be a simple matter of doing a standard Maven build:

```
mvn clean install
```

This will do a full build of apiman and execute all unit tests. However, the result will not include a ready-to-run version of apiman. For that, you may want to try the following:

```
mvn clean install -Pinstall-all-wildfly9
```

This command will do a full apiman build, but will also download WildFly 9 and install apiman into it. The result will be a fully configured install of apiman running in WildFly. The location of this WildFly install will be here:

```
apiman/tools/server-all/target/wildfly-{wildfly.version}/
```

At this point you can test apiman by simply running WildFly 9 from the above location using a command something like this:

```
./bin/standalone.sh -b 0.0.0.0
```

2.5. Setting up a Development Environment

Because apiman is a fairly standard maven project, it should be relatively easy to import the project into Eclipse or IntelliJ IDEA. For now we'll leave this as an exercise for the reader.

Chapter 3. Architecture

The basic architecture of apiman is fairly straightforward. There are several WARs that make up the default apiman installation. These include:

- API Manager REST back-end (JAX-RS WAR)
- API Manager UI (AngularJS/Hawtio WAR)
- API Gateway Config (JAX-RS WAR)
- API Gateway (Servlet WAR)

The API Manager REST back-end WAR is responsible for exposing a set of REST endpoints that make up the API Manager REST interface. The API Manager UI uses this REST API directly when the user manages the various entities in the data model.

The API Manager UI is a client-only AngularJS application. Aside from authentication related activities, this WAR only contains HTML, JavaScript, and CSS. The UI uses the browser to make direct, authenticated calls to the REST endpoints exposed by the API Manager REST back-end WAR.

The API Gateway Config exposes the standard apiman Gateway REST API so that the API Gateway can be remotely configured. This is the REST API that the API Manager uses whenever a user publishes an API or registers a Client App. It is responsible for configuring the API Gateway's embedded Policy Engine.

The API Gateway is the primary runtime component of apiman and is implemented as a servlet that embeds the apiman Policy Engine. All requests to the API Gateway WAR are assumed to be intended for managed APIs previously published to it.

Chapter 4. Plugins

The easiest way to extend the functionality of apiman is by implementing an apiman plugin. This section details how this is done and what functionality can be extended or provided.

4.1. Creating a Plugin

An apiman plugin is basically a java web archive (WAR) with a little bit of extra sauce. This approach makes it very easy to build using maven, and should be quite familiar to most Java developers. Because a plugin consists of some resources files, compiled java classes, front-end resource such as HTML and javascript, and dependencies in the form of JARs, the WAR format is a natural choice.

4.1.1. The Plugin Specification File

In addition to the standard layout of a Java Web Archive, an apiman plugin must contain the following plugin specification file (which contains information about the plugin):

```
META-INF/apiman/plugin.json
```

This *plugin.json* file contains the basic meta-data that describes the plugin, and should be of the following format:

```
{
  "frameworkVersion" : 1.0,
  "name" : "Plugin Name",
  "description" : "A plugin description goes here.",
  "version" : "3.1.9"
}
```

- **frameworkVersion**: Indicates the apiman plugin framework version this plugin is compatible with - this should simply be 1.0 for now (reserved for future use)
- **name**: The name of the plugin.
- **description**: The description of the plugin.
- **version**: The plugin version.

If this *plugin.json* file is missing from the plugin archive, then the plugin will fail to load.

4.1.2. Using Maven to Create a Plugin

One benefit of using WAR as the format of an apiman plugin is that plugins can easily be created using Maven. This section will describe how this can be done. Note that you can use the following simple plugin as a reference if you prefer:

https://github.com/apiman/apiman-plugins/tree/master/noop-policy

In order to create an apiman plugin using maven, simply create a new maven project and set its *packaging* type to **war**.

```
<packaging>war</packaging>
```

Next, obviously feel free to include any dependencies you might need:

```
<dependencies>
  <!-- apiman dependencies (must be excluded from the WAR) -->
  <dependency>
    <groupId>io.apiman</groupId>
    <artifactId>apiman-gateway-engine-core</artifactId>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

You'll want to make any apiman dependencies provided so that there aren't any classloading conflicts when executing your code.

Finally, we recommend that you put your plugin.json file in the following location in your maven project:

```
src/main/apiman
```

Of course, any resoures in that location are not automatially included in the final WAR, so you should add the following markup to your pom.xml:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-war-plugin</artifactId>
      <configuration>
        <failOnMissingWebXml>false</failOnMissingWebXml>
        <webResources>
          <resource>
            <directory>src/main/apiman</directory>
            <targetPath>META-INF/apiman</targetPath>
            <filtering>true</filtering>
          </resource>
        </webResources>
      </configuration>
    </plugin>
```

```
  </plugins>
</build>
```

This markup will ensure that resources in the **src/main/apiman** folder will be included in the correct location in the WAR. Also note that resource filtering is enabled, which will make it easier to maintain your **plugin.json** file:

```
{
  "frameworkVersion" : 1.0,
  "name" : "My Plugin Name",
  "description" : "My plugin description.",
  "version" : "${project.version}"
}
```

Note that the *version* of the plugin is set to **${project.version}**, which will get automatically changed to the version of your maven project at build time.

4.1.3. Making Your Plugin Available to apiman

Plugins are identified by their Maven coordinates (groupId, artifactId, version, classifier, type). Note that the classifier and type are optional. If the type is not specified when loading a plugin, apiman will assume *war*.

When loading a plugin for use, apiman will first check for the plugin in the local user's .m2 directory. This is useful when running apiman during development, but is unlikely to be available in a production environment. If the plugin cannot be found locally, apiman will attempt to download it from a remote repository such as Maven Central.

Tip

You can configure additional remote repositories when you set up apiman. Please refer to the Installation Guide for details.

This all means that when testing your plugin locally, you can simply use maven to install it into your local .m2 directory and then ask apiman to load it. In production, the plugin will need to be available from a remote maven repository.

4.2. Contributing a Policy

Now that you know how to create an apiman plugin, you might be wondering what you can actually do with it! The most important purpose of a plugin is to provide additional **Policies** that can be used when configuring Plans, APIs, and Client Apps in apiman. Although apiman comes with a set of useful built-in policies, it is often necessary for users to provide their own custom policies. The best way to do that is to create a plugin that provides such policies.

In order to provide a custom policy from a plugin, several things are needed:

- An implementation of IPolicy (Java code)
- A policy definition (JSON file)
- An optional policy configuration form that the API Manager UI will present to the user when configuring the policy

The next few sections explain each of these elements further, but note that they are all included in the apiman plugin WAR.

4.2.1. Policy Implementation

A policy implementation is the java code that is executed by the API Gateway when a managed API request is made. This is the bread and butter of the API Gateway; its primary purpose. For each request, the API Gateway creates a chain of policies that must be executed before proxying the request to the back-end API implementation. Each of the policies in that chain is an implementation of the *IPolicy* interface.

4.2.1.1. Standard IPolicy

All policies must implement the `IPolicy` interface, consisting of several methods.

The `apply` method with `ApiRequest` is called during the request phase, and the `apply` with `ApiResponse` during the response phase:

```
void apply(ApiRequest request, IPolicyContext context, Object config, IPolicyChain<ApiRequest>

void apply(ApiResponse response, IPolicyContext context, Object config, IPolicyChain<ApiRespons
```

The API objects, respectively, provide abstracted representations of the head of a request and response for a given conversation. These can be modified in any manner the implementor sees fit.

Tip

Policy instances are stateless, so it is not a good idea to use fields for any reason. The IPolicyContext can be used to pass information from the request phase to the response phase. Any state that must span multiple requests will need to use one of the policy components described in the **Provided Components** section.

```
Object parseConfiguration(String jsonConfiguration) throws ConfigurationParseException;
```

The final `IPolicy` method is used to parse JSON configuration into an arbitrary object configuration which will be passed in in its parsed form to `doApply`, where the implementor may cast it their native configuration object. This method will be invoked for each unique configuration of the policy.

For mroe information about policy configuration, see the **Policy Configuration** section below.

4.2.1.1.1. Indicating Successes

If a policy determines that the conversation can continue, `chain.doApply` should be signalled. Any modifications you wish to pass onto the next policy should be completed and included in the invocation.

4.2.1.1.2. Indicating Failures

If it is determined that a conversation should be interrupted for governance reasons (i.e. according to business logic and not exceptional), then `chain.doFailure` should be signalled. A useful `PolicyFailure` should be provided, which allows gateways to respond in a sensible way to the requestor.

