WSO2 API Manager

Documentation

Version 1.7.0

Table of Contents

WSO2 API Manager Documentation

Welcome to WSO2 API Manager Documentation! WSO2 API Manager (APIM) is a fully open source, complete solution for creating, publishing and managing all aspects of an API and its life cycle, and is ready for massively scalable deployments.

Use the descriptions below to find the section you need, and then browse the topics in the left navigation panel. You can also use the **Search** box on the left to find a term in this documentation, or use the search box in the top right-hand corner to search in all WSO2 product documentation.

To download a PDF of this document or a selected part of it, click here (generate only one PDF at a time). To export to a different format, click the **Browse** menu at the top of this screen, click **Space Operations**, and then select an **E xport** option.

About API Manager Introduces WSO2 API Manager, including the business cases it solves, its features, architecture and how to get help.		**Getting Started** Instructions to download, install, run and get started quickly with WSO2 API Manager.		**User Guide** Introduces the features and functionality of the API Manager, solution development, testing, debugging and deployment.	
Admin Guide Introduces product deployment and other system administration tasks.		**Samples** Real-life business use cases of the product.		**Published APIs** APIs to be used in your applications.	

About API Manager

The topics in this section introduce WSO2 API Manager, including the business cases it solves, its features, and architecture.

Introducing API Manager

As an organization implements SOA, it can benefit by exposing core processes, data and services as APIs to the public. External parties can mash up these APIs in innovative ways to build new solutions. A business can increase its growth potential and partnership advancements by facilitating developments that are powered by its APIs in a simple, decentralized manner.

However, leveraging APIs in a collaborative way introduces new challenges in exercising control, establishing trust, security and regulation. As a result, proper API management is crucial.

WSO2 API Manager overcomes these challenges with a set of features for API creation, publication, lifecycle management, versioning, monetization, governance, security etc. using proven WSO2 products such as WSO2 Enterprise Service Bus, WSO2 Identity Server, and WSO2 Governance Registry. In addition, as it is also powered by the WSO2 Business Activity Monitor and is immediately ready for massively scalable deployments.

WSO2 API Manager is fully open source and provides Web interfaces for development teams to deploy and monitor APIs. and for consumers to subscribe to, discover and consume APIs through a user-friendly storefront. The API Manager also provides complete API governance and shares the same metadata repository as WSO2 Governance Registry. If your setup requires to govern more than APIs, we recommend you to use WSO2 API manager for API governance and WSO2 Governance Registry for the other artifacts.

The WSO2 API Manager is an on-going project with continuous improvements and enhancements introduced with each new release to address new business challenges and customer expectations. WSO2 invites users, developers and enthusiasts to get involved or get the assistance of our development teams at many different levels through online forums, mailing lists and support options.

Features

Feature	Description
Creating a Store for your APIs	• Graphical experience similar to Android Marketplace or Apple App Store. • Browse APIs by provider, tags or name. • Self-registration to developer community to subscribe to APIs. • Subscribe to APIs and manage subscriptions on per-application basis. • Subscriptions can be at different service tiers based on expected usage levels. • Role based access to API Store; manage public and private APIs. • Manage subscriptions at a per-developer level. • Browse API documentation, download helpers for easy consumption. • Comment on and rate APIs. • Forum for discussing API usage issues. • Try APIs directly on the store front. • Internationalization (i18n) support.

Publishing and Governing API Usage	• Publish APIs to external consumers and partners, as well as internal users. • Supports publishing multiple protocols including SOAP, REST, JSON and XML style services as APIs. • Manage API versions and deployment status by version. • Govern the API lifecycle (publish, deprecate, retire). • Attach documentation (files, external URLs) to APIs. • Apply Security policies to APIs (authentication, authorization). • Associate API available to system defined service tiers. • Provision and Manage API keys. • Track consumers per API. • One-click deployment to API Gateway for immediate publishing.
Routing API Traffic	• Supports API authentication with OAuth2. • Extremely high performance pass-through message routing with sub-millisecond latency. • Enforce rate limiting and throttling policies for APIs by consumer. • Horizontally scalable with easy deployment into cluster using proven routing infrastructure. • Scales to millions of developers/users. • Capture all statistics and push to pluggable analytics system. • Configure API routing policies with capabilities of WSO2 Enterprise Service Bus. • Powered by WSO2 Enterprise Service Bus.
Managing the Community	• Self-sign up for API consumption. • Manage user account including password reset. • Developer interaction with APIs via comments and ratings. • Support for developer communication via forums. • Powered by WSO2 Identity Server.
Governing Complete API Lifecycle	• Manage API lifecycle from cradle to grave: create, publish, block, deprecate and retire. • Publish both production and sandbox keys for APIs to enable easy developer testing. • Publish APIs to partner networks such as Programmable Web (Available soon in future version). • Powered by WSO2 Governance Registry.
Monitoring API Usage and Performance	• All API usage published to pluggable analytics framework. • Out of the box support for WSO2 Business Activity Monitor and Google Analytics. • View metrics by user, API and more. • Customized reporting via plugging reporting engines. • Monitor SLA compliance. • Powered by WSO2 Business Activity Monitor.
Deploying with Ease in Enterprise Settings	• Role based access control for managing users and their authorization levels. • Store front can be deployed in DMZ for external access with Publisher inside the firewall for private control. • Different user stores for developer focused store-front and internal operations in publisher. • Integrates with enterprise identity systems including LDAP and Microsoft Active Directory. • Gateway can be deployed in DMZ with controlled access to WSO2 Identity Server (for authentication/authorization) and governance database behind firewall.

Customizing and Extending	• All components are highly customizable. You can change the styles and themes of the Web interfaces. • Storefront implemented with Jaggery (jaggeryjs.org) for easy customization. • Pluggable to third-party analytics systems and billing systems (Available soon in future version). • Pluggable to existing user stores including via JDBC and LDAP. • Components usable separately – storefront can be used to front APIs gatewayed via third party gateways such as Intel Expressway Service Gateway.

Architecture

The WSO2 API Manager comprises the following main components:

- API Publisher
- API Store
- API Gateway
- API Handlers
- API Key Manager

API Publisher

Provides an end-user, collaborative Web interface for API providers to publish APIs, share documentation, provision API keys, and gather feedback on API features, quality and usage. The API Publisher is powered by Jaggery, WSO2 Governance Registry and WSO2 Identity Server products.

API Developer Guide.For more information on API Publisher and its functionality, refer to sections

API Store

Provides an end-user, collaborative Web interface for consumers to self-register, discover API functionality, subscribe to APIs, evaluate them and interact with API publishers. The API Store is powered by Jaggery, WSO2 Governance Registry and WSO2 Identity Server products.

Application Developer Guide.For more information on the API Store and its functionality, refer to section

API Gateway

A runtime, back-end component developed using the WSO2 ESB, which is proven for its performance capability. API Gateway secures, protects, manages, and scales API calls. The API gateway is a simple API proxy that intercepts API requests and applies policies such as throttling and security checks. It is also instrumental in gathering API usage statistics. We use a set of handlers for security validation and throttling purposes in the API Gateway. Upon validation, it passes Web service calls to the actual back-end. If the service call is a token request call, API Gateway passes it directly to the API Key Manager Server to handle it.

The API Gateway is accessible through the URL: https://localhost:9443/carbon once the API Manager server is up and running.

You can integrate a monitoring and statistics component to the API Manager without any additional configuration effort. This monitoring component integrates with the WSO2 Business Activity Monitor, which can be deployed separately to analyze events generated by the API manager. For more information, see Publishing API Runtime .Statistics

> Although the API Gateway contains ESB features, it is recommended not to use it for ESB-specific tasks. Use it only for Gateway functionality related to API invocations. For example, if you want to call external services like SAP, use a separate ESB cluster.

API Handlers

When an API is published, a file with its synapse configuration is created on the API Gateway. The synapse configuration of each API has a set of handlers. Each of these handlers is executed on the APIs in the order they

appear in the configuration.
u can find a set of default handlers in any API Synapse definition as shown below.

```
<handlers>
   <handler
class="org.wso2.carbon.apimgt.gateway.handlers.security.APIAuthenticationHandler"/>
   <handler
class="org.wso2.carbon.apimgt.gateway.handlers.throttling.APIThrottleHandler">
        <property name="id" value="A"/>
        <property name="policyKey" value="gov:/apimgt/applicationdata/tiers.xml"/>
   </handler>
   <handler class="org.wso2.carbon.apimgt.usage.publisher.APIMgtUsageHandler"/>
   <handler
class="org.wso2.carbon.apimgt.usage.publisher.APIMgtGoogleAnalyticsTrackingHandler"/>
   <handler
class="org.wso2.carbon.apimgt.gateway.handlers.ext.APIManagerExtensionHandler"/>
</handlers>
```

Let's see what each handler does:

- **`APIAuthenticationHandler`** : Validates the OAuth2 bearer token used to invoke the API. It also determines whether the token is of type `Production` or `Sandbox` and sets `MessageContext` variables as appropriate. To extend the default authentication handler, see Writing Custom Handlers.
- **`APIThrottleHandler`** : Throttles requests based on the throttling policy specified by the `policyKey` prop erty. Throttling is applied both at the application level as well as subscription level.
- **`APIMgtUsageHandler`** : Publishes events to BAM for collection and analysis of statistics. This handler only comes to effect if API usage tracking is enabled. See Publishing API Runtime Statistics for more information.
- **`APIMgtGoogleAnalyticsTrackingHandler`** : Publishes events to Google Analytics. This handler only comes into effect if Google analytics tracking is enabled. See Integrating with Google Analytics for more information.
- **`APIManagerExtensionHandler`** : Extends the mediation flow of messages passing through the API Gateway. See Adding Mediation Extensions for more information.

API Key Manager

The API Key Manager component handles all security and key-related operations. When API Gateway receives API calls, it contacts the API Key Manager service to verify the validity of tokens and do security checks. When API Gateway receives calls to log in, it directly forwards the calls to Key Manager server. You must pass username, password, consumer key and consumer secret key with it to register. All tokens used for validation are based on OAuth 2.0.0 protocol. Secure authorization of APIs is provided by the OAuth 2.0 standard for key management. The API Gateway supports API authentication with OAuth 2.0, and enables IT organizations to enforce rate limits and throttling policies.
When the API Gateway receives API invocation calls, it similarly
contacts the API Key Manager service for verification. This verification call happens every time the Gateway receives an API invocation call if caching is not enabled at the Gateway level.

ation between API Gateway and Key Manager happens in either of the following ways:

- Through a Web service call
- Through a Thrift call

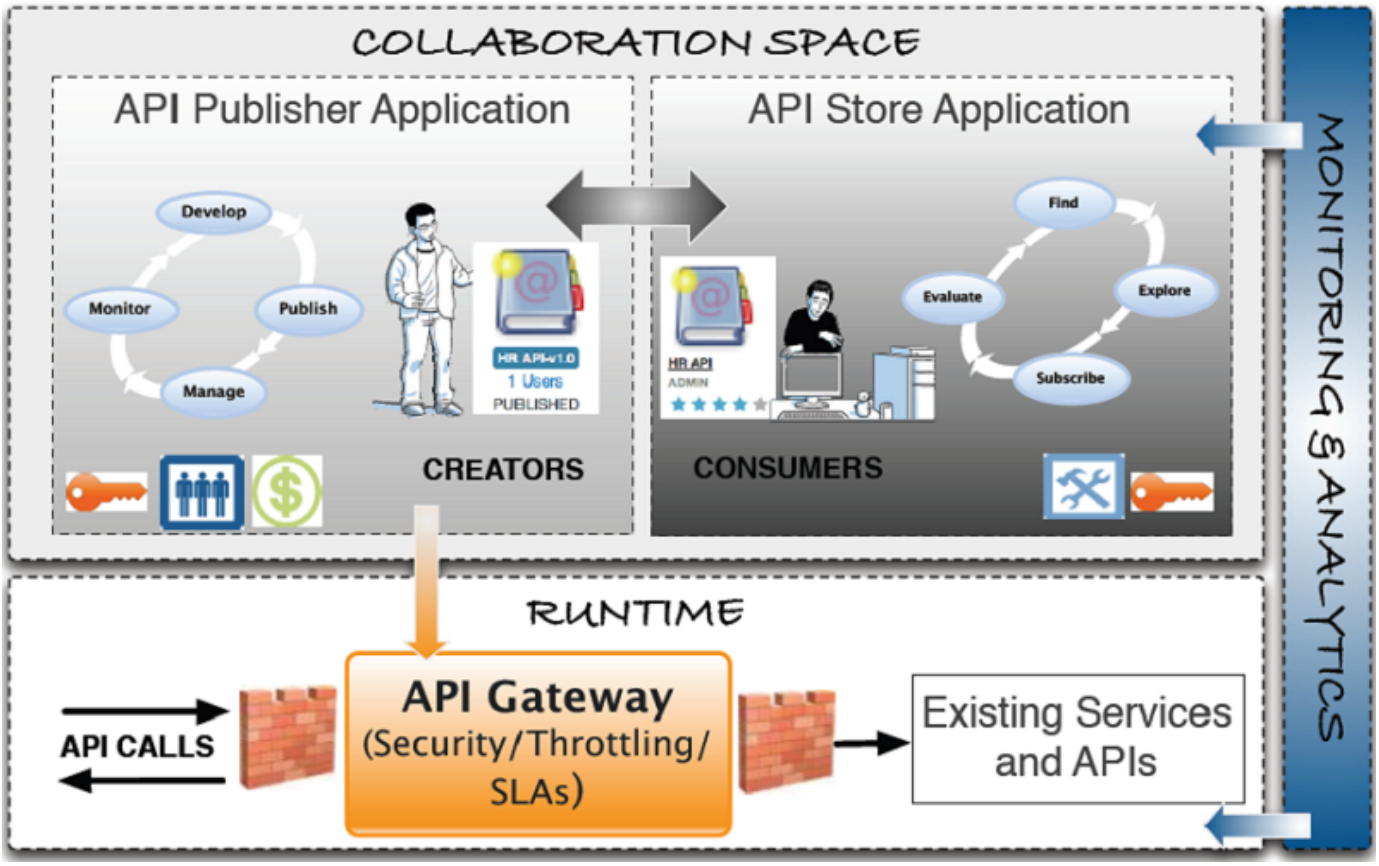
The default communication protocol of Key Manager is Thrift, which is an RPC framework used to develop scalable, cross-language services. Thrift is much faster than SOAP over HTTP communication.

> For detailed information on Thrift, see http://thrift.apache.org/static/files/thrift-20070401.pdf.

If your setup has a cluster of multiple Key Manager nodes that are fronted by a WSO2 ELB instance for load

balancing, change the key management protocol from Thrift to WSClient using the `<KeyValidatorClientType>` element in `<APIM_HOME>/repository/conf/api-manager.xml` file. Thrift uses TCP load balancing and the ELB does not support it.

The following diagram depicts the collaboration of these main components with an easily-integrable **monitoring and statistics** component.



About this Release

What is new in this release

The WSO2 API Manager version **1.7.0** is the successor of version **1.6.0**. It contains the following new features and enhancements:

- Capability to engage workflows when registering applications. See Workflow: Application Registration.
- Capability to provide custom error handling logic via custom fault sequence selected from the API Publisher Web interface.
- Links added from the API Publisher UI to API Store and also from the API Store UI to the API Publisher, to quickly navigate between the two applications. These links are configurable. See Adding Links to Navigate Between the Store and Publisher.
- Capability to parametrize the URL when defining API resources, so that the API Manager can map the incoming requests to the defined resource templates based on the message content and request URI. See URL Pattern section in API Resources.
- Capability to add a customized theme to your API Store in a multi-tenanted APIM setup. See Customizing the API Store.
- Improved search capability, including full-text search, by embedding Apache Solr with API Store.
- Capability for users to view their API usage statistics, billing rates etc. from the API Store. See Viewing API Statistics.
- Capability to generate a destination-based usage tracking graph that shows the number of times an API accesses its destination addresses. See API Manager Statistics dashboard.
- Capability to specify a default version form all API versions. See Default API Versions.

- New API visibility level where only users of the same tenant domain can view and use APIs you publish. See API Visibility.

Compatible WSO2 product versions

The following products were tested for compatibility with WSO2 APIM 1.7.0:

WSO2 APIM 1.7.0 is based on WSO2 Carbon 4.2.0 and is expected to be compatible with any other WSO2 product that is based on the same Carbon version. If you get any compatibility issues, please contact team WSO2. For information on the third-party software required with APIM 1.7.0, see Installation Prerequisites.

Fixed issues

For a list of fixed issues in this release, see WSO2 API Manager 1.7.0 - Fixed Issues.

Known issues

For a list of known issues in this release, see WSO2 API Manager 1.7.0 - Known Issues.

FAQ

- General API Manager questions
  - What is WSO2 API Manager?
  - What is the open source license of the API Manager?
  - How do I download and get started quickly?
  - Is their commercial support available for WSO2 API Manager?
  - What are the default ports opened in the API Manager?
  - What are the technologies used underneath WSO2 API Manager?
  - Can I get involved in APIM development activities?
- Installation questions
  - What are the minimum requirements to run WSO2 API Manager?
  - What Java versions are supported by the API Manager?
  - How do I deploy a third-party library into the API Manager?
  - Do you provide automated installation scripts based on Puppet or similar solutions?
  - Is it possible to connect the API Manager directly to an LDAP or Active Directory where the corporate identities are stored?
  - Can I extend the management console UI to add custom UIs?
  - I don't want some of the features that come with WSO2 API Manager. Can I remove them?
  - How can I change the memory allocation for the API Manager?
- Clustering and deployment questions
  - Where can I look up details of different deployment patterns and clustering configurations of the API Manager?
  - What is the recommended way to manage multiple artifacts in a product cluster?
  - Is it recommended to run multiple WSO2 products on a single server?
  - Can I install features of other WSO2 products to the API Manager?
- Authentication and security questions
  - How can I manage authentication centrally in a clustered environment?
  - How can I manage the API permissions/visibility?
  - How can I add security policies (UT, XACML etc.) for the services?
  - How can I disable self signup capability to the API Store? I want to engage my own approval mechanism.
  - Is there a way to lock a user's account after a certain number of failed login attempts to the API Store?
- Functionality questions
  - How do I change the default admin password and what files should I edit after changing it?
  - How can I recover the admin password used to log in to the management console?
- Troubleshooting questions
  - Why do I get the following warning: org.wso2.carbon.server.admin.module.handler.AuthenticationHandler - Illegal access attempt while trying to authenticate APIKeyValidationService?
  - I hit the DentityExpansionLimit and it gives an error as

General API Manager questions

What is WSO2 API Manager?

WSO2 API Manager is a complete solution for creating, publishing and managing all aspects of an API and its life cycle. See About API Manager.

What is the open source license of the API Manager?

Apache Software License Version 2.0

How do I download and get started quickly?

Go to http://wso2.com/products/api-manager to download the binary or source distributions. See Getting Started.

Is their commercial support available for WSO2 API Manager?

It is completely supported from evaluation to production. See WSO2 Support.

What are the default ports opened in the API Manager?

See Default Ports of WSO2 Products.

What are the technologies used underneath WSO2 API Manager?

The API Manager is built on top of WSO2 Carbon, an OSGi based components framework for SOA. See Architecture.

Can I get involved in APIM development activities?

Not only are you allowed, but also encouraged. You can start by subscribing to dev@wso2.org and architecture@wso2.org mailing lists. Feel free to provide ideas, feedback and help make our code better. For more information on contacts, mailing lists and forums, see Getting Support.

Installation questions

What are the minimum requirements to run WSO2 API Manager?

Minimum requirement is Oracle Java SE Development Kit (JDK). See Installation Prerequisites.

What Java versions are supported by the API Manager?

See Installation Prerequisites.

How do I deploy a third-party library into the API Manager?

Copy any third-party JARs to `<APIM_HOME>/repository/components/lib` directory and restart the server.

Do you provide automated installation scripts based on Puppet or similar solutions?

Yes. For information, contact us.

Is it possible to connect the API Manager directly to an LDAP or Active Directory where the corporate

identities are stored?

Yes. You can configure the API Manager with multiple user stores. See Configuring User Stores.

Can I extend the management console UI to add custom UIs?

Yes, you can extend the management console (default URL is `https://localhost:9443/carbon`) easily by writing a custom UI component and simply deploying the OSGi bundle.

I don't want some of the features that come with WSO2 API Manager. Can I remove them?

Yes, you can do this using the **Features** menu under the **Configure** menu of the management console (default URL is `https://localhost:9443/carbon`).

How can I change the memory allocation for the API Manager?

The memory allocation settings are in `<APIM_HOME>/bin/wso2server.sh` file.

Clustering and deployment questions

Where can I look up details of different deployment patterns and clustering configurations of the API Manager?

See WSO2 clustering and deployment guide.

What is the recommended way to manage multiple artifacts in a product cluster?

For artifact governance and lifecycle management, we recommend you to use a shared WSO2 Governance Registry instance.

Is it recommended to run multiple WSO2 products on a single server?

This is not recommend in a production environment involving multiple transactions. If you want to start several WSO2 products on a single server, you must change their default ports to avoid port conflicts. See Changing the Default Ports with Offset.

Can I install features of other WSO2 products to the API Manager?

Yes, you can do this using the management console. The API Manager already has features of WSO2 Identity Server, WSO2 Governance Registry, WSO2 ESB etc. embedded in it. However, if you require more features of a certain product, it is recommended to use a separate instance of it rather than instal its features to the API Manager.

Authentication and security questions

How can I manage authentication centrally in a clustered environment?

You can enable centralized authentication using a WSO2 Identity Server based security and identity gateway solution, which enables SSO (Single Sign On) across all the servers.

How can I manage the API permissions/visibility?

To set visibility of the API only to selected user roles in the server, see API Visibility.

How can I add security policies (UT, XACML etc.) for the services?

This should be done in the backend services in the Application Server or WSO2 ESB.

How can I disable self signup capability to the API Store? I want to engage my own approval mechanism.

To disable the self signup capability, set **`<SelfSignUp><Enabled>` element to false** in the `<APIM_HOME>/repository/conf/api-manager.xml` file.

Is there a way to lock a user's account after a certain number of failed login attempts to the API Store?

If your identity provider is WSO2 Identity Server, this facility comes out of the box. If not, install the identity-mgt feature to the API Manager and configure it. For information, see Account Lock/Unlock page in the Identity Server documentation.

Functionality questions

How do I change the default admin password and what files should I edit after changing it?

To change the default admin password, log in to the management console with admin/admin credentials and use the "Change my password" option. After changing the password, change the following elements in `<APIM_HOME>repository/conf/api-manager.xml` file:

```
<AuthManager>
   <Username>admin</Username>
   <Password>newpassword</Password>
</AuthManager>

<APIGateway>
   <Username>admin</Username>
   <Password>newpassword</Password>
</APIGateway>

<APIKeyManager>
   <Username>admin</Username>
   <Password>newpassword</Password>
</APIKeyManager>
```

How can I recover the admin password used to log in to the management console?

Use `<APIM_HOME>/bin/chpasswd.sh` script.

Troubleshooting questions

Why do I get the following warning: org.wso2.carbon.server.admin.module.handler.AuthenticationHandler - Illegal access attempt while trying to authenticate APIKeyValidationService?

- Did you change the default admin password? If so, you need to change the credentials stored in the `<APIKeyManager>` element of the `<APIM_HOME>/repository/conf/api-manager.xml` file of the API Gateway node/s.
- Have you set the priority of the `SAML2SSOAuthenticator` handler higher than that of the `BasicAuthenticator` handler in the authenticators.xml file? If so, the `SAML2SSOAuthenticator` handler tries to manage the basic authentication requests as well. Set a lower priority to the `SAML2SSOAuthenticator` than the `BasicAuthenticator` handler as follows:

```
<Authenticator name="SAML2SSOAuthenticator" disabled="false">
   <Priority>0</Priority>
   <Config>
      <Parameter name="LoginPage">/carbon/admin/login.jsp</Parameter>
      <Parameter name="ServiceProviderID">carbonServer</Parameter>
      <Parameter
name="IdentityProviderSSOServiceURL">https://localhost:9444/samlsso</Parameter>
      <Parameter
name="NameIDPolicyFormat">urn:oasis:names:tc:SAML:1.1:nameid-format:unspecified</
Parameter>
      <Parameter name="ISAuthnReqSigned">false</Parameter>
      <!-<Parameter
name="AssetionConsumerServiceURL">https://localhost:9443/acs</Parameter>->
   </Config>
</Authenticator>
```

I hit the `DentityExpansionLimit` and it gives an error as {org.wso2.carbon.apimgt.hostobjects.APIStoreHostObject} - Error while getting Recently Added APIs Information. What is the cause of this?

This error occurs in JDK 1.7.0_45 and is fixed in JDK 1.7.0_51 onwards. See here for details of the bug.

In JDK 1.7.0_45, all XML readers share the same `XMLSecurityManager` and `XMLLimitAnalyzer`. When the total count of all readers hits the entity expansion limit, which is 64000 by default, the XMLLimitanalyzer's total counter is accumulated and the `XMLInputFactory` cannot create more readers. If you still want to use update 45 of the JDK, try restarting the server with a higher value assigned to the `DentityExpansionLimit.`

When I call a REST API, I find that a lot of temporary files are created in my server and they are not cleared. This takes up a lot of space. What should I do?

There might be multiple configuration context objects created per each API invocation. Please check whether your client is creating a configuration context object per each API invocation. Also, configure a HouseKeeping task in the `<APIM_HOME>/repository/conf/carbon.xml` file to clear the temporary folders. For example.

```
<HouseKeeping>
        <AutoStart>true</AutoStart>

        <!-- The interval in *minutes*, between house-keeping runs -->
        <Interval>10</Interval>

        <!-- The maximum time in *minutes*, temp files are allowed to live in the
system. Files/directories which were modified more than
         "MaxTempFileLifetime" minutes ago will be removed by the house-keeping task
-->
        <MaxTempFileLifetime>30</MaxTempFileLifetime>
</HouseKeeping>
```

General technology questions

Does the API Manager use Thrift and where can I find information about it?

That the default communication protocol of Key Manager is Thrift. See http://thrift.apache.org/static/files/thrift-20070401.pdf for information on Thrift.

Getting Started

The following topics show how to download, install, run and get started quickly with WSO2 API Manager.

- Downloading the Product
- Installation Prerequisites
- Installing the Product
- Building from Source
- Running the Product
- Quick Start Guide
- Upgrading from the Previous Release

Downloading the Product

Follow the instructions below to download the product. You can also download and build the source code.

1. In your Web browser, go to http://wso2.com/products/api-manager.
2. If you are a new user downloading WSO2 products for the first time, register and log in.
3. Once you are logged in, click the **Binary** button in the upper right corner of the page.

The binary distribution contains the binary files for both MS Windows and Linux-based operating systems, compressed into a single ZIP file. This distribution is recommended for many users.

After downloading the binary distribution, go to Installation Prerequisites for instructions on installing the necessary supporting applications.

Installation Prerequisites

Prior to installing any WSO2 Carbon based product, it is necessary to have the appropriate prerequisite software installed on your system. Verify that the computer has the supported operating system and development platforms before starting the installation.

System requirements

Memory	• ~ 2 GB minimum • ~ 512 MB heap size. This is generally sufficient to process typical SOAP messages but the require ments vary with larger message sizes and the number of messages processed concurrently.
Disk	• ~ 180 MB, excluding space allocated for log files and databases.

Environment compatibility

- All WSO2 Carbon-based products are Java applications that can be run on **any platform that is Oracle JDK 1.6.*/1.7.* compliant. JDK 1.8 is not supported yet**. Also, we **do not recommend or support OpenJDK**.
- All WSO2 Carbon-based products are generally compatible with most common DBMSs. The embedded H2 database is suitable for development, testing, and some production environments. For most enterprise production environments, however, we recommend you use an industry-standard RDBMS such as Oracle, PostgreSQL, MySQL, MS SQL, etc. For more information, see Working with Databases. Additionally, we do not recommend the H2 database as a user store.
- It is **not recommended to use Apache DS** in a production environment due to scalability issues. Instead, use an LDAP like OpenLDAP for user management.
- For environments that WSO2 products are tested with, see Environments Tested with WSO2 Products.
- If you have difficulty in setting up any WSO2 product in a specific platform or database, please contact us.

Required applications

The following applications are required for running the API Manager and its samples or for building from the source code. Mandatory installs are marked with *.

Application	Purpose	Version	Download
Oracle Java S E Development Kit (JDK)*	Required to, • To launch the product as each product is a Java application. • To build the product from the source distribution (both JDK and Apache Maven are required). • To run Apache Ant.	1.6.27 or later / 1.7.* • If you are using **JDK 1.6**, you might need to replace the Java Cryptography Extension (JCE) policy files in your JDK with the Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy files. This will avoid "illegal key size" errors when you try to invoke a secured Web service. • **To build the product** from the source distribution, you must use JDK 1.6 instead of JDK 1.7. • **Oracle and IBM JRE 1.7 are also supported** when running (not building) WSO2 products. • If you are using **JDK 1.7 on a Mac OS or Solaris**, install the snappy-java library using the following steps: 1. Download the snappy-java JAR and extract it to a preferred location. This folder will be referred to as `<SNAPPY_HOME>`. 2. Copy the appropriate snappy-java library file `i386.jnilib` (32bit) or `x86_64.jnilib` (64bit), which is in the `<SNAPPY_HOME>/org/xerial/snappy/native/Mac/` directory, to the `<APIM_HOME>` directory. For more information on installing snappy-java library, see Snappy-java fails on Mac OS JDK 1.7. • We **do not recommend OpenJDK**.	http://java.s
Apache ActiveMQ J MS Provider	• To enable the product's JMS transport and try out JMS samples. The ActiveMQ client libraries must be installed in the product's classpath before you can enable the JMS transport.	5.5.0 or later If you use any other JMS provider (e.g., Apache Qpid), install any necessary libraries and/or components.	http://active
Apache Ant	• To compile and run the product samples.	1.7.0 or later	http://ant.a

SVN Client	• To check out the code to build the product from the source distribution. If you are installing by downloading and extracting the binary distribution instead of building from the source code, you do **not** need to install SVN.		• Linux - ges.htr • Windov .html
Apache Maven	• To build the product from the source distribution (both JDK and Apache Maven are required). If you are installing by downloading and extracting the binary distribution instead of building from the source code, you do **not** need to install Maven.	3.0.*	http://mave

Web Browser	• Required by all WSO2 products to access each product's Management Console. The Web Browser must be JavaScript enabled to take full advantage of the Management console. **NOTE:** On Windows Server 2003, you must not go below the medium security level in Internet Explorer 6.x.		

You are now ready to install. Click one of the following links for instructions:

- Installing on Linux or OS X
- Installing on Solaris
- Installing on Windows
- Installing as a Linux Service

Installing the Product

Installing WSO2 API Manager is very fast and easy. Before you begin, be sure you have met the installation prerequisites, and then follow the installation instructions for your platform. WSO2 API Manager also provides pre-configured packages for automated installation based on Puppet or similar solutions. For information, contact team WSO2.

- Installing on Linux or OS X
- Installing on Solaris
- Installing on Windows
- Installing as a Linux Service
- Installing as a Windows Service

Installing on Linux or OS X

> ⚠ **Before you begin**, please see our compatibility matrix to find out if this version of the product is fully tested on Linux or OS X.

Follow the instructions below to install API Manager on Linux or Mac OS X.

Installing the required applications

1. Log in to the command line (Terminal on Mac) either as root or obtain root permissions after logging in via `su` or `sudo` command.
2. Ensure that your system meets the Installation Prerequisites. Java Development Kit (JDK) is essential to run the product.

Installing the API Manager

1. Download the latest version of the API Manager as described in Downloadingthe Product.
2. Extract the archive file to a dedicated directory for the API Manager, which will hereafter be referred to as <API_HOME>.

Setting up JAVA_HOME

You must set your `JAVA_HOME` environment variable to point to the directory where the Java Development Kit (JDK) is installed on the computer.

> Environment variables are global system variables accessible by all the processes running under the operating system.

1. In your home directory, open the BASHRC file (.bash_profile file on Mac) using editors such as vi, emacs, pico, or mcedit.
2. Assuming you have JDK `1.6.0_25` in your system, add the following two lines at the bottom of the file, replacing `/usr/java/jdk1.6.0_25` with the actual directory where the JDK is installed.

```
On Linux:
export JAVA_HOME=/usr/java/jdk1.6.0_25
export PATH=${JAVA_HOME}/bin:${PATH}

On OS X:
export JAVA_HOME=/System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
```

3. Save the file.

> If you do not know how to work with text editors in a Linux SSH session, run the following command: `cat >> .bashrc.` Paste the string from the clipboard and press "Ctrl+D."

4. To verify that the `JAVA_HOME` variable is set correctly, execute the following command:

```
On Linux:
echo $JAVA_HOME

On OS X:
which java

If the above command gives you a path like /usr/bin/java, then it is a symbolic
link to the real location. To get the real location, run the following:
ls -l `which java`
```

5. The system returns the JDK installation path.

Setting system properties

If you need to set additional system properties when the server starts, you can take the following approaches:

- **Set the properties from a script**: Setting your system properties in the startup script is ideal, because it ensures that you set the properties every time you start the server. To avoid having to modify the script each time you upgrade, the best approach is to create your own startup script that wraps the WSO2 startup script and adds the properties you want to set, rather than editing the WSO2 startup script directly.
- **Set the properties from an external registry**: If you want to access properties from an external registry, you could create Java code that reads the properties at runtime from that registry. Be sure to store sensitive data

such as username and password to connect to the registry in a properties file instead of in the Java code and secure the properties file with the secure vault.

> When using SUSE Linux, it ignores `/etc/resolv.conf` and only looks at the `/etc/hosts` file. This means that the server will throw an exception on startup if you have not specified anything besides localhost. To avoid this error, add the following line above `127.0.0.1 localhost` in the `/etc/hosts` file : `<ip_address> <machine_name> localhost`

You are now ready to run the product.

Installing on Solaris

> **Before you begin**, please see our compatibility matrix to find out if this version of the product is fully tested on Solaris.

Follow the instructions below to install API Manager on Solaris.

Installing the required applications

1. Establish a SSH connection to the Solaris machine or log in on the text console. You should either log in as root or obtain root permissions after login via `su` or `sudo` command.
2. Be sure your system meets the Installation Prerequisites. Java Development Kit (JDK) is essential to run the product.

Installing the API Manager

1. Download the latest version of the API Manager as described in Downloading the Product.
2. Extract the archive file to a dedicated directory for the API Manager, which will hereafter be referred to as `<API_HOME>`.

Setting up JAVA_HOME

You must set your `JAVA_HOME` environment variable to point to the directory where the Java Development Kit (JDK) is installed on the computer.

> Environment variables are global system variables accessible by all the processes running under the operating system.

1. In your home directory, open the BASHRC file in your favorite text editor, such as vi, emacs, pico, or mcedit.
2. Assuming you have JDK 1.6.0_25 in your system, add the following two lines at the bottom of the file, replacing `/usr/java/jdk1.6.0_25` with the actual directory where the JDK is installed.

```
export JAVA_HOME=/usr/java/jdk1.6.0_25
export PATH=${JAVA_HOME}/bin:${PATH}
```

The file should now look like this:

```
# .bashrc

# Source global definitions
if [ -f /etc/bashrc ]; then
        . /etc/bashrc
fi

# User specific aliases and functions
export JAVA_HOME=/usr/java/jdk1.6.0_25
export PATH=${JAVA_HOME}/bin:${PATH}
export M2_HOME=/opt/apache-maven-3.0.3
export PATH=${M2_HOME}/bin:${PATH}
```

3. Save the file.

> If you do not know how to work with text editors in an SSH session, run the following command: `cat >> .bashrc`
>
> Paste the string from the clipboard and press "Ctrl+D."

4. To verify that the `JAVA_HOME` variable is set correctly, execute the following command:

   `echo $JAVA_HOME`

```
[suncoma@wso2 ~]$ echo $JAVA_HOME
/usr/java/jdk1.6.0_25
[suncoma@wso2 ~]$
```

5. The system returns the JDK installation path.

Setting system properties

If you need to set additional system properties when the server starts, you can take the following approaches:

- **Set the properties from a script**: Setting your system properties in the startup script is ideal, because it ensures that you set the properties every time you start the server. To avoid having to modify the script each time you upgrade, the best approach is to create your own startup script that wraps the WSO2 startup script and adds the properties you want to set, rather than editing the WSO2 startup script directly.
- **Set the properties from an external registry**: If you want to access properties from an external registry, you could create Java code that reads the properties at runtime from that registry. Be sure to store sensitive data such as username and password to connect to the registry in a properties file instead of in the Java code and secure the properties file with the secure vault.

You are now ready to run the product.

Installing on Windows

> ⚠ **Before you begin**, please see our compatibility matrix to find out if this version of the product is fully tested on Windows.

Follow the instructions below to install API Manager on Windows.

Installing the required applications

Installation Prerequisites

1. Be sure your system meets the. Java Development Kit (JDK) is essential to run the product.
2. Be sure that the `PATH` environment variable is set to "C:\Windows\System32", because the `findstr` window s exe is stored in this path.

Installing the API Manager

Downloadingthe Product

1. Download the latest version of the API Manager as described in.
2. Extract the archive file to a dedicated directory for the API Manager, which will hereafter be referred to as <AP

IM_HOME>.

Setting up JAVA_HOME

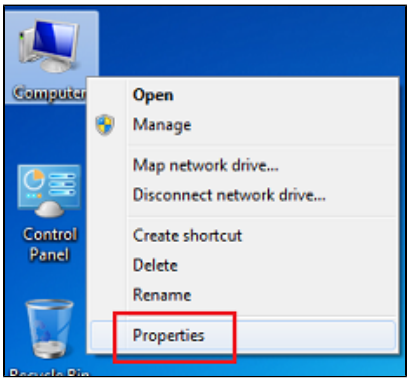
You must set your `JAVA_HOME` environment variable to point to the directory where the Java Development Kit (JDK) is installed on the computer. Typically, the JDK is installed in a directory under `C:/Program Files/Java`, such as `C:/Program Files/Java/jdk1.6.0_27`. If you have multiple versions installed, choose the latest one, which you can find by sorting by date.

> Environment variables are global system variables accessible by all the processes running under the operating system. You can define an environment variable as a system variable, which applies to all users, or as a user variable, which applies only to the user who is currently logged in.

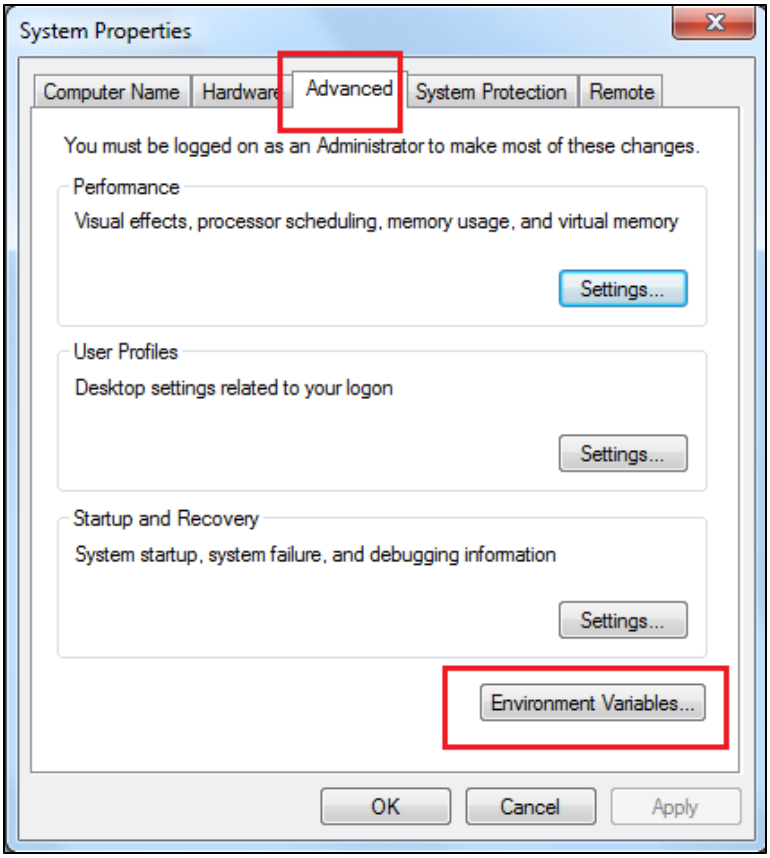
You set up JAVA_HOME using the System Properties, as described below. Alternatively, if you just want to set JAVA_HOME temporarily for the current command prompt window, set it at the command prompt.

Setting up JAVA_HOME using the system properties

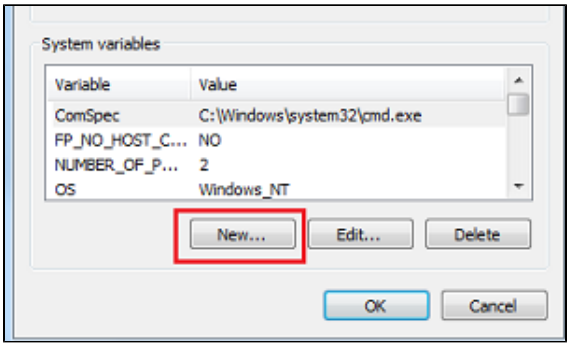
1. Right-click the **My Computer** icon on the desktop and choose **Properties**.



2. In the System Properties window, click the **Advanced** tab, and then click the **Environment Variables** button.



3. Click the New button under **System variables** (for all users) or under **User variables** (just for the user who is currently logged in).



4. Enter the following information:
   - In the **Variable name** field, enter: `JAVA_HOME`
   - In the **Variable value** field, enter the installation path of the Java Development Kit, such as: `c:/Program Files/Java jdk1.6.0_27`

The JAVA_HOME variable is now set and will apply to any subsequent command prompt windows you open. If you have existing command prompt windows running, you must close and reopen them for the JAVA_HOME variable to take effect, or manually set the JAVA_HOME variable in those command prompt windows as described in the next section. To verify that the `JAVA_HOME` variable is set correctly, open a command window (from the **Start** menu, click **Run**, and then type `CMD` and click **Enter**) and execute the following command:

```
set JAVA_HOME
```

The system returns the JDK installation path. You are now ready to run the product.

Setting JAVA_HOME temporarily using the Windows command prompt (CMD)

You can temporarily set the `JAVA_HOME` environment variable within a Windows command prompt window (CMD). This is useful when you have an existing command prompt window running and you do not want to restart it.

1. In the command prompt window, enter the following command where <JDK_INSTALLATION_PATH> is the JDK installation directory and press **Enter.**

   ```
   set JAVA_HOME=<JDK_INSTALLATION_PATH>
   ```

   For example: `set JAVA_HOME=c:/Program Files/java/jdk1.6.0_27`

   The JAVA_HOME variable is now set for the current CMD session only.
2. To verify that the `JAVA_HOME` variable is set correctly, execute the following command:

   ```
   set JAVA_HOME
   ```

3. The system returns the JDK installation path.

Setting system properties

If you need to set additional system properties when the server starts, you can take the following approaches:

- **Set the properties from a script**: Setting your system properties in the startup script is ideal, because it ensures that you set the properties every time you start the server. To avoid having to modify the script each time you upgrade, the best approach is to create your own startup script that wraps the WSO2 startup script and adds the properties you want to set, rather than editing the WSO2 startup script directly.
- **Set the properties from an external registry**: If you want to access properties from an external registry, you could create Java code that reads the properties at runtime from that registry. Be sure to store sensitive data such as username and password to connect to the registry in a properties file instead of in the Java code and secure the properties file with the secure vault.

You are now ready to run the product.

Installing as a Linux Service

Follow the sections below to run a WSO2 product as a Linux service:

- Prerequisites
- Setting up CARBON_HOME
- Running the product as a Linux service

Prerequisites

Install JDK 1.6.24 or later or 1.7.* and set up the `JAVA_HOME` environment variable.

Setting up CARBON_HOME

Extract the WSO2 product to a preferred directory in your machine and set the environment variable `CARBON_HOME` to the extracted directory location.

Running the product as a Linux service

1. To run the product as a service, create a startup script and add it to the boot sequence. The basic structure of the startup script has three parts (i.e., start, stop and restart) as follows:

```
#!/bin/bash

case “$1 in
start)
   echo “Starting the Service”
;;
stop)
   echo “Stopping the Service”
;;
restart)
   echo “Restarting the Service”
;;
*)
   echo $”Usage: $0 {start|stop|restart}”
exit 1
esac
```

Given below is a sample startup script. `<PRODUCT_HOME>` can vary depending on the WSO2 product's directory.

```
#! /bin/sh
export JAVA_HOME="/usr/lib/jvm/jdk1.7.0_07"

startcmd='<PRODUCT_HOME>/bin/wso2server.sh start > /dev/null &'
restartcmd='<PRODUCT_HOME>/bin/wso2server.sh restart > /dev/null &'
stopcmd='<PRODUCT_HOME>/bin/wso2server.sh stop > /dev/null &'

case "$1" in
start)
   echo "Starting the WSO2 Server ..."
   su -c "${startcmd}" user1
;;
restart)
   echo "Re-starting the WSO2 Server ..."
   su -c "${restartcmd}" user1
;;
stop)
   echo "Stopping the WSO2 Server ..."
   su -c "${stopcmd}" user1
;;
*)
   echo "Usage: $0 {start|stop|restart}"
exit 1
esac
```

In the above script, the server is started as a user by the name user1 rather than the root user. For example, `su -c "${startcmd}" user1`

2. Add the script to `/etc/init.d/` directory.

> If you want to keep the scripts in a location other than `/etc/init.d/` folder, you can add a symbolic link to the script in `/etc/init.d/` and keep the actual script in a separate location. Say your script name is prodserver and it is in `/opt/WSO2/` folder, then the commands for adding a link to `/etc/init.d/` is as follows:

- Make executable: `sudo chmod a+x /opt/WSO2/prodserver`
- Add a link to `/etc/init.d/`: `sudo ln -snf /opt/WSO2/prodserver /etc/init.d/prodserver`

3. Install the startup script to respective runlevels using the command `update-rc.d`. For example, give the following command for the sample script shown in step1:

```
sudo update-rc.d prodserver defaults
```

The `defaults` option in the above command makes the service to start in runlevels 2,3,4 and 5 and to stop in runlevels 0,1 and 6.

A **runlevel** is a mode of operation in Linux (or any Unix-style operating system). There are several runlevels in a Linux server and each of these runlevels is represented by a single digit integer. Each runlevel designates a different system configuration and allows access to a different combination of processes.

4. You can now start, stop and restart the server using `service <service name> {start|stop|restart}` command. You will be prompted for the password of the user (or root) who was used to start the service.

Installing as a Windows Service

WSO2 Carbon and any Carbon-based product can be run as a Windows service as described in the following sections:

- Prerequisites
- Setting up the YAJSW wrapper configuration file
- Setting up CARBON_HOME
- Running the product in console mode
- Working with the WSO2CARBON service

Prerequisites

- Install JDK 1.6.24 or later or 1.7.* and set up the `JAVA_HOME` environment variable.
- Download and install a service wrapper library to use for running your WSO2 product as a Windows service. WSO2 recommends Yet Another Java Service Wrapper (YAJSW) version 11.03, and several WSO2 products provide a default `wrapper.conf` file in their `<PRODUCT_HOME>/bin/yajsw/` directory. The instructions below describe how to set up this file.

Setting up the YAJSW wrapper configuration file

The configuration file used for wrapping Java Applications by YAJSW is `wrapper.conf`, which is located in the `<YAJSW_HOME>/conf/` directory and in the `<PRODUCT_HOME>/bin/yajsw/` directory of many WSO2 products. Following is the minimal `wrapper.conf` configuration for running a WSO2 product as a Windows service. Open your `wrapper.conf` file, set its properties as follows, and save it in `<YAJSW_HOME>/conf/` directory.

> If you want to set additional properties from an external registry at runtime, store sensitive information like usernames and passwords for connecting to the registry in a properties file and secure it with secure vault.

Minimal wrapper.conf configuration

```
#*****************************************************************
# working directory
#*****************************************************************
wrapper.working.dir=${carbon_home}\\
# Java Main class.
```

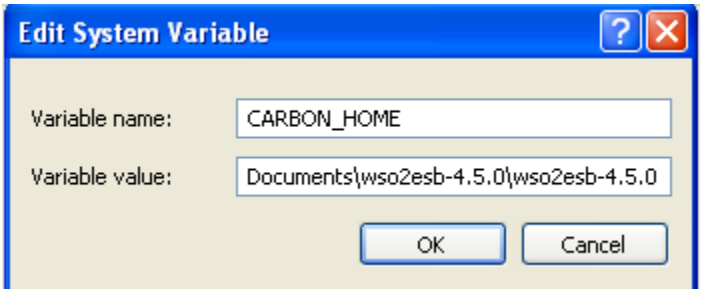
```
# YAJSW: default is "org.rzo.yajsw.app.WrapperJVMMain"
# DO NOT SET THIS PROPERTY UNLESS YOU HAVE YOUR OWN IMPLEMENTATION
# wrapper.java.mainclass=
#*********************************************************************
# tmp folder
# yajsw creates temporary files named in_.. out_.. err_.. jna..
# per default these are placed in jna.tmpdir.
# jna.tmpdir is set in setenv batch file to <yajsw>/tmp
#*********************************************************************
wrapper.tmp.path = ${jna_tmpdir}
#*********************************************************************
# Application main class or native executable
# One of the following properties MUST be defined
#*********************************************************************
# Java Application main class
wrapper.java.app.mainclass=org.wso2.carbon.bootstrap.Bootstrap
# Log Level for console output.  (See docs for log levels)
wrapper.console.loglevel=INFO
# Log file to use for wrapper output logging.
wrapper.logfile=${wrapper_home}\/log\/wrapper.log
# Format of output for the log file.  (See docs for formats)
#wrapper.logfile.format=LPTM
# Log Level for log file output.  (See docs for log levels)
#wrapper.logfile.loglevel=INFO
# Maximum size that the log file will be allowed to grow to before
#  the log is rolled. Size is specified in bytes.  The default value
#  of 0, disables log rolling by size.  May abbreviate with the 'k' (kB) or
#  'm' (mB) suffix.  For example: 10m = 10 megabytes.
# If wrapper.logfile does not contain the string ROLLNUM it will be automatically
added as suffix of the file name
wrapper.logfile.maxsize=10m
# Maximum number of rolled log files which will be allowed before old
#  files are deleted.  The default value of 0 implies no limit.
wrapper.logfile.maxfiles=10
# Title to use when running as a console
wrapper.console.title="WSO2 Carbon"
#*********************************************************************
# Wrapper Windows Service and Posix Daemon Properties
#*********************************************************************
# Name of the service
wrapper.ntservice.name="WSO2CARBON"
# Display name of the service
wrapper.ntservice.displayname="WSO2 Carbon"
# Description of the service
wrapper.ntservice.description="Carbon Kernel"
#*********************************************************************
# Wrapper System Tray Properties
#*********************************************************************
# enable system tray
wrapper.tray = true
# TCP/IP port. If none is defined multicast discovery is used to find the port
# Set the port in case multicast is not possible.
wrapper.tray.port = 15002
#*********************************************************************
# Exit Code Properties
# Restart on non zero exit code
#*********************************************************************
wrapper.on_exit.0=SHUTDOWN
wrapper.on_exit.default=RESTART
```

```
#*********************************************************************
# Trigger actions on console output
#*********************************************************************
# On Exception show message in system tray
wrapper.filter.trigger.0=Exception
wrapper.filter.script.0=scripts\/trayMessage.gv
wrapper.filter.script.0.args=Exception
#*********************************************************************
# genConfig: further Properties generated by genConfig
#*********************************************************************
placeHolderSoGenPropsComeHere=
wrapper.java.command = ${java_home}\\bin\\java
wrapper.java.classpath.1 = ${java_home}\\lib\\tools.jar
wrapper.java.classpath.2 = ${carbon_home}\\bin\\*.jar
wrapper.app.parameter.1 = org.wso2.carbon.bootstrap.Bootstrap
wrapper.app.parameter.2 = RUN
wrapper.java.additional.1 = -Xbootclasspath\/a:${carbon_home}\\lib\\xboot\\*.jar
wrapper.java.additional.2 = -Xms256m
wrapper.java.additional.3 = -Xmx1024m
wrapper.java.additional.4 = -XX:MaxPermSize=256m
wrapper.java.additional.5 = -XX:+HeapDumpOnOutOfMemoryError
wrapper.java.additional.6 =
-XX:HeapDumpPath=${carbon_home}\\repository\\logs\\heap-dump.hprof
wrapper.java.additional.7 = -Dcom.sun.management.jmxremote
wrapper.java.additional.8 =
-Djava.endorsed.dirs=${carbon_home}\\lib\\endorsed;${java_home}\\jre\\lib\\endorsed
wrapper.java.additional.9 = -Dcarbon.registry.root=\/
wrapper.java.additional.10 = -Dcarbon.home=${carbon_home}
wrapper.java.additional.11 = -Dwso2.server.standalone=true
wrapper.java.additional.12 = -Djava.command=${java_home}\\bin\\java
wrapper.java.additional.13 = -Djava.io.tmpdir=${carbon_home}\\tmp
wrapper.java.additional.14 = -Dcatalina.base=${carbon_home}\\lib\\tomcat
wrapper.java.additional.15 =
-Djava.util.logging.config.file=${carbon_home}\\repository\\conf\\log4j.properties
wrapper.java.additional.16 = -Dcarbon.config.dir.path=${carbon_home}\\repository\\conf

wrapper.java.additional.17 = -Dcarbon.logs.path=${carbon_home}\\repository\\logs
wrapper.java.additional.18 =
-Dcomponents.repo=${carbon_home}\\repository\\components\\plugins
wrapper.java.additional.19 = -Dconf.location=${carbon_home}\\repository\\conf
wrapper.java.additional.20 =
-Dcom.atomikos.icatch.file=${carbon_home}\\lib\\transactions.properties
wrapper.java.additional.21 = -Dcom.atomikos.icatch.hide_init_file_path=true
wrapper.java.additional.22 =
-Dorg.apache.jasper.runtime.BodyContentImpl.LIMIT_BUFFER=true
```

```
wrapper.java.additional.23 = -Dcom.sun.jndi.ldap.connect.pool.authentication=simple
wrapper.java.additional.24 = -Dcom.sun.jndi.ldap.connect.pool.timeout=3000
wrapper.java.additional.25 = -Dorg.terracotta.quartz.skipUpdateCheck=true
```

Setting up CARBON_HOME

Extract the Carbon-based product that you want to run as a Windows service, and then set the Windows environment variable `CARBON_HOME` to the extracted product directory location. For example, if you want to run ESB 4.5.0 as a Windows service, you would set `CARBON_HOME` to the extracted `wso2esb-4.5.0` directory.



Running the product in console mode

You will now verify that YAJSW is configured correctly for running the Carbon-based product as a Windows service.

1. Open a Windows command prompt and go to the `<YAJSW_HOME>/bat/` directory. For example:

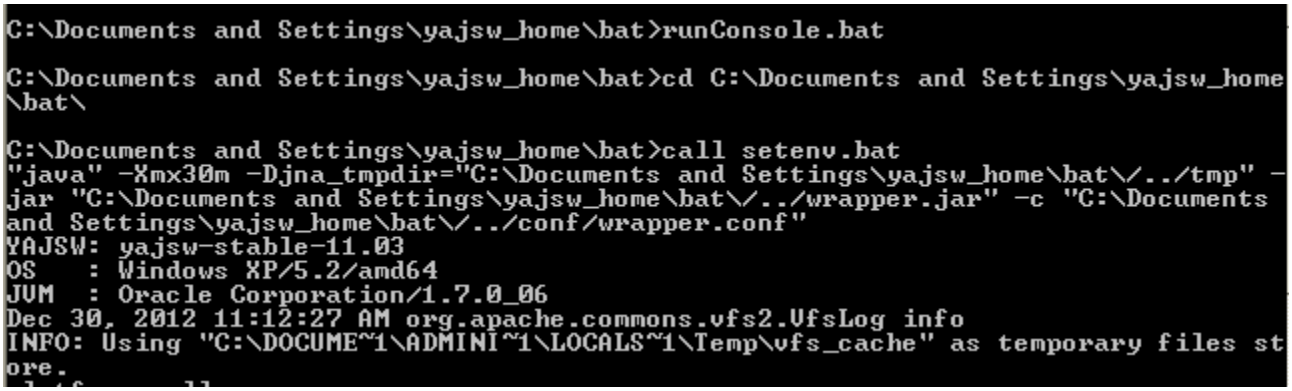
   ```
   cd C:\Documents and Settings\yajsw_home\bat
   ```

2. Start the wrapper in console mode using the following command:

   ```
   runConsole.bat
   ```

   For example:

If the configurations are set properly for YAJSW, you will see console output similar to the following and can now access the WSO2 management console from your web browser via https://localhost:9443/carbon.



Working with the WSO2CARBON service

To install the Carbon-based product as a Windows service, execute the following command in the `<YAJSW_HOME>/bat/` directory:

```
installService.bat
```

The console will display a message confirming that the WSO2CARBON service was installed.

```
C:\Documents and Settings\yajsw_home\bat>installService.bat

C:\Documents and Settings\yajsw_home\bat>cd C:\Documents and Settings\yajsw_home
\bat\

C:\Documents and Settings\yajsw_home\bat>call setenv.bat
"java" -Xmx30m -Djna_tmpdir="C:\Documents and Settings\yajsw_home\bat\/../tmp" -
jar "C:\Documents and Settings\yajsw_home\bat\/../wrapper.jar" -i "C:\Documents
and Settings\yajsw_home\bat\/../conf/wrapper.conf"
YAJSW: yajsw-stable-11.03
OS    : Windows XP/5.2/amd64
JVM   : Oracle Corporation/1.7.0_06
Dec 30, 2012 12:51:42 PM org.apache.commons.vfs2.VfsLog info
INFO: Using "C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\vfs_cache" as temporary files st
ore.
platform null
************* INSTALLING "WSO2CARBON" ***********************

Service "WSO2CARBON" installed
Press any key to continue . . .
```

To start the service, execute the following command in the same console window:

```
startService.bat
```

The console will display a message confirming that the WSO2CARBON service was started.

```
C:\Documents and Settings\yajsw_home\bat>startService.bat

C:\Documents and Settings\yajsw_home\bat>cd C:\Documents and Settings\yajsw_home
\bat\

C:\Documents and Settings\yajsw_home\bat>call setenv.bat
"java" -Xmx30m -Djna_tmpdir="C:\Documents and Settings\yajsw_home\bat\/../tmp" -
jar "C:\Documents and Settings\yajsw_home\bat\/../wrapper.jar" -t "C:\Documents
and Settings\yajsw_home\bat\/../conf/wrapper.conf"
YAJSW: yajsw-stable-11.03
OS    : Windows XP/5.2/amd64
JVM   : Oracle Corporation/1.7.0_06
Dec 30, 2012 1:09:00 PM org.apache.commons.vfs2.VfsLog info
INFO: Using "C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\vfs_cache" as temporary files st
ore.
platform null
************* STARTING "WSO2CARBON" ***********************

Service "WSO2CARBON" started
Press any key to continue . . .

C:\Documents and Settings\yajsw_home\bat>
```

To stop the service, execute the following command in the same console window:

```
stopService.bat
```

The console will display a message confirming that the WSO2CARBON service has stopped.

```
C:\Documents and Settings\yajsw_home\bat>stopService.bat

C:\Documents and Settings\yajsw_home\bat>cd C:\Documents and Settings\yajsw_home
\bat\

C:\Documents and Settings\yajsw_home\bat>call setenv.bat
"java" -Xmx30m -Djna_tmpdir="C:\Documents and Settings\yajsw_home\bat\/../tmp" -
jar "C:\Documents and Settings\yajsw_home\bat\/../wrapper.jar" -p "C:\Documents
and Settings\yajsw_home\bat\/../conf/wrapper.conf"
YAJSW: yajsw-stable-11.03
OS   : Windows XP/5.2/amd64
JVM  : Oracle Corporation/1.7.0_06
Dec 30, 2012 11:11:31 AM org.apache.commons.vfs2.VfsLog info
INFO: Using "C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\vfs_cache" as temporary files st
ore.
platform null
************* STOPPING "WSO2CARBON" ***********************

Service "WSO2CARBON" stopped
Press any key to continue . . .
```

To uninstall the service, execute the following command in the same console window:

```
uninstallService.bat
```

The console will display a message confirming that the WSO2CARBON service was removed.

```
C:\Documents and Settings\yajsw_home\bat>uninstallService.bat

C:\Documents and Settings\yajsw_home\bat>cd C:\Documents and Settings\yajsw_home
\bat\

C:\Documents and Settings\yajsw_home\bat>call setenv.bat
"java" -Xmx30m -Djna_tmpdir="C:\Documents and Settings\yajsw_home\bat\/../tmp" -
jar "C:\Documents and Settings\yajsw_home\bat\/../wrapper.jar" -r "C:\Documents
and Settings\yajsw_home\bat\/../conf/wrapper.conf"
YAJSW: yajsw-stable-11.03
OS   : Windows XP/5.2/amd64
JVM  : Oracle Corporation/1.7.0_06
Dec 30, 2012 1:19:14 PM org.apache.commons.vfs2.VfsLog info
INFO: Using "C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\vfs_cache" as temporary files st
ore.
platform null
************* REMOVING "WSO2CARBON" ***********************

Service "WSO2CARBON" removed
Press any key to continue . . .

C:\Documents and Settings\yajsw_home\bat>
```

Building from Source

WSO2 invites you to contribute by checking out the source from the Subversion (SVN) source control system, building the product and making changes, and then committing your changes back to the source repository. (For more information on Subversion, see http://svnbook.red-bean.com.) The following sections describe this process:

- Checking out the source
- Setting up your development environment
- Building the product
- Committing your changes

> Building from source is optional. Users who do not want to make changes to the source code can simply download the binary distribution of the product and install it.

Checking out the source

WSO2 products are built on top of WSO2 Carbon Kernel, which contains the Kernel libraries used by all products. When there are changes in the Carbon Kernel, they are bundled and released in a new WSO2 Carbon version (for example, WSO2 Carbon 4.2.0).

A WSO2 platform release is a set of WSO2 products based on the same Carbon release. For example, `Turing` is the platform release name for WSO2 Carbon 4.2.0 and the WSO2 products that are based on it. Usually, not all products in a platform get released at the same time, so they are released in **chunks**, each of which contains the Carbon release and a subset of products. For example, chunk 8 of the `Turing` platform release contains Carbon 4.2.0 plus Task Server 1.1.0, Data Services Server 3.2.0 and Complex Event Processor 3.1.0 .

Checking out the patches

Before checking out the product source, you need to checkout the patches related to the Carbon chunk using the following command.

```
$ svn checkout https://svn.wso2.org/repos/wso2/carbon/kernel/branches/4.2.0 <local-pl
atform-directory-1>
```

> Replace `<local-platform-directory-1>` with a meaningful name, such as `wso2carbon-platform.`

Downloading the product source

For products based on WSO2 Carbon 4.2.0, use the below command to download the product source:

```
$ svn checkout
https://svn.wso2.org/repos/wso2/carbon/platform/tags/turing-<release-chunk>/
<local-platform-directory-2>
```

Replace `<release-chunk>` with the release chunk, on which the specific product version is based on. To find out the respective release chunk, see the Release Matrix. For example, for products based on Chunk 08 of WSO2 Carbon 4.2.0, the command is as follows:

```
$ svn checkout https://svn.wso2.org/repos/wso2/carbon/platform/tags/turing-chunk08/
<local-platform-directory-2>
```

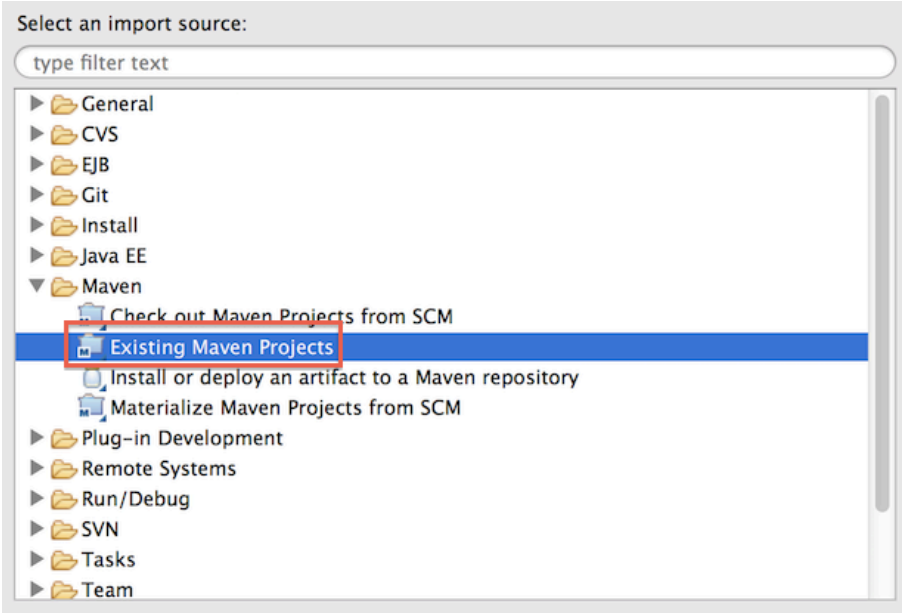
Setting up your development environment

Before you edit the source code in your IDE, set up your development environment by running one of the following commands:

If you are using this IDE...	Run this command...	Additional information
Eclipse	`mvn eclipse:eclipse`	http://maven.apache.org/plugins/maven-eclipse-plugin
IntelliJ IDEA	`mvn idea:idea`	http://maven.apache.org/plugins/maven-idea-plugin

If you are using a later Eclipse version and if you get errors (library path etc.) when trying to import the source code using the **Existing Projects into Workspace**, follow the steps below to solve them by importing the source code as a Maven project.

1. Build the source using the command: `mvn clean install`
2. Open Eclipse and click **Import** in the **File** menu and then click **Existing Maven Projects** as shown below**:**



Building the product

Follow the instructions below to build the product after editing the source code:

> Make sure the build server has an active Internet connection to download dependencies while building.

1. Install Maven and JDK. See Installation Prerequisites for compatible versions.
2. Set the environment variable `MAVEN_OPTS="-Xms1024m -Xmx4096m -XX:MaxPermSize=1024m"` to avoid the Maven `OutOfMemoryError`.
3. Navigate to each folder representing the patches within the `<local-platform-directory-1>` and run the following Apache Maven commands to build the patches. For information on the patches, which are applicable for the respective Carbon chunk release, go to Release Matrix.

This command...	Creates...
`mvn clean install`	The binary and source distributions of the chunk release.
`mvn clean install -Dmaven.test.skip=true`	The binary and source distributions, without running any of the unit tests.
`mvn clean install -Dmaven.test.skip=true -o`	The binary and source distributions, without running any of the unit tests, in offline mode. This can be done only if you've already built the source at least once.

4. For products based on Carbon 4.2.0, to create complete release artifacts of the products released with this chunk version, including the binary and source distributions, go to `<local-platform-directory-2>/product-releases/<release-chunk>/` directory and run the Apache Maven commands stated in the above step. **To build only a selected product/s** , open `<local-platform-directory-2>/product-releases/<release-chunk>/products/pom.xml` file, and comment out the products you do not want to build and run the relevant Maven command.

> After building the source, you can find the artifacts/product binary distribution package of the product in the `<local-platform-directory-2>` / `products/<product_name>/<product_release_version>/modules/distribution/target/` directory.

Committing your changes

If you are a committer, you can commit your changes using the following command (SVN will prompt you for your password):

```
$ svn commit --username your-username -m "A message"
```

Running the Product

To run WSO2 products, you start the product server at the command line. You can then run the Management Console application to configure and manage the product. This page describes how to run the product in the following sections:

- Starting the server
- Running the management console
- Stopping the server

> The Management Console uses the default HTTP-NIO transport , which is configured in the `catalina-server.xml` file in the `<APIM_HOME>/repository/conf/tomcat` directory. This transport must be properly configured in this file for the Management Console to be accessible.

Starting the server

To start the server, you run the script `wso2server.bat` (on Windows) or `wso2server.sh` (on Linux/Solaris) from the `bin`folder. Alternatively, you can install and run the server as a Windows service .

> To start and stop the server in the background mode of Linux, run **`wso2server.sh start`** and **`wso2server.sh stop`** commands.

1. Open a command prompt:
   - On Windows, choose **Start -> Run**, type `cmd` at the prompt, and press Enter.
   - On Linux/Solaris, establish a SSH connection to the server or log in to the text Linux console.
2. Execute one of the following commands, where `<APIM_HOME>` is the directory where you installed the product distribution:
   - On Windows: `<APIM_HOME>/bin/wso2server.bat --run`
   - On Linux/Solaris: `sh <APIM_HOME>/bin/wso2server.sh`

   > If you want to provide access to the production environment without allowing any user group (including admin) to log into the management console, execute one of the following commands.
   >
   > - On Windows: `<PRODUCT_HOME>\bin\wso2server.bat --run -DworkerNode`
   > - On Linux/Solaris: `sh <PRODUCT_HOME>/bin/wso2server.sh -DworkerNode`
   >
   > If you want to check any additional options available to be used with the startup commands, type `-help` after the command, such as: `sh <PRODUCT_HOME>/bin/wso2server.sh -help`.

The operation log appears. When the product server is running, the log displays the message "WSO2 Carbon started in 'n' seconds."

Running the management console

Once the server has started, you can run the Management Console by opening a Web browser and typing in the management console's URL. The URL is displayed as the last line in the start script's console and log. For example:

```
[2014-12-04 17:53:26,547]  INFO {org.wso2.carbon.core.internal.StartupFinalizerServiceComponent} -  WSO2 Carbon started in 45 sec
[2014-12-04 17:53:26,787]  INFO {org.wso2.carbon.ui.internal.CarbonUIServiceComponent} -  Mgt Console URL  : https://localhost:9443/carbon/
```

The URL should be in the following format: `https://<Server Host>:9443/carbon`

You can use this URL to access the Management Console on this computer from any other computer connected to the Internet or LAN. When accessing the Management Console from the same server where it's installed, you can type "localhost" instead of the IP address: `https://localhost:9443/carbon`.

At the sign-in screen, sign in to the Management Console using **admin** as both the username and password. You can then use the Management Console to manage the product. The tabs and menu items in the navigation pane on the left may vary depending on the features installed.

> To change or recover the admin password, see following FAQs:
> - How do I change the default admin password and what files should I edit after changing it?
> - How can I recover the admin password used to log in to the management console?

To view information about a particular page, click the **Help** link in the top right corner of that page , or click the **Docs** link to open this documentation for full information on managing the product.

> When the Management Console Sign-in page appears, the web browser will typically display an "insecure connection" message, which requires your confirmation before you can continue.
>
> The Management Console is based on HTTPS protocol, which is a combination of HTTP and SSL protocols. This protocol is generally used to encrypt the traffic from the client to server for security reasons. The certificate it works with is used for encryption only, and does not prove the server identity, so when you try to access the Management Console, a warning of untrusted connection is usually displayed. To continue working with this certificate, some steps should be taken to "accept" the certificate before access to the site is permitted. If you are using the Mozilla Firefox browser, this usually occurs only on the first access to the server, after which the certificate is stored in the browser database and marked as trusted. With other browsers, the insecure connection warning might be displayed every time you access the server.
>
> This scenario is suitable for testing purposes, or for running the program on the company's internal networks. If you want to make the Management Console available to external users, your organization should obtain a certificate signed by a well-known certificate authority, which verifies that the server actually has the name it is accessed by and that this server belongs to the given organization.

If you leave the Management Console unattended, the session will time out. The default timeout value is 15 minutes, but you can change this in the `<APIM_HOME>/repository/conf/tomcat/carbon/WEB-INF/web.xml` file as follows:

```
<session-config>
   <session-timeout>15</session-timeout>
</session-config>
```

Stopping the server

To stop the server, press **Ctrl+C** in the command window, or click the **Shutdown/Restart** link in the navigation pane in the Management Console.

Quick Start Guide

WSO2 API Manager is a complete solution for publishing APIs, creating and managing a developer community and for routing API traffic in a scalable manner. It leverages the integration, security and governance components from the WSO2 Enterprise Service Bus, WSO2 Identity Server, and WSO2 Governance Registry. In addition, as it is powered by the WSO2 Business Activity Monitor (BAM), the WSO2 API Manager is ready for massively scalable deployment immediately.

This guide walks you thorough the main usecases of the API Manager:

- Introduction to basic concepts

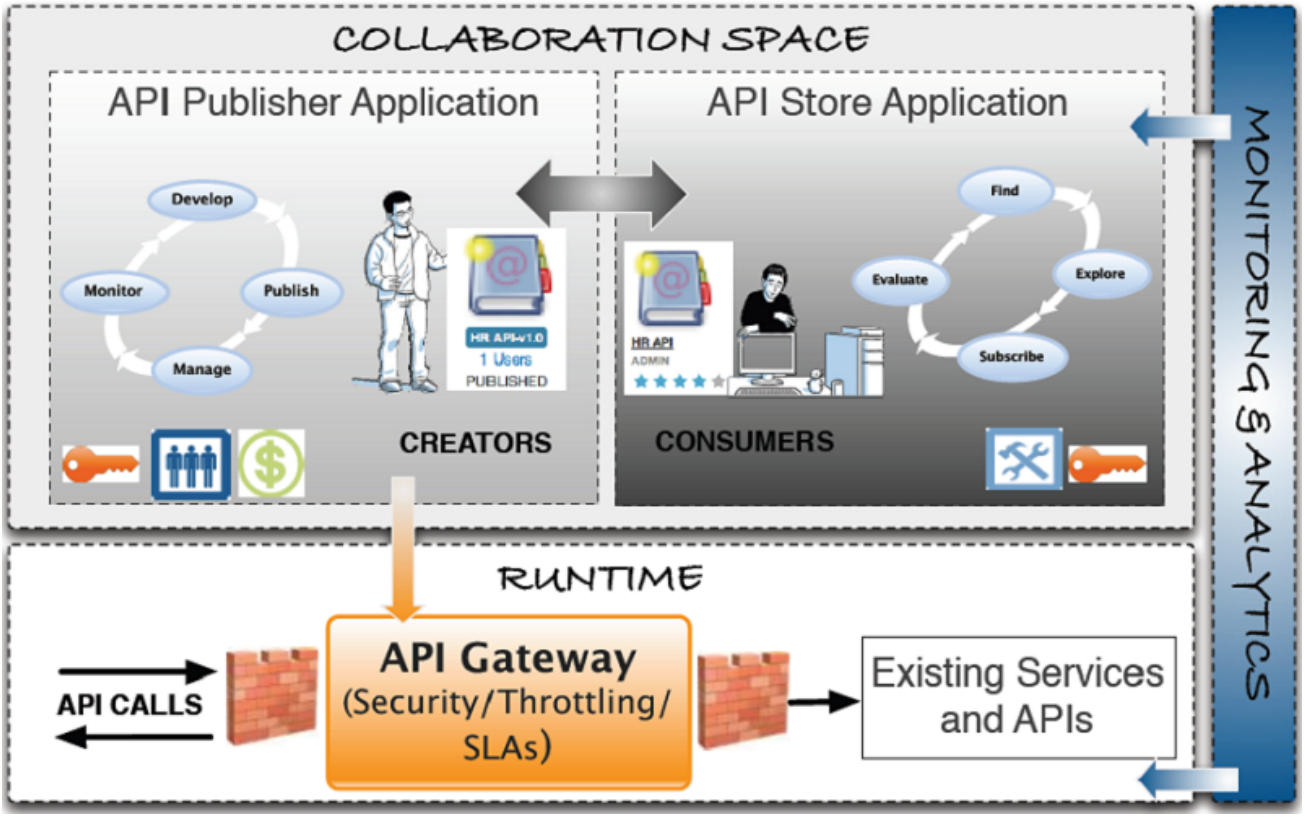
Introduction to basic concepts

Let's take a look at the basic concepts that you need to know before using the API Manager.

Components

The API manager comprises the following components:

- **API Gateway** : Secures, protects, manages, and scales API calls. It is a simple API proxy that intercepts API requests and applies policies such as throttling and security checks. It is also instrumental in gathering API usage statistics. The Web interface can be accessed via `https://<Server Host>:9443/carbon`.
- **API Key Manager** : Handles all security and key-related operations. API gateway connects with the key manager to check the validity of OAuth tokens when APIs are invoked . Key Manager also provides a token API to generate Oauth tokens that can be accessed via the Gateway.
- **API Publisher** : Enables API providers to publish APIs, share documentation, provision API keys, and gather feedback on API features, quality and usage. The Web interface can be accessed via `https://<Server Host>:9443/publisher`.
- **API Store** : Enables API consumers to self register, discover API functionality, subscribe to APIs, evaluate them and interact with API publishers. The Web interface can be accessed via `https://<Server Host>:9443/store`.
- Additionally, statistics are provided by the monitoring component, which integrates with WSO2 BAM.



Users and roles

The API manager offers three distinct community roles that are applicable to most enterprises:

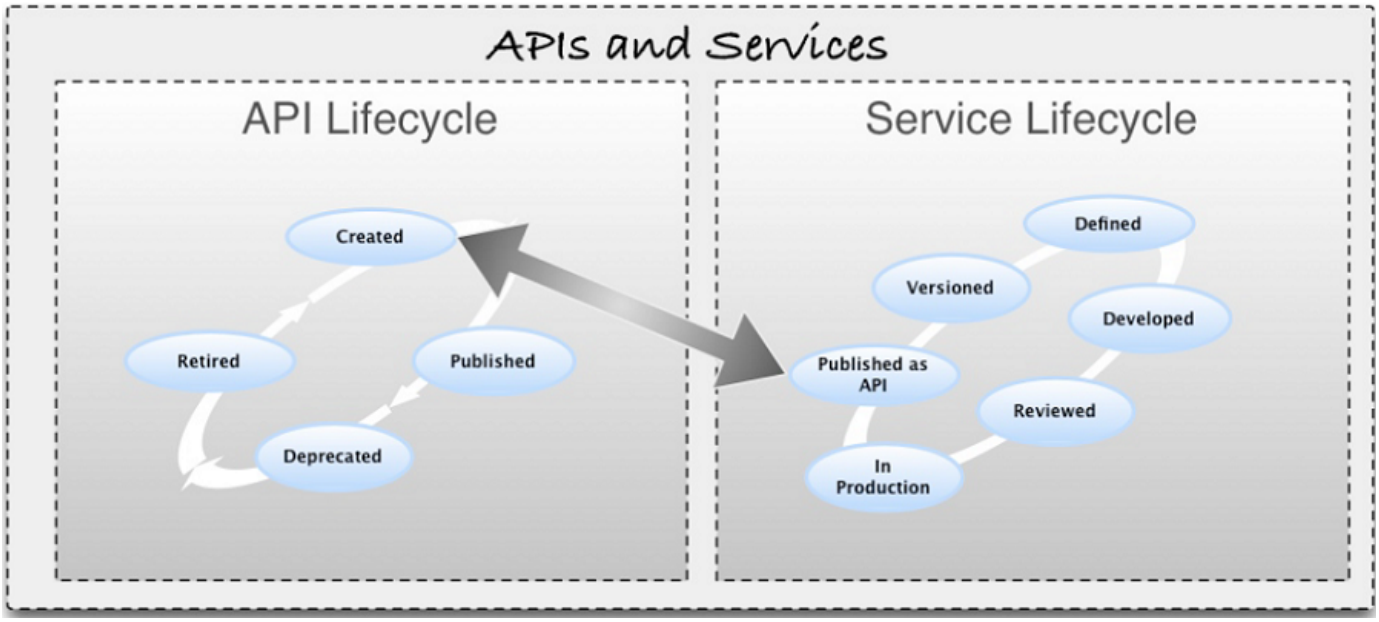
- **Creator** : a creator is a person in a technical role who understands the technical aspects of the API (interfaces, documentation, versions, how it is exposed by API Gateway) and uses the API publisher to provision APIs into the API store. The creator uses the API Store to consult ratings and feedback provided by API users. Creator can add APIs to the store but cannot manage their lifecycle (i.e., make them visible to the outside world).
- **Publisher** : a publisher manages a set of APIs across the enterprise or business unit and controls the API lifecycle and monetization aspects. The publisher is also interested in usage patterns for APIs and as such has access to all API statistics.
- **Consumer** : a consumer uses the API store to discover APIs, see the documentation and forums and rate/comment on the APIs. S/he subscribes to APIs to obtain API keys.

API lifecycle

An API is the published interface, while the service is the implementation running in the backend. APIs have their own lifecycles that are independent to the backend services they rely on. This lifecycle is exposed in the API publisher Web interface and is managed by the API publisher role.

The following stages are available in the default API life cycle:

- **CREATED** : API metadata is added to the API Store, but it is not visible to subscribers yet, nor deployed to the API gateway
- **PROTOTYPED** : API is deployed and published in the API Store as a prototype. A prototyped API is usually a mock implementation made public in order to get feedback about its usability. Users cannot subscribe to a prototyped API. They can only try out its functionality.
- **PUBLISHED** : API is visible in the API Store and available for subscription.
- **DEPRECATED** : API is still deployed into the API Gateway (i.e., available at runtime to existing users) but not visible to subscribers. An API can automatically be deprecated when a new version is published.
- **RETIRED** : API is unpublished from the API gateway and deleted from the store
- **BLOCKED** : Access is temporarily blocked. Runtime calls are blocked and the API is not shown in the API Store anymore.



You can manage the API and service lifecycles in the same governance registry/repository and automatically link them. This feature is available in WSO2 Governance Registry (version 4.5 onwards).

Applications

An application is primarily used to decouple the consumer from the APIs. It allows you to :

- Generate and use a single key for multiple APIs
- Subscribe multiple times to a single API with different SLA levels

You create an application to subscribe to an API. The API Manager comes with a default application and you can also create as many applications as you like.

Throttling tiers

Throttling tiers are associated to an API at subscription time. They define the throttling limits enforced by the API gateway. E.g., 10 TPS (transactions per second). You define the list of tiers that are available for a given API at the publisher level. The API Manager comes with three predefined tiers (`Gold/Silver/Bronze`) and a special tier called `Unlimited`, which can be disabled by editing the `<TierManagement>`element of `<PRODUCT_HOME>`/repository/conf/api-manager.xml file. To edit existing tiers or create your own tiers, see Adding New Throttling Tiers.

API keys

The API Manager supports two scenarios for authentication:

1. An access token is used to identify and authenticate a whole application
2. An access token is used to identify the final user of an application (for example, the final user of a mobile application deployed on many different devices)

Application access token

Application access tokens are generated by the API consumer and must be passed in the incoming API requests. The API Manager uses OAuth2 standard to provide key management. The API key is a simple string that you pass to an HTTP header (e.g., "`Authorization: Bearer NtBQkXoKElu0H1a1fQ0DWfo6IX4a`") and it works equally well for SOAP and REST calls.

Application access tokens are generated at the application level and valid for all APIs that are associated to the application. These tokens have a fixed expiration time, which is set to 60 minutes by default. You can change this to a longer time, even for several weeks. Consumers can re-generate the access token directly from the API Store Web interface. To change the default expiration time, you open `<APIM_HOME>/repository/conf/identity.xml` file and change the value for element `<ApplicationAccessTokenDefaultValidityPeriod>`. You set a negative value to `<ApplicationAccessTokenDefaultValidityPeriod>` element to never expire the application access token.

Application user access token

You can generate access tokens on demand using the token API. In case a token expires, you use the token API to refresh it.

Application user access tokens have a fixed expiration time, which is 60 minutes by default. You can update it to a longer time, such as several weeks, by editing the `<ApplicationAccessTokenDefaultValidityPeriod>` element in `<APIM_HOME>/repository/conf/identity.xml` file.

The token API takes the following parameters to generate the access token:

- Grant Type
- Username
- Password
- Scope

To generate a new access token, you issue a token API call with the above parameters where grant_type=password. The Token API then returns two tokens: an access token and a refresh token. The access token can then be stored in a session on the client side (the application itself does not need to manage users and passwords). On the API Gateway side, the access token is validated for each API call. When the token expires, you refresh the token by issuing a token API call with the above parameters where grant_type=refresh_token and passing the refresh token as a parameter.

Starting the API Manager

1. Download WSO2 API Manager from http://wso2.com/products/api-manager/.
2. Install Oracle Java SE Development Kit (JDK) version 1.6.24 or later or 1.7.*.
3. Set the JAVA_HOME environment variable.
4. Using the command line, go to <Installation directory>/bin and execute wso2server.bat (for Windows) or wso2server.sh (for Linux).
5. Wait until you see the message "WSO2 Carbon started in 'n' seconds."

   It indicates that the server started successfully. To stop the API Manager, simply hit Ctrl-C in the command window.

Creating users and roles

In section Users and roles , we introduced you to a set of users that are commonly found in many enterprises. To create these users in the API Manager, you log in to the management console as an administration user (credentials: admin/admin). The admin use can play the creator, publisher and subscriber roles described earlier. In this section, we explain how to set up these users or custom users and roles.

1. Log in to the management console user interface (`https://<hostname>:9443/carbon`) of the API Manager using admin/admin credentials.
2. Select the **Users and Roles** menu under the **Configure** menu.
3. Click **Add New Role** and provide `creator` as the role name.
4. Click **Next**.
5. Select the following permissions from the list that opens and click **Finish**.
   - Login
   - Manage > API > Create
   - Manage > Resources > Govern and all underlying permissions
6. Similarly, create the `publisher` role with the following permissions.
   - Login
   - Manage > API > Publish

   > **Tip**: As the `subscriber` role is available in the API Manager by default, you do not have to create it. If you want to create a new role with subscriber permissions, you can do so with the following permissions.
   >
   > - Login
   > - Manage > API > Subscribe
7. You can now create users for each of those roles. To do so, click the **Users and Roles** menu under the **Configure** menu.
8. Click **Users**.
9. Click **Add New User**, provide the username/password and click **Next**.
10. Select the role you want to assign to the user (e.g., `creator`, `publisher` or `subscriber`) and click **Finish**. Given below is a list of usernames and the roles we assign to them in this guide.

Username	Role
apicreator	creator
apipublisher	publisher

Repeat the steps to create at least one user for all roles.

Creating an API

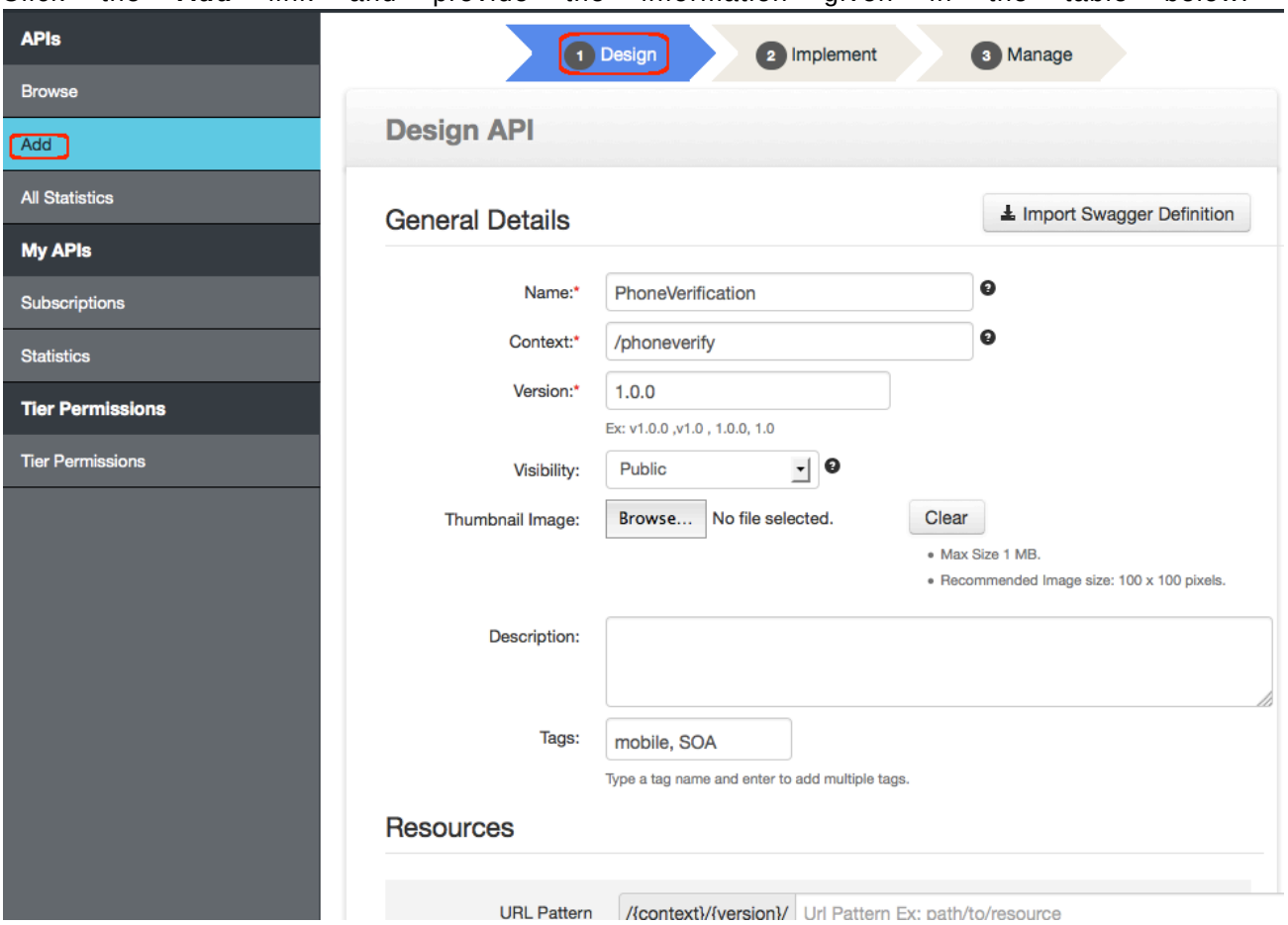
An API creator uses the API provider Web application to create and publish APIs into the API Store. In this section, we explain how to create an API and attach documentation to it.

In this guide, we work with a service exposed by the Cdyne services provider (www.cdyne.com). We use their phone

validation service, which has SOAP and REST interfaces and is documented using a WSDL file. This service is documented at : http://wiki.cdyne.com/index.php/Phone_Verification.

Let's create this API and add it to the API Store.

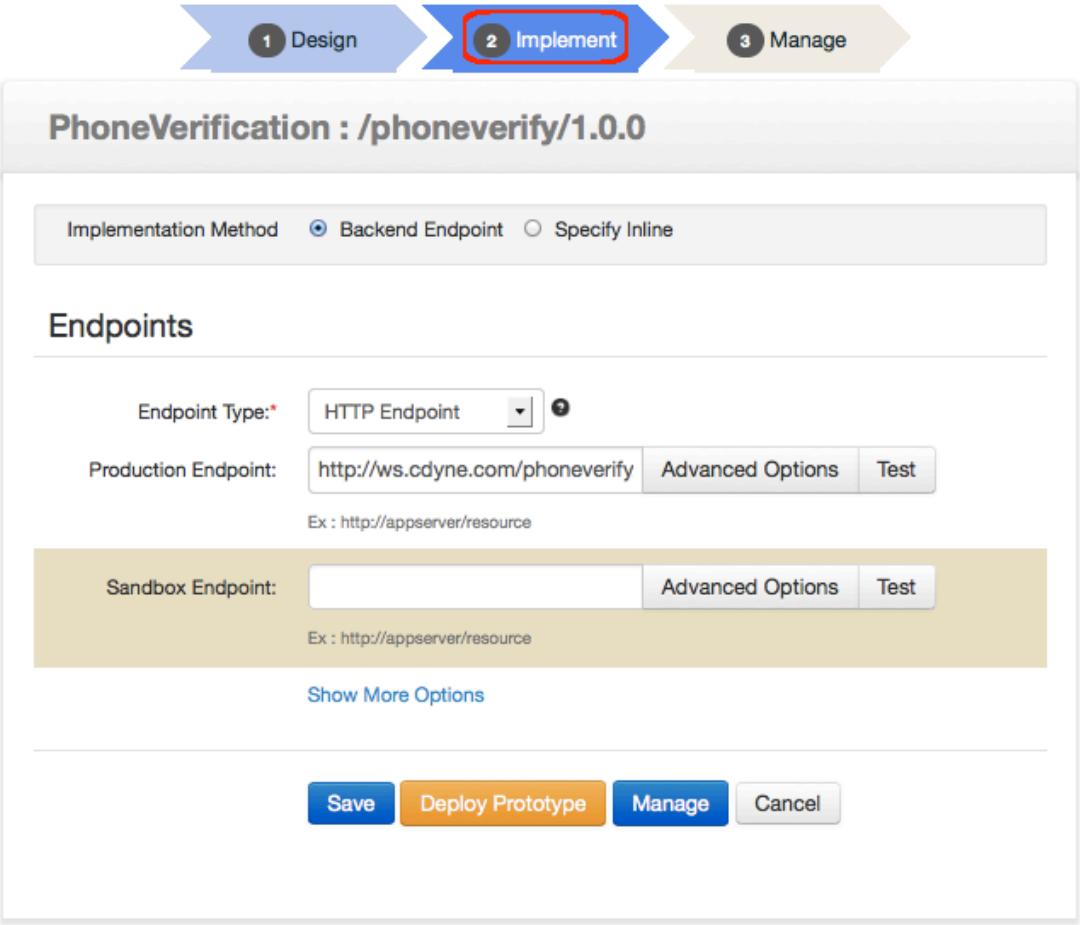
1. Open the API Publisher (`https://<hostname>:9443/publisher`) and log in as `apicreator`.
2. Click the **Add** link and provide the information given in the table below.



Field	Value	Description
Name	PhoneVerification	Name of API as you want it to appear in the API store
Context	/phoneverify	URI context path that is used by to API consumers
Version	1.0.0	API version (in the form of version.major.minor)

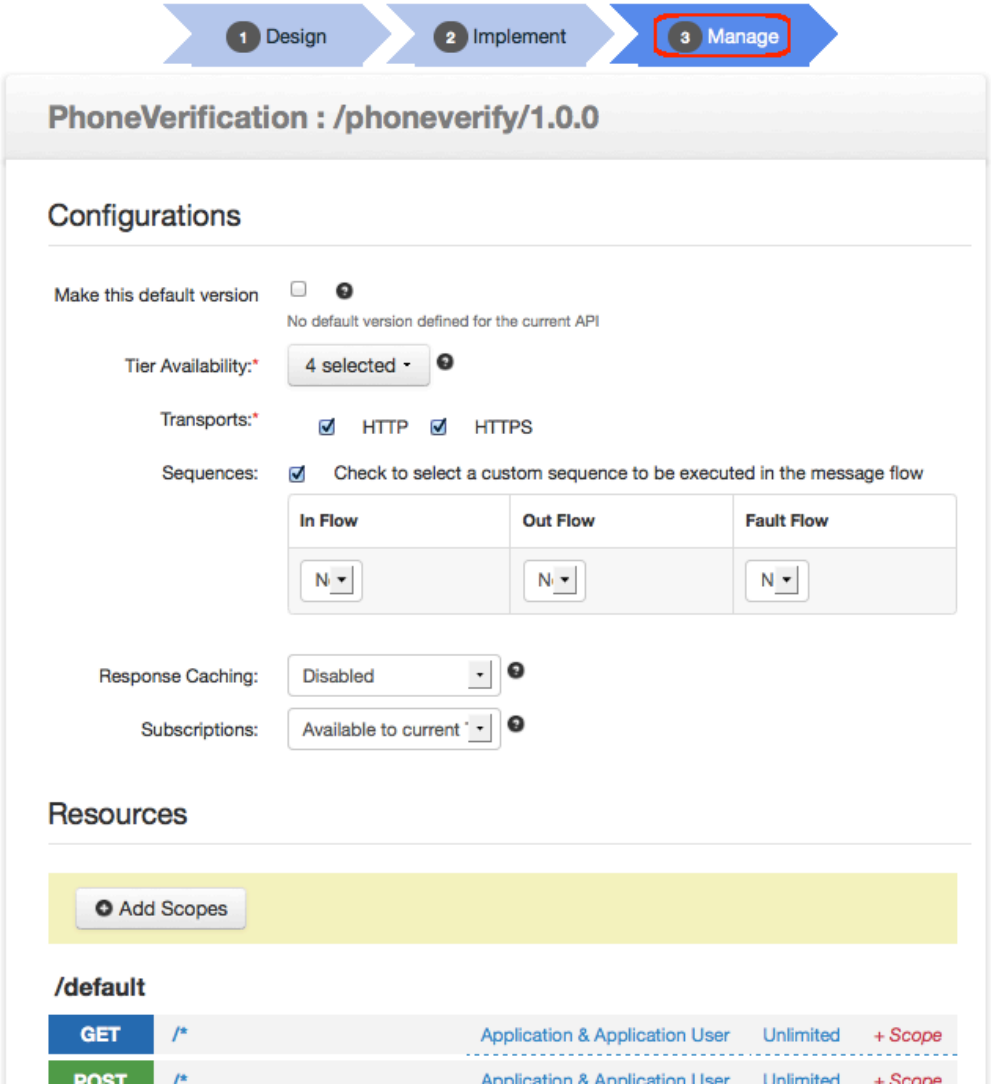
3. Click **Implement**.
4. It asks you to create a resource with wildcard characters (/*). Click **Yes**.
5. Note that a resource by the name `default` gets created as follows.

6. Click **Implement** again to go to the `Implement` tab and provide the following information.



Field	Value	Description
Implementation method	Backend endpoint	If you have a real backend implement to your API, select that option. Else, y can specify implementation in-line. Th latter approach is usually used in moc implementation for prototyped APIs.
Endpoint type	HTTP endpoint	
Production endpoint	http://ws.cdyne.com/phoneverify/phoneverify.asmx	
Endpoint security scheme	Non Secured	If the endpoint is secured, user is ask credentials of the backend service.
WSDL	URL: http://ws.cdyne.com/phoneverify/phoneverify.asmx?wsdl	URL of WSDL file (describing API inte

7. Click **Manage** to go to the `Manage` tab and provide the following information.



Field	Value	Description
Tier Availability	Bronze/Gold/Silver/Unlimited	The API can be available at different level of service; you can select multiple entries from the list. At subscription time, the consumer chooses which tier they are interested in.
Transports	HTTP/HTTPS	

API Resources

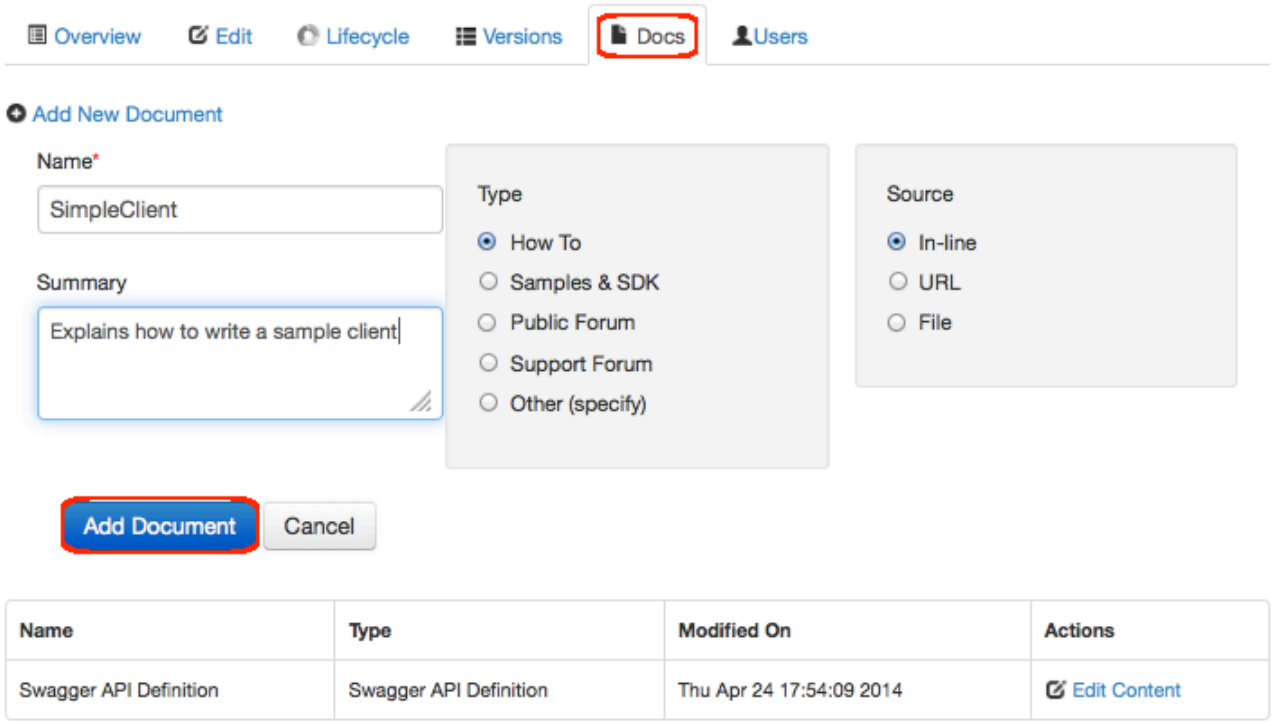
An API is made up of one or more resources. Each resource handles a particular type of request and is analogous to a method (function) in a larger API. API resources accept following optional attributes:

- verbs : Specifies the HTTP verbs a particular resource accepts. Allowed values are GET, POST, PUT, DELETE. Multiple values can be specified.
- uri-template : A URI template as defined in http://tools.ietf.org/html/rfc6570 (e.g., /phoneverify/<phoneNumber>)
- url-mapping : A URL mapping defined as per the servlet specification (extension mappings, path mappings and exact mappings)
- Throttling tiers : Limits the number of hits to a resource during a given period of time. For more information, see Managing Throttling Tiers.
- Auth-Type: Specifies the Resource level authentication along HTTP verbs. Auth-type can be None, Application or Application User.
    - None : Can access the particular API resource without any access tokens

- Application: Application access token required to access the API resource
- Application User: User access token required to access the API resource

Adding API documentation

1. After creating the API, click on its icon to open its details. Select the Docs tab.
2. Click **Add New Document** link.



Documentation can be provided inline, via a URL or as a file. For inline documentation, you can edit the content directly from the API publisher interface. You get several documents types:

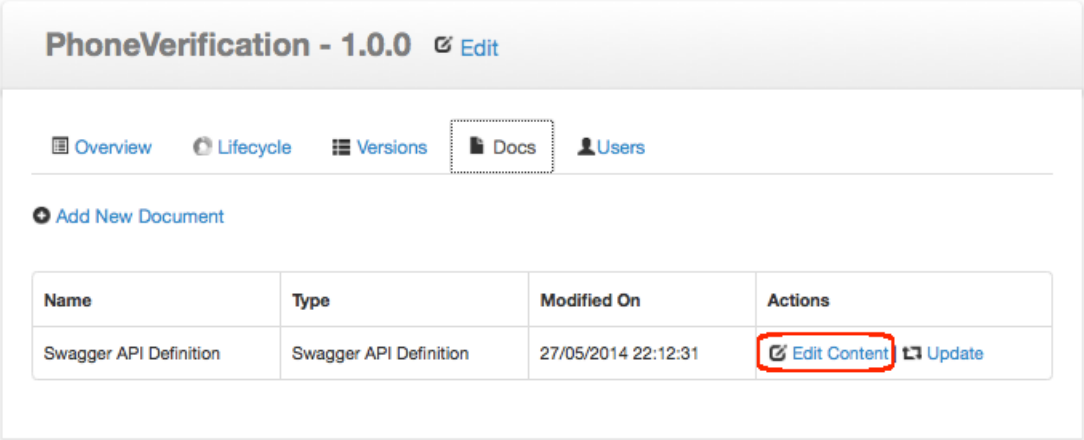
- Swagger documents
- How To
- Samples and SDK
- Public forum / Support forum (external link only)
- API message formats
- Other

3. Select the **How To** type, a name for the document and a short description, which will appear in the API Store. Select inline or provide a URL.
4. Click **Add Document**.
5. Once the document is added, click **Edit Content** link, which opens an embedded editor to edit the document contents.

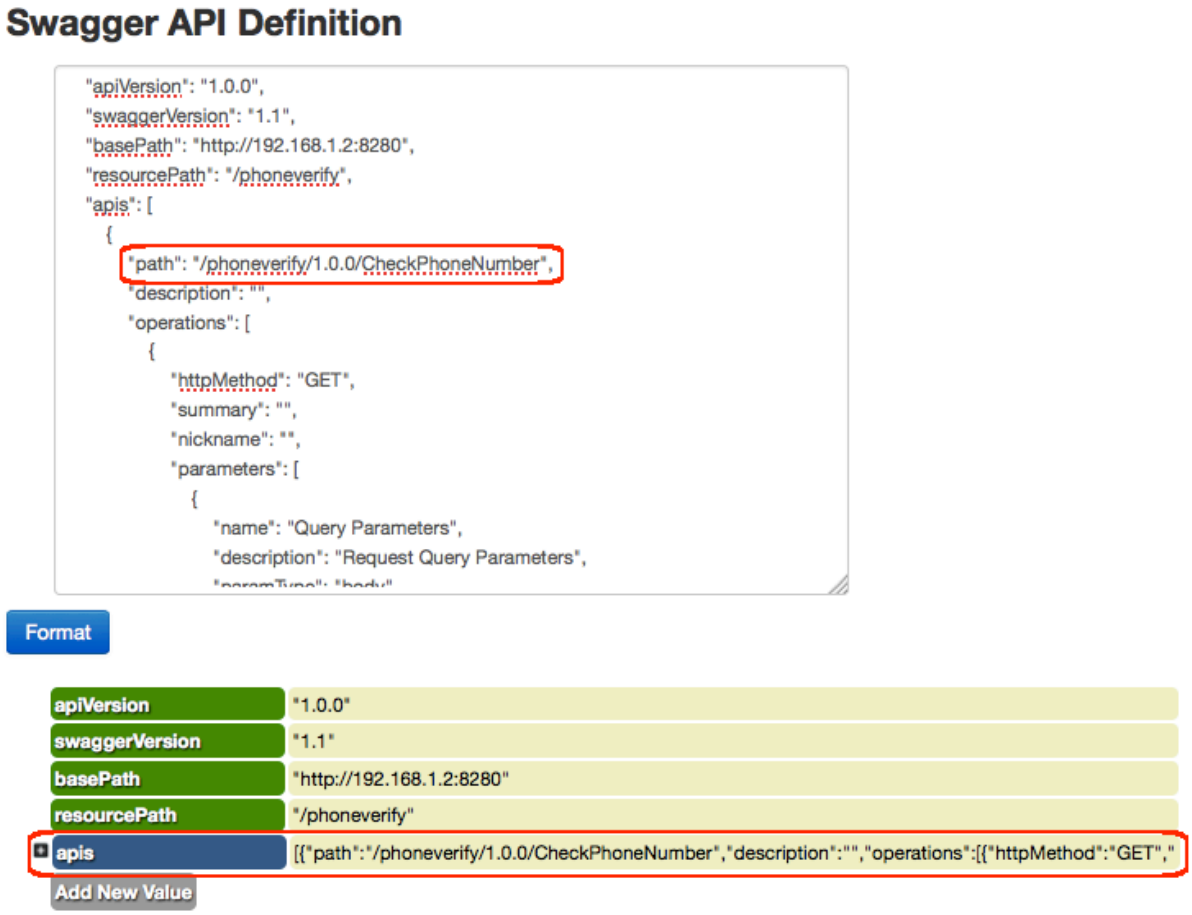
Adding interactive documentation using Swagger

The API Manager provides facility to add interactive documentation support through the integration of Swagger. Swagger is a specification and a complete framework implementation for describing, producing, consuming, and visualizing RESTful Web services. In Swagger, when APIs are described in simple static JSON representation, they can be loaded through the Swagger UI, which in turn provides the interactive documentation.

When an API is created, the JSON representation of that API is automatically generated and saved into the registry as API definition. This definition describes the API with the information provided at the API creation level. You can customize the automatically generated API definition by going to the Doc tab of the `PhoneVerification` API in the API Publisher.



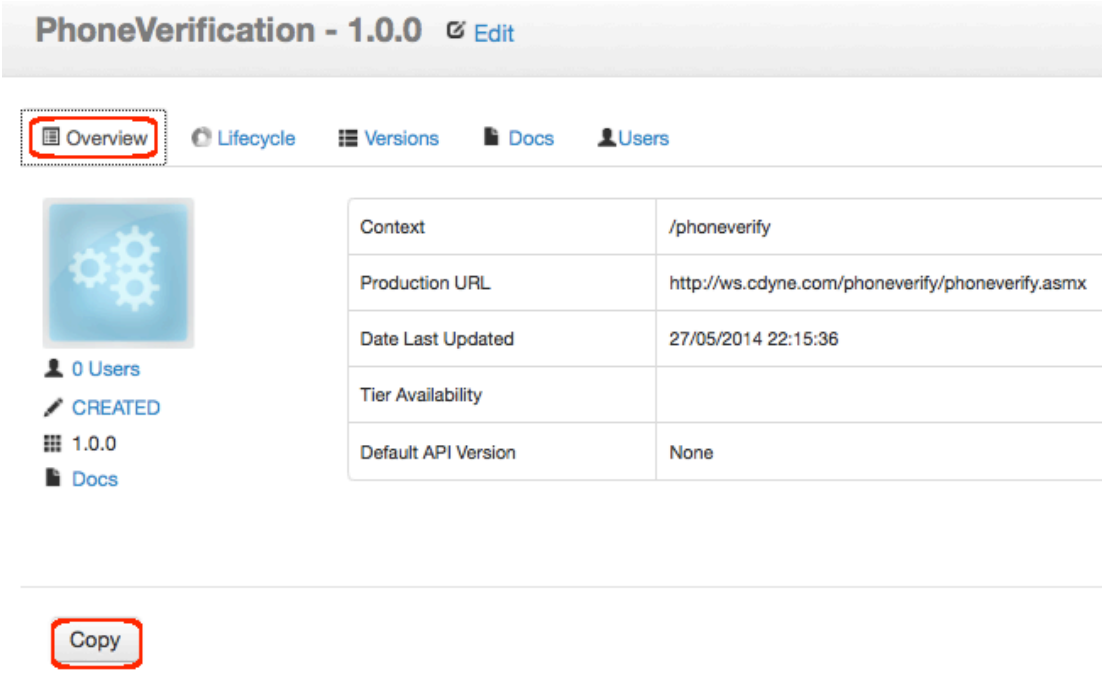
You can modify the paths, parameters, descriptions etc. by editing the JSON representation of API definition. For example, in the `PhoneVerification` API, we have changed the path for all the HTTP methods of API definition from `/phoneverify/1.0.0/` to `/phoneverify/1.0.0/CheckPhoneNumber` as follows:



Versioning the API

Next, we will create a new version of this API.

1. Log in to the API Publisher as `apicreator` if you are not logged in already.
2. Click on the `PhoneVerification` API and then the **Copy** button that appears in its Overview tab.

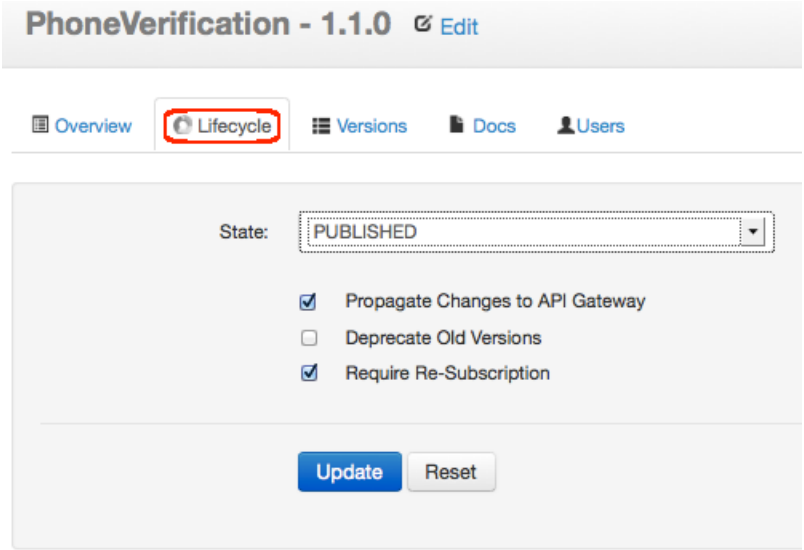


3. Specify a new version number in version.major.minor format (e.g., 1.1.0) and click **Done**.

A new version of the API is created. It is a duplication of all the contents of the original API, including the documentation. The API is now ready to be published. This is done by a user in the publisher role.

Publishing the API

1. Log in to the API Publisher Web application as apipublisher.
2. Click on the `PhoneVerification` API version 1.1.0 that you created before. Note that you can now see a tab as **API Lifecycle** in the API Publisher UI.
3. Go to the **Lifecycle** tab and select the state as `PUBLISHED` from the drop-down list.



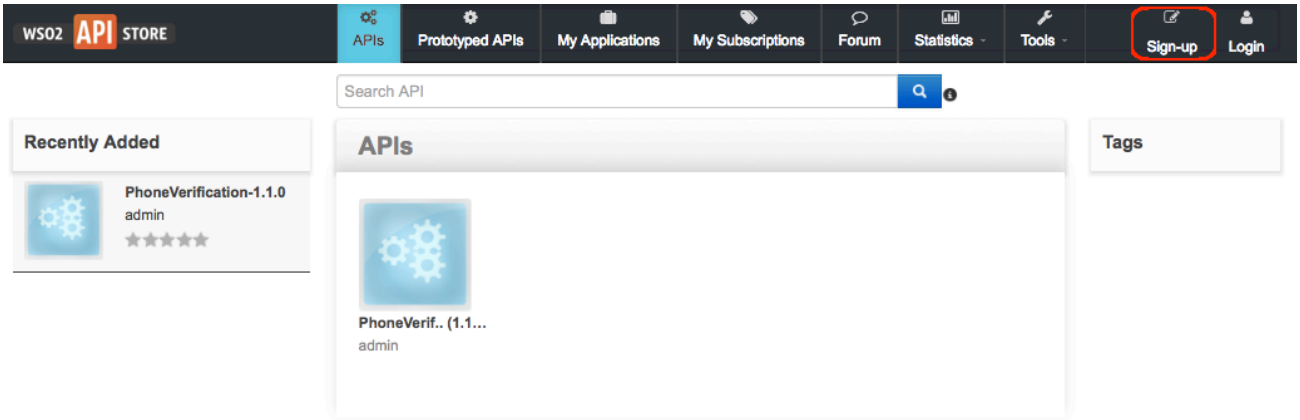
- **Propagate Changes to API Gateway**: Used to define an API proxy in the API Gateway runtime component, allowing the API to be exposed to the consumers via the API Gateway. If this option is left unselected, the API metadata will not change and you will have to manually configure the API Gateway according to the information published in the API Store.
- **Deprecate Old Versions**: If selected, any prior versions of the API will be set to the DEPRECATED state automatically.
- **Require Re-Subscription**: Invalidates current user subscriptions, forcing users to subscribe again.

The API is now published and visible to consumers in the API store.

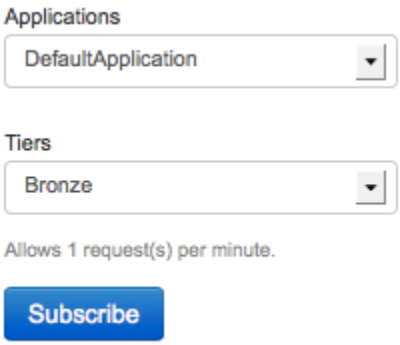
Subscribing to the API

You subscribe to APIs using the API Store Web application.

1. Open the API Store (`https://<hostname>:9443/store`) using your browser. Using the API Store, you can,
   - Search and browse APIs
   - Read documentation
   - Subscribe to APIs
   - Comment on, rate and share/advertize APIs
   - Take part in forums and request features etc.
2. The API you published earlier is available in the API Store. Self sign up to the API Store using the **Sign-up** link .



3. After subscription, log in to the API Store and click the API you published earlier (PhoneVerification 1.1.0).
4. Note that you can see the subscription option in the right hand side of the UI after logging in. Select the default application, `Bronze` tier and click **Subscribe**.



Applications

An application is a logical collection of one or more APIs, and is required when subscribing to an API. You can subscribe to multiple APIs using the same application. Instead of using the default application, you can also create your own by selecting the **New Application...** option in the above drop-down list or by going to the **My Applications** menu in the top menu bar.

5. Once the subscription is successful, go to **My Subscriptions** page.
6. In the My Subscriptions page, click the **Generate** buttons to generate production and sandbox access tokens and consumer key/secret pairs for the API. For more information on access tokens, see Working with Access Tokens.

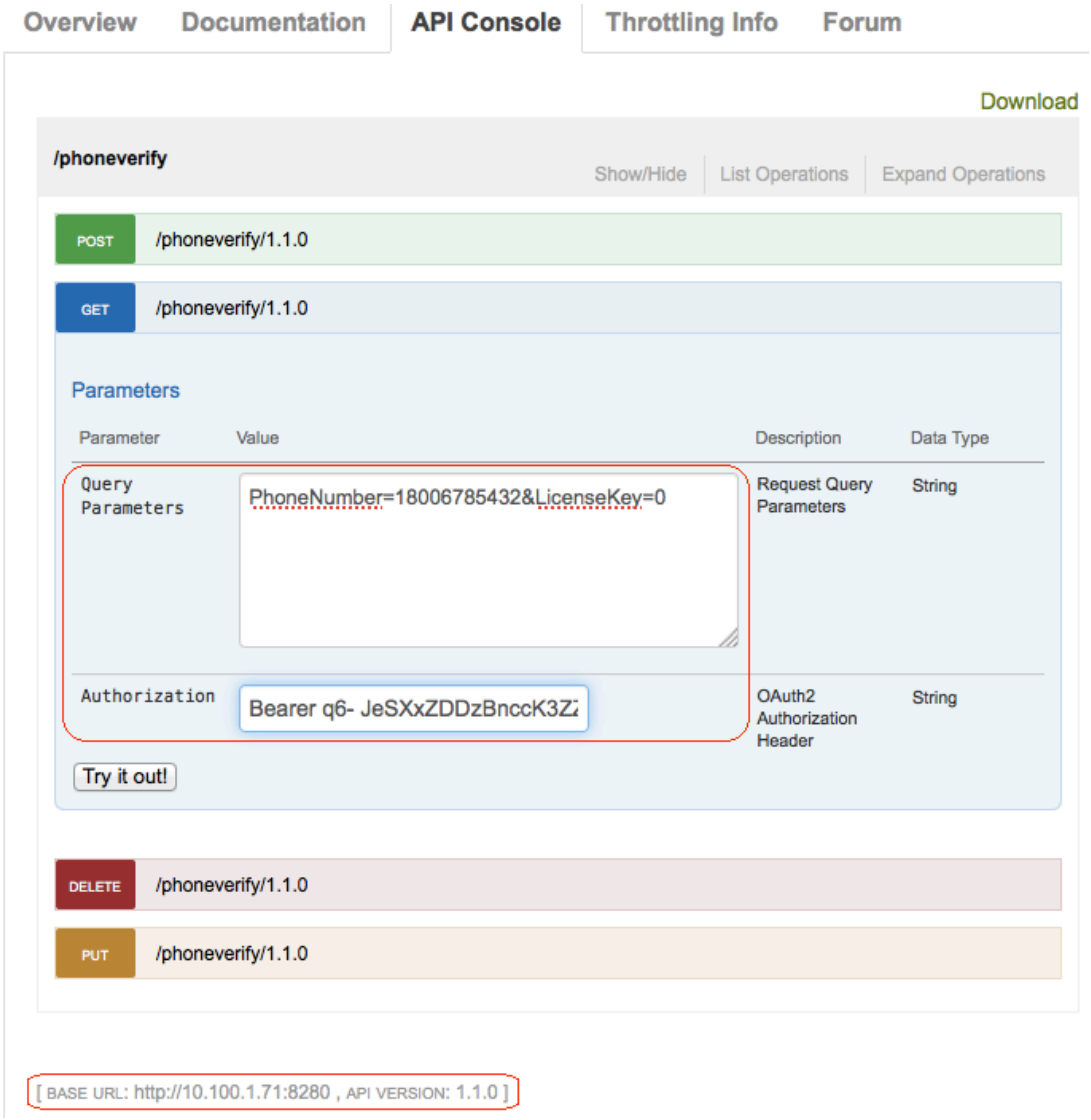
You are now successfully subscribed to the API and are ready to start using it.

Invoking the API

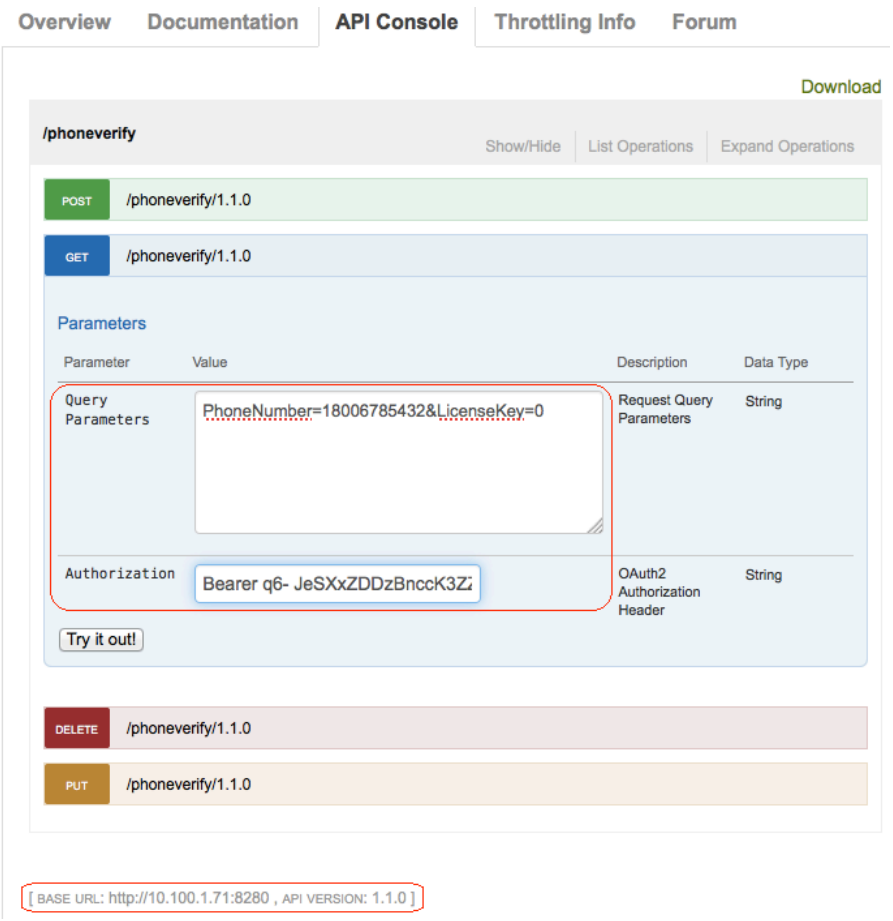
To invoke an API, you can use the integrated Swagger interactive documentation support (or any other simple

REST client application or curl).

1. Log in to the API Store (`https://<YourHostName>:9443/store`).
2. Click the `PhoneVerification 1.1.0` API that you published earlier.
3. Click the **API Console** tab associated with the API.



4. Provide the necessary parameters and click **Try it out** to call the API. For example, the `PhoneVerification` API takes two parameters: the phone number and a license key, which is set to 0 for testing purposes.



Note the following in the above UI:

Base URL	Appears at the bottom of the console. Using the base URL and the parameters, the system creates the API URL in the form `http://host:8280/<context>/<version>/<back end service requirements included as parameters>`. For example, `http://host:8280/phoneverify/1.1.0/CheckPhoneNumber`.
Query Parameters	Give the API payload as PhoneNumber=18006785432&LicenseKey=0 where /phoneverify is the context and 1.1.0 is the version. The rest of the URL is driven by the backend service requirements.
Authorization	In the authorization header, pass the application key that was generated at the time a user subscribes to an API. This is prefixed by the string "Bearer". For example, Bearer q6-JeSXxZDDzBnccK3ZZGf5_AZTk. WSO2 API Manager enforces OAuth security on all the published APIs. Consumers who talk to the API Manager should send their credentials (application key) as per the OAuth bearer token profile. If you don't send an application key or send a wrong key, you will receive a 401 Unauthorized response in return.

5. The response for the API invocation appears as follows:

Request URL

```
http://10.100.1.71:8280/phoneverify/1.1.0
```

Response Body

```
<html>

    <head>
   <link rel="alternate" type="text/xml" href="/phoneverify/phoneverify.asmx?disco" />

      <style type="text/css">

            BODY { color: #000000; background-color: white; font-family: Verdana; margin-left: 0px; margin-top: 0
px; }
            #content { margin-left: 30px; font-size: .70em; padding-bottom: 2em; }
            A:link { color: #336699; font-weight: bold; text-decoration: underline; }
            A:visited { color: #6699cc; font-weight: bold; text-decoration: underline; }
            A:active { color: #336699; font-weight: bold; text-decoration: underline; }
            A:hover { color: cc3300; font-weight: bold; text-decoration: underline; }
            P { color: #000000; margin-top: 0px; margin-bottom: 12px; font-family: Verdana; }
            pre { background-color: #e5e5cc; padding: 5px; font-family: Courier New; font-size: x-small; margin-to
p: -5px; border: 1px #f0f0e0 solid; }
            td { color: #000000; font-family: Verdana; font-size: .7em; }
            h2 { font-size: 1.5em; font-weight: bold; margin-top: 25px; margin-bottom: 10px; border-top: 1px solid
#003366; margin-left: -15px; color: #003366; }
            h3 { font-size: 1.1em; color: #000000; margin-left: -15px; margin-top: 10px; margin-bottom: 10px; }
```

Response Code

```
200
```

Response Headers

```
Content-Type: text/html; charset=utf-8
```

6. Within a minute after the first API invocation, make another attempt to invoke the API and note that the second invocation results in a throttling error.

   This is because you applied a Bronze tier at the time you subscribed to the API and the Bronze tier only allows one API call per minute.

Response Body

```
<amt:fault xmlns:amt="http://wso2.org/apimanager/throttling">
  <amt:code>900800</amt:code>
  <amt:message>Message Throttled Out</amt:message>
  <amt:description>You have exceeded your quota</amt:description>
</amt:fault>
```

Monitoring APIs and viewing statistics

Both the API publisher and store provide several statistical dashboards. Some of them are as follows:

- Number of subscriptions per API (across all versions of an API)

- Number of API calls being made per API (across all versions of an API)
- The subscribers who did the last 10 API invocations and the APIs/versions they invoked
- Usage of an API and from which resource path (per API version)
- Number of times a user has accessed an API
- The number of API invocations that failed to reach the endpoint per API per user
- API usage per application
- Users who make the most API invocations, per application
- API usage from resource path, per application

Configuring statistics

Steps below explain how to configure WSO2 BAM 2.4.1 with the API Manager.

1. Do the following changes in `<APIM_HOME>/repository/conf/api-manager.xml` file:
   - Enable API usage tracking by setting the `<APIUsageTracking>` element to true
   - Set the Thrift port to 7614
   - Set <BAMServerURL> to tcp://<BAM host IP>:7614/ where <BAM host IP> is the machine IP address. Do not use localhost unless you're in a disconnected mode.

```
<APIUsageTracking>
    <!-- Enable/Disable the API usage tracker. -->
    <Enabled>true</Enabled>

<PublisherClass>org.wso2.carbon.apimgt.usage.publisher.APIMgtUsageDataBridgeDataP
ublisher</PublisherClass>
    <ThriftPort>7614</ThriftPort>
    <BAMServerURL>tcp://<BAM host IP>:7614/</BAMServerURL>
    <BAMUsername>admin</BAMUsername>
    <BAMPassword>admin</BAMPassword>
    <!-- JNDI name of the data source to be used for getting BAM statistics. This
data source should
        be defined in the master-datasources.xml file in conf/datasources
directory. -->
    <DataSourceName>jdbc/WSO2AM_STATS_DB</DataSourceName>
</APIUsageTracking>
```

2. Specify the datasource definition in `<APIM_HOME>/repository/conf/datasources/master-datasources.xml` file as follows.

```
<datasource>
          <name>WSO2AM_STATS_DB</name>
          <description>The datasource used for getting statistics to API
Manager</description>
          <jndiConfig>
             <name>jdbc/WSO2AM_STATS_DB</name>
          </jndiConfig>
          <definition type="RDBMS">
             <configuration>
                 <!-- JDBC URL to query the database -->

<url>jdbc:h2:<BAM_HOME>/repository/database/APIMGTSTATS_DB;AUTO_SERVER=TRUE</url>
                 <username>wso2carbon</username>
                 <password>wso2carbon</password>
                 <driverClassName>org.h2.Driver</driverClassName>
                 <maxActive>50</maxActive>
                 <maxWait>60000</maxWait>
                 <testOnBorrow>true</testOnBorrow>
                 <validationQuery>SELECT 1</validationQuery>
                 <validationInterval>30000</validationInterval>
             </configuration>
         </definition>
</datasource>
```

Next, prepare BAM to collect and analyze statistics from API manager.

3. Download WSO2 BAM 2.4.1 or later from location: http://wso2.com/products/business-activity-monitor.
4. Change port offset of BAM to **3** by editing the file `<BAM_HOME>/repository/conf/carbon.xml` file (search for the offset node).

```
<Offset>3</Offset>
```

This increments all ports used by the server by 3, which means the BAM server will run on port 9446. Port offset is used to increment the default port by a given value. It avoids possible port conflicts when multiple WSO2 products run in same host.

5. Do the following changes in `<BAM_HOME>/repository/conf/datasources/bam_datasources.xml` file:
   - Copy/paste `WSO2_AMSTATS_DB` definition from API Manager's `master-datasources.xml` file. You edited it in step 2.
   - Replace the port of `WSO2BAM_CASSANDRA_DATASOURCE` in URL (`jdbc:cassandra://localhost:9163/EVENT_KS`). Note that localhost is used here; not the machine IP.

   > - Do not edit the `WSO2BAM_UTIL_DATASOURCE`, which is using the offset
   > - Cassandra is bound by default on localhost, unless you change the data-bridge/data-bridge-config.xml file

6. Copy the file `<APIM_HOME>/statistics/API_Manager_Analytics.tbox` to directory `<BAM_HOME>/repository/deployment/server/bam-toolbox`.

   If this folder is not in the BAM installation directory by default, create it. The toolbox describes the information collected, how to analyze the data, as well as the location of the database where the analyzed data is stored.
7. Open `<BAM_HOME>/repository/conf/etc/hector-config.xml` file and change the port to localhost:9163. You must add the other nodes too when configuring a clustered setup.

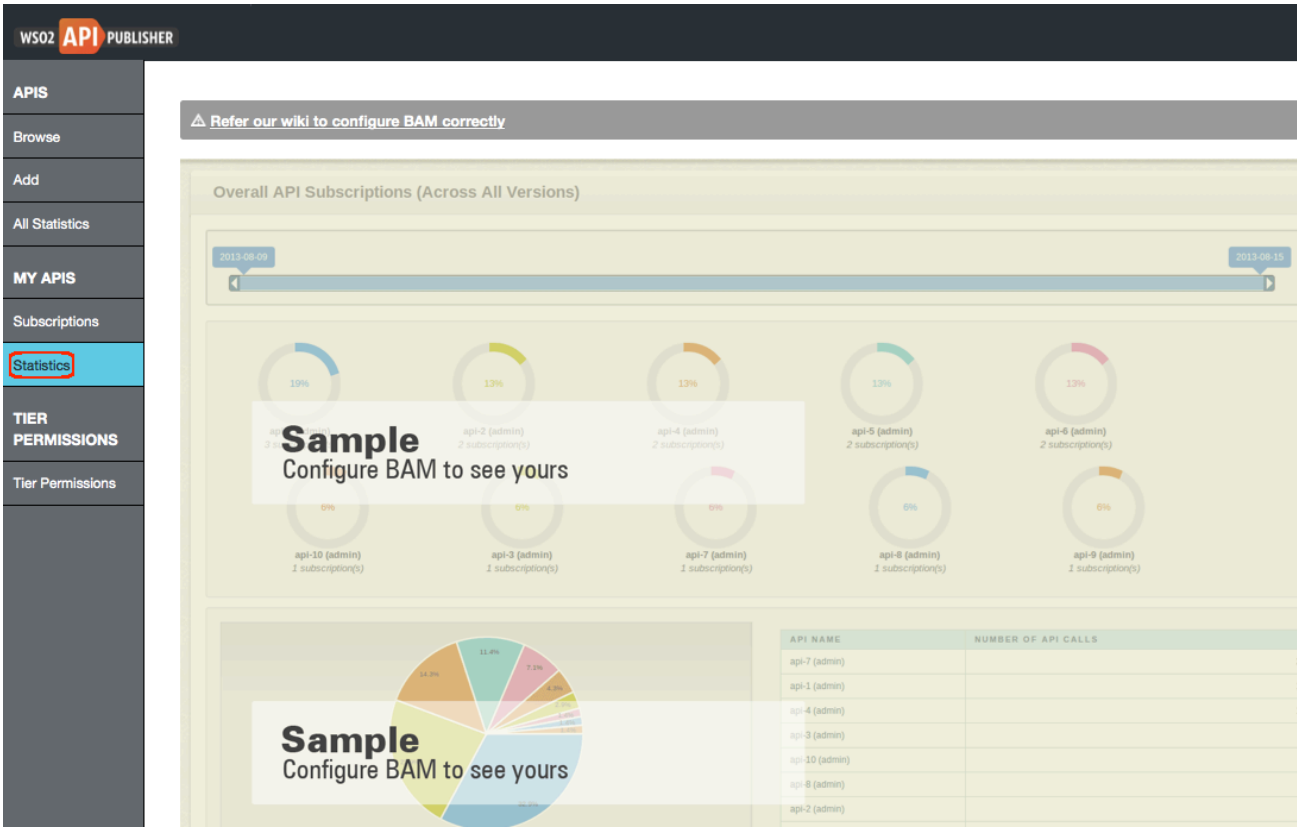
```
<Nodes>localhost:9163</Nodes>
```

8. Restart the BAM server by running `<BAM_HOME>/bin/wso2server.[sh/bat]`.

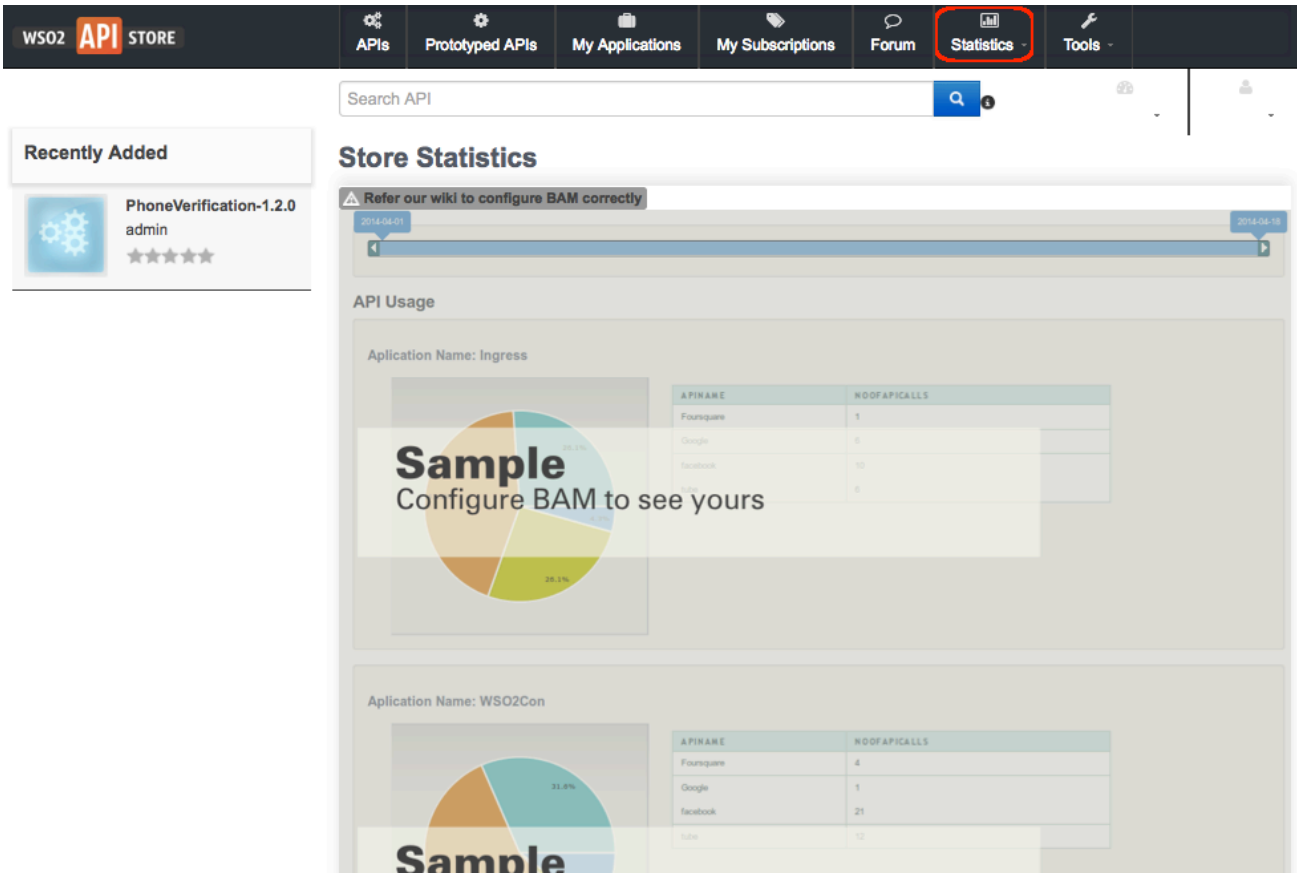
Viewing statistics

To see statistics, you first generate some traffic via the API Gateway (invoke the Cdyne API we use in this guide) and wait a few seconds. Then, follow these steps:

1. Connect to the API Publisher as a creator or publisher.
   In publisher role, you are able to see all stats and as creator, you see stats specific to the APIs you create.
2. Click the **Statistics** menu. We show the sample statistics here, but you will see graphs specific to your instance.



3. Similarly, API subscribers can also see statistics though the API Store. Click the **Statistics** menu as follows:



For more information, see Viewing API Statistics.

This concludes the API Manager quick start. You have set up the API Manager and taken a look at its common usecases. For more advanced usecases, please see the User Guide and the Admin Guide of the API Manager documentation.

Upgrading from the Previous Release

The following information describes how to upgrade your API Manager server from the release, which is APIM 1.6.0. To upgrade from a version older than 1.6.0, start from the doc that was released immediately after your current release and upgrade incrementally.

- Upgrading the product databases
- Migrating the configurations
- Upgrading APIM 1.6.0 to 1.7.0

Before you begin,

1. Stop all running API Manager server instances.
2. Download API Manager 1.7.0 from http://wso2.com/products/api-manager.
3. Replace all the files in `<APIM_1.7.0_HOME>/dbscripts/migration-1.6.0_to_1.7.0` folder with the files in this SVN location. This is because some of the scripts in the 1.7.0 distribution give migration issues. We have fixed this in APIM 1.8.0.

Upgrading the product databases

1. Back up the databases of your API Manager 1.6.0 server instance.
2. Go to `<APIM_1.7.0_HOME>/dbscripts/migration-1.6.0_to_1.7.0` and run the database upgrade scripts on your old database. You must select the script corresponding to your database type. For example, if your database is MySQL, execute `<APIM_1.7.0_HOME>/dbscripts/migration-1.6.0_to_1.7.0/mysql.sql` on it. The script adds all the schema changes done to API Manager tables in the 1.7.0 release.

> **Tip**: Do not use the migration scripts that are bundled in your product distribution as they cause migration issues. These issues are fixed in APIM 1.8.0 release.

3. Point the WSO2 Carbon Database(User Store and Registry) and API Manager Databases of your AM 1.6.0 instance to AM 1.7.0. (Configure AM_1.6.0/repository/datasource/master-datasources.xml to point same databases configured in AM 1.6.0)

Migrating the configurations

In this section, you move all existing API Manager configurations from the current environment to the new one.

1. Open `<APIM_1.7.0_HOME>/repository/conf/datasources/master-datasources.xml` file and copy the datasource configurations for the following databases from the same file in the APIM 1.6.0 instance over to the 1.7.0 instance.
   - User store/s
   - Registry database
   - APIM databases
2. Move all your synapse configurations by copying and replacing `<APIM_1.6.0_HOME>/repository/deployment/server/synapse-config/default` directory to `<APIM_1.7.0_HOME>/repository/deployment/server/synapse-config/default` directory.

> **Tip**: If you changed the default URLs in `AuthorizeAPI.xml` and `TokenAPI.xml` files, do not replace them when copying. They are application-specific APIs.

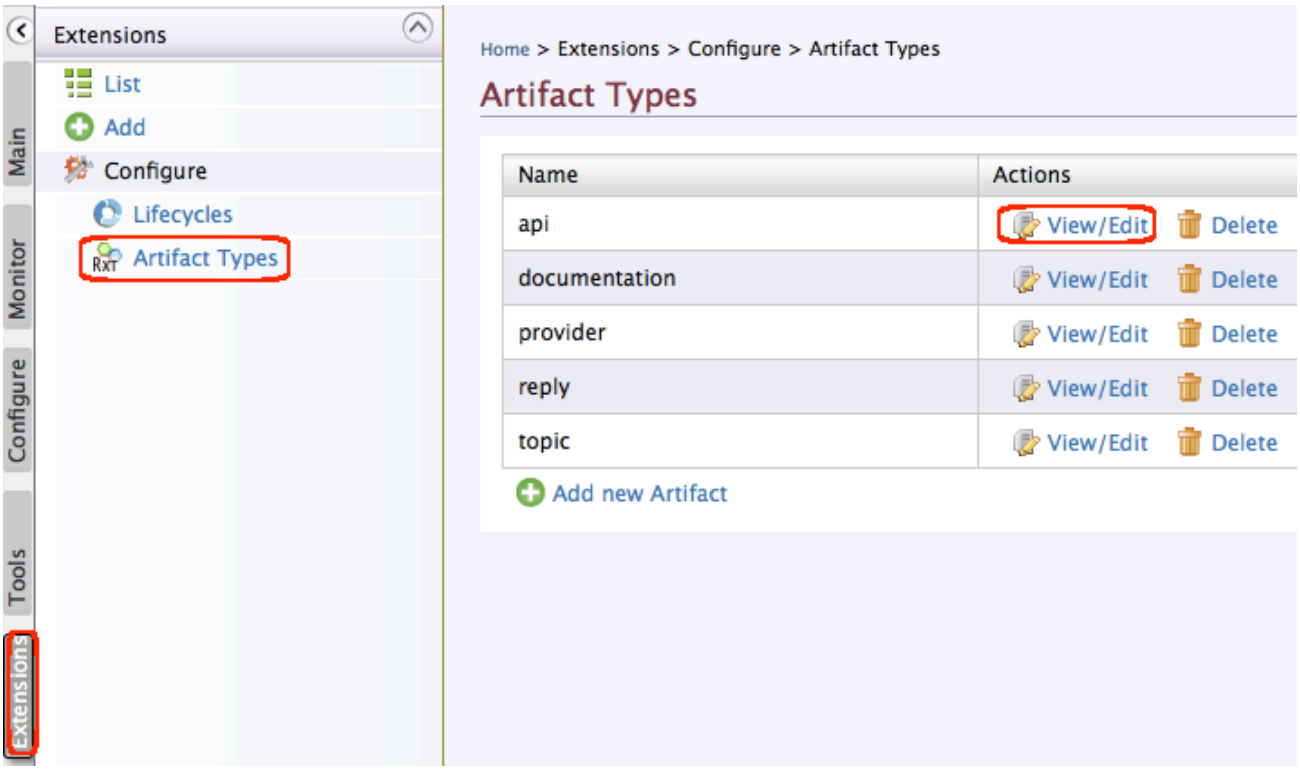
3. Copy the `<APIM_1.7.0_HOME>/dbscripts/migration-1.6.0_to_1.7.0/api-migration` directory to `<APIM_1.7.0_HOME>`. Once done, you have the `<APIM_1.7.0_HOME>/api_migration` directory path.
4. Add the following property to `<APIM_1.7.0_HOME>/api-migration/build.xml`.

```
apim.home= Path to your APIM 1.7.0 distribution location (If you have a
distributed setup, give the path to the Gateway node)
```

5. Go inside the `api-migration` directory and execute `ant run`. You should get a `BUILD SUCCESSFUL` message.

Upgrading APIM 1.6.0 to 1.7.0

1. Start the API Manager 1.7.0 and log in to its management console.
2. Select **Extensions -> Artifact Types** menu and click the **View/Edit** link associated with the api artifact type.



3. Replace the RXT file that opens in the management console with the content of `<APIM_1.7.0_HOME>/dbscripts/migration-1.6.0_to_1.7.0/rxt/api.rxt` file.
4. Similarly, using the management console, replace the `documentation.rxt` file with the content of `<APIM_1.7.0_HOME>/dbscripts/migration-1.6.0_to_1.7.0/rxt/documentation.rxt` file.
5. Copy the `<APIM_1.7.0_HOME>/dbscripts/migration-1.6.0_to_1.7.0/swagger-resource-migration` directory to `<APIM_1.7.0_HOME>`. Once done, you will have the `<APIM_1.7.0_HOME>/swagger-resource-migration` directory path.
6. Configure `<APIM_1.7.0_HOME>/swagger-resource-migration/build.xml` file with the following properties:

Property	Description
registry.home	Path to the APIM distribution. In a distributed setup, give the API Publisher node's path.
username	Username of the APIM server. For a tenant to log in, provide the tenant admin username.
password	Password for the server. For a tenant to log in, provide the tenant admin password.
host	IP of the running APIM server. In a distributed setup, give the host of the API Publisher node.
port	Port of the running APIM server. In a distributed setup, give the port of the APIM Publisher node.
version	Version of the APIM server.

7. Using the command line, go to `<APIM_1.7.0_HOME>/swagger-resource-migration` folder and execute **`ant run`**. If the above configuration is successful, you get a **`BUILD SUCCESSFUL`** message. It modifies the structure of Swagger content in the registry.
8. Copy the `<APIM_1.7.0_HOME>/dbscripts/migration-1.6.0_to_1.7.0/doc-file-migration` directory to `<APIM_1.7.0_HOME>`. Once done, you will have the `<APIM_1.7.0_HOME>/doc-file-migration` directory path.
9. Configure `<APIM_1.7.0_HOME>/doc-file-migration/build.xml` with the following properties.

Property	Description
registry.home	Path to the APIM distribution. In a distributed setup, give the API Publisher node's path.
username	Username of the APIM server
password	Password of the APIM server
host	IP of the running APIM server. In a distributed setup, give the host of the API Publisher node.
port	Port of the running APIM server. In a distributed setup, give the port of the APIM Publisher node.
version	Version of the APIM server

10. Using the command line, go to `<APIM_1.7.0_HOME>/doc-file-migration` folder and execute **`ant run`**. If the above configuration is successful, you get a **`BUILD SUCCESSFUL`** message.

Upgrading tenants

11. If you have **multiple tenants** added to your API Manager instance, follow the steps below to migrate tenant configurations:
    a. Copy the contents from your previous `<APIM_HOME>/repository/tenants` directory to the same directory in the API Manager 1.7.0.
    b. Execute steps 5 and 6 for all tenants in your system.
    c. Execute steps 8 to 10 for all tenants in your system.

Upgrading external stores

12. If you have **external stores** configured under the `<ExternalAPIStores>` element in `<APIM_1.6.0_HOMe>/repository/conf/api-manager.xml` file, follow the steps below:
    a. Log in to APIM 1.7.0 management console and click the **Resources -> Browse** menu.
    b. Load `/_system/governance/apimgt/externalstores/external-api-stores.xml` resource in the registry browser UI, configure your external stores there and save.

Upgrading Google analytics

13. If you have **Google Analytics** configured under `<GoogleAnalyticsTracking>` element in `<APIM_1.6.0_HOME>/repository/conf/api-manager.xml` file, follow the steps below:
    a. Log in to APIM 1.7.0 management console and go to **Resources -> Browse** menu.
    b. Load `/_system/governance/apimgt/statistics/ga-config.xml` resource in the registry browser UI, configure the Google analytics and save.

Upgrading workflows

14. If you have **Workflows** configured under `<WorkFlowExtensions>` element in `<APIM_1.6.0_HOME>/repository/conf/api-manager.xml` file, follow the steps below:
    a. Log in to APIM 1.7.0 management console and go to **Resources -> Browse** menu.
    b. Load `/_system/governance/apimgt/applicationdata/workflow-extensions.xml` resource in the registry browser UI, configure your workflows and save.

User Guide

The user guide provides information about the features, functionality, solution development, testing and debugging options of WSO2 API Manager.

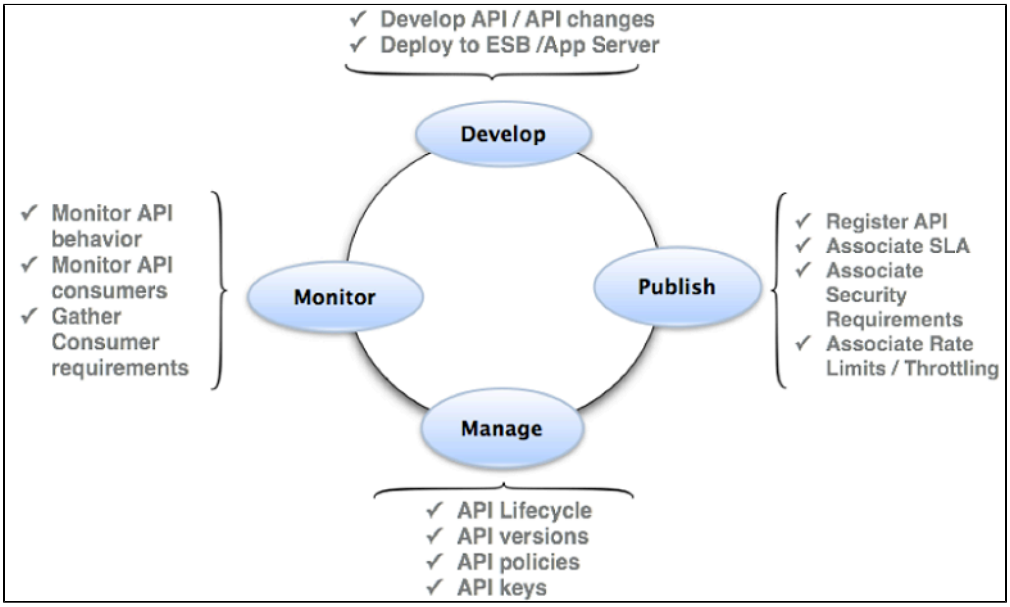
- API Developer Guide
- Application Developer Guide
- Customizing the API Store
- Monitoring, Statistics and Billing
- Extending API Manager
- Working with Security

API Developer Guide

API development is usually done by someone who understands the technical aspects of the API, interfaces, documentation, versions etc., while API management is typically carried out by someone who understands the business aspects of the APIs. In most business environments, API development is a responsibility that is distinct from API publication and management.

WSO2 API Manager provides a simplified Web interface called **WSO2 API Publisher** for API development, publication and management. It is a structured GUI designed for API creators to develop, document, scale and version APIs, while also facilitating more API management-related tasks such as publishing API, monetization, analyzing statistics, quality and usage and promoting and encouraging potential consumers and partners to adopt the API in their solutions.

Shown in the diagram below are common life cycle activities of an API developer/manager, supported by the WSO2 API Publisher:



To access the API development-related functionality provided by the WSO2 API Publisher, you need to create user roles with specific levels of permission. In this documentation, we use a role by the name creator to carry out more development-related tasks, and a role by the name publisher to carry out more management-related tasks. For instructions on adding the creator/publisher roles and assign them to users, refer to section User Roles in API Manager. Before accessing the Web interface of the API Publisher, make sure you run the API Manager using instructions given in section Running the Product. Once the server is up, type the following URL in your browser to access the API Published Web interface.

```
https://<YourHostName>:9443/publisher
```

```
You cannot access the API Publisher Web interface using HTTP. It is exposed as HTTPS only.
```

The API Publisher log-in page opens as follows:

Once you are successfully logged in to the API Publisher, refer to the following information to start developing and managing APIs.

- Creating and Managing APIs
- Editing and Deleting APIs
- Managing Throttling Tiers
- Documenting APIs
- Versioning APIs
- Publishing to API Stores
- Managing API Usage

Creating and Managing APIs

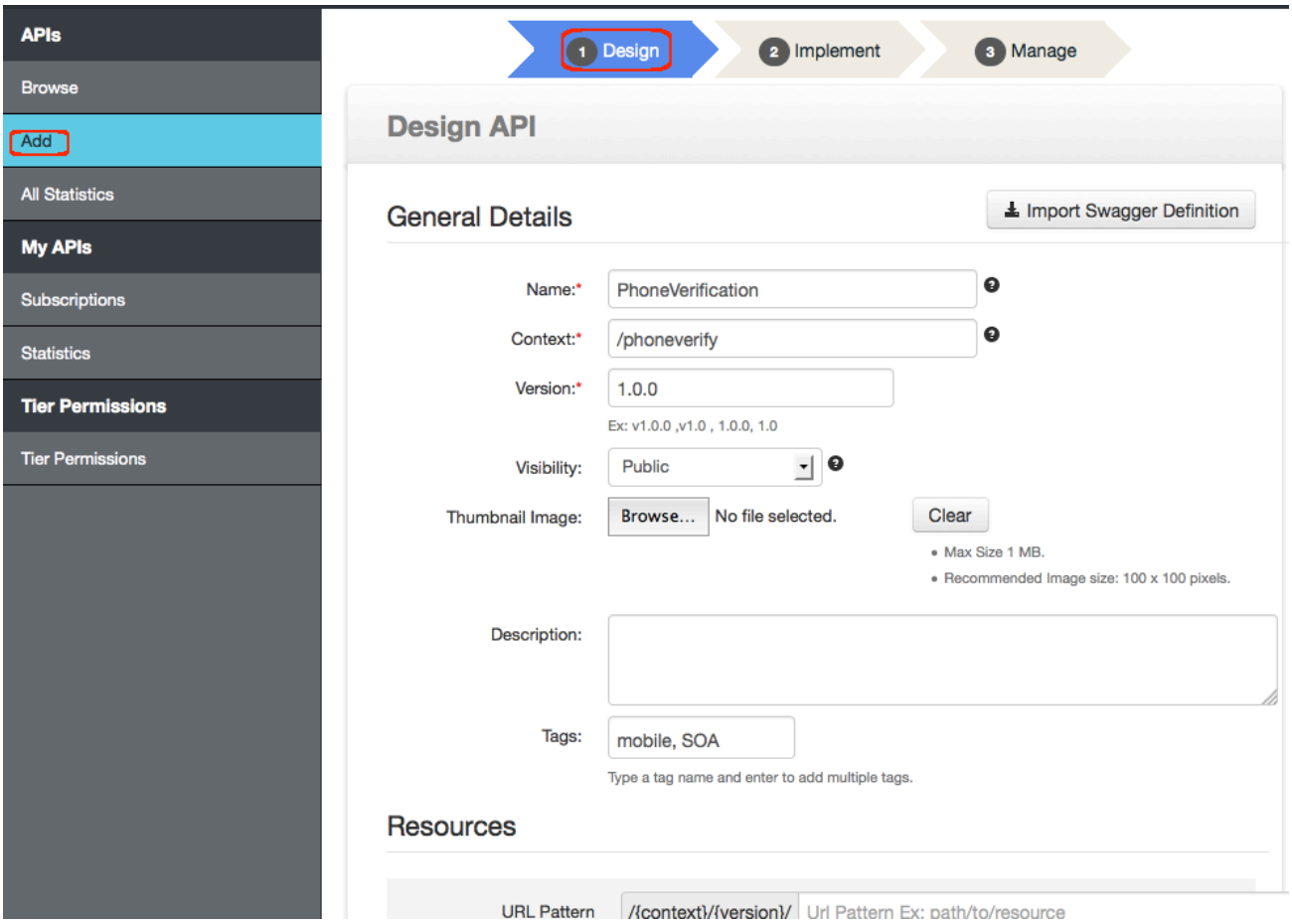
The following sections walk you through creating, documenting and managing an API:

- Designing APIs
- Implementing APIs
- Managing APIs

Designing APIs

Follow the steps below to start designing an API:

1. Log in to the API Publisher (http://localhost:9763/publisher) as a user who is assigned the creator role.
2. Click **Add** to open **Design API** window as follows:



The sections below explain the fields of the above window.

General details

Field	Description
Name*	Name of API as you want it to appear in the API store (E.g., PhoneVerification)
Context*	URI context path that is used by API consumers. (E.g., /phoneverify)
Version*	API version in the form of version.major.minor. (E.g., 1.0.0)
Visibility	See API visibility
Tags	Any number of tags separated by comma. Tags allow you to group/categorize APIs that have similar attributes and behaviors. When tagging, always use relevant keywords and common search terms. Once a tagged API gets published to the API Store, its tags appear on the dashboard as links to the API consumers, who can click on them to quickly jump to a category they are interested in.
Resources	See API Resources. (E.g., phoneID)

API visibility

Visibility settings prevent certain user roles from viewing and modifying APIs created by another user role. The visibility values mean the following:

- **Public** : The API is visible to all users (subscribers and anonymous users) of its tenant store. Also, the API can be advertised in multiple stores - a central store and/or non-WSO2 stores.
- **Visible to my domain** : The API is visible to all users who are registered in the API's tenant domain.

- **Restricted by Roles** : The API is visible only to specific user roles in the tenant store. When `Restricted by Roles` is selected, a new field called **Visible to Roles** appears where you can specify the user roles that have access to the API in a comma-separated list (no spaces).

> - Roles that have API creation and publication permission can see all APIs in their tenant store even if you restrict access to them. This is because any role that has API creation and publication permission can view and edit all APIs in the API Publisher. Therefore, there is no reason to hide the APIs from them in the Store.
>
> - If you restrict the default `subscriber` role under the `Visible to Roles` category, any user who self subscribes to the API Store will be able to access the API. This is because the API Manager assigns the subscriber role to all users who sign up to the API Store.

Given below is how visibility levels work for users in different tenant modes:

Visibility in super tenant mode

Subscribers in super tenant mode can see an API depending on its visibility level as follows:

- Anonymous users : can see APIs with `Public` visibility
- Signed-up users : can see all APIs with `Public` visibility as well as APIs that are `Restricted by Roles`, give that the user is assigned to the role the API is restricted by.

Visibility in multi-tenant mode

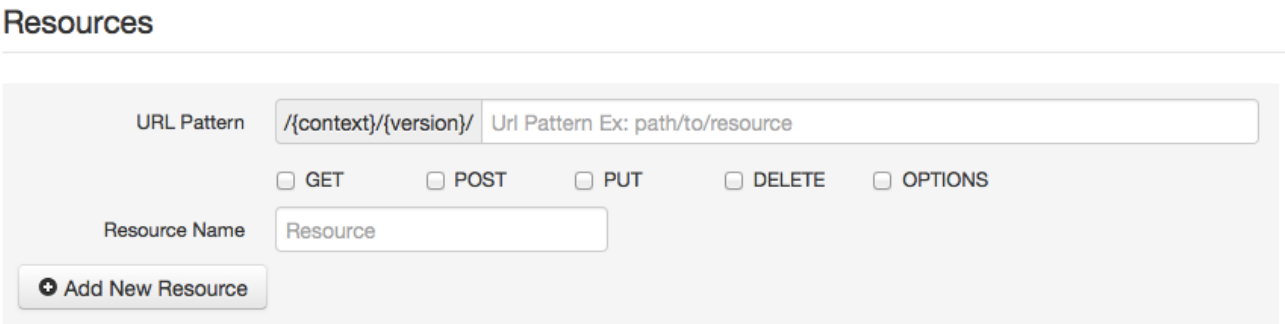
In multi tenant environment, a subscriber can see API Store URLs of existing tenants. Click a URL to browse the tenant's API Store.

A tenant's API Store is the API Store specific to the tenant domain the user belongs to. You can also access it with the URL `http://<hostname>/Store?tenant=<tenantdomain.com>`. Therefore, the APIs a subscriber sees in multi tenant mode depend on their visibility levels as well as which API Store s/he is looking at. Any subscriber viewing his/her tenant's API Store can see an API depending on its visibility level as follows:

- Anonymous users: can see APIs that have `Public` visibility and created within the current user's tenant domain
- Logged in users: can see,
  - APIs that have `Public` visibility and created within the current users tenant domain
  - `Restricted by Roles` APIs created within the current user's tenant domain and are allowed to be accessed by the role of the current user

API resources

An API is made up of one or more resources. Each resource handles a particular type of request and is analogous to a method (function) in a larger API.



API resources can accept the following attributes:

Attribute name	Description

URL Pattern	A URL pattern can be one of the following types:
	• As a url-mapping. E.g., `/state/town/*`
	• As a uri-template. E.g., `/{state}/{town}`
	
	The terms url-mapping and uri-template come from synapse configuration language. When an API is published in the API Publisher, a corresponding XML definition is created in the API Gateway. This XML file has a dedicated section for defining resources. See examples below:
	
	`<resource methods="POST GET" url-mapping="/state/town/*">`
	`<resource methods="POST GET" uri-template="/{state}/{town}">`
	
	url-mapping performs a one-to-one mapping with the request URL, whereas the uri-template performs a pattern matching.
	
	Parametrizing the URL allows the API Manager to map the incoming requests to the defined resource templates based on the message content and request URI. Once a uri-template is matched, the parameters in the template are populated appropriately. As per the above example, a request made to http://gatewa_host:gateway_port/api/v1/texas/houston sets the value of `state` to `texas` and the value of `town` to `houston`. You can use these parameters within the synapse configuration for various purposes and gain access to these property values through the `uri.var.province` and `uri.var.district` properties. For more information on how to use these properties, see Introduction to REST API and the HTTP Endpoint of the WSO2 ESB documentation.
	
	Also see http://tools.ietf.org/html/rfc6570 on URI templates.

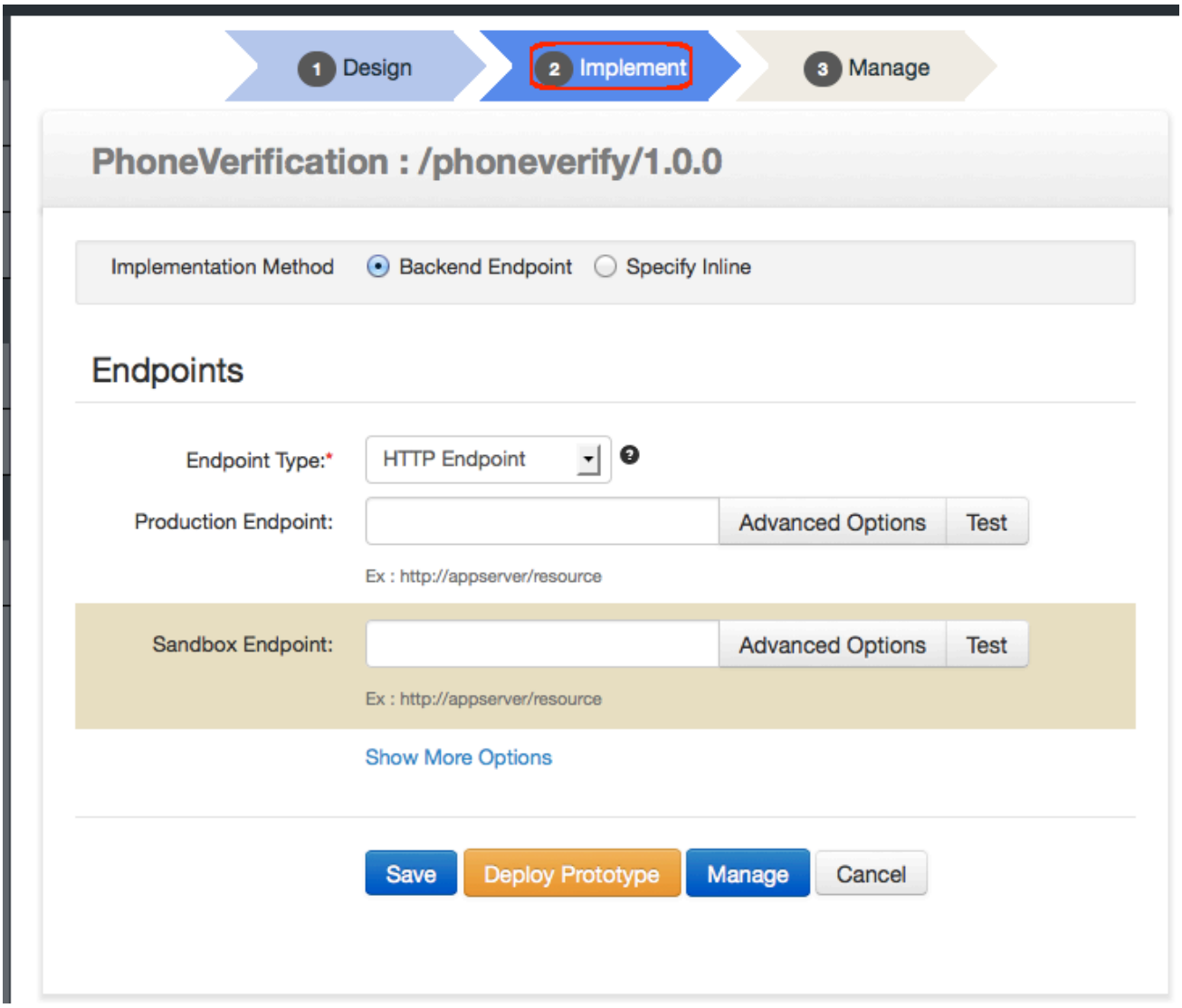
HTTP Verb	The HTTP methods that specify the desired action to be performed on the resource. These methods can be GET, POST, PUT, DELETE or OPTIONS. Multiple methods can be selected.

Once a request is accepted by a resource, it will be mediated through an in-sequence. Any response from the back-end is handled through the out-sequence. Fault sequences are used to mediate errors that might occur in either sequence. The default in-sequence, out-sequence and fault sequences are generated when the API is published.

3. After providing all design details, click **Implement** at the bottom of the above UI to start implementing the API.

Implementing APIs

You implement APIs using the following UI in the API Manager. To get to this UI, follow the steps in designing APIs.



You can configure an actual backend or specify the implementation inline. You can also deploy this API as a prototype.

- Backend endpoints
- Specify Inline
- Deploy as a prototype

Backend endpoints

An endpoint defines the external destination for an outgoing message.

Field	Description
Endpoint Type	WSO2 API Manager has support for a range of different endpoint types allowing the API Ga types of backends. The API Manager supports HTTP endpoints, URL endpoints (also term endpoints, Failover endpoints, Load-balanced endpoints. Also see Adding an Endpoint section in the ESB docs for details of the advanced configuratic

Production/Sandbox URLs	Endpoint of the back-end service URL and endpoint of sandbox (testing) back-end service. A online testing of an API with easy access to an API key. Also see Maintaining Separate Production and Sandbox Gateways. The system reads gateway endpoints from `api-manager.xml` file. When there are r defined, it picks the gateway endpoint of the production environment. You can de gateway endpoints as follows: `<GatewayEndpoint>http://${carbon.local.ip}:${http.nio.port},http}:${https.nio.port}</GatewayEndpoint>` If both types of endpoints are defined, the HTTPS endpoint will be picked as the serve You cannot call back-end services secured with OAuth through APIs created in the you can call only services secured with username/password. The API Manager allows you to expose both REST and SOAP services to consumers
Endpoint Security Scheme	Secured endpoint or Non secured endpoint. Default is non secured endpoint. If secured endpoint is selected, user is asked for credentials of the backend service. If you get a **`Hostname verfiication failed exception`** when trying to send re set `<parameter name="HostnameVerifier">` to `AllowAll` in `<APIM_HOME>/r` `xis2.xml` file's HTTPS transport sender configuration. For `name="HostnameVerifier">AllowAll</parameter>`. This parameter verifies the hostname of the certificate of a server when API Mana outbound service calls.
WSDL	URL of WSDL file describing API interface. (E.g., http://ws.cdyne.com/phoneverify/phoneveri When you provide the WSDL URL, the WSDL content will be saved as a resource file `e/apimgt/applicationdata/ wsdls` folder in the registry. API artifacts have resource. Its original service address location is reset to the API Gateway's address service endpoint directly. At the store, we will show the registry permlink of the wsdl the WSDL and create a service project out of that.
WADL	URL to WADL file (describing API interface).
Destination-based Usage Tracking	Enable this feature to generate a graph showing the number of times an API accesses its de is generated in the API Manager Statistics dashboard. It gives API Publishers an insight al Gateway to destination endpoints, especially useful in cases where the same API can Load-balanced endpoints).

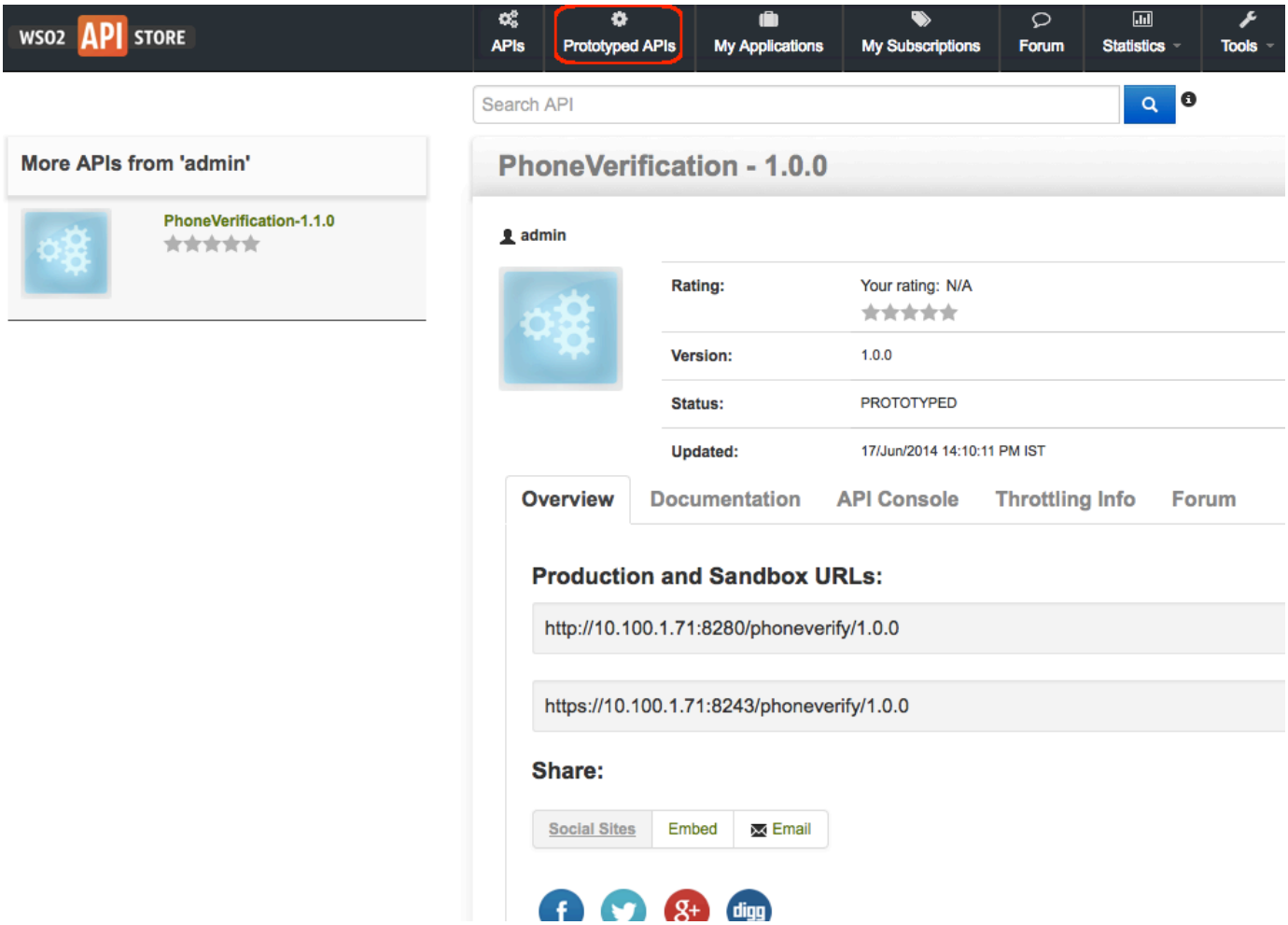
Specify Inline

You can specify the API implementation inline, without connecting to a backend where the API is implemented. Click the **Specify Inline** check box and you will find the resource created in the design section. For each HTTP method, you can write your own implementation in the **Script** section. For example,

Deploy as a prototype

If you click the **Deploy Prototype** button, the API will be deployed as a sample or a model API. The purpose of a prototyped API is to give the API users an early implementation of the API so that they can use it without subscribing, comment on its effectiveness and request improvements. You then change the API's implementation according to user comments and publish it. A published API is available for subscription and monetization.

Go to the API Store (https://localhost:9443/store/) and click the **Prototyped APIs** menu to see your API deployed there. Then, open the API. For example:

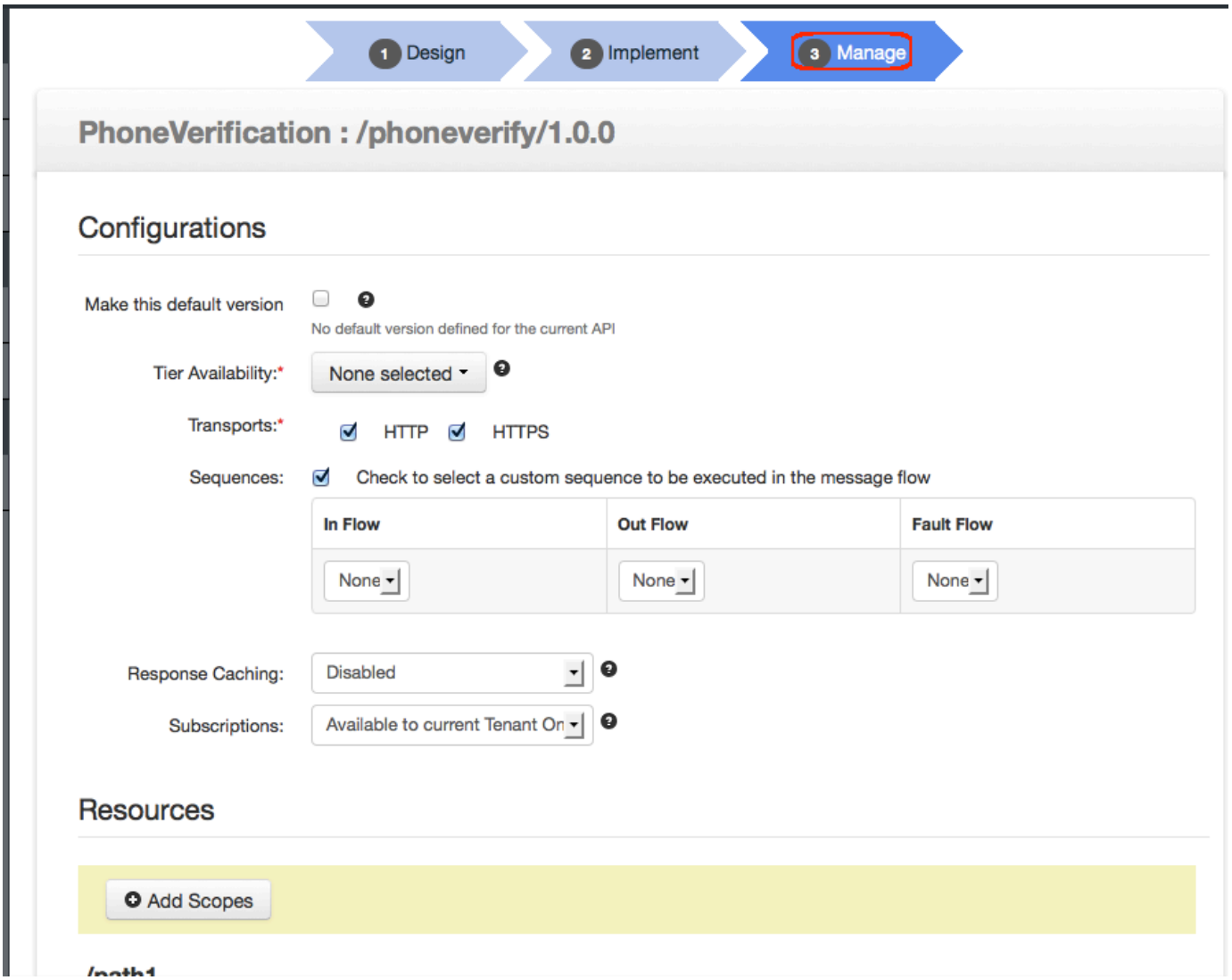


Note that the subscription options are not available for the API. But, users can test the API using the API Console tab, read documentation, engage in forums and other community features and share comments about the API.

Next, start managing the API.

Managing APIs

You manage APIs using the following UI in the API Manager. To get to this UI, follow the instructions in implementing APIs.



The tables below explain the fields of the above UI.

Field	Description
Make this the default version	All API contexts are suffixed with an API version. The default version option allows you to mark one API, from a group of API versions, as the default one, so that it can be invoked without specifying the version number in the URL. For example, say that the following API versions exist: • http://host:port/youtube/1.0 • http://host:port/youtube/2.0 • http://host:port/youtube/3.0 If you mark the third API as the default API, requests made to http://host:port/youtube/ get automatically routed to http://host:port/youtube/3.0. You can make any of the API versions as the default version at any time. However, if you mark an unpublished API as the default while the previously default API was a published one, then the users who invoke the default API will still be routed to the previous default version rather than the new one. This is because the new default API version is not published yet.
Tier Availability	See API-level throttling.

Transports	The transport protocol on which the API is exposed. Both HTTP and HTTPS transports are selected by default. If you want to limit API availability to only one transport (e.g., HTTPS), un-check the other transport.
Sequences	Custom sequences that you want to invoke in the message flow. For details, see per-API sequences.

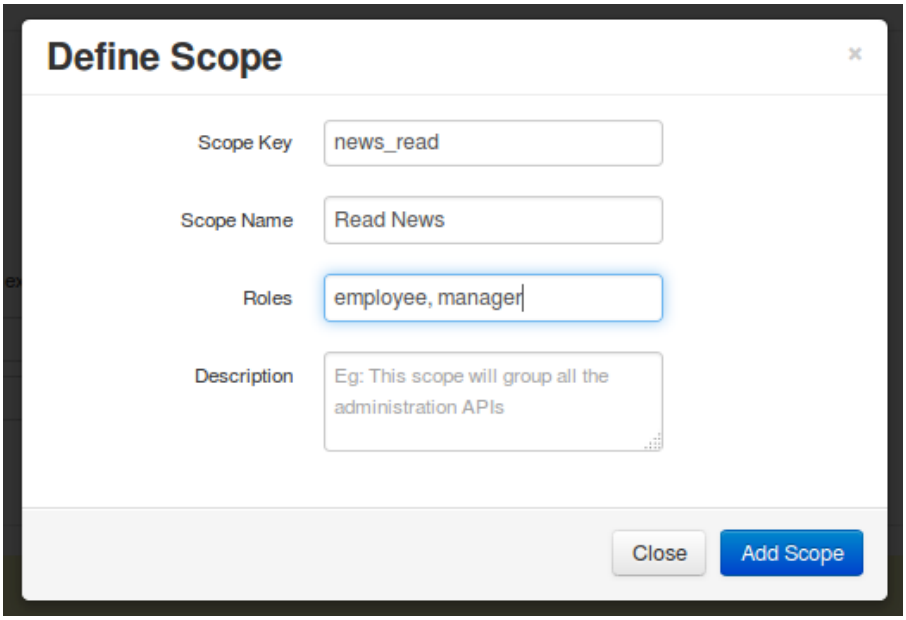
Response Caching	Used to enable caching of response messages per each API. Caching protects the backend systems from being exhausted due to serving the same response (for same request) multiple times. If you select the **enable** option, specify the cache timeout value (in seconds) within which the system tries to retrieve responses from the cache without going to the backend.
	
	To configure response caching, edit `<APIM_HOME>/repository/resources/api_templates/velocity_template.xml` file. The cache mediator properties in the XML file are as follows:
	
	• collector
	  • true: Specifies that the mediator instance is a response collection instance
	  • false: Specifies that it's a cache serving instance.
	• max MessageSize: Specifies the maximum size of a message to be cached in bytes. An optional attribute, with the default value as `unlimited`.
	
	• maxSize: Defines the maximum number of elements to be cached.

Subscriptions	Used to specify the tenants who can subscribe to an API, in a multi-tenanted API Manager deployment. The following types of subscription categories are available between tenants: • Available to current Tenant Only: Only users who are in the current tenant domain, i.e., the tenant domain of the API creator, can subscribe to this API. • Available to All the Tenants: Users of all tenant domains in the API Manager deployment can subscribe to this API. • Available to Specific Tenants: Users of specified tenant domains as well as the current tenant domain (i.e., the tenant domain of the API creator) can subscribe to this API.
Resource settings	**Scope**: See OAuth scopes **Auth type**: You can give the following levels of authentication to each HTTP method of the resource: • **None**: The API Gateway skips the authentication process • **Application**: Authentication is done by the application • **Application User**: Authentication is done by the application user • **Application and Application User**: both application and application user level authentication is applied. Note that if you select this option in the UI, it appears as **Any** in the API Manager's internal data storage and data representation and **Any** will appear in the response messages as well. The auth type is cached in the API Manager for better performance. If you change the auth type through the UI, it takes about 15 minutes to refresh the cache. During that time, the server returns the old auth type from the cache. If you want the changes to be reflected immediately, please restart the server after changing the auth type. **Tier**: See Resource-level throttling

OAuth scopes

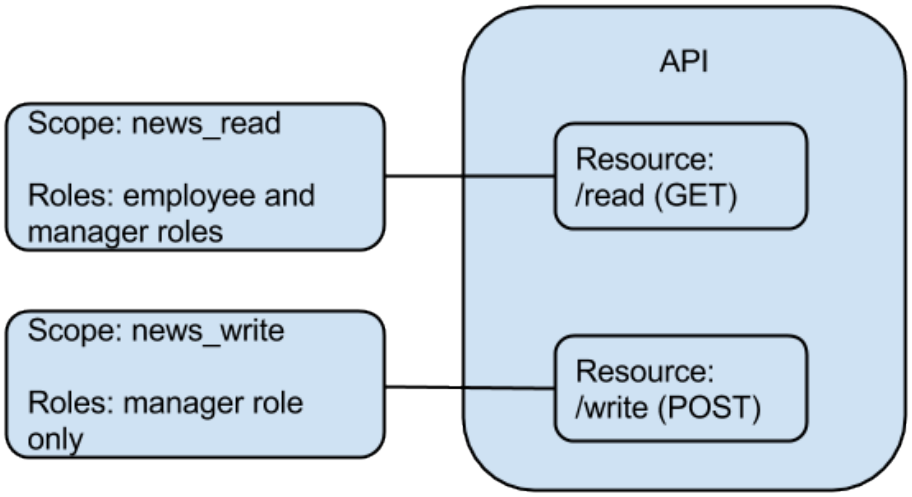
Scopes enable fine-grained access control to API resources based on user roles. You define scopes to an API's resources. When a user invokes the API, his/her OAuth 2 bearer token cannot grant access to any API resource beyond its associated scopes.