Tip

The platform's `IPolicyFailureFactoryComponent` can be used to generate failures. See the **Provided Components** section for more details on this component.

4.2.1.1.3. Handling Exceptions

As a factor of the asynchronous nature of apiman, any exceptions that may occur during the operation of a policy should be caught and explicitly handed to `chain.doError`. If exceptions are left uncaught, then it is possible that they will be lost.

4.2.1.2. IData Policy

Whilst standard policies are concerned only with the head of the conversation, it is also possible for policies to access and manipulate the body in transit. A data policy must implement the `IDataPolicy` interface.

Warning

Handling of data streams is a performance sensitive area, implementors should strive to be as efficient as possible and avoid any unnecessary interactions with the stream.

The `getRequestDataHandler` and `getResponseDataHandler` methods are the data corollaries of `apply`. Implementors must return `IReadWriteStream` streams, which apiman uses to write data chunks into policies, and the policies write data to subsequent policies:

```
IReadWriteStream<ApiRequest> getRequestDataHandler(ApiRequest request, IPolicyContext context);

IReadWriteStream<ApiResponse> getResponseDataHandler(ApiResponse response, IPolicyContext conte
```

Important

Do not return an `IApimanBuffer` with a different native type than you received. Use assign and append patterns instead.

Implementors must explicitly hand each chunk onto apiman when they are finished interacting with it. A convenient way to achieve this is via `AbstractStream<H>`:

```
@Override
public IReadWriteStream<ApiRequest> getRequestDataHandler(final ApiRequest request, final IPoli
  return new AbstractStream<ApiRequest>() {
    @Override
    public void write(IApimanBuffer chunk) {
      // Mutate chunk by appending a string.
      chunk.append("my modification");
      // We're finished: write the chunk back to apiman
      // using super.write().
      super.write(chunk);
    }

    @Override
    public void end() {
      // End of stream signalled, do cleanup, etc.
      super.end();
    }
  };
}
```

Important

Do not mutate an `IApimanBuffer` once handed over.

The request or response body will not begin streaming before the corresponding `doApply` has been called, however, it is still possible to interrupt the conversation during the streaming phase by signalling `doFailure` or `doError`.

4.2.1.3. Performance Considerations

Policies are amongst the most impactful elements of the system for performance. To minimise the impact of a policy implementors may wish to follow these guidelines:

- Maintain as little state within a policy instance as possible.
- Call `doApply`, `doFailure` or `doError` as soon as possible.
- Data policies should interact with the data stream as efficiently as possible and prefer mutating in-place (especially with small changes).
- If you are contributing a policy to apiman: implement any long-running tasks asynchronously (e.g. database calls); **do not** block the main thread (e.g. blocking futures, wait, sleep); use asynchronous techniques to interact with the outside world, such as callbacks.

4.2.1.4. Dependencies

Typically a policy implementation should minimize the number of third party libraries it depends on, but often times this is unavoidable. Plugins are isolated from one another, so it is a simple matter of including any required dependencies inside the plugin's WAR archive in the standard location of:

```
WEB-INF/lib
```

Tip

You should make sure that any apiman dependencies you use (for example the apiman core module that contains the IPlugin and other necessary interfaces) are marked as *provided* in your maven project so that they are not included in the plugin archive.

4.2.1.5. Provided Components

All policy implementations have access to various resources at runtime. These resources are primarily accessed through the **IPolicyContext** object that is passed to the policy when it is executed. Along with the ability to set conversation-level attributes, the policy context is how you access Policy Components.

A Policy Component is simply a runtime component that a policy implementation may find useful. To access a component, use the *getComponent* method found on the policy context, passing it the interface of the component you wish to use. The following components are available:

Component Name	Description
IPolicyFailureFactoryComponent	Used to create a policy failure that is needed to call *doFailure* on the policy chain (indicating that the policy failed).
ISharedStateComponent	Used to share state information across the conversation boundary.
IHttpClientComponent	Allows HTTP requests to be made from within a policy.
IRateLimiterComponent	Supports standard quota/rate limiting behavior, maintaining the current number of requests.
ILdapComponent	Provides the ability to authenticate with an LDAP server and execute simple queries against it.
IJdbcComponent	Enables querying of JDBC-capable datasources.

All of the components have asynchronous APIs in order to better support the runtime philosophy in the API Gateway.

Tip

For more information about each component, see its javadoc.

4.2.2. Policy Definition

The policy implementation is what allows the API Gateway to execute the policy at runtime. But how does the API Manager know about the policy so that users can add it to a Plan, API, or Client App from within the User Interface? The answer is that the plugin must also include a Policy Definition JSON file for each policy it is providing.

A plugin definition is a JSON file that must be located within the plugin archive here:

```
META-INF/apiman/policyDefs
```

The plugin definition file takes the following form:

```
{
  "id" : "policy_name",
  "name" : "Policy Name",
  "description" : "A useful description of what the policy does.",
        "policyImpl"    :    "plugin:${project.groupId}:${project.artifactId}:
${project.version}:${project.packaging}/com.example.plugins.MyFirstPolicy",
  "icon" : "document",
  "formType" : "JsonSchema",
  "form" : "schemas/policy_name.schema"
}
```

- **id**: The unique id of the policy.

- **name**: The name of the policy.
- **description**: The description of the policy.
- **policyImpl**: Identifies the java class that implements the policy.
- **icon**: The icon to use when displaying the policy in the UI (name of a Font Awesome icon).
- **formType**: The type of form to use in the UI when configuring an instance of the policy. See the Policy Configuration section below for details. Valid values: *Default*, *JsonSchema*
- **form**: (*optional*) Path to a UI form that should be used when configuring an instance of the policy. See the Policy Configuration section below for details.

The most important thing to get right in this file is probably the *policyImpl*. This is the information that the API Manager will use when it tries to instantiate the policy implementation at runtime. For policies that come from plugins, the format of the *policyImpl* is:

```
plugin:{pluginGroupId}:{pluginArtifactId}:{pluginVersion}:{pluginType}/
{fullyQualifiedClassname}
```

An example of what this string might look like if you cracked open a valid apiman plugin and had a peek at one of its policy definition files is:

```
plugin:io.apiman.plugins:apiman-plugins-example:6.3.3.Final:war/
io.apiman.plugins.example.ExamplePolicy
```

When building your plugin using the recommended maven configuration documented in the **Using Maven to Create a Plugin** section, it is extremely convenient to simply let Maven set the values for you:

```
plugin:${project.groupId}:${project.artifactId}:${project.version}:
${project.packaging}/com.example.plugins.ExamplePolicy
```

4.2.3. Policy Configuration Form

You may be wondering how configuration information specific to a Plan, API, or Client App is managed. Since the same policy implementation instance is used for all requests, unique configuration appropriate to a particular request must be passed to the policy implementation when it is executed. This configuration is created in the API Manager user interface when adding the policy to a Plan, API, or Client App.

Policy configuration takes the form of string data that is ultimately included when publishing an API to the API Gateway. That string data is parsed into a Java object via the *parseConfiguration* on the **IPolicy** interface and then passed to the policy during execution.

The string data is created in the API Manager user interface, either by interacting with a Policy Configuration Form contributed by the plugin, or (if no form is included in the plugin) by a default configuration form (a simple text area).

4.2.3.1. Default Policy Configuration

If the policy definition indicates that the configuration form type is **Default**, then it is up to the UI to determine how to display configuration information. For the policies provided by apiman itself, there are UI forms provided. If the policy is contributed from a plugin, then the UI has no way to know the format of the configuration data. In this case, a simple TextArea is presented to the user.

Warning

This approach is clearly not recommended, because users will likely have no idea what to enter into the TextArea presented to them.

4.2.3.2. JSON Schema Policy Configuration

Alternatively, the policy definition can specify a *JSON Schema* [http://json-schema.org/] in the policy definition JSON file. For example, the policy definition might include the following:

```
  "formType" : "JsonSchema",
  "form" : "schemas/policy_name.schema"
```

In this case, apiman will look for a file inside the plugin artifact in the following location:

```
META-INF/apiman/policyDefs/schemas/policy_name.schema
```

The file in this location must be a JSON Schema file, which describes the JSON format of the configuration data expected by the policy implementation. The UI will use this JSON schema to generate an appropriate UI form that can edit the JSON configuration data needed by the policy implementation.

Perhaps it's best if we have an example. The following illustrates a policy contributed from a plugin, its JSON Schema file, the resulting form displayed in the UI, and the configuration data format that will be passed to the policy implementation at runtime.

META-INF/apiman/policyDefs/my-policy.json.

```
{
  "id" : "my-policy",
  "name" : "My First Policy",
  "description" : "A policy with custom configuration!",
        "policyImpl"    :    "plugin:${project.groupId}:${project.artifactId}:
${project.version}:${project.packaging}/
io.apiman.plugins.config_policy.ConfigPolicy",
  "icon" : "pie-chart",
  "formType" : "JsonSchema",
```

```
  "templates" : [
    {
      "language": null,
      "template": "Set policy with @{property1} and @{property2}!"
    }
  ],
  "form" : "schemas/config-policyDef.schema"
}
```

Tip

The templates *language* field will support other languages in future, but for now is null (i.e. single-language only). The template field itself is *MVEL* [https://github.com/mvel/mvel] (Orb tag syntax), and displays in the UI after a plugin has been selected by a user.

META-INF/apiman/policyDefs/schemas/my-policy.schema.

```
{
  "title" : "Configure My Policy",
  "description" : "Configure all of the necessary properties used by my policy.",
  "type" : "object",
  "properties": {
    "property1": {
      "title" : "Property 1",
      "type" : "string",
      "minLength" : 1,
      "maxLength" : 64
      },
    "property2": {
      "title" : "Property 2",
      "type" : "string",
      "minLength" : 1,
      "maxLength" : 64
    }
  }
}
```

Generated UI Form.

JSON Configuration Data Format.

```
{
```

```
  "property1" : "USER_DATA_1",
  "property2" : "USER_DATA_2"
}
```

Tip

You can easily consume the JSON configuration data above in your policy implementation by having your policy implementation Java class extend the `AbstractMappedPolicy` base class provided by apiman (in the *apiman-gateway-engine-policies* module) and creating a simple Java Bean to hold the JSON configuration data.

First, here is the java bean used to (un)marshal the JSON configuration data.

```
public class MyConfigBean implements Serializable {

  private static final long serialVersionUID = 6834865169105914 77L;

  private String property1;
  private String property2;

  /**
   * Constructor.
   */
  public MyConfigBean() {
  }

  public String getProperty1() {
    return property1;
  }

  public void setProperty1(String property1) {
    this.property1 = property1;
  }

  public String getProperty2() {
    return property2;
  }

  public void setProperty2(String property2) {
    this.property2 = property2;
  }

}
```

Now have a look at how to use that class when extending the `AbstractMappedPolicy`.

```
public class MyPolicy extends AbstractMappedPolicy<MyConfigBean> {

  /**
   * Constructor.
   */
  public MyPolicy() {
  }

  @Override
  protected Class<MyConfigBean> getConfigurationClass() {
    return MyConfigBean.class;
  }

  @Override
  protected void doApply(ApiRequest request, IPolicyContext context, MyConfigBean config, IPoli
    // Do something with MyConfigBean here?  It has all the configuration data!
    super.doApply(request, context, My, chain);
  }

  @Override
  protected void doApply(ApiResponse response, IPolicyContext context, MyConfigBean config, IPo
    // Do something with MyConfigBean here?  It has all the configuration data!
    super.doApply(response, context, config, chain);
  }

}
```

4.2.3.3. JSON Schema Policy Configuration SDK

If you are creating a non-trivial JSON Schema (more than just a couple of simple fields) it can be difficult to get it right without a few iterations. For this reason, we have created a simple "SDK" to help you create your JSON Schema quickly. The SDK can be found in the apiman github repository at the following location:

```
manager/ui/war/src/main/sdk/json-schema.html
```

If you have the apiman source code checked out, you can simply open that file in your browser and start using it to author a custom JSON Schema.

Alternatively you can use "rawgit" and just go straight to the following URL:

http://rawgit.com/apiman/apiman/master/manager/ui/war/src/main/sdk/json-schema.html

The SDK provides a way to edit your JSON schema and then see how that schema will look in the apiman UI, as well as the format that the policy configuration data will ultimately be in when it is sent to your policy at runtime.

Tip

Once you have the JSON Schema finalized, you could also use the online *jsonschema2pojo* [http://www.jsonschema2pojo.org/] tool to generate a good starting point for a Java Bean that can be used to marshal/unmarshal your policy's configuration data at runtime. See the discussion about AbstractMappedPolicy above for additional information.

4.3. Unit Testing a Plugin Policy

While it is quite simple to create a custom policy for apiman, you may be wondering the best way to unit test your implementation. Fortunately we have made this extremely easy by including an easy-to-use Policy Testing junit framework. Once you have followed the instructions above to create your custom policy, refer to this section to learn how to test it using junit.

4.3.1. Import the Framework (Maven Dependency)

The first thing you will need is to include the appropriate maven dependencies in your project's pom.xml file. There is a single additional dependency that you will need (make sure to import it using the *test* maven scope):

```
<dependency>
   <groupId>io.apiman</groupId>
   <artifactId>apiman-test-policies</artifactId>
   <version>1.1.2-SNAPSHOT</version>
   <scope>test</scope>
</dependency>
```

4.3.2. Create and Annotate a JUnit Test Case

Once you have imported the appropriate dependency, you can go ahead and create a JUnit test case. The only additional thing you need is to annotate your test case appropriately and make sure your test case Java class extends the framework's *ApimanPolicyTest* base class.

The following annotations can then be added to your test:

- @TestingPolicy(<classname>) - indicates which of your policy implementations you wish to test
- @Configuration("<custom_policy_configuration_data>") - specifies the policy configuration to use for the test

The @TestingPolicy annotation is always placed at the class level, but the @Configuration annotation can either be global or specified at the test method level.

These annotations tell the apiman Policy Testing framework **what** policy you want to test and the policy configuration you want to use when testsing, but you still need to actually send requests to a "API". This is done using the "send(PolicyTestReqest)" method defined by the base class. The send() method allows you to send a request (that you build) to the mock back-end API governed by your policy. By default the mock back-end API is a simple "echo" API that responds to all requests with a JSON payload describing the request it received (more on how to override this default functionality later).

The send() method requires that you create and pass to it a valid PolicyTestRequest object. This can be created using the PolicyTestRequest.build() method. You can set the request's type, resource path, request headers, and body. If the request is successful, then a PolicyTestResponse object will be returned and you can perform assertions on it. If there is a policy failure, then the send() method will throw a PolicyFailureError.

Here is a full example of everything working together:

```
@TestingPolicy(CustomPolicy.class)
public class CustomPolicyTest extends ApimanPolicyTest {

    @Test
    @Configuration("{}")
    public void testGet() throws Throwable {
      // Send a test HTTP request to the API (resulting in executing the policy).
PolicyTestResponse response = send(PolicyTestRequest.build(PolicyTestRequestType.GET,"/
some/resource")
                .header("X-Test-Name", "testGet"));

        // Now do some assertions on the result!
        Assert.assertEquals(200, response.code());
        EchoResponse entity = response.entity(EchoResponse.class);
        Assert.assertEquals("GET", entity.getMethod());
        Assert.assertEquals("/some/resource", entity.getResource());
        Assert.assertEquals("testGet", entity.getHeaders().get("X-Test-Name"));
        // Assert the request header that was added by the policy
            Assert.assertEquals("Hello World", entity.getHeaders().get("X-MTP-
Header"));
        // Assert the response header was added by the policy
         Assert.assertEquals("Goodbye World", response.header("X-MTP-Response-
Header"));
    }

}
```

4.3.3. Providing a Custom Back-End API Mock

Sometimes the echo API is not sufficient when testing your custom policy. Perhaps the custom policy is more tightly coupled to the API it is protecting. In this case you may want to provide your own custom back-end API mock implementation. This can be done by simply annotating either the class or an individual test method with @BackEndApi. If you do this then you must supply the annotation with a class that implements the IPolicyTestBackEndApi interface. Here is an example of what this might look like in a test:

```
@TestingPolicy(CustomPolicy.class)
public class CustomPolicyTest extends ApimanPolicyTest {

    @Test
    @Configuration("{}")
    @BackEndApi(MyCustomBackEndApiImpl.class)
    public void testGetWithCustomBackEndSvc() throws Throwable {
      // Send a test HTTP request to the API (resulting in executing the policy).
PolicyTestResponse response = send(PolicyTestRequest.build(PolicyTestRequestType.GET,"/
some/resource")
                .header("X-Test-Name", "testGet"));

        // Now do some assertions on the result!
        MyCustomBackEndApiResponseBean entity = response.entity(MyCustomBackEndApiResponseBean.
    }

}
```

In this example everything works as it did before, but instead of responding with an Echo Response the send() method will return with a custom response (as created and returned by the provided custom back-end API implementation).