You can apply scopes to an API resource at the time the API is created or modified. In the API Publisher, click the **API -> Add** menu (to add a new API) or the **Edit** link next to an existing API. Then, navigate to the **Manage** tab and scroll down to see the **Add Scopes** button. A screen such as the following appears:



Scope Key	A unique key for identifying the scope. Typically, it is prefixed by part of the API's name for uniqueness, but is not necessarily reader-friendly.
Scope Name	A human-readable name for the scope. It typically says what the scope does.
Roles	The user role(s) that are allowed to obtain a token against this scope. E.g., manager, employee.

To illustrate the functionality of scopes, assume you have the following scopes attached to resources of an API:



Assume that users named **Tom** and **John** are assigned the employee role and both the employee and manager roles respectively.

Tom requests a token through the Token API as `grant_type=password&username=nuwan&password=xxxx&scope=`**`news_read news_write`**. However, as Tom is not in the manager role, he will only be granted a token bearing the news_read scope. The response from the Token API will be similar to the following:

```
"scope":"news_read","token_type":"bearer","expires_in":3299,
"refresh_token":"8579facb65d1d3eba74a395a2e78dd6",
"access_token":"eb51eff0b4d85cda1eb1d312c5b6a3b8"
```

Next, John requests a token as `grant_type=password&username=john&password=john123&scope=`**`news_read news_write`**. As john has both roles assigned, the token will bear both the requested scopes and the response will be similar to the following:

```
"scope":"news_read news_write", "token_type":"bearer", "expires_in":3299,
"refresh_token":"4ca244fb321bd555bd3d555df39315",
"access_token":"42a377a0101877d1d9e29c5f30857e"
```

This means that Tom can only access the GET operation of the API while John can access both as he is assigned to both the employee and manager roles. If Tom tries to access the POST operation, there will be an HTTP 403 Forbidden error as follows:

```
<ams:fault xmlns:ams="http://wso2.org/apimanager/security">
   <ams:code>900910</ams:code>
      <ams:message>The access token does not allow you to access the requested
resource</ams:message>
      <ams:description>Access failure for API: /orgnews, version: 1.0.0 with key:
eb51eff0b4d85cda1eb1d312c5b6a3b8
   </ams:description>
</ams:fault>
```

Tip: To invoke an API protected by scopes, you need to get an access token via the Token API. Tokens generated from the **My Subscriptions** page in the API Store will not work.

Click **Save & Publish** or click **Save** to publish the API later. For information on publishing APIs, see Publishing to API Stores.

Editing and Deleting APIs

The steps below explain how to modify an API's source code and delete an API.

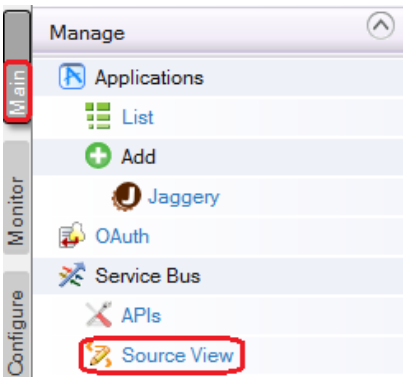
Editing an API

You create an API using the API Publisher Web interface. To edit an API, you select the API in the API Publisher and then click the **Edit** link next to its name. Similarly, most common configurations of the APIs are facilitated through the Web UI.

The **Edit** link is only visible to users with creator privileges. See Managing Users and Roles.

However, if you want to do more advanced configurations to this API, you have to go into its code-level configurations. You can do this using the steps given below.

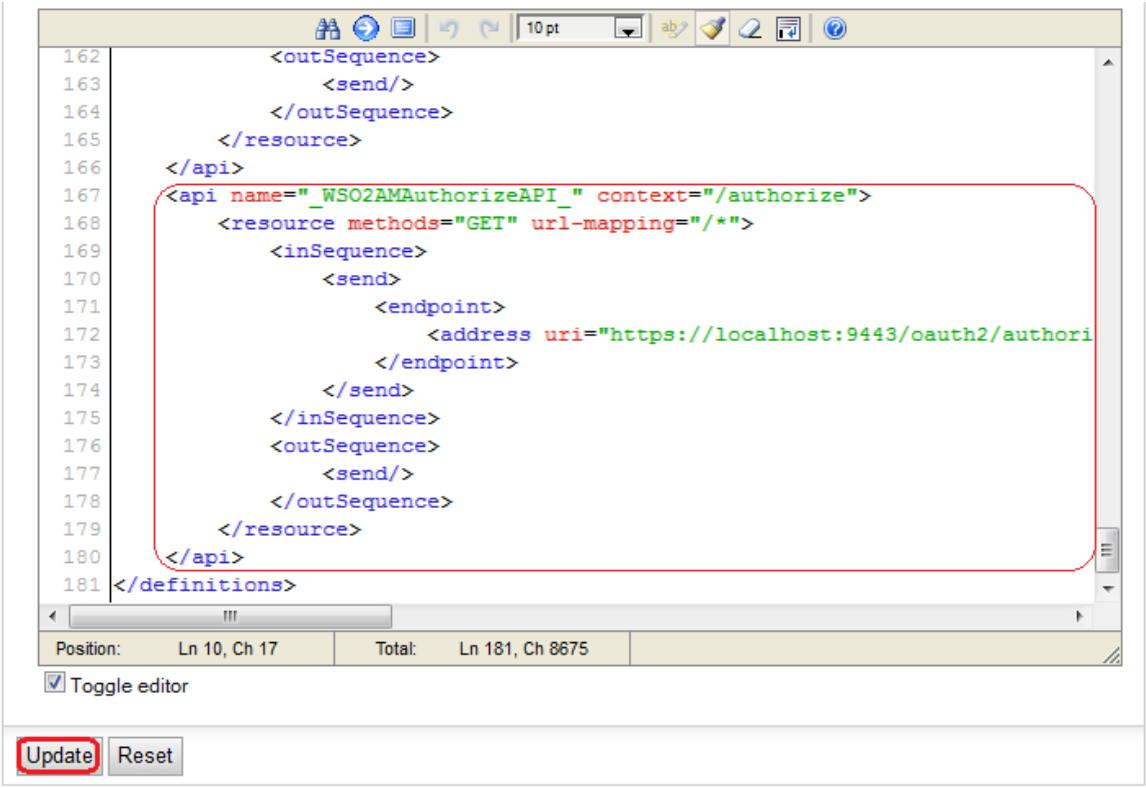
1. Log in to the Management Console UI (`https://localhost:9443/carbon`) using `admin/admin` credentials. Then, select **Source View** sub menu under the **Service Bus** menu.



2. Source view contains the entire configuration of the API Gateway. You can find sequences, filters, properties, APIs etc. defined there. Search for the name of the API you want, and edit its content wrapped by the `<api>` `</api>` elements.

> You should not remove the default filter mediator and handler configurations in your API. They are needed for routing requests based on the throttling/security policies. If you want to add a custom mediator in the `insequence` path of a request, add that inside the filter mediator configuration as shown in the following example.
>
> ```
> <filter source="$ctx:AM_KEY_TYPE" regex="PRODUCTION">
>  <then>
>  <class name="org.wso2.carbon.custommediator.CustomDataMediator"/>
>  .....................
>  </then>
> </filter>
> ```

3. Click **Update** to save your changes.



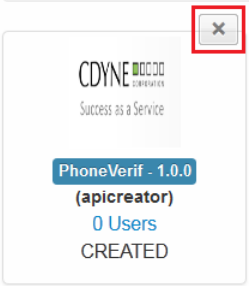
4. Restart the server.

Instead of editing the configuration through the UI, you can directly edit the file saved in `<APIM_HOME>/repository/deployment/server/synapse-configs/default/api` folder as well.

Deleting an API

Follow the instructions below to delete an API from the API Store through the API Publisher Web interface.

1. Log in to WSO2 API Publisher (`http://localhost:9763/publisher`) Web application with credentials of a user who has the creator role assigned. For more information on creating users and Managing Users and Roles.assigning roles, refer to section
2. The **All APIs** window opens.
3. Click the **Delete** icon at the top right of a selected API to remove it, and confirm the deletion.



Once deleted, it will no longer be available in the API Store or the API Publisher.

Managing Throttling Tiers

Throttling allows you to limit the number of hits to an API during a given period of time, typically in cases such as the following:

- To protect your APIs from common types of security attacks such as denial of service (DOS)
- To regulate traffic according to infrastructure availability
- To make an API, application or a resource available to a consumer at different levels of service, usually for monetization purpose

The API Manager comes with three default tiers as Gold, Silver and Bronze. Each tier defines a maximum number of requests per minute.

- Bronze - Allows 1 request for the API per minute
- Silver - Allows 5 requests for the API per minute
- Gold - Allows 20 requests for the API per minute

In addition, there is also a special tier called Unlimited, which allows unlimited access. It can be disabled by editing the `<TierManagement>` node of the api-manager.xml file. You can also add your own tiers to the API Manager using the instructions in section Adding New Throttling Tiers in the Admin Guide.

This section covers the following topics:

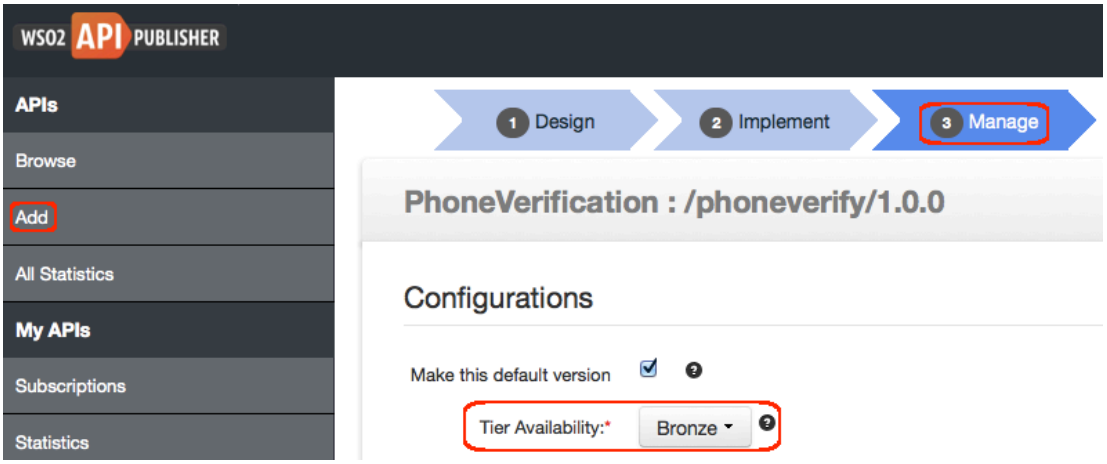
- Different levels of throttling
- How throttling tiers work
- How to write a throttling policy and engage it to APIs

Different levels of throttling

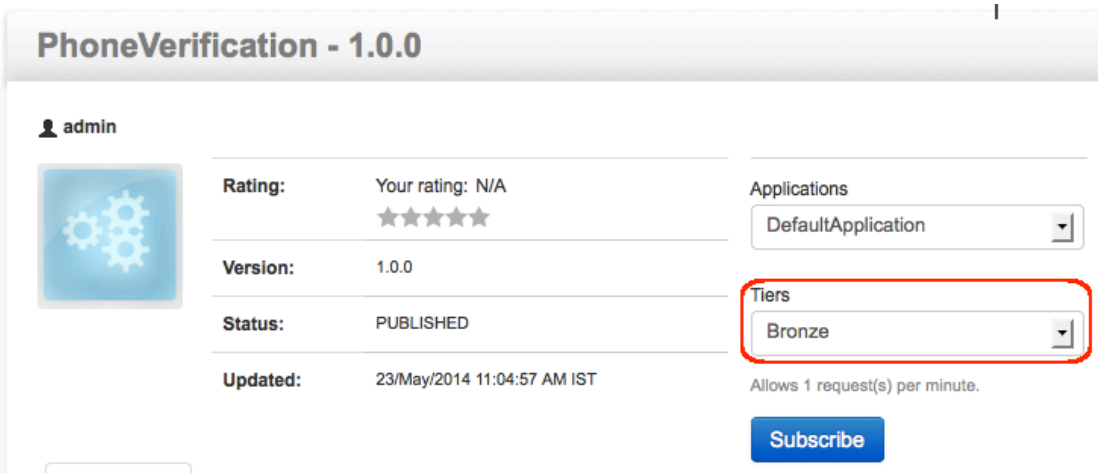
Throttling is enabled in the API Manager in different levels as API-level, application-level, resource-level and IP-level .

API-level throttling

API-level throttling tiers are defined when Managing APIs using the API Publisher portal. The UI looks as follows:



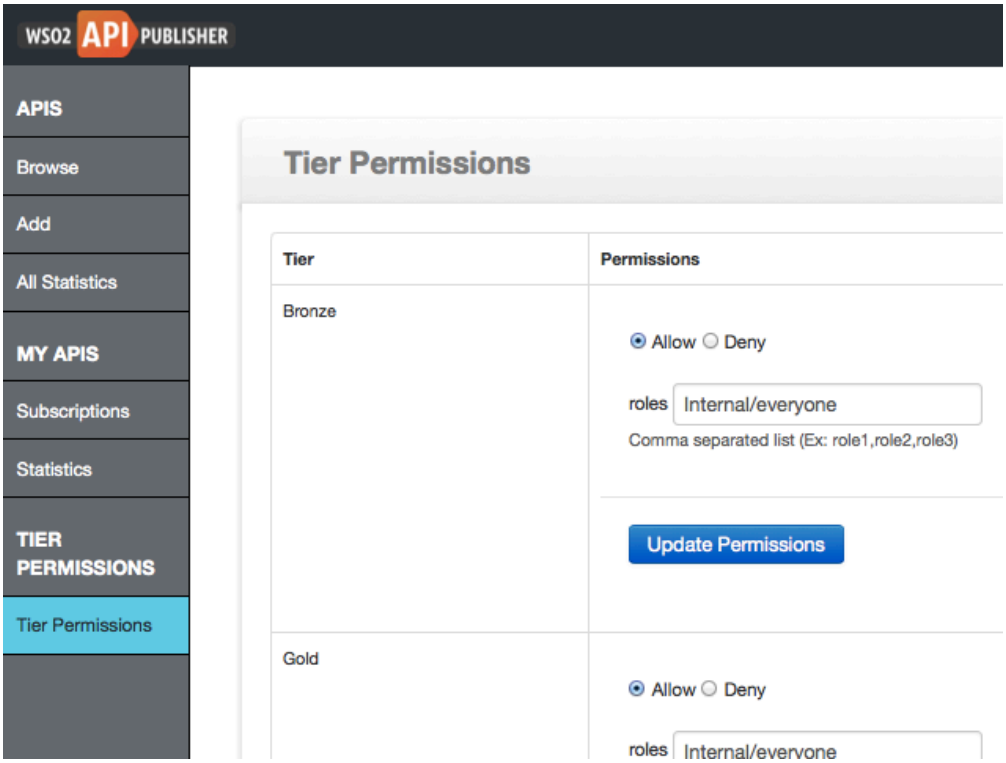
After API-level throttling tiers are set and the API is published, at subscription time, the consumers of the API can log in to the **API Store** and select which tier they are interested in as follows:



According to the tiers s/he selects, the subscriber is granted a maximum number of requests to the API.

Setting tier permissions

Users with `Manage Tiers` permission can set role-based permissions to API-level access throttling tiers. This is done using the **Tier Permissions** menu of API Publisher as shown below. For each tier, you can specify a comma-separated list of roles and either Allow or Deny access to the list.



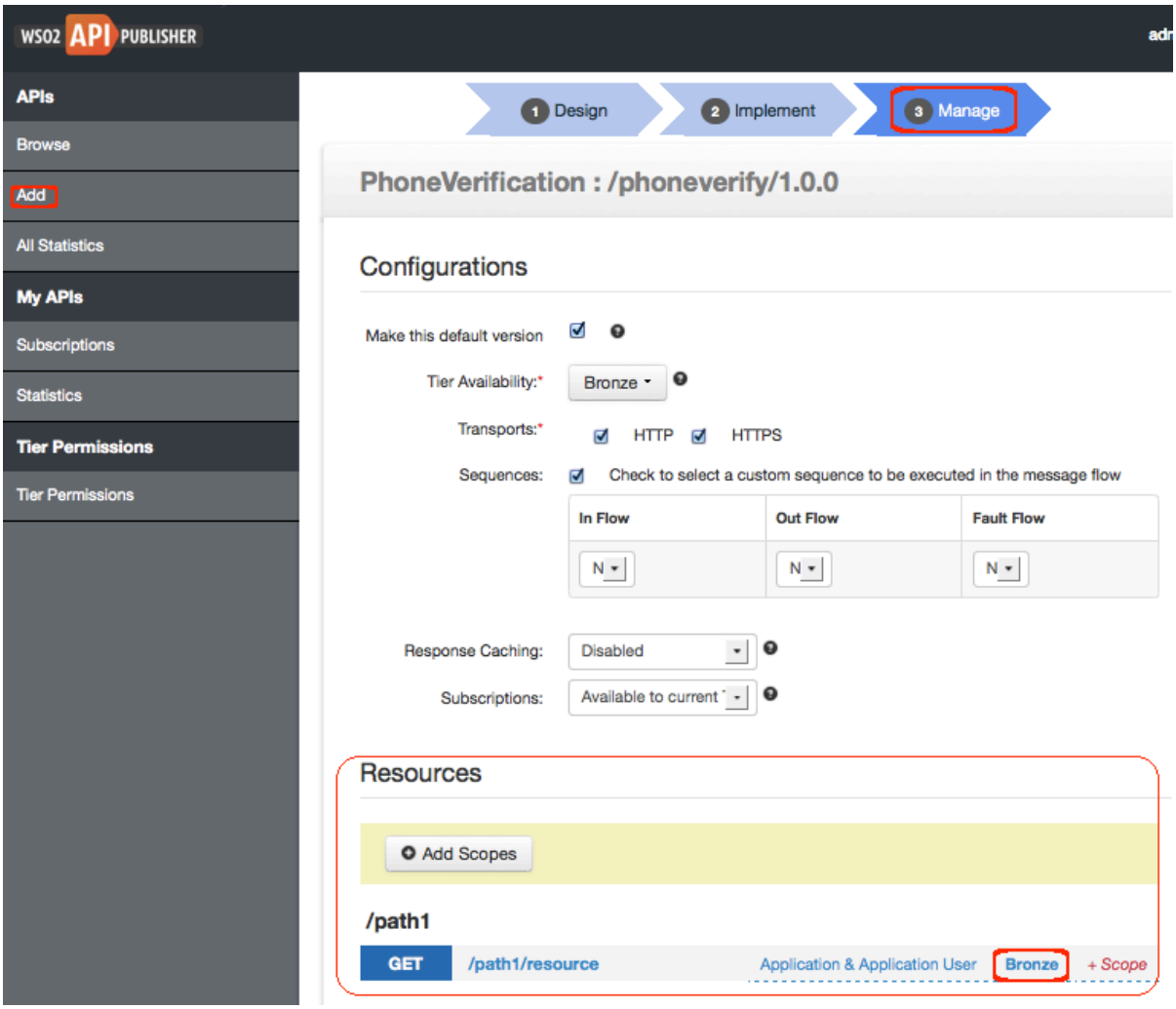
A subscriber logged into the API Store can consume APIs using a specific tier, only if s/he is assigned to a role that is allowed access. In the API Store, the subscriber sees a list of tiers that is filtered based on the subscriber's role. Only the ALLOWED roles appear here. By default, all tiers are allowed to everyone.

Application-level throttling

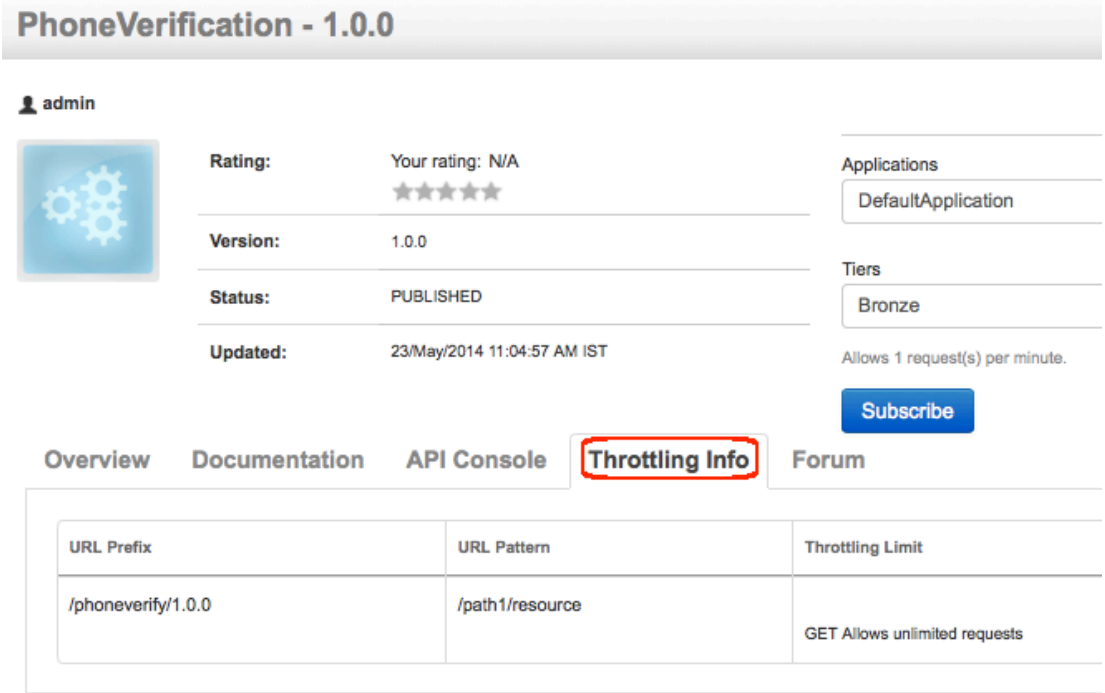
Application-level throttling tiers are defined at the time an application is created using the API Store. For information, see Applications and application-level throttling.

Resource-level throttling

Resource-level throttling tiers are set to HTTP verbs of an API's resources when Managing APIs using the API Publisher portal. The UI looks as follows:



When a subscriber views an API using the **API Store**, s/he can see the resource-level throttling tiers using the **Throttle Info** tab as follows:



Subscribers are not allowed to change these throttling tiers. They are simply notified of the limitations.

IP-level throttling

In IP-level throttling, you can limit the number of requests sent by a client IP (e.g., 10 calls from single client).

1. Log in to the management console and click the **Resources** -> **Browse** menu.
2. Navigate to the `tiers.xml` file in the registry location `/_system/governance/apimgt/applicationdata`.
3. Add your policy. For example, the throttling policy shown below allows only 1 API call per minute for a client from 10.1.1.1 and 2 calls per minute for a client from any other IP address.

```
<wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"

xmlns:throttle="http://www.wso2.org/products/wso2commons/throttle">
<throttle:MediatorThrottleAssertion>
<wsp:Policy>
<throttle:ID throttle:type="IP">10.1.1.1</throttle:ID>
<wsp:Policy>
<throttle:Control>
<wsp:Policy>
<throttle:MaximumCount>1</throttle:MaximumCount>
<throttle:UnitTime>60000</throttle:UnitTime>
</wsp:Policy>
</throttle:Control>
</wsp:Policy>
</wsp:Policy>

<wsp:Policy>
<throttle:ID throttle:type="IP">other</throttle:ID>
<wsp:Policy>
<throttle:Control>
<wsp:Policy>
<throttle:MaximumCount>2</throttle:MaximumCount>
<throttle:UnitTime>60000</throttle:UnitTime>
 </wsp:Policy>
</throttle:Control>
</wsp:Policy>
</wsp:Policy>
</throttle:MediatorThrottleAssertion></wsp:Policy>
```

How throttling tiers work

- When an API is invoked, it first checks whether the request is allowed by API-level throttling limit. If the consumer has exceeded his/her maximum number of allowed API requests, the new request will be terminated.
- If API-level limit is not exceeded, it then checks whether the request is allowed by application-level throttling limit. If it has exceeded, the request will be terminated.
- If application-level limit is not exceeded, it finally checks whether the request is allowed by resource-level throttling limit. If the limit is not exceeded, then the request will be granted.

With capability to define throttling at three levels, the final request limit granted to a given user on a given API is ultimately defined by the consolidated output of all throttling tiers together. For example, lets say two users subscribed to an API using the Gold subscription, which allows 20 requests per minute. They both use the application App1 for this subscription, which again has a throttling tier set to 20 requests per minute. All resource level throttling tiers are unlimited. In this scenario, although both users are eligible for 20 requests per minute access to the API, each ideally has a limit of only 10 requests per minute. This is due to the application-level limitation of 20 requests per minute.

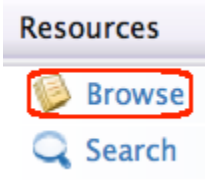
How to write a throttling policy and engage it to APIs

The steps below show how to write a throttling policy and engage it to an API pointing to a backend service.

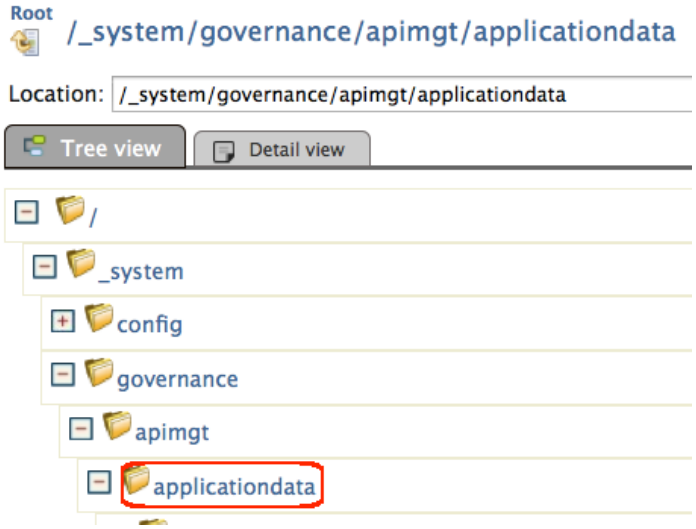
1. The following throttling policy allows 1000 concurrent requests to a service.

```
<wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-uti
lity-1.0.xsd"
xmlns:throttle="http://www.wso2.org/products/wso2commons/throttle"
            wsu:Id="WSO2MediatorThrottlingPolicy">
    <throttle:MediatorThrottleAssertion>
        <throttle:MaximumConcurrentAccess>1000</throttle:MaximumConcurrentAccess>
        <wsp:Policy>
            <throttle:ID throttle:type="IP">other</throttle:ID>
        </wsp:Policy>
    </throttle:MediatorThrottleAssertion>
</wsp:Policy>
```

2. Start the API Manager, log in to its management console (`https://localhost:9443/carbon`) and click the **Resource > Browse** menu to view the registry.



3. Click the `goverence/apimgt/applicationdata` path to go to its detailed view.



4. In the detail view, click the **Resource** link and upload the created policy file to the server as a registry resource.
5. In the management console, select the **Service Bus > Source View** menu.
6. The configurations of all APIs created in the API Manager instance opens. To engage the policy to a selected API, add it to your API definition. In this example, we add it to the login API.

```
<?xml version="1.0" encoding="UTF-8"?><api
xmlns="http://ws.apache.org/ns/synapse"
name="_WSO2AMLoginAPI_" context="/login">
    <resource methods="POST" url-mapping="/*">
        <inSequence>
            <send>
                <endpoint>
                    <address uri="https://localhost:9493/oauth2/token"/>
                </endpoint>
            </send>
        </inSequence>
        <outSequence>
            <send/>
        </outSequence>
    </resource>
    <handlers>
 <handler
class="org.wso2.carbon.apimgt.gateway.handlers.throttling.APIThrottleHandler">
       <property name="id" value="A"/>
       <property name="policyKey"
value="gov:/apimgt/applicationdata/throttle.xml"/>
       </handler>
<handler
class="org.wso2.carbon.apimgt.gateway.handlers.ext.APIManagerExtensionHandler"/>
    </handlers>
</api>
```

> ⚠ Be sure to specify the same path used in step 3 in the policy key of your API definition.

7. You have successfully engaged a throttling policy to an API.

Documenting APIs

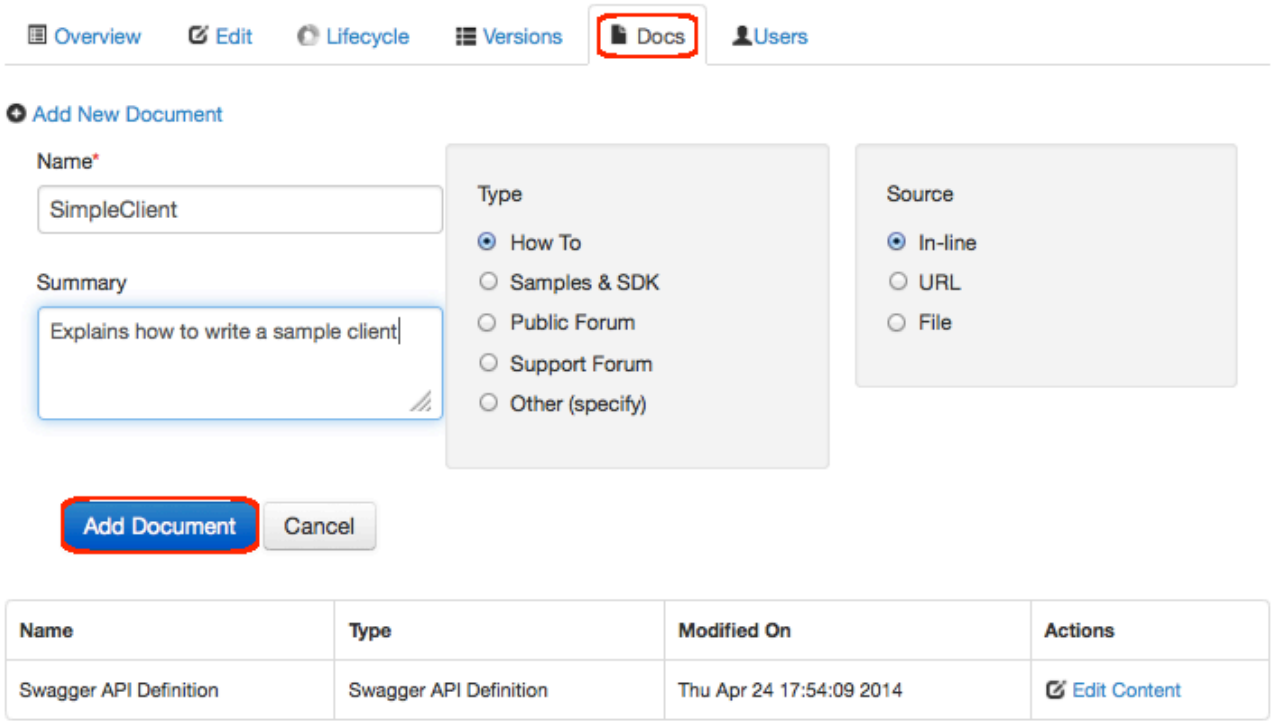
API Manager provides capability to associate comprehensive documentation to an API so that API consumers get a better understanding of its use in implementing their solutions. This section describes the following:

- Adding Documentation Using API Publisher
- Adding Documentation Using Swagger
- Adding Apache Solr-Based Indexing

Adding Documentation Using API Publisher

You can add different types of documents to an API. Proper documentation helps API publishers to market their APIs better and sustain competition. Follow the steps below to add documentation to an API using the API Publisher Web interface.

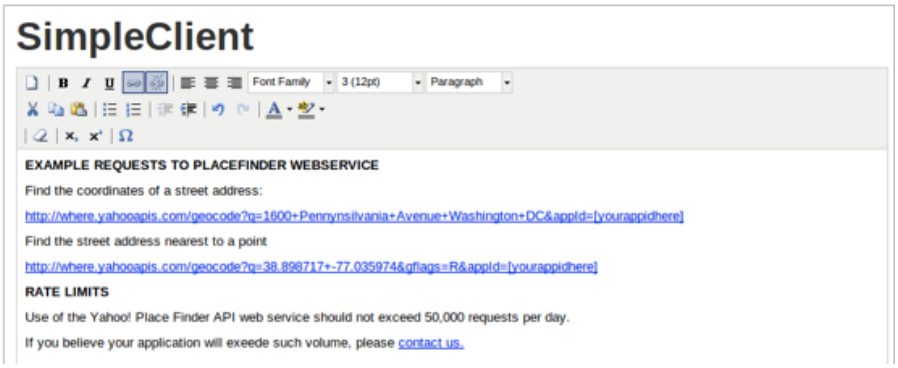
1. Log in to WSO2 API Publisher with a user who has been assigned the creator role. For information on users Managing Users and Roles.and roles, see
2. The currently available APIs appear on the **All APIs** window. Select the API to which you want to add documentation to.
3. Select the **Docs** tab of the API and click the **Add New Document** link.



Documentation can be provided inline, via a URL or as a file.

- **In-line**: Documentation hosted in the API Manager itself. For inline documentation, you can edit the contents directly from the API publisher interface. You get several documents types:
  - Swagger documents
  - How To
  - Samples and SDK
  - Public forum / Support forum (external link only)
  - API message formats
  - Other
- **URL**: If you already have comprehensive documentation managed by an external configuration management system, you can simply link to those file references (URLs) through the API Manager rather than importing them to the server.

4. Click the **Add Document** button to complete.
5. The added document shows on the same window. Click the **Edit Content** link associated with it.
6. The embedded editor opens allowing you to edit the document content.



All documents have unique URLs to help improve SEO support. After editing the API, publish it for it to be available .to external parties through the API Store

> By default, any document associated with an API has the same visibility level of the API. That is, if the API is public, its documentation is also visible to all users (registered and anonymous). To enable other visibility levels to the documentation, go to `<AM_HOME>/repository/conf/api-manager.xml` file, uncomment and set the following element to true:

```
<APIPublisher>
   ....
   <EnableAPIDocVisibilityLevels>true</EnableAPIDocVisibilityLevels>
</APIPublisher>
```

Then, log in to the API Publisher, go to the **Doc** tab and click **Add new Document** to see a new drop-down list added to select visibility from. The settings are as follows:

- **Same as API visibility**: Visible to the same user roles who can see the API. For example, if the API's visibility is public, its documentation is visible to all users.
- **Visible to my domain**: Visible to all registered users in the API's tenant d omain.
- **Private**: Visible only to the users who have permission to log in to the API Publisher Web interface and create and/or publish APIs to the API Store.

Next, see Adding Documentation Using Swagger.

Adding Documentation Using Swagger

Interactive documentation support helps users to understand and experience the APIs better. WSO2 API Manager provides this functionality through the integration of Swagger (https://developers.helloreverb.com/swagger). Swagger is a specification and a complete framework implementation for describing, producing, consuming, and visualizing RESTful Web services. You can load APIs that are described in simple, static JSON representation through the Swagger UI and and make them available as interactive documentation.

The idea of this interactive console is allowing users to test the APIs and get to know how they respond without subscribing to the APIs. When an API is created in API Publisher, the JSON representation of that API is automatically generated and saved into the registry as an API definition. This API definition describes the API using the information provided at the time it is created. You can modify the API definition using the **Doc** tab in the management console. In API Store, the Swagger UI discovers the API definition for each API and displays the interactive documentation in the API's **Documentation** tab.

The sections below explain how to create an interactive documentation for an API:

- Enabling cross-origin resource sharing
- Creating an API
- Updating the API definition
- Invoking the interactive documentation

Enabling cross-origin resource sharing

Swagger-based interactive documentation allows you to try out APIs from the documentation itself. It is a Java Script client that runs in the API Store and makes Java Script calls from the Store to the API Gateway. Since the API Store and Gateway run on two different ports, you must enable cross-origin resource sharing (CORS) between the two using CORS configuration in `<APIM_HOME>/repository/conf/api-manager.xml` file. Given below is a sample configuration of CORS and a description of its XML elements:

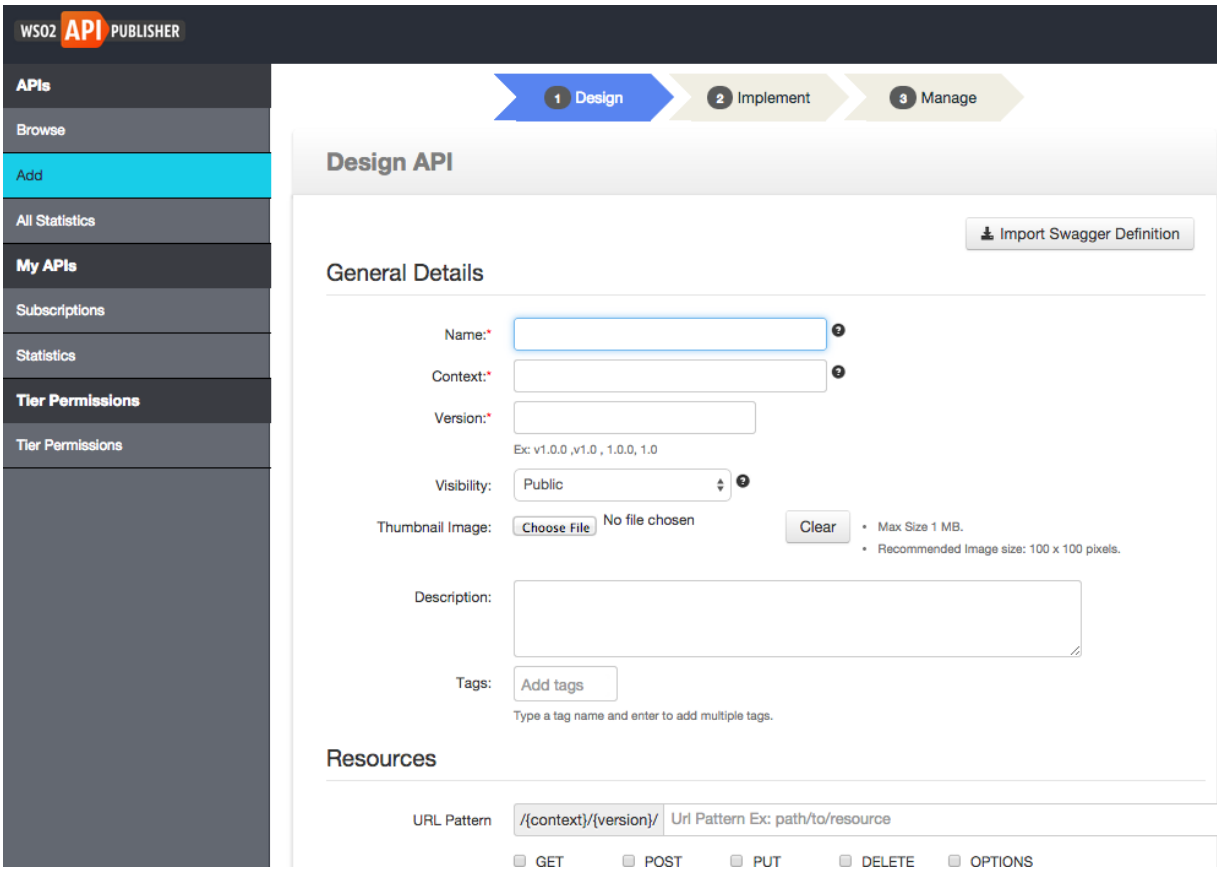
CORS Configuration in api-manager.xml

```
<CORSConfiguration>
    <Enabled>true</Enabled>

<Access-Control-Allow-Origin>https://localhost:9443,http://localhost:9763</Access-Cont
rol-Allow-Origin>

<Access-Control-Allow-Headers>authorization,Access-Control-Allow-Origin,Content-Type</
Access-Control-Allow-Headers>

    <!--Configure Access-Control-Allow-Methods-->

<Access-Control-Allow-Methods>GET,POST,PUT,DELETE,OPTIONS</Access-Control-Allow-Method
s>

    </CORSConfiguration>
```

XML Elements	Values	Description
<Enabled>	True/False	Used to enable/d CORS headers fı default, CORS is is needed for Sw properly.
<Access-Control-Allow-Origin>	HTTP and HTTPS Store Address. Change the Host and Port for correct values of your store. For example, `https://localhost:9443,http://localhost:9763`	The value of the `-Allow-Origir` values are API S are required for s properly.
<Access-Control-Allow-Headers>	Header values you need to pass when invoking the API. For example, `authorization, Access-Control-Allow-Origin, Content-Type`	Default values ar Swagger to funct
<Access-Control-Allow-Methods>	GET, POST, PUT, DELETE, OPTIONS	Methods requirec from the Swagge

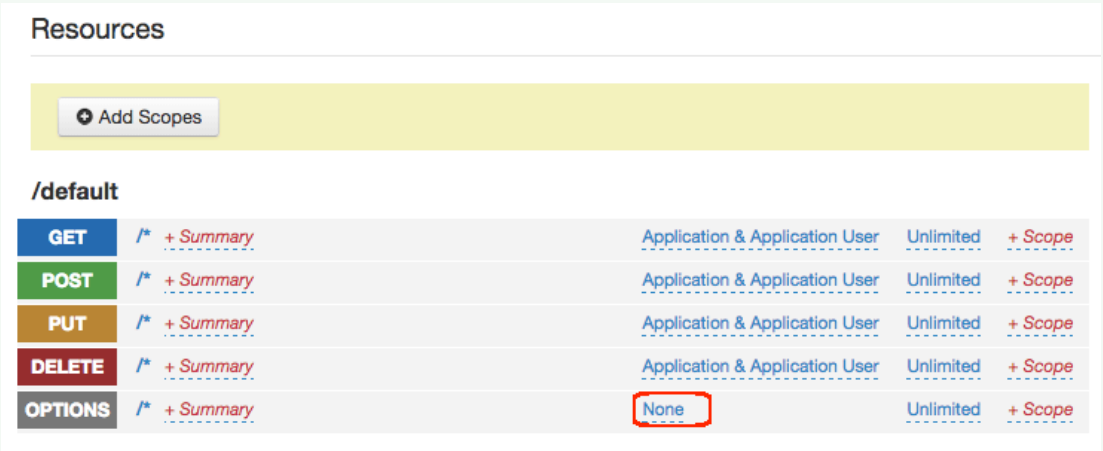
Creating an API

1. Log in to API Publisher Web interface (https://localhost:9443/publisher), and go to **Add API** page. Create a new API with following information by navigating to each tab.
    - Name: PhoneVerification
    - Context: /phoneverify
    - Version: 1.0.0
    - Choose to create a wildcard resource (/*)
    - Endpoint type: HTTP
    - Production Endpoint: http://ws.cdyne.com/phoneverify/phoneverify.asmx
    - Tier availability: Bronze/Gold/Silver/Unlimited
    - Transports: HTTP/HTTPS



In the **Manage** screen, you can specify an authentication type for the methods of the resource that you created earlier.

For each of the resource that has HTTP verbs requiring Authentication (i.e., Auth Type is not NONE), enable OPTIONS with None Auth type. For example, as the following screen shot shows, resources with /* URL Pattern has HTTP verbs with Auth Type as `Application & Application User`. Therefore, we must give None as the Auth Type of OPTIONS. This is to support CORS (Cross Origin Resource Sharing) between the API Store and Gateway. But, if no authentication is needed for any of the HTTP verbs, you don't have to specify None Auth type to OPTIONS.



2. Publish the API to the API Store.

Updating the API definition

The API creator can update/customize the automatically generated API definition for each API.

1. Log in to the API Publisher, go to the **Doc** tab of `PhoneVerify` API and click **Edit Content** under Swagger Documentation.

Name	Type	Modified On	Actions
Swagger API Definition	Swagger API Definition	Mon May 5 11:40:00 2014	Edit Content

2. The API definition opens. Note that the API definition contains its JSON representation.
   - By default, all the POST and PUT operations have a `Payload` parameter, which you can use to send any payload when invoking the API.
   - You can use the `Query parameters` in GET, DELETE operations to send URL-appended values (e.g.,: v=2&length=200).
3. Modify existing content, add/remove elements, change paths and parameters of the API definition using either of the following editors**: Text Editor** or the **Graphical Tree Editor**.

> - API Manager 1.5.0 onwards has integrated JSONMate as the editor for modifying the API Definition.
> - For the Swagger specification of API declaration, see `https://github.com/wordnik/swagger-core/wiki/API-Declaration`.

The example below shows how we have changed the path for all the HTTP methods of the API definition from `/phoneverify/1.0.0/` to `/phoneverify/1.0.0/CheckPhoneNumber` using both the text editor and graphical tree editor:

Swagger API Definition

```
"apiVersion": "1.0.0",
"swaggerVersion": "1.1",
"basePath": "http://192.168.1.2:8280",
"resourcePath": "/phoneverify",
"apis": [
  {
    "path": "/phoneverify/1.0.0/CheckPhoneNumber",
    "description": "",
    "operations": [
      {
        "httpMethod": "GET",
        "summary": "",
        "nickname": "",
        "parameters": [
          {
            "name": "Query Parameters",
            "description": "Request Query Parameters",
```

Format

apiVersion	"1.0.0"
swaggerVersion	"1.1"
basePath	"http://192.168.1.2:8280"
resourcePath	"/phoneverify"
apis	[{"path":"/phoneverify/1.0.0/CheckPhoneNumber","description":"","operations":[{"httpMethod":"GET","

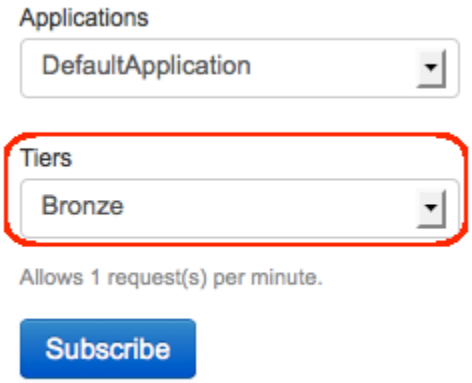
Add New Value

4. After the modifications are done, click **save**.

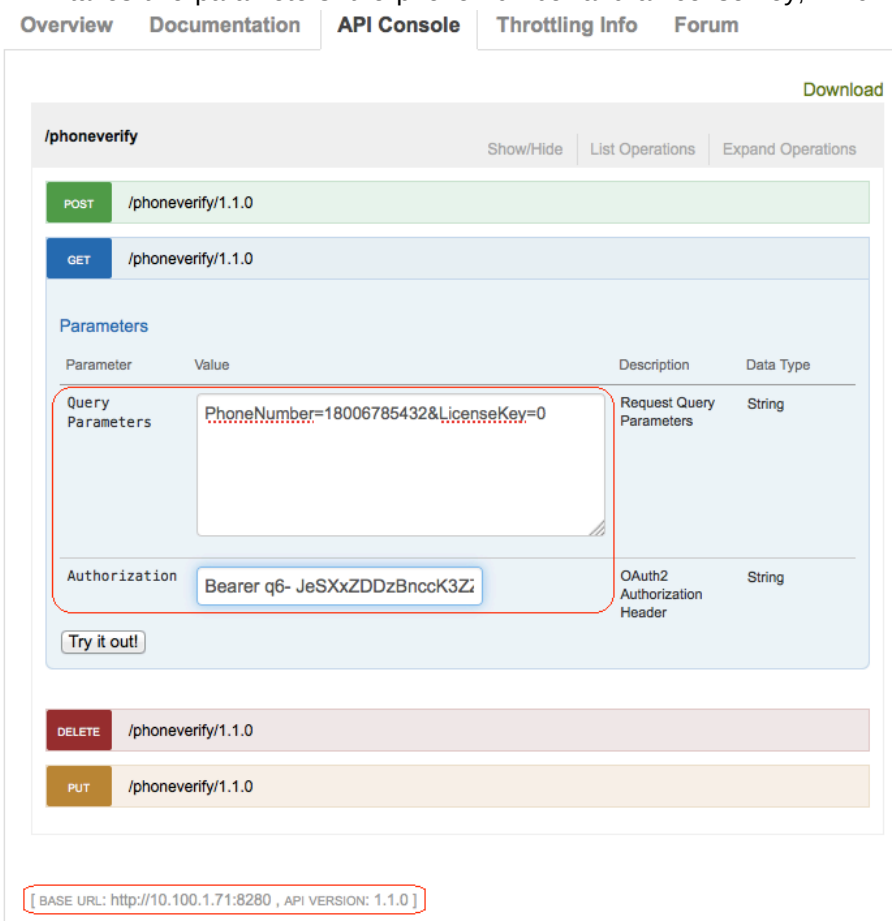
The Swagger JSON files are saved in the following location in the registry: `/_system/governance/apimgt/applicationdata/api-docs/<API name>/api-doc.json`. To browse the registry, log in to the management console (https://localhost:9443/carbon) as admin/admin and select **Resources -> Browse** menu.

Invoking the interactive documentation

1. Log in to the API Store Web interface (https://localhost:9443/store) and click the API published before.
2. Subscribe to the API using the Bronze tier.



3. Generate access tokens. You need them to invoke the API in the next steps.
4. Select the API again and go to the **API Console** tab, which shows the interactive documentation of the API.
5. Provide the necessary parameters and click **Try it out** to call the API. For example, the `PhoneVerification` API takes two parameters: the phone number and a license key, which is set to 0 for testing purposes.



Note the following in the above UI:

Base URL	Appears at the bottom of the console. Using the base URL and the parameters, the system creates the API URL in the form `http://host:8280/<context>/<version>/<back end service requirements included as parameters>`. For example, `http://host:8280/phoneverify/1.1.0/CheckPhoneNumber`.
Query Parameters	Give the API payload as **PhoneNumber=18006785432&LicenseKey=0** where /phoneverify is the context and 1.1.0 is the version. The rest of the URL is driven by the backend service requirements.
Authorization	In the authorization header, pass the application key that was generated at the time a user subscribes to an API. This is prefixed by the string "Bearer". For example, Bearer q6-JeSXxZDDzBnccK3ZZGf5_AZTk. WSO2 API Manager enforces OAuth security on all the published APIs. Consumers who talk to the API Manager should send their credentials (application key) as per the OAuth bearer token profile. If you don't send an application key or send a wrong key, you will receive a 401 Unauthorized response in return.

6. Note the response for the API invocation that appears as follows:

Request URL

```
http://10.100.1.71:8280/phoneverify/1.1.0
```

Response Body

```
<html>

    <head>
  <link rel="alternate" type="text/xml" href="/phoneverify/phoneverify.asmx?disco" />

      <style type="text/css">

            BODY { color: #000000; background-color: white; font-family: Verdana; margin-left: 0px; margin-top: 0
px; }
            #content { margin-left: 30px; font-size: .70em; padding-bottom: 2em; }
            A:link { color: #336699; font-weight: bold; text-decoration: underline; }
            A:visited { color: #6699cc; font-weight: bold; text-decoration: underline; }
            A:active { color: #336699; font-weight: bold; text-decoration: underline; }
            A:hover { color: cc3300; font-weight: bold; text-decoration: underline; }
            P { color: #000000; margin-top: 0px; margin-bottom: 12px; font-family: Verdana; }
            pre { background-color: #e5e5cc; padding: 5px; font-family: Courier New; font-size: x-small; margin-to
p: -5px; border: 1px #f0f0e0 solid; }
            td { color: #000000; font-family: Verdana; font-size: .7em; }
            h2 { font-size: 1.5em; font-weight: bold; margin-top: 25px; margin-bottom: 10px; border-top: 1px solid
#003366; margin-left: -15px; color: #003366; }
            h3 { font-size: 1.1em; color: #000000; margin-left: -15px; margin-top: 10px; margin-bottom: 10px; }
```

Response Code

```
200
```

Response Headers

```
Content-Type: text/html; charset=utf-8
```

7. Within a minute after the first API invocation, make another attempt to invoke the API and note that the second invocation results in a throttling error.

   This is because you applied a Bronze tier at the time you subscribed to the API and the Bronze tier only allows one API call per minute.

Response Body

```
<amt:fault xmlns:amt="http://wso2.org/apimanager/throttling">
  <amt:code>900800</amt:code>
  <amt:message>Message Throttled Out</amt:message>
  <amt:description>You have exceeded your quota</amt:description>
</amt:fault>
```

Adding Apache Solr-Based Indexing

The API Manager has Apache Solr based indexing for API documentation content. It provides both the API Publisher and Store full-text search facility to search through API documentation, find documents and related APIs. The search syntax is **doc:keyword.** Search criteria looks for the keyword in any word/phrase in the documentation content and returns both the matching documents and associated APIs.

The following media types have Apache Solr based indexers by default, configured using the `<Indexers>` element in `<APIM_HOME>/repository/conf/registry.xml`.

- Text : text/plain
- PDF : application/pdf
- MS word : application/msword
- MS Powerpoint : application/vnd.ms-powerpoint
- MS Excel : application/vnd.ms-excel
- XML : application/xml

Writing a custom index

In addition to the default ones, you can write your own indexer implementation and register it as follows:

1. Write a custom indexer. Given below is a sample indexer code.

```
package org.wso2.indexing.sample;

import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Arrays;
import org.apache.solr.common.SolrException;
import org.wso2.carbon.registry.core.exceptions.RegistryException;
import org.wso2.carbon.registry.core.utils.RegistryUtils;
import org.wso2.carbon.registry.indexing.IndexingConstants;
import org.wso2.carbon.registry.indexing.AsyncIndexer.File2Index;
import org.wso2.carbon.registry.indexing.indexer.Indexer;
import org.wso2.carbon.registry.indexing.solr.IndexDocument;

public class PlainTextIndexer implements Indexer {
    public IndexDocument getIndexedDocument(File2Index fileData) throws
SolrException,
            RegistryException {

             /* Create index document with resource path and raw content*/
             IndexDocument indexDoc = new IndexDocument(fileData.path,
RegistryUtils.decodeBytes(fileData.data), null);

             /* You can specify required field/value pairs for this indexing
document.
              * When searching we can query on these fields  */
             Map<String, List<String>> fields = new HashMap<String,
List<String>>();
             fields.put("path", Arrays.asList(fileData.path));

             if (fileData.mediaType != null) {
                         fields.put(IndexingConstants.FIELD_MEDIA_TYPE,
Arrays.asList(fileData.mediaType));
             } else {
                         fields.put(IndexingConstants.FIELD_MEDIA_TYPE,
Arrays.asList("text/plain"));
             }

             /* set fields for index document*/
             indexDoc.setFields(fields);
             return indexDoc;
    }
}
```

2. Add the custom indexer JAR file to `<APIM_HOME>/repository/components/lib` directory.
3. Update the `<Indexers>` element in `<APIM_HOME>/repository/conf/registry.xml` file with the new indexer. The content is indexed using this media type. For example,

```
<indexers>
     <indexer class="org.wso2.indexing.sample.PlainTextIndexer"
mediaTypeRegEx="text/plain" profiles="default,api-store,api-publisher"/>
</indexers>
```

The attributes of the above configuration are described below:

class	Java class name of the indexer
mefiaTypeRegEx	A regex pattern to match the media type
profiles	APIM profiles in which the indexer is available

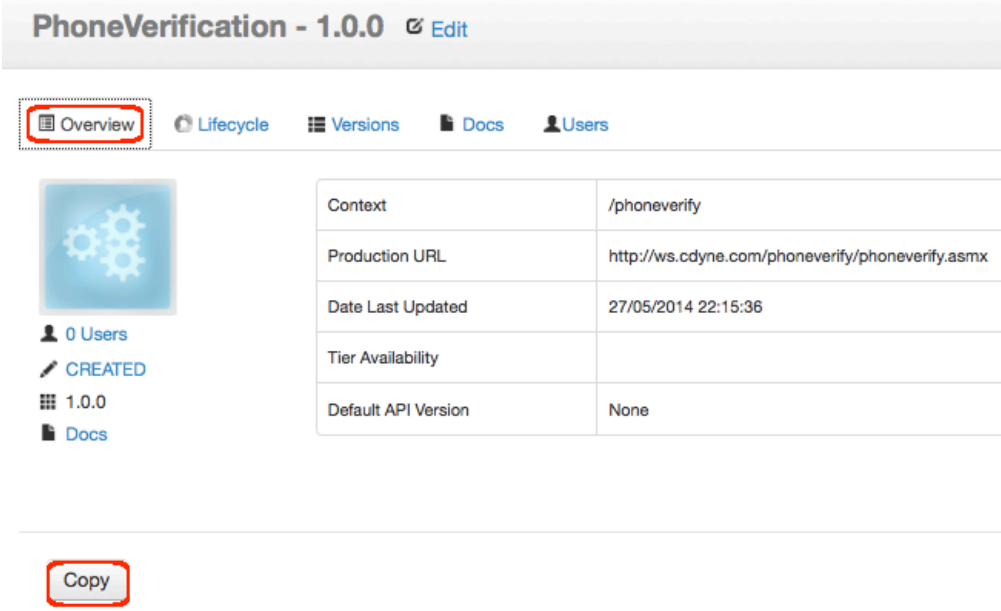
4. Restart the server.
5. Add API documentation using the new media type and then search some term in the documentation using the syntax (**doc:keyword**). You will see how the documentation has got indexed according to the media type.

Versioning APIs

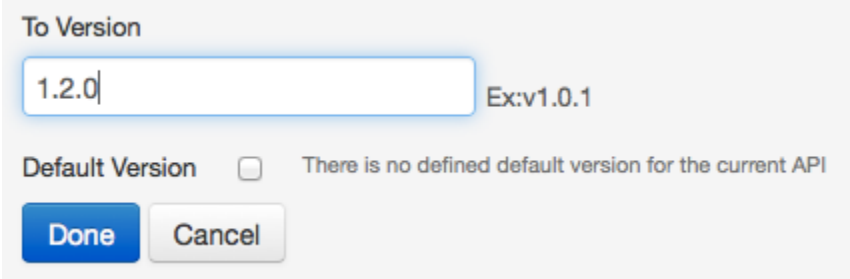
After creating an API, you might want to change its behavior, authentication mechanism, resources, throttling tiers, target audiences etc. at a later point in time, depending on new business or technical needs of the organization. But, you cannot do these changes to an API that is already published and has users plugged into it. A published API should be fixed. The way to modify it is by publishing a new version of the API with changes. The general practice is to encourage users to adopt the new version by depreciating the old API after giving them time to test their applications with the new API.

The API Manager facilitates API versioning as part of API life cycle management. The steps below show how to create a different version of an existing API.

1. Log in to the API Publisher as a user who has the creator role assigned. For more information on creating Managing Users and Roles.users and assigning roles, refer to section
2. Available APIs shows in the **All APIs** window of the API Publisher. Click on the API that you want to create a version of.
3. The API's **Overview** tab opens. Create a copy of the API using the **Copy** button in the **Overview** tab. For example,



4. S pecify a new API version. Generally recommended format is `version.major.minor`. For example, 1.2.0.



When several versions of an API exist, you might want to make one of them the default version. For more

information, see Default API version.

5. The newly-added version appears in the **All APIs** window of API Publisher.

After creating the new version, a user who has the publisher role assigned can publis h the API. At the time you publish it, you can select the **Deprecate Old Versions** option to automatically deprecate all previous versions of the API.

Publishing to API Stores.Next, see

Publishing to API Stores

While an API is the published interface, a corresponding service running in the back-end handles its actual implementation. APIs have their own lifecycle, independent from the back-end service they rely on. This section covers the following:

- The default API lifecycle
- Publishing an API
  - Publishing to multiple external API stores

The default API lifecycle

The default API lifecycle has the following stages:

- **CREATED**: API metadata is saved, but it is not visible to subscribers yet, nor deployed to the API Gateway.
- **PROTOTYPED** : API is deployed and published in the API Store as a prototype. A prototyped API is usually a mock implementation made public in order to get feedback from users about its usability. Users cannot subscribe to a prototyped API. They can only try out its functionality.
- **PUBLISHED**: API is visible in API Store, and eventually published if the `Propagate Changes to API Gateway` option is selected at publishing time.
- **DEPRECATED**: API is still deployed into API Gateway (available at runtime to existing users), but not visible to subscribers. An API is automatically deprecated when a new version is published.
- **RETIRED**: API is unpublished from the API gateway and deleted from the store.
- **BLOCKED**: Access is temporarily blocked. Runtime calls are blocked and the API is not shown in the API store anymore.

The diagram below shows the general API and backend service life cycle elements.

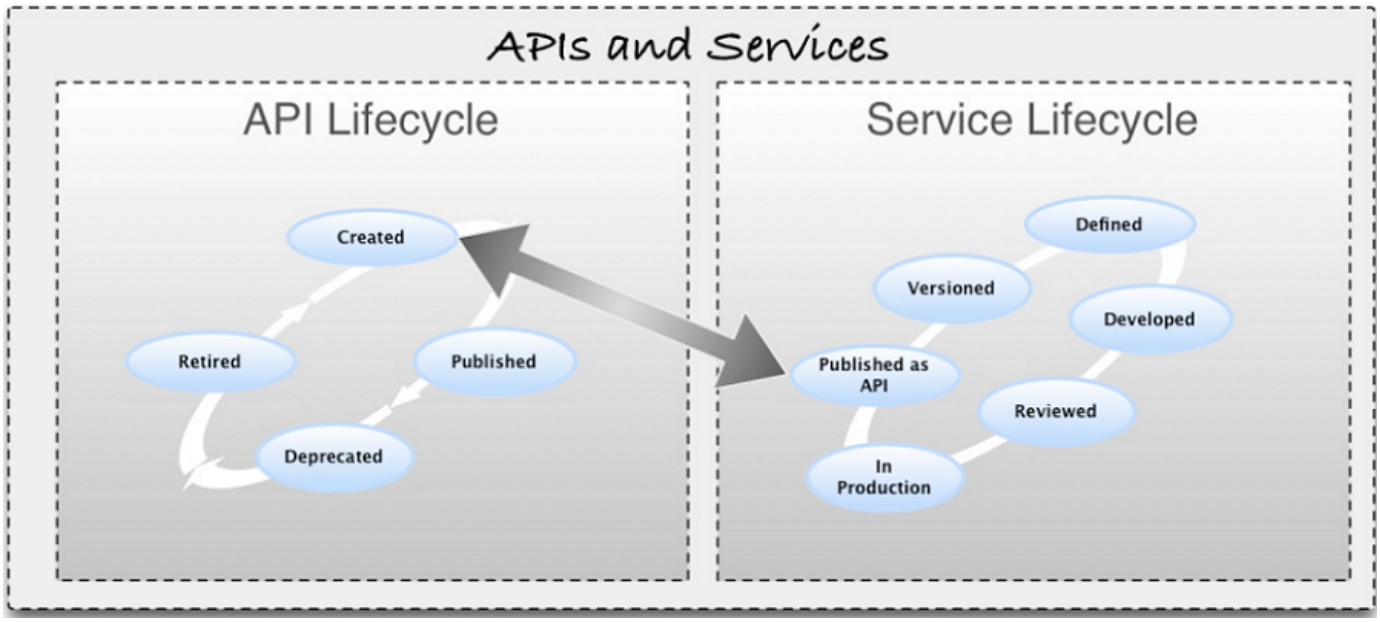


Figure: API and backend service life cycle elements

API Publisher has a separate tab called **Lifecycle** using which you can publish APIs to the API Store, depreciate, retire and perform other operations to an API. The Life Cycle tab is only visible to and manageable by a user who is assigned the publisher role. For instructions on creating a user with the publisher role, see

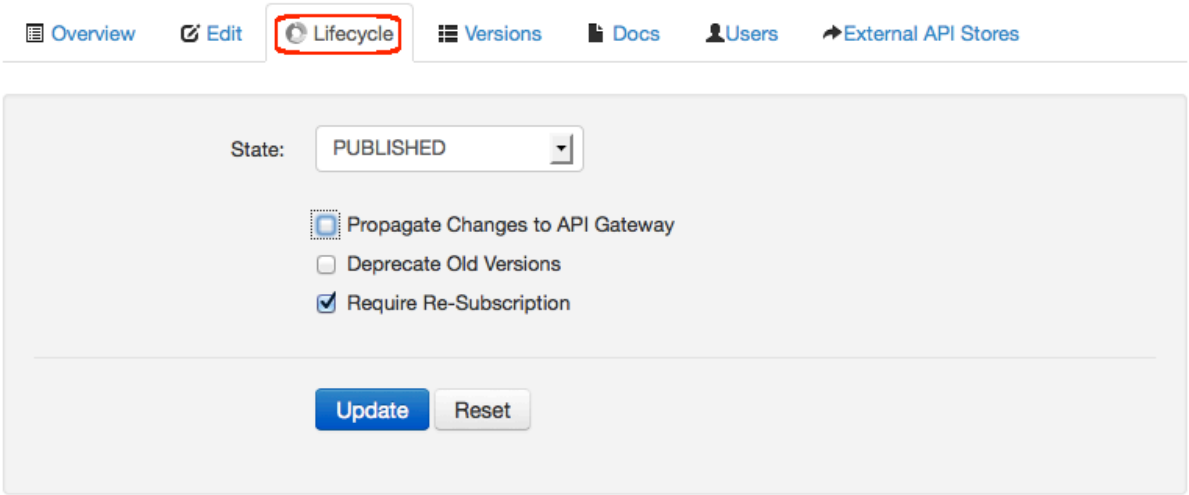
Managing Users and Roles. Let's take a look at how to perform some common life cycle operations on an API.

Publishing an API

1. Log in to the API Publisher (https://<HostName>:9443/store) as a user who has the `publisher` role assigned. See Managing Users and Roles.
2. Click on an API that you want to publish.
3. The API's overview window opens. Click the **Life Cycle** tab, which displays the API's available states.

   > The Life Cycle tab is only visible to users with publisher privileges.

4. To publish the API, select the PUBLISHED state from the drop-down list. You get three check boxes to select as follows:



   Propagate Changes to API Gateway

   Used to define an API proxy in the API Gateway runtime component, allowing the API to be exposed to the consumers via the API Gateway. If this option is left unselected, the API metadata will not change and you will have to manually configure the API Gateway according to the information published in the API Store.

   Deprecate Old Versions

   If selected, any prior versions of the API will be set to the DEPRECATED state automatically.

   Require Re-Subscription

   Invalidates current user subscriptions, forcing users to subscribe again.
5. Select the necessary options and click the **Update** button to publish the API to the API Store.

Similarly, you can deprecate, retire and block APIs.

Publishing to multiple external API stores

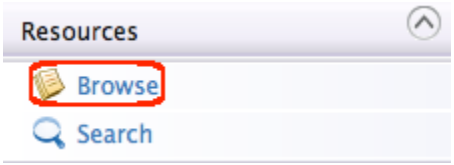
API publishers can share an API to application developers who are subscribed to multiple tenant-specific API Stores. This allows them to expose APIs to a wider community.

> ⚠ After publishing an API to external stores, it will be visible to the users of those stores. However, to subscribe to the API, the users must visit the original publisher's store.

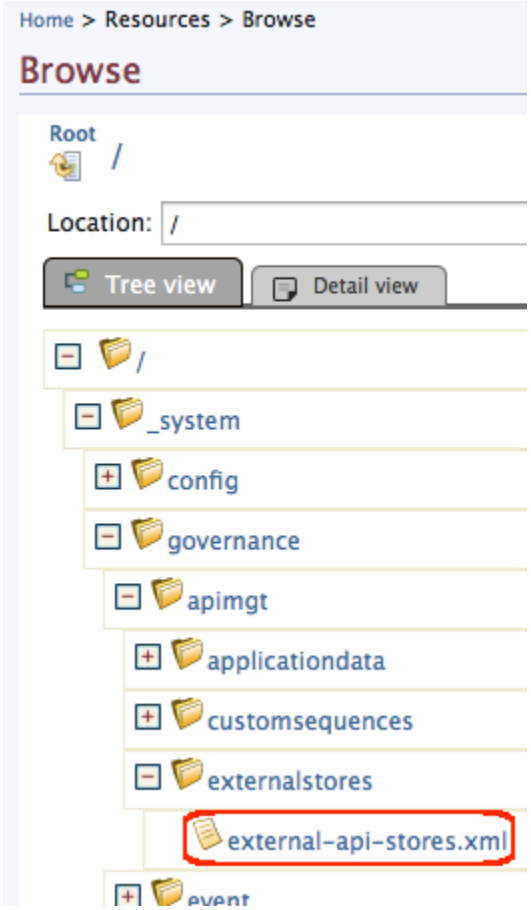
Follow the steps below to configure:

1. Log in to APIM admin console (`https://<Server Host>:9443/carbon`) as admin and select **Browse** m

enu under **Resources** .



2. The Registry opens. G o to `/_system/governance/apimgt/externalstores/external-api-stores.xml` r e s o u r c e .



3. Click the **Edit as Text** link and change the `<ExternalAPIStores>` element of each external API store that you need to publish APIs to. For example,

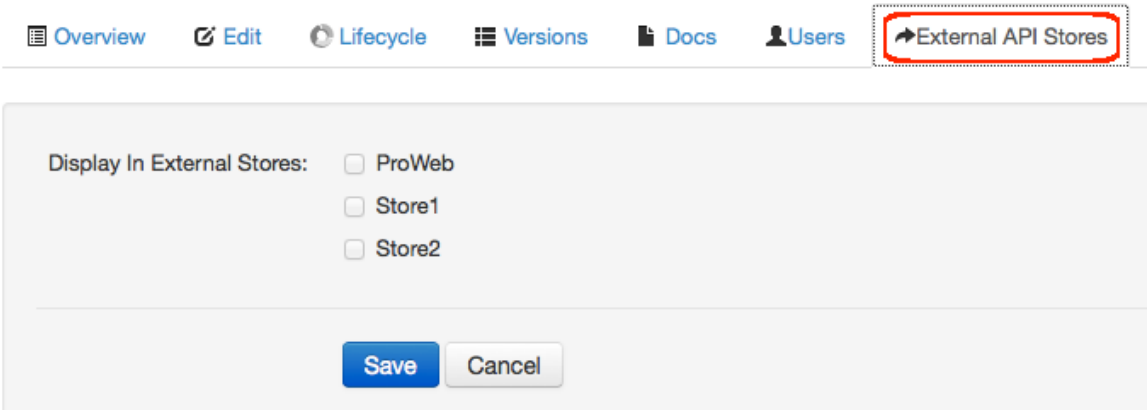
```
<ExternalAPIStores>
        <StoreURL>http://localhost:9763/store</StoreURL>
        <ExternalAPIStore id="Store1" type="wso2">
            <DisplayName>Store1</DisplayName>
            <Endpoint>http://localhost:9763/store</Endpoint>
            <Username>xxxx</Username>
            <Password>xxxx</Password>
        </ExternalAPIStore>
       <ExternalAPIStore id="ProWeb" type="proWeb">
            <Name>ProgrammableWeb</Name>
            <Endpoint>xxxxx</Endpoint>
        </ExternalAPIStore>
        <ExternalAPIStore id="Store2" type="wso2">
            <DisplayName>Store2</DisplayName>
            <Endpoint>http://localhost:9764/store</Endpoint>
            <Username>xxxx</Username>
            <Password>xxxx</Password>
        </ExternalAPIStore>
</ExternalAPIStores>
```

Note the following in the configuration above:

Element	Description
`<ExternalAPIStore id="" type="">`	`id`: The external store identifier, which is a unique value. `type`: Type of the Store. This can be a WSO2-specific API Store or an external one.
`<StoreURL>`	URL of the API store of the current APIM deployment. This is the URL to the API in the original publisher's store. APIs that are published to external stores will be redirected to this URL.
`<DisplayName>`	The name of the Store that is displayed in the publisher UI.
`<Endpoint>`	URL of the API Store.
`<Username>` & `<Password>`	Credentials of a user who has permissions to create and publish APIs.

Registry changes are applied dynamically. You do not need to restart the server.

4. Using the management console, create an API.
5. Click on the newly created API to see a new tab c alled console.



Note the following:

- You can select multiple external API stores and click **Save** to publish your API to them.
- I f the API creator updates the API after publication to external stores, either the creator or a publisher can simply push those changes to the published stores by selecting the stores and clicking **Save** again .
- If the API creator deletes the API, each external store that it is published to will receive a request to delete the API.

6. Log in to an external API store where the API is published to and click it to open.
7. A link appears as **View Publisher Store** and it directs you to the original publisher's store through which you can subscribe to the API.

Next, see how to manage subscriptions and access tokens in Managing API Usage.

Managing API Usage

API Publisher provides several mechanisms to control and monitor subscriber usage and monetize APIs. The following topics describe some of them:

- Blocking subscriptions
- Monitoring and billing

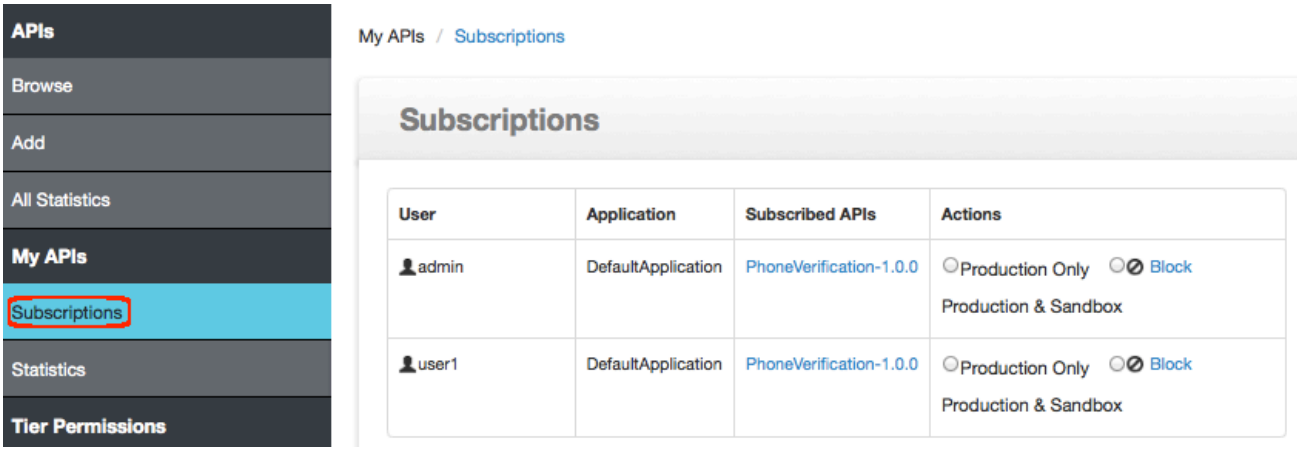
Blocking subscriptions

The API creator can block a particular subscription on an API to disable access to it until s/he decides to unblock it again. Once an API creator blocks access on a selected subscription, neither a consumer nor the application owner can invoke the subscribed API from the application, until it is unblocked again. This feature allows API creators to control usage of APIs among API consumers. The blocking can be done in two levels.

- **Block production and sandbox access**: API access is blocked with both production and sandbox keys.
- **Block production access only**: This blocking allows sandbox access. This is useful when a user wants to fix and test an issue in an API. Rather than blocking all access, the manager can block production access only, allowing the developer to fix and test.

API Publisher provides you the **Subscriptions** page to view and manage all subscriptions to the APIs you created. The steps below explain how to view subscriptions and revoke access rights.

1. Log in to the API Publisher (`https://<HostName>:9443/publisher`) as a user who has the creator role assigned. For more information on creating users and assigning roles, refer to section Managing Users and Roles.
2. Click the **Subscriptions** menu to open the **Subscriptions** window.



The window displays the following information:

- **Users**: Usernames of users who have subscribed to the API through the API Store. For instructions on subscribing, see Subscribing to APIs.
- **pplication**: An application is a logical collection of one or more APIs, and is required when subscribing to an API.
- **Subscribed APIs** : List of all APIs a given user is subscribed to on a given application. Since API keys are generated at the application-level and valid for all APIs that are associated with an application, all APIs subscribed through the same application can be accessed using a single API key.
- **Actions**: The supported actions on each subscription. Currently, the API Manager provides **Block** acti on to each subscription. It allows the API creator to block a particular subscription on an API. Once a subscription is blocked, neither its users nor the application owners can invoke the subscribed API from the application. To allow APIs invocations back, the API creator has to unblock the subscription.

3. To block a subscription, go to the **Actions** column. Choose one of the available Blocking options (e.g., Production or Production & Sandbox) and click **Block**. The link immediately turns to **Unblock**. You can click **Unblock** any time to unblock the subscription and allows API consumers to use the subscription again. Note that when API Gateway caching or Key Manager caching is enabled (validation information cache), even after blocking a subscription, user can access APIs until the cache expires. By default, Gateway caching is enabled in the API Manager.

Note

In an environment where Gateway caching is enabled (which it is by default), blocking a subscription will not affect the tokens that are already cached on the Gateway. Meaning that tokens belonging to the particular subscription will still be active on the Gateway until the cache is invalidated/expired.

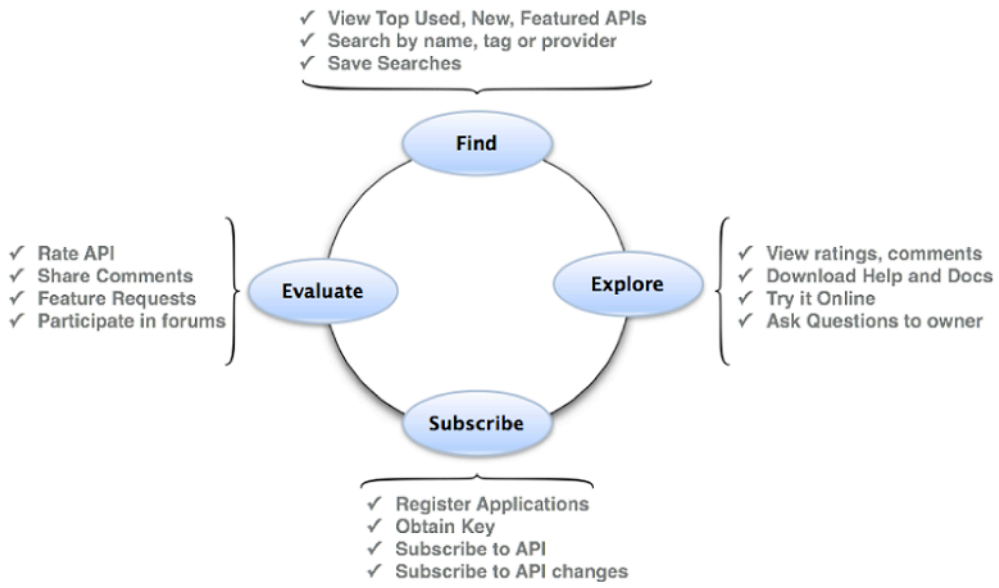
Monitoring and billing

Monitoring, Statistics and Billing.For information, see

Application Developer Guide

API Manager provides a structured Web interface called the **WSO2 API Store** to host published APIs. API consumers and partners can self-register to it on-demand to find, explore and subscribe to APIs, evaluate available resources and collaboration channels. The API store is where the interaction between potential API consumers and API providers happen. Its simplified UI reduces time and effort taken when evaluating enterprise-grade, secure, protected, authenticated API resources.

Shown in the diagram below are common API consumer life cycle activities supported by the API Store:

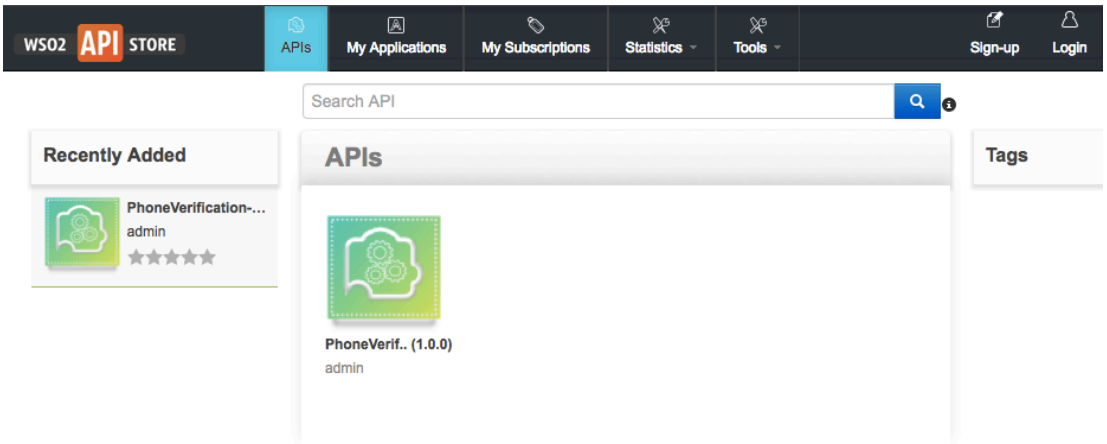


Before accessing the Web interface of the API Store, make sure you run the API Manager using instructions given in section Running API Manager . Once the server is up, type the following URL in your browser to access the API Store Web interface.

```
https://<HostName>:9443/store
```

You cannot access the API Store Web interface using HTTP. It is exposed as HTTPS only.

The API Store opens in anonymous mode as follows:



In anonymous mode, the API Store displays all public APIs that are published. Any user can click on a selected API to view its information, documentation and search APIs by name without logging in to the API Store. Search is not case-sensitive.

Follow the sections below for API Store functionality:

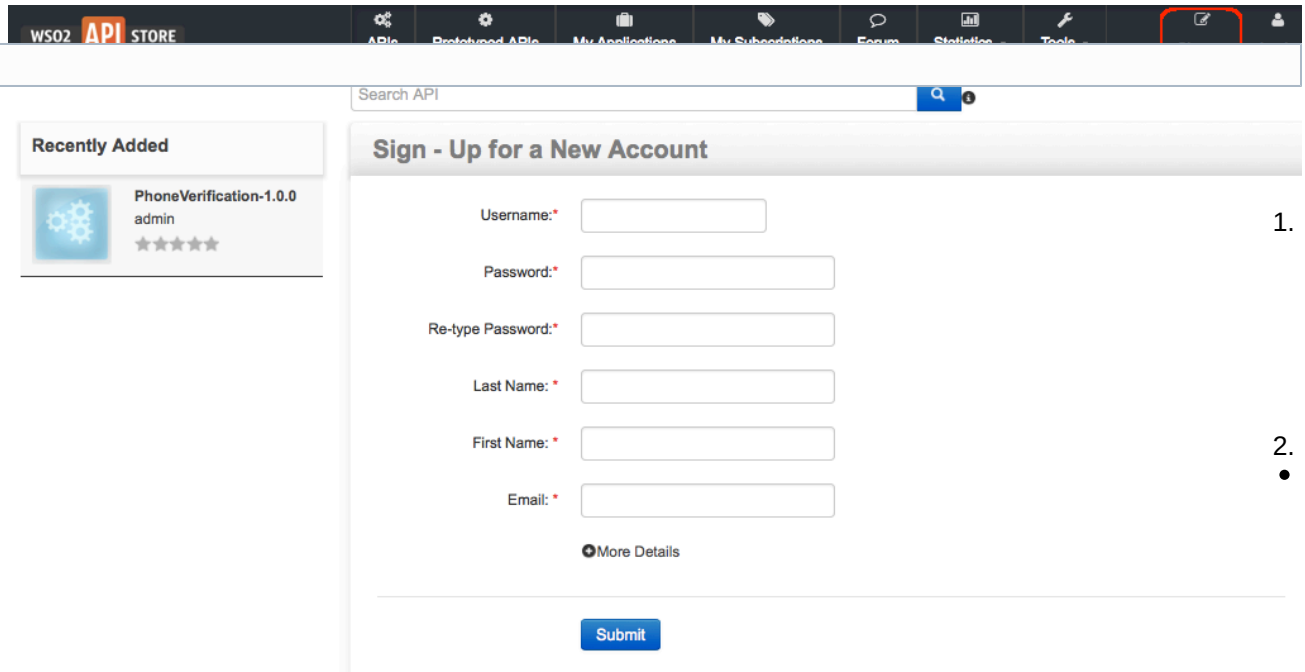
- Signing up to API Store
- Subscribing to APIs
- Working with Access Tokens
- Invoking APIs
- Engaging with Community

Signing up to API Store

Anonymous users can self-subscribe to the API Store using the instructions given below.

> To disable the self signup capability, set **`<SelfSignUp><Enabled>` element to false** in the `<APIM_HOME>/repository/conf/api-manager.xml` file.

1. Open the API Store Web application in a browser by typing the URL: `https://<HostName>:9443/store`.
2. Click the **Sign-up** link that appears in the top, right-hand corner of the window, fill the sign-up form that appears
   and click the **Submit** button.

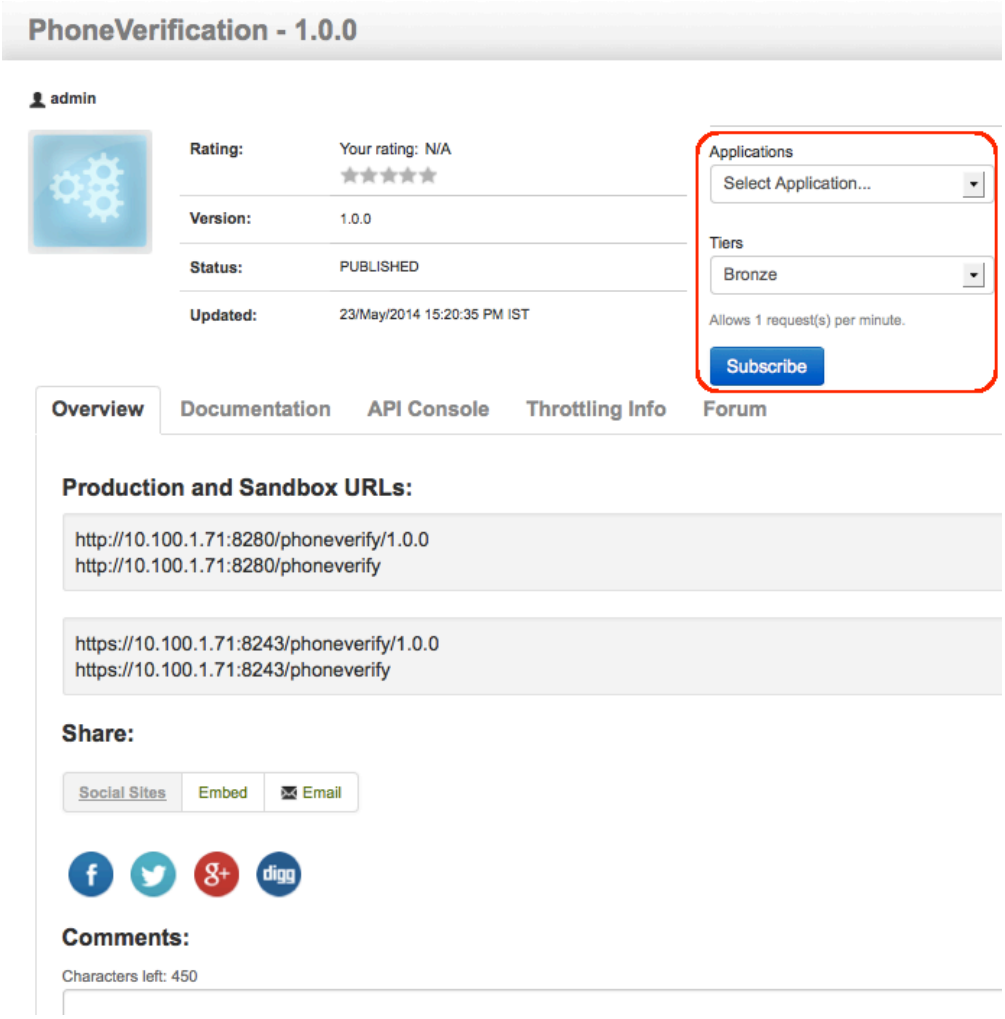


1.

2.

- User Roles in API Manager. This role has permission to subscribe to and consume APIs.
  You can also plug an external BPEL workflow to the user sign-up process. See Adding Workflow Extensions.
- **Tags** to look for APIs under a specific topic/label.

3. Click on a selected API to view its details. For example,



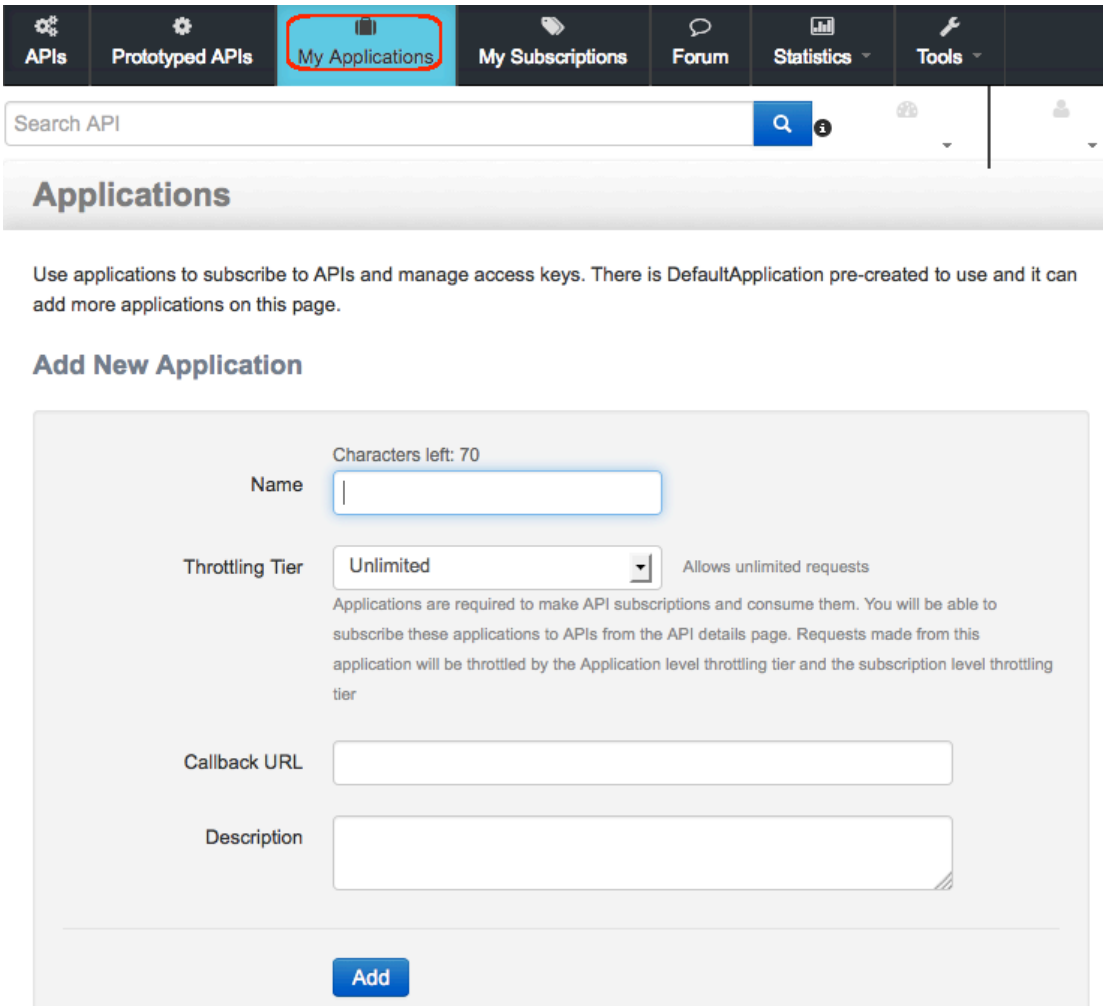
Note that subscribed users can add ratings and comments to an API or share it in social media, e-mail or their Websites. Note that the example above has two URLs for each for production and sandbox. This is because the API is marked as a default API. For information, see Default API version.