4.4. Using a Plugin Policy

Once you have built and unit tested your plugin policy, you will most likely want to actually use the policy in apiman. This can be done by adding the plugin to apiman via the Plugin Management UI in the API Manager user interface.

Tip

The Plugin Management UI is restricted to admin users of the API Manager.

For more information about how to use the Plugin Management UI, please see the apiman User Guide.

4.4.1. Iterating a Plugin Policy

When developing a custom plugin policy, it can be cumbersome to have to uninstall and reinstall the plugin every time you make a change. Hopefully, unit testing will help you quickly iterate your plugin policy implementation, but there are times when testing in a live environment is necessary.

At runtime, the API Gateway installs plugins from the local `.m2` directory. If the plugin is not found there, only then will apiman attempt to find and download the plugin from the configured remote maven repositories. Typically, the API Gateway will load and cache the plugin the first time it is used. However, if your plugin **version** ends with "-SNAPSHOT", then apiman will reload it every time it is used.

As a result, you can quickly iterate changes to your plugin policy using a live apiman environment by doing the following:

1. Ensure that you are testing a "-SNAPSHOT" version of your custom plugin policy
2. Configure the policy on one or more API
3. Publish the API(s) to the API Gateway
4. Send an HTTP request to an API that uses your custom policy
5. Make a change to your Policy implementation
6. Rebuild your plugin and "install" it into your .m2 directory (do not change the version)
7. Repeat starting at #4

Because the version of your plugin ends with "-SNAPSHOT", the API Gateway will not cache it, but instead will reload it each time you do step #4. This allows you to quickly make changes, rebuild, and re-test with a minimum of additional steps.

Tip

Don’t use -SNAPSHOT versions of plugins in production as the lack of policy caching will be a significant performance problem.

4.4.2. Uninstalling a Plugin

Again, you can use the Plugin Management UI to uninstall a plugin. Please note that when you do this, any API that is already configured to **use** the plugin will continue to work. If you wish for an API to no longer use a plugin policy, you must remove the policy from the API as a separate step.

4.4.3. Upgrading a Plugin

Often times new versions of a plugin may become available. When this happens you can use the Plugin Management UI to upgrade a plugin to a newer version. Please note that this will **not**

automatically upgrade any API using the older version of the plugin. Instead, to upgrade an API to use the newer plugin policy, you will need to remove the old policy configuration and re-add it. This will cause the API to pick up the newer version. Of course, any **new** APIs will always use the new version.

4.5. Contributing a Core Component

In addition to policies, the apiman plugin framework allows developers to provide custom implementations of core apiman components. What does this mean? Apiman is composed of a number of different core components, all working together to provide API Management functionality. Both the API Gateway and the API Manager have core components that can be customized by providing new implementations via plugins.

Some examples of API Manager components include (but are not limited to):

- Storage Component
- Query Component
- IDM Component
- Metrics Accessor (consumes metrics data recorded by the API Gateway at runtime)

Additionally, some examples of API Gateway components include:

- Configuration Registry
- Rate Limiting Component
- Metrics Emitter (records metrics data for each request)

By default, the apiman quickstart uses default values for all of these, resulting in a stable, working system with the following characteristics:

- Stores API Manager data in a JDBC database
- Records and queries metrics data via Elasticsearch
- Stores Gateway configuration information in Elasticsearch
- Uses Elasticsearch to share rate limiting state across gateway nodes

However, if you wish to provide a custom implementation of something, you can implement the appropriate Java interface for the correct component, bundle the implementation up into a plugin, and then tell apiman to use yours instead of the default.

4.5.1. Implementing a Custom Core Component

The procedure for creating a plugin to hold your custom component is exactly the same as already described in the **Creating a Plugin** section above. Once you have created your plugin, including

a custom implementation of a core component is simply a matter of creating a Java class that implements the appropriate component interface.

Let's try an example.

By default, apiman stores API Gateway configuration in Elasticsearch. The component responsible for this is called ESRegistry, and it implements this interface:

```
package io.apiman.gateway.engine;

public interface IRegistry {

    public void getContract(ApiRequest request, IAsyncResultHandler<ApiContract> handler);

    public void publishApi(Api api, IAsyncResultHandler<Void> handler);

    public void retireApi(Api api, IAsyncResultHandler<Void> handler);

   public void registerClient(Client client, IAsyncResultHandler<Void> handler);

    public void unregisterClient(Client client, IAsyncResultHandler<Void> handler);

    public void getApi(String organizationId, String apiId, String apiVersion, IAsyncResultHand

}
```

Perhaps you'd rather store the API Gateway configuration information into mongodb instead of Elasticsearch. Since we don't support a mongodb registry, you would need to implement your own and contribute it via a plugin. Simple create a new plugin and include in it the following Java class:

```
package org.example.apiman.plugins;

public class MongoDbRegistry implements IRegistry {

    public MongoDbRegistry(Map<String, String> config) {
        // TODO consume any config params - these come from apiman.properties
    }

    public void getContract(ApiRequest request, IAsyncResultHandler<ApiContract> handler) {
        // TODO implement mongodb specific logic here
    }

    public void publishApi(Api api, IAsyncResultHandler<Void> handler) {
        // TODO implement mongodb specific logic here
    }

    public void retireApi(Api api, IAsyncResultHandler<Void> handler) {
```

```
        // TODO implement mongodb specific logic here
    }

   public void registerClient(Client client, IAsyncResultHandler<Void> handler) {
        // TODO implement mongodb specific logic here
    }

    public void unregisterClient(Client client, IAsyncResultHandler<Void> handler) {
        // TODO implement mongodb specific logic here
    }

    public void getApi(String organizationId, String apiId, String apiVersion, IAsyncResultHand
        // TODO implement mongodb specific logic here
    }

}
```

Tip

While optional, it is often useful to provide a constructor that takes a map of configuration params. These values comes from the **apiman.properties** and is an arbitrary set of keys/values. It can be extremely helpful when, for example, configuring the mongodb connection information.

4.5.2. Enabling Your Custom Component

Now that you have a custom component built and included in a plugin, you will need to make sure that the plugin is available to your server. You can do this by deploying the plugin artifact to a maven repository and then making that repository available to apiman by adding its URL to the following property in **apiman.properties**:

```
apiman.plugins.repositories=http://repository.jboss.org/nexus/content/
groups/public/
```

Simply add your organization's maven repository to that (the value can be a comma separated list of URLs).

Alternatively, you can make sure your plugin is installed in the ".m2" directory on the machine that is running your server. Obviously you can use "mvn install" to accomplish this.

Next, simply enable the custom component implementation by updating your **apiman.properties** file like this (for example):

```
apiman-gateway.registry=plugin:GROUP_ID:ARTIFACT_ID:VERSION/
org.example.apiman.plugins.MongoDbRegistry
```

```
apiman-gateway.registry.mongo.host=localhost
apiman-gateway.registry.mongo.port=27017
apiman-gateway.registry.mongo.username=sa
apiman-gateway.registry.mongo.password=sa123!
apiman-gateway.registry.mongo.database=apiman
```

The most important part above is the format for the registry itself. It might look something like this:

```
apiman-gateway.registry=plugin:org.example.apiman-plugins:plugin-
mongodb:1.0.0.Final/org.example.apiman.plugins.MongoDbRegistry
```

Finally, the set of properties prefixed with "apiman-gateway.registry" will be processed and passed to your **MongoDbRegistry** class's **Map** constructor if one is provided. The map that is passed to the constructor will contain the following:

```
mongo.host=localhost
mongo.port=27017
mongo.username=sa
mongo.password=sa123!
mongo.database=apiman
```

4.5.3. Core Component Customization Points

This section lists all/most of the available customization points available within apiman. These represent all of the core apiman components that can be replaced by custom implementations provided via plugins.