4. To subscribe to this API, select an application from the **Applications** drop-down list. You can use the default application named **DefaultApplication**, or create a new one.

Applications

An application is a logical collection of one or more APIs, and is required when subscribing to an API. Consumers can create a logical application in WSO2 API Manager or use an existing one to subscribe to all the relevant APIs using that application. To invoke any API in an application, you need to obtain a key. Applications decouple the consumers from the APIs and allow a consumer to generate and use a single key to a collection of APIs in an application. Applications also enable a consumer to subscribe to one API multiple times with different SLA levels.

5. Click the **New Applications** to open the **Add New Application** page.



Through this window, new applications can be created, and the existing applications can be edited or deleted.

Application-Level Throttling Tiers

An application can be available to a consumer at different levels of service. For example, if you have infrastructure limitations in facilitating more than a certain number of requests to an application at a time, the throttling tiers can be set accordingly so that the application can have a maximum number of requests within a defined time. WSO2 API Manager comes with three default tiers, which are 'Gold', 'Silver' and 'Bronze' as defined below:

- Bronze - Allows 1 request per minute.
- Silver - Allows 5 requests per minute.
- Gold - Allows 20 requests per minute.

In addition, there is also a special tier called 'Unlimited' which gives unlimited access. The WSO2 API Manager provides an application out of the box by the name **Default Application** and it can have any number of requests per minute. That is, its throttling tier is unlimited. You can change this and set it to a restricted limit by editing the default application.
In addition to application-level throttling, you can also define other levels of throttling tiers. The final request limit granted to a given user on a given API is ultimately defined by the summed output of all of these different throttling tiers together. For example, lets say two users subscribe to an API using the Gold subscription, which allows 20 requests per minute. They both use the application App1 for this subscription, which again has a throttling tier set as 20 requests per minute. All resource level throttling tiers are unlimited. In this scenario, although both users are eligible for 20 requests per minute access to the API, each ideally has a limit of only 10 requests per minute. This is due to the application-level limitation of 20 requests per minute.

Callback URL

A callback URL is optional for an application. If specified, you can use it in the authorization code grant type when invoking an API. See Generating authorization code.

6. Once an application is selected, select a tier (API-level throttling tier) for the subscription from the **Tiers** drop-down list. This list of tiers is defined for the API at the time of API creation as described in section Adding an API -> Tier Availability .

   The description of each tier is shown below the **Throttling Tiers** field.
7. Once an application and a tier is selected, click the **Subscribe** button.
8. If the subscription is successful, a message appears. Select **Go to My Subscriptions**.
9. The **My Subscriptions** tab opens. You have now successfully subscribed to an API.

If the subscribed API needs authentication to invoke it, you need to have an access token before using the API in your applications. Find out how to obtain an access token to invoke an API.

Working with Access Tokens

Access tokens are used to authenticate users to invoke an API. Access tokens are generated by API consumers and need to be passed in the incoming API requests. The API key (i.e., the generated access token) is a simple string that is passed as an HTTP header. For example, `"Authorization: Bearer NtBQkXoKElu0H1a1fQ0DWfo6IX4a."` It works equally well for SOAP and REST calls.

Authorizing requests coming to published APIs using access tokens helps you **prevent DoS attacks**. If the token passed with a request is invalid, the API Manager discards that requests in the first stage of processing itself.

WSO2 API Manager provides two types of access tokens for authentication:

- **Application Access Tokens** : Tokens to identify and authenticate an entire application. An application is a logical collection of many APIs. With a single application access token, you can invoke all of these APIs.
- **User Access Tokens** : Tokens to identify the final user of an application. For example, the final user of a mobile application deployed on different devices.

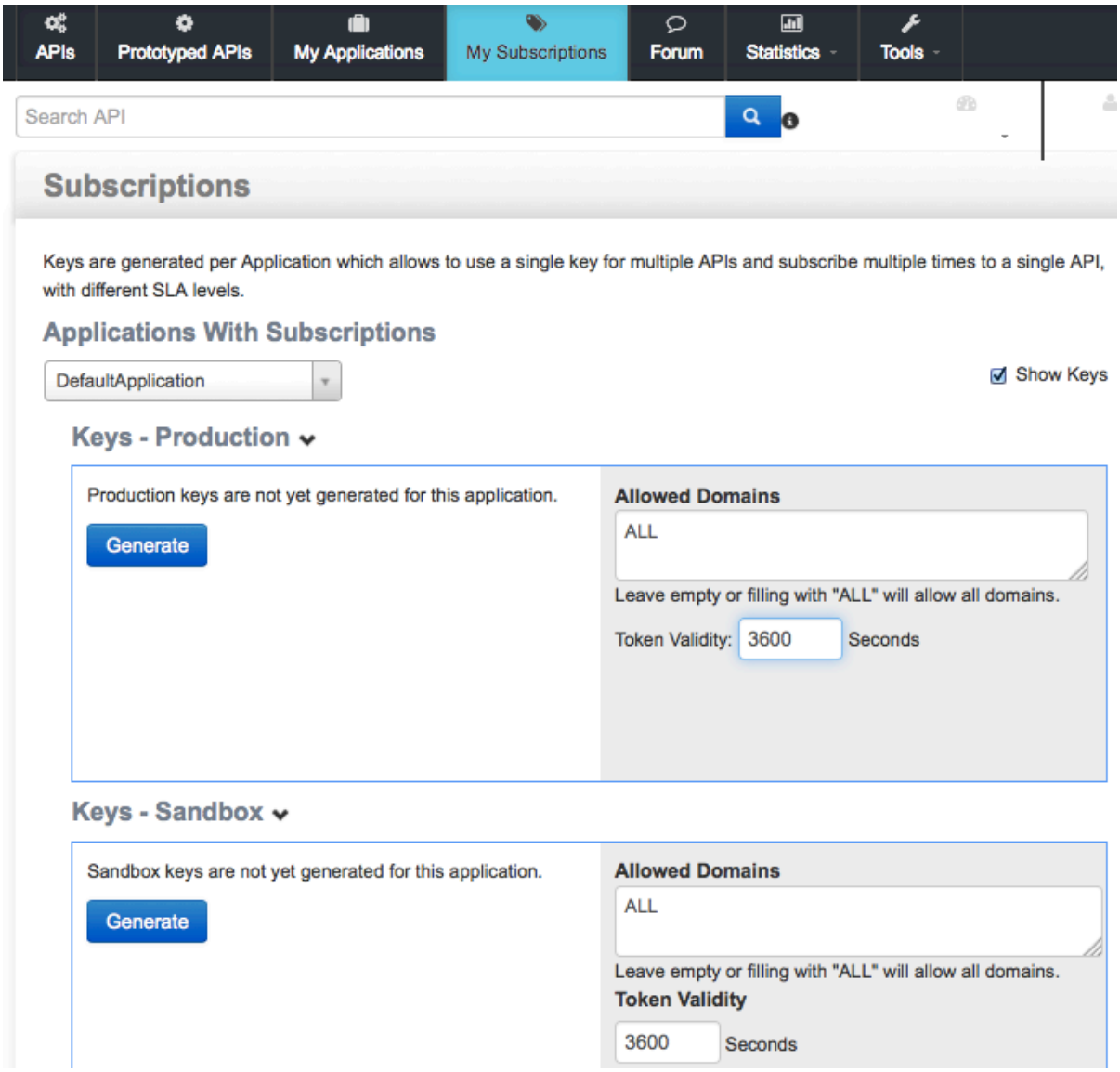
Let's take a look at how to generate and renew each type of access token in detail.

- Generating application access tokens
- Restricting access to specific domains
- Generating user access tokens
- Renewing application access tokens
- Renewing user access tokens
- Changing the default token expiration time

Generating application access tokens

Application access tokens are generated at the application level and valid for all APIs associated with an application. This allows you to access multiple APIs with a single token and also subscribe multiple times to a single API with different SLA levels. It leverages OAuth2 to provide a simple, easy-to-use key management mechanism. Following steps describe how to generate application access tokens.

1. Log in to the API Store (`https://<hostname>:9443/store`).
2. Click **My Subscriptions** from the menu bar at the top of the screen. The **Subscriptions** page opens with the following options:



Generate button: Use the **Generate** button associated with each type of key to generate an access token. It generates an application key and also a consumer key and a consumer secret. For testing purposes, you also can create a sandbox key.

Allowed Domains Text Area: Allows you to associate a set of domains, as a comma separated list, to a token. API Gateway allows requests to the APIs from those domains only. Others will be restricted. Leave this field blank to allow all domains.

With this allowed domains feature, a client from a different domain cannot access an API even if an application key is stolen (when the key is placed in client side JS code). Whenever an API call happens, the Gateway checks if the request originated from an allowed list of domains. This is the same solution done in Google Maps.

When the client makes a request to an API that is only allowed to some domains, the request message must have an HTTP header to specify its domain name. APIM admin can configure this header name using `<ClientDomainHeader>` element under the `<APIGateway>` element in `<APIM_HOME>/repository/conf/api-manager.xml`. For example, if the file contains `<ClientDomainHeader>domain</ClientDomainHeader>`, then the API invocation request must contain an HTTP header called `domain` with values as shown in the example below:

```
curl -v -H "Authorization: Bearer xxx" -H "domain: wso2.com" http://localhost:8280/twitter/1.0.0/search.atom?q=cat
```

Sending this header is mandatory only if the API is restricted to certain domains.

Token Validity Text Area: Set a period in which the token will be expired after generation. Set to a negative value to ensure that the token never expires. Also see Changing the default token expiration time.

Generating user access tokens

User access tokens are generated at user-level and valid for all APIs subscribed to a user. User-level tokens allow u sers to invoke an API even from a third-party application like a mobile app. You can generate a user-level token by c alling the API Manager Login API through a REST client. For more information on generating user-level tokens, refer to Token APIs.

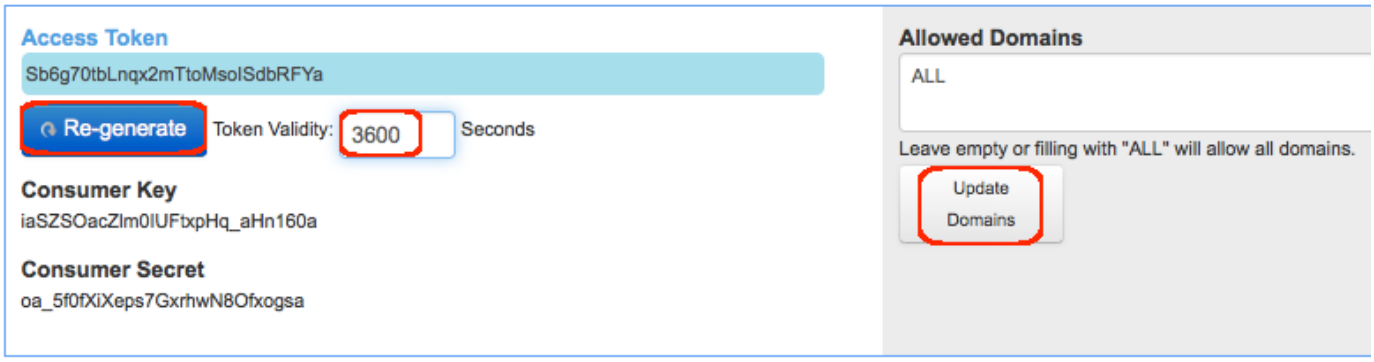
After an access token is generated, users sometimes want to renew the old token due to expiration or security concerns. API Consumers can re-generate/refresh access tokens in the following ways.

> By default, access tokens, consumer keys and consumer secrets are not encrypted in the database. An admin can enable encryption as follows:
>
> - Set the value of the <EncryptPersistedTokens> to true inside the <APIKeyManager> section of the <APIM_HOME>/repository/conf/api-manager.xml file.
> - Change the `<TokenPersistenceProcessor>` to `org.wso2.carbon.identity.oauth.tokenprocessor.EncryptionDecryptionPersistenceProcessor` in the `<APIM_HOME>/repository/conf/identity.xml` file.

> If you want to keep authorization headers in massages that are going out of the API Gateway, an admin can go to `<API Gateway Node>/repository/conf/api-manager.xml` file, uncomment the `<RemoveOAuthHeadersFromOutMessage>` element, set its value to `false` and then restart the server to apply the changes.

Renewing application access tokens

When an application access token expires, consumers can refresh the token by logging into API Store, selecting the **My Subscriptions** link at the top of the screen, and clicking **Re-generate**. You can also specify a token expiration time for the application access token or change its allowed domains. Set to a negative value to ensure that the token never expires.



Renewing user access token

To renew a user token, issue a REST call to WSO2 Login API through a REST client. For more information, see Renew User Tokens.

> You can configure the API Manager instances to store access tokens in different tables according to their user store domain. This is referred to as **user token partitioning** and it ensures better security when there are multiple user stores in the system. For configuration details, see user token partitioning.

Changing the default token expiration time

Access Tokens have a expiration time, which is set to 60 minutes by default.

- To change the default expiration time of application access tokens,
  - Change the value of `<ApplicationAccessTokenDefaultValidityPeriod>` element in `<APIM_HOME>/repository/conf/identity.xml` file. Set to a negative value to ensure that the token never expires.
  - Alternatively, you can set a default expiration time through the UI when generating/regenerating the application access token. This is explained in previous sections.
- Similarly, to change the default expiration time of user access tokens, edit the value of `<UserAccessTokenDefaultValidityPeriod>` element in `identity.xml` file.

Also see Configuring Caching for several caching options available to optimize key validation.

After subscribing to an API and generating a key to access it, the next step is to invoke the API through the Gateway using the steps given in section Invoking APIs.

Invoking APIs

There is a number of utilities available for invoking APIs. Some of them are covered in the following topics:

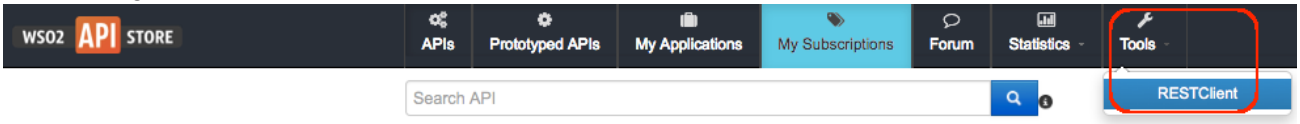
- Browser-based REST clients

Browser-based REST clients

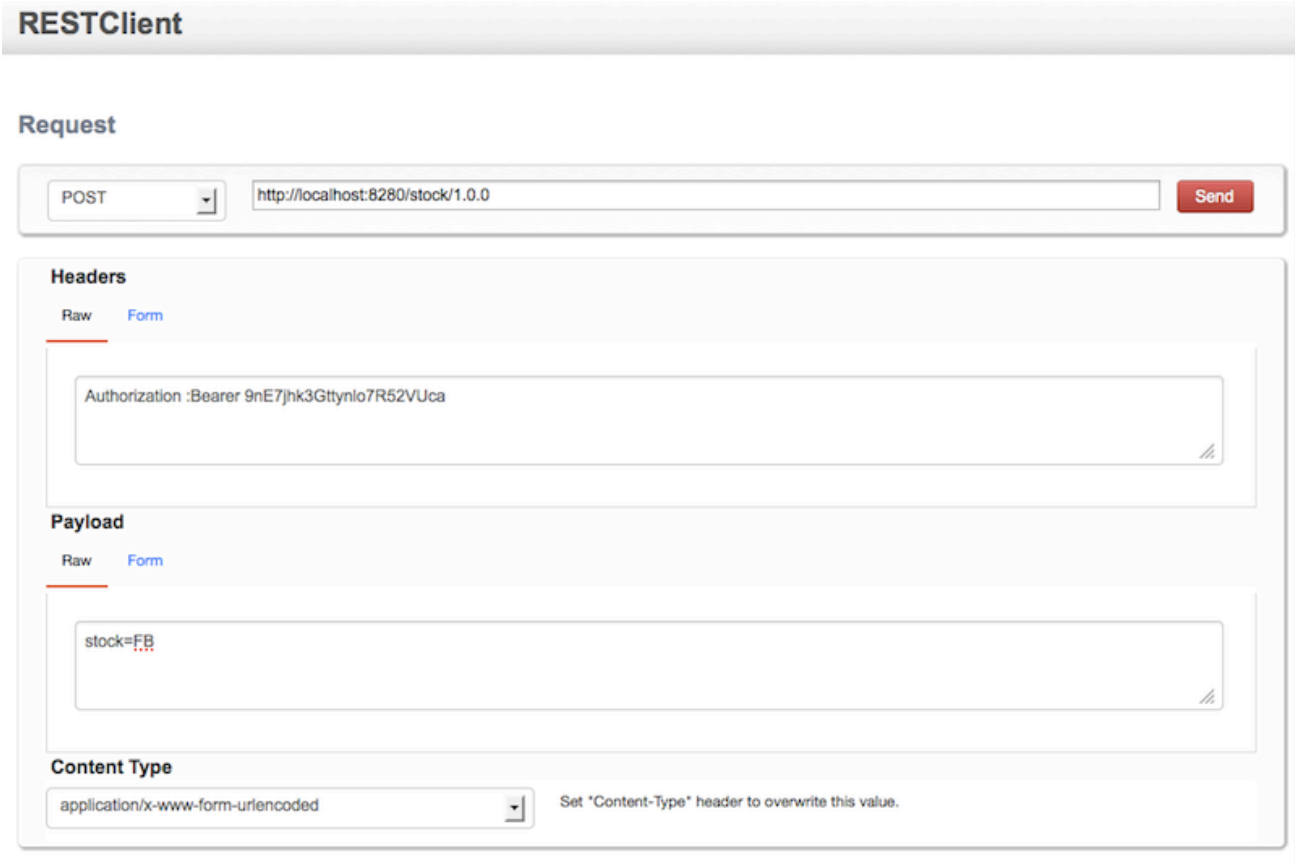
WSO2 API Manager comes with a REST Client by default. It helps you to invoke and test an API through the API Store. WSO2 REST Client has a simple Web interface and facilitates a range of HTTP verbs from simple GET method to POST, PUT, DELETE, OPTIONS. It also includes capability to move data around in header and payload fields. The REST Client is a useful alternative to similar tools like cURL.

Follow the instructions below to invoke the REST Client.

1. Open the API Store (`https://<YourHostName>:9443/store`) in a Web browser.
2. The **REST Client** menu appears under the **Tools** menu at the top of the screen. Clicking it opens the REST client on a global level.



3. For example, shown below is how to invoke Google API using the REST Client.



API URL

The API URL takes the form `http://host:8280/<context>/<version>/<back end service requirements included as parameters>`. For example, `http://localhost:8280/stock/1.0.0`.

Header

In the above request, the application key generated at the time a user subscribes to an API is passed with the authorization header, which is prefixed by the string "Bearer". This is because, WSO2 API Manager enforces OAuth security on all the published APIs. Any consumer that talks to the API Manager should send their credential (application key) as per the OAuth bearer token profile. If you don't send an application key or send a wrong key, you will receive a 401 Unauthorized response in return.

> ⚠ The number of API calls you can make depends on the throttling tier applied to the API. For example, if a Bronze tier is applied, the number of API calls is limited to 1 per minute. Another attempt to call the API during that time results in a throttling error.

Engaging with Community

WSO2 API Manager provides capability to build and nurture an active community of API consumers with various community features such as commenting and rating.

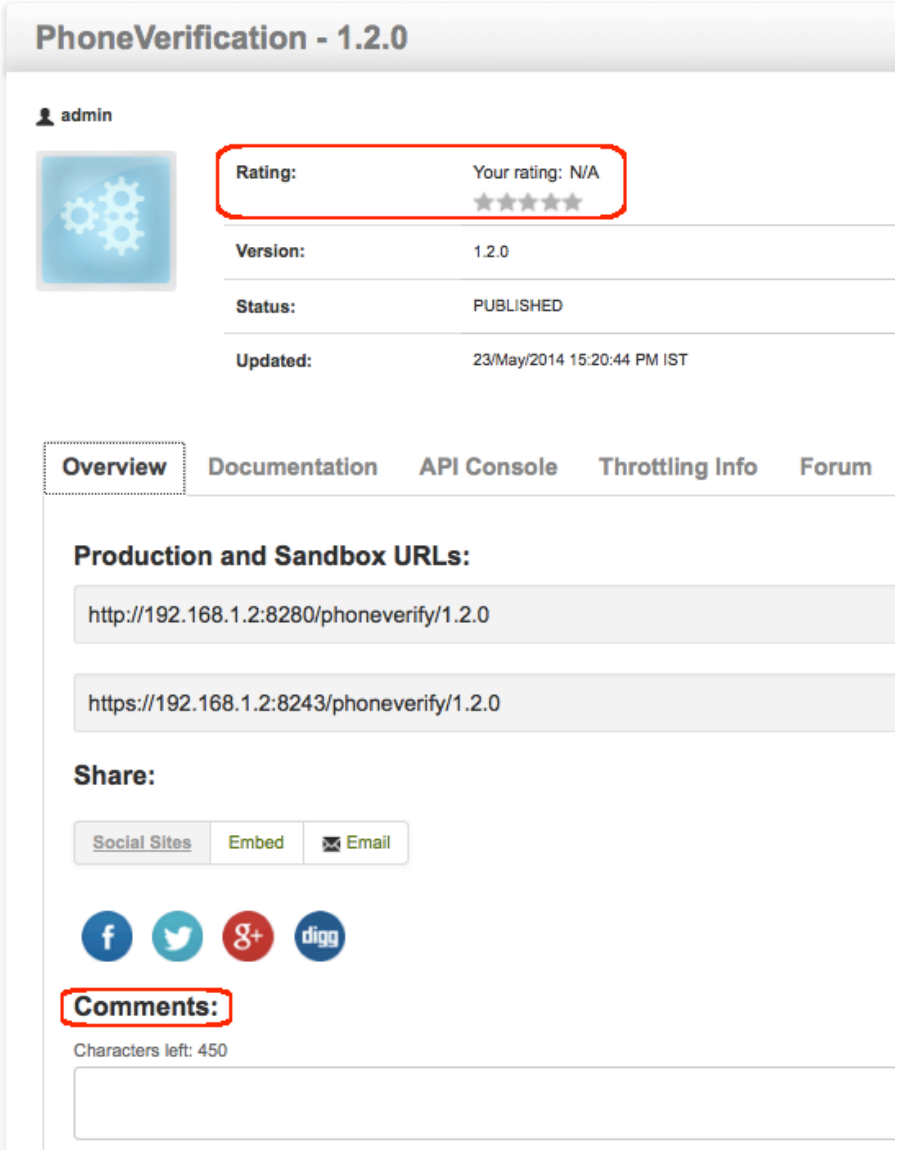
- Rating and commenting
- Sharing on social media / E-mail
- Embedding an API widget
- Participating in the forum

Rating and commenting

Consumers can rate APIs per version and comment on them. Potential API subscribers can use comments and rates as guidelines on the quality and usefulness of an API. Commenting and rating help create a community around a particular API. Comments appear sorted by the time it was entered, alongside the author's name. Commenting is

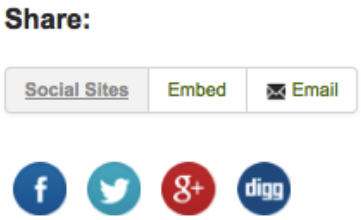
similar to a forum for subscribers, who can discuss common issues/features pertaining to a given API version.

Shown below is how comments/rates of a published API is visible to subscribers through the API Store. To rate an users can sign up to the API StoreAPI, and click on any available API.



Sharing on social media / E-mail

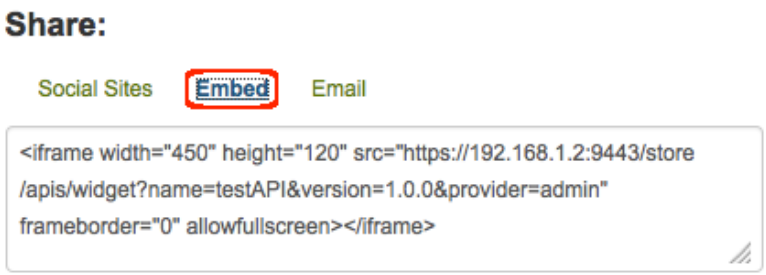
You can share APIs on Facebook, Twitter, Google+, digg etc. or e-mails. This allows developers to share their views on selected APIs through supported social networking sites.



Embedding an API widget

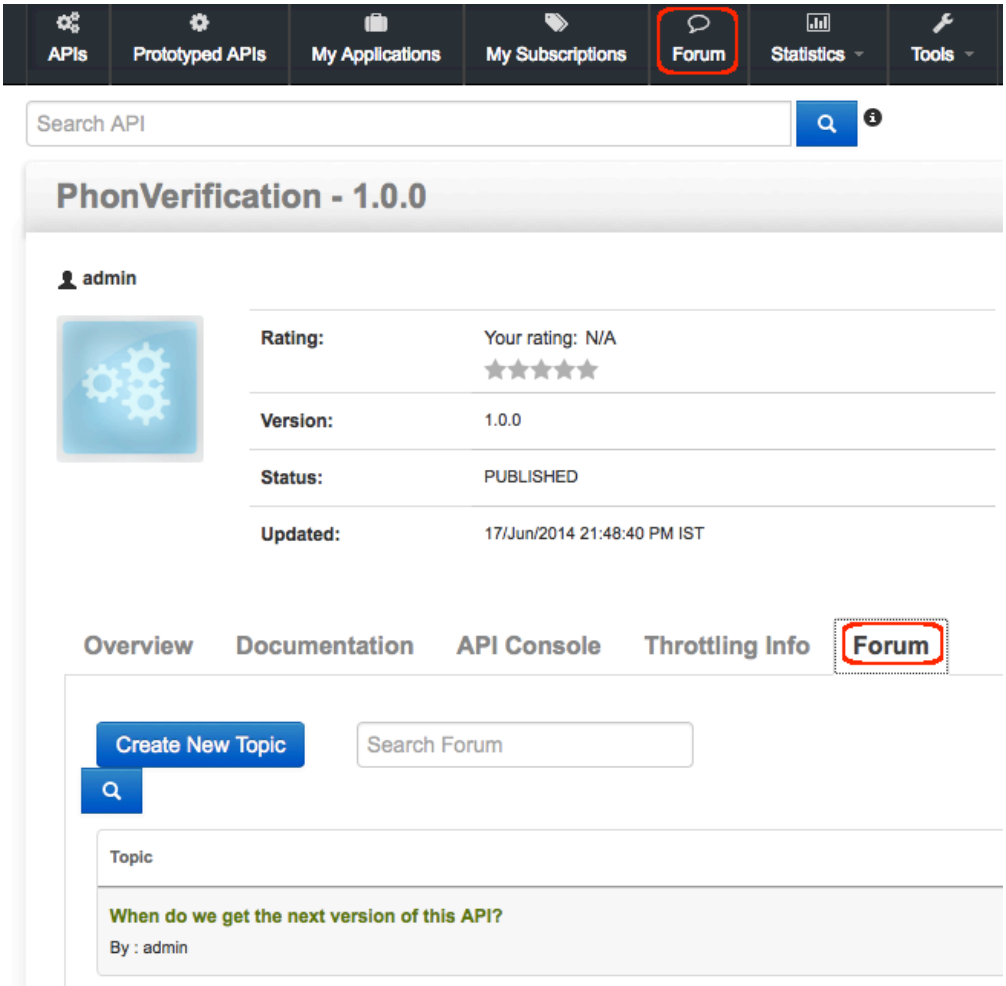
You can generate an embeddable version of API details in HTML and share this HTML widget, which points to an

API, in Web pages. This is similar to how Youtube videos can be embedded in a Web page.



Participating in the forum

Use the **Forum** tab or menu to go to the forum, initiate conversations and communicate with other users subscribed to the API Store:



Customizing the API Store

There are several ways in which you can customize the look and feel, features and functionality of the API Store as discussed below:

- Changing the theme
- Changing language settings
- Single login for all apps
- Categorizing APIs

Changing the theme

You can change the look and feel of your API Store by applying new themes or customizing the default theme's logo, footer notes, `About` page etc.

The default API Store theme is inside `<AM_HOME>/repository/deployment/server/jaggeryapps/store/site/themes` folder. You can override it by adding your customizations inside `/repository/deployment/server/jaggeryapps/store/site/themes/fancy/subthemes` folder. Follow this tutorial in WSO2 library for instructions to change the theme: http://wso2.org/library/articles/2012/06/api-store-themes.

If you are using a Cloud-based API Manager setup (e.g., WSO2 API Cloud) and do not have access to the distribution directory, you can change the theme by logging into the **WSO2 Workflow Admin** Web application (`https://<Server Host>:9443/workflow-admin`) as a tenant admin. Bundle the new theme (CSS, images etc.) into a zip archive and upload it through the Workflow Admin Web app.

Changing language settings

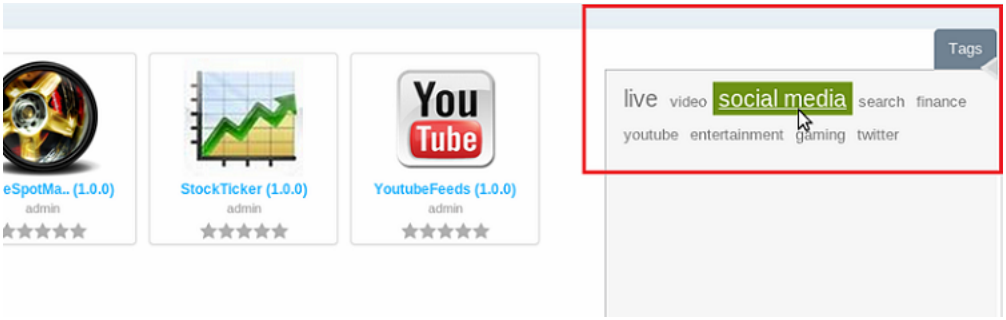
You can change the language of the API Store Web interface to your local language. For configuration steps, see Adding Internationalization and Localization.

Single login for all apps

You can configure single sign-on (SSO) in API Manager so that users who are subscribed to one application can log in to multiple other applications that they are authorized to access, using the same credentials. They do not have to repeatedly authenticate themselves. For configuration steps, see Configuring Single Sign-on with SAML 2.0.

Categorizing APIs

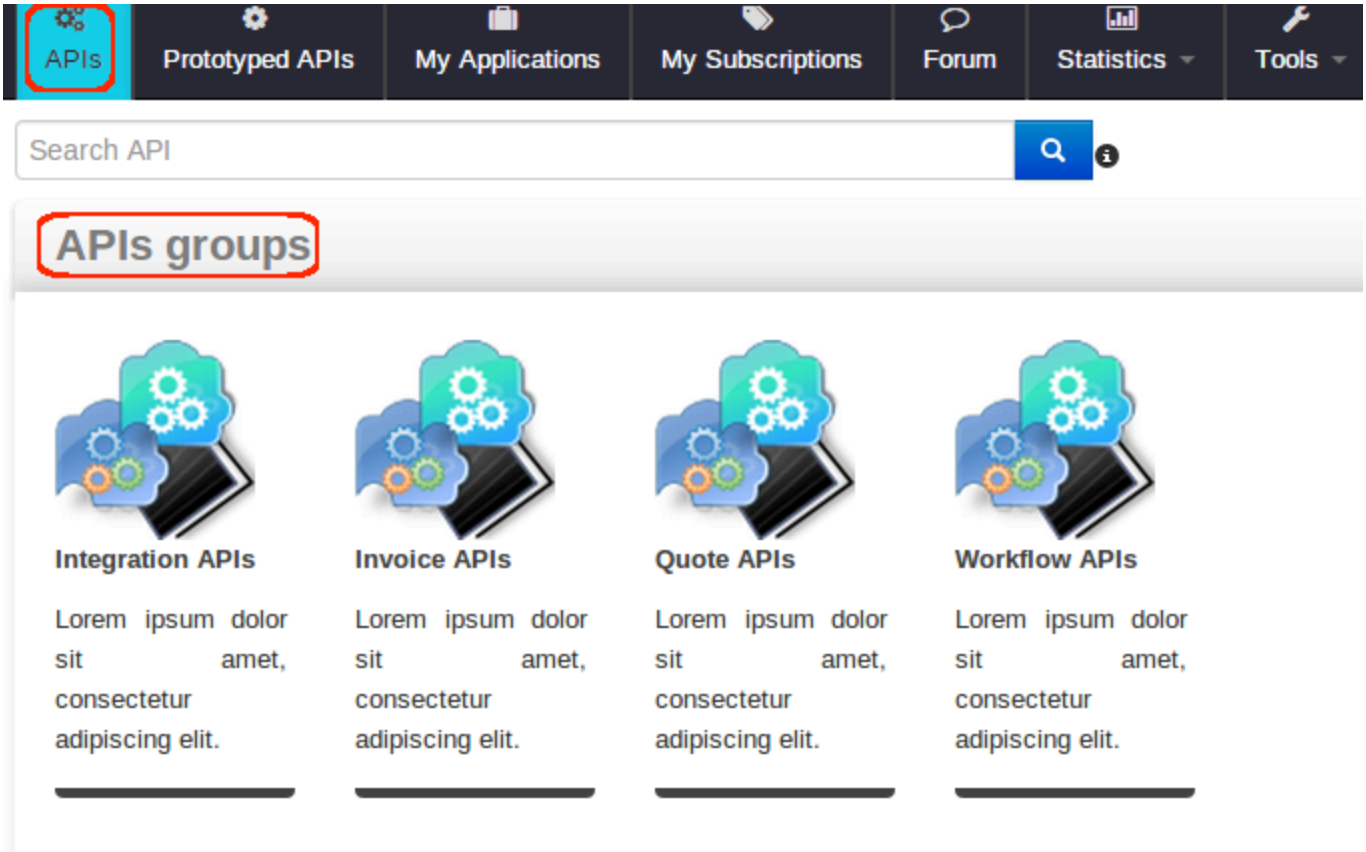
API providers add tags to APIs when designing them using the API Publisher. Tags allow API providers to categorise APIs that have similar attributes. Once a tagged API gets published to the API Store, its tags appear as clickable links to the API consumers, who can use them to quickly jump to a category of interest.



If you want to see the APIs grouped according to different topics in the API Store, do the following:

1. Go to `<APIM_HOME>/repository/deployment/server/jaggeryapps/store/site/conf` directory, open the `site.json` file and set the `tagWiseMode` attribute as true.
2. Go to the API Publisher and add tags with the suffix "-group" to APIs (e.g., Workflow APIs-group, Integration APIs-group, Quote APIs-group.)
3. Restart the server.

After you publish the APIs, you see the APIs listed under their groups. You can click on a group to check what the APIs are inside it.



Monitoring, Statistics and Billing

The following topics describe how to monitor API invocations and how to collect and summarize statistics in order to monetize API usage.

- Publishing API Runtime Statistics
- Integrating with Google Analytics
- Monetization of API Usage
- Viewing API Statistics

Publishing API Runtime Statistics

You can set up **WSO2 Business Activity Monitor** (**version 2.4.1** used here) to collect and analyze runtime statistics from the API Manager. To publish data from the API Manager to BAM, the Thrift protocol is used. Information processed in BAM is stored in a database from which the API Publisher retrieves information before displaying in the corresponding UI screens.

By default, `org.wso2.carbon.apimgt.usage.publisher.APIMgtUsageDataPublisher` is configured to collect data events from WSO2 BAM. If you use a system other than WSO2 BAM to collect and analyze runtime statistics, you write a new data publishing agent by extending `APIMgtUsageDataPublisher` . Find the API templates inside `<APIM_HOME>/repository/resources/api_templates`. When writing a new data publishing agent, make sure the data publishing logic you implement has a minimal impact to API invocation.

> The data source and database names used in this guide are just examples. They may vary depending on your configurations.

- Prerequisites
- Configuring WSO2 API Manager
- Configuring WSO2 BAM
- Troubleshooting common issues
- Changing the statistics database

Prerequisites

- JDK 1.6.* or 1.7

  > If you install JDK in Program Files in the Windows environment, avoid the space by using PROGRA~1 when specifying environment variables for JAVA_HOME and PATH. Else, the server throws an exception.

- Cygwin (http://www.cygwin.com) : Required **only if you are using Windows**. WSO2 BAM analytics framework depends on Apache Hadoop, which requires Cygwin in order to run on Windows. Install at least the basic net (OpenSSH,tcp_wrapper packages) and security related Cygwin packages. After Cygwin installation, update the PATH variable with `C:/cygwin/bin` and restart BAM.

Configuring WSO2 API Manager

1. Do the following changes in `<APIM_HOME>/repository/conf/api-manager.xml` file:
   - Enable API usage tracking by setting the `<APIUsageTracking>` element to true.
   - Because you will apply an offset to the default BAM port later in this guide, you need to apply the same offset to the default Thrift port. To do that, change the port value to 7614 in the `<ThriftPort>` eleme nt of this file. The API Manager will then push the data to BAM through port 7614, using the Thrift protocol.
   - Uncomment the `<DataSourceName>` element. It sets the datasource used to get statistics from BAM
   - Set `<BAMServerURL>` to `tcp://<BAM host IP>:7614/` where `<BAM host IP>` is the machine IP address. Do not use localhost unless you're in a disconnected mode.

     > `<BAMServerURL>` refers to the endpoint to which events will be published from the API Gateway. This endpoint is also known as the event receiver. You can define multiple event receiver groups, each with one or more receivers. A receiver group is defined within curly braces and receiver URLs are delimited by commas.
     >
     > For example, `<BAMServerURL>{tcp://localhost:7612/,tcp://localhost:7613/},{tcp://localhost:7712/,tcp://localhost:7713/}</BAMServerURL>`. This example has two receiver groups defined with two receivers in each group. When a request passes through the API Gateway, an event will be published to one selected receiver in each group.

```
<APIUsageTracking>
    <!-- Enable/Disable the API usage tracker. -->
    <Enabled>true</Enabled>

<PublisherClass>org.wso2.carbon.apimgt.usage.publisher.APIMgtUsageDataBridgeDataP
ublisher</PublisherClass>
    <ThriftPort>7614</ThriftPort>
    <BAMServerURL>tcp://<BAM host IP>:7614/</BAMServerURL>
    <BAMUsername>admin</BAMUsername>
    <BAMPassword>admin</BAMPassword>
    <!-- JNDI name of the data source to be used for getting BAM statistics. This
data source should
        be defined in the master-datasources.xml file in conf/datasources
directory. -->
    <DataSourceName>jdbc/WSO2AM_STATS_DB</DataSourceName>
</APIUsageTracking>
```

2. Specify the datasource definition under the `<datasource>` element in the `<APIM_HOME>/repository/co`

`nf/datasources/master-datasources.xml` file. The tables are created automatically when the Hive script runs. You just need to create the schema. The example below connects to a MySQL instance:

> The **WSO2AM_STATS_DB** database is not available in `<BAM_HOME>/repository/database` dire ctory at this point. It is created only after BAM starts up.

```
<datasource>
    <name>WSO2AM_STATS_DB</name>
    <description>The datasource used for getting statistics to API
Manager</description>
    <jndiConfig>
        <name>jdbc/WSO2AM_STATS_DB</name>
    </jndiConfig>
    <definition type="RDBMS">
        <configuration>

<url>jdbc:mysql://localhost:3306/stats_db?autoReconnect=true&amp;</url>
            <username>db_username</username>
            <password>db_password</password>
            <driverClassName>com.mysql.jdbc.Driver</driverClassName>
            <maxActive>50</maxActive>
            <maxWait>60000</maxWait>
            <testOnBorrow>true</testOnBorrow>
            <validationQuery>SELECT 1</validationQuery>
            <validationInterval>30000</validationInterval>
         </configuration>
    </definition>
 </datasource>
```

3. Save the database driver JAR inside both `<AM_HOME>/repository/components/lib` and `<BAM_HOME>/repository/components/lib` folders.

Next, prepare BAM to collect and analyze statistics from the API Manager.

Configuring WSO2 BAM

1. Download WSO2 BAM 2.4.1 from location: http://wso2.com/products/business-activity-monitor.
2. Apply an offset of 3 to the default BAM port by editing the `<BAM_HOME>/repository/conf/carbon.xml` f ile. This step is done when you run both products on the same server.

   ```
   <Offset>3</Offset>
   ```

   This increments all ports used by the server by 3, which means the BAM server will run on port 9446. Port offset is used to increment the default port by a given value. It avoids possible port conflicts when multiple WSO2 products run in same host.
3. Do the following changes in `<BAM_HOME>/repository/conf/datasources/bam_datasources.xml` fil e:
   - Copy/paste `WSO2AM_STATS_DB` definition from API Manager's `master-datasources.xml` file. You edited it in step 2. `WSO2AM_STATS_DB` is used to fetch analytical data from the database.
   - Replace the port of `WSO2BAM_CASSANDRA_DATASOURCE` in URL (`jdbc:cassandra://localhos t : 9 1 6 3 / E V E N T _ K S )` .
     Note that localhost is used here; not the machine IP. Cassandra is bound by default on localhost, unless you change the data-bridge/data-bridge-config.xml file. Also, if you are running BAM on a different server, the port will be different.

> Do not edit the `WSO2BAM_UTIL_DATASOURCE` with an offset value as it is using the offset.

> Manually updating the port of `WSO2BAM_CASSANDRA_DATASOURCE` is not needed if you are using **WSO2 BAM 2.5.0**.

4. Copy the file `<APIM_HOME>/statistics/API_Manager_Analytics.tbox` to directory `<BAM_HOME>/repository/deployment/server/bam-toolbox`.
   If this folder is not in the BAM installation directory by default, create it. The toolbox describes the information collected, how to analyze the data, as well as the location of the database where the analyzed data is stored.
5. Open `<BAM_HOME>/repository/conf/etc/hector-config.xml` file and change the port to localhost:9163. You must add the other nodes too when configuring a clustered setup.

```
<Nodes>localhost:9163</Nodes>
```

> Step 5 is not needed if you are using **WSO2 BAM 2.5.0**.

6. Restart BAM server by running `<BAM_HOME>/bin/wso2server.[sh/bat]`.
7. **Optional**: If you want to host the BAM server on a different machine or change the running port, you must edit the `<APIUsageTracking>` node in `<APIM_HOME>/repository/conf/api-manager.xml` file as follows:

```
<!--API usage tracker configuration used by the BAM data publisher in API
gateway.-->
    <APIUsageTracking>
        <!-- Enable/Disable the API usage tracker.-->
        <Enabled>true</Enabled>

        <!-- API Usage Data Publisher.-->

<PublisherClass>org.wso2.carbon.apimgt.usage.publisher.APIMgtUsageDataBridgeDataP
ublisher</PublisherClass>

        <!--Thrift port of the remote BAM server.-->
        <ThriftPort>7612</ThriftPort>

        <!-- Server URL of the remote BAM server used to collect statistics. Must
be specified in protocol://hostname:port/ format.-->
        <BAMServerURL>tcp://localhost:7614</BAMServerURL>

        <!--Administrator username to login to the remote BAM server.-->
        <BAMUsername>admin</BAMUsername>

        <!--Administrator password to login to the remote BAM server.-->
        <BAMPassword>admin</BAMPassword>

        <!--JNDI name of the data source to be used for getting BAM
statistics.This data source should be defined in the master
            datasources.xml file in conf/datasources directory.-->
        <DataSourceName>jdbc/WSO2AM_STATS_DB</DataSourceName>
    </APIUsageTracking>
```

Troubleshooting common issues

Given below is how to do troubleshoot some common issues users come across:

1. Do you get an exception as **unable to connect to server** Cassandra?
   Check if you changed the Cassandra port according to the port offset applied to the default BAM port. See Step 3 under configuring BAM section.
2. Do you get a **connection refused exception** on the BAM console?
   This happens when you execute Hive scripts prior to changing the default port. Add the following line at the beginning of the Hive scripts and rerun: `drop table <hive_cassandra_table_name>;` You can find the Hive scripts deployed with the toolbox file, which is inside `<BAM_HOME>/repository/deployment/server/bam-toolbox` folder. For information, see Editing an Analytic Script in WSO2 BAM documentation.

Changing the statistics database

To use a different database than the default H2 for statistical publishing, you must change the properties of the datasource element, and additionally delete some metadata tables created by previous executions of the Hive script, if there are any.

To delete the metadata tables,

1. Log in to BAM management console and select **Add** in **Analytics** menu.
2. Go to the Script Editor in the window that opens.
3. Execute the following script.

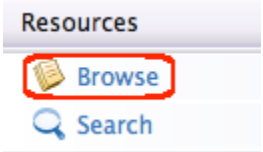
```
drop TABLE APIRequestData;
drop TABLE APIRequestSummaryData;
drop TABLE APIVersionUsageSummaryData;
drop TABLE APIResourcePathUsageSummaryData;
drop TABLE APIResponseData;
drop TABLE APIResponseSummaryData;
drop TABLE APIFaultData;
drop TABLE APIFaultSummaryData;
drop TABLE APIDestinationData;
drop TABLE APIDestinationDataSummaryData;
```

After configuring WSO2 BAM to render and produce statistics of APIs hosted and managed in the API Manager, you can view them through various statistical dashboards in the API Publisher, depending on your permission levels. For information, refer to section Viewing API Statistics.

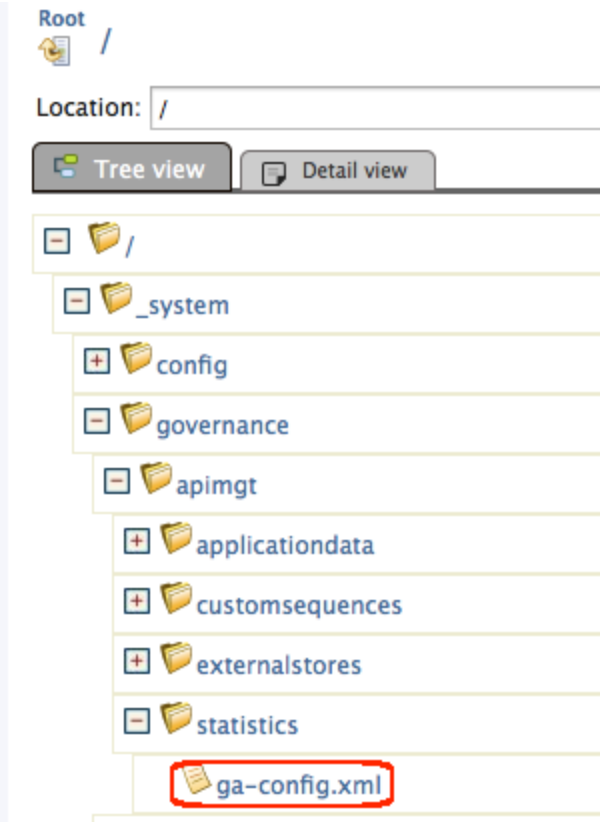
u can configure the API Manager to track runtime statistics of API invocations through Google Analytics (http://www.google.com/analytics). Google Analytics is a service that allows you to track visits to a website and generate detailed statistics on them.

This guide explains how to setup API Manager in order to feed runtime statistics to Google analytics for summarization and display.

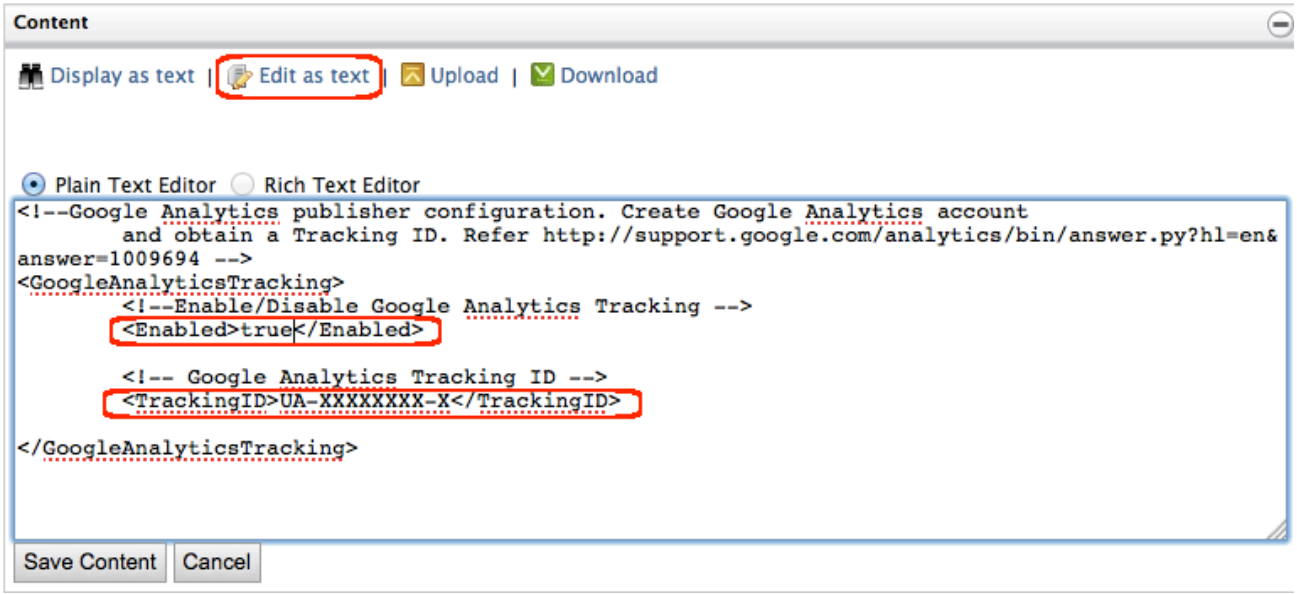
1. Setup a Google Analytics account if not subscribed already and receive a Tracking ID, which is of the format "UA-XXXXXXXX-X". A Tracking ID is issued at the time an account is created with Google Analytics.
2. Log in to the API Manager management console (`https://localhost:9443/carbon`) using admin/admin credentials and go to **Main -> Resources -> Browse** menu.



3. Navigate to /_system/governance/apimgt/statistics/ga-config.xml file.



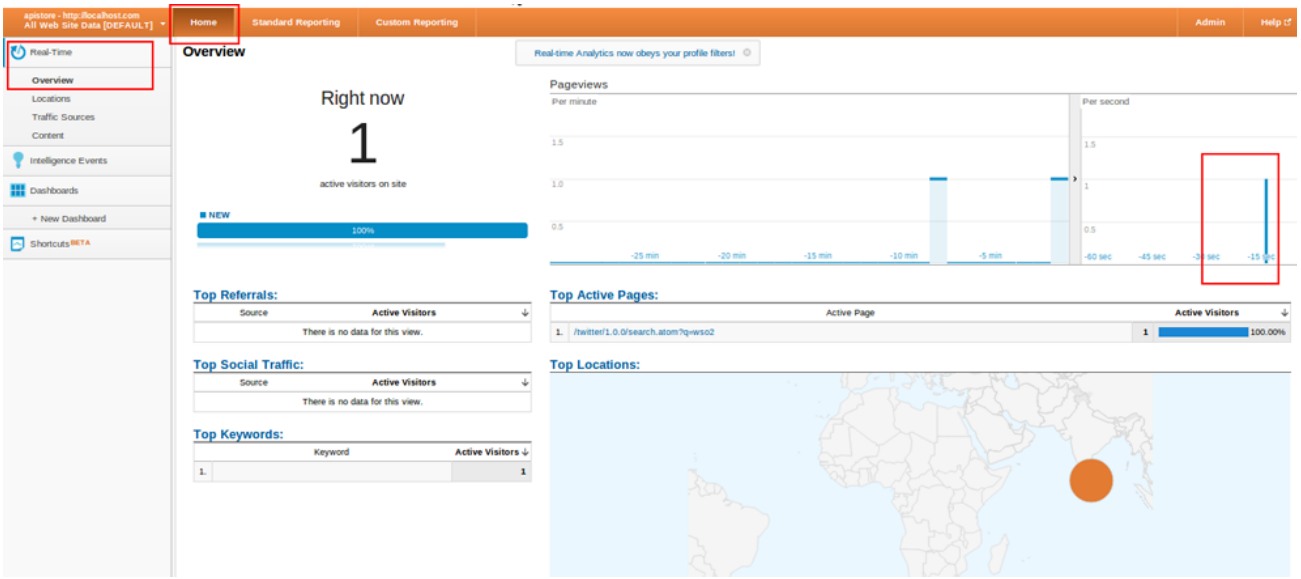
4. Change the <Enabled> element to `true`, set your tracking ID in <TrackingID> element and **Save**.



5. API Manager is now integrated with Google Analytics. A user who has subscribed to a published API through the API Store should see an icon as `Real-Time` after logging into their Google Analytics account. Click on this icon and select **Overview**.
6. Invoke the above API using the embedded WSO2 REST Client (or any third-part rest client such as cURL).

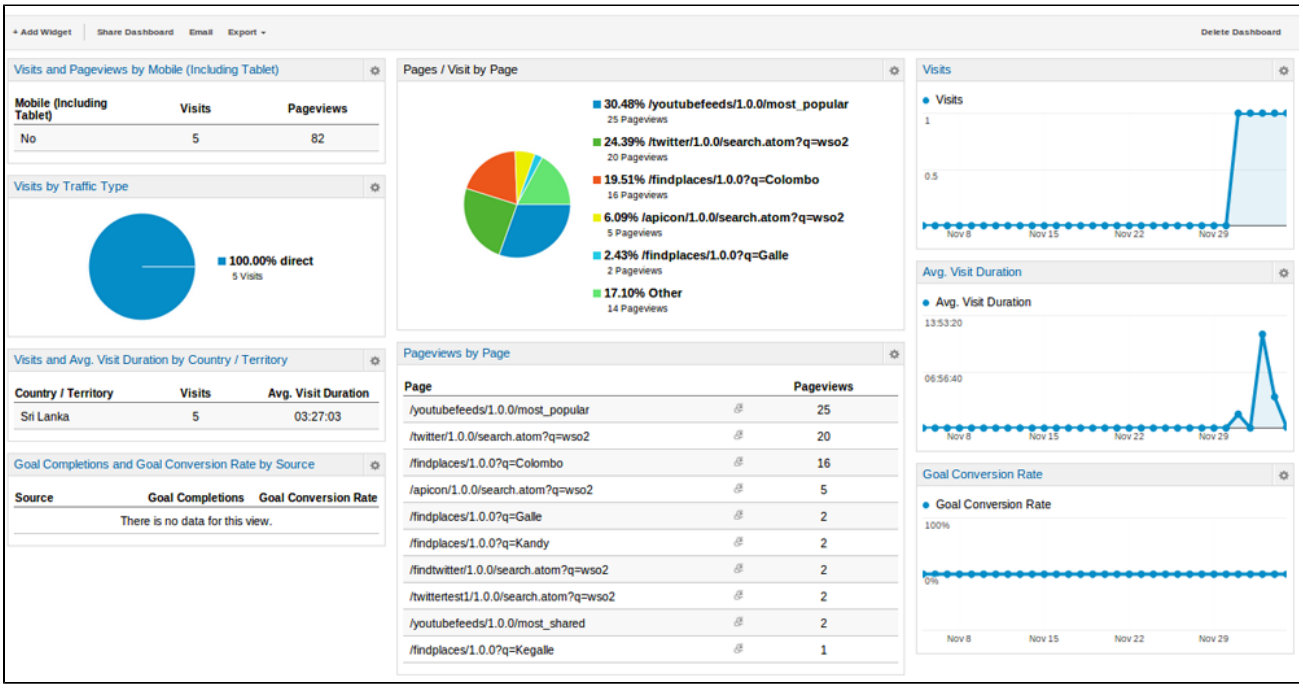
Real-time statistics

7. This is one invocation of the API. Accordingly, Google Analytics graphs and statistics will be displayed at runtime. This example displays the **PageViews** per second graph and 1 user as active.



Reporting statistics

Google analytics reporting statistics take more than 24 hours from the time of invocation to populate. Shown below is a sample Dashboard with populated statistics.



There are widgets with statistics related to Audience, Traffic, Page Content, Visit Duration etc. You can add any widget of your preference to dashboard.

Monetization of API Usage

You can set up WSO2 Business Activity Monitor (BAM) to collect and summarize runtime statistics from the WSO2 API Manager and generate bills for API consumers on usage. See a sample with full configuration steps in Generating Billing Data.

Viewing API Statistics

API statistics are provided in both API Publisher and API Store Web applications. Apart from the number of subscriptions per API, all other statistical dashboards require that an instance of WSO2 Business Activity Monitor (version 2.3.0 or above) is installed. For instructions to set up BAM, see Publishing API Runtime Statistics. Once BAM is set up, follow the instructions below to view statistics through the API Publisher.

First, trigger some activities via the API gateway as explained in section Browser-Based REST Clients and wait a few seconds.

> Even if you haven't triggered real statistics using WSO2 BAM, you still see sample graphs and charts when you access the statistical dashboards in the API Manager. They are provided for reference only and are not based on real runtime statistics of your server.

The sections below explain how to access the statistical dashboards:

- API Publisher statistics
- API Store statistics

API Publisher statistics

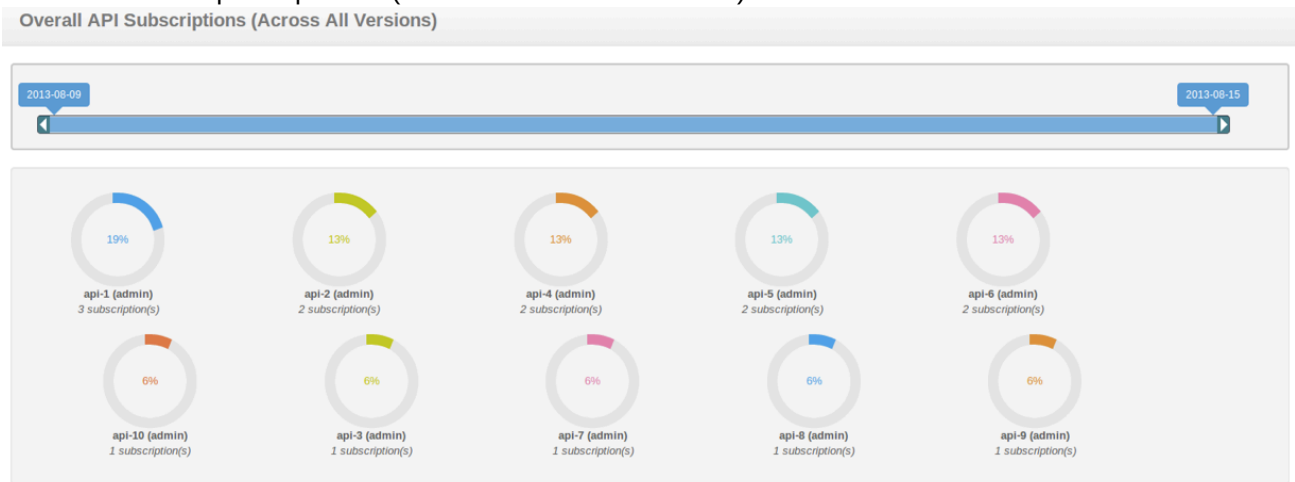
Log in to the API Publisher (https://localhost:9443/publisher) as a user with `creator` or `publisher` role assigned. Depending on the role, the statistical menu items change as described below:

- If you are logged in as a `publisher`, the **All Statistics** menu is visible in the left panel of the API Publisher Web interface.
- If you are logged in as a `creator`, in addition to the **All Statistics** menu, you also see **Statistics** men u in the left panel of the API Publisher Web interface. The latter shows stats specific to the APIs created by you.
- Both `creator` and `publisher` roles can view API-level usage and subscription statistics by clicking on a selected API and referring to its **Versions** and **Users** tabs.

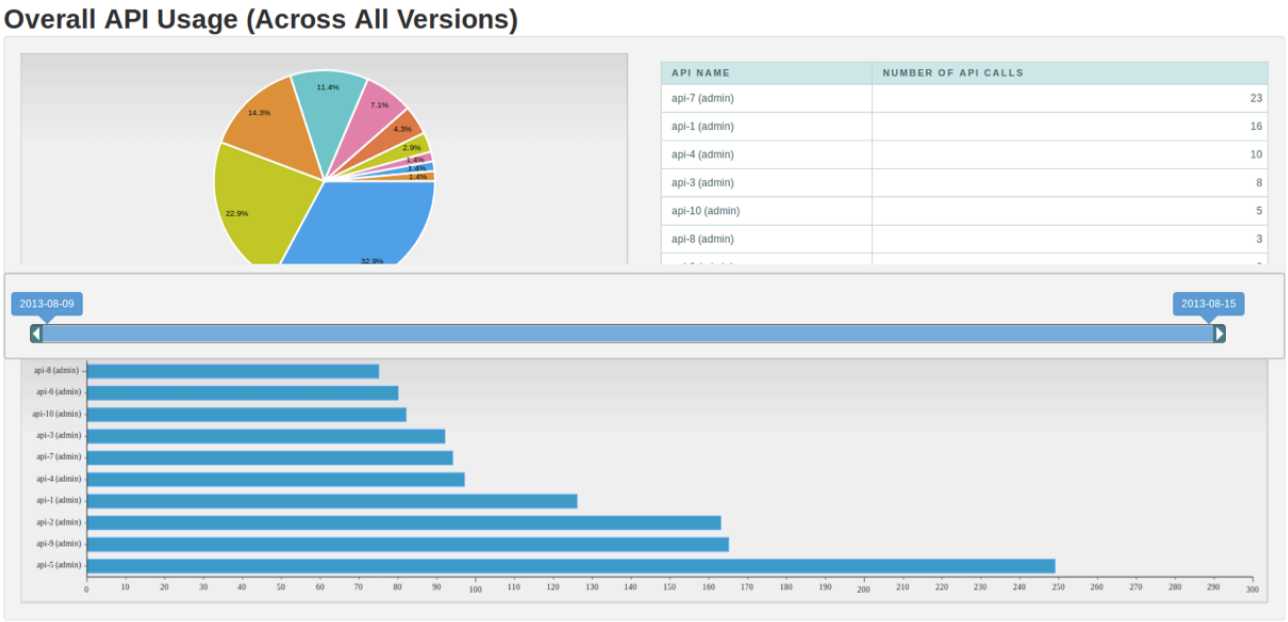
Several examples of usage and performance statistics are given below:

> See Creating an API on how to see a destination-based usage tracking graph of your APIs.

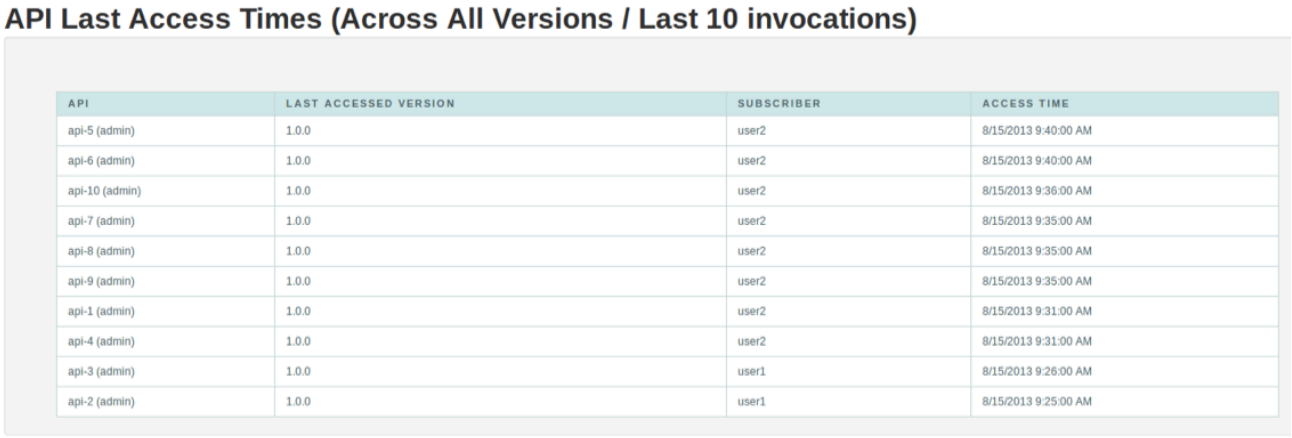
- Number of subscriptions per API (across all versions of an API)



- Number of API calls being made per API (across all versions of an API)

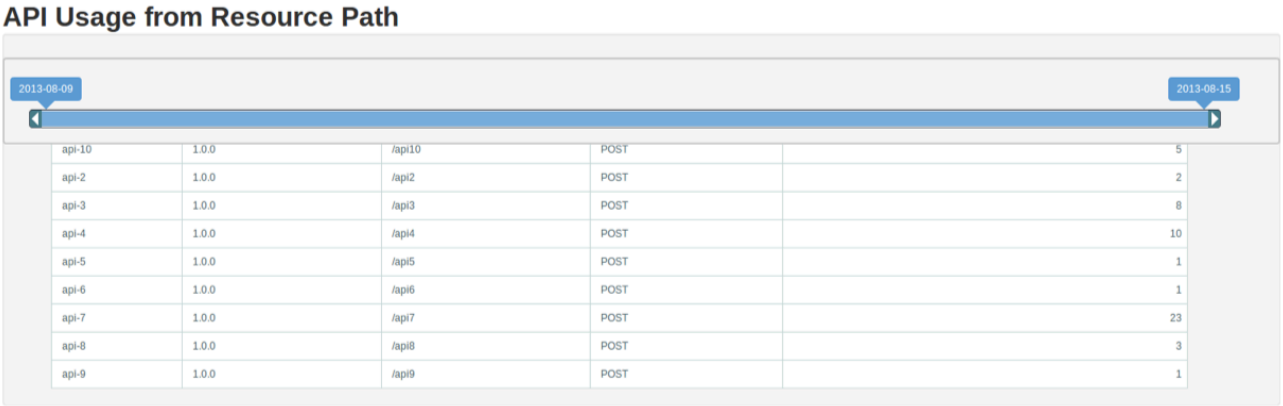


- The subscribers who did the last 10 API invocations and the APIs/versions they invoked



API	LAST ACCESSED VERSION	SUBSCRIBER	ACCESS TIME
api-5 (admin)	1.0.0	user2	8/15/2013 9:40:00 AM
api-6 (admin)	1.0.0	user2	8/15/2013 9:40:00 AM
api-10 (admin)	1.0.0	user2	8/15/2013 9:36:00 AM
api-7 (admin)	1.0.0	user2	8/15/2013 9:35:00 AM
api-8 (admin)	1.0.0	user2	8/15/2013 9:35:00 AM
api-9 (admin)	1.0.0	user2	8/15/2013 9:35:00 AM
api-1 (admin)	1.0.0	user2	8/15/2013 9:31:00 AM
api-4 (admin)	1.0.0	user2	8/15/2013 9:31:00 AM
api-3 (admin)	1.0.0	user1	8/15/2013 9:26:00 AM
api-2 (admin)	1.0.0	user1	8/15/2013 9:25:00 AM

- Usage of an API and from which resource path (per API version)



api-10	1.0.0	/api10	POST	5
api-2	1.0.0	/api2	POST	2
api-3	1.0.0	/api3	POST	8
api-4	1.0.0	/api4	POST	10
api-5	1.0.0	/api5	POST	1
api-6	1.0.0	/api6	POST	1
api-7	1.0.0	/api7	POST	23
api-8	1.0.0	/api8	POST	3
api-9	1.0.0	/api9	POST	1

- Number of times a user has accessed an API

API Usage By User

API	VERSION	USER	NUMBER OF ACCESS
api-7	1.0.0	user2	23
api-1	1.0.0	user1	13
api-3	1.0.0	user1	8
api-4	1.0.0	user2	7
api-10	1.0.0	user2	5
api-1	1.0.0	user2	3
api-4	1.0.0	user1	3
api-8	1.0.0	user2	3

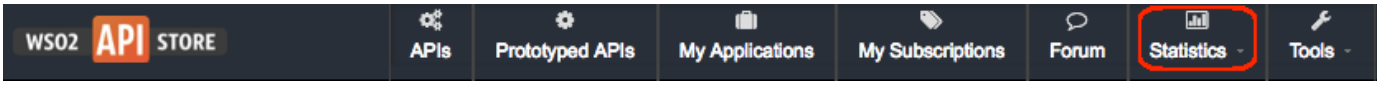
- The number of API invocations that failed to reach the endpoint per API per user
  In a faulty API invocation, the message is mediated though the `fault` sequence. By default, the API Manager considers an API invocation to be faulty when the backend service is unavailable.

Faulty Invocations

api-8	1.0.0	user2	3

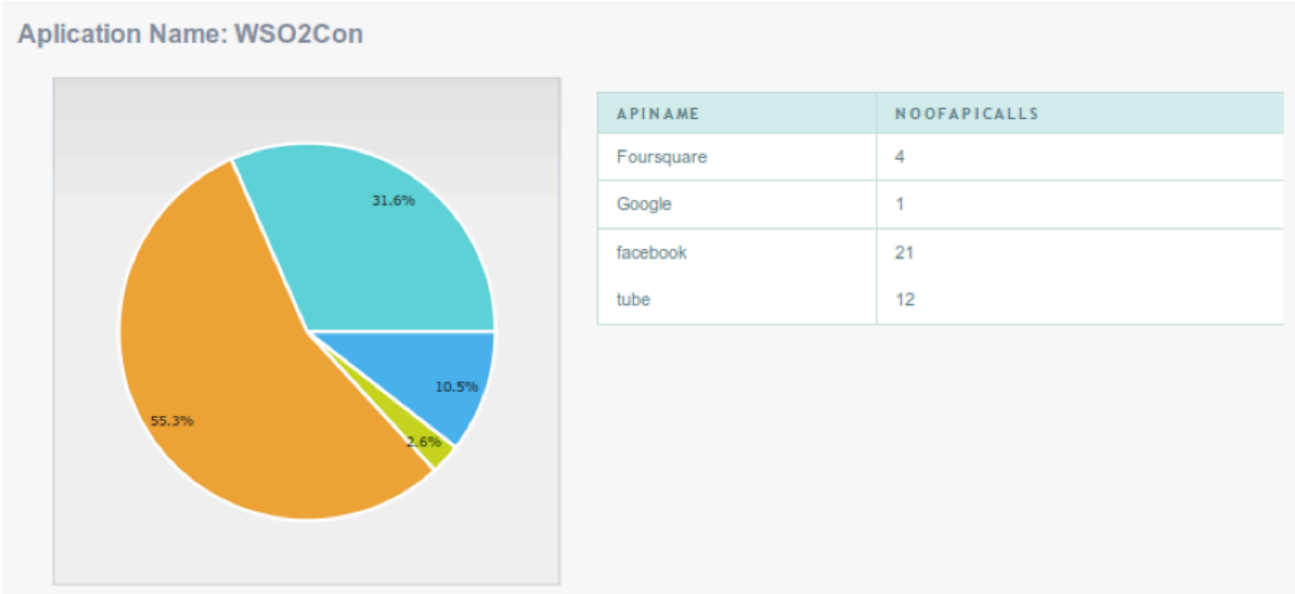
API Store statistics

Log in to the API Store (https://localhost:9443/store). You can self subscribe to the store. Next, click the Statistics menu.



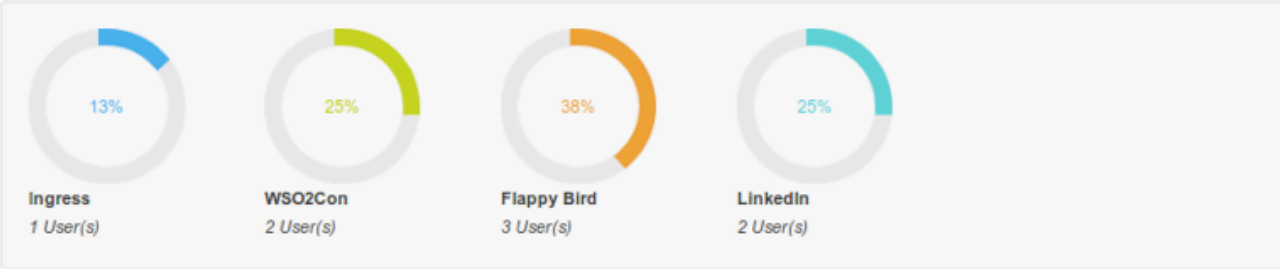
Several examples of usage and performance statistics are given below:

- API usage per application

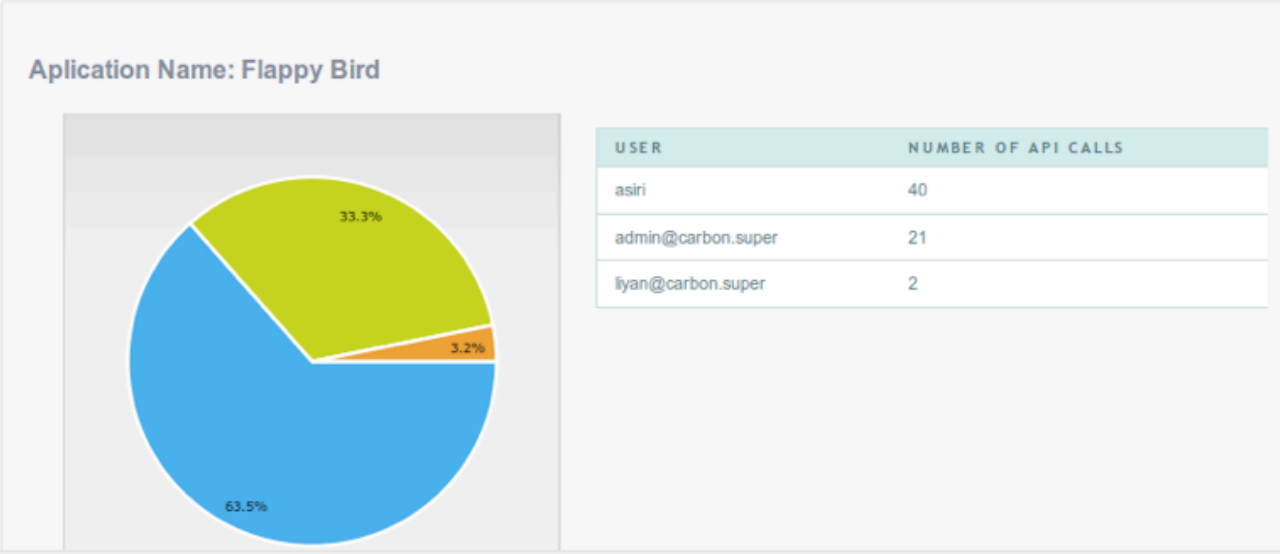


- Users who make the most API invocations, per application

Registered Users For Applications

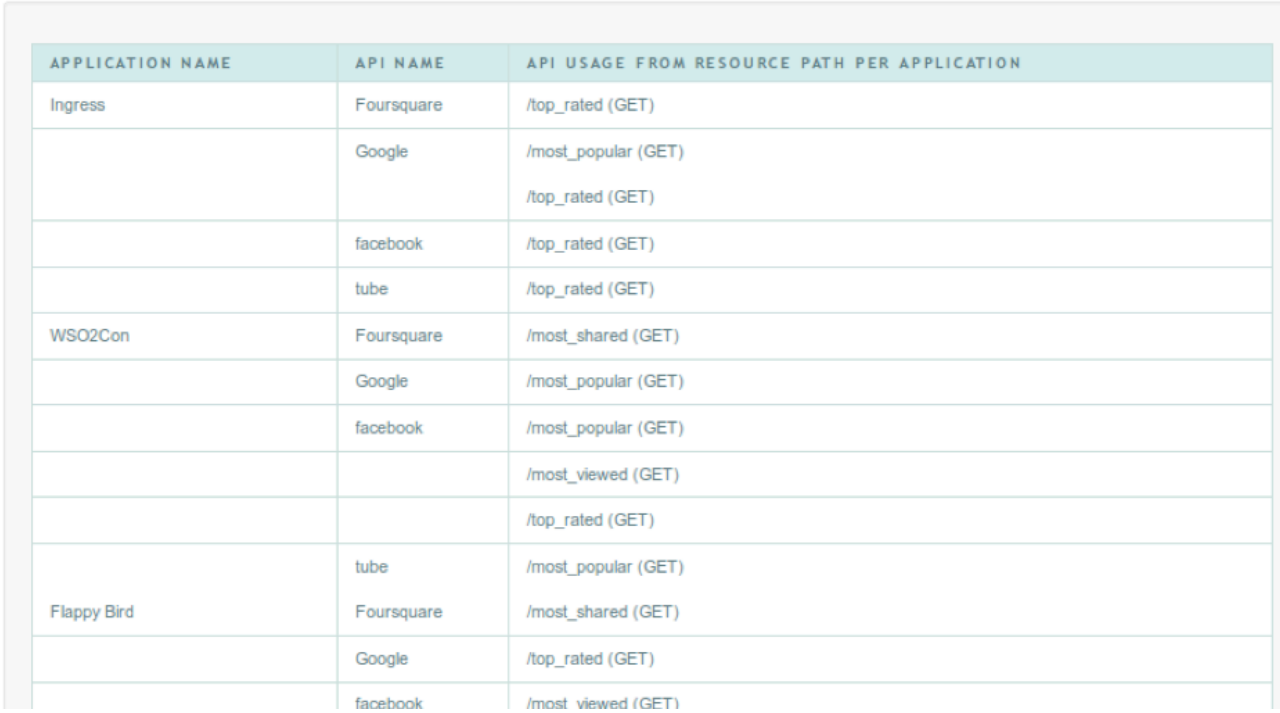


Top Users For Applications



- API usage from resource path, per application

API Usage from Resource Path



APPLICATION NAME	API NAME	API USAGE FROM RESOURCE PATH PER APPLICATION
Ingress	Foursquare	/top_rated (GET)
	Google	/most_popular (GET) /top_rated (GET)
	facebook	/top_rated (GET)
	tube	/top_rated (GET)
WSO2Con	Foursquare	/most_shared (GET)
	Google	/most_popular (GET)
	facebook	/most_popular (GET)
		/most_viewed (GET)
		/top_rated (GET)
Flappy Bird	tube Foursquare	/most_popular (GET) /most_shared (GET)
	Google	/top_rated (GET)
	facebook	/most_viewed (GET)

- Number of faulty API invocations, per application
  In a faulty API invocation, the message is mediated though the `fault` sequence. By default, the API

Manager considers an API invocation to be faulty when the backend service is unavailable.

Faulty Invocations per Application

APPLICATION NAME	API NAME	FAULTY API INVOCATION COUNT
Ingress	Foursquare	1
	Google	6
WSO2Con	Foursquare	4
	Google	1
Flappy Bird	Foursquare	8
	Google	6

Extending API Manager

The following topics cover different ways in which you can extend the API Manager:

- Editing API Templates
- Implementing an API facade with WSO2 API Manager
- Writing Custom Handlers
- Integrating with WSO2 Governance Registry Services
- Adding Mediation Extensions
- Adding Workflow Extensions
- Transforming API Message Payload
- Customizing the Management Console
- Writing Test Cases

Editing API Templates

Each API in API manager is represented by an XML file. The elements of this XML file and their attributes are defined in `<APIM_HOME>/repository/resources/api_templates/velocity_template.xml` file, which is the default API template that comes with the API Manager. By editing the default template definitions, you can change the synapse configuration of all APIs that are created.

If you are using a distributed API Manager setup (i.e., Publisher, Store, Gateway and Key Manager components are running on separate JVMs), edit the template in the Publisher node.

Implementing an API facade with WSO2 API Manager

WSO2 API Manager shares most of the components of WSO2 ESB. Both products are built on top of the same component-based WSO2 Carbon platform. Therefore, API Manager supports most of the ESB's functionality such as exposing SOAP services as REST-JSON.

Using both the API Manager and WSO2 ESB, you can implement an API facade architecture pattern. WSO2 recommends this architecture if you are performing heavy mediation in your setup. For implementation details of an API facade, see implementing an API facade with WSO2 API management platform. Since the API Manager does not have the ESB's GUI to perform mediation functions, you need to use the XML-based source view for configuration. Alternatively, you can create the necessary mediation sequences using the GUI of the ESB, and copy them from the ESB to the API Manager.

Also see the following usecases in WSO2 ESB documentation for more information on REST to SOAP conversion.

Writing Custom Handlers

This section introduces handlers and using an example, explains how to write a custom handler:

- Introducing Handlers
- Writing a custom handler
- Engaging the custom handler

Introducing Handlers

W You find the default handlers in any API's Synapse definition as shown below.
hen an API is created, a file with its synapse configuration is added to the API Gateway. You can find it in the `<APIM_HOME>/repository/deployment/server/synapse-configs/default/api` folder. It has a set of handlers, each of which is executed on the APIs in the same order they appear in the configuration.

```
<handlers>
   <handler
class="org.wso2.carbon.apimgt.gateway.handlers.security.APIAuthenticationHandler"/>
   <handler
class="org.wso2.carbon.apimgt.gateway.handlers.throttling.APIThrottleHandler">
        <property name="id" value="A"/>
        <property name="policyKey" value="gov:/apimgt/applicationdata/tiers.xml"/>
   </handler>
   <handler class="org.wso2.carbon.apimgt.usage.publisher.APIMgtUsageHandler"/>
   <handler
class="org.wso2.carbon.apimgt.usage.publisher.APIMgtGoogleAnalyticsTrackingHandler"/>
   <handler
class="org.wso2.carbon.apimgt.gateway.handlers.ext.APIManagerExtensionHandler"/>
</handlers>
```

Let's see what each handler does:

- **`APIAuthenticationHandler:`** Validates the OAuth2 bearer token used to invoke the API. It also determines whether the token is of type `Production` or `Sandbox` and sets `MessageContext` variables as appropriate.
- **`APIThrottleHandler:`** Throttles requests based on the throttling policy specified by the `policyKey` property. Throttling is applied both at the application level as well as subscription level.
- **`APIMgtUsageHandler:`** Publishes events to BAM for collection and analysis of statistics. This handler only comes to effect if API usage tracking is enabled. See Publishing API Runtime Statistics for more information.
- **`APIMgtGoogleAnalyticsTrackingHandler:`** Publishes events to Google Analytics. This handler only comes into effect if Google analytics tracking is enabled. See Integrating with Google Analytics for more information.
- **`APIManagerExtensionHandler:`** Triggers extension sequences. By default, the extension handler is listed at last in the handler chain, and therefore is executed last. To configure the API Gateway to execute extension handlers first, uncomment the `<ExtensionHandlerPosition>` section in the `<APIM_HOME>/repository/conf/api-manager.xml` file and provide the value `top`. This is useful when you want to execute your own extensions before our default handlers in situations like doing additional security checks such as signature verification on access tokens before executing the default security handler. See Adding Mediation Extensions.

Writing a custom handler

Let's see how you can write a custom handler and apply it to the API Manager. In this example, we extend the authentication handler. Make sure your custom handler name is not the same as the name of an existing handler.

WSO2 API Manager provides the OAuth2 bearer token as its default authentication mechanism. The source code of the implementation is here. Similarly, you can extend the API Manager to support any custom authentication mechanism by writing your own authentication handler class. This custom handler must extend `org.apache.synapse.rest.AbstractHandler` class and implement the `handleRequest()` and `handleResponse()` methods.

Given below is an example implementation:

```
package org.wso2.carbon.test;

import org.apache.synapse.MessageContext;
import org.apache.synapse.core.axis2.Axis2MessageContext;
import org.apache.synapse.rest.AbstractHandler;

import java.util.Map;

public class CustomAPIAuthenticationHandler extends AbstractHandler {

    public boolean handleRequest(MessageContext messageContext) {
        try {
            if (authenticate(messageContext)) {
                return true;
            }
        } catch (APISecurityException e) {
            e.printStackTrace();
        }
        return false;
    }

    public boolean handleResponse(MessageContext messageContext) {
        return true;
    }

    public boolean authenticate(MessageContext synCtx) throws APISecurityException {
        Map headers = getTransportHeaders(synCtx);
        String authHeader = getAuthorizationHeader(headers);
        if (authHeader.startsWith("userName")) {
            return true;
        }
        return false;
    }

    private String getAuthorizationHeader(Map headers) {
        return (String) headers.get("Authorization");
    }

    private Map getTransportHeaders(MessageContext messageContext) {
        return (Map) ((Axis2MessageContext) messageContext).getAxis2MessageContext().

getProperty(org.apache.axis2.context.MessageContext.TRANSPORT_HEADERS);
    }
}
```

Engaging the custom handler

You can engage a custom handler to all APIs at once or only to selected APIs.

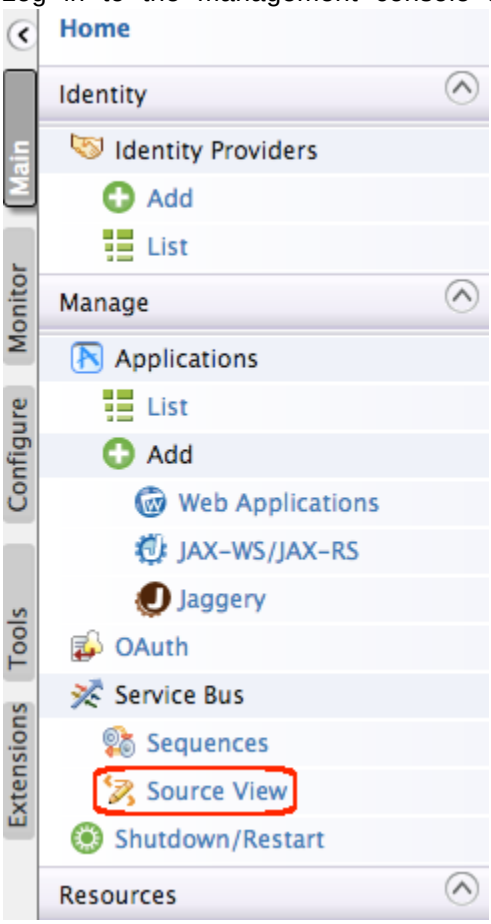
To engage to all APIs, the recommended approach is to add it to the **`<APIM_HOME>/repository/resources/api_templates/velocity_template.xml`** file. For example, the following code segment adds the custom authentication handler that you wrote earlier to the `velocity_template.xml` file while making sure that it skips the default `APIAuthenticationHandler` implementation:

```
<handler
class="org.wso2.carbon.apimgt.custom.authentication.handler.CustomAPIAuthenticationHan
dler" />
       #foreach($handler in $handlers)
          #if(!($handler.className ==
"org.wso2.carbon.apimgt.gateway.handlers.security.APIAuthenticationHandler"))
           <handler xmlns="http://ws.apache.org/ns/synapse"
class="$handler.className">
            #if($handler.hasProperties())
                #set ($map = $handler.getProperties() )
                #foreach($property in $map.entrySet())
                    <property name="$!property.key" value="$!property.value"/>
                #end
            #end
           </handler>
         #end
        #end
</handlers>
```

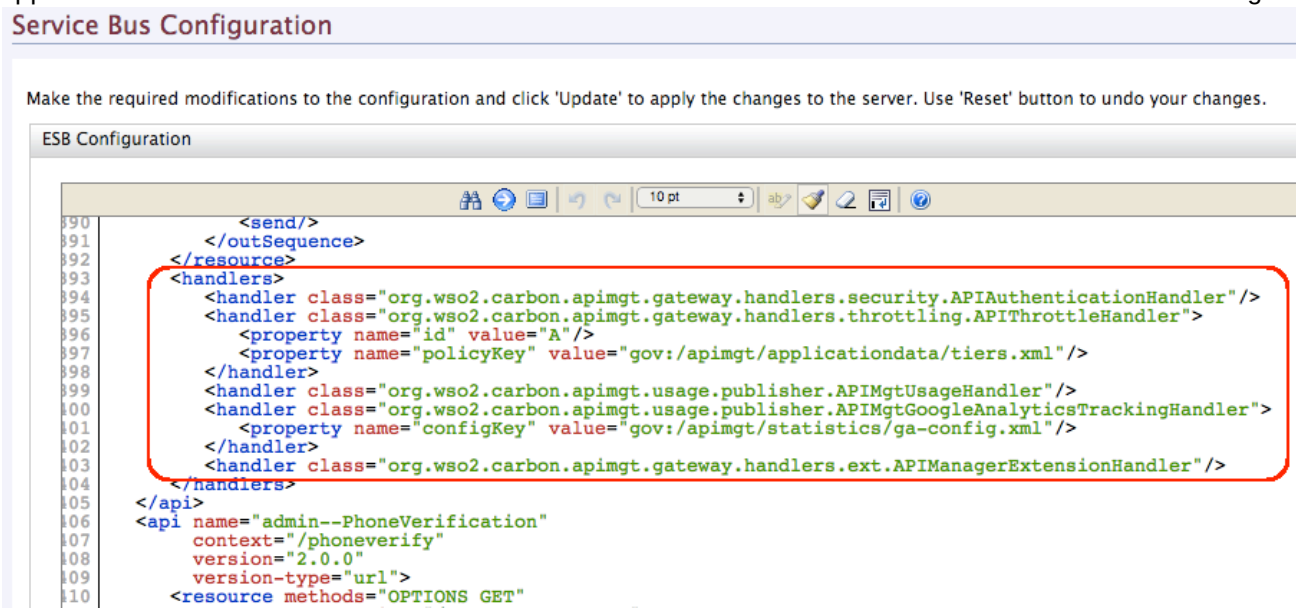
Given below is how to engage handlers to a single API, by editing its source view.

> **Note** that when you engage a handler by editing the API's source view, your changes will be overwritten every time you save the API through the API Publisher.

1. Build the class and copy the JAR file to `<APIM_HOME>/repository/components/lib` folder.
2. Log in to the management console and select **Service Bus > Source View** in the **Main** menu.



3. In the configuration that opens, select an API and navigate to the <Handlers> section. The following line appears as the first handler. This is the current authentication handler used in the API Manager.



4. Replace the above line with the handler that you created. It will engage your custom handler to the API Manager instance. According to this example, it is as follows:

```
<handler
class="org.wso2.carbon.apimgt.gateway.handlers.security.CustomAPIAuthenticationHa
ndler"/>
```

Integrating with WSO2 Governance Registry Services

WSO2 Governance Registry is a registry-repository for storing and managing metadata related to services and other artifacts. Services in the Governance Registry are implemented as configurable governance artifacts (RXT files). Us ually, APIs are created using the API Publisher Web interface. Instead, you can integrate the API Manager with the Governance Registry to directly create APIs in the API Publisher using the services deployed in the Governance Registry.

The steps below explain how to configure the two products to expose services in the Governance Registry as APIs.

- The following steps apply to WSO2 Governance Registry version 4.6.0 or after.
- In WSO2 Governance Registry 4.6.0, we do a simple POST to create APIs in the API Publisher. It does not involve registry mounting.

Follow the steps below to publish services on Governance Registry to the API Manager.

1. Download both WSO2 Governance Registry (G-Reg) and WSO2 API Manager.
2. Provide the API Manager credentials in `<GREG_HOME>/repository/resources/lifecycles/configu rations.xml` file. For example, the following code block defines an `execution` element in `production` st ate. It provides the API Manager's endpoint, username and password as executor parameters.

```
<execution forEvent="Publish"
class="org.wso2.carbon.governance.registry.extensions.executors.apistore.ApiStore
Executor">
   <parameter name="apim.endpoint" value="http://localhost:9763/"/>
   <parameter name="apim.username" value="admin"/>
   <parameter name="apim.password" value="admin"/>
   <parameter name="default.tier" value="Unlimited"/>
   <parameter name="throttlingTier"
value="Unlimited,Unlimited,Unlimited,Unlimited,Unlimited"/>
</execution>
```

Note: If you started the G-Reg server at least once before executing step 2, editing the `configurations.xml` file and restarting the server does not apply the configurations. You need to add the configurations using the G-Reg management console as follows:

a. Log in to the G-Reg Management console and select **Extensions -> Configure -> Lifecycles** menu.
b. Click the Edit link associated with **`ServiceLifeCycle`** .
c. Add the configuration given in step 2 above and **Save**.

3. Run the G-Reg and the API Manager.

   When running more than one WSO2 products on the same server, change the default port of one product to avoid port conflicts. You can do this by changing the `<offset>` value of one product in `<PRODUCT_HOME>/repository/conf/carbon.xml` file. In this example, we set the port offset value of Governance Registry to 1 as follows: `< Offset >1</ Offset>`

   Note: If you offset the default API Manager port, you must also change the default API endpoints and the Thrift port accordingly. See Changing the Default Ports with Offset.

4. Access the API Manager server using the following URL: https://<HostName>:9443/carbon. As you changed the default port of G-Reg, you can access the server using the following URL: https://<HostName>:<9443+offset>/carbon.
5. Log in to the G-Reg management console and create a new service in it and attach the default service lifecycle to it. For instructions on how to add a new service and associate a new lifecycle, see http://docs.wso2.org/governance-registry/Managing+Services in the Governance Registry documentation.
6. Promote the service until it gets to the production state.
7. When it is in the production state, publish it using the **Publish** button. You should get a confirmation message once the API is successfully published.
8. You have now created an API using a service in the Governance Registry. Open the API Publisher to see that this service is successfully created as an API.

Use Secure Vault to secure the API Manager username and Password in a production deployment. See Adding API Manager username and password to secure vault.

Adding API Manager username and password to secure vault

1. Run ciphertool.sh/.bat with -Dconfigure parameter.
2. Add **apim.username** and **apim.password** as aliases to ciper-text.properties.
3. Run cipertool.sh (on Linux) or cipertool.bat (on Windows) and encrypt username and password values.
4. Add the encrypted text to `ciper-text.properties` file after the other ailas and encrypted pairs and restart the server. For example,

```
apim.username=klVWQ32mbNKBxiRp78kK1Et7ZDnLPEsFQTwYjNEzTdpYAISFWJht4cqMjtQ6sXRc7eu
buFxBaGVYP6LBA33XjIc855a+kDiJKXjtGhcCejyHrZoKrHb2PCJ2y0TDWtczEfHHFMhn/0u+AJafU47H
yOgBXZDLcbfGiC5mdJqEoj4=
apim.password=klVWQ32mbNKBxiRp78kK1Et7ZDnLPEsFQTwYjNEzTdpYAISFWJht4cqMjtQ6sXRc7eu
buFxBaGVYP6LBA33XjIc855a+kDiJKXjtGhcCejyHrZoKrHb2PCJ2y0TDWtczEfHHFMhn/0u+AJafU47H
yOgBXZDLcbfGiC5mdJqEoj4=
```

Adding Mediation Extensions

The API Gateway has a default mediation flow that is executed in each API invocation. You can do additional custom mediation for the messages in the API Gateway by extending its mediation flow. An extension is provided as a synapse mediation sequence.

You can design all sequences using a tool like WSO2 Developer Studio, and store the sequence.xml file in the governance registry. For information, see Creating ESB Artifacts in the Developer Studio documentation. The registry collection where sequences are stored is `customsequences` , which is available by default in `apimgt` governance registry location. Given below are the registry paths:

Sequence	Registry path
in	`/_system/governance/apimgt/customsequences/in`
out	`/_system/governance/apimgt/customsequences/out`
fault	`/_system/governance/apimgt/customsequences/fault`

For example, if you have an in sequence file as `testInSequence`, you must save it in `/_system/governance/apimgt/customsequences/in/testInSequence.xml.`

There are two ways to apply mediation extensions to messages:

- Global Extensions : Apply to all APIs
- Per-API Extensions : Apply only to an intended API

The difference between a global extension and a per-API extension is simply in the name given to the sequence that you use to create it.

Creating global extensions

Given below is the naming pattern of a global extension sequence.

`WSO2AM--Ext--<DIRECTION>`

The `<DIRECTION>` can be `In` or `Out`. To change the default fault sequence, you can either modify the default sequence or write a custom fault sequence and engage it to APIs through the API Publisher. When the direction of the sequence is `In`, the extension is triggered on the in-flow (request path). Similarly, when the direction of the sequence is `Out`, the extension is triggered on the out-flow (response path). Shown below is an example synapse configuration of a global extension sequence.

Global Extension Sequence Example

```
<sequence xmlns="http://ws.apache.org/ns/synapse" name="WSO2AM--Ext--In">
    <log level="custom">
        <property name="TRACE" value="Global Mediation Extension"/>
    </log>
</sequence>
```

To test the code, copy it to an XML file (e.g., global_ext.xml) and save the file in the `<APIM_HOME>/repository/deployment/server/synapse-configs/default/sequences` directory. The above sequence prints a log message on the console on every API invocation.

Creating per-API extensions

Given below is the naming pattern of a per-API extension sequence.

`<API_NAME>:v<VERSION>--<DIRECTION>`

Shown below is an example synapse configuration of a per-API extension sequence. It is created for an API named admin--TwitterSearch with version 1.0.0.

API Extension Sequence Example

```
<sequence xmlns="http://ws.apache.org/ns/synapse"
name="admin--TwitterSearch:v1.0.0--In">
    <log level="custom">
        <property name="TRACE" value="API Mediation Extension"/>
    </log>
</sequence>
```

NOTE: The tenant username must be given as **`<username>-AT-<domain>`** in the configuration. For example, if the tenant username is `testuser` and the domain is `wso2.com`, then the name attribute in the above configuration must be `testuser-AT-wso2.com--TwitterSearch:v1.0.0–In`. The `@` sign must be given as AT.

To test the code in super-tenant mode, copy it to an XML file (e.g., `twittersearch_ext.xml`) and save the file in the `<APIM_HOME>/repository/deployment/server/synapse-configs/default/sequences` directory, if you are in single-tenant mode. In multi-tenant mode, copy the file to the tenant's synapse sequence folder. For example, if tenant id is 1, then copy it to `<API_Gateway>/repository/tenants/1/synapse-configs/default/sequences` folder.

The above sequence prints a log message on the console whenever the `TwitterSearch` API is invoked.

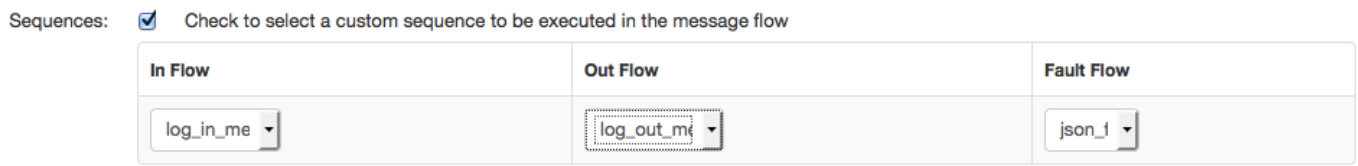
Alternatively, you can create the XML file and upload it to the registry using the management console UI.

1. Open the APIM management console (https://localhost:9443/carbon with admin/admin as the default credentials) and select **Resources -> Browse**.
2. Navigate to `/_system/governance/apimgt/customsequences` registry location.
3. Click **Add Resource** link to upload the XML file.

Selecting predefined APIs from the UI

You can attach pre-defined extension sequences to an API using the API Publisher Web interface, at the time the API is created. Log in to the API Publisher (https://localhost:9443/publisher) and click **Add** from the left panel. In the

Add New API page that opens, navigate to the **Manager** section where you find **Sequences**. There, you can select `In/Out/Fault` sequences for the API from the drop-down lists. For example,



To populate these drop-down lists, you must add mediation sequences as explained at the beginning.

Invoking the extension sequences

When an API is published, a file with its synapse configuration is created on the API Gateway. This synapse configuration has a set of handlers as shown in the following example:

API Configuration

```
<handlers>
        <handler
class="org.wso2.carbon.apimgt.gateway.handlers.security.APIAuthenticationHandler"/>
        <handler class="org.wso2.carbon.apimgt.usage.publisher.APIMgtUsageHandler"/>
        <handler
class="org.wso2.carbon.apimgt.usage.publisher.APIMgtGoogleAnalyticsTrackingHandler"/>
        <handler
class="org.wso2.carbon.apimgt.gateway.handlers.throttling.APIThrottleHandler">
            <property name="id" value="A"/>
            <property name="policyKey" value="gov:/apimgt/applicationdata/tiers.xml"/>
        </handler>
        <handler
class="org.wso2.carbon.apimgt.gateway.handlers.ext.APIManagerExtensionHandler"/>
</handlers>
```

The handler by the name `APIManagerExtensionHandler` triggers both global as well as per-API extension sequences. It reads the sequence names and determines what APIs must be invoked. By default, the extension handler is listed at last in the handler chain, and therefore is executed last. You can configure the API Gateway to execute extension handlers first. To do that, open `<APIM_HOME>/repository/conf/api-manager.xml` file, uncomment the `<ExtensionHandlerPosition>` section and provide the value `top` as follows:

```
<ExtensionHandlerPosition>top</ExtensionHandlerPosition>
```

This is useful when you want to execute your own extensions before our default handlers. For example, if you want to have additional security checks such as signature verification on access tokens before executing the default security handler, you can define an extension and configure the Gateway to execute extension handlers first.

For more information on Handlers, see Architecture.

Adding Workflow Extensions

Workflow extensions allow you to attach a custom workflow to various operations in the API Manager such as user signup, application creation, registration, subscription etc. By default, the API Manager workflows have **Simple Workflow Executor** engaged in them. The Simple Workflow Executor carries out an operation without any intervention by a workflow admin. For example, when the user creates an application, the Simple Workflow Executor allows the application to be created without the need for an admin to approve the creation process.

In order to enforce intervention by a workflow admin, you can engage the **WS Workflow Executor**. It invokes an external Web service when executing a workflow and the process completes based on the output of the Web

service. For example, when the user creates an application, the request goes into an intermediary state where it remains until authorized by a workflow admin.

By default, the WS Workflow Executor comes with,

- A sample BPEL and Human Task for each standard workflow such as application creation, registration, subscription etc. You can also customize the default implementations.
- A Jaggery-based Web application named `workflow-admin` (`https://localhost:9443/workflow-admin`). It provides a GUI for the workflow admin to approve/reject pending Human Tasks.

  When executing a workflow, an entry is added to the `AM_WORKFLOWS` table in the API Manager Database, indicating the workflow status and workflow external reference along with other information. This entry is used to track the progress of the workflow throughout its lifecycle. At a given time, the status of a workflow can be `CREATED`, `APPROVED` or `REJECTED`. `CREATED` is the default status of a workflow. It gets promoted to `APPROVED` or `REJECTED`, based on the response from the workflow engine.

  > You can maintain any number of states/steps for a workflow in between the `CREATED` and `APPROVED/REJECTED` states inside the workflow engine. The API Manager only acknowledges the `CREATED/REJECTED` states.

The sections below explain different workflows provided by the API Manager to engage business processes with API management operations. They also explain how to customize the default workflows:

- Adding an Application Creation Workflow
- Adding an Application Registration Workflow
- Adding an API Subscription Workflow
- Adding a User Signup Workflow
- Invoking API Manager from the BPEL Engine
- Customizing a Workflow Extension
- Configuring Workflows for Tenants

Adding an Application Creation Workflow

This section explains how to attach a custom workflow to the application creation operation in the API Manager. First, see Workflow Extensions for information on different types of workflow executors.

Configuring the Business Process Server

1. Download WSO2 Business Process Server.
2. Set an offset of 2 to the default BPS port in `<BPS_HOME>/repository/conf/carbon.xml` file. This prevents port conflicts that occur when you start more than one WSO2 product on the same server.

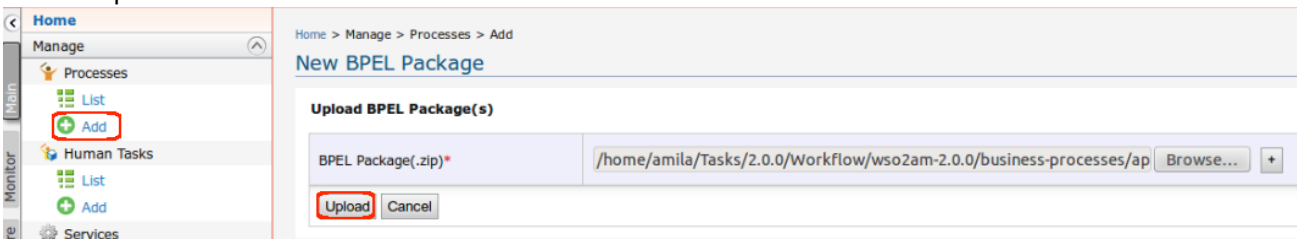
   ```
   <Offset>2</Offset>
   ```

   > ⚠ If you change the port offset to a value other than 2 or run the API Manager and BPS on different machines (therefore, want to set the `hostname` to a different value than `localhost`), you must do the following:
   >
   > - Search and replace the value 9765 in all the files (.epr) inside `<APIM_HOME>/business-processes` folder with the new port
   > - Search and replace port 9445 in `<AM_HOME>/repository/deployment/server/jaggeryapps/admin-dashboard/site/conf/site.json` file
   > - Also change the hard-coded endpoints described in Changing the Default Ports with Offset

3. Copy the following from `<APIM_HOME>/business-processes/epr` to `<BPS_HOME>/repository/conf/epr` folder.
   - `ApplicationService.epr`

- `ApplicationCallbackService.epr`

4. Start the BPS server and log in to its management console (`https://<Server Host>:9443+<port offset>/carbon`).
5. Select **Add** under **Processes** menu and upload the `<APIM_HOME>/business-processes/application-creation/BPEL/ApplicationApprovalWorkFlowProcess_1.0.0.zip` file to BPS. This is the business process archive file.

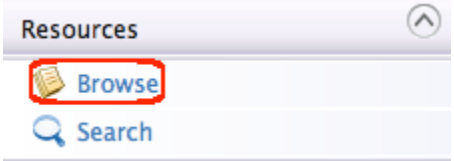


6. Select **Add** under the **Human Tasks** menu and upload `<APIM_HOME>/business-processes/application-creation/HumanTask/ApplicationsApprovalTask-1.0.0.zip` to BPS. This is the human task archived file.

Engaging the WS Workflow Executor in the API Manager

First, enable the application creation workflow**.**

1. Log in to APIM management console (`https://<Server Host>:9443/carbon`) and select **Browse** under **Resources**.



2. Go to `/_system/governance/apimgt/applicationdata/workflow-extensions.xml` resource, disable the Simple Workflow Executor and enable WS Workflow Executor. Also specify the service endpoint where the workflow engine is hosted and the credentials required to access the said service via basic authentication (i.e., username/password based authentication).

```
<WorkFlowExtensions>
    <!--ApplicationCreation
executor="org.wso2.carbon.apimgt.impl.workflow.ApplicationCreationSimpleWorkflowE
xecutor"/-->
    <ApplicationCreation
executor="org.wso2.carbon.apimgt.impl.workflow.ApplicationCreationWSWorkflowExecu
tor">
        <Property
name="serviceEndpoint">http://localhost:9765/services/ApplicationApprovalWorkFlow
Process/</Property>
        <Property name="username">admin</Property>
        <Property name="password">admin</Property>
        <Property
name="callbackURL">https://localhost:8243/services/WorkflowCallbackService</Prope
rty>
    </ApplicationCreation>
</WorkFlowExtensions>
```

The application creation WS Workflow Executor is now engaged.

3. Go to the API Store Web interface, open **My Applications** page and create a new application.
It invokes the application creation process and creates a Human Task instance that holds the execution of the

BPEL process until some action is performed on it.

4. Note the message that appears if the BPEL is invoked correctly, saying that the request is successfully submitted.
5. Log in to the workflow-admin app (`https://localhost:9443/workflow-admin`), list all the tasks for application creation and approve the task. It resumes the BPEL process and completes the application creation.
6. Go back to the **My Applications** page on the API Store and see the created application.

Whenever a user tries to create an application in the API Store, a request is sent to the workflow endpoint. Given below is a sample:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:wor="http://workflow.subscription.apimgt.carbon.wso2.org">
   <soapenv:Header />
   <soapenv:Body>
      <wor:createApplication
xmlns:wor="http://workflow.application.apimgt.carbon.wso2.org">
         <wor:applicationName>application1</wor:applicationName>
         <wor:applicationTier>Gold</wor:applicationTier>

<wor:applicationCallbackUrl>http://webapp/url</wor:applicationCallbackUrl>
         <wor:applicationDescription>Application 1</wor:applicationDescription>
         <wor:tenantDomain>wso2.com</wor:tenantDomain>
         <wor:userName>user1</wor:userName>

<wor:workflowExternalRef>c0aad878-278c-4439-8d7e-712ee71d3f1c</wor:workflowExtern
alRef>

<wor:callBackURL>https://localhost:8243/services/WorkflowCallbackService</wor:cal
lBackURL>
      </wor:createApplication>
   </soapenv:Body>
</soapenv:Envelope>
```

Elements of the above configuration are described below:

Element	Description
`applicationName`	Name of the application the user creates.
`applicationTier`	Throttling tier of the application.
`applicationCallbackUrl`	When the OAuth2 Authorization Code grant type is applied, this is the endpoint on which the callback needs to happen after the user is authenticated. This is an attribute of the actual application registered on the API Store.
`applicationDescription`	Description of the application
`tenantDomain`	Tenant domain associated with the application (domain of the user creating the application).
`userName`	username of the user creating the application.

workflowExternalRef	The unique reference against which a workflow is tracked. This needs to be sent back from the workflow engine to the API Manager at the time of workflow completion.
`callBackURL`	At the time of workflow completion, the workflow-completion request is sent to this URL by the workflow engine. This property is configured in the <callBackURL> element in the api-manager.xml.

Adding an Application Registration Workflow

This section explains how to attach a custom workflow to the application registration operation in the API Manager. First, see Workflow Extensions for information on different types of workflow executors.

Introduction to the application registration workflow

Application creation and registration are different workflows. After an application is created, you can subscribe to available APIs, but you get the consumer key/secret and access tokens only after registering the application. There are two types of registrations that can be done to an application: production and sandbox. You change the default application registration workflow in situations such as the following:

1. To issue only sandbox keys when creating production keys is deferred until testing is complete.
2. To restrict untrusted applications from creating production keys. You allow only the creation of sandbox keys.
3. To make API subscribers go through an approval process before creating any type of access token.

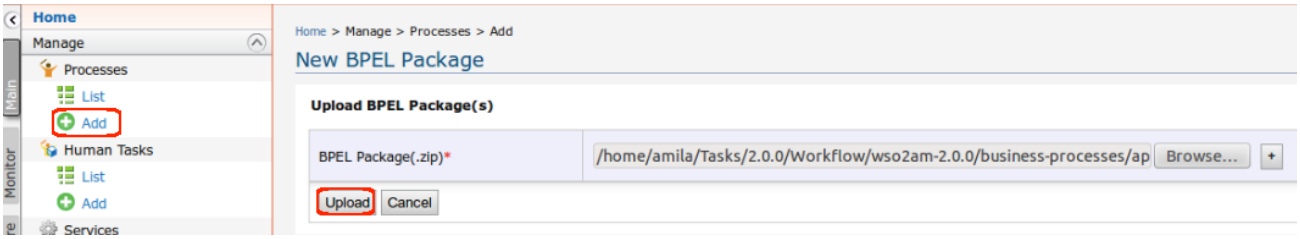
Configuring the Business Process Server

1. Download WSO2 Business Process Server.
2. Set an offset of 2 to the default BPS port in `<BPS_HOME>/repository/conf/carbon.xml` file. This prevents port conflicts that occur when you start more than one WSO2 product on the same server. Also see Changing the Default Ports with Offset.

```
<Offset>2</Offset>
```

> If you change the port offset to a value other than 2 or run the API Manager and BPS on different machines (therefore, want to set the `hostname` to a different value than `localhost`), you must do the following:
>
> - Search and replace the value 9765 in all the files (.epr, .wsdl files inside the ZIP archives) inside `<APIM_HOME>/business-processes` folder with the new port
> - Zip the files you unzipped earlier and deploy the newly created zip file in BPS as explained in the steps below
> - Search and replace port 9445 in `<AM_HOME>/repository/deployment/server/jaggeryapps/admin-dashboard/site/conf/site.json` file

3. Copy the following from `<APIM_HOME>/business-processes/epr` to `<BPS_HOME>/repository/conf/epr` folder.
   - `RegistrationService.epr`
   - `RegistrationCallbackService.epr`
4. Start the BPS server and log in to its management console (`https://<Server Host>:9443+<port offset>/carbon`).
5. Select **Add** under **Processes** menu and upload t he `<APIM_HOME>/business-processes/application-registration/BPEL/A pplicationRegistrationWorkflowProcess_1.0.0.zip` file to BPS. This is the business process archive file.

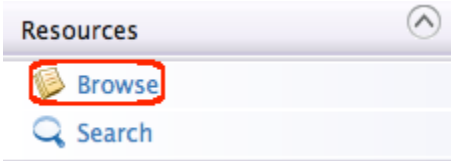


6. Select **Add** under the **Human Tasks** menu and upload `<APIM_HOME>/business-processes/application-registration/HumanTaskBPEL/A pplicationRegistrationTask-1.0.0.zip` to BPS. This is the human task archived file.

Engaging the WS Workflow Executor in the API Manager

First, enable the application registration workflow**.**

1. Log in to APIM management console (`https://<Server Host>:9443/carbon`) and select **Browse** under **Resources** .

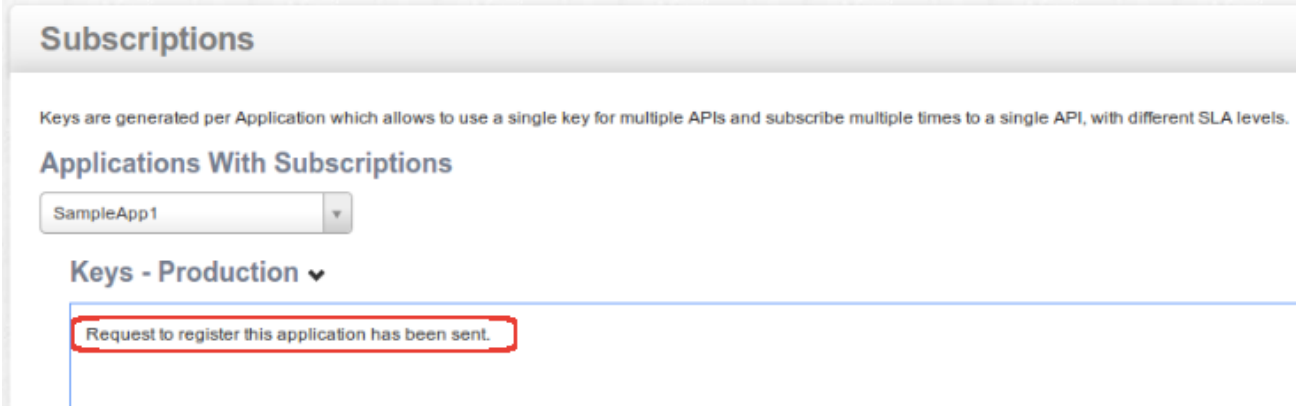


2. Go to `/_system/governance/apimgt/applicationdata/workflow-extensions.xml` resource, disable the Simple Workflow Executor and enable WS Workflow Executor:

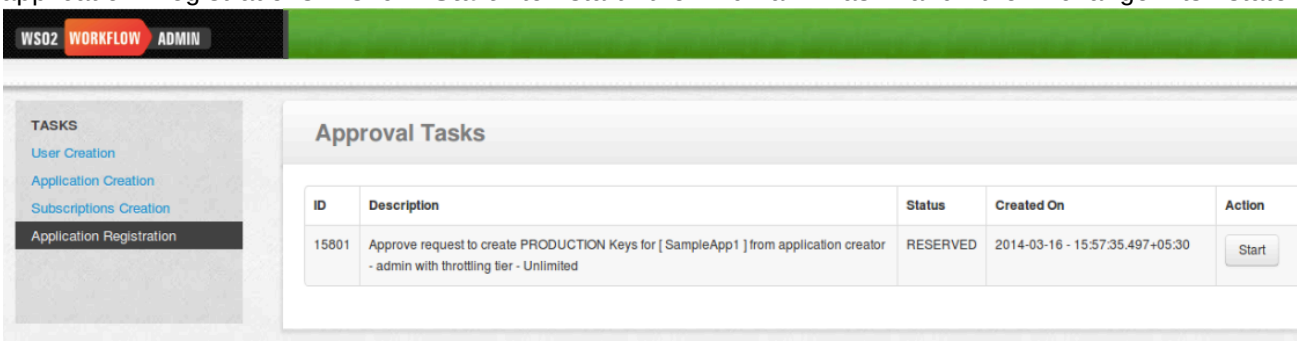
```
<WorkFlowExtensions>
    <!--ProductionApplicationRegistration
executor="org.wso2.carbon.apimgt.impl.workflow.ApplicationRegistrationSimpleWorkf
lowExecutor"/-->
    <ProductionApplicationRegistration
executor="org.wso2.carbon.apimgt.impl.workflow.ApplicationRegistrationWSWorkflowE
xecutor">
        <Property
name="serviceEndpoint">http://localhost:9765/services/ApplicationRegistrationWork
FlowProcess/</Property>
        <Property name="username">admin</Property>
        <Property name="password">admin</Property>
        <Property
name="callbackURL">https://localhost:8248/services/WorkflowCallbackService</Prope
rty>
    </ProductionApplicationRegistration>
    <!--SandboxApplicationRegistration
executor="org.wso2.carbon.apimgt.impl.workflow.ApplicationRegistrationSimpleWorkf
lowExecutor"/-->
    <SandboxApplicationRegistration
executor="org.wso2.carbon.apimgt.impl.workflow.ApplicationRegistrationWSWorkflowE
xecutor">
        <Property
name="serviceEndpoint">http://localhost:9765/services/ApplicationRegistrationWork
FlowProcess/</Property>
        <Property name="username">admin</Property>
        <Property name="password">admin</Property>
        <Property
name="callbackURL">https://localhost:8248/services/WorkflowCallbackService</Prope
rty>
    </SandboxApplicationRegistration>
</WorkFlowExtensions>
```

> ⚠ Note that all workflow process services of the BPS run on port 9765 as you changed its default port with an offset of 2.

3. Go to the API Store Web interface, open **My Subscriptions** page, select an application and click the **Generate** button associated with the production key.
   It invokes the `ApplicationRegistrationWorkFlowProcess.bpel` that is bundled with `ApplicationRegistrationWorkflowProcess_1.0.0.zip` and creates a HumanTask instance that holds the execution of the BPEL process until some action is performed on it.
4. Note a message that appears saying that the request is successfully submitted if the BPEL was invoked correctly. For example,



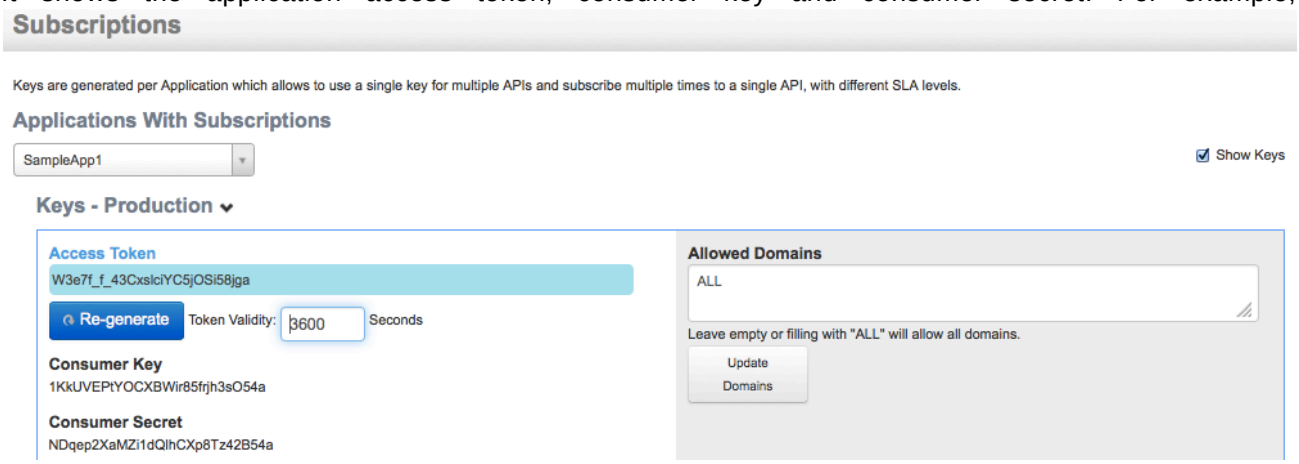
5. Log in to the workflow-admin app (`https://localhost:9443/workflow-admin`) and list all the tasks for application registrations. Click **Start** to start the Human Task and then change its state.



Once you approve the task, it resumes the BPEL process and completes the registration.

6. Go back to the **My Subscriptions** page on the API Store and view your application.

It shows the application access token, consumer key and consumer secret. For example,



After the registration request is approved, keys are generated by invoking the `APIKeyMgtSubscriber` service hosted in Key Manger nodes. Even when the request is approved, key generation can fail if this service becomes unavailable. To address such failures, you can configure to trigger key generation at a time Key Manager nodes become available again.

Given below is the message used to invoke the BPEL process:

```
<applicationregistrationworkflowprocessrequest
xmlns:wor="http://workflow.application.apimgt.carbon.wso2.org"
xmlns="http://workflow.application.apimgt.carbon.wso2.org">
   <applicationname>NewApp5</applicationname>
   <applicationtier>Unlimited</applicationtier>
   <applicationcallbackurl></applicationcallbackurl>
   <applicationdescription></applicationdescription>
   <tenantdomain>carbon.super</tenantdomain>
   <username>admin</username>

<workflowexternalref>4a20749b-a10d-4fa5-819b-4fae5f57ffaf</workflowexternalref>

<callbackurl>https://localhost:8243/services/WorkflowCallbackService</callbackurl
>
   <keytype>PRODUCTION</keytype>
</applicationregistrationworkflowprocessrequest>
```

Adding an API Subscription Workflow

This section explains how to attach a custom workflow to the API subscription operation in the API Manager. First, see Workflow Extensions for information on different types of workflows executors.

Attaching a custom workflow to API subscription enables you to add throttling tiers to an API that consumers cannot choose at the time of subscribing. Only admins can set these tiers to APIs. It also allows you to restrict API consumers to only subscribe to sandbox, and then go through an approval process to go to the next level of subscription.

Configuring the Business Process Server

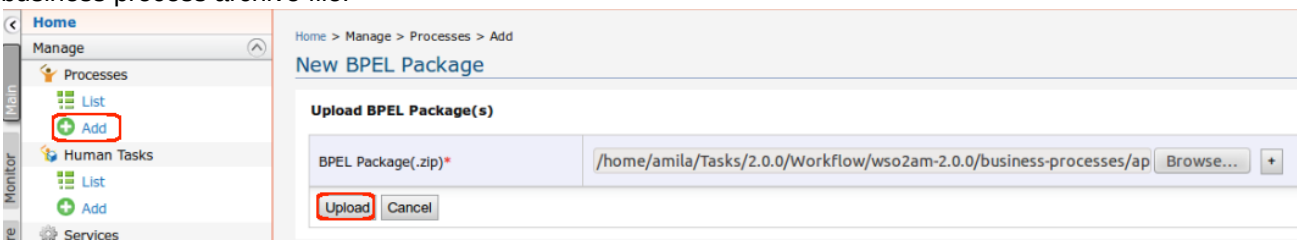
1. Download WSO2 Business Process Server.
2. Set an offset of 2 to the default BPS port in `<BPS_HOME>/repository/conf/carbon.xml` file. This prevents port conflicts that occur when you start more than one WSO2 product on the same server. Also see Changing the Default Ports with Offset.

   ```
   <Offset>2</Offset>
   ```

   > If you change the port offset to a value other than 2 or run the API Manager and BPS on different machines (therefore, want to set the `hostname` to a different value than `localhost`), you must do the following:
   >
   > - Search and replace the value 9765 in all the files (.epr, .wsdl files inside the ZIP archives) inside `<APIM_HOME>/business-processes` folder with the new port
   > - Zip the files you unzipped earlier and deploy the newly created zip file in BPS as explained in the steps below
   > - Search and replace port 9445 in `<AM_HOME>/repository/deployment/server/jaggeryapps/admin-dashboard/site/conf/site.json` file

3. Copy the following from `<APIM_HOME>/business-processes/epr` to `<BPS_HOME>/repository/conf/epr` folder.
   - `SubscriptionService.epr`
   - `SubscriptionCallbackService.epr`
4. Start the BPS server and log in to its management console (`https://<Server Host>:9443+<port offset>/carbon`).

5. Select **Add** under **Processes** menu and upload t he `<APIM_HOME>/business-processes/subscription-creation/BPEL/SubscriptionApprovalWorkFlowProcess_1.0.0.zip` file to BPS. This is the business process archive file.



6. Select **Add** under the **Human Tasks** menu and upload `<APIM_HOME>/business-processes/subscription-creation/HumanTask/SubscriptionsApprovalTask-1.0.0.zip` to BPS. This is the human task archived file.

Engaging the WS Workflow Executor in the API Manager

First, enable the API subscription workflow**.**

1. Log in to APIM admin console (`https://<Server Host>:9443/carbon`) and select **Browse** under **Resources**.

2. Go to `/_system/governance/apimgt/applicationdata/workflow-extensions.xml` resource, disable the Simple Workflow Executor and enable WS Workflow Executor. Also specify the service endpoint where the workflow engine is hosted and the credentials required to access the said service via basic authentication (i.e., username/password based authentication).

```
<WorkFlowExtensions>
    <!--SubscriptionCreation
executor="org.wso2.carbon.apimgt.impl.workflow.SubscriptionCreationSimpleWorkflow
Executor"/-->
    <SubscriptionCreation
executor="org.wso2.carbon.apimgt.impl.workflow.SubscriptionCreationWSWorkflowExec
utor">
        <Property
name="serviceEndpoint">http://localhost:9765/services/SubscriptionApprovalWorkFlo
wProcess/</Property>
        <Property name="username">admin</Property>
        <Property name="password">admin</Property>
        <Property
name="callbackURL">https://localhost:8243/services/WorkflowCallbackService</Prope
rty>
    </SubscriptionCreation>
</WorkFlowExtensions>
```

The application creation WS Workflow Executor is now engaged.

3. Go to the API Store Web interface and subscribe to an API.
   It invokes the API subscription process and creates a Human Task instance that holds the execution of the BPEL until some action is performed on it.
4. Note the message that appears if the BPEL is invoked correctly, saying that the request is successfully submitted.

5. Log in to the workflow-admin app (`https://localhost:9443/workflow-admin`), list all the tasks for API subscription and approve the task. It resumes the BPEL process and completes the API subscription.
6. Go back to the API Store and see that the user is now subscribed to the API.

   Whenever a user tries to subscribe to an API, a request of the following format is sent to the workflow endpoint:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:wor="http://workflow.subscription.apimgt.carbon.wso2.org">
   <soapenv:Header/>
   <soapenv:Body>
      <wor:createSubscription>
         <wor:apiName>sampleAPI</wor:apiName>
         <wor:apiVersion>1.0.0</wor:apiVersion>
         <wor:apiContext>/sample</wor:apiContext>
         <wor:apiProvider>admin</wor:apiProvider>
         <wor:subscriber>subscriber1</wor:subscriber>
         <wor:applicationName>application1</wor:applicationName>
         <wor:tierName>gold</wor:tierName>
         <wor:workflowExternalRef></wor:workflowExternalRef>
         <wor:callBackURL>?</wor:callBackURL>
      </wor:createSubscription>
   </soapenv:Body>
</soapenv:Envelope>
```

Elements of the above configuration are described below:

Element	Description
`apiName`	Name of the API to which subscription is requested.
`apiVersion`	Version of the API the user subscribes to.
`apiContext`	Context in which the requested API is to be accessed.
`apiProvider`	Provider of the API.
`subscriber`	Name of the user requesting subscription.
`applicationName`	Name of the application through which the user subscribes to the API.
`tierName`	Throttling tiers specified for the application.
workflowExternalRef	The unique reference against which a workflow is tracked. This needs to be sent back from the workflow engine to the API Manager at the time of workflow completion.
`callBackURL`	The URL to which the Workflow completion request is sent to by the workflow engine, at the time of workflow completion. This property is configured under the callBackURL property in the api-manager.xml.

Adding a User Signup Workflow

This section explains how to attach a custom workflow to the application creation operation in the API Manager.

First, see Workflow Extensions for information on different types of workflow executors.

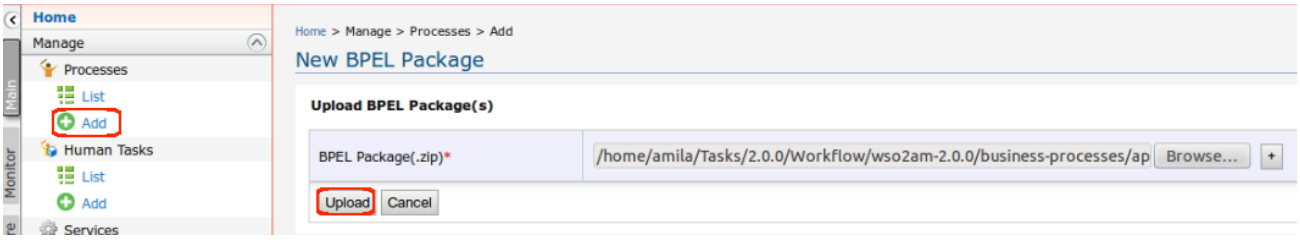
Configuring the Business Process Server

1. Download WSO2 Business Process Server.
2. Set an offset of 2 to the default BPS port in `<BPS_HOME>/repository/conf/carbon.xml` file. This prevents port conflicts that occur when you start more than one WSO2 product on the same server. Also see Changing the Default Ports with Offset.

```
<Offset>2</Offset>
```

> If you change the port offset to a value other than 2 or run the API Manager and BPS on different machines (therefore, want to set the `hostname` to a different value than `localhost`), you must do the following:
>
> - Search and replace the value 9765 in all the files (.epr, .wsdl files inside the ZIP archives) inside `<APIM_HOME>/business-processes` folder with the new port
> - Zip the files you unzipped earlier and deploy the newly created zip file in BPS as explained in the steps below
> - Search and replace port 9445 in `<AM_HOME>/repository/deployment/server/jaggeryapps/admin-dashboard/site/conf/site.json` file

3. Copy the following from `<APIM_HOME>/business-processes/epr` to `<BPS_HOME>/repository/conf/epr` folder.
   - `UserSignupService.epr`
   - `UserSignupProcess.epr`
4. Start the BPS server and log in to its management console (`https://<Server Host>:9443+<port offset>/carbon`).
5. Select **Add** under **Processes** menu and upload the `<APIM_HOME>/business-processes/user-signup/BPEL/UserSignupApprovalProcess_1.0.0.zip` file to BPS. This is the business process archive file.

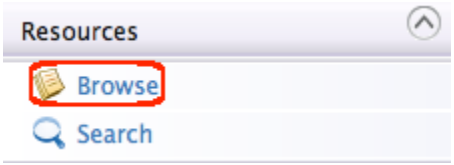


6. Select **Add** under the **Human Tasks** menu and upload `<APIM_HOME>/business-processes/user-signup/HumanTask/UserApprovalTask-1.0.0.zip` to BPS. This is the human task archived file.

Engaging the WS Workflow Executor in the API Manager

First, enable the user signup workflow**.**

1. Log in to APIM management console (`https://<Server Host>:9443/carbon`) and select **Browse** under **Resources**.



2. Go to `/_system/governance/apimgt/applicationdata/workflow-extensions.xml` resource, disable the Simple Workflow Executor and enable WS Workflow Executor. Also specify the service endpoint

where the workflow engine is hosted and the credentials required to access the said service via basic authentication (i.e., username/password based authentication).

```
<WorkFlowExtensions>
    <!--UserSignUp
executor="org.wso2.carbon.apimgt.impl.workflow.UserSignUpSimpleWorkflowExecutor"/
-->
    <UserSignUp
executor="org.wso2.carbon.apimgt.impl.workflow.UserSignUpWSWorkflowExecutor">
         <Property
name="serviceEndpoint">http://localhost:9765/services/UserSignupProcess/</Propert
y>
         <Property name="username">admin</Property>
         <Property name="password">admin</Property>
         <Property
name="callbackURL">https://localhost:8243/services/WorkflowCallbackService</Prope
rty>
    </UserSignUp>
</WorkFlowExtensions>
```

3. Go to the API Store Web interface and sign up. It invokes the signup process and creates a Human Task instance that holds the execution of the BPEL until some action is performed on it.
4. Note the message that appears if the BPEL is invoked correctly, saying that the request is successfully submitted.
5. Log in to the workflow-admin app (`https://localhost:9443/workflow-admin`) and approve the user signup task. It resumes the BPEL process and completes the signup process.
6. Go back to the API Store and see that the user is now registered.

Whenever a user tries to sign up to the API Store, a request of the following format is sent to the workflow endpoint:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:wor="http://workflow.subscription.apimgt.carbon.wso2.org">
   <soapenv:Header />
   <soapenv:Body>
      <wor:registerUser
xmlns:wor="http://workflow.registeruser.apimgt.carbon.wso2.org">
         <wor:userName>sampleuser</wor:userName>
         <wor:tenantDomain>foo.com</wor:tenantDomain>

<wor:workflowExternalRef>c0aad878-278c-4439-8d7e-712ee71d3f1c</wor:workflowExtern
alRef>

<wor:callbackURL>https://localhost:8243/services/WorkflowCallbackService</wor:cal
lBackURL>
      </wor:registerUser>
   </soapenv:Body>
</soapenv:Envelope>
```

Elements of the above configuration are described below:

Element	Description

`userName`	The user name requested by the user
tenantDomain	Domain to which the user belongs to
workflowExternalRef	The unique reference against which a workflow is tracked. This needs to be sent from the workflow engine to the API Manager at the time of workflow completion.
`callBackURL`	The URL to which the workflow completion request is sent by the workflow engine, at the time of workflow completion. This property is configured under the "callBackURL" property in the api-manager.xml.

Invoking API Manager from the BPEL Engine

Once the workflow is finalized at BPEL end the call back url (originally configured in the api-manager.xml and sent to BPEL Engine in the outflow) of API Manager will be called to progress the workflow. In AM, endpoint has been made available in both SOAP and REST variants. They are respectively;

Type	URI
SOAP	https://localhost:8243/services/WorkflowCallbackService WSDLLocation : http://localhost:8280/services/WorkflowCallbackService?wsdl
REST	https://localhost:9443/store/site/blocks/workflow/workflow-listener/ajax/workflow-listener.jag

Both the endpoints have been secured via Basic Authentication. Hence when invoking either, an Authorization header including a base64 encoded value of the User's username and password needs to be included, along with the request. (E.g : `Authorization: Basic <base64 encoded username:password>`)

The endpoint expects the following list of parameters.

Parameter	Description	Mandatory
`workflowReference`	The unique identifier sent to the BPEL against which the workflow is tracked in API Manager	YES
`status`	The next status to which the workflow needs to be promoted to.	YES
`description`	Notes, that may need to be persisted against a particular workflow.	NO

A sample curl request for invoking the REST endpoint would be as follows.

```
curl -H "Authorization:Basic YWRtaW46YWRtaW4=" -X POST
http://localhost:9763/store/site/blocks/workflow/workflow-listener/ajax/workflow-liste
ner.jag -d
'workflowReference=b530be39-9174-43b3-acb3-2603a223b094&status=APPROVED&description=DE
SCRIPTION'
```

A sample SOAP request would be as below.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:cal="http://callback.workflow.apimgt.carbon.wso2.org">
   <soapenv:Header/>
   <soapenv:Body>
      <cal:resumeEvent>

<cal:workflowReference>b530be39-9174-43b3-acb3-2603a223b094</cal:workflowReference>
         <cal:status>APPROVED</cal:status>
         <cal:description>DESCRIPTION</cal:description>
      </cal:resumeEvent>
   </soapenv:Body>
</soapenv:Envelope>
```

Customizing a Workflow Extension

Each workflow executor in the WSO2 API Manager is inherited from the **`org.wso2.carbon.apimgt.impl.workflow.WorkflowExecutor`** abstract class, which has two abstract methods:

- **`execute`**: contains the implementation of the workflow execution
- **`complete`**: contains the implementation of the workflow completion
- **`getWorkflowType`**: abstract method that returns the type of the workflow as a String
- **`getWorkflowDetails(String workflowStatus)`**: abstract method that returns a list of `WorkflowDTO` objects. This method is not used at the moment and it returns null for the time being.

To customize the default workflow extension, you override the **`execute()`** and **`complete()`** methods with your custom implementation. For example, the following class is a sample implementation of the Subscription Creation workflow. It returns an email to an address provided through the configuration on each subscription creation:

```
package org.wso2.sample.workflow;

import java.util.List;
import java.util.Properties;
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.PasswordAuthentication;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;
import org.wso2.carbon.apimgt.api.APIManagementException;
import org.wso2.carbon.apimgt.impl.APIConstants;
import org.wso2.carbon.apimgt.impl.dao.ApiMgtDAO;
import org.wso2.carbon.apimgt.impl.dto.SubscriptionWorkflowDTO;
import org.wso2.carbon.apimgt.impl.dto.WorkflowDTO;
import org.wso2.carbon.apimgt.impl.workflow.WorkflowConstants;
import org.wso2.carbon.apimgt.impl.workflow.WorkflowException;
import org.wso2.carbon.apimgt.impl.workflow.WorkflowExecutor;
import org.wso2.carbon.apimgt.impl.workflow.WorkflowStatus;

public class SubsCreationEmailSender extends WorkflowExecutor {
    private String adminEmail;
    private String emailAddress;
    private String emailPassword;

    @Override
```

```
    public List<WorkflowDTO> getWorkflowDetails(String arg0)
            throws WorkflowException {
        return null;
    }

    @Override
    public String getWorkflowType() {
        return WorkflowConstants.WF_TYPE_AM_SUBSCRIPTION_CREATION;
    }

    @Override
    public void execute(WorkflowDTO workflowDTO) throws WorkflowException{
        SubscriptionWorkflowDTO subsCreationWFDTO =
(SubscriptionWorkflowDTO)workflowDTO;

        Properties props = new Properties();
        props.put("mail.smtp.auth", "true");
        props.put("mail.smtp.starttls.enable", "true");
        props.put("mail.smtp.host", "smtp.gmail.com");
        props.put("mail.smtp.port", "587");

        Session session = Session.getInstance(props,
                new javax.mail.Authenticator() {
                    protected PasswordAuthentication getPasswordAuthentication() {
                        return new PasswordAuthentication(emailAddress,
                                emailPassword);
                    }
                });

        try {

            Message message = new MimeMessage(session);
            message.setFrom(new InternetAddress(emailAddress));
            message.setRecipients(Message.RecipientType.TO,
                InternetAddress.parse(adminEmail));
            message.setSubject("Subscription Creation");
            message.setText("Subscription created for API " +
subsCreationWFDTO.getApiName() +
                            " using Application " +
subsCreationWFDTO.getApplicationName() +
                            " by user " + subsCreationWFDTO.getSubscriber());

            Transport.send(message);
            System.out.println("Sent email to notify subscription creation");
            //Call the execute method of the parent class. This will create a
reference for the
            //workflow execution in the database.
            super.execute(workflowDTO);
            //Set the workflow Status to APPROVED and Immediately complete the
workflow since we
            //are not waiting for an external party to complete this.
            workflowDTO.setStatus(WorkflowStatus.APPROVED);
            complete(workflowDTO);

        } catch (MessagingException e) {
            e.printStackTrace();
            throw new WorkflowException(e.getMessage());
        } catch (Exception e){
            e.printStackTrace();
```

```
        throw new WorkflowException(e.getMessage());
    }
}

@Override
public void complete(WorkflowDTO workflowDTO) throws WorkflowException{
    workflowDTO.setUpdatedTime(System.currentTimeMillis());
    super.complete(workflowDTO);
    ApiMgtDAO apiMgtDAO = new ApiMgtDAO();
    try {
        apiMgtDAO.updateSubscriptionStatus(
                Integer.parseInt(workflowDTO.getWorkflowReference()),
                APIConstants.SubscriptionStatus.UNBLOCKED);
    } catch (APIManagementException e) {
        throw new WorkflowException(
                "Could not complete subscription creation workflow", e);
    }
}
public String getAdminEmail() {
    return adminEmail;
}
public void setAdminEmail(String adminEmail) {
    this.adminEmail = adminEmail;
}
public String getEmailAddress() {
    return emailAddress;
}
public void setEmailAddress(String emailAddress) {
    this.emailAddress = emailAddress;
}
public String getEmailPassword() {
    return emailPassword;
}
public void setEmailPassword(String emailPassword) {
```

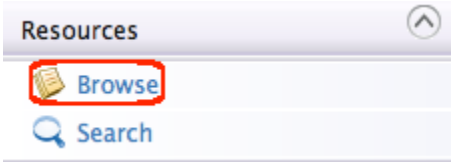
```
        this.emailPassword = emailPassword;
    }
}
```

Note the following regarding the above sample:

- The `execute()` method takes in a `WorkflowDTO` object (**SubscriptionWorkflowDTO** class) that contains information about the subscription that is being created.
- The `adminEmail`, `emailAddress` and `emailPassword` are private String variables with public `getter` and `setter` methods. The values for these variables are populated through the server configuration.
- After sending the email, a call is made to the super class's `execute()` method in order to create a reference entry in the database. This entry is generally used to look up the workflow when the workflow happens asynchronously (via a human approval).
- The complete() method contains the code to mark the subscription active. Until then, the subscription is in ON_HOLD state.
- In this sample, the `complete()` method is called immediately to make the subscription active instantly. If the completion of your workflow happens asynchronously, you must not call the `complete()` method from the `execute()` method.
- The `WorkflowException` is thrown to roll back the subscription in case of a failure.

After the implementation of the class is done, follow the steps below to implement the new workflow extension in the API Manager:

1. Compile the class and export it as a JAR file. Make sure you have the following JARs in the classpath before compilation.
   - `<AM_HOME>/repository/components/plugins/org.wso2.carbon.apimgt.impl_1.2.1.jar`
   - `<AM_HOME>/repository/components/plugins/org.wso2.carbon.apimgt.api_1.2.1.jar`
   - `javax.mail.jar`: see https://java.net/projects/javamail/pages/Home to download the JAR
2. After exporting the JAR, copy it to `<AM_HOME>/repository/components/lib`.
3. Log in to APIM management console (`https://<Server Host>:9443/carbon`) and select **Browse** under **Resources**.



4. Go to `/_system/governance/apimgt/applicationdata/workflow-extensions.xml` resource, disable the Simple Workflow Executor and enable WS Workflow Executor. Also specify the service endpoint where the workflow engine is hosted and the credentials required to access the said service via basic authentication (i.e., username/password based authentication). For example:

```
<WorkFlowExtensions>
    <!--SubscriptionCreation
executor="org.wso2.carbon.apimgt.impl.workflow.SubscriptionCreationSimpleWorkflow
Executor"/-->
    <SubscriptionCreation
executor="org.wso2.sample.workflow.SubsCreationEmailSender">
       <Property name="adminEmail">to_user@email.com</Property>
       <Property name="emailAddress">from _user@email.com</Property>
       <Property name="emailPassword">from_user_password</Property>
    </SubscriptionCreation>
</WorkFlowExtensions>
```

Note that the `adminEmail`, `emailAddress` and `emailPassword` properties will be assigned to the appropriate variables defined in the class through the public `setter` methods of those variables.

If you use the same or similar sample to return an email, you must remove the `org.jaggeryjs.hostobjects.email_0.9.0.ALPHA4_wso2v1.jar` file from `<AM_HOME>/repository/components/plugins` directory. Removing it results in a `ClassNotFoundException` thrown at server startup, but it does not affect the server's functionality.

Configuring Workflows for Tenants

Using the API Manager, you can configure custom workflows that get invoked at the event of a user signup, application creation, registration, subscription etc. You do these configurations in the api-manager.xml as described in the previous sections.

However, in a multi-tenant API Manager setup, not all tenants have access to the file system and not all tenants want to use the same workflow that the super admin has configured in the api-manager.xml file. For example, different departments in an enterprise can act as different tenants using the same API Manager instance and they can have different workflows. Also, an enterprise can combine WSO2 API Manager and WSO2 Business Process Server (BPS) to provide API Management As a Service to its clients. In this case, each client is a separate enterprise represented by a separate tenant. In both cases, the authority to approve business operations (workflows) resides within a tenant's space.

To allow different tenants to define their own custom workflows without editing configuration files, the API Manager provides configuration in tenant-specific locations in the registry, which you can access through the UI.

The topics below explain how to deploy a BPEL/human task using WSO2 BPS and how to point them to services deployed in the tenant spaces in the API Manager.

Deploying a BPEL and a HumanTask for a tenant

Only the users registered in the BPS can deploy BPELs and human tasks in it. Registration adds you to the user store in the BPS. In this guide, the API Manager and BPS use the same user store and all the users present in the BPS are visible to the API Manager as well. This is depicted by the diagram below:

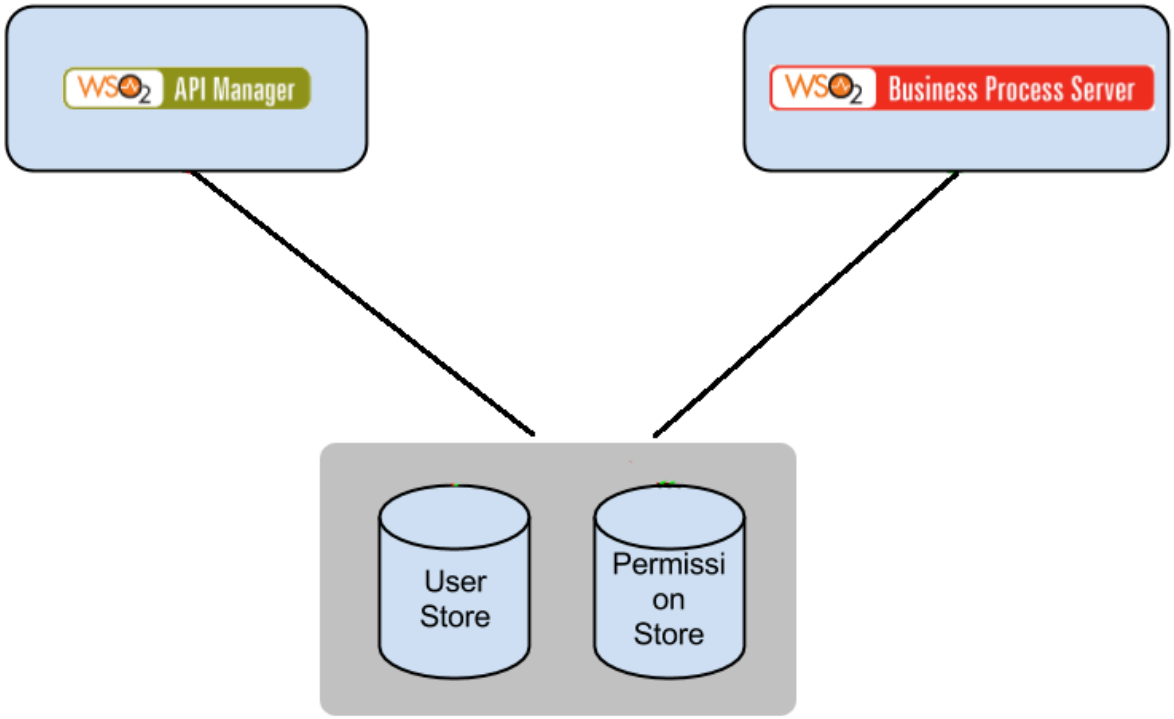


Figure: API Manager and BPS share the same user and permission store

Follow the steps below to deploy a BPEL and a human task for a tenant in the API Manager:

Sharing the user/permission stores with the BPS and API Manager

1. Create a database for the shared user store as follows:

```
mysql> create database workflow_ustore;
Query OK, 1 row affected (0.00 sec)
```

> **Tip**: Copy the database driver (in this case, the MySql driver) to the `/repository/components/lib` folder before starting each server.

2. Open the `<APIM_HOME>repository/conf/datasources/master-datasources.xml` and create a datasource pointing to the newly created database. For example,

```
<datasource>
    <name>USTORE</name>
    <description>The datasource used for API Manager database</description>
    <jndiConfig>
        <name>jdbc/ustore</name>
    </jndiConfig>
    <definition type="RDBMS">
        <configuration>

<url>jdbc:mysql://127.0.0.1:3306/workflow_ustore?autoReconnect=true&amp;relaxAuto
Commit=true</url>
            <username>root</username>
            <password>root</password>
            <driverClassName>com.mysql.jdbc.Driver</driverClassName>
            <maxActive>50</maxActive>
            <maxWait>60000</maxWait>
            <testOnBorrow>true</testOnBorrow>
            <validationQuery>SELECT 1</validationQuery>
            <validationInterval>30000</validationInterval>
        </configuration>
    </definition>
</datasource>
```

3. Repeat step 2 for the BPS as well.
4. Point the datasource name in `<APIM_HOME>repository/conf/user-mgt.xml` to the new datasource. (note that the user store is configured using the `<UserStoreManager>` element).

   In the following example, the same JDBC user store (that is shared by both the API Manager and the BPS) is used as the permission store as well:

```
<Configuration>
    <AddAdmin>true</AddAdmin>
    <AdminRole>admin</AdminRole>
        <AdminUser>
            <UserName>admin</UserName>
            <Password>admin</Password>
        </AdminUser>
    <EveryOneRoleName>everyone</EveryOneRoleName> <!-- By default users in this
role sees the registry root -->
    <Property name="dataSource">jdbc/ustore</Property>
</Configuration>
```

5. Repeat step 4 for the BPS as well.

Sharing the data in the registry with the BPS and API Manager

To deploy BPELs in an API Manager tenant space, the tenant space should be accessible by both the BPS and API Manager and certain tenant specific data such as key stores needs to be shared with both products. Follow the steps below to create a registry mount to share the data stored in the registry:

1. Create a separate database for the registry:

```
mysql> create database workflow_regdb;
Query OK, 1 row affected (0.00 sec)
```

2. Create a new datasource in `<APIM_HOME>repository/conf/datasources/master-datasources.xml` as done before:

```
<datasource>
    <name>REG_DB</name>
    <description>The datasource used for API Manager database</description>
    <jndiConfig>
        <name>jdbc/regdb</name>
    </jndiConfig>
    <definition type="RDBMS">
        <configuration>

<url>jdbc:mysql://127.0.0.1:3306/workflow_regdb?autoReconnect=true&amp;relaxAutoC
ommit=true</url>
            <username>root</username>
            <password>root</password>
            <driverClassName>com.mysql.jdbc.Driver</driverClassName>
            <maxActive>50</maxActive>
            <maxWait>60000</maxWait>
            <testOnBorrow>true</testOnBorrow>
            <validationQuery>SELECT 1</validationQuery>
            <validationInterval>30000</validationInterval>
        </configuration>
    </definition>
</datasource>
```

3. Add the following entries to `<APIM_HOME>/repository/conf/registry.xml`:

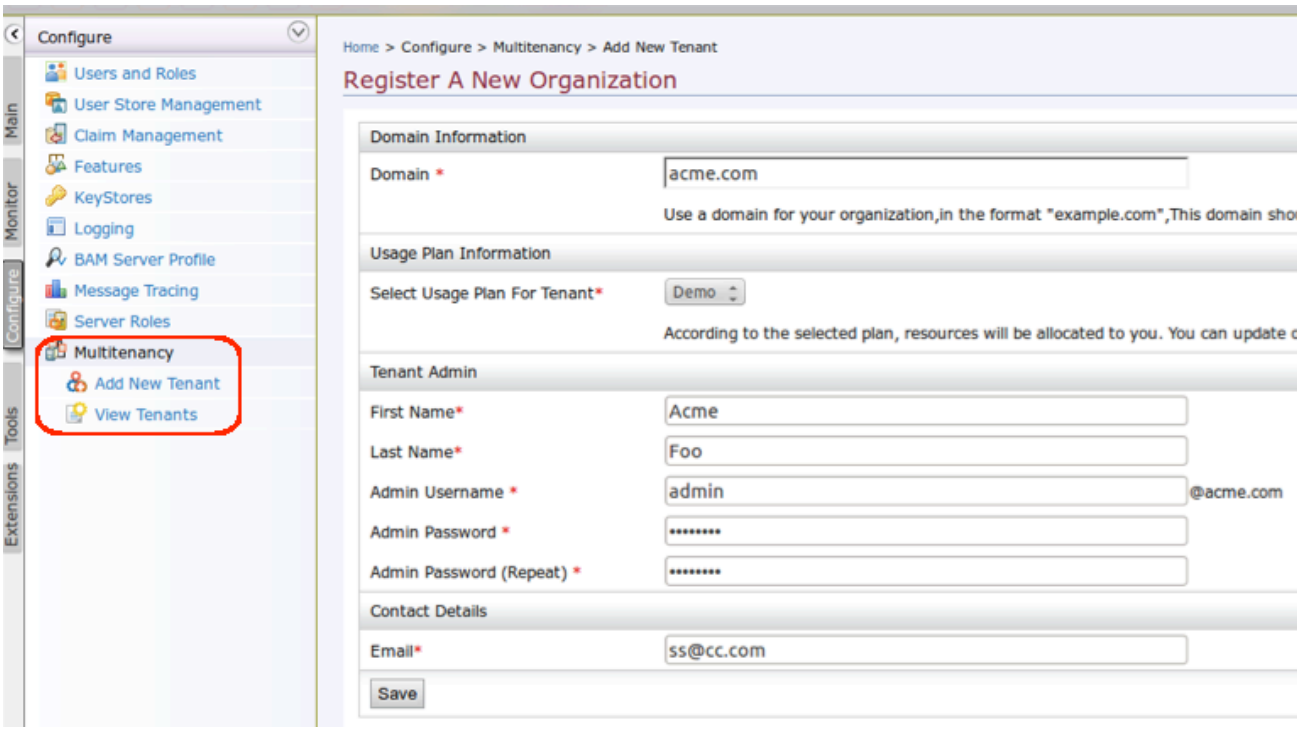
```
<dbConfig name="sharedregistry">
    <dataSource>jdbc/regdb</dataSource>
</dbConfig>

<remoteInstance url="https://localhost:9443/registry">
    <id>mount</id>
    <dbConfig>sharedregistry</dbConfig>
    <readOnly>false</readOnly>
    <enableCache>true</enableCache>
    <registryRoot>/</registryRoot>
</remoteInstance>

<!-- This defines the mount configuration to be used with the remote instance and
the target path for the mount -->
<mount path="/_system/config" overwrite="true">
    <instanceId>mount</instanceId>
    <targetPath>/_system/nodes</targetPath>
</mount>

<mount path="/_system/governance" overwrite="true">
    <instanceId>mount</instanceId>
    <targetPath>/_system/governance</targetPath>
</mount>

<mount path="/_system/governance/repository/security" overwrite="true">
    <instanceId>mountInstance</instanceId>
    <targetPath>/_system/governance/repository/security</targetPath>
</mount>
```

4. Repeat the above three steps for the BPS as well.

Creating a BPEL

In this section, you create a BPEL that has service endpoints pointing to services hosted in the tenant's space. This example uses the Application Creation Workflow.

1. Set a port offset of 2 to the BPS using the `<BPS_HOME>/repository/conf/carbon.xml` file. This prevents any port conflicts when you start more than one WSO2 products on the same server.
2. Log in to the API Manager's management console (`https://localhost:9443/carbon`) and create a tenant using the **Configure -> Multitenancy** menu.



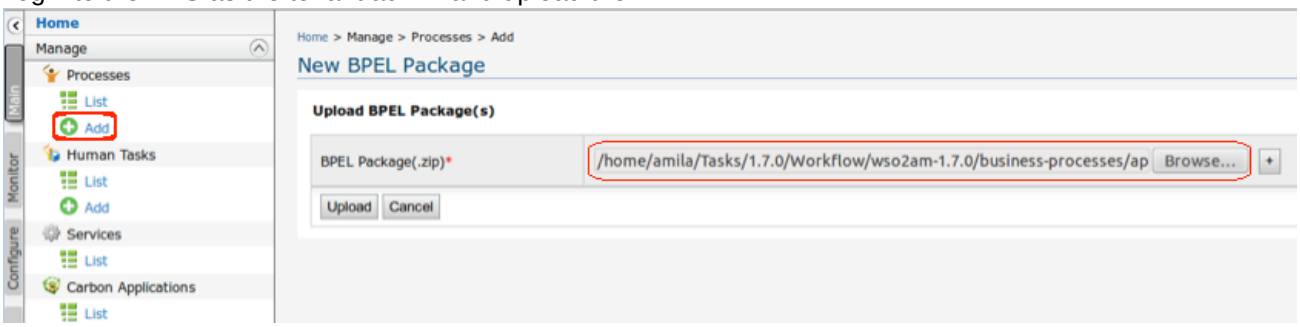
3. Create a copy of the BPEL located in `<APIM_HOME>/business-processes/application-creation/BPEL`.
4. Extract the contents of the new BPEL archive.
5. Copy the `ApplicationService.epr` and `ApplicationCallbackService.epr` to the extracted folder and rename them as `ApplicationService-Tenant.epr` and `ApplicationCallbackService-Tenant.epr` respectively.
6. Open `ApplicationService-Tenant.epr` and change the `wsa:Address` to `http://localhost:9765/services/t/<tenant domain>/ApplicationService`.
7. Point the `deploy.xml` file to the new .epr files provided in the BPEL archive. For example,

```
<invoke partnerLink="AAPL">
   <service name="applications:ApplicationService" port="ApplicationPort">
      <endpoint xmlns="http://wso2.org/bps/bpel/endpoint/config"
endpointReference="ApplicationService-Tenant.epr"></endpoint>
   </service>
</invoke>

<invoke partnerLink="CBPL">
   <service
name="callback.workflow.apimgt.carbon.wso2.org:WorkflowCallbackService"
port="WorkflowCallbackServiceHttpsSoap11Endpoint">
      <endpoint xmlns="http://wso2.org/bps/bpel/endpoint/config"
endpointReference="ApplicationCallbackService-Tenant.epr"></endpoint>
   </service>
</invoke>
```

8. Zip the content and create a BPEL archive in the following format:

```
ApplicationApprovalWorkFlowProcess_1.0.0-Tenant.zip
     |_ApplicationApprovalWorkFlowProcess.bpel
     |_ApplicationApprovalWorkFlowProcessArtifacts.wsdl
     |_ApplicationCallbackService-Tenant.epr
     |_ApplicationService-Tenant.epr
     |_ApplicationsApprovalTaskService.wsdl
     |_SecuredService-service.xml
     |_WorkflowCallbackService.wsdl
     |_deploy.xml
```

9. Log into the BPS as the tenant admin and upload the BPEL.



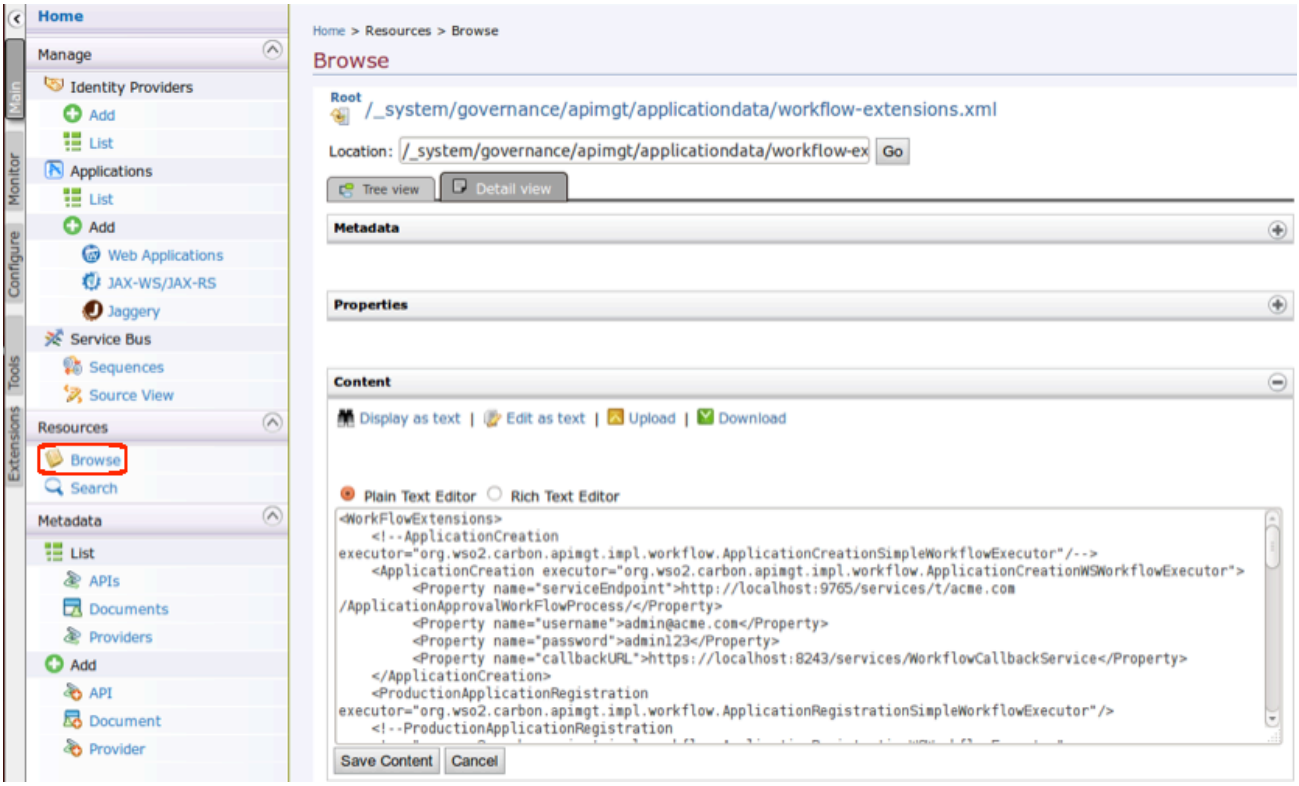
Creating a human task

Similar to creating a BPEL, create a HumaTask that has service endpoints pointing to services hosted in the tenant's space.

1. Create a copy of the HumanTask archive in `<APIM_HOME>/business-processes/application-creation/HumanTask` and extract its contents.
2. Edit the following section in `ApplicationApprovalTaskService.wsdl`:

```
<invoke partnerLink="AAPL">
   <service name="applications:ApplicationService" port="ApplicationPort">
         <endpoint xmlns="http://wso2.org/bps/bpel/endpoint/config"
endpointReference="ApplicationService-Tenant.epr"></endpoint>
   </service>
</invoke>

<invoke partnerLink="CBPL">
   <service
name="callback.workflow.apimgt.carbon.wso2.org:WorkflowCallbackService"
port="WorkflowCallbackServiceHttpsSoap11Endpoint">
         <endpoint xmlns="http://wso2.org/bps/bpel/endpoint/config"
endpointReference="ApplicationCallbackService-Tenant.epr"></endpoint>
   </service>
</invoke>
```

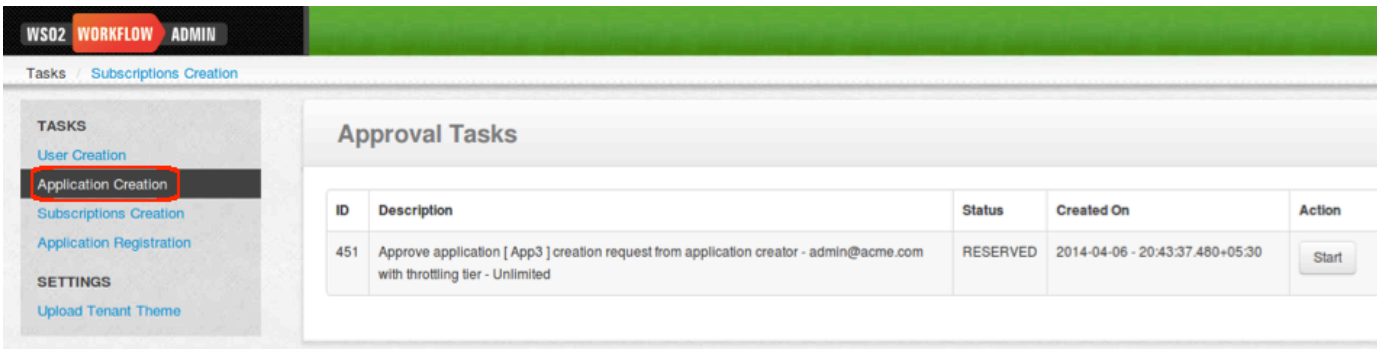
3. Create the HumanTask archive by zipping all the extracted files.
4. Log into the BPS as the tenant admin and upload the HumanTask.
5. Log into the API Manager's management console as the tenant admin and select **Resources > Browse** menu.
6. Go to the `/_system/governance/apimgt/applicationdata/workflow-extensions.xml` in the registry and change the service URL and the credentials of the `ApplicationCreationWSWorkflowExecutor`. For example,



> ⚠ Be sure to disable the `SimpleWorkflowExecutor` and enable the `ApplicationCreationWSWorkflowExecutor`.

Testing the workflow

You have now completed configuring the Application Creation workflow for a tenant. Whenever a tenant user logs in to the tenant store and create an application, the workflow will be invoked. You log in to the **Workflow Admin** Web application (`https://<Server Host>:9443/workflow-admin`) as the tenant admin and browse **Application Creation** menu to see all approval tasks have been created for newly created applications. For example,



Transforming API Message Payload

You can send API messages through the API Manager without any transformation configurations, if the back-end accepts messages of the same format. For example, the API Manager handles JSON to JSON transformations out of the box. In cases where the back-end does not accept the same format, the transformations are done as described below:

- JSON message builders and formatters
- XML representation of JSON payloads
- Converting a payload between XML and JSON

Also see the following sections in the WSO2 ESB documentation. WSO2 ESB is used to implement the API Gateway through which API messages are transformed:

- Accessing content from JSON payloads
- Logging JSON payloads
- Constructing and transforming JSON payloads
- Troubleshooting, debugging, and logging

JSON message builders and formatters

There are two types of message builders and formatters for JSON. The default builder and formatter keep the JSON representation intact without converting it to XML. You can access the payload content using JSON Path or XPath and convert the payload to XML at any point in the mediation flow.

- `org.apache.synapse.commons.json.JsonStreamBuilder`
- `org.apache.synapse.commons.json.JsonStreamFormatter`

If you want to convert the JSON representation to XML before the mediation flow begins, use the following builder and formatter instead. Note that some data loss can occur during the JSON to XML to JSON conversion process.

- `org.apache.synapse.commons.json.JsonBuilder`
- `org.apache.synapse.commons.json.JsonFormatter`

The builders and formatters are configured in the `messageBuilders` and `messageFormatters` sections, respectively, of the Axis2 configuration files located in the `<PRODUCT_HOME>/repository/conf/axis2` directory . Both types of JSON builders use StAXON as the underlying JSON processor.

The following builders and formatters are also included for compatibility with older API Manager versions:

- `org.apache.axis2.json.JSONBuilder/JSONMessageFormatter`
- `org.apache.axis2.json.JSONStreamBuilder/JSONStreamFormatter`
- `org.apache.axis2.json.JSONBadgerfishOMBuilder/JSONBadgerfishMessageFormatter`

> ⚠ Always use the same type of builder and formatter combination. Mixing different builders and formatters will cause errors at runtime.

If you want to handle JSON payloads that are sent using a media type other than `application/json`, you must register the JSON builder and formatter for that media type in the following two files at minimum (for best results, register them in all Axis2 configuration files found in the `<PRODUCT_HOME>/repository/conf/axis2` directory):

- `<PRODUCT_HOME>/repository/conf/axis2/axis2.xml`
- `<PRODUCT_HOME>/repository/conf/axis2/axis2_blocking_client.xml`

For example, if the media type is `text/javascript`, register the message builder and formatter as follows:

```
<messageBuilder contentType="text/javascript"
               class="org.apache.synapse.commons.json.JsonStreamBuilder"/>

<messageFormatter contentType="text/javascript"
                class="org.apache.synapse.commons.json.JsonStreamFormatter"/>
```

> ⚠ When you modify the builders/formatters in Axis2 configuration, make sure that you have enabled only one correct message builder/formatter pair for a given media type.

XML representation of JSON payloads

When building the XML tree, JSON builders attach the converted XML infoset to a special XML element that acts as the root element of the final XML tree. If the original JSON payload is of type `object`, the special element is `<json`

`Object/>`. If it is an `array`, the special element is `<jsonArray/>`. Following are examples of JSON and XML representations of various objects and arrays.

Null objects

JSON:

```
{"object":null}
```

XML:

```
<jsonObject>
    <object></object>
</jsonObject>
```

Empty objects

JSON:

```
{"object":{}}
```

XML:

```
<jsonObject>
   <object></object>
</jsonObject>
```

Empty strings

JSON:

```
{"object":""}
```

XML:

```
<jsonObject>
   <object></object>
</jsonObject>
```

Empty array

JSON:

```
[]
```

XML (JsonStreamBuilder):

```
<jsonArray></jsonArray>
```

XML (JsonBuilder):

```
<jsonArray>
   <?xml-multiple jsonElement?>
</jsonArray>
```

Named arrays

JSON:

```
{"array":[1,2]}
```

XML (JsonStreamBuilder):

```
<jsonObject>
   <array>1</array>
   <array>2</array>
</jsonObject>
```

XML (JsonBuilder):

```
<jsonObject>
   <?xml-multiple array?>
   <array>1</array>
   <array>2</array>
</jsonObject>
```

JSON:

```
{"array":[]}
```

XML (JsonStreamBuilder):

```
<jsonObject></jsonObject>
```

XML (JsonBuilder):

```
<jsonObject>
   <?xml-multiple array?>
</jsonObject>
```

Anonymous arrays

JSON:

```
[1,2]
```

XML (JsonStreamBuilder):

```
<jsonArray>
   <jsonElement>1</jsonElement>
   <jsonElement>2</jsonElement>
</jsonArray>
```

XML (JsonBuilder):

```
<jsonArray>
   <?xml-multiple jsonElement?>
   <jsonElement>1</jsonElement>
   <jsonElement>2</jsonElement>
</jsonArray>
```

JSON:

```
[1, []]
```

XML (JsonStreamBuilder):

```
<jsonArray>
   <jsonElement>1</jsonElement>
   <jsonElement>
       <jsonArray></jsonArray>
   </jsonElement>
</jsonArray>
```

XML (JsonBuilder):

```
<jsonArray>
   <?xml-multiple jsonElement?>
   <jsonElement>1</jsonElement>
   <jsonElement>
       <jsonArray>
           <?xml-multiple jsonElement?>
       </jsonArray>
   </jsonElement>
</jsonArray>
```

XML processing instructions (PIs)

Note that the addition of `xml-multiple` processing instructions to the XML payloads whose JSON representations contain arrays. `JsonBuilder` (via StAXON) adds these instructions to the XML payload that it builds during the JSON to XML conversion so that during the XML to JSON conversion, `JsonFormatter` can reconstruct the arrays

that are present in the original JSON payload. `JsonFormatter` interprets the elements immediately following a processing instruction to construct an array.

Special characters

When building XML elements, the '$' character and digits are handled in a special manner when they appear as the first character of a JSON key. Following are examples of two such occurrences. Note the addition of the `_JsonReader_PS_` and `_JsonReader_PD_` prefixes in place of the '$' and digit characters, respectively.

JSON:

```
{"$key":1234}
```

XML:

```
<jsonObject>
   <_JsonReader_PS_key>1234</_JsonReader_PS_key>
</jsonObject>
```

JSON:

```
{"32X32":"image_32x32.png"}
```

XML:

```
<jsonObject>
   <_JsonReader_PD_32X32>image_32x32.png</_JsonReader_PD_32X32>
</jsonObject>
```

Converting a payload between XML and JSON

To convert an XML payload to JSON, set the `messageType` property to `application/json` in the axis2 scope before sending message to an endpoint. Similarly, to convert a JSON payload to XML, set the `messageType` property to `application/xml` or `text/xml`. For example:

```
<api name="admin--TOJSON" context="/tojson" version="1.0" version-type="url">
        <resource methods="POST GET DELETE OPTIONS PUT" url-mapping="/*">
            <inSequence>
                <property name="POST_TO_URI" value="true" scope="axis2"/>
                <property name="messageType" value="application/json" scope="axis2"/>
                <filter source="$ctx:AM_KEY_TYPE" regex="PRODUCTION">
                    <then>
                        <send>
                            <endpoint name="admin--Test_APIproductionEndpoint_0">
                                <http
uri-template="http://localhost:9767/services/StudentService">
                                    <timeout>
                                        <duration>30000</duration>
                                        <responseAction>fault</responseAction>
                                    </timeout>
                                    <suspendOnFailure>
                                        <errorCodes>-1</errorCodes>
                                        <initialDuration>0</initialDuration>
                                        <progressionFactor>1.0</progressionFactor>
                                        <maximumDuration>0</maximumDuration>
                                    </suspendOnFailure>
                                    <markForSuspension>
                                        <errorCodes>-1</errorCodes>
                                    </markForSuspension>
                                </http>
                            </endpoint>
                        </send>
                    </then>
                    <else>
                        <sequence key="_sandbox_key_error_"/>
                    </else>
                </filter>
            </inSequence>
            <outSequence>
                <send/>
            </outSequence>
        </resource>
        <handlers>
            <handler
class="org.wso2.carbon.apimgt.gateway.handlers.security.APIAuthenticationHandler"/>
            <handler
class="org.wso2.carbon.apimgt.gateway.handlers.throttling.APIThrottleHandler">
                <property name="id" value="A"/>
                <property name="policyKey"
value="gov:/apimgt/applicationdata/tiers.xml"/>
            </handler>
            <handler
class="org.wso2.carbon.apimgt.usage.publisher.APIMgtUsageHandler"/>
            <handler
class="org.wso2.carbon.apimgt.usage.publisher.APIMgtGoogleAnalyticsTrackingHandler"/>
            <handler
class="org.wso2.carbon.apimgt.gateway.handlers.ext.APIManagerExtensionHandler"/>
        </handlers>
    </api>
```

An example command to invoke above API:

```
curl -v -X POST -H "Content-Type:application/xml" -H "Authorization: Bearer xxx"
-d@request1.xml "http://10.100.1.110:8280/tojson/1.0"
```

If the request payload is as follows:

```
<coordinates>
   <location>
       <name>Bermuda Triangle</name>
       <n>25.0000</n>
       <w>71.0000</w>
   </location>
   <location>
       <name>Eiffel Tower</name>
       <n>48.8582</n>
       <e>2.2945</e>
   </location>
</coordinates>
```

The response payload will look like this:

```
{
  "coordinates":{
     "location":[
        {
           "name":"Bermuda Triangle",
           "n":25.0000,
           "w":71.0000
        },
        {
           "name":"Eiffel Tower",
           "n":48.8582,
           "e":2.2945
        }
     ]
  }
}
```

Note that we have used the Property mediator to mark the outgoing payload to be formatted as JSON. For more information about the Property Mediator, see the Property Mediator page on WSO2 ESB documentation.

```
<property name="messageType" value="application/json" scope="axis2"/>
```

Similarly if the response message needs to be transformed, set the messageType property in the outSequence.

```
<api name="admin--TOJSON" context="/tojson" version="1.0" version-type="url">
        <resource methods="POST GET DELETE OPTIONS PUT" url-mapping="/*">
            <inSequence>
                <property name="POST_TO_URI" value="true" scope="axis2"/>
                <filter source="$ctx:AM_KEY_TYPE" regex="PRODUCTION">
                    <then>
                        <send>
                            <endpoint name="admin--Test_APIproductionEndpoint_0">
                                <http
uri-template="http://localhost:9767/services/StudentService">
                                    <timeout>
                                        <duration>30000</duration>
                                        <responseAction>fault</responseAction>
                                    </timeout>
                                    <suspendOnFailure>
                                        <errorCodes>-1</errorCodes>
                                        <initialDuration>0</initialDuration>
                                        <progressionFactor>1.0</progressionFactor>
                                        <maximumDuration>0</maximumDuration>
                                    </suspendOnFailure>
                                    <markForSuspension>
                                        <errorCodes>-1</errorCodes>
                                    </markForSuspension>
                                </http>
                            </endpoint>
                        </send>
                    </then>
                    <else>
                        <sequence key="_sandbox_key_error_"/>
                    </else>
                </filter>
            </inSequence>
            <outSequence>
                <property name="messageType" value="application/json" scope="axis2"/>
                <send/>
            </outSequence>
        </resource>
        <handlers>
            <handler
class="org.wso2.carbon.apimgt.gateway.handlers.security.APIAuthenticationHandler"/>
            <handler
class="org.wso2.carbon.apimgt.gateway.handlers.throttling.APIThrottleHandler">
                <property name="id" value="A"/>
                <property name="policyKey"
value="gov:/apimgt/applicationdata/tiers.xml"/>
            </handler>
            <handler
class="org.wso2.carbon.apimgt.usage.publisher.APIMgtUsageHandler"/>
            <handler
class="org.wso2.carbon.apimgt.usage.publisher.APIMgtGoogleAnalyticsTrackingHandler"/>
            <handler
class="org.wso2.carbon.apimgt.gateway.handlers.ext.APIManagerExtensionHandler"/>
        </handlers>
    </api>
```

Customizing the Management Console

The management console user interface (`https://localhost:9443/carbon`) of a Carbon product consists of two layers:

1. **UI inherited from WSO2 Carbon platform** contains the templates, styles (css files), and images that are stored in the core Carbon UI bundle stored in `<PRODUCT_HOME>/repository/components/plugins/org.wso2.carbon.ui_<version-number>.jar` where `<version-number>` is the version of the Carbon kernel that the product is built on. This bundle is responsible for the overall look and feel of the entire Carbon platform.
2. **UI unique to each product** contains all the styles and images that override the ones in core Carbon platform. This file is in `<PRODUCT_HOME>/repository/components/plugins/org.wso2.<product-name>.styles_<version-number>.jar` where `<version-number>` is the version of the product.

The following topics explain how to download a Carbon product and customize its user interface.

- Setting up the development environment
- Customizing the user interface
- Starting the server

Setting up the development environment

To download and set up the product environment for editing, take the following steps.

1. Download your product.
2. Extract the ZIP file into a separate folder in your hard drive.
3. Go to the `<PRODUCT_HOME>/repository/components/plugins/` directory to find the required JAR files:
   - `org.wso2.carbon.ui_<version-number>.jar`
   - `org.wso2.<product-name>.styles_<version-number>.jar`
4. Copy the JAR files to a separate location on your hard drive. Since the JAR files are zipped, you must unzip them to make them editable.

You can now customize the look and feel of your product by modifying the contents of the JAR files as described in the next section.

Customizing the user interface

Customizing the product interface involves changing the layout/design of the Carbon framework as well as changing the styles and images specific to the product. The following topics explain how some of the main changes to the product interface can be done.

- Changing the layout
- Changing the styles on the Carbon framework
- Changing the product specific styles and images

Changing the layout

The layout of the Carbon framework is built using a tiles JSP tag library. The use of tiles allows us to break the presentation of the layout into small JSP snippets that perform a specific function. For example, `header.jsp` and `footer.jsp` are the tiles corresponding to the header and footer in the layout. The `template.jsp` file controls the main layout page of the Carbon framework, which holds all the tiles together. That is, the header part in the `template.jsp` file is replaced with the `<tiles:insertAttribute name="header"/>` tag, which refers to the `header.jsp` file. The `template.jsp` file as well as the JSP files corresponding to the tiles are located in the `org.wso2.<product-name>.styles_<version-name>.jar/web/admin/layout/` directory.

Therefore, changing the layout of your product primarily involves changing the `template.jsp` page (main layout page) and the JSP files of the relevant JSP tiles.

> ⚠ Ensure that you do not change or remove the ID attributes on the `.jsp` files.

Changing the styles on the Carbon framework

The `global.css` file, which determines the styles of the Carbon framework, is located in the `org.wso2.carbon.ui_<version-name>.jar/web/admin/css/` directory. You can edit this file as per your requirement. Alternatively, you can apply a completely new stylesheet to your framework instead of the default `global.css` stylesheet.

To apply a new style sheet to the carbon framework:

1. Copy your new CSS file to this same location.
2. Open the `template.jsp` file located in the `org.wso2.carbon.ui_<version-name>.jar/web/admin/layout/` directory, which contains the main layout of the page and the default JavaScript libraries.
3. Replace `global.css` with the new style sheet by pointing the `String globalCSS` attribute to the new stylesheet file.

```
//Customization of UI theming per tenant
 String tenantDomain = null;
 String globalCSS = "../admin/css/global.css";
 String mainCSS = "";
```

Changing the product specific styles and images

The styles and images unique to your product is location in the `org.wso2.<product-name>.styles_<version-number>.jar` folder. To modify product specific styles and images, take the following steps.

1. Copy the necessary images to the `org.wso2.<product-name>.styles_<version-number>.jar/web/styles/images/` directory. For example, if you want to change the product banner, add the new image file to this directory.
2. Open the `main.css` file located in the `org.wso2.<product-name>.styles_<version-name>.jar/web/styles/css/` directory.
3. To specify a new product banner, change the `background-image` attribute of `org.wso2.<product-name>.styles_<version-name>.jar/web/styles/css/main.css` file as follows:

```
/* ---------------- header styles ------------------ */
div#header-div {
    background-image: url( ../images/newproduct-header-bg.png);
 height:70px;
}
```

Note that the size of the images you use will affect the overall UI of your product. For example, if the height of the product logo image exceeds 28 pixels, you must adjust the `main.css` file in the `org.wso2.<product-name>.styles_<version-name>.jar/web/styles/css/` directory to ensure that the other UI elements of your product aligns with the product logo.

Starting the server

In the preceding steps, you have done the changes to the product interface after copying the JAR files to a separate location on your hard drive. Therefore, before you start your production server, these files must be correctly copied back to your production environment as explained below.