4.5.3.1. API Manager Components

Component Interface	Description
io.apiman.manager.api.core.INewUserBootstrapper	Allows customizing users upon first login (e.g. create an org for the user).
io.apiman.manager.api.core.IStorage	Primary storage of all API Manager data.
io.apiman.manager.api.core.IStorageQuery	Allows querying of the API Manager data.
io.apiman.manager.api.core.IMetricsAccessor	Used by the API Manager to query Metrics data collected by the API Gateway.
io.apiman.manager.api.core.IApiKeyGenerator	Used to create an API Key for each created API Contract.
io.apiman.common.util.crypt.IDataEncrypter	Used primarily by the storage layer to encrypt potentially sensitive data prior to storing it.
io.apiman.manager.api.core.IApiCatalog	Provides access to external APIs which users may wish to import.

4.5.3.1.1. io.apiman.manager.api.core.INewUserBootstrapper Example Configuration

```
apiman-manager.user-
bootstrapper.type=plugin:com.example.groupId:artifactId:1.0.Final/
com.example.apiman.FooUserBootstrapperImpl
apiman-manager.user-bootstrapper.foo1=value-1
apiman-manager.user-bootstrapper.foo2=value-2
```

4.5.3.1.2. io.apiman.manager.api.core.IStorage Example Configuration

```
apiman-manager.storage.type=plugin:com.example.groupId:artifactId:1.0.Final/
com.example.apiman.FooStorageImpl
apiman-manager.storage.foo1=value-1
apiman-manager.storage.foo2=value-2
```

4.5.3.1.3. io.apiman.manager.api.core.IStorageQuery Example Configuration

```
apiman-manager.storage-
query.type=plugin:com.example.groupId:artifactId:1.0.Final/
com.example.apiman.FooStorageQueryImpl
apiman-manager.storage-query.foo1=value-1
apiman-manager.storage-query.foo2=value-2
```

Tip

If your custom IStorage implementation **also** implements IStorageQuery, then it will be used instead of trying to create a separate instance of IStorageQuery.

4.5.3.1.4. io.apiman.manager.api.core.IMetricsAccessor Example Configuration

```
apiman-manager.metrics.type=plugin:com.example.groupId:artifactId:1.0.Final/
com.example.apiman.FooMetricsAccessorImpl
apiman-manager.metrics.foo1=value-1
apiman-manager.metrics.foo2=value-2
```

4.5.3.1.5. io.apiman.manager.api.core.IApiKeyGenerator Example Configuration

```
apiman-manager.api-
keys.generator.type=plugin:com.example.groupId:artifactId:1.0.Final/
com.example.apiman.FooApiKeyGeneratorImpl
```

```
apiman-manager.api-keys.generator.foo1=value-1
apiman-manager.api-keys.generator.foo2=value-2
```

4.5.3.1.6. io.apiman.common.util.crypt.IDataEncrypter Example Configuration

```
apiman.encrypter.type=plugin:com.example.groupId:artifactId:1.0.Final/
com.example.apiman.FooDataEncrypter
apiman.encrypter.foo1=value-1
apiman.encrypter.foo2=value-2
```

4.5.3.1.7. io.apiman.manager.api.core.IApiCatalog Example Configuration

```
apiman-manager.api-
catalog.type=plugin:com.example.groupId:artifactId:1.0.Final/
com.example.apiman.FooApiCatalogImpl
apiman-manager.api-catalog.foo1=value-1
apiman-manager.api-catalog.foo2=value-2
```

4.5.3.2. API Gateway Components

Component Interface	Description
io.apiman.gateway.engine.IRegistry	Stores gateway configuration data (e.g. published APIs).
io.apiman.common.util.crypt.IDataEncrypter	Used to encrypt potentially sensitive data prior to storing in the registry.
io.apiman.gateway.engine.IConnectorFactory	Creates connectors to back-end APIs based on API meta-information.
io.apiman.gateway.engine.policy.IPolicyFactory	Loads policy implementations (from plugins or else internally).
io.apiman.gateway.engine.IPolicyFailureWriter	Writes a policy failure to the HTTP response.
io.apiman.gateway.engine.IPolicyErrorWriter	Writes a policy error to the HTTP response.
io.apiman.gateway.engine.components.IBufferFactoryComponent	Creates ApimanBuffer (typically this is provided by the platform support).
io.apiman.gateway.engine.components.ICacheStoreComponent	Allows storing data into a cache store.
io.apiman.gateway.engine.components.IHttpClientComponent	Creates HTTP clients for use in policies.
io.apiman.gateway.engine.components.IJdbcComponent	Async component used to perform JDBC operations in policies.
io.apiman.gateway.engine.components.ILdapComponent	Async component used to perform LDAP operations in policies.
io.apiman.gateway.engine.components.IPeriodicComponent	Creates timers (for use by policies).

Component Interface	Description
io.apiman.gateway.engine.components.IPolicyFactoryComponent	Create policy implementations (for use by policies).
io.apiman.gateway.engine.components.IRateLimiterComponent	Used by the rate limiting and quota policies.
io.apiman.gateway.engine.components.ISharedStateComponent	General purpose component to share state across policy invokations.

4.5.3.2.1. io.apiman.gateway.engine.IRegistry Example Configuration

```
apiman-gateway.registry=plugin:com.example.groupId:artifactId:1.0.Final/
com.example.apiman.FooRegistryImpl
apiman-gateway.registry.foo1=value-1
```

4.5.3.2.2. io.apiman.common.util.crypt.IDataEncrypter Example Configuration

```
apiman.encrypter.type=plugin:com.example.groupId:artifactId:1.0.Final/
com.example.apiman.FooDataEncrypter
apiman.encrypter.foo1=value-1
apiman.encrypter.foo2=value-2
```

4.5.3.2.3. io.apiman.gateway.engine.IConnectorFactory Example Configuration

```
apiman-gateway.connector-
factory=plugin:com.example.groupId:artifactId:1.0.Final/
com.example.apiman.FooConnectorFactoryImpl
apiman-gateway.connector-factory.foo1=value-1
apiman-gateway.connector-factory.foo2=value-2
```

4.5.3.2.4. io.apiman.gateway.engine.policy.IPolicyFactory Example Configuration

```
apiman-gateway.policy-
factory=plugin:com.example.groupId:artifactId:1.0.Final/
com.example.apiman.FooPolicyFactoryImpl
apiman-gateway.policy-factory.foo1=value-1
apiman-gateway.policy-factory.foo2=value-2
```

Note: there is rarely a reason to provide a custom policy factory.

4.5.3.2.5. io.apiman.gateway.engine.IPolicyFailureWriter Example Configuration

```
apiman-gateway.writers.policy-
failure=plugin:com.example.groupId:artifactId:1.0.Final/
com.example.apiman.FooPolicyFailureWriterImpl
apiman-gateway.writers.policy-failure.foo1=value-1
apiman-gateway.writers.policy-failure.foo2=value-2
```

4.5.3.2.6. io.apiman.gateway.engine.IPolicyErrorWriter Example Configuration

```
apiman-gateway.writers.policy-
error=plugin:com.example.groupId:artifactId:1.0.Final/
com.example.apiman.FooPolicyErrorWriterImpl
apiman-gateway.writers.policy-error.foo1=value-1
apiman-gateway.writers.policy-error.foo2=value-2
```

4.5.3.2.7. io.apiman.gateway.engine.components.IBufferFactoryComponent Example Configuration

```
apiman-
gateway.components.IBufferFactoryComponent=plugin:com.example.groupId:artifactId:1.0.Final/
com.example.apiman.FooBufferFactoryComponentImpl
apiman-gateway.components.IBufferFactoryComponent.foo1=value-1
apiman-gateway.components.IBufferFactoryComponent.foo2=value-2
```

Note: typically the buffer factory is specific to the platform. For example, there is a buffer factory used when the API Gateway is running in EAP or WildFly. There is a different buffer factory used when the API Gateway is running in vert.x. There is typically not another reason to override this.

4.5.3.2.8. io.apiman.gateway.engine.components.ICacheStoreComponent Example Configuration

```
apiman-
gateway.components.ICacheStoreComponent=plugin:com.example.groupId:artifactId:1.0.Final/
com.example.apiman.FooCacheStoreComponentImpl
apiman-gateway.components.ICacheStoreComponent.foo1=value-1
apiman-gateway.components.ICacheStoreComponent.foo2=value-2
```

4.5.3.2.9. io.apiman.gateway.engine.components.IHttpClientComponent Example Configuration

```
apiman-
gateway.components.IHttpClientComponent=plugin:com.example.groupId:artifactId:1.0.Final/
com.example.apiman.FooHttpClientComponentImpl
apiman-gateway.components.IHttpClientComponent.foo1=value-1
apiman-gateway.components.IHttpClientComponent.foo2=value-2
```

4.5.3.2.10. io.apiman.gateway.engine.components.IJdbcComponent Example Configuration

```
apiman-
gateway.components.IJdbcComponent=plugin:com.example.groupId:artifactId:1.0.Final/
com.example.apiman.FooJdbcComponentImpl
apiman-gateway.components.IJdbcComponent.foo1=value-1
apiman-gateway.components.IJdbcComponent.foo2=value-2
```

4.5.3.2.11. io.apiman.gateway.engine.components.ILdapComponent Example Configuration

```
apiman-
gateway.components.ILdapComponent=plugin:com.example.groupId:artifactId:1.0.Final/
com.example.apiman.FooLdapComponentImpl
apiman-gateway.components.ILdapComponent.foo1=value-1
apiman-gateway.components.ILdapComponent.foo2=value-2
```

4.5.3.2.12. io.apiman.gateway.engine.components.IPeriodicComponent Example Configuration

```
apiman-
gateway.components.IPeriodicComponent=plugin:com.example.groupId:artifactId:1.0.Final/
com.example.apiman.FooPeriodicComponentImpl
apiman-gateway.components.IPeriodicComponent.foo1=value-1
apiman-gateway.components.IPeriodicComponent.foo2=value-2
```

4.5.3.2.13. io.apiman.gateway.engine.components.IPolicyFailureFactoryComponent Example Configuration

```
apiman-
gateway.components.IPolicyFailureFactoryComponent=plugin:com.example.groupId:artifactId:1.0.
com.example.apiman.FooPolicyFailureFactoryComponentImpl
apiman-gateway.components.IPolicyFailureFactoryComponent.foo1=value-1
apiman-gateway.components.IPolicyFailureFactoryComponent.foo2=value-2
```

4.5.3.2.14. io.apiman.gateway.engine.components.IRateLimiterComponent Example Configuration

```
apiman-
gateway.components.IRateLimiterComponent=plugin:com.example.groupId:artifactId:1.0.Final/
com.example.apiman.FooRateLimiterComponentImpl
apiman-gateway.components.IRateLimiterComponent.foo1=value-1
apiman-gateway.components.IRateLimiterComponent.foo2=value-2
```

4.5.3.2.15. io.apiman.gateway.engine.components.ISharedStateComponent Example Configuration

```
apiman-
gateway.components.ISharedStateComponent=plugin:com.example.groupId:artifactId:1.0.Final/
com.example.apiman.FooSharedStateComponentImpl
apiman-gateway.components.ISharedStateComponent.foo1=value-1
apiman-gateway.components.ISharedStateComponent.foo2=value-2
```

4.5.4. Providing a Custom API Catalog

Apiman allows users to import one or more API (to be managed) from a globally configured API Catalog. This feature makes it easier to manage APIs that are "known" by providing API catalog entries which include information such as the endpoint, endpoint type, etc. Importing an API from the catalog brings those fields into apiman, so that users don't have to manually set them.

When installing apiman, a custom API Catalog can be easily configured by creating a propertly formatted JSON file with all of the appropriate information included. See the **Installation Guide** for more information about configuring a JSON based custom API Catalog.

Additionally, it is possible to completely replace the API Catalog implementation, providing your own custom version which retrieves API information from wherever you like. Like most components, a custom API Catalog implementation is simply a Java class which implements a specific interface and is enabled/configured in the *apiman.properties* file.

The interface you must implement is **io.apiman.manager.api.core.IApiCatalog** and looked like this at the time of this writing:

```
/**
 * Represents some sort of catalog of live APIs.  This is used to lookup
 * APIs to import into apiman.
 */
public interface IApiCatalog {

    /**
    * Called to find available APIs that match the given search keyword.  Note that
```

```
     * the search keyword may be a partial word (for example "ech" instead of "echo").  It
     * is up to the implementation to decide how to handle partial cases.  Typically this
     * should return all APIs that contain the partial keyword, thus returning things
     * like "echo" "public-echo" and "echo-location".
     *
     * @param keyword the search keyword
     * @return the available APIs
     */
    public List<AvailableApiBean> search(String keyword);

}
```

The catalog is simply one method which returns a list of **AvailableApiBean** objects. That class looks something like this:

```
/**
 * A bean modeling an API available in one of the configured API catalogs.
 */
@JsonSerialize(include=JsonSerialize.Inclusion.NON_NULL)
public class AvailableApiBean implements Serializable {

    private String id;
    private String icon;
    private String endpoint;
    private EndpointType endpointType = EndpointType.rest;
    private String name;
    private String description;
    private String definitionUrl;
    private ApiDefinitionType definitionType;

    /**
     * Constructor.
     */
    public AvailableApiBean() {
    }

    /** SNIPPED ALL GETTERS/SETTERS **/
}
```

Create an implementation of this interface and include it in a valid apiman plugin.

Tip

See the "Creating a Plugin" section of this guide for more information.

Once the plugin is created with your class inside, configure the catalog in *apiman.properties* like this:

```
apiman-manager.api-
catalog.type=plugin:com.example.groupId:artifactId:1.0.Final/
com.example.apiman.ApiCatalogImpl
apiman-manager.api-catalog.property1=value-1
apiman-manager.api-catalog.property2=value-2
```

Remember, if your implementation class has a constructor that accepts a Map<String, String>, then apiman will pass the set of applicable configuration properties it finds in apiman.properties when the class is instantiated.

4.5.5. Providing a Custom Data Encrypter

Whenever apiman stores data, either in the API Manager or in the API Gateway, it uses a Data Encrypter to first encrypt potentially sensitive information. Examples are:

- Policy Configuration
- Endpoint Properties

By default, the apiman quickstart comes with a default encrypter that performs very simple synchronous encryption on this data. However, because it is built-in, it is not secure (it uses a hard-coded encryption key, for example). Depending on your security needs, you may wish to implement a custom data encrypter - one that is more secure and perhaps uses externally configured keys.

In order to provide a custom data encrypter, the interface you must implement is **io.apiman.common.util.crypt.IDataEncrypter**. This same interface is used in both the API Manager and the API Gateway. The IDataEncrypter interface looks something like this:

```
/**
 * Provides a way to encrypt and decrypt data. This is useful when encrypting sensitive
 * data prior to storing it in the database.
 */
public interface IDataEncrypter {

    public String encrypt(String plainText);

    public String decrypt(String encryptedText);

}
```

When creating a custom implementation, all you need to do is provide a Java class which implements the above interface inside a valid apiman plugin.

Tip

See the "Creating a Plugin" section of this guide for more information.

Once the plugin is created with your class inside, configure the data encrypter in *apiman.properties* like this (**note**: it only needs to be configured in a single place for both the Manager and Gateway):

```
apiman.encrypter.type=plugin:com.example.groupId:artifactId:1.0.Final/
com.example.apiman.DataEncrypterImpl
apiman.encrypter.property1=value-1
apiman.encrypter.property2=value-2
```

Remember, if your implementation class has a constructor that accepts a Map<String, String>, then apiman will pass the set of applicable configuration properties it finds in apiman.properties when the class is instantiated. In the example above, your DataEncrypterImpl class will be instantiated, with a Map passed to its constructor containing the following:

- property1=value-1
- property2=value-2

4.5.6. Providing a Custom Policy Failure/Error Writer

When a policy fails (or an error occurs) in the API Gateway, the result of the failure must be sent back to the calling HTTP client. By default, apiman has a particular format (either JSON or XML depending on the Content-Type of the API being called) it uses when responding to the client. However, some installers may prefer a custom format for these. This can be accomplished by providing a custom implementation of **io.apiman.gateway.engine.IPolicyFailureWriter** and/or a custom implementation of **io.apiman.gateway.engine.IPolicyErrorWriter**.

```
public interface IPolicyFailureWriter {

    public void write(ApiRequest request, PolicyFailure failure, IApiClientResponse response);

}
```

```
public interface IPolicyErrorWriter {

    public void write(ApiRequest request, Throwable error, IApiClientResponse response);

}
```

When creating a custom implementation, all you need to do is provide a Java class which implements the above interface(s) inside a valid apiman plugin.

Tip

See the "Creating a Plugin" section of this guide for more information.