1. Compress the contents of the `org.wso2.carbon.ui_<version-number>.jar` and `org.wso2.<product-name>.styles_<product-version>.jar` folders into separate ZIP files.
2. Change the name of the ZIP file to `org.wso2.carbon.ui_<version-number>.jar` and `org.wso2.<pr`

oduct-name>.styles_<version-number>.jar respectively.
3. Copy these two new JAR files to the <PRODUCT_HOME> /repository/components/plugins/ directory in your product installation.
4. Start the server.

Writing Test Cases

You can use WSO2 Test Automation Framework (TAF) to write automated test scripts for the API Manager. For an example, see Writing a Test Case for API Manager in TAF documentation.

Working with Security

WSO2 API Manager provides many methods for implementing and managing security, as described in the following topics:

- Passing Enduser Attributes to the Backend Using JWT
- Saving Access Tokens in Separate Tables
- Fixing Security Vulnerabilities
- Encrypting Passwords

Passing Enduser Attributes to the Backend Using JWT

To authenticate endusers, the API Manager passes attributes of the API invoker to the backend API implementation. **JSON Web Token (JWT)** is used to represent claims that are transferred between the enduser and the backend. A claim is an attribute of the user that is mapped to the underlying user store. A set of claims is called a dialect (e.g., h ttp://wso2.org/claims). The general format of a JWT is {token infor}.{claims list}.{signature}. The API implementation uses information such as logging, content filtering and authentication/authorization that is stored in this token. The token is Base64-encoded and sent to the API implementation in a HTTP header variable. F or more information on JWT, look here.

An example of a JWT passed in the API Manager is given below:

```
{
    "typ":"JWT",
    "alg":"NONE"
 }{
    "iss":"wso2.org/products/am",
    "exp":1345183492181,
    "http://wso2.org/claims/subscriber":"admin",
    "http://wso2.org/claims/applicationname":"app2",
    "http://wso2.org/claims/apicontext":"/placeFinder",
    "http://wso2.org/claims/version":"1.0.0",
    "http://wso2.org/claims/tier":"Silver",
    "http://wso2.org/claims/enduser":"sumedha"
 }
```

The above token contains,

- Token expiration time ("exp")
- Subscriber to the API, usually the app developer ("http://wso2.org/claims/subscriber")
- Application through which API invocation is done ("http://wso2.org/claims/applicationname")
- Context of the API ("http://wso2.org/claims/apicontext")
- API version ("http://wso2.org/claims/version")
- Tier/price band for the subscription ("http://wso2.org/claims/tier")
- Enduser of the app who's action invoked the API ("http://wso2.org/claims/enduser")

Configuring JWT

Given below is how to configure JWT generation in the API Manager.

1. Open `<APIM_HOME>/repository/conf/api-manager.xml` file and enable JWT as follows.

```
<EnableTokenGeneration>true</EnableTokenGeneration>
```

> ⚠ If you publish APIs before JWT is enabled, you have to republish them to include JWT.

2. Configure the rest of the elements in the same XML file as described in the table below. If you do not specify values to the elements, the default values will be applied.

Element	Description
<SecurityContextHeader/>	The name of the HTTP header to which the JWT is attached.
<ClaimsRetrieverImplClass/>	By default, the following are encoded to the JWT: • subscriber name • application name • API context • API version • authorised resource owner name In addition, you can also write your own class by extending the interface c methods in this interface are described below: **Method** / **Description** void init() throws APIManagementException; — Used to perform initialization tasks. Is execute SortedMap<String,String> getClaims(String endUserName) throws APIManagementException; — Returns a sorted map of claims. The key of th corresponding user attribute value. The order ordering defined by the sorted map. String getDialectURI(String endUserName); — The dialect URI to which the attribute names r example, if the `getClaims` method returns `{email:u` urns `http://wso2.org/claims`, the JWT `2.com”,“http://wso2.org/claims/ema` The default implementation (`org.wso2.carb` the user's attributes defined under the dialect with the same dialect URI. The order of encoc value is specified, no additional claims will be
<ConsumerDialectURI/>	The dialect URI under which the user's claims are be looked for. Only works defined above. JWT token contains all claims define in the `<ConsumerDialectURI>` eleme To get a list of users to be included in the JWT, simply uncomment this eleme `/claims` to the JWT token.

<SignatureAlgorithm/>	The signing algorithm used to sign the JWT. The general format of the JWT i is specified as the algorithm, signing is turned off and the JWT looks as `{tok` and a period at the end. This element can have only two values - the default value (SHA256WITHRS/

In a multi-tenanted setup with JWT token generation enabled, if a user who is in a secondary user store tries to invoke an API published within the same tenant store, you get an error. This issue is fixed from 1.8.0 version onwards.

Saving Access Tokens in Separate Tables

You can configure the API Manager instances to store access tokens in different tables according to their user store domain. This is referred to as **user token partitioning** and it ensures better security when there are multiple user stores configured in the system. For information on configuring user stores other than the default one, see Configuring Secondary User Stores. To enable user token partitioning, you should change the `<EnableAssertions>` and `<AccessTokenPartitioning>` elements in <APIM_HOME>/repository/conf/identity.xml file.

<EnableAssertions>

Assertions are used to embed parameters into tokens in order to generate a strong access token. You can also use these parameters later for various other processing functionality. At the moment, API Manager only supports UserName as an assertion.

By default, assertions are set to `false` in <APIM_HOME>/repository/conf/identity.xml.

```
<EnableAssertions>
        <UserName>false</UserName>
</EnableAssertions>
```

You can make it true by setting `<UserName>` element to `true`. You can add a user name to an access token when generating the key, and verify it by Base64-decoding the retrieved access token.

<AccessTokenPartitioning>

This parameter implies whether you need to store the keys in different tables or not. It can be used only if `<UserName>` assertion is enabled. If it is, set the `<EnableAccessTokenPartitioning>` element to true in <APIM_HOME>/repository/conf/identity.xml to store the keys in different tables.

```
<EnableAccessTokenPartitioning>true</EnableAccessTokenPartitioning>
```

Also set the user store domain names and mappings to new table names. For example,

- if userId = foo.com/admin where 'foo.com' is the user store domain name, then a 'mapping:domain' combo can be defined as 'A:foo.com'.
- 'A' is the mapping for the table that stores tokens relevant to users coming from 'foo.com' user store.

In this case, the actual table name is 'IDN_OAUTH2_ACCESS_TOKEN_A'. We use a mapping simply to prevent any issues caused by lengthy tables names when lengthy domain names are used. You need to manually create the tables you are going to use to store the access tokens in each user-store (i.e., tables 'IDN_OAUTH2_ACCESS_TOKEN_A' and 'IDN_OAUTH2_ACCESS_TOKEN_B' should be manually created according to the following defined domain mapping). This table structure is similar to the 'IDN_OAUTH2_ACCESS_TOKEN' table defined in api-manager dbscript, which is inside <AMIM_HOME>/dbscripts/apimgt directory.

You can provide multiple mappings separated by commas as follows. Note that the domain names need to be specified in upper case.

```
<AccessTokenPartitioningDomains>A:FOO.COM, B:BAR.COM</AccessTokenPartitioningDomains>
```

According to the information given above, change the `<APIKeyManager>` element in the identity.xml file as shown in the following example:

identity.xml

```
<!-- Assertions can be used to embedd parameters into access token.-->
<EnableAssertions>
     <UserName>false</UserName>
</EnableAssertions>

<!-- This should be set to true when using multiple user stores and keys should saved
into different tables according to the user store. By default all the application keys
are saved in to the same table. UserName Assertion should be 'true' to use this.-->
<AccessTokenPartitioning>
     <EnableAccessTokenPartitioning>false</EnableAccessTokenPartitioning>
     <!-- user store domain names and mappings to new table names. eg: if you provide
'A:foo.com', foo.com should be the user store domain
     name and 'A' represent the relavant mapping of token storing table, i.e. tokens
relevant to the users comming from foo.com user store
     will be added to a table called IDN_OAUTH2_ACCESS_TOKEN_A. -->
     <AccessTokenPartitioningDomains><!-- A:foo.com, B:bar.com
--></AccessTokenPartitioningDomains>
</AccessTokenPartitioning>
```

Fixing Security Vulnerabilities

A cipher is an algorithm for performing encryption or decryption. You can disable the weak ciphers in the Tomcat server by modifying the `cipher` attribute in the SSL Connector container, which is in the `catalina-server.xml` f ile. Enter the ciphers that you want your server to support in a comma-separated list. By default, all ciphers, whether they are strong or weak, will be enabled. However, if you do not add the `cipher` attribute or keep it blank, all SSL ciphers by JSSE will be supported by your server. This will enable the weak ciphers.

The steps below explain how to disable weak and enable strong ciphers in a product:

1. Take a backup of `<PRODUCT_HOME>/repository/conf/tomcat/catalina-server.xml` file.
2. Stop the server.
3. Add the `cipher` attribute to the existing configuration in the `catalina-server.xml` file with the list of ciphers that you want your server to support as follows:

   ```
   ciphers="<cipher-name>,<cipher-name>"
   ```

   The code below shows how a connector looks after an example configuration is done:

```
<Connector protocol="org.apache.coyote.http11.Http11NioProtocol"
                port="9443"
                bindOnInit="false"
                sslProtocol="TLS"
                maxHttpHeaderSize="8192"
                acceptorThreadCount="2"
                maxThreads="250"
                minSpareThreads="50"
                disableUploadTimeout="false"
                enableLookups="false"
                connectionUploadTimeout="120000"
                maxKeepAliveRequests="200"
                acceptCount="200"
                server="WSO2 Carbon Server"
                clientAuth="false"
                compression="on"
                scheme="https"
                secure="true"
                SSLEnabled="true"
                compressionMinSize="2048"
                noCompressionUserAgents="gozilla, traviata"
                compressableMimeType="text/html,text/javascript,application/x-


javascript,application/javascript,application/xml,text/css,application/xslt+xml,
                text/xsl,image/gif,image/jpg,image/jpeg"

ciphers="SSL_RSA_WITH_RC4_128_MD5,SSL_RSA_WITH_RC4_128_SHA,TLS_RSA_WITH_AES_128_C
BC_SHA,

TLS_DHE_RSA_WITH_AES_128_CBC_SHA,TLS_DHE_DSS_WITH_AES_128_CBC_SHA,SSL_RSA_WITH_3D
ES_EDE_CBC_SHA,

SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA,SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA"

keystoreFile="${carbon.home}/repository/resources/security/wso2carbon.jks"
                keystorePass="wso2carbon"
                URIEncoding="UTF-8"/>
```

4. Save the `catalina-server.xml` file.
5. Restart the server.

Encrypting Passwords

Encrypting passwords provides better security and less vulnerability to security attacks than saving passwords in plain text. It is recommended in a production setup. WSO2 API Manager provides a secure vault implementation that encrypts passwords, stores them in the registry, maps them to aliases and uses the alias instead of the actual passwords in configuration files. At runtime, the API Manager looks up aliases and decrypts the passwords. The secure vault is unable to encrypt the passwords of registry resources at the moment.

The steps below explain how to encrypt passwords in different contexts:

- Encrypting passwords in configuration files
- Encrypting secure endpoint passwords

Encrypting passwords in configuration files

1. Shutdown the server if it is already running and open `<AM_HOME>/repository/conf/security/cipher`

`-tool.properties` file. It contains all the aliases to different server components.

2. Uncomment the entries you want to encrypt. If you want to secure an additional property, add it to the end of the file as alias name and the value where the value is `file name/xpath`.

```
transports.https.keystorePass=mgt-transports.xml//transports/transport[@name='htt
ps']/parameter[@name='keystorePass'],false
Carbon.Security.KeyStore.Password=carbon.xml//Server/Security/KeyStore/Password,t
rue
Carbon.Security.KeyStore.KeyPassword=carbon.xml//Server/Security/KeyStore/KeyPass
word,true
Carbon.Security.TrustStore.Password=carbon.xml//Server/Security/TrustStore/Passwo
rd,true
UserManager.AdminUser.Password=user-mgt.xml//UserManager/Realm/Configuration/Admi
nUser/Password,true
Datasources.WSO2_CARBON_DB.Configuration.Password=master-datasources.xml//datasou
rces-configuration/datasources/datasource[name='WSO2_CARBON_DB']/definition[@type
='RDBMS']/configuration/password,false
#Datasource.WSO2AM_DB.configuration.password=master-datasources.xml//datasources-
configuration/datasources/datasource[name='WSO2AM_DB']/definition[@type='RDBMS']/
configuration/password,false
#Datasource.WSO2AM_STATS_DB.configuration.password=master-datasources.xml//dataso
urces-configuration/datasources/datasource[name='WSO2AM_STATS_DB']/definition[@ty
pe='RDBMS']/configuration/password,false
#UserStoreManager.Property.ConnectionPassword=user-mgt.xml//UserManager/Realm/Use
rStoreManager/Property[@name='ConnectionPassword'],true
#UserStoreManager.Property.password=user-mgt.xml//UserManager/Realm/UserStoreMana
ger/Property[@name='password'],true
#AuthManager.Password=api-manager.xml//APIManager/AuthManager/Password,true

...
```

3. Run the cipher tool available in `<APIM_HOME>/bin`. If on windows, the file is `ciphertool.bat`. If you are using the default keystore, give **`wso2carbon`** as the primary keystore password when prompted.

```
sh ciphertool.sh -Dconfigure
```

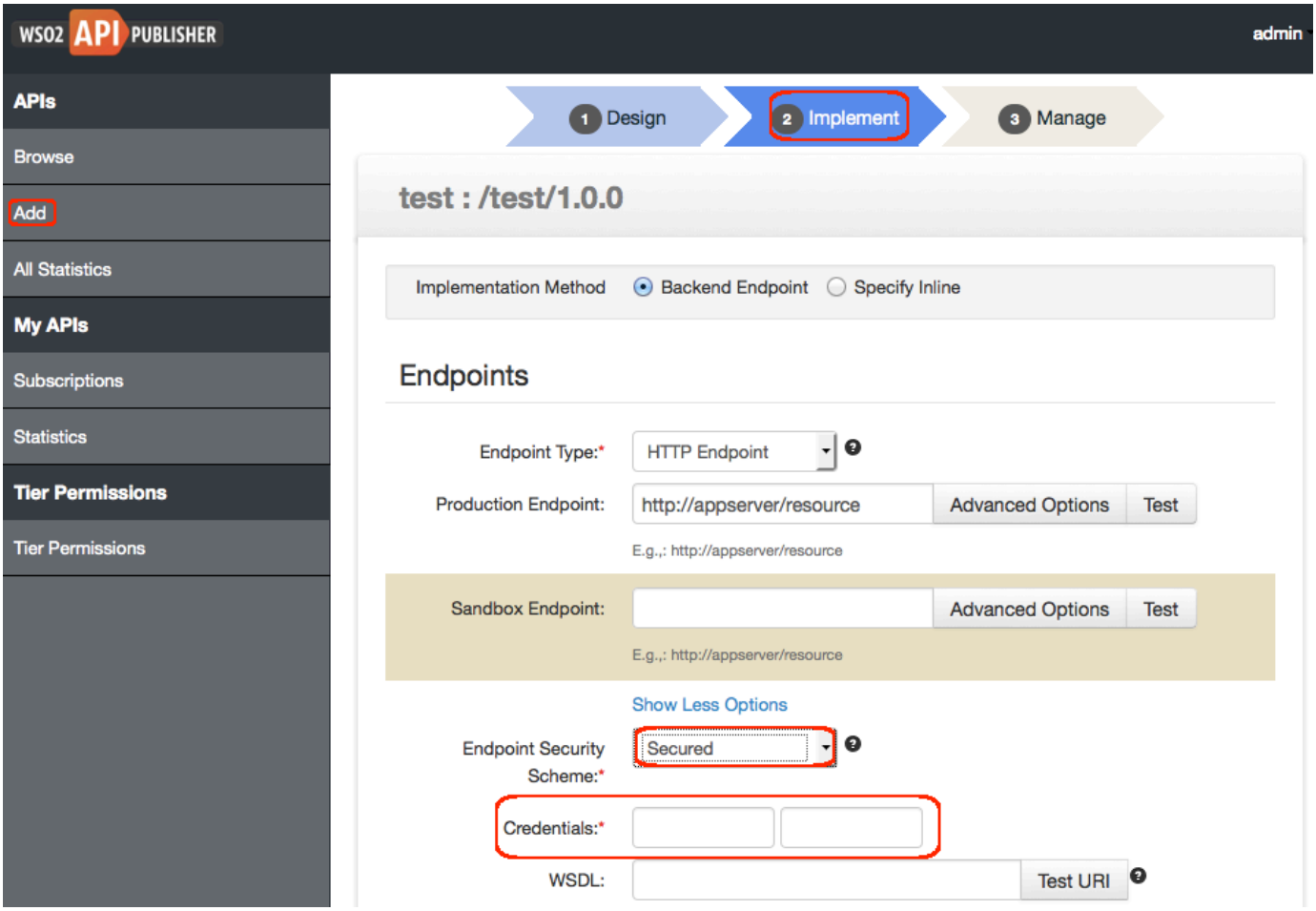
4. Note that the cipher tool creates an encrypted password and uses the alias name in places where the `plain-text password is used in configuration files.` For example, as the `Carbon.Security.KeyStore.Password` property is uncommented, after you run the cipher tool, the plain-text password will be replaced by the alias name in `<APIM_HOME>/repository/conf/carbon.xml` file as follows.

```
<KeyStore>
...
    <!-- Keystore password-->
    <Password
svns:secretAlias="Carbon.Security.KeyStore.Password">password</Password>
...
</KeyStore>
```

5. Note that after the above steps, you are prompted to enter the primary keystore password every time you start the API Manager.

Encrypting secure endpoint passwords

When creating an API using the API Publisher, you specify the endpoint of its backend implementation in the **Implement** tab. If you select the endpoint as secured, you are prompted to give credentials in plain-text.



The steps below show how to secure the endpoint's password that is given in plain-text in the UI.

1. Shutdown the server if it is already running and set the element `<EnableSecureVault>` in `<APIM_HOME>/repository/conf/api-manager.xml` to `true`. By default, the system stores passwords in configuration files in plain text because this values is set to `false`.
2. Define synapse property in the synapse.properties file as follows: `synapse.xpath.func.extensions=org.wso2.carbon.mediation.security.vault.xpath.SecureVaultLookupXPathFunctionProvider.`
3. Run the cipher tool available in `<APIM_HOME>/bin`. If on windows, the file is `ciphertool.bat`. If you are using the default keystore, give **`wso2carbon`** as the primary keystore password when prompted.

```
sh ciphertool.sh -Dconfigure
```

Tip: See Fixing Security Vulnerabilities for information on configuring cipher at the Tomcat level.

Admin Guide

The following topics explore various product deployment scenarios and other topics useful for system administrators.

Managing Users and Roles

This chapter contains the following information:

User Roles in the API Manager

Roles contain permissions for users to manage the server. You can create different roles with various combinations of permissions and assign them to a user or a group of users. User roles can be reused throughout the system and prevent the overhead of granting multiple permissions to each and every user individually. Through the Management Console, you can also edit and delete an existing user role.

WSO2 API Manager allows you to log in to the Management Console as an admin user, and create custom roles with different levels of permission. These roles can then be assigned to different users according to your requirement. We identify four distinct user roles that are typically used in many organizational situations:

- **Admin** : Admin is the API management provider, who hosts and manages the API Gateway. S/he is responsible for creating user roles in the system, assign users to roles, managing databases, security etc. Also see the Admin Guide. The Admin role is available by default with credentials admin/admin.
- **creator**: A creator is typically a person in a technical role who understands the technical aspects of the API (coding, interfaces, documentation, versions, how it is exposed by API gateway) and uses the API Publisher Web application to develop and provision APIs into the API store. The creator uses the API store to consult ratings and feedback provided by API consumers. Creator can add APIs to the store but cannot manage their lifecycles (that is, make them visible to the outside world).
- **publisher**: A publisher is typically a person in a managerial role and overlooks a set of APIs across the enterprise or a business unit, and controls the API lifecycle and monetization aspects. The publisher also analyzes usage patterns for APIs and has access to all API statistics.
- **consumer** : A consumer is typically an anonymous user or an application developer who searches the API store to discover APIs and use them. He/she reads the documentation, forums, rates/comments on APIs.

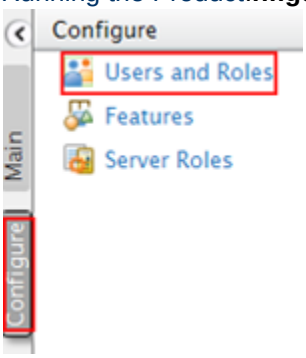
> We have identified the three roles above as common in many organizational situations. They are used throughout this documentation. However, you can also define different user roles depending on your unique requirements.

Administrators of the API Manager can use the Management Console UI to add user roles. Roles contain different

levels of permissions to manage the Server. You can create different roles with various combinations of permissions. Follow the instructions below to create the `creator`, `publisher` and `subscriber` roles.

Creating user roles

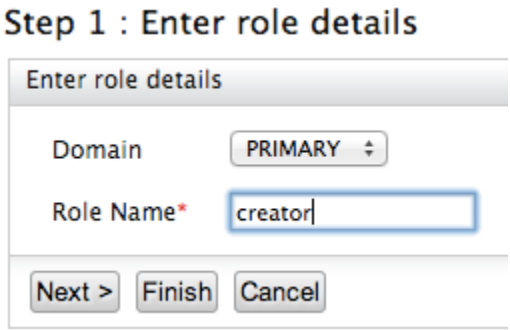
1. Log in to the Management Console (https://localhost:9443/carbon) and select **Users and Roles** under the **Co** Running the Product.**nfigure** menu. For instructions on accessing the Management Console, see



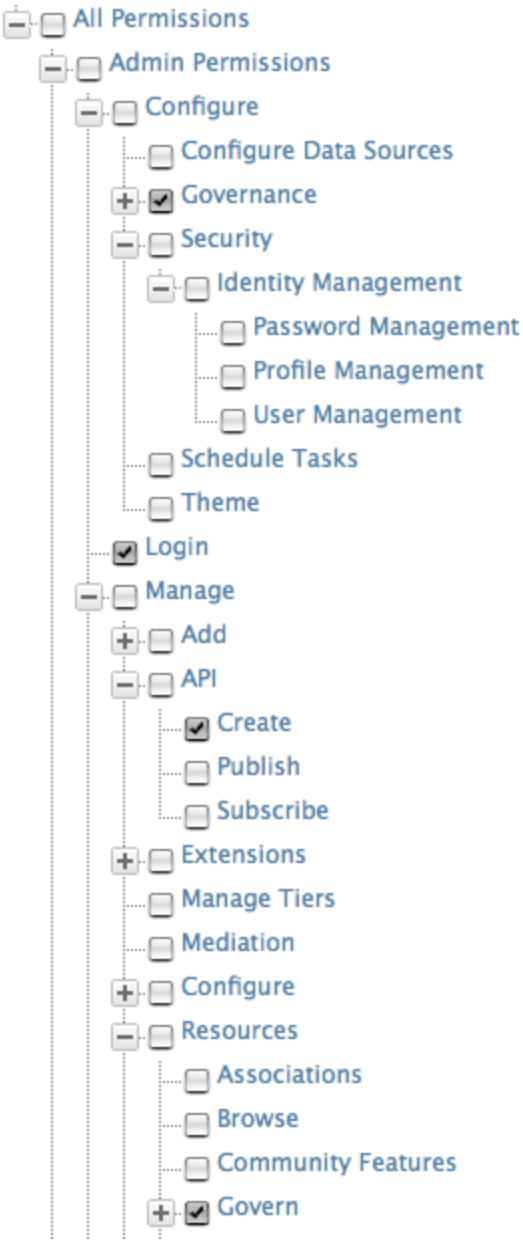
2. In the **User Management** page that opens, click **Roles** and **Add New Role** link.
   - Adding the creator role
   - Adding the publisher role
   - The default subscriber role

Adding the `creator` role

3. Add user role as `creator` and click **Next**. The **Domain** drop-down list contains all user stores configured for this product instance. By default, you only have the PRIMARY user store. To configure secondary user stores, see Configuring Secondary User Stores.

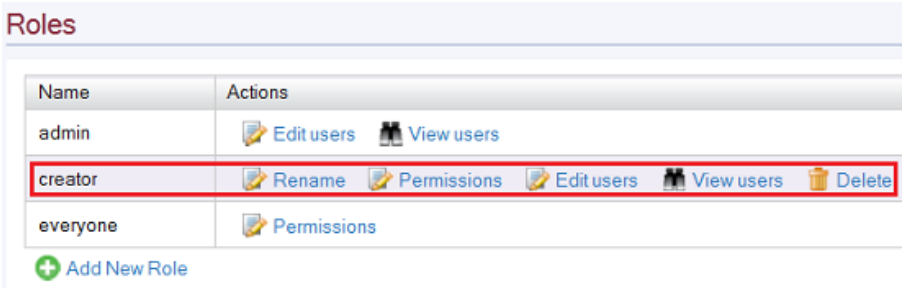


4. Give the following privileges to the creator role. You can select them from the list of permissions that appears.
   - Configure > Governance and all underlying permissions.
   - Login
   - Manage > API > Create
   - Manage > Resources > Govern and all underlying permissions



Any user with the above permissions assigned is able to create, update and manage APIs using the API Publisher Web interface.

5. Click **Finish** once you are done adding permission. The role will be listed in the **Roles** window as follows:

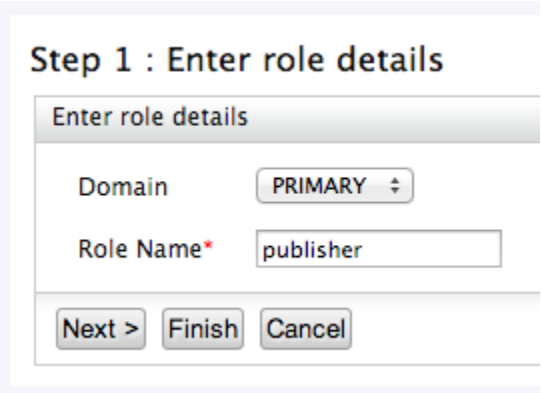


From here, you can rename, edit, delete or assign users to the role.

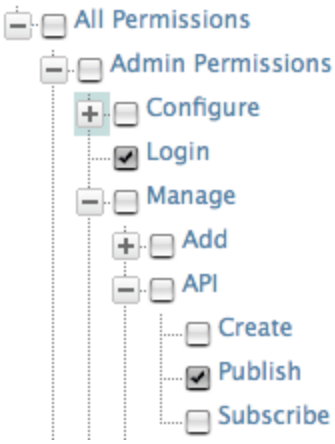
Adding the `publisher` role

6. In the **Add Role** page, add user role as **publisher** and click **Next**. The **Domain** drop-down list contains all

user stores configured for this product instance. By default, you only have the PRIMARY user store. To configure secondary user stores, see Configuring Secondary User Stores.

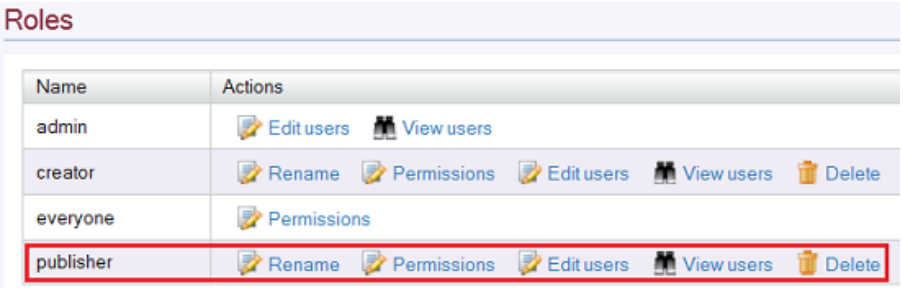


7. Give the following privileges to the publisher role by selecting them from the list of permissions that appears.
   - Login
   - Manage > API > Publish



Any user with the above permissions assigned is able to manage the API's life cycle using the API Publisher Web interface.

8. Click **Finish** once you are done adding permission. The role will be listed in the **Roles** window as follows:



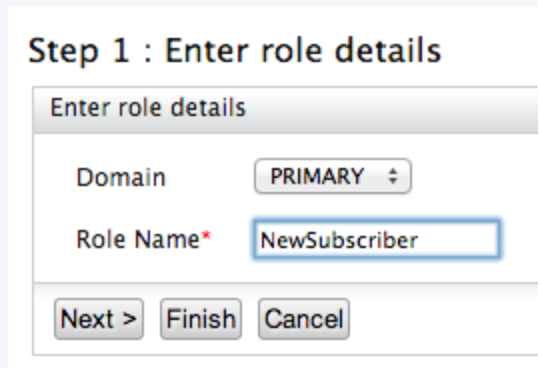
From here, you can rename, edit, delete or assign users to the role.

The default `subscriber` role

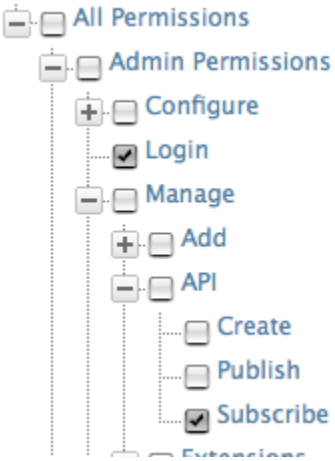
When you first log in to the Management Console, you can see the subscriber role already there, defined out of the box. The reason is because API Manager assigns this default subscriber role to all users who self-register to the API Store.

Follow the instructions below to create a different role with the same permission levels.

9. In the **Add Role** window, add a suitable name for the role and click **Next**. For example,



10. Give the following privileges to the new role.
    - Login
    - Manage > API > Subscribe



    Any user with the above permissions assigned is able to log in to the API Store and perform operations on the published APIs.
11. Click **Finish** once you are done adding permission. The role will be listed in the **Roles** window.
12. Open `<APIM_HOME>/repository/conf/ api-manager.xml` file and edit the `<SelfSignUp>` element to reflect the newly added role. For example,

```
<SelfSignUp>
     <Enabled>true</Enabled>
     <SubscriberRoleName>NewSubscriber</SubscriberRoleName>
     <CreateSubscriberRole>true</CreateSubscriberRole>
</SelfSignUp>
```

    Editing this file ensures that all users who self-sign-up to API Store are automatically assigned the `NewSubscriber` role.

> **Note**: The `<CreateSubscriberRole>` parameter specifies whether the subscriber role should be created in the local user store or not. It is only used when the API subscribers are authenticated against the local user store. That means the local Carbon server is acting as the AuthManager.
>
> Set this parameter to false if a remote Carbon server acts as the AuthManager.

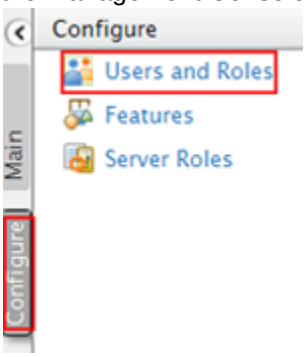
Adding Users

Users are consumers who interact with your organizational applications, databases or any other systems. These users can be a person, a device or another application/program within or outside of the organization's network.

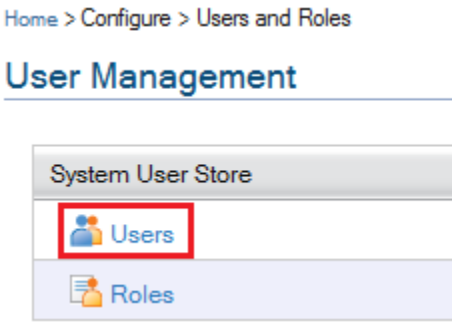
Since these users interact with internal systems and access data, the need to define which user is allowed to do what is critical. This is how the concept of user management developed. To enable users to log into the product's management console, you create user accounts and assign them roles, which are sets of permissions. You can add individual users or import users in bulk.

Follow the steps below to create users and assign them to roles that you created in section User Roles in the API Manager. Also see how to add an e-mail address as the username of a user.

the Management Console and select **Users and Roles** from the **Configure** menu.



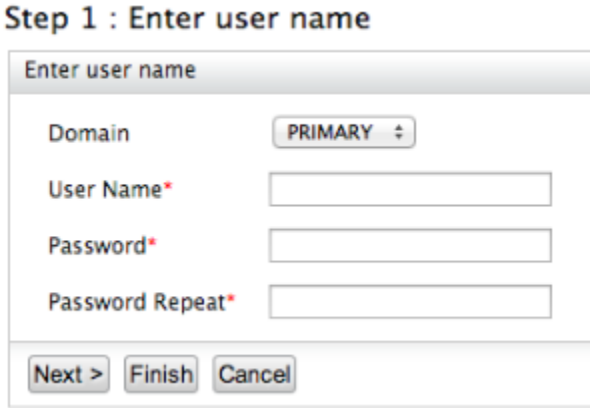
2. Click **Users** in the **User Management** window that opens.



> The **Users** link is only visible to users with administrator permission. It is used to add new user accounts and modify or delete existing accounts. The `admin` user has administrator privileges.

3. Click **Add New User.**

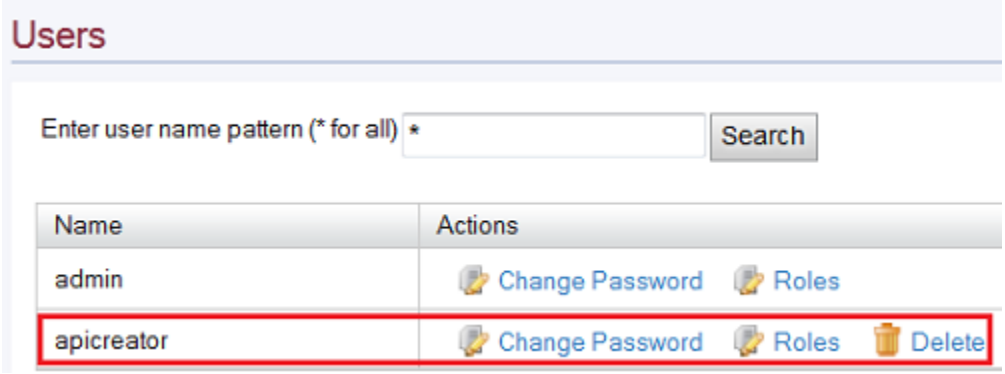
4. The **Add User** page opens. Provide the user name and password and click **Next**. The **Domain** drop-down list contains all user stores configured for this product instance. By default, you only have the PRIMARY user store. To configure other user stores, see Configuring User Stores.



5. Select the roles you want to assign to the user. In this example, we assign the user the `creator` role defined

in section User Roles in the API Manager.

6. Click **Finish** to complete. The new use appears in the **Users** list.



From here, you can change the user's password, assign different roles or delete it. Since the apicreator user is assigned the creator role, it now has permission to create and manage APIs through the API Manager. Similarly, you can create users and assign them the publisher and subscriber roles.

> You cannot change the user name of an existing user.

Using the e-mail as the username

When adding a user, if you provide an e-mail address as the username, modify the following files accordingly:

- In `<AM_HOME>/repository/conf/carbon.xml` file, set `<EnableEmailUserName>true</EnableEmailUserName>`
- In `<AM_HOME>/repository/conf/api-manager.xml` file, set

```
<LoginConfig>
         <UserIdLogin primary="true">
         <ClaimUri></ClaimUri>
         </UserIdLogin>
         <EmailLogin primary="false">
             <ClaimUri>http://wso2.org/claims/emailaddress&lt;/ClaimUri>
         </EmailLogin>
</LoginConfig>
```

- In `<AM_HOME>/repository/conf/user-mgt.xml` file, set

```
<UserStoreManager class="org.wso2.carbon.user.core.jdbc.JDBCUserStoreManager">
 ...
   <Property name="IsEmailUserName">true</Property>
   <Property
name="UsernameWithEmailJavaScriptRegEx">[a-zA-Z0-9@._-|//]{3,30}$</Property>
 ...
</UserStoreManager>
```

> ⚠ If there are **multiple tenants** set up in your environment, e-mail login does not work for any tenant including the super tenant. This facility is currently only available in single tenant mode (i.e., users of the `carbon.super` tenant only). However, this limitation does not apply to user provisioning based on a social network login.

Configuring User Stores

A user store is the database where information of the users and/or user roles is stored. User information includes log-in name, password, fist name, last name, e-mail etc.

All WSO2 products have an embedded H2 database except for WSO2 Identity Server, which has an embedded LDAP as its user store. Permission is stored in a separate database called the user management database, which by default is H2. However, users have the ability to connect to external user stores as well.

The user stores of Carbon products can be configured to operate in either one of the following modes.

- User store operates in read/write mode - In Read/Write mode, WSO2 Carbon reads/writes into the user store.
- User store operates in read only mode - In Read Only mode, WSO2 Carbon guarantees that it does not modify any data in the user store. Carbon maintains roles and permissions in the Carbon database but it can read users/roles from the configured user store.

The sections below provide configuration details:

- Realm Configuration
- Changing the RDBMS
- Configuring Primary User Stores
- Configuring Secondary User Stores

Realm Configuration

The `<Configuration>` section at the top of the `<PRODUCT_HOME>/repository/conf/user-mgt.xml` file allows you to specify basic configuration for connecting to this user store (also called a **realm**).

```
<Realm>
  <Configuration>
   <AddAdmin>true</AddAdmin>
   <AdminRole>admin</AdminRole>
   <AdminUser>
       <UserName>admin</UserName>
       <Password>admin</Password>
   </AdminUser>
   <EveryOneRoleName>everyone</EveryOneRoleName> <!-- By default users in this role
see the registry root -->
   <Property name="dataSource">jdbc/WSO2CarbonDB</Property>
  </Configuration>
...
</Realm>
```

Note the following regarding the configuration above.

Element	Description
`<AddAdmin>`	When `true`, this element creates the admin user based on the `AdminUser` element. It also indicates whether to create the specified admin user if it doesn't already exist. When connecting to an external read-only LDAP or Active Directory user store, this property needs to be `false` if an admin user and admin role exist within the user store. If the admin user and admin role do not exist in the user store, this value should be `true`, so that the role is added to the user management database. However, if the admin user is not there in the user store, we must add that user to the user store manually. If the `AddAdmin` value is set to `true` in this case, it will generate an exception.

`<AdminRole>wso2admin</AdminRole>`	This is the role that has all administrative privileges of the WSO2 product, so all users having this role are admins of the product. You can provide any meaningful name for this role. This role is created in the internal H2 database when the product starts. This role has permission to carry out any actions related to the Management Console. If the user store is read-only, this role is added to the system as a special internal role where users are from an external user store.
`<AdminUser>`	Configures the default administrator for the WSO2 product. If the user store is read-only, the admin user must exist in the user store or the system will not start. If the external user store is read-only, you must select a user already existing in the external user store and add it as the admin user that is defined in the `<AdminUser>` el ement. If the external user store is in read/write mode, and you set `<AddAdmin>` to `true`, the user you specify will be automatically created.
`<UserName>`	This is the username of the default administrator or super tenant of the user store. If the user store is read-only, the admin user MUST exist in the user store for the process to work.
`<Password>`	Do NOT put the password here but leave the default value as it is if the user store is read-only as this element and its value are ignored. This password is used only if the user store is read-write and the `AddAdmin` value is set to `true`. Note that the password in the `user-mgt.xml` file is written to the primary user store when the server starts for the first time. Thereafter, the password will be validated from the primary user store and not from the `user-mgt.xml` file. Therefore, if you need to change the admin password stored in the user store, you cannot simply change the value in the `user-mgt.xml` file. To change the admin password, you must use the **Change Password** option from the management console.
`<EveryOneRoleName>`	The name of the "everyone" role. All users in the system belong to this role.

The main property given below contains details of the database connection.

Property Name	Description	Mandat
`dataSource`	Data sources are configured in the `<PRODUCT_HOME>/repository/conf/datasou rces/master-datasources.xml` file. This property indicates the relevant data source configuration for the User Management database.	Mandatc

Given below are optional properties that can be used.

Property Name	Description

`testOnBorrow`	It is recommended to set this property to 'true' so that object cor validated before being borrowed from the JDBC pool. For this proper the `validationQuery` parameter in the `<PRODUCT_HOME>/repos tasources/master-datasources.xml` file should be a non-st setting will avoid connection failures. See the section on performance products for more information.
`CaseSensitiveAuthorizationRules`	Permissions, and the rules (role name, action, resource) linked to eac stored in the RDBMS of the server. By default, these rules are not ca property can be used if you want to make the rules case sensitive.

Changing the RDBMS

The default database of user manager is the H2 database that comes with WSO2 products. You can configure it to point to databases by other vendors.

1. Add the JDBC driver to the `classpath` by dropping the JAR into `<PRODUCT_HOME>/repository/compon ents/lib`.
2. Change values of properties given in on the Realm Configuration page appropriately.
3. Create the database by running the relevant script in `<PRODUCT_HOME>/dbscript` and restart the server:
   - **For Linux**: `sh wso2server.sh` or `sh wso2server.sh -Dsetup`
   - **For Windows**: `wso2server.bat` or `wso2server.bat -Dsetup`

Configuring Primary User Stores

Every WSO2 product comes with an embedded, internal user store, which is configured in `<PRODUCT_HOME>/repo sitory/conf/user-mgt.xml` file. In WSO2 Identity Server, the embedded user store is LDAP, and in other products it is JDBC. Because the domain name (unique identifier) of this default user store is set to `PRIMARY` by default, it is called the primary user store.

Instead of using the embedded user store, you can set your own user store as the primary user store. Because the user store you want to connect to might have different schemas from the ones available in the embedded user store, it needs to go through an adaptation process. WSO2 products provide the following adapters to enable you to authenticate users from different types of user stores and plug into LDAP, Active Directory, and JDBC to perform authentication:

- Use `ReadOnlyLDAPUserStoreManager` to do read-only operations for external LDAP user stores.
- Use `ReadWriteLDAPUserStoreManager` for external LDAP user stores to do both read and write operations.
- Use `ActiveDirectoryUserStoreManager` to configure an Active Directory Domain Service (AD DS) or Active Directory Lightweight Directory Service (AD LDS). This can be used for both read-only and read/write operations.
- Use `JDBCUserStoreManager` for both internal and external JDBC user stores.

The following topics provide details on the various primary user stores you can configure.

- Configuring an external LDAP user store/active directory
- Configuring an internal/external JDBC user store

Configuring an external LDAP user store/active directory

All WSO2 products can read and write users and roles from external Active Directory/LDAP user stores. You can configure WSO2 products to access the Active Directory/LDAP user stores using one of the following modes.

- Read-only mode
- Read/write mode

Read-only mode

When you configure a product to read users/roles from your company LDAP in the 'Read Only' mode, it does not write any data into the LDAP.

1. Given below are samples for LDAP and Active Directory user stores in the `<PRODUCT_HOME> /repository /conf/**user-mgt.xml**` file.

   LDAP User StoreActive Directory User Store

   LDAP user store sample:

```
<UserManager>
 <Realm>
   <Configuration>
      <AdminRole>admin</AdminRole>
      <AdminUser>
         <UserName>admin</UserName>
         <Password>XXXXXX</Password>
      </AdminUser>
      <EveryOneRoleName>everyone</EveryOneRoleName>
      <!-- By default users in this role sees the registry root -->
      <Property name="dataSource">jdbc/WSO2CarbonDB</Property>
      <Property
name="MultiTenantRealmConfigBuilder">org.wso2.carbon.user.core.config.multitenanc
y.SimpleRealmConfigBuilder</Property>
   </Configuration>

   <UserStoreManager
class="org.wso2.carbon.user.core.ldap.ReadOnlyLDAPUserStoreManager">
      <Property
name="TenantManager">org.wso2.carbon.user.core.tenant.CommonHybridLDAPTenantManag
er</Property>
   <Property name="ConnectionURL">ldap://localhost:10389</Property>
      <Property name="ConnectionName">uid=admin,ou=system</Property>
      <Property name="ConnectionPassword">admin123</Property>
      <Property name="UserSearchBase">ou=system</Property>
      <Property name="UserNameListFilter">(objectClass=person)</Property>
      <Property name="UserNameAttribute">uid</Property>
      <Property name="ReadLDAPGroups">false</Property>
      <Property name="GroupSearchBase">ou=system</Property>
      <Property
name="GroupNameSearchFilter">(objectClass=groupOfNames)</Property>
      <Property name="GroupNameAttribute">cn</Property>
      <Property name="MembershipAttribute">member</Property>
   </UserStoreManager>

 </Realm>
</UserManager>
```

   Active directory user store sample:

```
<UserManager>
  <Realm>
   <Configuration>
      <AdminRole>admin</AdminRole>
      <AdminUser>
         <UserName>admin</UserName>
         <Password>XXXXXX</Password>
```

```
      </AdminUser>
      <EveryOneRoleName>everyone</EveryOneRoleName>
      <!-- By default users in this role sees the registry root -->
      <Property name="dataSource">jdbc/WSO2CarbonDB</Property>
      <Property
name="MultiTenantRealmConfigBuilder">org.wso2.carbon.user.core.config.multitenanc
y.SimpleRealmConfigBuilder</Property>
   </Configuration>

    <!-- Active directory configuration follows -->
    <UserStoreManager
class="org.wso2.carbon.user.core.ldap.ActiveDirectoryUserStoreManager">
            <Property
name="TenantManager">org.wso2.carbon.user.core.tenant.CommonHybridLDAPTenantManag
er</Property>
            <Property name="defaultRealmName">WSO2.ORG</Property>
            <Property name="Disabled">false</Property>

            <Property name="kdcEnabled">false</Property>
            <Property name="ConnectionURL">ldaps://10.100.1.100:636</Property>
            <Property
name="ConnectionName">CN=admin,CN=Users,DC=WSO2,DC=Com</Property>
            <Property name="ConnectionPassword">A1b2c3d4</Property>
      <Property name="passwordHashMethod">PLAIN_TEXT</Property>
            <Property name="UserSearchBase">CN=Users,DC=WSO2,DC=Com</Property>
            <Property name="UserEntryObjectClass">user</Property>
            <Property name="UserNameAttribute">cn</Property>
            <Property name="isADLDSRole">false</Property>
      <Property name="userAccountControl">512</Property>
            <Property name="UserNameListFilter">(objectClass=user)</Property>
      <Property
name="UserNameSearchFilter">(&amp;(objectClass=user)(cn=?))</Property>
            <Property
name="UsernameJavaRegEx">[a-zA-Z0-9._-|//]{3,30}$</Property>
            <Property name="UsernameJavaScriptRegEx">^[\S]{3,30}$</Property>
            <Property name="PasswordJavaScriptRegEx">^[\S]{5,30}$</Property>
      <Property name="RolenameJavaScriptRegEx">^[\S]{3,30}$</Property>
            <Property
name="RolenameJavaRegEx">[a-zA-Z0-9._-|//]{3,30}$</Property>
      <Property name="ReadGroups">true</Property>
      <Property name="WriteGroups">false</Property>
      <Property name="EmptyRolesAllowed">true</Property>
            <Property name="GroupSearchBase">CN=Users,DC=WSO2,DC=Com</Property>
      <Property name="GroupEntryObjectClass">group</Property>
            <Property name="GroupNameAttribute">cn</Property>
            <Property name="SharedGroupNameAttribute">cn</Property>
            <Property
name="SharedGroupSearchBase">ou=SharedGroups,dc=wso2,dc=org</Property>
            <Property name="SharedGroupEntryObjectClass">groups</Property>
            <Property
name="SharedTenantNameListFilter">(object=organizationalUnit)</Property>
            <Property name="SharedTenantNameAttribute">ou</Property>
            <Property
name="SharedTenantObjectClass">organizationalUnit</Property>
            <Property name="MembershipAttribute">member</Property>
            <Property
name="GroupNameListFilter">(objectcategory=group)</Property>
      <Property
name="GroupNameSearchFilter">(&amp;(objectClass=group)(cn=?))</Property>
```

```
            <Property name="UserRolesCacheEnabled">true</Property>
            <Property name="Referral">follow</Property>
      <Property name="BackLinksEnabled">true</Property>
            <Property name="MaxRoleNameListLength">100</Property>
            <Property name="MaxUserNameListLength">100</Property>
            <Property name="SCIMEnabled">false</Property>
</UserStoreManager>
```

```
  </Realm>
</UserManager>
```

The following tags in your file indicate whether it is an Active Directory or LDAP:

- Active Directory: `<UserStoreManager class="org.wso2.carbon.user.core.ldap.ActiveDirectoryUserStoreManager">`
- LDAP: `<UserStoreManager class="org.wso2.carbon.user.core.ldap.ReadOnlyLDAPUserStoreManager">`

> If you create the `user-mgt.xml` file yourself, be sure to save it in the `<PRODUCT_HOME>/repository/conf` directory.

Find a valid user that resides in the directory server. For example, if the valid username is `AdminSOA`, update the Admin user section of your LDAP configuration as follows. You do not have to update the password element; leave it as is.

```
<AdminRole>wso2admin</AdminRole>
<AdminUser>
   <UserName>AdminSOA</UserName>
   <Password>XXXXXX</Password>
</AdminUser>
```

Note the following regarding the configuration above:

Element	Description
`<AdminRole>wso2admin</AdminRole>`	This is the role that has all administrative privileges of the WSC admins of the product. You can provide any meaningful name f internal H2 database when the product starts.

`<AdminUser>`	Configure the default administrator for the WSO2 product. If th select a user already existing in the external user store and ad er> element. If the external user store is in read/write mode, e inUser> element does not exist in the external user store, it w
	
	If you are connecting WSO2 BAM with an external LDAF HOME>/repository/conf/etc/cassandra-auth. the <AdminUser> element of the user-mgt.xml fil access Cassandra Keyspaces using the BAM managem nSOA as the admin user, the cassandra-auth.xml fil
	
	`<Cassandra>`
	`  <!--  local transport -->`
	`  <EPR>local://services/CassandraSharedK`
	`  <!-- HTTP transport -->`
	`  <!--`
	`<EPR>https://localhost:9443/services/Cassa`
	`-->`
	`  <User>AdminSOA</User>`
	`  <Password>xxxxx</Password>`
	`</Cassandra>`

`<UserName>`	Username of the default administrator. This user MUST exist ir read-only, the admin user must exist in the user store for the p
`<Password>`	Do NOT put the password here. Just leave it empty or place sc read-only, this element and its value are ignored.

2. Update the connection details to suit your Directory Server. For example:

```
<Property name="ConnectionURL">ldap://localhost:10389</Property>
```

3. Obtain a user who has permission to read all users/attributes and perform searches on the Directory Server from your LDAP administrator. For example, if the privileged user is "AdminLDAP" and the password is "2010#Avrudu", update the following sections of the realm configuration as follows:

```
<Property name="ConnectionName">uid=AdminLDAP,ou=system</Property>
<Property name="ConnectionPassword">2010#Avrudu</Property>
```

4. Update `<Property name="UserSearchBase">` by providing the directory name where the users are stored. When LDAP searches for users, it will start from this location of the directory.

```
<Property name="UserSearchBase">ou=system</Property>
```

5. Set the attribute to use as the username. The most common case is to use either `cn` or `uid` as the username. If you are not sure what attribute is available in your LDAP, check with your LDAP administrator.

```
<Property name="UserNameAttribute">uid</Property>
```

For Active Directory this will differ as follows:

```
<Property name="UserNameAttribute">sAMAccountName</Property>
```

> Ideally, `<Property name="UserNameAttribute">` and `<Property name="UserNameSearchFilter">` should refer to the same attribute.

6. Optionally, configure the realm to read roles from the Directory Server by reading the user/role mapping based on a membership (user list) or backlink attribute, as follows:

   - The following code snippet represents reading roles based on a membership attribute. This is used by the `ApacheDirectory` server and `OpenLDAP`.

     ```
     <Property name="ReadLDAPGroups">false</Property>
     <Property name="GroupSearchBase">ou=system</Property>
     <Property name="GroupSearchFilter">(objectClass=groupOfNames)</Property>
     <Property name="GroupNameAttribute">cn</Property>
     <Property name="MembershipAttribute">member</Property>
     ```

   - The following code snippet represents reading roles based on a backlink attribute. This is used by the Active Directory.

     ```
     <Property name="ReadLDAPGroups">true</Property>
     <Property name="GroupSearchBase">cn=users,dc=wso2,dc=lk</Property>
     <Property name="GroupSearchFilter">(objectcategory=group)</Property>
     <Property name="GroupNameAttribute">cn</Property>
     <Property name="MemberOfAttribute">memberOf</Property>
     ```

7. Start your server and try to log in as "AdminSOA". The password is the AdminSOA's password in the LDAP server.

Read/write mode

If you want to connect to an external LDAP user store, such that only the user entries are written to the external LDAP and roles are not written to the external LDAP, the only difference from the steps in section Read-only mode i s the following:

```
<UserStoreManager
class="org.wso2.carbon.user.core.ldap.ReadWriteLDAPUserStoreManager">
```

The `<PRODUCT_HOME>/repository/conf/user-mgt.xml` file has commented-out configurations for external LDAP user stores.

1. Enable the `<ReadWriteLDAPUserStoreManager>` element in the `user-mgt.xml` file by uncommenting the code. When it is enabled, the user manager reads/writes into the LDAP user store.
2. The default configuration for the external read/write LDAP user store in the `user-mgt.xml` file is as follows. Change the values according to your requirements.

LDAP User StoreActive Directory User Store

LDAP user store sample:

```
<UserStoreManager
class="org.wso2.carbon.user.core.ldap.ReadWriteLDAPUserStoreManager">
   <Property
name="TenantManager">org.wso2.carbon.user.core.tenant.CommonHybridLDAPTenantManag
er</Property>
   <Property
name="ConnectionURL">ldap://localhost:${Ports.EmbeddedLDAP.LDAPServerPort}</Prope
rty>
   <Property name="ConnectionName">uid=admin,ou=system</Property>
   <Property name="ConnectionPassword">admin</Property>
   <Property name="passwordHashMethod">SHA</Property>
   <Property name="UserNameListFilter">(objectClass=person)</Property>
   <Property name="UserEntryObjectClass">wso2Person</Property>
   <Property name="UserSearchBase">ou=Users,dc=wso2,dc=org</Property>
   <Property
name="UserNameSearchFilter">(&amp;(objectClass=person)(uid=?))</Property>
   <Property name="UserNameAttribute">uid</Property>
   <Property name="PasswordJavaScriptRegEx">[\\S]{5,30}</Property>
   <Property name="UsernameJavaScriptRegEx">[\\S]{3,30}</Property>
   <Property
name="UsernameJavaRegEx">^[^~!@#$;%^*+={}\\|\\\\&lt;&gt;,\'\"]{3,30}$</Property>
   <Property name="RolenameJavaScriptRegEx">[\\S]{3,30}</Property>
   <Property
name="RolenameJavaRegEx">^[^~!@#$;%^*+={}\\|\\\\&lt;&gt;,\'\"]{3,30}$</Property>
   <Property name="ReadLDAPGroups">true</Property>
   <Property name="WriteLDAPGroups">true</Property>
   <Property name="EmptyRolesAllowed">true</Property>
   <Property name="GroupSearchBase">ou=Groups,dc=wso2,dc=org</Property>
   <Property name="GroupNameListFilter">(objectClass=groupOfNames)</Property>
   <Property name="GroupEntryObjectClass">groupOfNames</Property>
   <Property
name="GroupNameSearchFilter">(&amp;(objectClass=groupOfNames)(cn=?))</Property>
   <Property name="GroupNameAttribute">cn</Property>
   <Property name="MembershipAttribute">member</Property>
   <Property name="UserRolesCacheEnabled">true</Property>
   <Property name="UserDNPattern">uid={0},ou=Users,dc=wso2,dc=org</Property>
</UserStoreManager>
```

Active directory user store sample:

```
<UserStoreManager
class="org.wso2.carbon.user.core.ldap.ActiveDirectoryUserStoreManager">
            <Property
name="TenantManager">org.wso2.carbon.user.core.tenant.CommonHybridLDAPTenantManag
er</Property>
            <Property name="defaultRealmName">WSO2.ORG</Property>
            <Property name="Disabled">false</Property>

            <Property name="kdcEnabled">false</Property>
            <Property name="ConnectionURL">ldaps://10.100.1.100:636</Property>
            <Property
name="ConnectionName">CN=admin,CN=Users,DC=WSO2,DC=Com</Property>
            <Property name="ConnectionPassword">A1b2c3d4</Property>
      <Property name="passwordHashMethod">PLAIN_TEXT</Property>
            <Property name="UserSearchBase">CN=Users,DC=WSO2,DC=Com</Property>
            <Property name="UserEntryObjectClass">user</Property>
            <Property name="UserNameAttribute">cn</Property>
            <Property name="isADLDSRole">false</Property>
      <Property name="userAccountControl">512</Property>
            <Property name="UserNameListFilter">(objectClass=user)</Property>
      <Property
name="UserNameSearchFilter">(&amp;(objectClass=user)(cn=?))</Property>
            <Property
name="UsernameJavaRegEx">[a-zA-Z0-9._-|//]{3,30}$</Property>
            <Property name="UsernameJavaScriptRegEx">^[\S]{3,30}$</Property>
            <Property name="PasswordJavaScriptRegEx">^[\S]{5,30}$</Property>
      <Property name="RolenameJavaScriptRegEx">^[\S]{3,30}$</Property>
            <Property
name="RolenameJavaRegEx">[a-zA-Z0-9._-|//]{3,30}$</Property>
      <Property name="ReadGroups">true</Property>
      <Property name="WriteGroups">true</Property>
      <Property name="EmptyRolesAllowed">true</Property>
            <Property name="GroupSearchBase">CN=Users,DC=WSO2,DC=Com</Property>
      <Property name="GroupEntryObjectClass">group</Property>
            <Property name="GroupNameAttribute">cn</Property>
            <Property name="SharedGroupNameAttribute">cn</Property>
            <Property
name="SharedGroupSearchBase">ou=SharedGroups,dc=wso2,dc=org</Property>
            <Property name="SharedGroupEntryObjectClass">groups</Property>
            <Property
name="SharedTenantNameListFilter">(object=organizationalUnit)</Property>
            <Property name="SharedTenantNameAttribute">ou</Property>
            <Property
name="SharedTenantObjectClass">organizationalUnit</Property>
            <Property name="MembershipAttribute">member</Property>
            <Property
name="GroupNameListFilter">(objectcategory=group)</Property>
      <Property
name="GroupNameSearchFilter">(&amp;(objectClass=group)(cn=?))</Property>
            <Property name="UserRolesCacheEnabled">true</Property>
            <Property name="Referral">follow</Property>
      <Property name="BackLinksEnabled">true</Property>
            <Property name="MaxRoleNameListLength">100</Property>
            <Property name="MaxUserNameListLength">100</Property>
            <Property name="SCIMEnabled">false</Property>
</UserStoreManager>
```

> For active directory configurations, the `WriteGroups` property is set to true for read/write mode and false for read-only mode.

Configuring an internal/external JDBC user store

The default internal JDBC user store reads/writes into the internal database of the Carbon server. JDBC user stores can be configured using the `<PRODUCT_HOME>/repository/conf/user-mgt.xml` file's `JDBCUserStoreManager` configuration section. In addition to this, all Carbon-based products can work with external RDBMSs. You can configure Carbon to read users/roles from your company RDBMS and even write to it. Therefore, in this scenario, the user core connects to two databases:

- The Carbon database where authorization information is stored internally.
- Your company database where users/roles reside.

So the `user-mgt.xml` file must contain details for two database connections. The connection details mentioned earlier are used by the authorization manager. If we specify another set of database connection details inside the `UserStoreManager`, it reads/writes users to that database. The following are step-by-step guidelines for connecting to an internal and external JDBC user store in read-only mode:

1. A sample file for the JDBC user store (`user-mgt.xml`) is available in the `<PRODUCT_HOME> /repository /conf` directory. Uncomment the following section in your file if it is commented out:

```
<UserStoreManager class="org.wso2.carbon.user.core.jdbc.JDBCUserStoreManager">
```

The following are samples for the internal and external JDBC user store configuration:

Internal JDBC User StoreExternal JDBC User Store

Internal JDBC user store configuration sample:

```
<UserStoreManager class="org.wso2.carbon.user.core.jdbc.JDBCUserStoreManager">
  <Property
name="TenantManager">org.wso2.carbon.user.core.tenant.JDBCTenantManager</Property
>
  <Property name="ReadOnly">false</Property>
  <Property name="MaxUserNameListLength">100</Property>
  <Property name="IsEmailUserName">false</Property>
  <Property name="DomainCalculation">default</Property>
     <Property name="PasswordDigest">SHA-256</Property>
  <Property name="StoreSaltedPassword">true</Property>
  <Property name="UserNameUniqueAcrossTenants">false</Property>
  <Property name="PasswordJavaRegEx">[\S]{5,30}$</Property>
  <Property name="PasswordJavaScriptRegEx">[\\S]{5,30}</Property>
  <Property
name="UsernameJavaRegEx">^[^~!#$;%^*+={}\\|\\\\&lt;&gt;,\'\"]{3,30}$</Property>
  <Property name="UsernameJavaScriptRegEx">[\\S]{3,30}</Property>
  <Property
name="RolenameJavaRegEx">^[^~!@#$;%^*+={}\\|\\\\&lt;&gt;,\'\"]{3,30}$</Property>
  <Property name="RolenameJavaScriptRegEx">[\\S]{3,30}</Property>
  <Property name="UserRolesCacheEnabled">true</Property>
</UserStoreManager>
```

External JDBC user store configuration sample:

```
<UserStoreManager class="org.wso2.carbon.user.core.jdbc.JDBCUserStoreManager">
      <Property name="driverName">com.mysql.jdbc.Driver</Property>
      <Property name="url">jdbc://localhost:3306/test</Property>
      <Property name="userName">admin</Property>
      <Property name="password">admin</Property>
      <Property name="Disabled">false</Property>
      <Property name="MaxUserNameListLength">100</Property>
      <Property name="MaxRoleNameListLength">100</Property>
      <Property name="UserRolesCacheEnabled">true</Property>
      <Property name="PasswordDigest">SHA-256</Property>
      <Property name="ReadGroups">true</Property>
      <Property name="ReadOnly">false</Property>
      <Property name="IsEmailUserName">false</Property>
      <Property name="DomainCalculation">default</Property>
      <Property name="StoreSaltedPassword">true</Property>
      <Property name="WriteGroups">false</Property>
      <Property name="UserNameUniqueAcrossTenants">false</Property>
      <Property name="PasswordJavaRegEx">^[\S]{5,30}$</Property>
      <Property name="PasswordJavaScriptRegEx">^[\S]{5,30}$</Property>
      <Property name="UsernameJavaRegEx">^[\S]{5,30}$</Property>
      <Property name="UsernameJavaScriptRegEx">^[\S]{5,30}$</Property>
      <Property name="RolenameJavaRegEx">^[\S]{5,30}$</Property>
      <Property name="RolenameJavaScriptRegEx">^[\S]{5,30}$</Property>
      <Property name="SCIMEnabled">false</Property>
      <Property name="SelectUserSQL">SELECT * FROM UM_USER WHERE UM_USER_NAME=?
AND UM_TENANT_ID=?</Property>
      <Property name="GetRoleListSQL">SELECT UM_ROLE_NAME, UM_TENANT_ID,
UM_SHARED_ROLE FROM UM_ROLE WHERE UM_ROLE_NAME LIKE ? AND UM_TENANT_ID=? AND
UM_SHARED_ROLE ='0' ORDER BY UM_ROLE_NAME</Property>
      <Property name="GetSharedRoleListSQL">SELECT UM_ROLE_NAME, UM_TENANT_ID,
UM_SHARED_ROLE FROM UM_ROLE WHERE UM_ROLE_NAME LIKE ? AND UM_SHARED_ROLE ='1'
ORDER BY UM_ROLE_NAME</Property>
      <Property name="UserFilterSQL">SELECT UM_USER_NAME FROM UM_USER WHERE
UM_USER_NAME LIKE ? AND UM_TENANT_ID=? ORDER BY UM_USER_NAME</Property>
      <Property name="UserRoleSQL">SELECT UM_ROLE_NAME FROM UM_USER_ROLE,
UM_ROLE, UM_USER WHERE UM_USER.UM_USER_NAME=? AND
UM_USER.UM_ID=UM_USER_ROLE.UM_USER_ID AND UM_ROLE.UM_ID=UM_USER_ROLE.UM_ROLE_ID
AND UM_USER_ROLE.UM_TENANT_ID=? AND UM_ROLE.UM_TENANT_ID=? AND
UM_USER.UM_TENANT_ID=?</Property>
      <Property name="UserSharedRoleSQL">SELECT UM_ROLE_NAME,
UM_ROLE.UM_TENANT_ID, UM_SHARED_ROLE FROM UM_SHARED_USER_ROLE INNER JOIN UM_USER
ON UM_SHARED_USER_ROLE.UM_USER_ID = UM_USER.UM_ID INNER JOIN UM_ROLE ON
UM_SHARED_USER_ROLE.UM_ROLE_ID = UM_ROLE.UM_ID WHERE UM_USER.UM_USER_NAME = ? AND
UM_SHARED_USER_ROLE.UM_USER_TENANT_ID = UM_USER.UM_TENANT_ID AND
UM_SHARED_USER_ROLE.UM_ROLE_TENANT_ID = UM_ROLE.UM_TENANT_ID AND
UM_SHARED_USER_ROLE.UM_USER_TENANT_ID = ?</Property>
      <Property name="IsRoleExistingSQL">SELECT UM_ID FROM UM_ROLE WHERE
UM_ROLE_NAME=? AND UM_TENANT_ID=?</Property>
      <Property name="GetUserListOfRoleSQL">SELECT UM_USER_NAME FROM
UM_USER_ROLE, UM_ROLE, UM_USER WHERE UM_ROLE.UM_ROLE_NAME=? AND
UM_USER.UM_ID=UM_USER_ROLE.UM_USER_ID AND UM_ROLE.UM_ID=UM_USER_ROLE.UM_ROLE_ID
AND UM_USER_ROLE.UM_TENANT_ID=? AND UM_ROLE.UM_TENANT_ID=? AND
UM_USER.UM_TENANT_ID=?</Property>
      <Property name="GetUserListOfSharedRoleSQL">SELECT UM_USER_NAME FROM
UM_SHARED_USER_ROLE INNER JOIN UM_USER ON UM_SHARED_USER_ROLE.UM_USER_ID =
UM_USER.UM_ID INNER JOIN UM_ROLE ON UM_SHARED_USER_ROLE.UM_ROLE_ID =
UM_ROLE.UM_ID WHERE UM_ROLE.UM_ROLE_NAME= ? AND
UM_SHARED_USER_ROLE.UM_USER_TENANT_ID = UM_USER.UM_TENANT_ID AND
UM_SHARED_USER_ROLE.UM_ROLE_TENANT_ID = UM_ROLE.UM_TENANT_ID</Property>
```

```
      <Property name="IsUserExistingSQL">SELECT UM_ID FROM UM_USER WHERE
UM_USER_NAME=? AND UM_TENANT_ID=?</Property>
      <Property name="GetUserPropertiesForProfileSQL">SELECT UM_ATTR_NAME,
UM_ATTR_VALUE FROM UM_USER_ATTRIBUTE, UM_USER WHERE UM_USER.UM_ID =
UM_USER_ATTRIBUTE.UM_USER_ID AND UM_USER.UM_USER_NAME=? AND UM_PROFILE_ID=? AND
UM_USER_ATTRIBUTE.UM_TENANT_ID=? AND UM_USER.UM_TENANT_ID=?</Property>
      <Property name="GetUserPropertyForProfileSQL">SELECT UM_ATTR_VALUE FROM
UM_USER_ATTRIBUTE, UM_USER WHERE UM_USER.UM_ID = UM_USER_ATTRIBUTE.UM_USER_ID AND
UM_USER.UM_USER_NAME=? AND UM_ATTR_NAME=? AND UM_PROFILE_ID=? AND
UM_USER_ATTRIBUTE.UM_TENANT_ID=? AND UM_USER.UM_TENANT_ID=?</Property>
      <Property name="GetUserLisForPropertySQL">SELECT UM_USER_NAME FROM UM_USER,
UM_USER_ATTRIBUTE WHERE UM_USER_ATTRIBUTE.UM_USER_ID = UM_USER.UM_ID AND
UM_USER_ATTRIBUTE.UM_ATTR_NAME =? AND UM_USER_ATTRIBUTE.UM_ATTR_VALUE =? AND
UM_USER_ATTRIBUTE.UM_PROFILE_ID=? AND UM_USER_ATTRIBUTE.UM_TENANT_ID=? AND
UM_USER.UM_TENANT_ID=?</Property>
      <Property name="GetProfileNamesSQL">SELECT DISTINCT UM_PROFILE_ID FROM
UM_USER_ATTRIBUTE WHERE UM_TENANT_ID=?</Property>
      <Property name="GetUserProfileNamesSQL">SELECT DISTINCT UM_PROFILE_ID FROM
UM_USER_ATTRIBUTE WHERE UM_USER_ID=(SELECT UM_ID FROM UM_USER WHERE
UM_USER_NAME=? AND UM_TENANT_ID=?) AND UM_TENANT_ID=?</Property>
      <Property name="GetUserIDFromUserNameSQL">SELECT UM_ID FROM UM_USER WHERE
UM_USER_NAME=? AND UM_TENANT_ID=?</Property>
      <Property name="GetUserNameFromTenantIDSQL">SELECT UM_USER_NAME FROM
UM_USER WHERE UM_TENANT_ID=?</Property>
      <Property name="GetTenantIDFromUserNameSQL">SELECT UM_TENANT_ID FROM
UM_USER WHERE UM_USER_NAME=?</Property>
      <Property name="AddUserSQL">INSERT INTO UM_USER (UM_USER_NAME,
UM_USER_PASSWORD, UM_SALT_VALUE, UM_REQUIRE_CHANGE, UM_CHANGED_TIME,
UM_TENANT_ID) VALUES (?, ?, ?, ?, ?, ?)</Property>
      <Property name="AddUserToRoleSQL">INSERT INTO UM_USER_ROLE (UM_USER_ID,
UM_ROLE_ID, UM_TENANT_ID) VALUES ((SELECT UM_ID FROM UM_USER WHERE UM_USER_NAME=?
AND UM_TENANT_ID=?),(SELECT UM_ID FROM UM_ROLE WHERE UM_ROLE_NAME=? AND
UM_TENANT_ID=?), ?)</Property>
      <Property name="AddRoleSQL">INSERT INTO UM_ROLE (UM_ROLE_NAME,
UM_TENANT_ID) VALUES (?, ?)</Property>
      <Property name="AddSharedRoleSQL">UPDATE UM_ROLE SET UM_SHARED_ROLE = ?
WHERE UM_ROLE_NAME = ? AND UM_TENANT_ID = ?</Property>
      <Property name="AddRoleToUserSQL">INSERT INTO UM_USER_ROLE (UM_ROLE_ID,
UM_USER_ID, UM_TENANT_ID) VALUES ((SELECT UM_ID FROM UM_ROLE WHERE UM_ROLE_NAME=?
AND UM_TENANT_ID=?),(SELECT UM_ID FROM UM_USER WHERE UM_USER_NAME=? AND
UM_TENANT_ID=?), ?)</Property>
      <Property name="AddSharedRoleToUserSQL">INSERT INTO UM_SHARED_USER_ROLE
(UM_ROLE_ID, UM_USER_ID, UM_USER_TENANT_ID, UM_ROLE_TENANT_ID) VALUES ((SELECT
UM_ID FROM UM_ROLE WHERE UM_ROLE_NAME=? AND UM_TENANT_ID=?),(SELECT UM_ID FROM
UM_USER WHERE UM_USER_NAME=? AND UM_TENANT_ID=?), ?, ?)</Property>
      <Property name="RemoveUserFromSharedRoleSQL">DELETE FROM
UM_SHARED_USER_ROLE WHERE   UM_ROLE_ID=(SELECT UM_ID FROM UM_ROLE WHERE
UM_ROLE_NAME=? AND UM_TENANT_ID=?) AND UM_USER_ID=(SELECT UM_ID FROM UM_USER
WHERE UM_USER_NAME=? AND UM_TENANT_ID=?) AND UM_USER_TENANT_ID=? AND
UM_ROLE_TENANT_ID = ?</Property>
      <Property name="RemoveUserFromRoleSQL">DELETE FROM UM_USER_ROLE WHERE
UM_USER_ID=(SELECT UM_ID FROM UM_USER WHERE UM_USER_NAME=? AND UM_TENANT_ID=?)
AND UM_ROLE_ID=(SELECT UM_ID FROM UM_ROLE WHERE UM_ROLE_NAME=? AND
UM_TENANT_ID=?) AND UM_TENANT_ID=?</Property>
      <Property name="RemoveRoleFromUserSQL">DELETE FROM UM_USER_ROLE WHERE
UM_ROLE_ID=(SELECT UM_ID FROM UM_ROLE WHERE UM_ROLE_NAME=? AND UM_TENANT_ID=?)
AND UM_USER_ID=(SELECT UM_ID FROM UM_USER WHERE UM_USER_NAME=? AND
UM_TENANT_ID=?) AND UM_TENANT_ID=?</Property>
      <Property name="DeleteRoleSQL">DELETE FROM UM_ROLE WHERE UM_ROLE_NAME = ?
```

```
AND UM_TENANT_ID=?</Property>
      <Property name="OnDeleteRoleRemoveUserRoleMappingSQL">DELETE FROM
UM_USER_ROLE WHERE UM_ROLE_ID=(SELECT UM_ID FROM UM_ROLE WHERE UM_ROLE_NAME=? AND
UM_TENANT_ID=?) AND UM_TENANT_ID=?</Property>
      <Property name="DeleteUserSQL">DELETE FROM UM_USER WHERE UM_USER_NAME = ?
AND UM_TENANT_ID=?</Property>
      <Property name="OnDeleteUserRemoveUserRoleMappingSQL">DELETE FROM
UM_USER_ROLE WHERE UM_USER_ID=(SELECT UM_ID FROM UM_USER WHERE UM_USER_NAME=? AND
UM_TENANT_ID=?) AND UM_TENANT_ID=?</Property>
      <Property name="OnDeleteUserRemoveUserAttributeSQL">DELETE FROM
UM_USER_ATTRIBUTE WHERE UM_USER_ID=(SELECT UM_ID FROM UM_USER WHERE
UM_USER_NAME=? AND UM_TENANT_ID=?) AND UM_TENANT_ID=?</Property>
      <Property name="UpdateUserPasswordSQL">UPDATE UM_USER SET UM_USER_PASSWORD=
?, UM_SALT_VALUE=?, UM_REQUIRE_CHANGE=?, UM_CHANGED_TIME=? WHERE UM_USER_NAME= ?
AND UM_TENANT_ID=?</Property>
      <Property name="UpdateRoleNameSQL">UPDATE UM_ROLE set UM_ROLE_NAME=? WHERE
UM_ROLE_NAME = ? AND UM_TENANT_ID=?</Property>
      <Property name="AddUserPropertySQL">INSERT INTO UM_USER_ATTRIBUTE
(UM_USER_ID, UM_ATTR_NAME, UM_ATTR_VALUE, UM_PROFILE_ID, UM_TENANT_ID) VALUES
((SELECT UM_ID FROM UM_USER WHERE UM_USER_NAME=? AND UM_TENANT_ID=?), ?, ?, ?,
?)</Property>
      <Property name="UpdateUserPropertySQL">UPDATE UM_USER_ATTRIBUTE SET
UM_ATTR_VALUE=? WHERE UM_USER_ID=(SELECT UM_ID FROM UM_USER WHERE UM_USER_NAME=?
AND UM_TENANT_ID=?) AND UM_ATTR_NAME=? AND UM_PROFILE_ID=? AND
UM_TENANT_ID=?</Property>
      <Property name="DeleteUserPropertySQL">DELETE FROM UM_USER_ATTRIBUTE WHERE
UM_USER_ID=(SELECT UM_ID FROM UM_USER WHERE UM_USER_NAME=? AND UM_TENANT_ID=?)
AND UM_ATTR_NAME=? AND UM_PROFILE_ID=? AND UM_TENANT_ID=?</Property>
      <Property name="UserNameUniqueAcrossTenantsSQL">SELECT UM_ID FROM UM_USER
WHERE UM_USER_NAME=?</Property>
      <Property name="IsDomainExistingSQL">SELECT UM_DOMAIN_ID FROM UM_DOMAIN
WHERE UM_DOMAIN_NAME=? AND UM_TENANT_ID=?</Property>
      <Property name="AddDomainSQL">INSERT INTO UM_DOMAIN (UM_DOMAIN_NAME,
UM_TENANT_ID) VALUES (?, ?)</Property>
      <Property name="AddUserToRoleSQL-mssql">INSERT INTO UM_USER_ROLE
(UM_USER_ID, UM_ROLE_ID, UM_TENANT_ID) SELECT (SELECT UM_ID FROM UM_USER WHERE
UM_USER_NAME=? AND UM_TENANT_ID=?),(SELECT UM_ID FROM UM_ROLE WHERE
UM_ROLE_NAME=? AND UM_TENANT_ID=?),(?)</Property>
      <Property name="AddRoleToUserSQL-mssql">INSERT INTO UM_USER_ROLE
(UM_ROLE_ID, UM_USER_ID, UM_TENANT_ID) SELECT (SELECT UM_ID FROM UM_ROLE WHERE
UM_ROLE_NAME=? AND UM_TENANT_ID=?),(SELECT UM_ID FROM UM_USER WHERE
UM_USER_NAME=? AND UM_TENANT_ID=?), (?)</Property>
      <Property name="AddUserPropertySQL-mssql">INSERT INTO UM_USER_ATTRIBUTE
(UM_USER_ID, UM_ATTR_NAME, UM_ATTR_VALUE, UM_PROFILE_ID, UM_TENANT_ID) SELECT
(SELECT UM_ID FROM UM_USER WHERE UM_USER_NAME=? AND UM_TENANT_ID=?), (?), (?),
(?), (?)</Property>
      <Property name="AddUserToRoleSQL-openedge">INSERT INTO UM_USER_ROLE
(UM_USER_ID, UM_ROLE_ID, UM_TENANT_ID) SELECT UU.UM_ID, UR.UM_ID, ? FROM UM_USER
UU, UM_ROLE UR WHERE UU.UM_USER_NAME=? AND UU.UM_TENANT_ID=? AND
UR.UM_ROLE_NAME=? AND UR.UM_TENANT_ID=?</Property>
      <Property name="AddRoleToUserSQL-openedge">INSERT INTO UM_USER_ROLE
(UM_ROLE_ID, UM_USER_ID, UM_TENANT_ID) SELECT UR.UM_ID, UU.UM_ID, ? FROM UM_ROLE
UR, UM_USER UU WHERE UR.UM_ROLE_NAME=? AND UR.UM_TENANT_ID=? AND
UU.UM_USER_NAME=? AND UU.UM_TENANT_ID=?</Property>
      <Property name="AddUserPropertySQL-openedge">INSERT INTO UM_USER_ATTRIBUTE
(UM_USER_ID, UM_ATTR_NAME, UM_ATTR_VALUE, UM_PROFILE_ID, UM_TENANT_ID) SELECT
UM_ID, ?, ?, ?, ? FROM UM_USER WHERE UM_USER_NAME=? AND UM_TENANT_ID=?</Property>
```

```
      <Property name="DomainName">wso2.org</Property>
      <Property name="Description"/>
</UserStoreManager>
```

> The sample for the external JDBC user store consists of properties pertaining to various SQL statements. This is because the schema may be different for an external user store, and these adjustments need to be made in order to streamline the configurations with WSO2 products.

> You can define a "data source" in `repository/conf/datasources/master-datasources.xml` and refer to it from the `user-mgt.xml` file. This takes the properties defined in the `master-datasources.xml` file and reuses them in the `user-mgt.xml` file. To do this, you need to define the following property: `< Property name = "dataSource" >jdbc/WSO2CarbonDB</ Property >`

2. Find a valid user that resides in the RDBMS. For example, say a valid username is `AdminSOA`. Update the Admin user section of your LDAP configuration as follows. You do not have to update the password element; leave it as is.

```
<AdminUser>
   <UserName>AdminSOA</UserName>
   <Password>XXXXXX</Password>
</AdminUser>
```

3. In the `user-mgt.xml` file, add the `passwordHashMethod` property within the `JDBCUserStoreManager`. For example:

```
<UserStoreManager class="org.wso2.carbon.user.core.jdbc.JDBCUserStoreManager">
     <Property name="passwordHashMethod">SHA</Property>
     ...
</UserStoreManager>
```

   The `passwordHashMethod` property specifies how the password should be stored. It usually has the following values:
   - `SHA` - Uses SHA digest method.
   - `MD5` - Uses MD 5 digest method.
   - `PLAIN_TEXT` - Plain text passwords.

   In addition, it also supports all digest methods in http://docs.oracle.com/javase/6/docs/api/java/security/MessageDigest.html.
4. Update the connection details found within the `<UserStoreManager>` class based on your preferences.
5. In the `user-mgt.xml` file, under the realm configuration, set the value of the `MultiTenantRealmConfigBuilder` property to `org.wso2.carbon.user.core.config.multitenancy.SimpleRealmConfigBuilder`. For example:

```
<Property
name="MultiTenantRealmConfigBuilder">org.wso2.carbon.user.core.config.multitenanc
y.SimpleRealmConfigBuilder</Property>
```

6. Add the JDBC driver to the classpath by dropping the JAR into the `<PRODUCT_HOME>/repository/components/lib` directory.
7. Edit the SQLs in the `user-mgt.xml` file according to your requirements, and start the server.

Properties of Primary User Stores

The following can give you a better understanding of the properties used to configure primary user stores:

Using properties

Property name	Description
`MaxUserNameListLength`	This property controls the number of users listed in the user store of a WSO2 product. You might have hundreds or even thousands of users hence you may not want to list them all. While you have the ability to control hundreds of users with this property, you can use the number 0 as well.
`ConnectionURL`	Connection URL to the LDAP server. In the case of default LDAP in Carbon, the port is mentioned in the `carbon.xml` file and a reference to that port is mentioned in the above configuration.
`ConnectionName`	This is the username used to connect to the database. This user must have permissions to read the user list and user's attributes. This property is used to perform various operations on the external LDAP. In the case of `ReadOnlyLDAPUserStoreManager`, use this for search operations such as user searches or group searches on the external LDAP user store. This user does not have to be an administrator in the LDAP user store or have an administrator role in the WSO2 product that you are using, but this user MUST be able to do search operations on the LDAP user store. The value we put here is the DN (`DistinguishName`) attribute of the user. Note that this is a mandatory configuration.
`ConnectionPassword`	Password relevant to the `ConnectionName` of the user.
`passwordHashMethod`	Password Hash method when storing user entries in the LDAP.
`UserNameListFilter`	Filtering criteria for listing all the user entries in the LDAP. This LDAP query or filter is used when doing search operations on users. In this case, the search operation only provides the objects created from the specified class.
`UserEntryObjectClass`	Object class used to construct user entries. In the case of default LDAP in Carbon, it is a custom object class defined with the name- `wso2Person`
`UserSearchBase`	DN of the context or object under which the user entries are stored in the LDAP. In this case it is the "users" container. Different databases have different search bases.
`UserNameSearchFilter`	Filtering criteria for searching a particular user entry.

`UserNameAttribute`	This is the attribute used for uniquely identifying a user entry. Users can be authenticated using their email address, uid etc. The name of the attribute is considered as the username.
`PasswordJavaScriptRegEx`	Policy that defines the password format.
`UsernameJavaScriptRegEx`	The regular expression used by the front-end components for username validation.
`UsernameJavaRegEx`	A regular expression to validate usernames. By default, strings having a length between 5 to 30 with non-empty characters are allowed.
`RolenameJavaScriptRegEx`	The regular expression used by the front-end components for role name validation.
`RolenameJavaRegEx`	A regular expression to validate role names. By default, strings having a length between 5 to 30 with non-empty characters are allowed.
`ReadLDAPGroups`	Specifies whether groups should be read from LDAP. If this is disabled by setting it to `false`, none of the groups in the LDAP user store can be read. If you are setting the value of this to "false", the following group configurations are NOT mandatory: `GroupSearchBase`, `GroupNameListFilter` and `GroupNameAttribute`.
`WriteLDAPGroups`	Specifies whether groups should be written to LDAP.
`EmptyRolesAllowed`	Specifies whether the underlying LDAP user store allows empty groups to be created. In the case of LDAP in Carbon, the schema is modified such that empty groups are allowed to be created. Usually LDAP servers do not allow to create empty groups.
`GroupSearchBase`	DN of the context under which user entries are stored in the LDAP.
`GroupSearchFilter`	The LDAP query used to search for groups.
`GroupNameListFilter`	Filtering criteria for listing all the group entries in the LDAP. Groups are created using the "`groupOfName`" class. The group search operation only returns objects created from the above class.
`GroupEntryObjectClass`	Object class used to construct user entries.
`GroupNameSearchFilter`	Filtering criteria for searching a particular group entry.
`GroupNameAttribute`	Attribute used for uniquely identifying a user entry. This attribute is to be treated as the group name.
`MembershipAttribute`	Attribute used to define members of LDAP groups.
`UserRolesCacheEnabled`	This is to indicate whether to cache the role list of a user. By default this is set to `true`. Set it to `false` if the user roles are changed by external means and those changes should be instantly reflected in the Carbon instance.

`UserDNPattern`	The patten for user's DN. It can be defined to improve the LDAP search. When there are many user entries in the LADP, defining a `UserDNPattern` provides more impact on performances as the LDAP does not have to travel through the entire tree to find users.
`ReplaceEscapeCharactersAtUserLogin`	If the user name has special characters it replaces it to validate the user logging in. Only "****" and "****" are identified as escape characters.
`TenantManager`	Includes the location of the tenant manager.
`ReadOnly`	Indicates whether the user store of this realm operates in the user read only mode or not.
`IsEmailUserName`	Indicates whether the user's email is used as their username (apply when realm operates in read only mode).
`DomainCalculation`	Can be either default or custom (this applies when the realm operates in read only mode).
`PasswordDigest`	Digesting algorithm of the password. Has values such as, PLAIN_TEXT, SHA etc.
`StoreSaltedPassword`	Indicates whether to salt the password.
`UserNameUniqueAcrossTenants`	An attribute used for multi-tenancy.
`PasswordJavaRegEx`	A regular expression to validate passwords. By default, strings having a length between 5 to 30 with non-empty characters are allowed.
`PasswordJavaScriptRegEx`	The regular expression used by the front-end components for password validation.
`UsernameJavaRegEx`	A regular expression to validate usernames. By default, strings having a length 5 to 30 between with non-empty characters are allowed.
`UsernameJavaScriptRegEx`	The regular expression used by the front-end components for username validation.
`RolenameJavaRegEx`	A regular expression to validate role names. By default, strings having a length between 5 to 30 with non-empty characters are allowed.
`RolenameJavaScriptRegEx`	The regular expression used by the front-end components for rolename validation.
`MultiTenantRealmConfigBuilder`	Tenant Manager specific realm config parameter. Can be used to build different types of realms for the tenant.

Configuring Secondary User Stores

The default configurations of WSO2 products have a single, embedded user store. If required, you can configure WSO2 products to connect to several secondary user stores as well. After configuration, users from different stores can log in and perform operations depending on their roles/permissions. You can also configure your own customized user stores and connect them with the products as secondary stores.

The topics below explain how to configure secondary user stores manually or using the management console:

- Configuring using the management console
- Configuring manually

Tip: If you set up a database other than the default H2 that comes with the product to store user information, select the script relevant to your database type from the `<APIM_HOME>/dbscripts` folder and run it on your database. It creates the necessary tables.

Configuring using the management console

1. Log in to the management console and click **User Store Management** sub menu under **Configure** menu.
2. The **User Store Management** page opens. Initially, there are no secondary user stores.

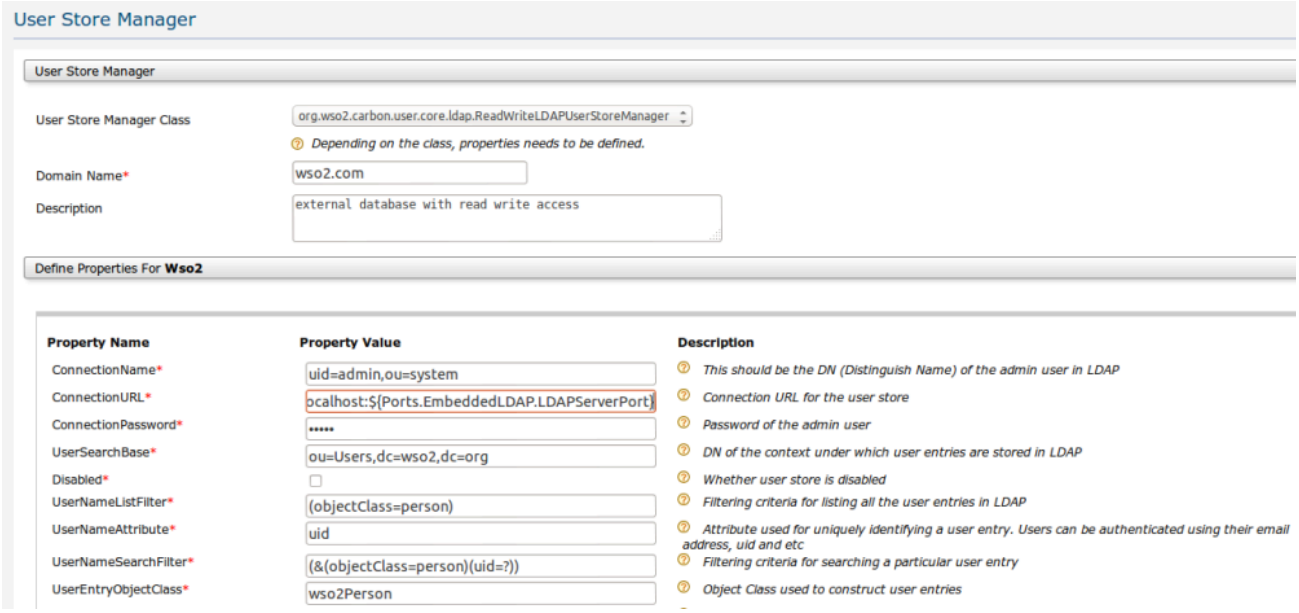
   Note: You cannot update the PRIMARY user store at run time, so it is not visible on this page.

3. Click **Add Secondary User Store**.
4. The **User Store Manager** page opens. Enter a unique domain name and fill in the rest of the data.

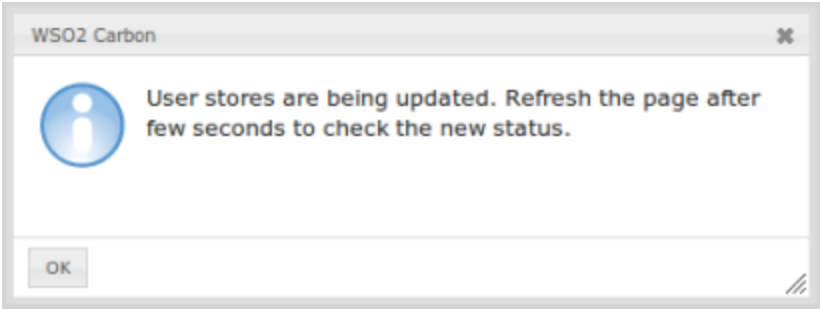
   Domain names must be unique and must not include underscore character (_).