Once the plugin is created with your class inside, configure either the failure writer, the error writer, or both in *apiman.properties* like this:

```
apiman-gateway.writers.policy-
failure=plugin:com.example.groupId:artifactId:1.0.Final/
com.example.apiman.PolicyFailureWriterImpl
apiman-gateway.writers.policy-failure.property1=value-1
apiman-gateway.writers.policy-failure.property2=value-2
```

```
apiman-gateway.writers.policy-
error=plugin:com.example.groupId:artifactId:1.0.Final/
com.example.apiman.PolicyErrorWriterImpl
apiman-gateway.writers.policy-error.property1=value-1
apiman-gateway.writers.policy-error.property2=value-2
```

Remember, if your implementation class has a constructor that accepts a Map<String, String>, then apiman will pass the set of applicable configuration properties it finds in apiman.properties when the class is instantiated. In the example above, your DataEncrypterImpl class will be instantiated, with a Map passed to its constructor containing the following:

- property1=value-1
- property2=value-2

4.5.7. Providing a Custom User Bootstrapper

Whenever a new user is added to apiman, a record is added for her in the API Manager data store. No additional steps are taken by default. However, in some cases you may want to perform some specific bootstrapping tasks when a new user is created, for example:

- Grant specific roles to the user
- Auto-create an Organization for the user

This can be done by providing your own custom implementation of **io.apiman.manager.api.core.INewUserBootstrapper**:

```
/**
```

```
 * This class is used to bootstrap new users.  This bootstrapper is used
 * whenever a new user logs into the API Manager UI for the first time.
 */
public interface INewUserBootstrapper {

    /**
     * Called to bootstrap a user.
     */
    public void bootstrapUser(UserBean user, IStorage storage) throws StorageException;

}
```

When invoked, the boostrap method is given the **UserBean** of the user being created as well as the storage object. The storage object can be used to create additional entities for the user, such as new organizations or new memberships in roles.

When creating a custom implementation, all you need to do is provide a Java class which implements the above interface inside a valid apiman plugin.

Tip

See the "Creating a Plugin" section of this guide for more information.

Once the plugin is created with your class inside, configure the user bootstrapper in in *apiman.properties* like this:

```
apiman-manager.user-
bootstrapper.type=plugin:com.example.groupId:artifactId:1.0.Final/
com.example.apiman.UserBootstrapperImpl
apiman-manager.user-bootstrapper.property1=value-1
apiman-manager.user-bootstrapper.property2=value-2
```

Remember, if your implementation class has a constructor that accepts a Map<String, String>, then apiman will pass the set of applicable configuration properties it finds in apiman.properties when the class is instantiated. In the example above, your DataEncrypterImpl class will be instantiated, with a Map passed to its constructor containing the following:

- property1=value-1
- property2=value-2

Chapter 5. Gateway Implementations

At the heart of any apiman gateway implementation is the flexible, lightweight apiman-core. The core serves to execute policies upon the traffic passing through it, determining whether a given conversation should continue or not.

A set of simple, asynchronous interfaces are provided which an implementor should fulfill using the platform's native functionality to allow apiman to interact with its various components and services.

5.1. Implementing IApimanBuffer

Before you can send any data through apiman, you must implement the `IApimanBuffer` interface. It provides a set of methods which allow apiman to access your native buffer format as effectively as possible. Any data you pass into apiman must be wrapped in your implementation of `IApimanBuffer`, whilst any data returned to you by apiman will be an `IApimanBuffer` which you can extricate your native buffer from.

Implementation is fairy self explanatory, but a few points are worth noting:

```
public class YourApimanBufferImpl implements IApimanBuffer {

  private YourNativeBuffer nativeBuffer;

  public VertxApimanBuffer(YourNativeBuffer nativeBuffer) {
    this.nativeBuffer = nativeBuffer;
  }

  // This is your mechanism to efficiently yank your native buffer back
  @Override
  public Object getNativeBuffer() {
    return nativeBuffer;
  }

  @Override
  public int length() {
    return nativeBuffer.length();
  }

  @Override
  public void insert(int index, IApimanBuffer buffer) {
    nativeBuffer.setBuffer(index, (Buffer) buffer.getNativeBuffer());
  }

  <...>
```

```
}
```

Important

Implementors of `IApimanBuffer` should ensure that the native format is preserved within the instance, this allows it to be retrieved again using `getNativeBuffer`. Any mutation should be on the native buffer.

5.2. Executing apiman-core

Let's consider the following snippet:

```
IEngine engine = new <your engine>.createEngine();

// Request executor, through which we can send chunks and indicate end.
final IApiRequestExecutor requestExecutor = engine.executor(request,
  new IAsyncResultHandler<IEngineResult>() {
    public void handle(IAsyncResult<IEngineResult> result) { ... }
  });

// streamHandler called when back-end connector is ready to receive data.
requestExecutor.streamHandler(new IAsyncHandler<IApiConnection>() {
  public void handle(final IApiConnection writeStream) { ... }
}

// Execute the request
executor.execute();
```

After instantiating your engine implementation, you can call `execute`. This is the main point through which you pipe data into and out of apiman. In order to avoid any buffering you must write body data through `streamHandler`'s `IApiConnection` which will be called when the connection to the backend API is ready to receive. The result is provided to `executor`'s `IAsyncResultHandler`, which can be evaluated to determine the result of the call, and, if successful, retrieve a `ApiResponse` and attach handlers to receive response data.

5.2.1. Streaming data

Exploring `streamHandler` further:

```
requestExecutor.streamHandler(new IAsyncHandler<IApiConnection>() {

  @Override
  public void handle(final IApiConnection writeStream) {
```

```
    // Just for illustrative purposes
    IApimanBuffer apimanBuffer =
      new YourApimanBufferImpl(nativeBuffer);

    // Call #write as many times as desired.
    writeStream.write(apimanBuffer);

    // Call #end only once.
    writeStream.end();
  }
}
```

Any data flowing into the executor must first be wrapped in your implementation of `IApimanBuffer` before being passed to `write`. You may call `write` an unlimited number of times, and indicate that transmission has completed by signalling `end`.

Important

No further calls to `write` should occur after `end` has been called.

5.2.2. Handling results

An excerpt of the executor's result handler and considering a successful result:

```
engine.executor(request, new IAsyncResultHandler<IEngineResult>() {
  public void handle(IAsyncResult<IEngineResult> result) {
    // Did an exception occur?
    if (result.isSuccess()) {
      IEngineResult engineResult = result.getResult();

      if (engineResult.isResponse()) {
        // Our successfully returned API response.
        ApiResponse response = engineResult.getApiResponse();

        // Set a bodyHandler to receive the response's body chunks.
        engineResult.bodyHandler(new IAsyncHandler<IApimanBuffer>() {

          @Override
          public void handle(IApimanBuffer chunk) {
           // Important: for efficiency, retrieve native buffer format directly
 if possible.
            if(chunk.getNativeBuffer() instanceof YourNativeBuffer) {
           YourNativeBuffer buffer = (YourNativeBuffer) chunk.getNativeBuffer();
            }
          }
        });
```

```
        // Set an endHandler to receive the end signal.
        engineResult.endHandler(new IAsyncHandler<Void>() {

          @Override
          public void handle(Void flag) {
            // Transmission has now completed.
          }
        });

      } else {
        // Handle policy failure.
      }

    } else {
      // Handle exception.
    }
  }
});
```

After testing `IAsyncResult.isSuccess`, we can be certain that the request completed without an exception occurring. Next, we verify `IEngineResult.isFailure`, which indicates whether there was a policy failure or the response returned successfully.

Upon success the `ApiResponse` can be extracted, and a `bodyHandler` and `endHandler` can be attached in order to receive the response's associated data as it arrives. At this point the data has exited apiman, and can handled as makes sense for your implementation. For instance, you may wish to translate the `ApiResponse` into its native equivalent and return it to the requestor.

Tip

Where possible, it is advisable to use `getNativeBuffer` on any `IApimanBuffer` chunks you receive; avoiding any expensive format conversions. You must cast it back to your native format; `instanceof` is helpful to ensure the the correct type has been received.

5.2.3. Handling Failures

In the case of errors or policy failures, a variety of information is provided which can be used to construct a sensible response:

```
if (result.isSuccess()) {
  IEngineResult engineResult = result.getResult();

  if (!engineResult.isFailure()) {
```

```
    <...>
  } else {
    PolicyFailure policyFailure = engineResult.getPolicyFailure();
    log.info("Failure type: " + policyFailure.getType());
    log.info("Failure code: " + policyFailure.getFailureCode());
    log.info("Failure Message: " + policyFailure.getMessage());
    log.info("Failure Headers: " + policyFailure.getHeaders());
  }
} else {
  Throwable throwable = engineResult.getError();
  log.error("Something bad happened: " + throwable);
}
```

The appropriate response to failures will vary widely depending upon implementation. For instance, a RESTful platform may wish to transmit an appropriate HTTP error code, message and possibly body.

5.3. Creating an API Connector

Connectors enable apiman to transmit and receive data from the backend APIs under management. For instance, should your system need to connect to an HTTP API, an HTTP connector must be created. The following samples illustrate in general terms how an implementor may go about creating a connector, and although the specifics will vary extremely widely depending upon the platform some general principals should be obeyed.

5.3.1. Connector basics

Inside of your `IConnectorFactory` implementation you must return an `IApiConnector` corresponding to the type of request and API being interacted with:

```
public class ConnectorFactory implements IConnectorFactory {

  public IApiConnector createConnector(ApiRequest request, Api api) {
    return new IApiConnector() {
        ...
    }
  }
}
```

Inspecting the `IApiConnector` more closely, we can see the key interface of a connector:

```
public IApiConnection request(ApiRequest request,
  IAsyncResultHandler<IApiConnectionResponse> resultHandler) {
        ...
  }
```

```
}
```

The `IApiConnection` you must return is used by apiman to write request chunks; hence, it will be **read** by your connector. Conversely, the `IApiConnectionResponse` handler must be called in order to send the `ApiResponse` and its associated data chunks back to apiman once a response has returned from the API; hence, you will **write** data to it.

The `IAsyncResultHandler` is also used to indicate whether an exception has occurred during the conversation with the backend.

5.3.2. Creating the IApiConnection

Generally, an implementor must attempt to return their `IApiConnection` as soon as it is valid for apiman to write data to the backend. Until you respond, apiman will not fire `IApiRequestExecutor.streamHandler`, and hence no data will arrive prematurely to your connector. Following this guideline should help to minimise or eliminate any buffering requirements in your connectors.

Looking at an example:

```
// Native platform's connector (e.g. HTTP)
ImaginaryBackendConnector imaginaryConnector = ...;
    Connection c = imaginaryConnector.establishConnection(api.getEndpoint(), ...);

// Prepare in advance to do something sensible with the response
// See next section for more detail.
c.responseHandler(<Handle the response; return an IApiConnectionResponse>);

// From our perspective IApiConnection is
// *inbound data* (i.e. the user writes to us).
return new IApiConnection() {
  boolean finished = false;

  @Override
  public void write(IApimanBuffer chunk) {
    // Handle arriving data chunk
    YourNativeBuffer nativeBuffer =
      (YourNativeBuffer) chunk.getNativeBuffer();

    imaginaryConnector.write(nativeBuffer);
  }

  @Override
  public void end() {
    // Handle the signal to indicate stream has completed
    imaginaryConnector.finish_connection();
    finished = true;
```

```
  }

  @Override
  public void abort() {
    // Handle immediate abort, for instance by closing your connection.
    imaginaryConnector.abort();
    finished = true;
  }


  @Override
  public boolean isFinished() {
    return finished;
  }
};
```

`imaginaryConnector` represents your platform's backend connector. After establishing a connection that can accept data, you should return an `IApiConnection`, allowing data to be written to your connector. You can extract your native buffer format using `getNativeBuffer` plus a cast. Although we haven't yet explored how to handle a response, we can imagine that the platform's `ImaginaryBackendConnector` would allows us to set a `responseHandler`, which will be fired when a response has arrived; this is point at which we can build an `IApiConnectionResponse`.

5.3.3. Creating the IApiConnectionResponse

5.3.3.1. Handling a successful response

Apiman's `resultHandler` should be called with an `IApiConnectionResponse` when your connector has received a response from the API.

Let's imagine that `responseHandler` is called when the platform's response has arrived, and looks like this:

```
c.responseHandler(new Handler<ImaginaryResponse> {
  public void handle(ImaginaryResponse response) {
    ...
  }
});
```

This is where we must build our apiman response, using the data returned in the platform's response, and attaching appropriate handlers to capture any data that arrives.

In the following example, we expand the response `handle` method to build an `IApiConnectionResponse`:

```
void handle(final ImaginaryResponse response) {

  IApiConnectionResponse readStream = new IApiConnectionResponse() {
    IAsyncHandler<IApimanBuffer> bodyHandler;
    IAsyncHandler<IApimanBuffer> endHandler;
    boolean finished = false;
    ApiResponse response = YourResponseBuilder.build(response);

    public IApiConnectionResponse() {
      doConnection();
    }

    private void doConnection() {
      // We stop any data arriving
      response.pause();

      // This will be called when we resume transmission
      response.bodyHandler(new Handler<NativeDataChunk>() {

        void handle(NativeDataChunk chunk) {
          IApimanBuffer apimanBuffer =
            new YourApimanBufferImpl(nativeBuffer);

          bodyHandler.handle(apimanBuffer);
        }
      });

      // Transmission has finished
      response.endHandler(new Handler<Void>) {

        void handle(Void flag) {
          endHandler.handle((Void) null);
          // You may want to close your backend connection here.
        }
      }
    }

    @Override
    public void bodyHandler(IAsyncHandler<IApimanBuffer> bodyHandler) {
      this.bodyHandler = bodyHandler;
    }

    @Override
    public void endHandler(IAsyncHandler<Void> endHandler) {
      this.endHandler = endHandler;
    }

    @Override
```

```
    public ApiResponse getHead() {
      return apiResponse;
    }

    @Override
    public boolean isFinished() {
      return finished;
    }

    @Override
    public void abort() {
      // Abort
    }

    // We explicitly resume transmission
    @Override
    public void transmit() {
      response.resume();
    }
  };

  // We're ready to transmit the response, let apiman know.
  IAsyncResult result = AsyncResultImpl.
    <IApiConnectionResponse> create(readStream);

  resultHandler.handle(result);
}
```

We imagine that our `response` object contains what we need to build a `ApiResponse`, and that handlers can be attached in order to retrieve body data and an end signal. It can be paused using `pause`, which prevents any data from arriving until `resume` is called.

Importantly, data transmission **must not** begin until `transmit` has been called, otherwise the appropriate handlers may not yet have been set, and data will be liable to disappear. Hence, in this example, `resume` is called in `transmit` where we are certain that it's safe to send data.

After `end` has been signalled, clean up on the native connection can be performed, such as closing it. In this example was assume the connection is closed for us.

Once we are sure our stream is ready, we pass it to apiman using `resultHandler.handle` wrapped inside of an IAsyncResult indicating we were successful. Some helpful `create` methods are available in `AsyncResultImpl`.

Whilst a given platform's implementation may look very different, implementors must be careful to preserve the same external behaviour; some platforms may require buffering of data if pause-like functionality is not available. In many cases it may be possible to implement `IApiConnectionResponse` and `IApiConnection` in the same class.

Important

Do not transmit any response data into apiman until `transmit` has been signalled.

5.3.3.2. Handling an error

If an error occurs, you must return a failure `IAsyncResult`, which may be caused, for instance, by an endpoint being unresolvable. The simplest way to share this is by using `AsyncResultImpl`:

```
try { ... }
catch(Exception e) {
  IAsyncResult errorResult =
    AsyncResultImpl.<IApiConnectionResponse> create(e);

  resultHandler.handle(errorResult);
}
```

Tip

Remember to clean up any resources you may have left open.

5.3.4. Implementation strategies

Implementors may notice that the only overlap between the `IApiConnection` and `IApiConnectionResponse` interfaces is the `isFinished` method. Hence, it is often possible to implement both interfaces using the same class, which may be a cleaner way to orchestrate the process.

Implementation exemplars:

- link:*https://github.com/apiman/apiman/blob/master/gateway/platforms/servlet/src/main/java/io/apiman/gateway/platforms/servlet/connectors/HttpApiConnection.java*[Servlet HTTP Connector] is a more traditional synchronous implementations.
- link: *https://github.com/apiman/apiman/blob/master/gateway/platforms/vertx/src/main/java/io/apiman/gateway/vertx/connector/HttpConnector.java*[Vert.x HTTP Connector] is an asynchronous HTTP implementation.