   For details on each property, see the respective property description that is provided. Also, select the required implementation of user store manager from the **User Store Manager Class** drop-down list. The displayed property list varies depending on the selected user store manager implementation. By default, all WSO2 products come with four user store manager implementations as follows:
   - ReadWriteLDAPUserStoreManager
   - ReadOnlyLDAPUserStoreManager
   - ActiveDirectoryUserStoreManager
   - JDBCUserStoreManager

   You can also populate this drop-down list with custom user store manager implementations by adding them to the server. A sample custom user store manager can be found in the repository.



5. Ensure that all the mandatory fields are filled and a valid domain name is given and click **Add**.
6. A message appears saying that the user stores are being added.



> **Note**: The above message does not imply that the user store is added successfully. It simply means that the server is attempting to add the new user store to the end of the available chain of stores.

7. Refresh the page after a few seconds to check the status.
8. If the new user store is successfully added, it will appear in the **User Store Management** page.
9. After adding to the server, you can edit the properties of the new secondary user store and enable/disable it in a dynamic manner.

Configuring manually

By default, the configuration of the primary user store is saved in the `user-mgt.xml` file. When you create a secondary user store using the management console as explained above, its configuration is saved to an XML file with the same name as the domain name you specify. Alternatively, you can create this XML file manually and save it as follows:

- - When you configure multiple user stores, you must **give a unique domain name to each user store** in the `<DomainName>` element. If you configure a user store without specifying a domain name, the server throws an exception at start up.
  - If it is the configuration of a super tenant, save the secondary user store definitions in `<PRODUCT_HOME>/repository/deployment/server/userstores` directory.
  - If it is a general tenant, save the configuration in `<PRODUCT_HOME>/repository/tenants/<tenantid>/userstores` directory.
  - The the secondary user store configuration file must have the same name as the domain with an underscore (_) in place of the period. For example, if the domain is `wso2.com`, name the file as `wso2_com.xml`.
  - One file only contains the definition for one user store domain.

Deploying and Clustering the API Manager

You can install multiple instances of WSO2 products in a cluster to ensure proper load balancing. When one instance becomes unavailable or is experiencing high traffic, another instance handles the requests. For complete information on clustering, see Clustering WSO2 API Manager.

Working with Databases

The default database of user manager is the H2 database that comes with WSO2 products. You can configure it to point to databases by other vendors such as IBM DB2, Oracle, MySQL using the scripts provided by WSO2 for installing and configuring relational databases. The scripts in **`<PRODUCT_HOME>/dbscript`** folder are available in all WSO2 products. They store data related to WSO2 Carbon, on top of which all WSO2 products are built. There is a separate set of database scripts in **`<PRODUCT_HOME>/dbscript/apimgt`** folder. These scripts are to create databases that store API Manager-specific data.

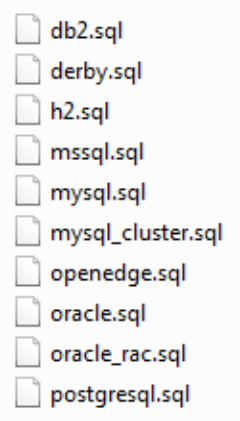
Each database you create supports stored procedures, which allow business logic to be embedded inside the database as an API, providing a powerful mechanism for interacting with a relational database. Because stored procedures are stored in a precompiled format within the database itself, execution speed is much faster. Client programs can be restricted to accessing a database via stored procedures only, thereby enforcing fine-grained security, preservation of data integrity, and improved productivity.

After you set up the physical database, you add data sources in the Management Console to enable the server to connect to that database.

> The embedded H2 database is suitable for development, testing, and some production environments. For most enterprise production environments, however, we recommend you use an industry-standard RDBMS such as Oracle, PostgreSQL, MySQL, MS SQL, etc.

Setting up the Physical Database

In the `<PRODUCT_HOME>/dbscripts` folder, the following scripts are available for installing and configuring a database.



The following topics describe how to use these scripts to set up each type of physical database. After you set up the database, you can use the Management Console to create the datasources that connect to that database.

- Setting up with Derby
- Setting up with H2 Database
- Setting up with MS SQL
- Setting up with MySQL
- Setting up with MySQL Cluster
- Setting up with OpenEdge
- Setting up with Oracle
- Setting up with PostgreSQL

Setting up with Derby

You can set up either an embedded Derby database or a remote one according to the information given below:

- Setting up with Embedded Derby
- Setting up with Remote Derby

Setting up with Embedded Derby

Follow the instructions below to set up an embedded Derby database.

Preparing the Derby Database | Setup Configuration Files | Setup Drivers | Create Database

---

Preparing the Derby Database

1. Download Apache Derby from http://apache.mesi.com.ar/db/derby/db-derby-10.8.2.2/ and save it to your computer.

2. Install Apache Derby on your computer by following the instructions at: http://db.apache.org/derby/manuals

Setup Configuration Files

1. Edit the default database configuration defined in the `master-datasources.xml` file located at `<PRODUCT_HOME>/repository/conf/datasources` directory as below. Both the database configurations in `registry.xml` and `user-mgt.xml` refer this data source.

```
<datasources-configuration xmlns:svns="http://org.wso2.securevault/configuration">

    <providers>
        <provider>org.wso2.carbon.ndatasource.rdbms.RDBMSDataSourceReader</provider>
    </providers>

    <datasources>
       <datasource>
            <name>WSO2_CARBON_DB</name>
            <description>The datasource used for registry and user manager</description>
            <jndiConfig>
                <name>jdbc/WSO2CarbonDB</name>
            </jndiConfig>
            <definition type="RDBMS">
                <configuration>
                    <url>jdbc:derby://localhost:1527/db;create=true</url>
                    <userName>regadmin</userName>
                    <password>regadmin</password>
                    <driverClassName>org.apache.derby.jdbc.EmbeddedDriver</driverClassName>
                    <maxActive>80</maxActive>
                    <maxWait>60000</maxWait>
                    <minIdle>5</minIdle>
                    <testOnBorrow>true</testOnBorrow>
                    <validationQuery>SELECT 1</validationQuery>
                    <validationInterval>30000</validationInterval>
                </configuration>
            </definition>
        </datasource>
```

Note

The configurations should be replaced with your own database name, username, and password.

```
<datasource>
     <name>WSO2_CARBON_DB</name>
     <description>The datasource used for registry and user manager</description>
     <jndiConfig>
         <name>jdbc/WSO2CarbonDB</name>
     </jndiConfig>
     <definition type="RDBMS">
         <configuration>
               <url>jdbc:derby://localhost:1527/db;create=true</url>
               <userName>regadmin</userName>
               <password>regadmin</password>
               <driverClassName>org.apache.derby.jdbc.EmbeddedDriver</driverClassName>
               <maxActive>80</maxActive>
               <maxWait>60000</maxWait>
               <minIdle>5</minIdle>
               <testOnBorrow>true</testOnBorrow>
               <validationQuery>SELECT 1</validationQuery>
               <validationInterval>30000</validationInterval>
         </configuration>
     </definition>
</datasource>
```

The database configuration options

- **url** - The URL of the database.
- **username** - The name of the database user.
- **password** - The password of the database user.
- **driverClassName** - The class name of the database driver.
- **maxActive** - The maximum number of active connections that can be allocated from this pool at the same time or negative for no limit.
- **maxWait** - The maximum number of milliseconds that the pool will wait (when there are no available connections) for a connection to be returned before throwing an exception or <= 0 to wait indefinitely.
- **minIdle** - The minimum number of active connections that can remain idle in the pool, without extra ones being created, or 0 to create none.

Note

In contrast to the remote Derby, in embedded mode, you will set the database driver name (the `driverName` element) to the value `org.apache.derby.jdbc.EmbeddedDriver` and the database URL (the `url` element) to the database directory location relative to the installation. In the above sample configuration, it is inside the `database/WSO2CARBON_DB` directory.

Setup Drivers

Copy `derby.jar`, `derbyclient.jar` and `derbynet.jar` from `$DERBY_HOME/lib` in to `$CARBON_HOME/repository/components/extensions` directory (to the class path of the WSO2 Carbon web application).

Create Database

Automatic Database Creation

1. The first time you start the server, run with the `-Dsetup` option so it will create the Derby database.

- For Linux:

```
wso2server.sh -Dsetup
```

- For Windows:

```
wso2server.bat -Dsetup
```

2. The product is configured to run using an embedded Apache Derby database.

Manual Database Creation

1. Run the `ij` tool located in the `<derby-installation-directory>/bin` directory.

```
client@wso2:~/dtb/db-derby-10.8.1.2-bin/bin$ ./ij
ij version 10.8
ij>
```

2. Create the database and connect to it using the following command inside the `ij` prompt.

```
ij> connect  jdbc:derby:repository/database/WSO2CARBON_DB;create=true'
```

> **Note**
>
> Replace the database file path in the below command to suit your requirements.

```
connect 'jdbc:derby:repository/database/WSO2CARBON_DB;create=true';
```

> **Note**
>
> Here you need to give the full path to your database in place of `/WSO2CARBON_DB`.

3. Exit from the the `ij` tool by typing the `exit` command.

```
exit;
```

```
ij> exit
> ;
client@wso2:~/dtb/db-derby-10.8.1.2-bin/bin$
```

4. Login to the `ij` tool with the username and password you set in the `registry.xml` and `user-mgt.xml`.

```
connect 'jdbc:derby:repository/database/WSO2CARBON_DB' user 'regadmin' password
'regadmin';
```

```
ij> connect 'jdbc:derby:repository/database/WSO2CARBON_DB' user 'regadmin' password 'regadmin'
```

5. Run the derby scripts for both the registry and user manager (embedded) databases, provided with the product using the below command.

```
run 'CARBON_HOME/dbscripts/derby.sql';
```

6. Restart the server. Now the product is running using a remote Apache Derby database.

Setting up with Remote Derby

Follow the below instructions to set up the remote Derby database.

Preparing the Derby Database | Setup Configuration Files | Setup Drivers | Create Database

Preparing the Derby Database

1. Download Apache Derby from http://apache.mesi.com.ar/db/derby/db-derby-10.8.2.2/ and save it to your computer.

2. Install Apache Derby on your computer using instructions given at http://db.apache.org/derby/manuals.

3. Go to the `<derby-installation directory>/bin` directory and run the Derby network server start script. Usually, it is named `startNetworkServer`.

Setup Configuration Files

1. Edit the default database configuration defined in the `master-datasources.xml` file located at `$CARBON_HOME/repository/conf/datasources` directory as below. Both the database configurations in `registry.xml` and `user-mgt.xml` refer this data source.

```
<datasources-configuration xmlns:svns="http://org.wso2.securevault/configuration">

    <providers>
        <provider>org.wso2.carbon.ndatasource.rdbms.RDBMSDataSourceReader</provider>
    </providers>

    <datasources>

        <datasource>
            <name>WSO2_CARBON_DB</name>
            <description>The datasource used for registry and user manager</description>
            <jndiConfig>
                <name>jdbc/WSO2CarbonDB</name>
            </jndiConfig>
            <definition type="RDBMS">
                <configuration>
                    <url>jdbc:derby://localhost:1527/db;create=true</url>
                    <userName>regadmin</userName>
                    <password>regadmin</password>
                    <driverName>org.apache.derby.jdbc.ClientDriver</driverName>
                    <maxActive>80</maxActive>
                    <maxWait>60000</maxWait>
                    <minIdle>5</minIdle>
                    <testOnBorrow>true</testOnBorrow>
                    <validationQuery>SELECT 1</validationQuery>
                    <validationInterval>30000</validationInterval>
                </configuration>
            </definition>
        </datasource>
```

Note

Replace the following settings with your own custom values:

```
<datasource>
     <name>WSO2_CARBON_DB</name>
     <description>The datasource used for registry and user manager</description>
     <jndiConfig>
           <name>jdbc/WSO2CarbonDB</name>
     </jndiConfig>
     <definition type="RDBMS">
           <configuration>
               <url>jdbc:derby://localhost:1527/db;create=true</url>
               <userName>regadmin</userName>
               <password>regadmin</password>
               <driverClassName>org.apache.derby.jdbc.ClientDriver</driverClassName>
               <maxActive>80</maxActive>
               <maxWait>60000</maxWait>
               <minIdle>5</minIdle>
               <testOnBorrow>true</testOnBorrow>
               <validationQuery>SELECT 1</validationQuery>
               <validationInterval>30000</validationInterval>
           </configuration>
     </definition>
</datasource>
```

The Database Configuration Options

- **url** - The URL of the database.
- **username** - The name of the database user.
- **password** - The password of the database user.
- **driverClassName** - The class name of the database driver.
- **maxActive** - The maximum number of active connections that can be allocated from this pool at the same time or negative for no limit.
- **maxWait** - The maximum number of milliseconds that the pool will wait (when there are no available connections) for a connection to be returned before throwing an exception, or <= 0 to wait indefinitely.
- **minIdle** - The minimum number of active connections that can remain idle in the pool without extra ones being created or 0 to create none.

Note

In contrast to the embedded Derby, in the remote registry, you will set the database driver name (the `driverName` element) to the value `org.apache.derby.jdbc.ClientDriver` and the database URL (the `url` element) to the database remote location.

Setup Drivers

Copy `derby.jar`, `derbyclient.jar` and `derbynet.jar` from `$DERBY_HOME/lib` in to `$CARBON_HOME/repository/components/extensions` directory (to the class path of the Web application).

Create Database

Automatic Database Creation

1. The first time you start the server, run it with the -Dsetup option so that it will create the Derby database.

- For Windows:

```
wso2server.bat -Dsetup
```

- For Linux:

```
wso2server.sh -Dsetup
```

2. The product is configured to run using a remote Apache Derby database.

Manual Database Creation

1. Run the `ij` tool located in the `<derby-installation directory>/bin` directory.

2. Create the database and connect to it using the following command inside the `ij` prompt:

> **Note**
>
> Replace the database file path, user name, and password in the below command to suit your requirements.

```
connect 'jdbc:derby://localhost:1527/db;user=regadmin;password=regadmin;create=true';
```

3. Exit from the `ij` tool by typing the exit command.

```
exit;
```

4. Log in to the `ij` tool with the username and password you just used to create the database.

```
connect 'jdbc:derby://localhost:1527/db' user 'regadmin' password 'regadmin';
```

5. Run the Derby scripts for both the registry and user manager (embedded) databases, provided with the product

using the following command:

```
run 'CARBON_HOME/dbscripts/derby.sql';
```

```
client@wso2:~/dtb/db-derby-10.8.1.2-bin/bin$ ./ij
ij version 10.8
ij> connect 'jdbc:derby://localhost:1527/our_new_db' user 'regadmin' password 'regadmin';
ij> run '/home/client/wso2/derby/greg/wso2greg-4.1.1/dbscripts/derby.sql';
```

6. Restart the server. Now the product is running using a remote Apache Derby database.

Setting up with H2 Database

You can set up either an embedded H2 database or a remote one using the instructions given below:

- Setting up with Embedded H2
- Setting up with Remote H2

Setting up with Embedded H2

Follow the instructions below to set up an embedded H2 database.

Preparing the Embedded H2 Database | Setup configuration Files | Setup Drivers | Create Database

Preparing the Embedded H2 Database

Download and install the H2 database in your computer, if it is not already done, using the installation guide at http://www.h2database.com/html/quickstart.html.

Setup configuration Files

1. Edit the default database configuration defined in the `master-datasources.xml` file located at `$CARBON_HOME/repository/conf/datasources` directory as below. Both the database configurations in `registry.xml` and `user-mgt.xml` refer this data source.

```
<datasources-configuration xmlns:svns="http://org.wso2.securevault/configuration">

    <providers>
        <provider>org.wso2.carbon.ndatasource.rdbms.RDBMSDataSourceReader</provider>
    </providers>

        <datasource>
            <name>WSO2_CARBON_DB</name>
            <description>The datasource used for registry and user manager</description>
            <jndiConfig>
                <name>jdbc/WSO2CarbonDB</name>
            </jndiConfig>
            <definition type="RDBMS">
                <configuration>
                    <url>jdbc:h2:repository/database/WSO2CARBON_DB;DB_CLOSE_ON_EXIT=FALSE;LOCK_TIMEOUT=60000</url>
                    <username>wso2carbon</username>
                    <password>wso2carbon</password>
                    <driverClassName>org.h2.Driver</driverClassName>
                    <maxActive>50</maxActive>
                    <maxWait>60000</maxWait>
                    <testOnBorrow>true</testOnBorrow>
                    <validationQuery>SELECT 1</validationQuery>
                    <validationInterval>30000</validationInterval>
                </configuration>
            </definition>
        </datasource>
```

Note

The configurations should be replaced with your own database name, username, and password.

```
<datasource>
    <name>WSO2_CARBON_DB</name>
    <description>The datasource used for registry and user manager</description>
    <jndiConfig>
         <name>jdbc/WSO2CarbonDB</name>
    </jndiConfig>
    <definition type="RDBMS">
         <configuration>

<url>jdbc:h2:repository/database/WSO2CARBON_DB;DB_CLOSE_ON_EXIT=FALSE;LOCK_TIMEOUT=600
00</url>
             <username>wso2carbon</username>
             <password>wso2carbon</password>
             <driverClassName>org.h2.Driver</driverClassName>
             <maxActive>50</maxActive>
             <maxWait>60000</maxWait>
             <testOnBorrow>true</testOnBorrow>
             <validationQuery>SELECT 1</validationQuery>
             <validationInterval>30000</validationInterval>
         </configuration>
    </definition>
</datasource>
```

The database configuration options.

- **url** - The URL of the database.
- **userName** - The name of the database user.
- **password** - The password of the database user.
- **driverClassName** - The class name of the database driver.
- **maxActive** - The maximum number of active connections that can be allocated from this pool at the same time or negative for no limit.
- **maxWait** - The maximum number of milliseconds that the pool will wait (when there are no available connections) for a connection to be returned before throwing an exception, or <= 0 to wait indefinitely.
- **minIdle** - The minimum number of active connections that can remain idle in the pool, without extra ones being created, or 0 to create none.

Setup Drivers

Tip

Currently, H2 database version h2-1.2.140.* and the related H2 database driver are shipped with the product.

If you wish to use a new H2 database driver other than the version shipped with the product, follow the steps below.

1. Delete H2 database-related JARs shipped with the product. One could find them in the following locations.

- `<PRODUCT_HOME>/repository/components/plugins/h2-database-engine_1.2.140.wso2v3.jar`

2. Copy the new H2 database driver (`org.h2.Driver`) to `<PRODUCT_HOME>/repository/components/lib`. One could find the required driver JAR in `$H2_HOME/bin/h2-*.jar`.

> **Tip**
>
> `$H2_HOME` is the installation directory of H2.

Create Database

Automatic Database Creation

Next, at the first time you start the server, run with the `-Dsetup` option. It will create the H2 database with all the underlying tables.

- For Linux:

```
sh wso2server.sh -Dsetup
```

- For Windows:

```
wso2server.bat -Dsetup
```

Manual Table Creation using scripts

Tables can be manually created by logging into the created database and running and following script in H2 shell or Web Console.

```
<PRODUCT_HOME>/dbscripts/h2.
```

After setting up the DB tables, start the server with the below commands.

- For Linux:

```
sh wso2server.sh
```

- For Windows:

```
wso2server.bat
```

Setting up with Remote H2

Follow the instructions below to set up a remote H2 database.

Preparing the remote H2 DB | Setup Configuration Files | Setup Drivers | Create Database

Preparing the remote H2 DB

1. Download and install the H2 database on your computer. H2 installation guide can be found at: http://www.h2database.com/html/quickstart.html.

```
client@wso2:~/dtb$ wget -c http://www.h2database.com/h2-2011-09-11.zip
--2011-09-30 00:28:27--  http://www.h2database.com/h2-2011-09-11.zip
Resolving www.h2database.com... 80.74.147.171
Connecting to www.h2database.com|80.74.147.171|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 6007851 (5.7M) [application/zip]
Saving to: "h2-2011-09-11.zip"

15% [====================>            ] 923,304      111K/s  eta 45s
```

2. Go to the `$H2_HOME/bin` directory and run the H2 network server starting script.

```
client@wso2:~/dtb/h2/bin$ chmod 0744 h2.sh
```

> **Tip**
>
> `$H2_HOME` is the installation directory of H2.

3. Run the H2 database server with the following commands.

- For Linux:

```
$ ./h2.sh
```

```
client@wso2:~/dtb/h2/bin$ ./h2.sh
```

- For Windows:

```
$ h2.bat
```

The script will start the database engine and bring up a pop-up window with a "Start Browser" button. The "Start Browser" button will open a web browser containing a client application, which you can connect to a database. H2 will automatically create a database if a database does not exist by the name you provide in the "JDBC URL" text box.

Setup Configuration Files

1. Edit the default database configuration defined in the `master-datasources.xml` file located at `$CARBON_HOME/repository/conf/datasources` directory instance as follows. Both the database configurations in `registry.xml` and `user-mgt.xml` refer this data source.

```
<datasources-configuration xmlns:svns="http://org.wso2.securevault/configuration">

    <providers>
        <provider>org.wso2.carbon.ndatasource.rdbms.RDBMSDataSourceReader</provider>
    </providers>

    <datasources>
       <datasource>
            <name>WSO2_CARBON_DB</name>
            <description>The datasource used for registry and user manager</description>
            <jndiConfig>
                <name>jdbc/WSO2CarbonDB</name>
            </jndiConfig>
            <definition type="RDBMS">
                <configuration>
                    <url>jdbc:h2:tcp://localhost/~/registryDB;create=true</url>
                    <userName>regadmin</userName>
                    <password>regadmin</password>
                    <driverName>org.h2.Driver</driverName>
                    <maxActive>80</maxActive>
                    <maxWait>60000</maxWait>
                    <minIdle>5</minIdle>
                    <testOnBorrow>true</testOnBorrow>
                    <validationQuery>SELECT 1</validationQuery>
                    <validationInterval>30000</validationInterval>
                </configuration>
            </definition>
        </datasource>
```

Note

The configurations should be replaced with your own database name, username, and password.

```
<datasource>
      <name>WSO2_CARBON_DB</name>
      <description>The datasource used for registry and user manager</description>
      <jndiConfig>
           <name>jdbc/WSO2CarbonDB</name>
      </jndiConfig>
      <definition type="RDBMS">
           <configuration>
               <url>jdbc:h2:tcp://localhost/~/registryDB;create=true</url>
               <username>regadmin</username>
               <password>regadmin</password>
               <driverClassName>org.h2.Driver</driverClassName>
               <maxActive>80</maxActive>
               <maxWait>60000</maxWait>
               <minIdle>5</minIdle>
               <testOnBorrow>true</testOnBorrow>
               <validationQuery>SELECT 1</validationQuery>
               <validationInterval>30000</validationInterval>
           </configuration>
      </definition>
</datasource>
```

The database configuration options

- **url** - The URL of the database.
- **username** - The name of the database user.
- **password** - The password of the database user.
- **driverClassName** - The class name of the database driver.
- **maxActive** - The maximum number of active connections that can be allocated from this pool at the same time or negative for no limit.
- **maxWait** - The maximum number of milliseconds that the pool will wait (when there are no available connections) for a connection to be returned before throwing an exception, or <= 0 to wait indefinitely.
- **minIdle** - The minimum number of active connections that can remain idle in the pool, without extra ones being created, or 0 to create none.

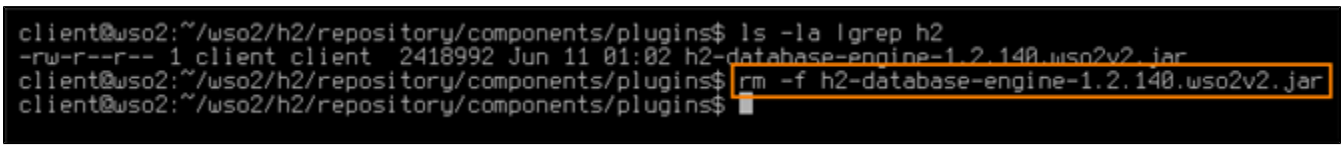
Setup Drivers

> **Tip**
>
> The H2 database version h2-1.2.140 and the related H2 database driver are currently shipped with the product.

To use a new H2 database driver other than the version shipped with the product, do the following.

1. Delete the following H2 database related JARs. Some of them may already be excluded from the configuration.

- `<PRODUCT_HOME>/repository/components/plugins/h2-database-engine_1.2.140.wso2v3.jar`



```
client@wso2:~/wso2/h2/lib$ rm -f h2-database-engine-1.2.140.wso2v2.jar
```

2. Copy the new H2 database driver (`org.h2.Driver`) to `<PRODUCT_HOME>/repository/components/lib`. You can find the required driver JAR in `$H2_HOME/bin/h2-*.jar`.

```
client@wso2:~/dtb/h2/bin$ cp h2-1.3.160.jar /home/client/wso2/h2/repository/components/lib/
```

Create Database

Automatic Database Creation

1. The first time you start the server, run it with the `-Dsetup` option. It will create the H2 database with all the underlying tables.

- For Linux:

```
sh wso2server.sh -Dsetup
```

- For Windows:

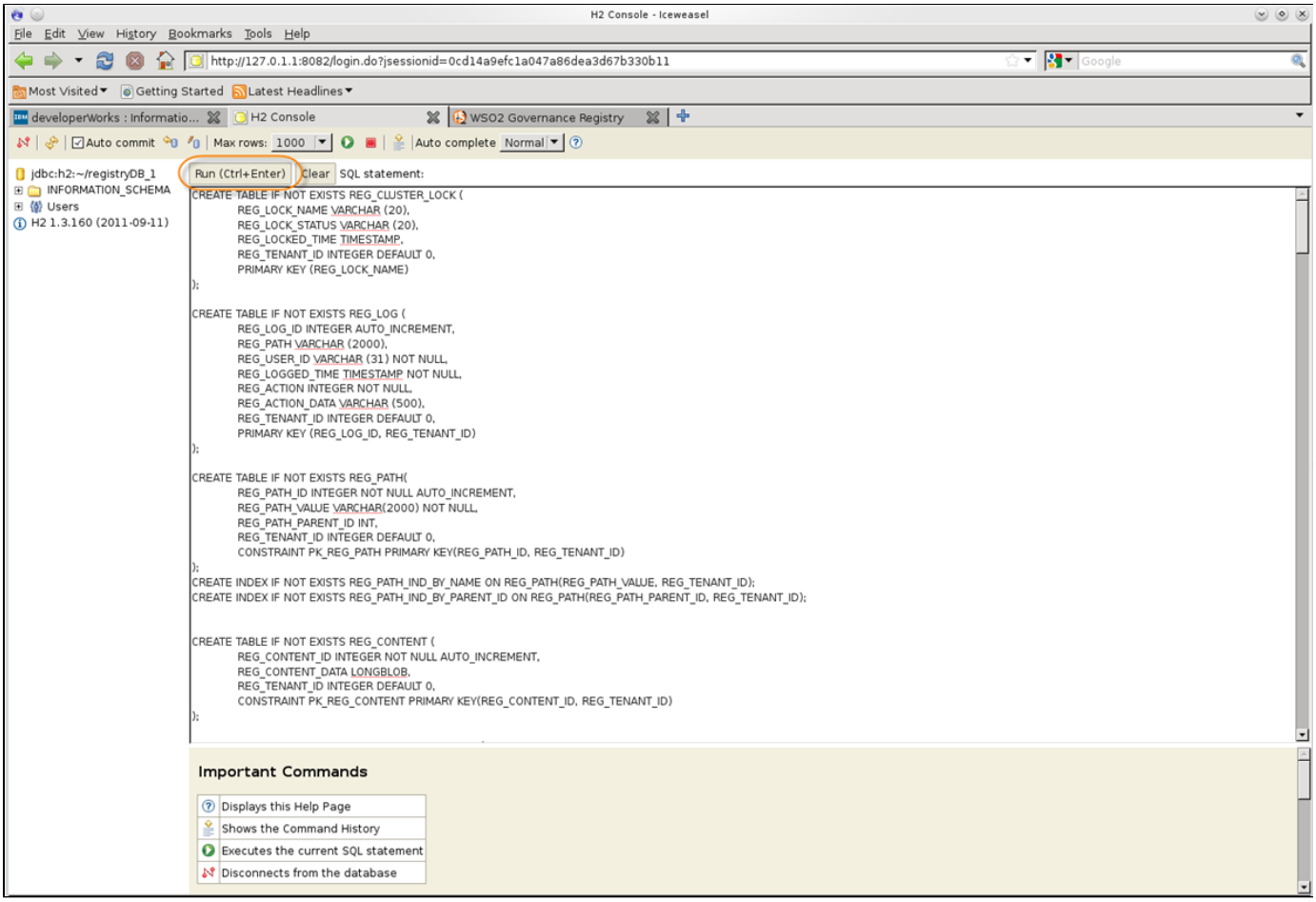
```
wso2server.bat -Dsetup
```

Manual Table Creation using scripts

1. Tables can be manually created by logging into the created database and running the following script in H2 shell or web console.

> **Note**
>
> Use the `./h2.sh` command to start the web console. After that copy the script text from the SQL file, paste it into the console and click "Run."

```
PRODUCT_HOME/dbscripts/h2.sql
```



2. After setting up the DB tables, start the server with the following commands.

- For Linux:

```
sh wso2server.sh
```

- For Windows:

```
wso2server.bat
```

Setting up with MS SQL

Follow the instructions below to set up the MS SQL database.

Setup Database and User | Setup Configuration File | Copy JDBC Driver | Create Database Tables

Setup Database and User

Enable TCP/IP

1. Open "SQL Server Configuration Manager" from Start › Programs › Microsoft SQL Server 2005 › Configuration Tools › SQL Server Configuration Manager.

2. Enable "TCP/IP" and disable "Named Pipes" from protocols of your MSSQL server.

3. Open TCP/IP Properties by double clicking "TCP/IP." Set "Listen All" to "Yes" in the "Protocol" tab.

4. From the "IP Address" tab, disable "TCP Dynamic Ports" by leaving it blank and give a valid "TCP Port" so that MSSQL server will listen in that port.

> **Tip**
>
> You can use port 1433 in order processor services, so it is better to use that port.

5. Similarly, enable TCP/IP from "SQL Native Client Configuration" and disable "Named Pipes." Also, check whether the port is set correctly. Port should be 1433.

6. Restart MSSQL Server.

Create Database and User

1. Open "MSSQL Management Studio" to create a database and user.

2. Go to Database › New Database and specify all the options to create a new database.

3. Go to Logins › New Login and specify all the necessary options.

Grant Permission

Give required grants/permission to newly created user. Grant should allow newly created user to login, create tables, insert data to tables in newly created database.

Setup Configuration File

1. Edit the default database configuration defined in the `master-datasources.xml` file located at `<PRODUCT_HOME>/repository/conf/datasources` directory as below. Both the database configurations in `registry.xml` and `user-mgt.xml` refer this data source.

> Be sure to replace these settings with your custom values.

```
<datasource>
    <name>WSO2_CARBON_DB</name>
    <description>The datasource used for registry and user manager</description>
    <jndiConfig>
           <name>jdbc/WSO2CarbonDB</name>
    </jndiConfig>
    <definition type="RDBMS">
           <configuration>
                <defaultAutoCommit>false</defaultAutoCommit>
                <url>jdbc:jtds:sqlserver://10.100.3.251:1433/wso2greg</url>
                <username>regadmin</username>
                <password>regadmin</password>
                <driverClassName>net.sourceforge.jtds.jdbc.Driver</driverClassName>
                <maxActive>80</maxActive>
                <maxWait>60000</maxWait>
                <minIdle>5</minIdle>
                <testOnBorrow>true</testOnBorrow>
                <validationQuery>SELECT 1</validationQuery>
                <validationInterval>30000</validationInterval>
           </configuration>
    </definition>
</datasource>
```

The database configuration options

- **defaultAutoCommit** - Set to false.
- **url** - The URL of the database.
- **username** - The name of the database user.
- **password** - The password of the database user.
- **driverClassName** - The class name of the database driver.
- **maxActive** - The maximum number of active connections that can be allocated from this pool at the same time or negative for no limit.
- **maxWait** - The maximum number of milliseconds that the pool will wait (when there are no available connections) for a connection to be returned before throwing an exception, or <= 0 to wait indefinitely.
- **minIdle** - The minimum number of active connections that can remain idle in the pool without extra ones being created or 0 to create none.

> **Tip**
>
> Default port for MSSQL is 1433.

Copy JDBC Driver

Download and copy the sqljdbc4 Microsoft SQL JDBC driver file to the WSO2 product's `<PRODUCT_HOME>/repository/components/lib/` directory. Use`com.microsoft.sqlserver.jdbc.SQLServerDriver` as the `<driverClassName>` in your datasource configuration in`<PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml` file.

Create Database Tables

Database tables can be created either manually by running scripts or automatically by using start-up parameters.

- **Using Scripts**

Database tables can be created manually by logging in to created database and running `<PRODUCT_HOME>/dbscripts/mssql.sql`.

- **Using start-up Parameters**

Windows users can run `<PRODUCT_HOME>/bin/wso2server.bat -Dsetup` to create the database tables when starting the product for the first time.

Linux users should use `<PRODUCT_HOME>/bin/wso2server.sh -Dsetup`.

Setting up with MySQL

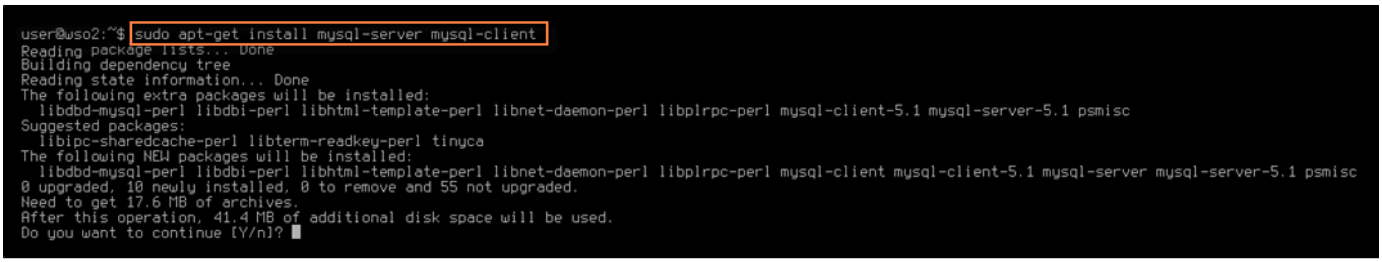
Follow the below instructions to set up a MySQL database.

Setup Database and the Database User | Setup Configuration Files | Setup Drivers | Create Database

Setup Database and the Database User

1. Download and install MySQL on your computer. Use the following command:

```
sudo apt-get install mysql-server mysql-client
```



2. Start the MySQL service using the following command:

```
sudo /etc/init.d/mysql start
```

3. Log in to the MySQL client as the root user (or any other user with database creation privileges).

```
mysql -u root -p
```

4. You will be prompted to enter the password.

> **Tip**
>
> In most systems, the default root password is blank. Press "enter" without typing anything if you have not changed the default root password.

After that, you will see the MySQL command prompt.

```
user@wso2:~$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 40
Server version: 5.1.49-3 (Debian)

Copyright (c) 2000, 2010, Oracle and/or its affiliates. All rights reserved.
This software comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to modify and redistribute it under the GPL v2 license

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

5. In the MySQL prompt, create the database using the following command.

```
create database regdb;
```

6. Give authorization of the database to the user "regadmin."

```
GRANT ALL ON regdb.* TO regadmin@localhost IDENTIFIED BY "regadmin"
```

7. Log out from the MySQL prompt by typing the "quit" command.

```
quit;
```

Setup Configuration Files

1. Edit the default database configuration defined in the `master-datasources.xml` file located at `$CARBON_HOME/repository/conf/datasources` directory as below. Both the database configurations in `registry.xml` and `user-mgt.xml` refer this data source.

```
<datasources-configuration xmlns:svns="http://org.wso2.securevault/configuration">

    <providers>
        <provider>org.wso2.carbon.ndatasource.rdbms.RDBMSDataSourceReader</provider>
    </providers>

    <datasources>
       <datasource>
            <name>WSO2_CARBON_DB</name>
            <description>The datasource used for registry and user manager</description>
            <jndiConfig>
                <name>jdbc/WSO2CarbonDB</name>
            </jndiConfig>
            <definition type="RDBMS">
                <configuration>
                    <url>jdbc:mysql://localhost:3306/regdb</url>
                    <userName>regadmin</userName>
                    <password>regadmin</password>
                    <driverClassName>com.mysql.jdbc.Driver</driverClassName>
                    <maxActive>80</maxActive>
                    <maxWait>60000</maxWait>
                    <minIdle>5</minIdle>
                    <testOnBorrow>true</testOnBorrow>
                    <validationQuery>SELECT 1</validationQuery>
                    <validationInterval>30000</validationInterval>
                </configuration>
            </definition>
        </datasource>
```

Note

Replace these settings with your own custom values:

```
<datasource>
       <name>WSO2_CARBON_DB</name>
       <description>The datasource used for registry and user manager</description>
       <jndiConfig>
           <name>jdbc/WSO2CarbonDB</name>
       </jndiConfig>
       <definition type="RDBMS">
           <configuration>
               <url>jdbc:mysql://localhost:3306/regdb</url>
               <username>regadmin</username>
               <password>regadmin</password>
               <driverClassName>com.mysql.jdbc.Driver</driverClassName>
               <maxActive>80</maxActive>
               <maxWait>60000</maxWait>
               <minIdle>5</minIdle>
               <testOnBorrow>true</testOnBorrow>
               <validationQuery>SELECT 1</validationQuery>
               <validationInterval>30000</validationInterval>
           </configuration>
       </definition>
</datasource>
```

The database configuration options

- **url** - The URL of the database.
- **username** - The name of the database user.
- **password** - The password of the database user.
- **driverClassName** - The class name of the database driver.
- **maxActive** - The maximum number of active connections that can be allocated from this pool at the same time or negative for no limit.
- **maxWait** - The maximum number of milliseconds that the pool will wait (when there are no available connections) for a connection to be returned before throwing an exception or <= 0 to wait indefinitely.
- **minIdle** - The minimum number of active connections that can remain idle in the pool without extra ones being created or 0 to create none.

Setup Drivers

Download the MySQL Java connector JAR from http://dev.mysql.com/downloads/connector/j/5.1.html and place it in the `$CARBON_HOME/repository/components/lib` directory.

> **Tip**
>
> Here, `$CARBON_HOME` refers to the directory where you are running the product instance.

Create Database

Automatic Database Creation

1. When you start the server for the first time, use the `-Dsetup` option. It will create all the tables in the given MySQL database.

- For Linux:

```
wso2server.sh -Dsetup
```

- For Windows:

```
wso2server.bat -Dsetup
```

2. The product is configured to run with MySQL database.

Manual Database Creation

1. Run the MySQL scripts for both registry and user manager (embedded) databases, provided with the product, using the below commands (outside the MySQL prompt).

```
mysql -u regadmin -p -Dregdb < 'CARBON_HOME/dbscripts/mysql.sql';
```

> **Note**
>
> You will be prompted to enter the password for each command.

2. Start the WSO2 Carbon instance.

- For Linux:

```
wso2server.sh
```

- For Windows:

```
wso2server.bat
```

Setting up with MySQL Cluster

Find instruction on setting up any WSO2 product with MySQL cluster, refer to the following article published on WSO2 library:

- http://wso2.org/library/articles/2012/06/deploying-wso2-platform-mysql-cluster

Setting up with OpenEdge

Follow the instructions below to set up the OpenEdge.

Setup Database And The Database User | Setup Configuration Files | Setup Drivers | Create Database

---

Setup Database And The Database User

1. Download and install OpenEdge on you computer if it is not already done.

2. Go to the `<OE-installation-directory>/bin` folder and use the `proenv` script to setup the environment variables. After doing that, add the `<OE-insallation-directory>/java/prosp.jar` to the `CLASSPATH` envir onment variable.

3. Create an empty database using the `prodb` script. This script creates a database by copying an existing database provided with the installation.

```
prodb CARBON_DB <OE-installation-directory>/empty8
```

4. Start the database using the `proserve` script. Provide the database name and a port as arguments to this script using the `-db` and `-S` parameters.

```
proserve -db CARBON_DB -S 6767
```

5. Use the `sqlexp` script to start the default SQL explorer that comes with the OpenEdge installation. Connect to the database that was created previously by using the `-db` and `-S` parameters.

```
sqlexp -db CARBON_DB -S 6767
```

6. Now use the following commands to create a user and grant the permissions to the database.

```
CREATE USER 'wso2carbon','wso2carbon';
    GRANT dba,resource TO 'wso2carbon';
    COMMIT;
```

7. Now log out from the SQL explorer by typing "exit."

Setup Configuration Files

1. Edit the default database configuration defined in the `master-datasources.xml` file located at `PRODUCT_HOME/repository/conf/datasources` directory as below. Both the database configurations in `registry.xml` and `user-mgt.xml` refer this data source.

```
<datasource>
    <name>WSO2_CARBON_DB</name>
    <description>The datasource used for registry and user manager</description>
    <jndiConfig>
        <name>jdbc/WSO2CarbonDB</name>
    </jndiConfig>
    <definition type="RDBMS">
        <configuration>
            <url>jdbc:datadirect:openedge://localhost:6767;databaseName=CARBON_DB</url>
            <username>regadmin</username>
            <password>regadmin</password>
            <driverClassName>com.ddtek.jdbc.openedge.OpenEdgeDriver</driverClassName>
            <maxActive>80</maxActive>
            <maxWait>60000</maxWait>
            <minIdle>5</minIdle>
            <testOnBorrow>true</testOnBorrow>
            <validationQuery>SELECT 1</validationQuery>
            <validationInterval>30000</validationInterval>
        </configuration>
    </definition>
</datasource>
```

Note

You have to replace these settings with your custom values.

```
<datasource>
     <name>WSO2_CARBON_DB</name>
     <description>The datasource used for registry and user manager</description>
     <jndiConfig>
           <name>jdbc/WSO2CarbonDB</name>
     </jndiConfig>
     <definition type="RDBMS">
           <configuration>

<url>jdbc:datadirect:openedge://localhost:6767;databaseName=CARBON_DB</url>
                <username>regadmin</username>
                <password>regadmin</password>

<driverClassName>com.ddtek.jdbc.openedge.OpenEdgeDriver</driverClassName>
                <maxActive>80</maxActive>
                <maxWait>60000</maxWait>
                <minIdle>5</minIdle>
                <testOnBorrow>true</testOnBorrow>
                <validationQuery>SELECT 1</validationQuery>
                <validationInterval>30000</validationInterval>
           </configuration>
     </definition>
</datasource>
```

The database configuration options.

- **url** - The URL of the database.
- **username** - The name of the database user.
- **password** - The password of the database user.
- **driverName** - The class name of the database driver.
- **maxActive** - The maximum number of active connections that can be allocated from this pool at the same time or negative for no limit.
- **maxWait** - The maximum number of milliseconds that the pool will wait (when there are no available connections) for a connection to be returned before throwing an exception or <= 0 to wait indefinitely.
- **minIdle** - The minimum number of active connections that can remain idle in the pool, without extra ones being created, or 0 to create none.

> **Note**
>
> Please note that we do not support running User Manager on OpenEdge in this release.

Setup Drivers

Copy the `<OE-insallation-directory>/java/openedge.jar` to the `$CARBON_HOME/repository/components/lib` directory. Here `PRODUCT_HOME` refers to the directory where you run the product instance.

Create Database

Automatic Database Creation

1. Next at the first time you start the server, run with the -Dsetup option. It will create all the tables in a given OpenEdge database.

- For Linux:

```
wso2server.sh -Dsetup
```

- For Windows:

```
wso2server.bat -Dsetup
```

2. The product is configured to run with the OpenEdge database.

Manual Database Creation

1. For creating the tables manually, the OpenEdge script provided with the product has to be modified.

Make a backup of the `<PRODUCT_HOME>/dbscripts/openedge.sql` under the name of `openedge_manual.sql`.

2. Replace all the "/" symbols in the `openedge_manual.sql` script with the ";" symbol.

3. At the end of the `openedge_manual.sql` script, add the following line and save the script.

```
COMMIT;
```

4. Run the modified script using the SQL explorer.

```
sqlexp -db CARBON_DB -S 6767 -user wso2carbon -password wso2carbon <
PRODUCT_HOME/dbscripts/openedge_manual.sql
```

5. Start the server.

- For Linux:

```
wso2server.sh
```

- For Windows:

```
wso2server.bat
```

Setting up with Oracle

Follow the instructions below to set up the Oracle database.

> Setting up with Oracle RAC.For Oracle RAC, refer to

Setup Database and User | Setup Configuration File | Copy JDBC Driver | Create Database Tables

---

Setup Database and User

1. Create a new database. This can be done by either using the Oracle database configuration assistant (dbca) or

manually. Do necessary changes in the Oracle `tnsnames.ora` and `listner.ora` files in order to define databases addresses for establishing connections to the newly created database. After configuring them, startup the Oracle instance.

```
$ sudo /etc/init.d/oracle-xe restart
```

```
user-1@wso2:/usr/lib/oracle/xe/app/oracle/product/10.2.0/server/config/scripts$ sudo /etc/init.d/oracle-xe start
user-1@wso2:/usr/lib/oracle/xe/app/oracle/product/10.2.0/server/config/scripts$ sudo /etc/init.d/oracle-xe restart
Shutting down Oracle Database 10g Express Edition Instance.
Stopping Oracle Net Listener.

Starting Oracle Net Listener.
Starting Oracle Database 10g Express Edition Instance.

user-1@wso2:/usr/lib/oracle/xe/app/oracle/product/10.2.0/server/config/scripts$ ./sqlplus.sh sys/18091980

SQL*Plus: Release 10.2.0.1.0 - Production on Mon Oct 10 12:04:47 2011

Copyright (c) 1982, 2005, Oracle.  All rights reserved.

SQL> startup
ORA-01031: insufficient privileges
SQL>
SQL>
SQL>
SQL>
SQL>
SQL> select * from dual;
SP2-0640: Not connected
SQL>
SQL>
SQL>
SQL>
SQL> connect
Enter user-name: sysadm
Enter password:
Connected.
SQL>
SQL> select * from dual;

D
-
X

SQL> clear
SQL> Create user USER_NAME identified by PASSWORD account unlock;
```

2. Connect to Oracle using SQL*Plus as sysdba.

```
$ ./<ORACLE_HOME>/config/scripts/sqlplus.sh sysadm/password as sysdba
```

```
user-1@wso2:/usr/lib/oracle/xe/app/oracle/product/10.2.0/server/config/scripts$ ./sqlplus.sh sysadm/18091980 as sysdba
```

2.1. Connect to instance with username and password.

```
$ connect
```

3. As SYS DBA, create a database user and grant privileges to the user as shown below:

```
Create user USER_NAME identified by PASSWORD account unlock;
grant connect to USER_NAME;
grant create session, dba to USER_NAME;
commit;
```

For example,

```
SQL> Create user dbgreg identified by dbgreg account unlock;
User created.
SQL> grant connect to dbgreg;
Grant succeeded.
SQL> grant create session, dba to dbgreg;
Grant succeeded.
SQL> commit;
Commit complete.
SQL>
```

4. Exit from the SQL*Plus session by typing the "quit" command.

```
SQL> quit
```

```
SQL> connect
Enter user-name: sysadm
Enter password:
Connected.
SQL> quit
Disconnected from Oracle Database 10g Express Edition Release 10.2.0.1.0 - Production
user-1@wso2:/usr/lib/oracle/xe/app/oracle/product/10.2.0/server/config/scripts$
```

Setup Configuration File

1. Edit the default database configuration defined in the `master-datasources.xml` file located at `<APIM_HOME>/repository/conf/datasources` directory as follows. Both the database configurations in `registry.xml` and `user-mgt.xml` refer this data source.

```
<datasources-configuration xmlns:svns="http://org.wso2.securevault/configuration">

    <providers>
        <provider>org.wso2.carbon.ndatasource.rdbms.RDBMSDataSourceReader</provider>
    </providers>

    <datasources>
       <datasource>
           <name>WSO2_CARBON_DB</name>
           <description>The datasource used for registry and user manager</description>
           <jndiConfig>
               <name>jdbc/WSO2CarbonDB</name>
           </jndiConfig>
           <definition type="RDBMS">
               <configuration>
                   <url>jdbc:oracle:thin:@localhost:1521/XE</url>
                   <userName>regadmin</userName>
                   <password>regadmin</password>
                   <driverClassName>oracle.jdbc.driver.OracleDriver</driverClassName>
                   <maxActive>80</maxActive>
                   <maxWait>60000</maxWait>
                   <minIdle>5</minIdle>
                   <testOnBorrow>true</testOnBorrow>
                   <validationQuery>SELECT 1</validationQuery>
                   <validationInterval>30000</validationInterval>
               </configuration>
           </definition>
        </datasource>
```

Replace these settings with your own custom values:

```
<datasource>
     <name>WSO2_CARBON_DB</name>
     <description>The datasource used for registry and user manager</description>
     <jndiConfig>
           <name>jdbc/WSO2CarbonDB</name>
     </jndiConfig>
     <definition type="RDBMS">
           <configuration>
               <url>jdbc:oracle:thin:@SERVER_NAME:PORT/DB_NAME</url>
               <username>regadmin</userName>
               <password>regadmin</password>
               <driverClassName>oracle.jdbc.driver.OracleDriver</driverClassName>
               <maxActive>80</maxActive>
               <maxWait>60000</maxWait>
               <minIdle>5</minIdle>
               <testOnBorrow>true</testOnBorrow>
               <validationQuery>SELECT 1 FROM DUAL</validationQuery>
               <validationInterval>30000</validationInterval>
           </configuration>
     </definition>
</datasource>
```

The database configuration options

- **url** - The URL of the database.
- **username** - The name of the database user.

- **password** - The password of the database user.
- **driverClassName** - The class name of the database driver.
- **maxActive** - The maximum number of active connections that can be allocated from this pool at the same time or negative for no limit.
- **maxWait** - The maximum number of milliseconds that the pool will wait (when there are no available connections) for a connection to be returned before throwing an exception or <= 0 to wait indefinitely.
- **minIdle** - The minimum number of active connections that can remain idle in the pool without extra ones being created or 0 to create none.

> **Tip**
>
> Default port for Oracle is 1521.

Copy JDBC Driver

Copy the Oracle JDBC libraries to `<APIM_HOME>/repository/components/lib`. For example, `<ORACLE_HOME>/jdbc/lib/ojdbc14.jar`.

> Remove old database driver from <APIM_HOME>/repository/components/dropins , when you upgrade the database driver.

> When using the `ojdbc6.jar` with WSO2 servers there is a possibility of throwing an `timezone region not found` error. To overcome this issue it is necessary to set the java property as **`export JAVA_OPTS="-Duser.timezone='+05:30'".`** The value of this property should be the GMT difference of the country.
>
> if it is necessary to set this property permanently, then it should be defined inside the `wso2server.sh` as a new JAVA_OPT property.

Create Database Tables

Database tables can be created either manually by running scripts or automatically by using start-up parameters.

Using Scripts

Database tables can be created manually by logging in to the created database and running the following scripts in SQL*Plus:

```
SQL> @<APIM_HOME>/dbscripts/oracle.sql
```

Start the Carbon instance.

```
$ ./<APIM_HOME>/bin/wso2server.sh
```

Using start-up Parameters

- For Windows users:

```
<APIM_HOME>/bin/wso2server.bat -Dsetup
```

To create the database tables when starting the product for the first time.

- For Linux Users

```
$ ./<APIM_HOME>/bin/wso2server.sh -Dsetup
```

Setting up with Oracle RAC

Oracle Real Application Clusters (RAC) is an option for the Oracle Database for clustering and high availability in Oracle database environments. In Oracle RAC environment, some of the commands used in `oracle.sql` is Setting up with Oracleconsidered inefficient (refer to). Therefore, the product has a separate SQL script `oracle_rac.sql` for Oracle RAC. The Oracle RAC-friendly script is located in `dbscripts` folder together with other `.sql` scripts.

> **Tip**
>
> To test products on a Oracle RAC, please, rename `oracle_rac.sql` to `oracle.sql` before running `-Dsetup`.

Setup User | Setup Configuration File | Copy JDBC Driver | Create Database Tables

Setup User

1. Set environment variables `ORACLE_HOME, PATH, ORACLE_SID` with the corresponding values `/oracle/app/oracle/product/11.2.0/dbhome_1, $PATH:$ORACLE_HOME/bin, orcl1`.

```
[oracle@node1 ~]$ export ORACLE_HOME=/oracle/app/oracle/product/11.2.0/dbhome_1
[oracle@node1 ~]$ export PATH=$PATH:$ORACLE_HOME/bin
[oracle@node1 ~]$ export ORACLE_SID=orcl1
```

2. Connect to Oracle using SQL*Plus as sysdba.

```
[oracle@node1 ~]$ sqlplus SYSDBA/1 as sysdba

SQL*Plus: Release 11.2.0.1.0 Production on Fri Nov 18 18:10:42 2011

Copyright (c) 1982, 2009, Oracle.  All rights reserved.


Connected to:
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - 64bit Production
With the Partitioning, Real Application Clusters, Automatic Storage Management, OLAP,
Data Mining and Real Application Testing options

SQL> select 2+2 from dual;

       2+2
----------
         4

SQL> create user dbgreg identified by dbgreg account unlock;

User created.

SQL> grant connect to dbgreg;

Grant succeeded.

SQL> grant create session, dba to dbgreg;

Grant succeeded.

SQL> commit;

Commit complete.

SQL> quit
Disconnected from Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - 64bit Production
With the Partitioning, Real Application Clusters, Automatic Storage Management, OLAP,
Data Mining and Real Application Testing options
[oracle@node1 ~]$
```

3. Create a database user and grant privileges to the user as shown below:

```
Create user USER_NAME identified by PASSWORD account unlock;
grant connect to USER_NAME;
grant create session, dba to USER_NAME;
commit;
```

For example,

```
SQL> create user dbgreg identified by dbgreg account unlock;

User created.

SQL> grant connect to dbgreg;

Grant succeeded.

SQL> grant create session, dba to dbgreg;

Grant succeeded.

SQL> commit;

Commit complete.
```

4. Exit from the SQL*Plus session by typing the "quit" command.

```
SQL> quit
```

```
SQL> quit
Disconnected from Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - 64bit Production
With the Partitioning, Real Application Clusters, Automatic Storage Management, OLAP,
Data Mining and Real Application Testing options
[oracle@node1 ~]$
```

Setup Configuration File

1. Edit the default database configuration defined in the `master-datasources.xml` file located at `$CARBON_HOME/repository/conf/datasources` directory as follows. Both the database configurations in `registry.xml` and `user-mgt.xml` refer this data source.

```
    <datasource>
        <name>WSO2_CARBON_DB</name>
        <description>The datasource used for registry and user manager</description>
        <jndiConfig>
            <name>jdbc/WSO2CarbonDB</name>
        </jndiConfig>
        <definition type="RDBMS">
            <configuration>
                <url>jdbc:oracle:thin:@(DESCRIPTION=(LOAD_BALANCE=on)
                    (ADDRESS=(PROTOCOL=TCP)(HOST=racnode1) (PORT=1521))
                    (ADDRESS=(PROTOCOL=TCP)(HOST=racnode2) (PORT=1521))
                    (CONNECT_DATA=(SERVICE_NAME=rac)))</url>
                <username>regadmin</userName>
                <password>regadmin</password>
                <driverClassName>oracle.jdbc.driver.OracleDriver</driverClassName>
                <maxActive>80</maxActive>
                <maxWait>60000</maxWait>
                <minIdle>5</minIdle>
                <testOnBorrow>true</testOnBorrow>
                <validationQuery>SELECT 1</validationQuery>
                <validationInterval>30000</validationInterval>
            </configuration>
        </definition>
    </datasource>
```

Replace these settings with your own custom values.

```
<datasource>
     <name>WSO2_CARBON_DB</name>
     <description>The datasource used for registry and user manager</description>
     <jndiConfig>
          <name>jdbc/WSO2CarbonDB</name>
     </jndiConfig>
     <definition type="RDBMS">
          <configuration>
            <url>jdbc:oracle:thin:@(DESCRIPTION=(LOAD_BALANCE=on)
             (ADDRESS=(PROTOCOL=TCP)(HOST=racnode1) (PORT=1521))
             (ADDRESS=(PROTOCOL=TCP)(HOST=racnode2) (PORT=1521))
             (CONNECT_DATA=(SERVICE_NAME=service_name)))</url>
            <username>regadmin</userName>
            <password>regadmin</password>
            <driverClassName>oracle.jdbc.driver.OracleDriver</driverClassName>
            <maxActive>80</maxActive>
            <maxWait>60000</maxWait>
            <minIdle>5</minIdle>
            <testOnBorrow>true</testOnBorrow>
            <validationQuery>SELECT 1 FROM DUAL</validationQuery>
            <validationInterval>30000</validationInterval>
          </configuration>
     </definition>
</datasource>
```

The database configuration options

- **url** - The URL of the database.
- **username** - The name of the database user.
- **password** - The password of the database user.

- **driverClassName** - The class name of the database driver.
- **maxActive** - The maximum number of active connections that can be allocated from this pool at the same time or negative for no limit.
- **maxWait** - The maximum number of milliseconds that the pool will wait (when there are no available connections) for a connection to be returned before throwing an exception or <= 0 to wait indefinitely.
- **minIdle** - The minimum number of active connections that can remain idle in the pool without extra ones being created or 0 to create none.

Copy JDBC Driver

Copy the Oracle JDBC libraries to `<PRODUCT_HOME>/repository/components/lib` . For example, - `$ORACLE_HOME/jdbc/lib/ojdbc14.jar` .

> **Note:** Remove old database driver from `<PRODUCT_HOME>/repository/components/dropins` ,when you upgrade the database driver.

Create Database Tables

Database tables can be created either manually by running scripts or automatically by using start-up parameters.
Using Scripts

Database tables can be created manually by logging in to the created database and running the following scripts in SQL*Plus:

```
SQL> @${PRODUCT_HOME}/dbscripts/oracle.sql
```

```
Connected to:
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - 64bit Production
With the Partitioning, Real Application Clusters, Automatic Storage Management, OLAP,
Data Mining and Real Application Testing options

SQL> @/opt/wso2greg-4.1.0/dbscripts/oracle_rac.sql

Table created.


Table created.


Sequence created.


Trigger created.


Table created.


Index created.


Index created.


Sequence created.


Trigger created.


Table created.


Sequence created.
```

Start the product.

```
$ ./${PRODUCT_HOME}/bin/wso2server.sh
```

Using Start-up Parameters

- For Windows users:

```
$PRODUCT_HOME/bin/wso2server.bat -Dsetup
```

To create the database tables when starting the product for the first time.

- For Linux Users

```
$ ./${PRODUCT_HOME}/bin/wso2server.sh -Dsetup
```

Setting up with PostgreSQL

Follow the below instructions to set up PostgreSQL.

Setup Database And The Login Role | Setup Configuration Files | Setup Drivers | Create Database

Setup Database And The Login Role

1. Install PostgreSQL on your computer.

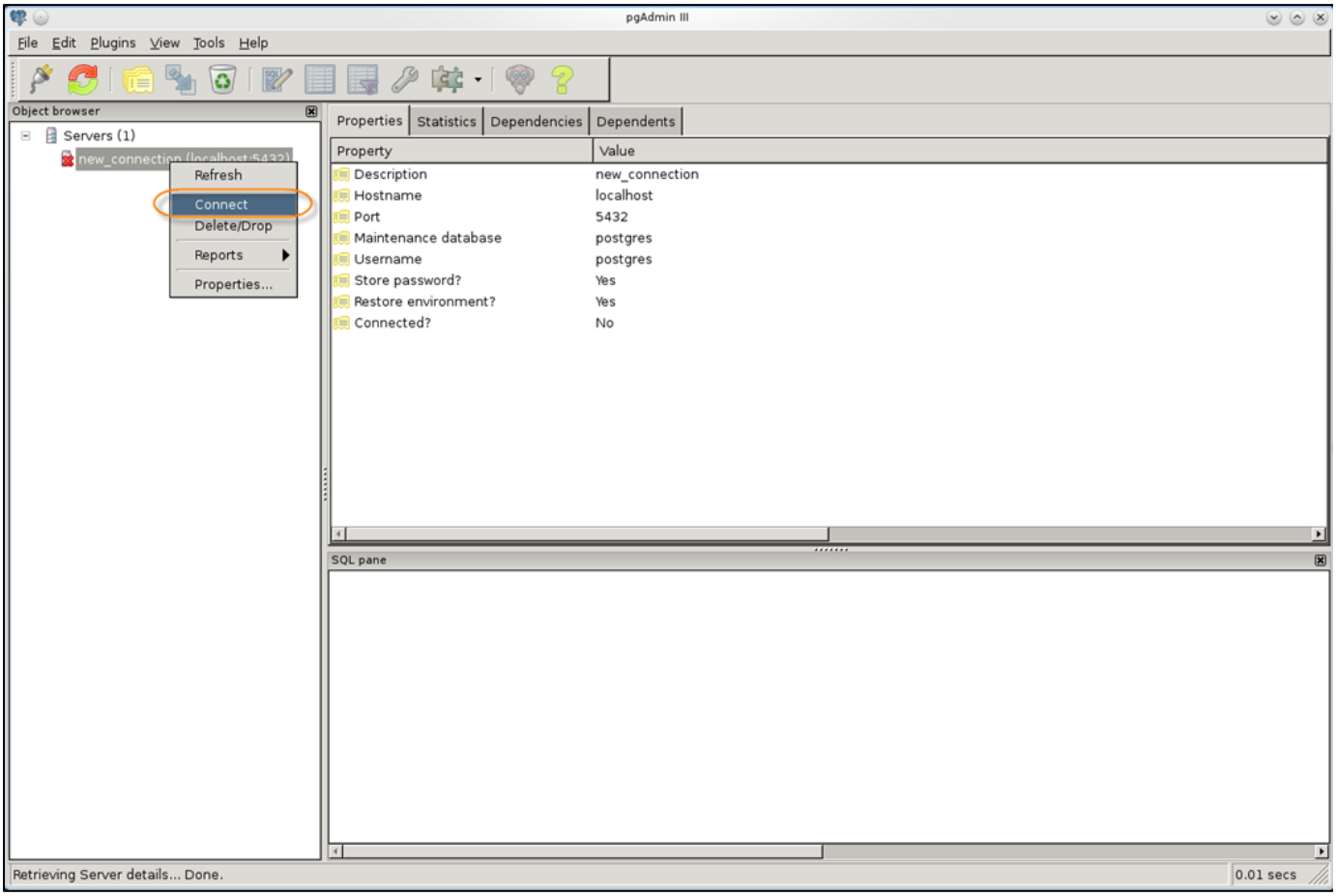
```
client@wso2:~/dtb$ sudo apt-get install postgresql
```

2. Start the PostgreSQL service.

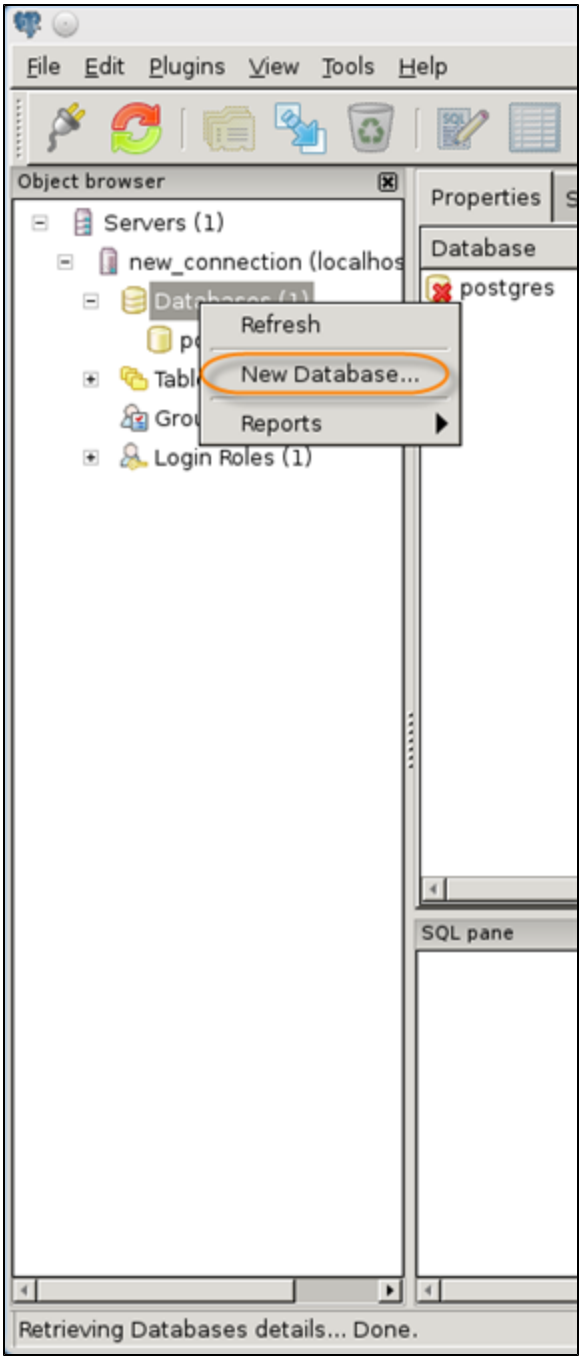
```
client@wso2:~$ sudo /etc/init.d/postgresql start
Starting PostgreSQL 8.4 database server: main.
client@wso2:~$
```

3. You can create a database and the login role from a GUI using the PGAdminIII tool: http://www.pgadmin.org/download.

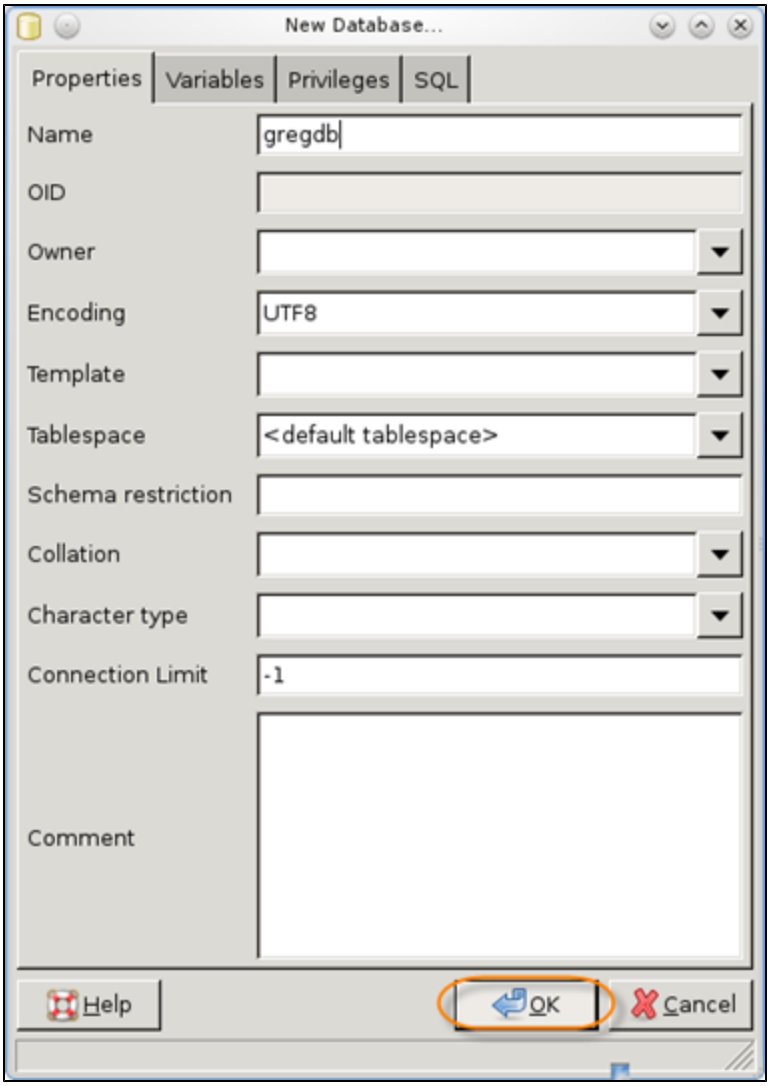
4. To connect PGAdminIII to a PostgreSQL database server, locate the server from the object browser, right-click the client, and click "Connect." This will show you the databases, tablespaces, and login roles. For example,



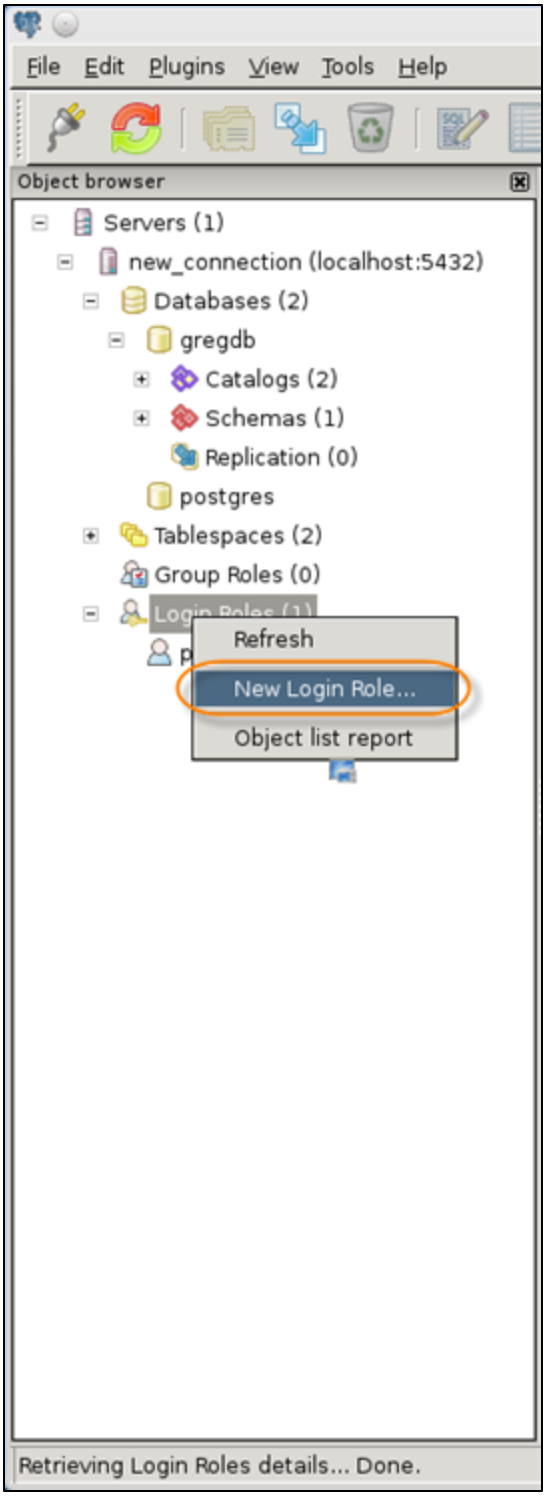
5. To create a database, click the "Databases" entry in the tree (inside the object browser), and click "New Database."



6. From the "New Database" dialog box, give a name to the database (for example, "gregdb") and click the "OK" button.



7. To create a login role, click the "Login Roles" entry in the tree (inside the object browser), and click "New Login Role." Supply the role name and a password.

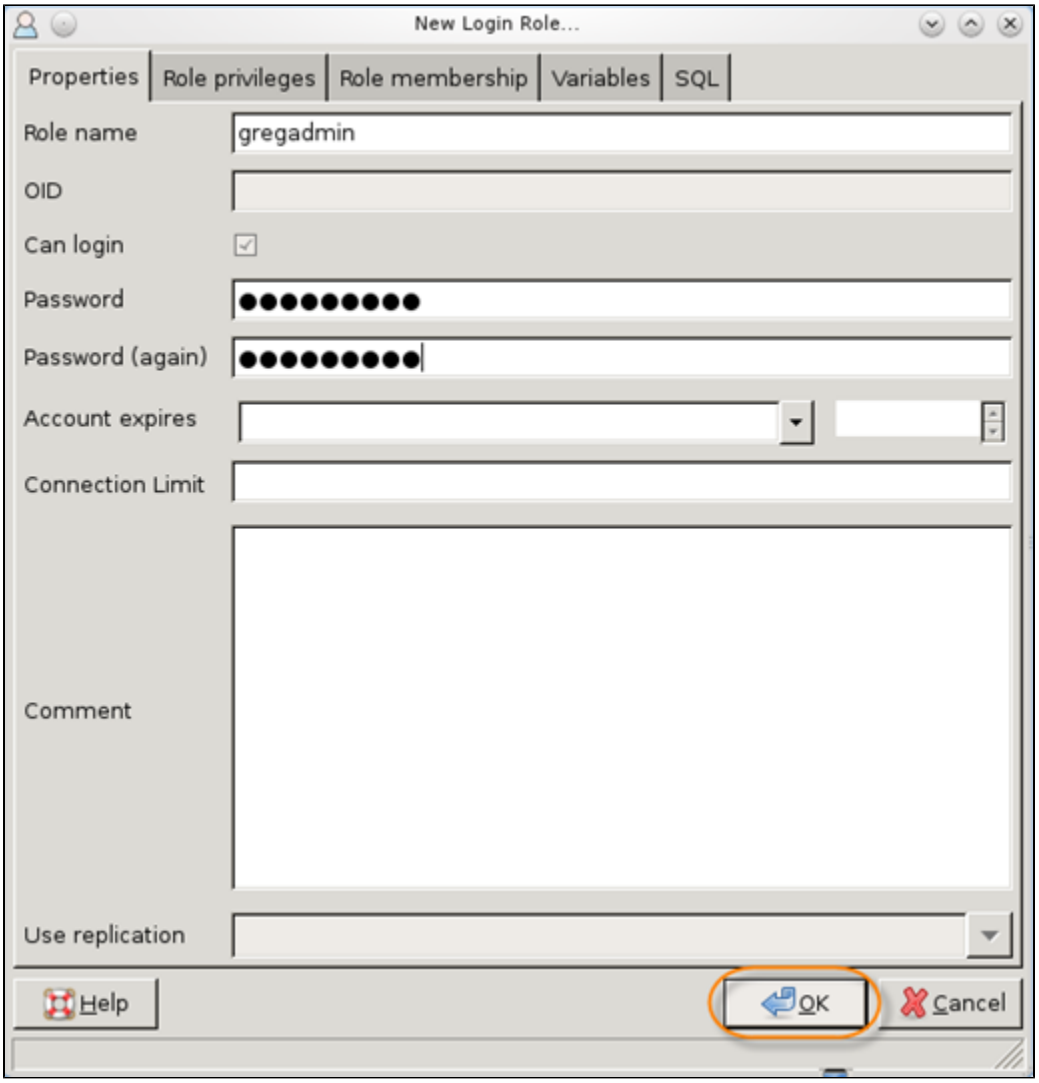


These values will be used in the product configurations as described in the following sections.

> **Tip**
>
> In the sample configuration, we are using "gregadmin" as the role name and "greadmin" as the password.

You can provide other policies, such as the expiration time for the login and the connection limit, which are optional.

Click the "OK" button to finish creating the login role. For example,



Setup Configuration Files

1. Edit the default database configuration defined in the `master-datasources.xml` file located at `$CARBON_HOME/repository/conf/datasources` directory as follows. Both the database configurations in `registry.xml` and `user-mgt.xml` refer this data source.

```
<datasources>

   <datasource>
        <name>WSO2_CARBON_DB</name>
        <description>The datasource used for registry and user manager</description>
        <jndiConfig>
            <name>jdbc/WSO2CarbonDB</name>
        </jndiConfig>
        <definition type="RDBMS">
            <configuration>
                <url>jdbc:postgresql://localhost:5432/gregdb</url>
                <username>regadmin</userName>
                <password>regadmin</password>
                <driverClassName>org.postgresql.Driver</driverClassName>
                <maxActive>80</maxActive>
                <maxWait>60000</maxWait>
                <minIdle>5</minIdle>
                <testOnBorrow>true</testOnBorrow>
                <validationQuery>SELECT 1</validationQuery>
                <validationInterval>30000</validationInterval>
            </configuration>
        </definition>
    </datasource>
```

Note

Replace these settings with your own custom values:

```
<datasource>
      <name>WSO2_CARBON_DB</name>
      <description>The datasource used for registry and user manager</description>
      <jndiConfig>
            <name>jdbc/WSO2CarbonDB</name>
      </jndiConfig>
      <definition type="RDBMS">
            <configuration>
                <url>jdbc:postgresql://localhost:5432/gregdb</url>
                <username>regadmin</userName>
                <password>regadmin</password>
                <driverClassName>org.postgresql.Driver</driverClassName>
                <maxActive>80</maxActive>
                <maxWait>60000</maxWait>
                <minIdle>5</minIdle>
                <testOnBorrow>true</testOnBorrow>
                <validationQuery>SELECT 1</validationQuery>
                <validationInterval>30000</validationInterval>
           </configuration>
       </definition>
</datasource>
```

The database configuration options

- **url** - The URL of the database.
- **username** - The name of the database user.

- **password** - The password of the database user.
- **driverClassName** - The class name of the database driver.
- **maxActive** - The maximum number of active connections that can be allocated from this pool at the same time or negative for no limit.
- **maxWait** - The maximum number of milliseconds that the pool will wait (when there are no available connections) for a connection to be returned before throwing an exception or <= 0 to wait indefinitely.
- **minIdle** - The minimum number of active connections that can remain idle in the pool, without extra ones being created, or 0 to create none.

Setup Drivers

1. Download the PostgreSQL JDBC4 driver from http://jdbc.postgresql.org/download.html.

2. Place the driver in the `PRODUCT_HOME/repository/components/lib` directory. Here, `$CARBON_HOME` refers to the directory where you are running the product instance.

Create Database

1. The first time you start the Carbon server, run it with the `-Dsetup` option. It will create all the tables in the given PostgreSQL database.

- For Linux:

```
wso2server.sh -Dsetup
```

- For Windows:

```
wso2server.bat -Dsetup
```

2. The product is now configured to run with PostgreSQL database.

Managing Datasources

A **datasource** provides information that a server can use to connect to a database. Datasource management is provided by the following feature in the WSO2 feature repository:

Name : WSO2 Carbon - Datasource Management Feature
Identifier : org.wso2.carbon.datasource.feature.group

If datasource management capability is not included in your product by default, you can add it by installing the above feature.

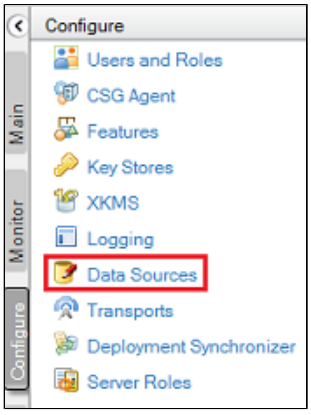
You can view, edit, and delete the datasources in your product instance by clicking **Data Sources** on the Configure tab of the product's management console. Note that you cannot edit or delete the default `WSO2_CARBON_DB` datasource.

The following topics describe how to manage datasources:

- Adding Datasources
- Configuring an RDBMS Datasource
- Configuring a Custom Datasource

Adding Datasources

If the Datasource Management feature is installed in your WSO2 product instance, you can add datasources that allow the server to connect to databases and other external data stores.
To add a datasource:

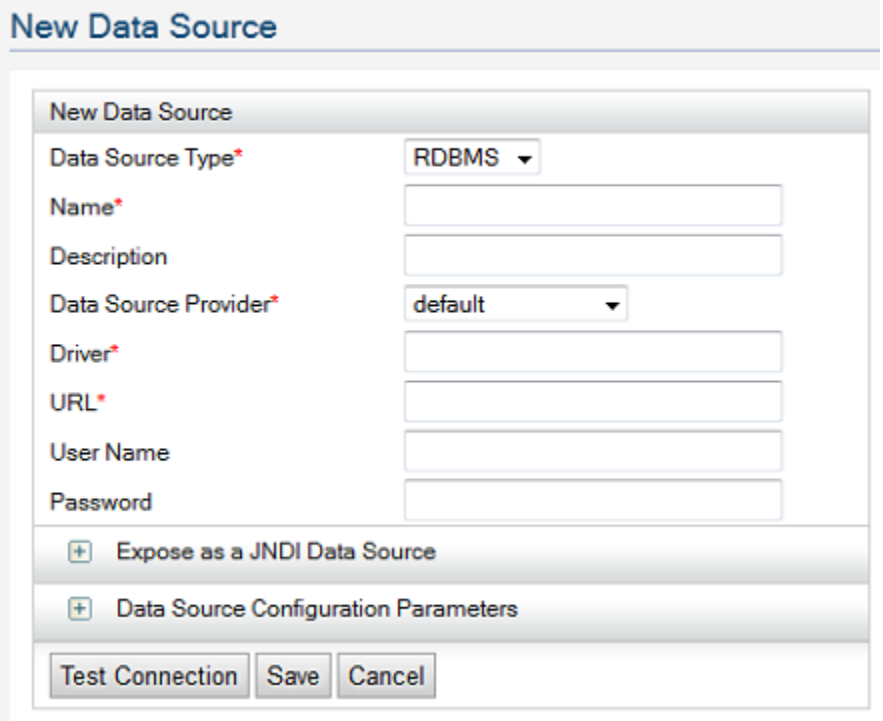


1. In the management console, click the **Configure** tab, and then click **Data Sources**.
2. Click **Add Data Source**.
3. Specify the required options for connecting to the database. The available options are based on the type of datasource you are creating:
   - Configuring an RDBMS Datasource
   - Configuring a Custom Datasource

After adding datasources, they appear on the Data Sources page. You can edit and delete them as needed by clicking their **Edit** or **Delete** links.

Configuring an RDBMS Datasource

When adding a datasource , if you select RDBMS as the datasource type, the following screen appears:



This is the default RDBMS datasource configuration provided by WSO2. You can also write your own RDBMS configuration by selecting the custom datasource option. Enter values for the following fields when using the default RDBMS datasource configuration:

- **Data Source Type**: RDBMS
- **Name**: Name of the datasource (must be a unique value)
- **Data Source Provider**: Specify the datasource provider.
- **Driver**: The class name of the JDBC driver to use. Be sure to copy the JDBC driver relevant to the database engine to the `<PRODUCT_HOME>/repository/components/dropins` and `<PRODUCT_HOME>/reposit`

`ory/components/lib` directories. For example, if you are using MySQL, you would specify `com.mysql.jdbc.Driver` as the driver and would copy `mysql-connector-java-5.XX-bin.jar` to these directories. If you do not copy the driver to these directories when you create the datasource, you will get an exception similar to "Cannot load JDBC driver class com.mysql.jdbc.Driver".

- **URL**: The connection URL to pass to the JDBC driver to establish the connection
- **User Name**: The connection user name to pass to the JDBC driver to establish the connection
- **Password**: The connection password to pass to the JDBC driver to establish the connection
- **Expose as a JNDI Data Souce**: Allows you to specify the JNDI data source as described below
- **Data Source Configuration Parameters**: Allows you to specify the datasource connection pool parameters when creating an RDBMS datasource

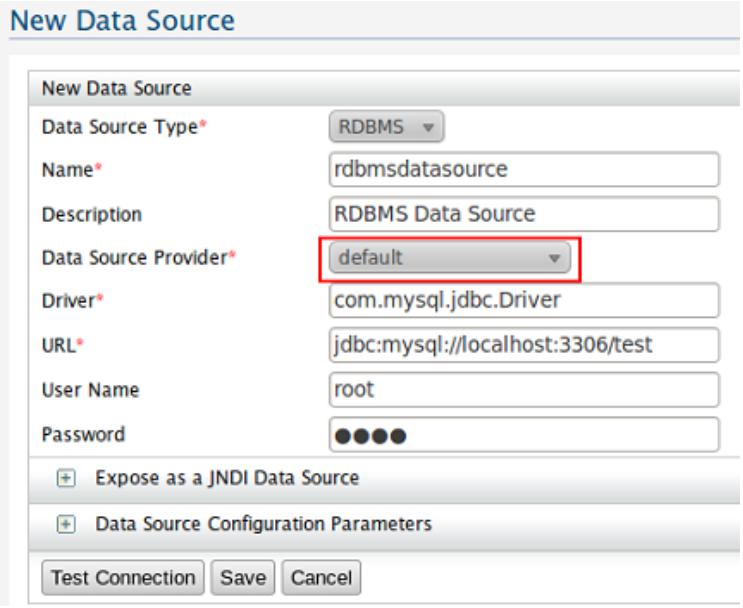
After creating datasources, they appear on the Data Sources page. You can edit and delete them as needed by clicking their **Edit** or **Delete** links.

Configuring the Datasource Provider

A datasource provider connects to a source of data such as a database, accesses its data, and returns the results of the access queries. When creating an RDBMS datasource, you can use the default provider or link to an external provider.

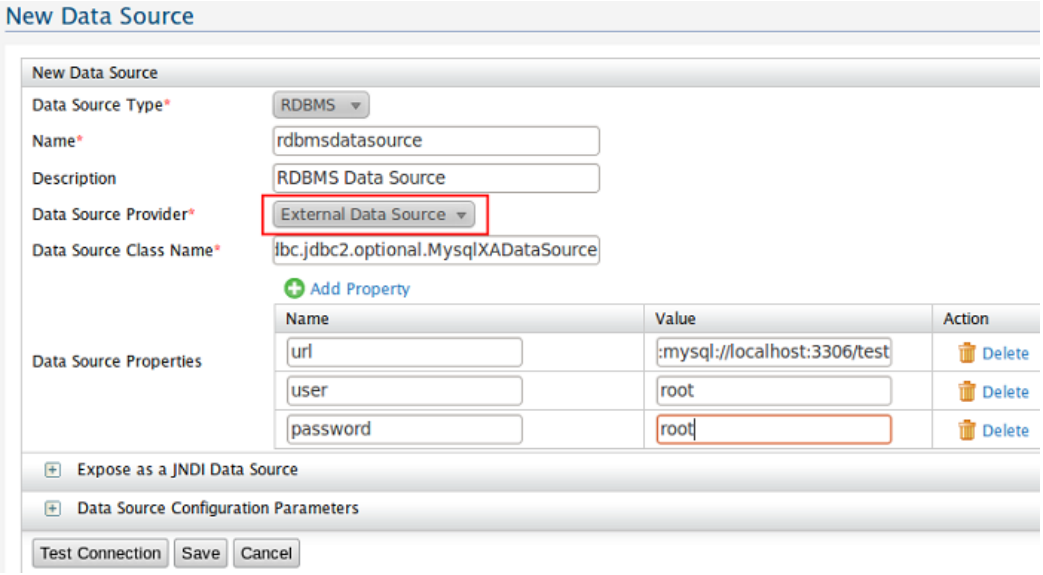
Default datasource provider

To use the default datasource provider, select **default**, and then enter the connection properties Driver, URL, User Name, and Password as follows:



External datasource provider

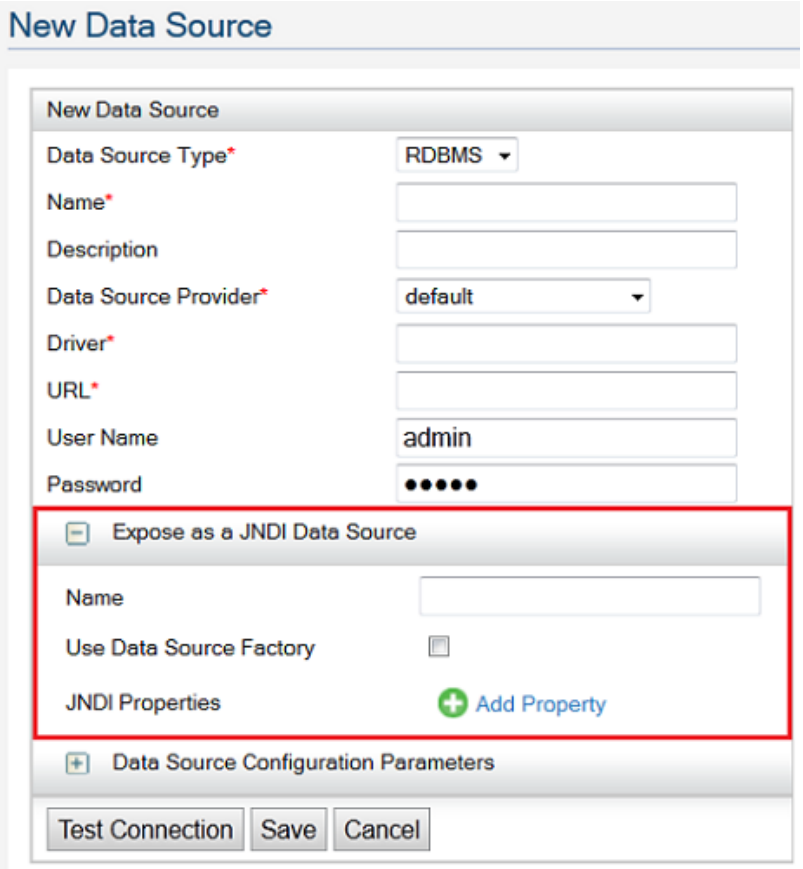
If you need to add a datasource supported by an external provider class such as `com.mysql.jdbc.jdbc2.optional.MysqlXADataSource`, select **External Data Source**, click **Add Property**, and then enter the name and value of each connection property you need to configure. Following is an example datasource for an external datasource provider.



Configuring a JNDI Datasource

Java Naming and Directory Interface (JNDI) is a Java application programming interface (API) that provides naming and directory functionality for Java software clients to discover and look up data and objects via a name. It helps decouple object creation from the object look-up. When you have registered a datasource with JNDI, others can discover it through a JNDI look-up and use it.

When adding a datasource, to expose an RDBMS datasource as a JNDI datasource, click **Expose as a JNDI Data Source** to display the JNDI fields:



Following are descriptions of the JNDI fields:

- **Name**: Name of the JNDI datasource that will be visible to others in object look-up

- **Use Data Source Factory**: To make the datasource accessible from an external environment, you must use a data source factory. When this option is selected, a reference object will be created with the defined datasource properties. The data source factory will create the datasource instance based on the values of the reference object when accessing the datasource from an external environment. In the datasource configuration, this is set as follows: `<jndiConfig useDataSourceFactory="true">`
- **JNDI Properties**: Properties related to the JNDI datasource (such as password). When you select this option, set the following properties:
  - `java.naming.factory.initial`: Selects the registry service provider as the initial context
  - `java.naming.provider.url`: Specifies the location of the registry when the registry is being used as the initial context

Configuring the Datasource Connection Pool Parameters

When the server processes a database operation, it spawns a database connection from an associated datasource. After using this connection, the server returns it to the pool of connections. This is called **datasource connection pooling** and is a recommended way to gain more performance/throughput in the system. In datasource connection pooling, the physical connection is not dropped with the database server unless it becomes stale or the datasource connection is closed.

RDBMS datasources in WSO2 products use Tomcat JDBC connection pool (`org.apache.tomcat.jdbc.pool`). It is common to all components that access databases for data persistence, such as the registry, user management (if configured against a JDBC userstore), etc.

You can configure the datasource connection pool parameters, such as how long a connection is persisted in the pool, using the datasource configuration parameters section that appears in the management console when creating a datasource. Click and expand the option as shown below:

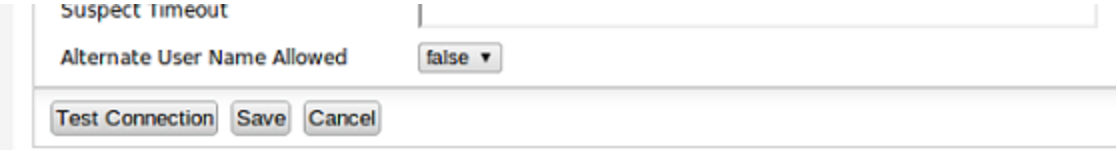
Add New Data Source

New Data Source

DataSource Id*	oracle-ds
Data Source Type*	RDBMS ▾ Non-XA-DataSource ▾
Database Engine*	Oracle ▾
Driver Class*	oracle.jdbc.driver.OracleDriver
URL*	jdbc:oracle:[drivertype]:[username/password]@[host]:[port]/[database]
User Name	
Password	

Data source configuration parameters

Transaction Isolation	TRANSACTION_UNKNOWN ▾
Initial Size	
Max. Active	
Max. Idle	
Min. Idle	
Max. Wait	
Validation Query	
Test On Return	false ▾
Test On Borrow	true ▾
Test While Idle	false ▾
Time Between Eviction Runs Mills	
Minimum Evictable Idle Time	
Remove Abandoned	false ▾
Remove Abandoned Timeout	
Log Abandoned	false ▾
Default Auto Commit	false ▾
Default Read Only	false ▾
Default Catalog	
Validator Class Name	
Connection Properties	
Init SQL	
JDBC Interceptors	
Validation Interval	
JMX Enabled	false ▾
Fair Queue	false ▾
Abandon When Percentage Full	
Max Age	
Use Equals	false ▾



Following are descriptions of the parameters you can configure. For more details on datasource configuration parameters, refer to http://tomcat.apache.org/tomcat-7.0-doc/jdbc-pool.html and the DBCP configuration guide at http://commons.apache.org/proper/commons-dbcp/configuration.html.

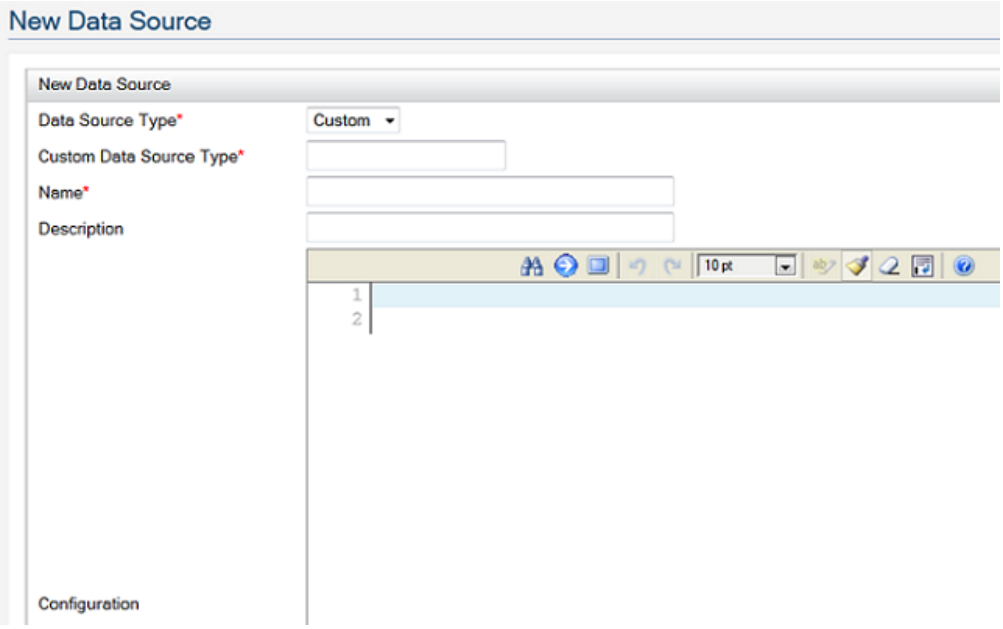
Parameter Name	Description
Transaction Isolation	The default TransactionIsolation state of connections created by this pool. • TRANSACTION_UNKNOWN • TRANSACTION_NONE • TRANSACTION_READ_COMMITTED • TRANSACTION_READ_UNCOMMITTED • TRANSACTION_REPEATABLE_READ • TRANSACTION_SERIALIZABLE
Initial Size	(int) The initial number of connections created when the pool is started. Default value is 0.
Max. Active	(int) The maximum number of active connections that can be allocated from this pool at the same time. The default value is 100.
Max. Idle	(int) The maximum number of connections that should be kept in the pool at all times. Default value is 8. Idle connections are checked periodically (if enabled) and connections that have been idle for longer than minEvictableIdleTimeMillis will be released. (also see testWhileIdle)
Min. Idle	(int) The minimum number of established connections that should be kept in the pool at all times. The connection pool can shrink below this number if validation queries fail. Default value is 0. (also see testWhileIdle)
Max. Wait	(int) Maximum number of milliseconds that the pool waits (when there are no available connections) for a connection to be returned before throwing an exception. Default value is 30000 (30 seconds).
Validation Query	(String) The SQL query used to validate connections from this pool before returning them to the caller. If specified, this query does not have to return any data, it just can't throw a SQLException. The default value is null. Example values are SELECT 1(mysql), select 1 from dual(oracle), SELECT 1(MS Sql Server).
Test On Return	(boolean) Used to indicate if objects will be validated before returned to the pool. NOTE - for a true value to have any effect, the validationQuery parameter must be set to a non-null string. The default value is false.

Test On Borrow	(boolean) Used to indicate if objects will be validated before borrowed from the pool. If the object fails to validate, it will be dropped from the pool, and we will attempt to borrow another. NOTE - for a true value to have any effect, the validationQuery parameter must be set to a non-null string. In order to have a more efficient validation, see validationInterval. Default value is false.
Test While Idle	(boolean) The indication of whether objects will be validated by the idle object evictor (if any). If an object fails to validate, it will be dropped from the pool. NOTE - for a true value to have any effect, the validationQuery parameter must be set to a non-null string. The default value is false and this property has to be set in order for the pool cleaner/test thread to run (also see timeBetweenEvictionRunsMillis).
Time Between Eviction Runs Mills	(int) The number of milliseconds to sleep between runs of the idle connection validation/cleaner thread. This value should not be set under 1 second. It dictates how often we check for idle, abandoned connections, and how often we validate idle connections. The default value is 5000 (5 seconds).
Minimum Evictable Idle Time	(int) The minimum amount of time an object may sit idle in the pool before it is eligible for eviction. The default value is 60000 (60 seconds).
Remove Abandoned	(boolean) Flag to remove abandoned connections if they exceed the removeAbandonedTimout. If set to true a connection is considered abandoned and eligible for removal if it has been in use longer than the removeAbandonedTimeout Setting this to true can recover db connections from applications that fail to close a connection. See also logAbandoned. The default value is false.
Remove Abandoned Timeout	(int) Timeout in seconds before an abandoned(in use) connection can be removed. The default value is 60 (60 seconds). The value should be set to the longest running query your applications might have.
Log Abandoned	(boolean) Flag to log stack traces for application code which abandoned a Connection. Logging of abandoned Connections adds overhead for every Connection borrow because a stack trace has to be generated. The default value is false.
Auto Commit	(boolean) The default auto-commit state of connections created by this pool. If not set, default is JDBC driver default (If not set then the setAutoCommit method will not be called.)
Default Read Only	(boolean) The default read-only state of connections created by this pool. If not set then the setReadOnly method will not be called. (Some drivers don't support read only mode, ex: Informix)
Default Catalog	(String) The default catalog of connections created by this pool.
Validator Class Name	(String) The name of a class which implements the org.apache.tomcat.jdbc.pool.Validator interface and provides a no-arg constructor (may be implicit). If specified, the class will be used to create a Validator instance which is then used instead of any validation query to validate connections. The default value is null. An example value is com.mycompany.project.SimpleValidator.
Connection Properties	(String) The connection properties that will be sent to our JDBC driver when establishing new connections. Format of the string must be [propertyName=property;]* NOTE - The "user" and "password" properties will be passed explicitly, so they do not need to be included here. The default value is null.
Init SQL	The ability to run a SQL statement exactly once, when the connection is created.
JDBC Interceptors	Flexible and pluggable interceptors to create any customizations around the pool, the query execution and the result set handling.

Validation Interval	(long) avoid excess validation, only run validation at most at this frequency - time in milliseconds. If a connection is due for validation, but has been validated previously within this interval, it will not be validated again. The default value is 30000 (30 seconds).
JMX Enabled	(boolean) Register the pool with JMX or not. The default value is true.
Fair Queue	(boolean) Set to true if you wish that calls to getConnection should be treated fairly in a true FIFO fashion. This uses the org.apache.tomcat.jdbc.pool.FairBlockingQueue implementation for the list of the idle connections. The default value is true. This flag is required when you want to use asynchronous connection retrieval. Setting this flag ensures that threads receive connections in the order they arrive. During performance tests, there is a very large difference in how locks and lock waiting is implemented. When fairQueue=true there is a decision making process based on what operating system the system is running. If the system is running on Linux (property os.name=Linux. To disable this Linux specific behavior and still use the fair queue, simply add the property org.apache.tomcat.jdbc.pool.FairBlockingQueue.ignoreOS=true to your system properties before the connection pool classes are loaded.
Abandon When Percentage Full	(int) Connections that have been abandoned (timed out) wont get closed and reported up unless the number of connections in use are above the percentage defined by abandonWhenPercentageFull. The value should be between 0-100. The default value is 0, which implies that connections are eligible for closure as soon as removeAbandonedTimeout has been reached.
Max Age	(long) Time in milliseconds to keep this connection. When a connection is returned to the pool, the pool will check to see if the now - time-when-connected > maxAge has been reached, and if so, it closes the connection rather than returning it to the pool. The default value is 0, which implies that connections will be left open and no age check will be done upon returning the connection to the pool.
Use Equals	(boolean) Set to true if you wish the ProxyConnection class to use String.equals and set to false when you wish to use == when comparing method names. This property does not apply to added interceptors as those are configured individually. The default value is true.
Suspect Timeout	(int) Timeout value in seconds. Default value is 0. Similar to to the removeAbandonedTimeout value but instead of treating the connection as abandoned, and potentially closing the connection, this simply logs the warning if logAbandoned is set to true. If this value is equal or less than 0, no suspect checking will be performed. Suspect checking only takes place if the timeout value is larger than 0 and the connection was not abandoned or if abandon check is disabled. If a connection is suspect a WARN message gets logged and a JMX notification gets sent once.
Alternate User Name Allowed	(boolean) By default, the jdbc-pool will ignore the DataSource.getConnection(username,password) call, and simply return a previously pooled connection under the globally configured properties username and password, for performance reasons. The pool can however be configured to allow use of different credentials each time a connection is requested. To enable the functionality described in the DataSource.getConnection(username,password) call, simply set the property alternateUsernameAllowed to true. Should you request a connection with the credentials user1/password1 and the connection was previously connected using different user2/password2, the connection will be closed, and reopened with the requested credentials. This way, the pool size is still managed on a global level, and not on a per schema level. The default value is false.

Configuring a Custom Datasource

When adding a datasource , if you select the Custom datasource type, the following screen will appear:



Following are descriptions of the custom datasource fields:

- **Data Source Type**: Custom
- **Custom Data Source Type**: Specify whether the data is in a table or accessed through a query as described below
- **Name**: Enter a unique name for this datasource
- **Description**: Description of the datasource
- **Configuration**: XML configuration of the datasource

Custom datasource type

When creating a custom datasource, you specify whether the datasource type is DS_CUSTOM_TABULAR (the data is stored in tables) or DS_CUSTOM_QUERY (non-tabular data accessed through a query). Following is more information about each type.

Custom tabular datasources

Tabular datasources are used for accessing tabular data, that is, the data is stored in rows in named tables that can be queried later. To implement tabular datasources, the interface `org.wso2.carbon.dataservices.core.custom.datasource.TabularDataBasedDS` is used. You can see a sample implementation of a tabular custom datasource at `InMemoryDataSource` .

A tabular datasource is typically associated with a SQL data services query. WSO2 products use an internal SQL parser to execute SQL against the custom datasource. You can see a sample data service descriptor at `InMemoryDSSample` . Carbon datasources also support tabular data with the datasource reader implementation `org.wso2.carbon.dataservices.core.custom.datasource.CustomTabularDataSourceReader` . If you have Data Services Server installed, you can see a sample Carbon datasource configuration file at `<DSS_HOME>\repository\conf\datasources\custom-datasources.xml`.

Custom query datasources

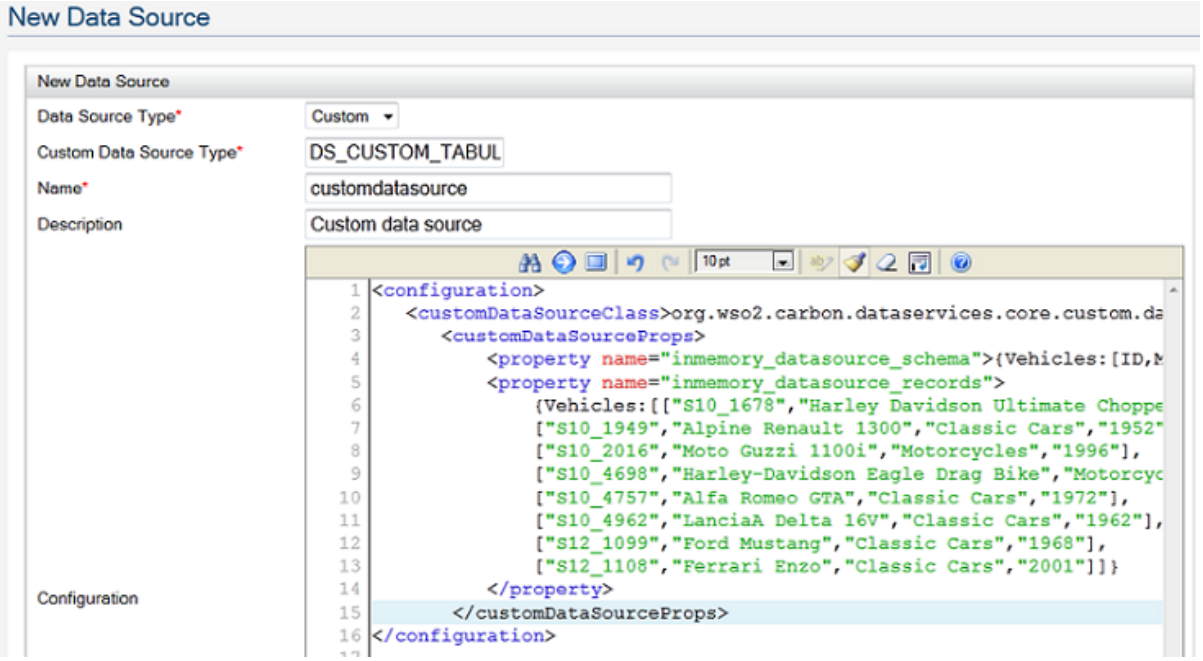
Custom query-based datasources are used for accessing non-tabular data through a query expression. To implement query-based datasources, the interface `org.wso2.carbon.dataservices.core.custom.datasource.CustomQueryBasedDS` is used. You can create any non-tabular datasource using the query-based approach. Even if the target datasource does not have a query expression format, you can create and use your own. For example, you can support any NoSQL type datasource using this type of a datasource.

You can see a sample implementation of a custom query-based datasource at `EchoDataSource` . You can see a

sample data service descriptor with custom query datasources in `InMemoryDSSample` . Carbon datasources also support query-based data with the datasource reader implementation `org.wso2.carbon.dataservices.core.custom.datasource.CustomQueryDataSourceReader` . If you have Data Services Server installed, you can see a sample Carbon datasource configuration file at `<DSS_HOME>\repository\conf\datasources\custom-datasources.xml`.

In the "init" methods of all custom datasources, user-supplied properties will be parsed to initialize the datasource accordingly. Also, a property named "__DATASOURCE_ID__", which contains a UUID to uniquely identify the current datasource, will be passed. This can be used by custom datasource authors to identify the datasources accordingly, such as datasource instances communicating within a server cluster for data synchronization.

Shown below is an example configuration of a custom datasource of type 'DS_CUSTOM_TABULAR'.



After creating datasources, they appear on the Data Sources page. You can edit and delete them as needed by clicking their **Edit** or **Delete** links.

Configuring Caching

When an API call hits the API Gateway, the Gateway carries out security checks to verify if the token is valid. During these verifications, the API Gateway extracts parameters such as access token, API and API version that are passed on to it. Since the entire load of traffic to APIs goes through the API Gateway, this verification process needs to be fast and efficient in order to prevent overhead and delays. The API Manager uses caching for this purpose, where the validation information is cached with the token, API name and version, and the cache is stored in either the API Gateway or the key manager server.

Caching at API Gateway

When caching is enabled at the Gateway and a request hits the Gateway, it first populates the cached entry for a given token. If a cache entry does not exist in cache, it calls the key manager server. This process is carried out using Web service calls. Once the key manager server returns the validation information, it gets stored in the Gateway. Because the API Gateway issues a Web service call to the key manager server only if it does not have a cache entry, this method reduces the number of Web service calls to the key manager server. Therefore, it is faster than the alternative method.

By default, the API Gateway cache is enabled by setting the `<EnableGatewayKeyCache>` element to true in `<API M_HOME>/repository/conf/api-manager.xml` file:

```
<EnableGatewayKeyCache>true</EnableGatewayKeyCache>
```

Clearing the API Gateway cache

To remove old tokens that might still remain active in the Gateway cache, you configure the `<RevokeAPIURL>` elem ent in `api-manager.xml` file by providing the URL of the Revoke API that is deployed in the API Gateway node. The revoke API invokes the cache clear handler, which extracts information form transport headers of the revoke request and clears all associated cache entries. If there's a cluster of API Gateways in your setup, provide the URL of the revoke API deployed in one node in the cluster. This way, all revoke requests route to the OAuth service through the Revoke API.

Given below is how to configure this in a distributed API Manager setup.

1. In the `api-manager.xml` file of the key manager node, point the revoke endpoint as follows:

```
<RevokeAPIURL>https://${carbon.local.ip}:${https.nio.port}/revoke</RevokeAPIURL>
```

2. In the API Gateway, point the Revoke API to the OAuth application deployed in the key manager node. For example,

```
<api name="_WSO2AMRevokeAPI_" context="/revoke">
        <resource methods="POST" url-mapping="/*" faultSequence="_token_fault_">
            <inSequence>
                <send>
                    <endpoint>
                        <address
uri="https://keymgt.wso2.com:9445/oauth2/revoke"/>
                    </endpoint>
                </send>
            </inSequence>
            <outSequence>
                <send/>
            </outSequence>
        </resource>
        <handlers>
            <handler
class="org.wso2.carbon.apimgt.gateway.handlers.ext.APIManagerCacheExtensionHandle
r"/>
        </handlers>
</api>
```

Caching at Key Manager server

In this method, the cache is maintained at the key manager server rather than the API Gateway. As a result, for each and every API call that hits the API Gateway, the Gateway issues a Web service call to the key manager server. If the cache entry is available in the key manager server, it is returned to the Gateway. Else, the database will be checked for the validity of the token.

This method has low performance compared to the earlier one, but the the advantage of this method over the other is that we do not have to store any security-related information at the Gateway side.

By default, caching is enabled at the Gateway side as it is the faster method. If you want to change this default configuration, disable caching at the Gateway side and enable it at the key manager server side by using the instructions below.

1. Disable caching at API Gateway by adding the following entry to **APIGateway** section of `<APIM_HOME>/repository/conf/api-manager.xml` file.

```
<EnableGatewayKeyCache>false</EnableGatewayKeyCache>
```

2. Enable key manager server-side caching by adding the following entry to **APIKeyManager** section of the api-manager.xml file.

```
<EnableKeyMgtValidationInfoCache>true</EnableKeyMgtValidationInfoCache>
```

3. The API Manager generates JWT tokens for each validation information object. Usually, JWT tokens also get cached with the validation information object, but you might want to generate JWT per each call. You can do this by enabling JWT caching at key manager server. Add the following entry to **APIKeyManager** section of the api-manager.xml file.

```
<EnableJWTCache>true</EnableJWTCache>
```

> Note that you must disable caching at the key manager server side in order to generate JWT per each call.

Also enable token generation by setting the following entry to `true` at the root level of the api-manager.xml file.

```
<APIConsumerAuthentication>
    <EnableTokenGeneration>true</EnableTokenGeneration>
    ...
</APIConsumerAuthentication>
```

Response caching

The API Manager uses WSO2 ESB's cache mediator to cache response messages per each API. You can configure response caching at the time an API is created. For information, see Response Caching.

Configuring Single Sign-on with SAML 2.0

Single sign-on (SSO) allows users, who are authenticated against one application, gain access to multiple other related applications as well without having to repeatedly authenticate themselves. It also allows the Web applications gain access to a set of back-end services with the logged-in user's access rights, and the back-end services can authorize the user based on different claims like user role.

WSO2 API Manager includes **Single Sign-On with SAML 2.0** feature, which is implemented according to the SAML 2.0 Web browser-based SSO support that is facilitated by WSO2 Identity Server (IS). This feature is available in any IS version from 4.1.0 onwards. We use **IS 5.0.0** in this guide. WSO2 Identity Server acts as an identity service provider of systems enabled with single sign-on, while the Web applications such as API Manager apps act as SSO service providers. Using this feature, you can configure SSO across the two API Manager Web applications, which are API Publisher and API Store as well as other Web applications in your organization. After configuring, you will be able to access API Store or API Publisher in a single authentication attempt.

> To learn more about Single Sign-On with WSO2 Identity Server, see the following article in WSO2 library: http://wso2.org/library/articles/2010/07/saml2-web-browser-based-sso-wso2-identity-server.

The topics below explain the configurations:

- Sharing the registry space
- Configuring WSO2 Identity Server as a SAML 2.0 SSO Identity Provider
- Configuring WSO2 API Manager Apps as SAML 2.0 SSO Service Providers

For example, let's take a common JDBC user store (MySQL) for both IS and API Manager.

1. Create a MySQL database (e.g., 410_um_db) and run the `<AM_HOME>/dbscripts/mysql.sql` script on it to create the required tables.
   If you are using a different database type, find the relevant script from the `<AM_HOME>/dbscripts` directory.
2. Open `<AM_HOME>/repository/conf/datasources/master-datasources.xml` file and add the datasource configuration for the database that you use for the shared user store and user management information. For example,

```
<datasource>
     <name>WSO2_UM_DB</name>
     <description>The datasource used for registry and user manager</description>
     <jndiConfig>
              <name>jdbc/WSO2UMDB</name>
     </jndiConfig>
     <definition type="RDBMS">
              <configuration>
                  <url>jdbc:mysql://localhost:3306/410_um_db</url>
                  <username>username</username>
                  <password>password</password>
                  <driverClassName>com.mysql.jdbc.Driver</driverClassName>
                  <maxActive>50</maxActive>
                  <maxWait>60000</maxWait>
                  <testOnBorrow>true</testOnBorrow>
                  <validationQuery>SELECT 1</validationQuery>
                  <validationInterval>30000</validationInterval>
              </configuration>
       </definition>
</datasource>
```

3. Add the same datasource configuration above to `<IS_HOME>/repository/conf/datasources/master-datasources.xml` file.
4. Copy the database driver JAR file to the `<IS_HOME>/repository/components/lib` and `<AM_HOME>/repository/components/lib` directories.
5. Open `<AM_HOME>/repository/conf/user-mgt.xml` file. The `dataSource` property points to the default H2 database. Change it to the jndiConfig name given above (i.e., `jdbc/WSO2UMDB`). This changes the datasource reference that is pointing to the default H2 database.

```
<Realm>
        <Configuration>
            ...
            <Property name="dataSource">jdbc/WSO2UMDB</Property>
        </Configuration>
        ...
</Realm>
```

6. Add the same configuration above to the `<IS_HOME>/repository/conf/user-mgt.xml` file.
7. The Identity Server has an embedded LDAP user store by default. As this is enabled by default, follow the instructions in Internal JDBC User Store Configuration to disable the default LDAP and enable the JDBC user store instead.

Sharing the registry space

Let's share a common registry space between the IS and APIM. This can be done by creating a registry database and mounting it on both the IS and APIM.

1. Create a MySQL database (e.g., registry) and run the `<IS_HOME>/dbscripts/mysql.sql` script on it to create the required tables.
   If you are using a different database type, find the relevant script from the `<IS_HOME>/dbscripts` directory.
2. Add the following datasource configuration to both the `<IS_HOME>/repository/conf/datasources/master-datasources.xml` and `<AM_HOME>/repository/conf/datasources/master-datasources.xml` files.

```
<datasource>
    <name>WSO2REG_DB</name>
    <description>The datasource used for registry</description>
    <jndiConfig>
        <name>jdbc/WSO2REG_DB</name>
    </jndiConfig>
    <definition type="RDBMS">
        <configuration>

<url>jdbc:mysql://localhost:3306/registry?autoReconnect=true&amp;relaxAutoCommit=
true&amp;</url>
            <username>apiuser</username>
            <password>apimanager</password>
            <driverClassName>com.mysql.jdbc.Driver</driverClassName>
            <maxActive>50</maxActive>
            <maxWait>60000</maxWait>
            <testOnBorrow>true</testOnBorrow>
            <validationQuery>SELECT 1</validationQuery>
            <validationInterval>30000</validationInterval>
        </configuration>
    </definition>
</datasource>
```

3. Create the registry mounts by inserting the following sections into the `<IS_HOME>/repository/conf/registry.xml` file.

> When doing this change, do not replace the existing `<dbConfig>` for "`wso2registry`". Simply add the following configuration to the existing configurations.

```
<dbConfig name="govregistry">
        <dataSource>jdbc/WSO2REG_DB</dataSource>
</dbConfig>

<remoteInstance url="https://localhost">
        <id>gov</id>
        <dbConfig>govregistry</dbConfig>
        <readOnly>false</readOnly>
        <enableCache>true</enableCache>
        <registryRoot>/</registryRoot>
</remoteInstance>

<mount path="/_system/governance" overwrite="true">
        <instanceId>gov</instanceId>
        <targetPath>/_system/governance</targetPath>
</mount>

<mount path="/_system/config" overwrite="true">
       <instanceId>gov</instanceId>
       <targetPath>/_system/config</targetPath>
</mount>
```

4. Repeat the above step in the `<AM_HOME>/repository/conf/registry.xml` file as well.

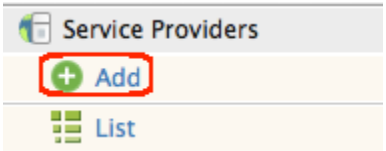
Next, let us look at the SSO configurations.

Configuring WSO2 Identity Server as a SAML 2.0 SSO Identity Provider

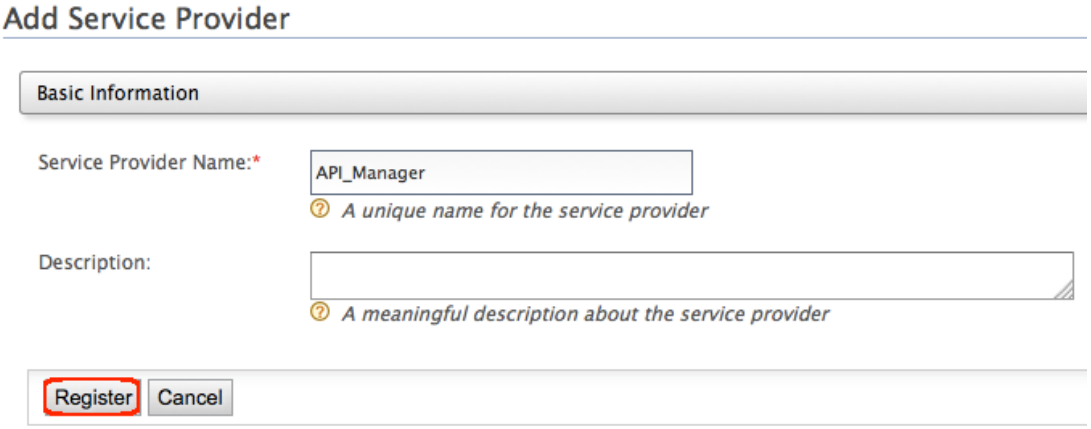
1. Start the IS server and log in to its Management Console UI (`https://localhost:9443/carbon`).

   If you use login pages that are hosted externally to log in to the Identity Server, give the absolute URLs of those login pages in the `authenticators.xml` and `application-authenticators.xml` files in the `<IS_HOME>/repository/conf/security` directory.

2. Select **Add** under **Service Providers** menu.

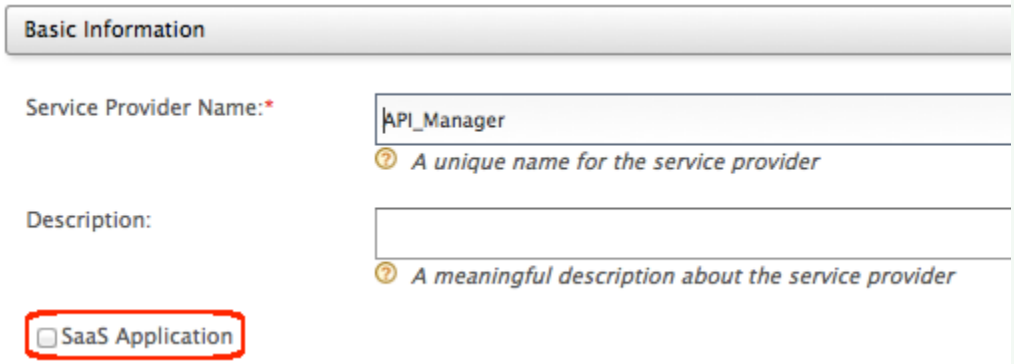


3. Give a service provider name and click **Register**.



Tip: If you are working in a multi tenanted environment and you want all tenants to be able to log in to the APIM Web applications, you must click the **SaaS Application** option that appears after registering the service provider.

If not, only users in the current tenant domain (the one you are defining the service provider in) will be allowed to log in to the Web application and you have to register new service providers for all Web applications (API Store and API Publisher in this case) from each tenant space separately. For example, let's say you have three tenants as TA, TB and TC and you register the service provider in TA only. If you tick the **SaaS Application** option, all users in TA, TB, TC tenant domains will be able to log in. Else, only users in TA will be able to log in.



4. You are navigated to the detailed configuration page. Expand **SAML2 Web SSO Configuration** inside the **In bound Authentication Configuration** section.
5. Expand **SAML2 Web SSO Configuration** inside the **Inbound Authentication Configuration** section.
6. Pro vide the following configurations to register the API Manager Web applications as SSO service providers.

In the following configurations, use the exact values that were used to configure the API Manager Web applications.

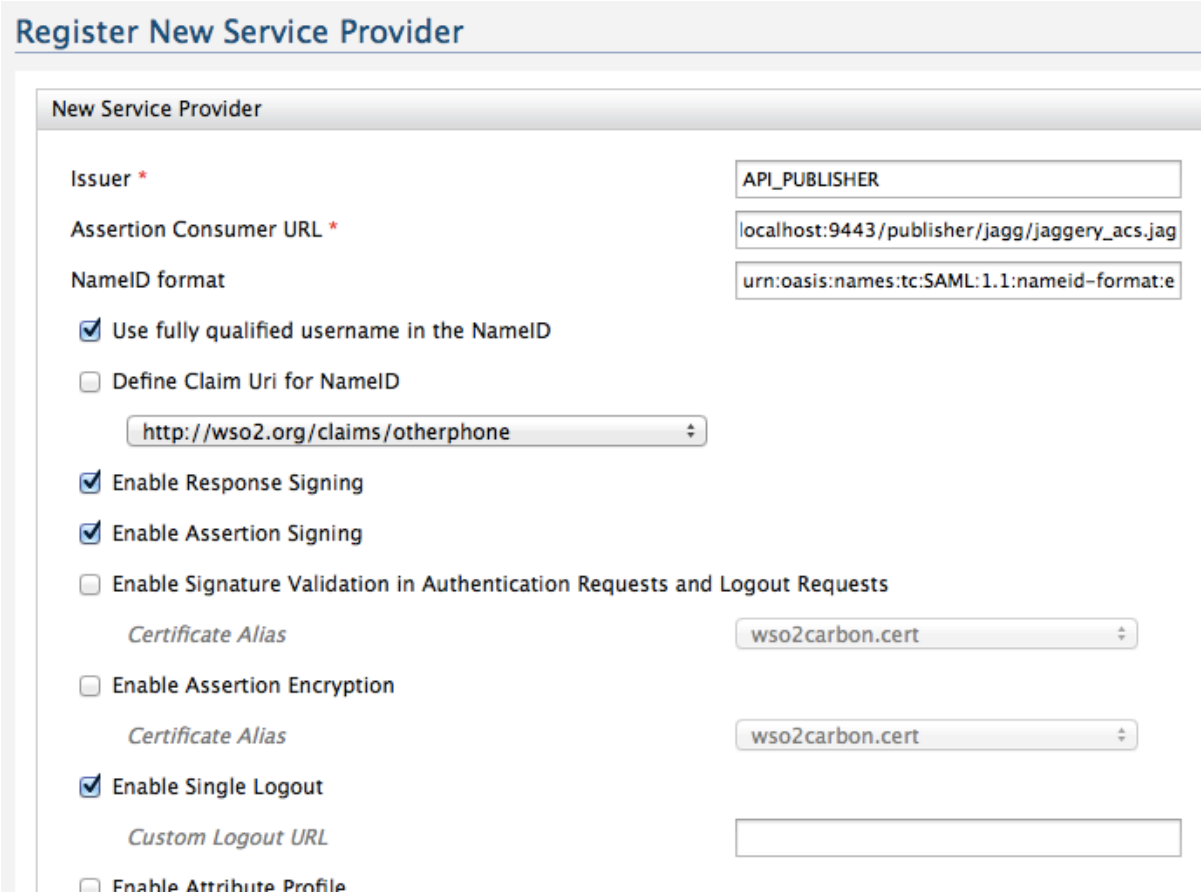
To register API Publisher as an SSO service provider:

- Issuer : API_PUBLISHER
- Assertion Consumer URL: https://localhost:9443/publisher/jagg/jaggery_acs.jag. Change the IP and port accordingly. This is the URL for the acs page in your running publisher app.
- Select the following options:
  - **Use fully qualified username in the NameID**
  - **Enable Response Signing**
  - **Enable Assertion Signing**
  - **Enable Single Logout**
- Click **Register** once done.

To register API Store as an SSO service provider:

- Issuer : API_STORE
- Assertion Consumer URL: https://localhost:9443/store/jagg/jaggery_acs.jag. Change the IP and port accordingly. This is the URL for the acs page in your running store app.
- Select the following options:
  - **Use fully qualified username in the NameID**
  - **Enable Response Signing**
  - **Enable Assertion Signing**
  - **Enable Single Logout**
- Click **Register** once done.

For example:



Configuring WSO2 API Manager Apps as SAML 2.0 SSO Service Providers

1. Open `<AM_Home>/repository/deployment/server/jaggeryapps/publisher/site/conf/site.json` and modify the following configurations found under **ssoConfiguration**.
   - **keyStoreName**: The keystore of the running IDP. As you use a remote instance of WSO2 IS here, you can import the public certificate of the IS keystore to the APIM and then point to the APIM keystore. The default keystore of the APIM is `<APIM_HOME>/repository/resources/security/wso2carbon.jks`. **Be sure to give the full path of the keystore here.**
   - **keyStorePassword:** Password for the above keystore.
   - **identityAlias**: wso2carbon.
   - **enabled:** Set this value to **true** to enable SSO in the application.
   - **issuer:** API_PUBLISHER. This value can change depending on the **Issuer** value defined in WSO2 IS SSO configuration above.
   - **identityProviderURL:** https://localhost:9444/samlsso. Change the IP and port accordingly. This is the redirecting SSO URL in your running WSO2 IS server instance.
2. Similarly, configure the API Store with SSO. The only difference in API Store SSO configurations is setting **API_STORE** as the **issuer**.
3. Access the API Publisher : https://localhost:<Port number>/publisher (e.g., https://localhost:9443/publisher). Observe the request redirect to the WSO2 IS SAML2.0 based SSO login page. For example,

   ? Unknown Attachment

4. Enter user credentials. If the user authentication is successful against WSO2 IS, it will redirect to the API Publisher Web application with the user already authenticated.
5. Access the API Store application, click its **Login** link (top, right-hand corner) and verify that the same user is already authenticated in API Store.

Even with SSO enabled, if the user doesn't have sufficient privileges to access API Publisher/Store or any other application, s/he will not be authorized to access them.

If there are many WSO2 products in your environment, you can configure SSO for the management consoles of all products by changing the `SAML2SSOAuthenticator` configuration in `<PRODUCT_HOME>/repository/conf/security/authenticators.xml` file as follows:

- Set `disabled` attributes in `<Authenticator>` element to `false`
- `ServiceProviderID` : In this example, it is the issuer name of the service provider created in step 1
- `IdentityProviderSSOServiceURL` : In this example, it is the Identity Server port

```
<Authenticator name="SAML2SSOAuthenticator" disabled="false">
        <Priority>10</Priority>
        <Config>
            <Parameter
name="LoginPage">/carbon/admin/login.jsp</Parameter>
            <Parameter name="ServiceProviderID">carbonserver1</Parameter>
            <Parameter
name="IdentityProviderSSOServiceURL">https://localhost:9444/samlsso&lt;/Pa
rameter>
           <Parameter
name="NameIDPolicyFormat">urn:oasis:names:tc:SAML:1.1:nameid-format:unspec
ified</Parameter>
</Config>
```

Make sure the `<priority>` of the `SAML2SSOAuthenticator` is less than that of the `BasicAuthenticator` handler. See here for more information.

Maintaining Primary and Secondary Logins

In a standalone deployment of the API Manager instance, users of the API Store can have a secondary login name in addition to the primary login name. This gives the user flexibility to provide either an email or a user name to log in. You can configure the API Store to treat both login names as belonging to a single user. Users can invoke APIs with the same Accestoken without having to create a new one for the secondary login.

You can configure this capability using the steps below.

1. Configure user login under the `<OAuth>` element in <APIM_HOME>/repository/conf/identity.xml file.
    a. Mention your primary and secondary login names. Set the `primary` attribute of the primary login to `true` and the `primary` attribute of the secondary login to `false`.
    b. Primary login doesn't have a `ClaimUri`. Leave this field empty.
    c. Provide the correct `ClaimUri` value for the secondary login

An example is given below:

```
<OAuth>
    ..... . ....
    <LoginConfig>
        <UserIdLogin primary="true">
             <ClaimUri></ClaimUri>
        </UserIdLogin>
        <EmailLogin primary="false">
             <ClaimUri>http://wso2.org/claims/emailaddress</ClaimUri>
        </EmailLogin>
     </LoginConfig>
</OAuth>
```

2. In the API Store of a distributed setup, the `serverURL` element in the api-manager.xml file should point to the key manager instance's service endpoint. This allows users to connect to the key manager's user store to perform any operations related to API Store such as login, access token generation etc. For example,

```
<AuthManager>
   <!--Server URL of the Authentication service -->
   <ServerURL>https://localhost:9444/services/</ServerURL>

   <!-- Admin username for the Authentication manager. -->
   <Username>admin</Username>

   <!-- Admin password for the Authentication manager.-->
   <Password>admin</Password>
</AuthManager>
```

3. In the distributed setup, API Store's user store needs to point to the key manager user store.

Keeping the secondary login name unique for each user is the user's responsibility.

Adding Internationalization and Localization

The API Manager comes with two Web interfaces as API Publisher and API Store. The instructions given below show how to localize the Web interface of API Publisher in Spanish. Same instructions apply to localize API Store as well in any other language.

Changing the browser settings

1. First, set the browser language to a preferred language. Instructions should be available in the web browser's user guide. For example, language can be selected in Google Chrome through **Settings -> Show advanced settings -> Languages** menu.
2. Set the browser's encoding type to UTF-8.

Introduction to resource files

3. Go to `<AM_HOME>/repository/deployment/server/jaggeryapps/publisher` directory where `<AM_HOME>` is the API Manager distribution home.
4. There are two types of resource files used to define localization strings in WSO2 API Manager.
   - The resource file used to store the strings defined in .jag files according to browser locale (For example, `locale_en.json`) is located in `.../publisher/site/conf/locales/jaggery` folder.
   - The resource file i18nResources.json, which is used to store strings defined in client-side javascript files such as pop-up messages when a UI event is triggered, is located in .../publisher/site/conf/locales/js folder.

For example,

```
./locales/jaggery:
locale_en.json

./locales/js:
i18nResources.json
```

To implement localization support for jaggery, we use its in-built script module 'i18n'. For more information, refer to http://jaggeryjs.org/apidocs/i18n.jag.

Localizing strings in Jaggery files

5. To localize the API publisher to Spanish, first localize the strings defined in jaggery files. Create a new file by the name **locale_{lolcaleCode}.json** inside **...publisher/site/conf/locales/jaggery** folder. For example, if the language set in the browser is Spanish, the locale code is **es** and the file name should be **locale_es.json**.
6. Add the key-value pairs to locale_es.json file. For an example on adding key value pairs, refer to **locale_en.j son** file in **...publisher/site/conf/locales/jaggery** folder. It is the default resource file for jaggery.

   In addition, a section of a sample locale_es.json file is shown below for your reference.

```
{
    "name" : "Nombre" ,
    "context" : "Contexto" ,
    "version" : "Versión" ,
    "description" : "Descripción" ,
    "visibility" : "Visibilidad" ,
    "thumbnail" : "Uña del pulgar" ,
    "endpoint" : "Producción URL" ,
    "sandbox" : "Cajón de arena URL" ,
```

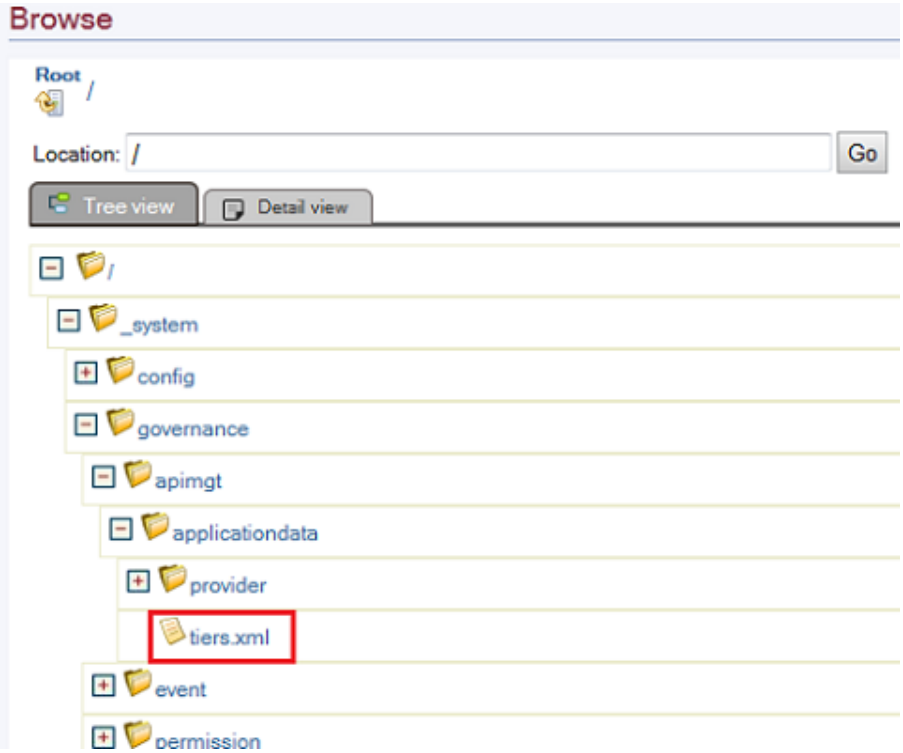
Localizing strings in client-side Javascript files

7. To localize the javascript UI messages, navigate to publisher/site/conf/locales/js folder and update **i18nReso urces.json** file with relevant values for the key strings.
8. Once done, open the API Publisher web application in your browser (`https: //<YourHostName>:9443/p ublisher)`.
9. Note that the UI is now changed to Spanish.

Adding New Throttling Tiers

API Manager admins can add new throttling tiers and define extra properties to throttling tiers using the management console as discussed below. For a description of throttling tiers, see API-level throttling.

1. Log in to the API Manager's Management Console and select **Browse** under **Resources** menu.
2. Select the file: `/_system/governance/apimgt/applicationdata/tiers.xml`.



3. In the **Contents** panel, click **Edit as text** link and the throttling policy opens.
4. You can add a new policy configuration by editing the XML code. For example, we have added a new tier called `Platinum` by including the following XML code block soon after the `<throttle:MediatorThrottleAssertion>` element.

   Tier DisplayName : You can add this **optional** attribute to each throttle ID of tiers.xml file in order to decouple the throttle policy name defined in tiers.xml from the tier name showing in APIPublisher/Store UIs. That is, a user can add a different throttle display name to appear in APIPublisher/Store UIs without changing the throttle ID policy name. The configuration below has a displayName as `platino` for the throttle value `platinum`. This value is displayed in APIPublisher/Store apps.

   Tier Attributes : In the configuration below, there's a commented out XML section starting from the XML tag `<throttle:Attributes>`. You can use it to define additional attributes related to each throttling tier definition. For example, if the throttling tier `Platinum` has attributes called `PaymentPlan` and `Availability`, first uncomment the `<throttle:Attributes>` section and then define the new attributes as follows:

```
<wsp:Policy>
 <throttle:ID throttle:type="ROLE"
throttle:displayName="platino">Platinum</throttle:ID>
   <wsp:Policy>
     <throttle:Control>
        <wsp:Policy>
          <throttle:MaximumCount>50</throttle:MaximumCount>
             <throttle:UnitTime>60000</throttle:UnitTime>
               <!--It's possible to define tier level attributes as below for
each tier level.For eg:Payment Plan for a tier-->
                  <wsp:Policy>
                     <throttle:Attributes>
                        <!--throttle:Attribute1>xxxx</throttle:Attribute1-->
                        <!--throttle:Attribute2>xxxx</throttle:Attribute2-->
                        <throttle:PaymentPlan>monthly</throttle:PaymentPlan>
                        <throttle:Availability>FullTime</throttle:Availability>
                     </throttle:Attributes>
                  </wsp:Policy>
        </wsp:Policy>
     </throttle:Control>
   </wsp:Policy>
</wsp:Policy>
```

5. After the edits, click **Save Content**. Your new throttling policy (Platinum) is now successfully saved in the Repository used by WSO2 API Manager. You can view this new throttle tier available for selection when creating a new API through the API Publisher.

Maintaining Separate Production and Sandbox Gateways

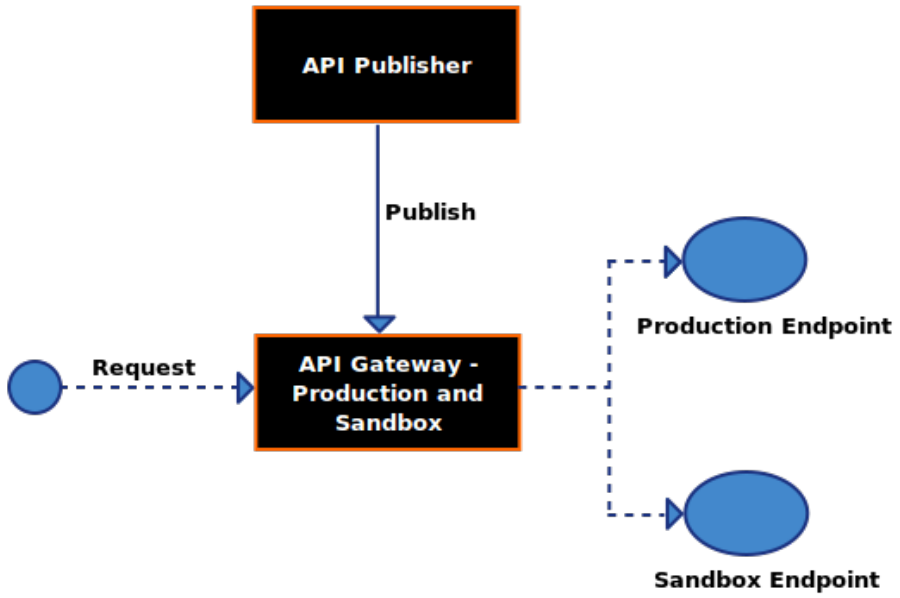
With WSO2 API Manager, you can maintain a production and a sandbox endpoint for a given API. The production endpoint is the actual location of the API, whereas the sandbox endpoint points to its testing/pre-production environment.

When you publish an API using the API Publisher, it gets deployed on the API Gateway. By default, there's a single Gateway instance (deployed either externally or embedded within the publisher), but you can also set up multiple Gateways:

- Single Gateway to handle both production and sandbox requests
- Multiple Gateways to handle production and sandbox requests separately

Single Gateway to handle both production and sandbox requests

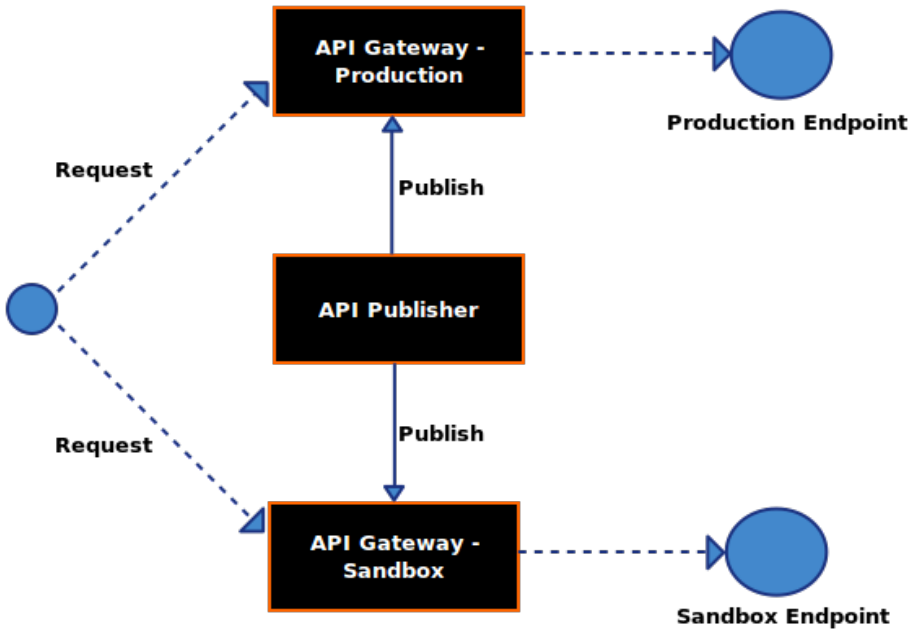
This is the default scenario. Because this Gateway instance handles both production and sandbox token traffic, it is called a hybrid API Gateway. When an API request comes to the API Gateway, it checks whether the requesting token is of type PRODUCTION or SANDBOX and forwards the request to the appropriate endpoint. The diagram below depicts this scenario.



Multiple Gateways to handle production and sandbox requests separately

Having a single gateway instance to pass through both types of requests can negatively impact the performance of the production server. To avoid this, you can set up separate API Gateways. The production API Gateway handles requests that are made using PRODUCTION type tokens and the sandbox API Gateway handles requests that are made using SANDBOX type tokens.

The diagram below depicts this using two Gateways:



In either of the two approaches, if an API Gateway receives an invalid token, it returns an error to the requesting client saying that the token is invalid.

You configure production and sandbox gateways using the `<Environments>` element in the `<AM_HOME>/repository/conf/api-manager.xml` file as shown in the following example:

```
<Environments>
 <Environment type="production">
        <Name>Production and Sandbox</Name>
        <ServerURL>https://localhost:9445/services/&lt;/ServerURL>
        <Username>admin</Username>
        <Password>admin</Password>

<GatewayEndpoint>http://localhost:8282,https://localhost:8245&lt;/GatewayEndpoint>
    </Environment>

    <Environment type="sandbox">
        <Name>Production and Sandbox</Name>
        <ServerURL>https://localhost:9448/services/&lt;/ServerURL>
        <Username>admin</Username>
        <Password>admin</Password>

<GatewayEndpoint>http://localhost:8285,https://localhost:8248&lt;/GatewayEndpoint>
    </Environment>
</Environments>
```

The `type` attribute of the `<Environment>` element can take the following values:

- **Production**: A production type Gateway
- **Sandbox**: A sandbox type Gateway
- **Hybrid**: The Gateway handles both types of tokens

Changing the Default Transport

APIs are synapse configurations in the back-end and API Manager accesses them using HTTP-NIO transport by default. You can switch to a different transport such as PassThrough. To change the default transport of API Manager, go to `<APIM_HOME>/repository/conf/axis2` folder and rename `axis2.xml_PT` file to `axis2.xml`. Similarly, you can switch back to NHTTP by simply renaming `axis2.xml_NHTTP` file to `axis2.xml`.

The following topics explain HTTP-NIO and PassThrough transports:

- HTTP-NIO transport
- HTTP PassThrough transport

HTTP-NIO transport

HTTP-NIO transport is a module of the Apache Synapse project. Apache Synapse as well as WSO2 APIM ship the HTTP-NIO transport as the default HTTP transport implementation. The two classes that implement the receiver and sender APIs are `org.apache.synapse.transport.nhttp.HttpCoreNIOListener` and `org.apache.synapse.transport.nhttp.HttpCoreNIOSender` respectively. These classes are available in the JAR file named `synapse-nhttp-transport.jar`. This non-blocking transport implementation improves performance. The transport implementation is based on Apache HTTP Core - NIO and uses a configurable pool of non-blocking worker threads to grab incoming HTTP messages off the wire.

Transport receiver parameters

> In transport parameter tables, literals displayed in italic mode under the "Possible Values" column should be considered as fixed literal constant values. Those values can be directly put in transport configurations.

Parameter Name	Description	Requried	Possible Values	Default Value

port	The port on which this transport receiver should listen for incoming messages.	No	A positive integer less than 65535	8280
non-blocking	Setting this parameter to true is vital for reliable messaging and a number of other scenarios to work properly.	Yes	*true*	
bind-address	The address of the interface to which the transport listener should bind.	No	A host name or an IP address	127.0.0.1
hostname	The host name of the server to be displayed in service EPRs, WSDLs etc. This parameter takes effect only when the WSDLEPRPrefix parameter is not set.	No	A host name or an IP address	localhost
WSDLEPRPrefix	A URL prefix which will be added to all service EPRs and EPRs in WSDLs etc.	No	A URL of the form <protocol>://<hostname>:<port>/	

Transport sender parameters

Parameter Name	Description	Requried	Possible Values	Default Value
http.proxyHost	If the outgoing messages should be sent through an HTTP proxy server, use this parameter to specify the target proxy.	No	A host name or an IP address	
http.proxyPort	The port through which the target proxy accepts HTTP traffic.	No	A positive integer less than 65535	
http.nonProxyHosts	The list of hosts to which the HTTP traffic should be sent directly without going through the proxy.	No	A list of host names or IP addresses separated by '|'	
non-blocking	Setting this parameter to true is vital for reliable messaging and a number of other scenarios to work properly.	Yes	*true*	

HTTP PassThrough transport

HTTP PassThrough Transport is the default, non-blocking HTTP transport implementation based on HTTP Core NIO and is specially designed for streaming messages. It is similar to the old message relay transport, but it does not care about the content type and simply streams all received messages through. It also has a simpler and cleaner model for forw arding messages back and forth. It can be used as an alternative to the NHTTP transport.

The HTTP PassThrough Transport is enabled by default. If you want to use the NHTTP transport instead, uncomment the relevant NHTTP transport entries in `axis2.xml` and comment out the HTTP PassThrough transport entries. T he PassThrough Transport does not require the binary relay builder and expanding formatter.

Connection throttling

With the HTTP PassThrough and HTTP NIO transports, you can enable connection throttling to restrict the number

of simultaneous open connections. To enable connection throttling, edit the `<PRODUCT_HOME>/repository/conf/nhttp.properties` (for the HTTP NIO transport) or `<PRODUCT_HOME>/repository/conf/passthru.properties` (for the PassThrough transport) and add the following line: `max_open_connections = 2`

This will restrict simultaneous open incoming connections to 2. To disable throttling, delete the `max_open_connections` setting or set it to -1.

> Connection throttling is never exact. For example, setting this property to 2 will result in roughly two simultaneous open connections at any given time.

> WSO2 products do not use the HTTP/S servlet transport configurations that are in `axis2.xml` file. Instead, they use Tomcat-level servlet transports, which are used by the management console in `<PRODUCT_HOME>/repository/conf/tomcat/catalina-server.xml` file.

Running the Product on a Preferred Profile

When a WSO2 product server starts, it starts all features and related artifacts bundled in the product. Multi-profile support allows you to run the product on a selected profile so that only features specific to that profile along with common features start up with the server. This enables better resource utilization.

Execute one of the following commands to start a product on a preferred profile.

OS	Command
Windows	`<PRODUCT_HOME>/bin/wso2server.bat -Dprofile=<preferred-profile> --run`
Linux/Solaris	`sh <PRODUCT _HOME>/bin/wso2server.sh -Dprofile=<preferred-profile>`

Given below are the profiles available in WSO2 API Manager. They are based on the main components of API Manager.

Profile	Command Option with Profile Name	Description
Gateway manager	`-Dprofile=gateway-manager`	Used when the API Gateway acts as a manager node in a cluster. This profile starts front-end/UI features such as login as well as back-end services that allow the product instance to communicate with other nodes in the cluster.
Gateway worker	`-Dprofile=gateway-worker`	Used when API Gateway acts as a worker node in a cluster. This profile only starts the back-end features for data processing and communicating with the manager node.
Key Manager	`-Dprofile=api-key-manager`	Starts only the features relevant to the Key Manager component of API Manager.
API Publisher	`-Dprofile=api-publisher`	Starts only the front-end/back-end features relevant to the API Publisher Web interface.
API Store	`-Dprofile=api-store`	Starts only the front-end/back-end features relevant to the API Store Web interface.

> Note that the WSO2 products platform currently doesn't block/allow Web applications depending on profiles. Starting a product on a preferred profile only blocks/allows the relevant OSGI bundles. As a result, even if you start the server on a profile such as the `api-store` for example, you will still be able to access the API Publisher Web application.

How multi-profiling works

Starting a product on a preferred profile starts only a subset of features bundled in the product. In order to identify what feature bundles apply to which profile, each product maintains a set of `bundles.info` files in `<PRODUCT_HOME>/repository/components/<profile-name>/configuration/org.eclipse.equinox.simpleconfigurator` directories. The `bundles.info` files contain references to the actual bundles. Note that `<profile-name>` in the directory path refers to the name of the profile. For example, when there's a product profile named webapp, references to all the feature bundles required for webapp profile to function are in a `bundles.info` file saved in `<PRODUCT_HOME>/repository/components/webapp/configuration/org.eclipse.equinox.simpleconfigurator` directory.

Note that when you start the server without using a preferred profile, the server refers to `<PRODUCT_HOME>/repository/components/default/configuration/org.eclipse.equinox.simpleconfigurator/bundles.info` file by default. This file contains references to all bundles in `<PRODUCT_HOME>/repository/components/plugins` directory, which is where all components/bundles of a product are saved.

Tuning Performance

This section describes some recommended performance tuning configurations to optimize the API Manager. It assumes that you have set up the API Manager on Unix/Linux, which is recommended for a production deployment. We also recommend a distributed API Manager setup for most production systems. Out of all components of an API Manager distributed setup, the API Gateway is the most critical, because it handles all inbound calls to APIs. Therefore, we recommend you to have at least a 2-node cluster of API Gateways in a distributed setup.

> The values we discuss below are only general recommendations for the API Gateway. Generally, they work best when there are 350 to 30000 calls per second to the API Gateway, but these values might not be optimal for the specific hardware configurations in your environment. We recommend you to carry out load tests on your environment to tune the API Manager accordingly.

Improvement Area	Performance Recommendations
API Gateway nodes	Increase memory allocated by modifying `/bin/wso2server.sh` with the following setting: • -Xms2048m -Xmx2048m -XX:MaxPermSize=1024m

NHTTP transport of API Gateway	Recommended values for `<AM_HOME>/repository/conf/nhttp.properties` file are given below. **Note** that the commented out values in this file are the default values that will be applied if you do not change anything. **Property descriptions:** snd_t_core — Transport sender worker pool's initial thread count snd_t_max — Transport sender worker pool's maximum thread count snd_io_threads — Sender-side IO workers, which is recommended to be equal to the number of CPU cores. I/O reactors usually employ a small number of dispatch threads (often as few as one) to dispatch I/O event notifications to a greater number (often as many as several thousands) of I/O sessions or connections. Generally, one dispatch thread is maintained per CPU core. snd_alive_sec — Sender-side keep-alive seconds snd_qlen — Sender queue length, which is infinite by default **Recommended values:** # HTTP Sender thread pool parameters • snd_t_core=200 • snd_t_max=250 • snd_alive_sec=5 • snd_qlen=-1 • snd_io_threads=16 # HTTP Listener thread pool parameters • lst_t_core=200 • lst_t_max=250 • lst_alive_sec=5 • lst_qlen=-1 • lst_io_threads=16

PassThrough transport of API Gateway	Recommended values for `<AM_HOME>/repository/conf/passthru-http.properties` file are given below. **Note** that the commented out values in this file are the default values that will be applied if you do not change anything. **Property descriptions** worker_thread_keepalive_sec — Defines the keep-alive time for extra threads in the worker pool worker_pool_queue_length — Defines the length of the queue that is used to hold runnable tasks to be executed by the worker pool io_threads_per_reactor — Defines the number of IO dispatcher threads used per reactor http.max.connection.per.host.port — Defines the maximum number of connections per host port worker_pool_queue_length — Determines the length of the queue used by the PassThrough transport thread pool to store pending jobs. **Recommended values** • worker_thread_keepalive_sec : Default value is 60s. This should be less than the socket timeout. • worker_pool_queue_length : Set to -1 to use an unbounded queue. If a bound queue is used and the queue gets filled to its capacity, any further attempts to submit jobs will fail, causing some messages to be dropped by Synapse. The thread pool starts queuing jobs when all the existing threads are busy and the pool has reached the maximum number of threads. So, the recommended queue length is -1. • io_threads_per_reactor : Value is based on the number of processor cores in the system. (Runtime.getRuntime().availableProcessors()) • http.max.connection.per.host.port : Default value is 32767, which works for most systems but you can tune it based on your operating system (for example, Linux supports 65K connections). • http.socket.timeout=120000 • worker_pool_size_core=400 • worker_pool_size_max=500 • io_buffer_size=16384 • http.socket.timeout=60000 • snd_t_core=200 • snd_t_max=250 • snd_io_threads=16 • lst_t_core=200 • lst_t_max=250 • lst_io_threads=16 Make the number of threads equal to the number of processor cores.

Key management nodes	Set the following in `<APIM_HOME>/repository/conf/axis2/axis2_client.xml` file:
	`<parameter name="defaultMaxConnPerHost">1000</parameter>`
	`<parameter name="maxTotalConnections">30000</parameter>`
	
	Set the MySQL maximum connections:
	`mysql> show variables like "max_connections";`
	` max_connections was 151`
	` set to global max_connections = 250;`
	
	Set the open files limit to 200000 by editing the `/etc/sysctl.conf` file:
	`sudo sysctl -p`
	
	Set the following in `CatlinaServer.sh` batch file:
	`maxThreads="750"`
	`minSpareThreads="150"`
	`disableUploadTimeout="false"`
	`enableLookups="false"`
	`connectionUploadTimeout="120000"`
	`maxKeepAliveRequests="600"`
	`acceptCount="600"`
	
	Set the following connection pool elements in `<APIM_HOME>/repository/conf/datasources/master-datasources.xml` file:
	`<maxActive>50</maxActive>`
	`<maxWait>60000</maxWait>`
	`<testOnBorrow>true</testOnBorrow>`
	`<validationQuery>SELECT 1</validationQuery>`
	`<validationInterval>30000</validationInterval>`
	
	Note that you set the `<testOnBorrow>` element to `true` and provide a validation query (e.g., in Oracle, `SELECT 1 FROM DUAL`), which is run to refresh any stale connections in the connection pool. Set a suitable value for the `<validationInterval>` element, which defaults to 30000 milliseconds. It determines the time period after which the next iteration of the validation query will be run on a particular connection. It avoids excess validations and ensures better performance.

Directing the Root Context to API Store

WSO2 API Manager maintains separate UIs for API publishers and subscribers as the API Publisher and API Store. The root context of the API Manager is set to direct to the API Publisher Web interface by default. For example, assume that the API Manager is hosted on a domain named `apis.com` with default ports. The URLs of the API Store and API Publisher Web interfaces will be as follows:

- API Store - https://apis.com:9443/store
- API Publisher - https://apis.com:9443/publisher

If you open the root context (https://apis.com:9443) in your browser, it directs to the API Publisher by default. Follow the steps below to make it direct to the API Store instead of the API publisher:

1. Open the bundle `<AM_HOME>/repository/components/plugins/org.wso2.am.styles_1.x.x.jar`.
2. Open the `component.xml` file that is inside `META-INF` directory.
3. Change the <context-name> element, which points to publisher by default, to store:

```
<context>
        <context-id>default-context</context-id>
        <context-name>store</context-name>
        <protocol>http</protocol>
        <description>API Publisher Default Context</description>
 </context>
```

4. Restart the server.
5. Open the default context (https://apis.com:9443) again in a browser and note that it directs to the API Store.

Changing the Default Ports with Offset

When you run multiple WSO2 products, multiple instances of the same product, or multiple WSO2 product clusters on the same server or virtual machines (VMs), you must change their default ports with an offset value to avoid port conflicts. The default HTTP and HTTPS ports (without offset) of a WSO2 product are 9763 and 9443 respectively. Port offset defines the number by which all ports defined in the runtime such as the HTTP/S ports will be changed. For example, if the default HTTP port is 9763 and the port offset is 1, the effective HTTP port will change to 9764. For each additional WSO2 product instance, you set the port offset to a unique value. The default port offset is 0.

There are two ways to set an offset to a port:

- Pass the port offset to the server during startup. The following command starts the server with the default port incremented by `3:./wso2server.sh -DportOffset=3`
- Set the Ports section of `<PRODUCT_HOME>/repository/conf/carbon.xml` as follows:`<Offset>3</Offset>`

Usually, when you offset the port of the server, all ports it uses are changed automatically. However, there are few exceptions as follows in which you have to change the ports manually according to the offset.

Changing endpoints of default APIs

After offsetting a port, be sure to edit any hard-coded endpoints used in a product, if there are any, according to the offset. There are few default APIs deployed in the API Manager with hard-coded endpoints. For example, the **Login API**'s Token endpoint URL is hardcoded as follows: `<address uri="https://localhost:9443/oauth2endpoints/token">`. If you offset the Key Manger node's port by 2, change the token endpoint URL to `<address uri="https://localhost:9445/oauth2endpoints/token"/>`.

Find all default APIs of the API Manager in `<APIM_HOME>/repository/deployment/server/synapse-configs/default/api` folder. Few examples are **Authorize API, Login API, Token API** and **Revoke API.**

Changing the Thrift client and server ports

The port offset specified earlier in carbon.xml does not affect the ports of the Thrift client and server because Thrift is run as a separate server within WSO2 servers. Therefore, you must change the Thrift ports separately using `<ThfirtClientPort>` and `<ThriftServerPort>` elements in the `<APIM_HOME>/repository/conf/api-manager.xml` file. For example, the following configuration sets an offset of 2 to the default Thrift port, which is 10397:

```
<!--
        Configurations related to enable thrift support for key-management related
communication.
        If you want to switch back to Web Service Client, change the value of
"KeyValidatorClientType" to "WSClient".
        In a distributed environment;
        -If you are at the Gateway node, you need to point "ThriftClientPort" value to
the "ThriftServerPort" value given at KeyManager node.
        -If you need to start two API Manager instances in the same machine, you need
to give different ports to "ThriftServerPort" value in two nodes.
        -ThriftServerHost - Allows to configure a hostname for the thrift server. It
uses the carbon hostname by default.
        -->

        <KeyValidatorClientType>ThriftClient</KeyValidatorClientType>
        <ThriftClientPort>10399</ThriftClientPort>
        <ThriftClientConnectionTimeOut>10000</ThriftClientConnectionTimeOut>
        <ThriftServerPort>10399</ThriftServerPort>
    <!--ThriftServerHost>localhost</ThriftServerHost-->
    <EnableThriftServer>true</EnableThriftServer>
```

When you run multiple instances of the API Manager in distributed mode, the Gateway and Key Manager (used for validation and authentication) can run on two different JVMs.Communication between API Gateway and Key When the API Gateway receives API invocation calls, it
contacts the API Key Manager service for verification (given that caching is not enabled at the Gateway level).
nager happens in either of the following ways:

- Through a Web service call
- Through a Thrift call

The default communication mode is using Thrift. Assume that the Gateway port is offset by 2, Key Manager port by 5 and the default Thrift port is 10397. If the Thrift ports are changed by the offsets of Gateway and Key Manager, the Thrift client port (Gateway) will now be 10399 while the Thrift server port (Key Manager) will change to 10402. This causes communication between the Gateway and Key Manager to fail because the Thrift client and server ports are different.

To fix this, you must change the Thrift client and server ports of Gateway and Key Manager to the same value. In this case, the difference between the two offsets is 3, so you can either increase the default Thrift client port by 3 or else reduce the Thrift server port by 3.

Changing the offset of the Workflow Callback Service

The API Manager has a Service which listens for workflow callbacks. This service configuration can be found at `<AM_HOME>/repository/deployment/server/synapse-configs/default/proxy-services/WorkflowCallbackService.xml`. Open this file and change the port value of the <address uri accordingly.

For example,

```
<address
uri="https://localhost:9445/store/site/blocks/workflow/workflow-listener/ajax/workflow
-listener.jag" format="rest"/>
```

For a list of all default ports opened in WSO2 API Manager, see Default Ports of WSO2 Products.

Adding Links to Navigate Between the Store and Publisher

By default, there are no links in the UIs of the API Store and API Publisher applications to traverse between the two

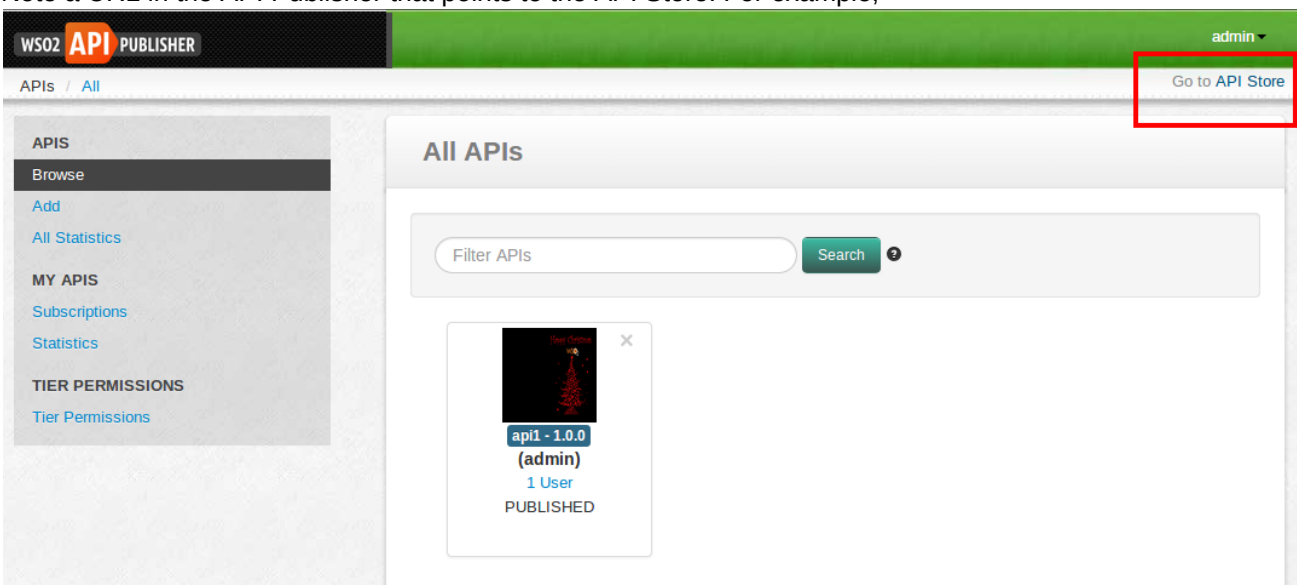
apps.

To add a link in API Publisher to API Store:

1. In `<AM_HOME>/repository/conf/api-manager.xml` file, set the `<DisplayURL>` to true and provide the URL of the Store.

```
<APIStore>
       <DisplayURL>true</DisplayURL>
       <URL>https://${carbon.local.ip}:${mgt.transport.https.port}/store&lt;/URL>
</APIStore>
```

2. Note a URL in the API Publisher that points to the API Store. For example,

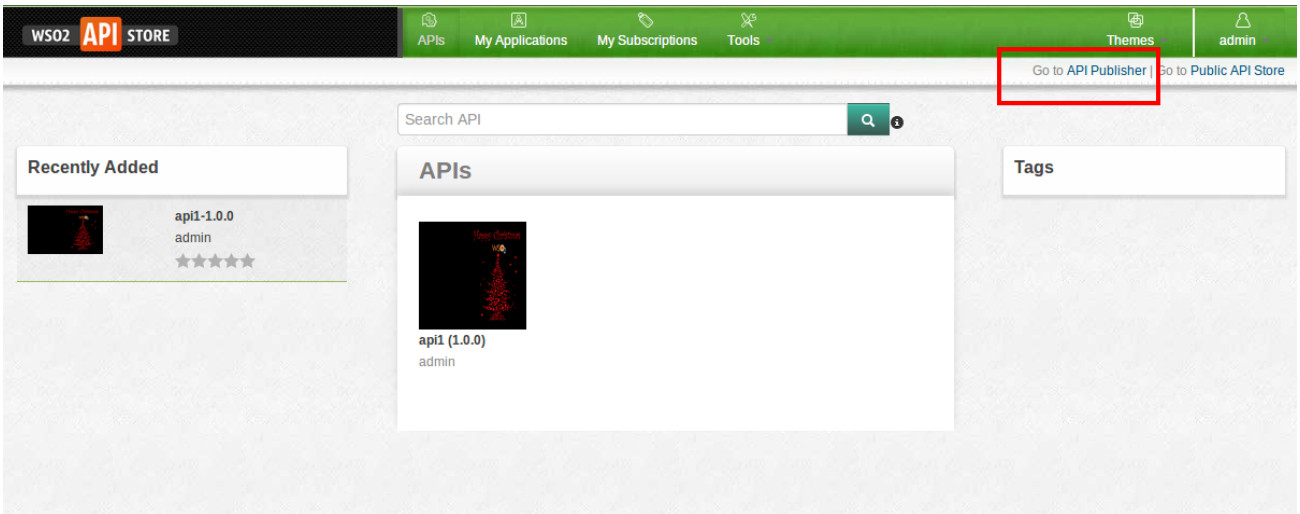


To add a link in API Store to API Publisher:

1. In `<AM_HOME>/repository/conf/api-manager.xml` file, set the `<DisplayURL>` to true and provide the URL of the Publisher.

```
<APIPublisher>
       <DisplayURL>true</DisplayURL>

<URL>https://${carbon.local.ip}:${mgt.transport.https.port}/publisher&lt;/URL>
</APIPublisher>
```

2. Note a URL in the API Store that points to the API Publisher. For example,



Migrating the API Manager

If you have multiple instances of the WSO2 API Manager and want to move your data and deployment artifacts from one instance to another (such as moving from development to test or production), follow the steps below.

1. Get a data dump from all the tables in the apimgt schema and dump them to the schema in the new environment.
2. Open `<APIM_HOME>/repository/conf/datasources/master-datasources.xml` file and provide the datasource configurations for the following databases in the new environment.
   - User Store
   - Registry database
   - API Manager Databases
3. Move all your synapse configurations by copying and replacing `<APIM_HOME>/repository/deployment/server/synapse-config/default` directory to the same directory in the new environment.

   > If you changed the default URLs in `AuthorizeAPI.xml` and `TokenAPI.xml` files, do not replace them when copying. They are application-specific APIs.

   Migrate tenants

4. If you have **multiple tenants** added to your API Manager instance, follow the steps below to migrate tenant configurations:
   a. Copy the contents from `<APIM_HOME>/repository/tenants` directory to the same directory in the new environment.
   b. Execute the following steps for all tenants in your system.

   Migrate external stores

5. If you have **external stores** configured under the `<ExternalAPIStores>` element in `<APIM_HOME>/repository/conf/api-manager.xml` file, follow the steps below:
   a. Log in to APIM management console and click the **Resources -> Browse** menu.
   b. Load `/_system/governance/apimgt/externalstores/external-api-stores.xml` resource in the registry browser UI, configure your external stores there and save.

   Migrate Google analytics

6. If you have **Google Analytics** configured under `<GoogleAnalyticsTracking>` element in `<APIM_HOME>/repository/conf/api-manager.xml` file, follow the steps below:
   a. Log in to APIM management console and go to **Resources -> Browse** menu.
   b. Load `/_system/governance/apimgt/statistics/ga-config.xml` resource in the registry browser UI, configure the Google analytics and save.

Migrate workflows

7. If you have **Workflows** configured under `<WorkFlowExtensions>` element in `<APIM_HOME>/repository/conf/api-manager.xml` file, follow the steps below:
    a. Log in to APIM management console and go to **Resources -> Browse** menu.
    b. Load `/_system/governance/apimgt/applicationdata/workflow-extensions.xml` resource in the registry browser UI, configure your workflows and save.

Upgrading from a Previous Release

See Upgrading from the Previous Release in the following situations:

- The new environment you are migrating to has a different database version. In this case, you must upgrade the older database.
- You want to upgrade from a previous API Manager release to a new one.

Configuring WSO2 Identity Server as the Key Manager

If your production environment already has an instance of WSO2 Identity Server, you can use it as the Key Manager rather than setting up an additional WSO2 API Manager instance to work as the Key Manager. If you set up the Identity Server, you can get the added advantage of being able to use authentication/authorization features specific to the Identity Server.

For setup instructions, see the Clustering Guide.

Configuring Multiple Tenants

The goal of multitenancy is to maximize resource sharing by allowing multiple users (tenants) to log in and use a single sever/cluster at the same time, in a tenant-isolated manner. That is, each user is given the experience of using his/her own server, rather than a shared environment. Multitenancy ensures optimal performance of the system's resources such as memory and hardware and also secures each tenant's personal data.

You can register tenant domains using the Management Console of WSO2 products.

This section covers the following topics:

- Multi Tenant Architecture
- Managing Tenants
- Tenant-Aware Load Balancing using WSO2 ELB

Multi Tenant Architecture

The multi tenant architecture of WSO2 products allows you to deploy Web applications, Web services, ESB mediators, mashups etc. in an environment that supports the following:

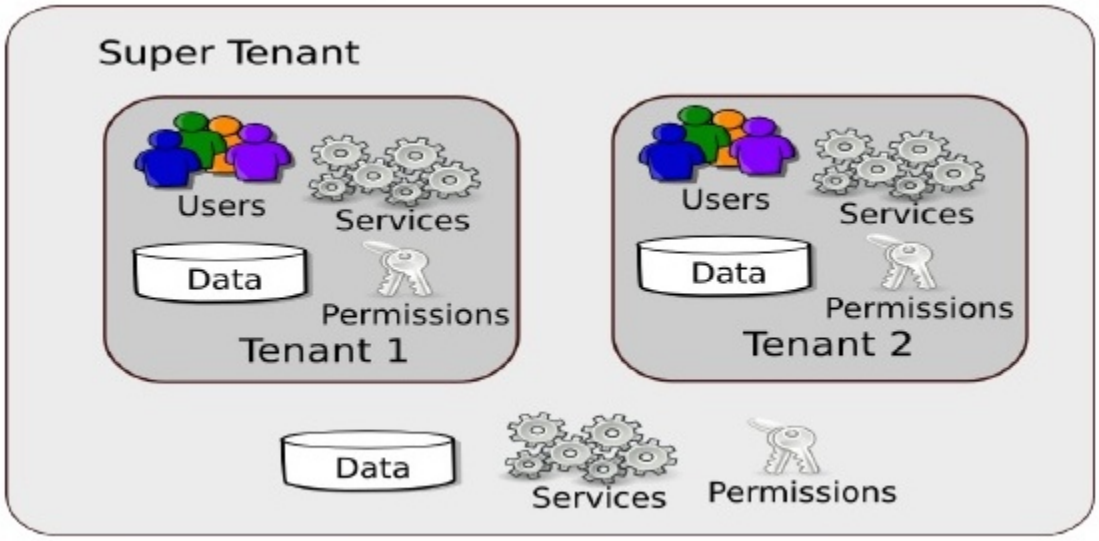
- **Tenant isolation:** Each tenant has its own domain, which the other tenants cannot access.
- **Data isolation:** Each tenant can manage its data securely, in an isolated manner.
- **Execution isolation:** Each tenant can carry out business processes and workflows independent of the other tenants. No action of a tenant is triggered or inhibited by another tenant.
- **Performance Isolation:** No tenant has an impact on the performance of another tenant.

Architecture

The super tenant is the complete server space of a WSO2 product instance. Separate spaces within this server space are allocated to individual tenants.

The super tenant as well as each individual tenant has its own configuration and context module.

Each tenant has its own security domain. A domain has a set of users, and permissions for those users to access resources. Thus, a tenant is restricted by the users and permissions of the domain assigned to it. The artifact repositories of the tenants are separated from each other.



An individual tenant can carry out the following activities within the boundaries of its own configuration and context module:

- Deploying artifacts
- Applying security
- User management
- Data management
- Request throttling
- Response caching

WSO2 Carbon provides a number of Admin services which have special privileges to manage the server. These admin services are deployed in the super tenant. Other tenants can make use of these admin services to manage their deployment. The admin services operate in a tenant aware fashion. Thus, privileges and restrictions that apply to any client using an admin service are taken into account.

Resource sharing

WSO2 Carbon supports the following methods for sharing resources among tenants:

- **Private Jet mode**: This method allows the load of a tenant ID to be deployed in a single tenant mode. A single tenant is allocated an entire service cluster. The purpose of this approach is to allow special privileges (such as priority processing and improved performance) to a tenant.
- **Separation at hardware level**: This method allows different tenants to share a common set of resources, but each tenant has to run its own operating system. This approach helps to achieve a high level of isolation, but it also incurs a high overhead cost.
- **Separation at JVM level**: This method allows tenants to share the same operating system. This is done by enabling each tenant to run a separate JVM instance in the operating system.
- **Native multitenancy**: This method involves allowing all the tenants to share a single JVM instance. This method minimises the overhead cost.

Lazy loading

Lazy loading is a design pattern used specifically in cloud deployments to prolong the initialization of an object or artifact until it is requested by a tenant or an internal process.

Tenants

Lazy loading of tenants is a feature that is built into all WSO2 products. This feature ensures that all the tenants are not loaded at the time the server starts in an environment with multiple tenants. Instead, they are loaded only when a request is made to a particular tenant. If a tenant is not utilized for a certain period of time (30 minutes by default), it will be unloaded from the memory.

You can change the default time period allowed for tenant inactiveness by adding `-Dtenant.idle.time=<time_in_minutes>` java property to the startup scrip of the product (`./wso2server.sh` file for Linux and `wso2server.bat` for Windows) as shown below.

```
JAVA_OPTS \
    -Dtenant.idle.time=30 \
```

Artifacts

Lazy loading of artifacts is a feature that is used by some WSO2 products, which can be enabled via the Carbon server configuration file (`carbon.xml`). The deployer that handles lazy loading of artifacts is called the `GhostDeployer`. A flag to enable or disable the `Ghost Deployer` is shown below. This is set to `false` by default because the `Ghost Deployer` works only with the HTTP/S transports. Therefore, if other transports are used, the `Ghost Deployer` does not have to be enabled.

```
<GhostDeployment>
   <Enabled>false</Enabled>
   <PartialUpdate>false</PartialUpdate>
</GhostDeployment>
```

When a stand-alone WSO2 product instance is started with lazy loading enabled, its services, applications and other artifacts are not deployed immediately. They are first loaded in the Ghost form and the actual artifact is deployed only when a request for the artifact is made. In addition, if an artifact has not been utilized for a certain period of time, it will be unloaded from the memory.

When lazy loading of artifacts is enabled for PaaS deployments, lazy loading applies both for tenants as well as a tenant artifacts. As a result, lazy loading is applicable on both levels for a tenant in a cloud environment. Therefore, the associated performance improvements and resource utilization efficiencies are optimal.

Restrictions

The following restrictions are imposed to ensure that each individual tenant has the required level of isolation and maintains fine grained security control over its own services without affecting the other tenants.

- Only the super tenant can modify its own configuration. In addition, it can add, view and delete tenants.
- When a tenant logs into the system, it can only access artifacts deployed under its own configuration. One tenant cannot manipulate the code of another tenant.
- The super admin or tenant admin can add user stores to their own domain. Dynamic configurations are possible only for secondary user stores and the primary user store is not configurable at run time. This is because primary user stores are available for all tenants and allowing changes to the configuration at run time can lead to instability of the system. Therefore, the primary user store is treated as a static property in the implementation and it should be configured prior to run time.
- A tenant's code cannot invoke sensitive server side functionality. This is achieved via Java security.
- Tenants share the transports provided by the system. They are not allowed to create their own transports.

Request dispatching

This section describes how the multi tenancy architecture described above works in a request dispatching scenario.

When a Carbon server receives a request, the message is first received by the handlers and dispatchers defined for the server configuration (i.e. super tenant). The server configuration may include handlers that implement cross tenant policies and Service Level Agreement (SLA) management. For example, a priority based dispatcher can be applied at this stage to offer differentiated qualities of service to different clients. Once the relevant handlers and dispatchers are applied, the request is sent to the tenant to which it is addressed. Then the message dispatchers and handlers specific to that tenant will be applied. See Viewing Handlers in Message Flows for further information on message handlers and dispatchers.

The following example further illustrates how message dispatching is carried out in a multi tenant server.

For example, two tenants named foo.com and bar.com may deploy a service named MyService. When this service is hosted on the two tenants, they would have the following URLs.

```
http://example.com/t/foo.com/services/MyService
http://example.com/t/bar.com/services/MyService
```

The name of the tenant in the URL allows the tenant to be identified when the Carbon server receives a message which is addressed to a specific client. Alternatively, you may configure a CNAME record in DNS (Domain Name System) as an alias for this information.

If a request is addressed to the `MyService` service hosted by `foo.com`, the message handlers and dispatchers of the super tenant will be applied and the tenant `foo.com` will be identified by the tenant name in the URL. Then the request will be sent to `foo.com` where it will be processed.

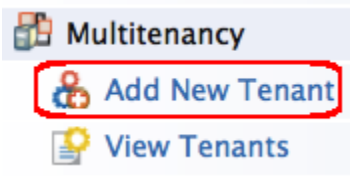
Scaling

The multi tenancy architecture described above mainly refers to a scenario where a single instance of a Carbon server acts as a single multi tenant node. In a situation where a very high load of requests are handles, you may need multiple multi tenant nodes. In order to operate with multiple multi tenant nodes, you need load balancing. The load balancer you use also needs to be tenant-aware. See Tenant Aware Load Balancing Using the WSO2 Elastic Load Balancer for further information.

Managing Tenants

You can add a new tenant in the management console and then view it by following the procedure below. In order to add a new tenant, you should be logged in as a super user.

1. Click **Add New Tenant** in the **Configure** tab of your product's management console.

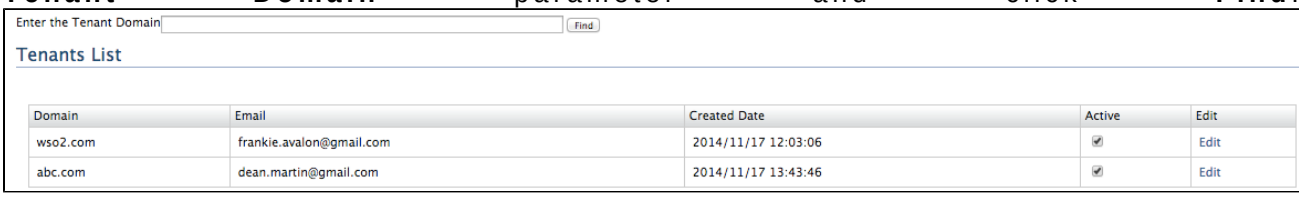


2. Enter the tenant information in **Register A New Organization** screen as follows, and click **Save**.

Parameter Name	Description
Domain	The domain name for the organization, which should be unique (e.g., abc.com). This is used as a unique identifier for your domain. You can use it to log into the admin console to be redirected to your specific tenant. The domain is also used in URLs to distinguish one tenant from another.
Select Usage Plan for Tenant	The usage plan defines limitations (such as number of users, bandwidth etc.) for the tenant.
First Name/Last Name	The name of the tenant admin.
Admin Username	The login username of the tenant admin. The username always ends with the domain name (e.g., admin@abc.com)
Admin Password	The password used to log in using the admin username specified.

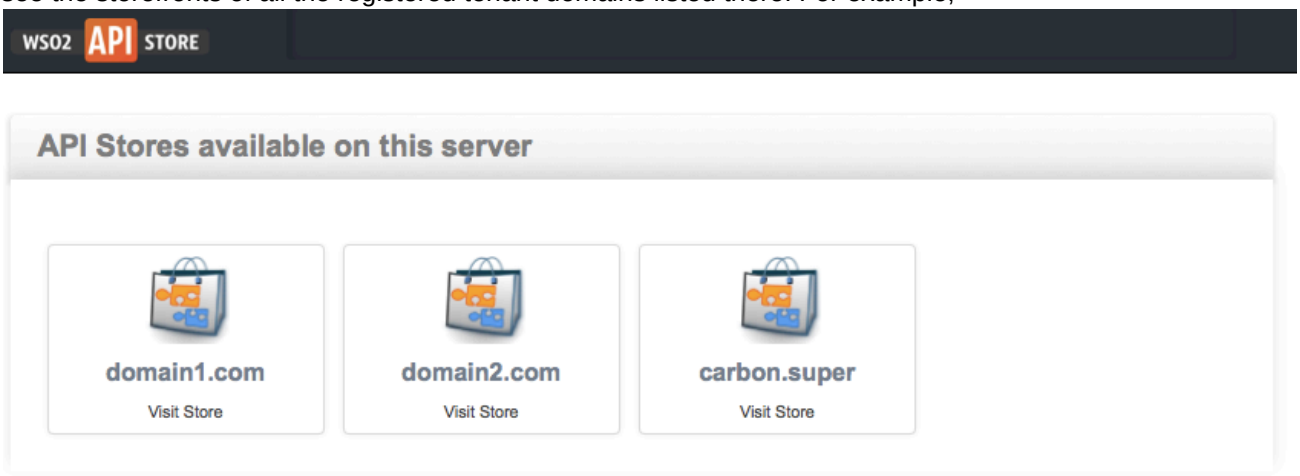
Admin Password (Repeat)	Repeat the password to confirm.
Email	The email address of the admin.

3. After saving, the newly added tenant appears in the **Tenants List** page as shown below. Click **View Tenants** in the **Configure** tab of the management console to see information of all the tenants that currently exist in the system. If you want to view only tenants of a specific domain, enter the domain name in the **Enter the Tenant Domain** parameter and click **Find**.



When you create multiple tenants in an API Manager deployment, the API Stores of each tenant are displayed in a muti-tenanted view for all users to browse and permitted users to subscribe to as shown below:

1. Access the API Store URL (by default, `https://localhost:9443/store`) using a Web browser. You see the storefronts of all the registered tenant domains listed there. For example,



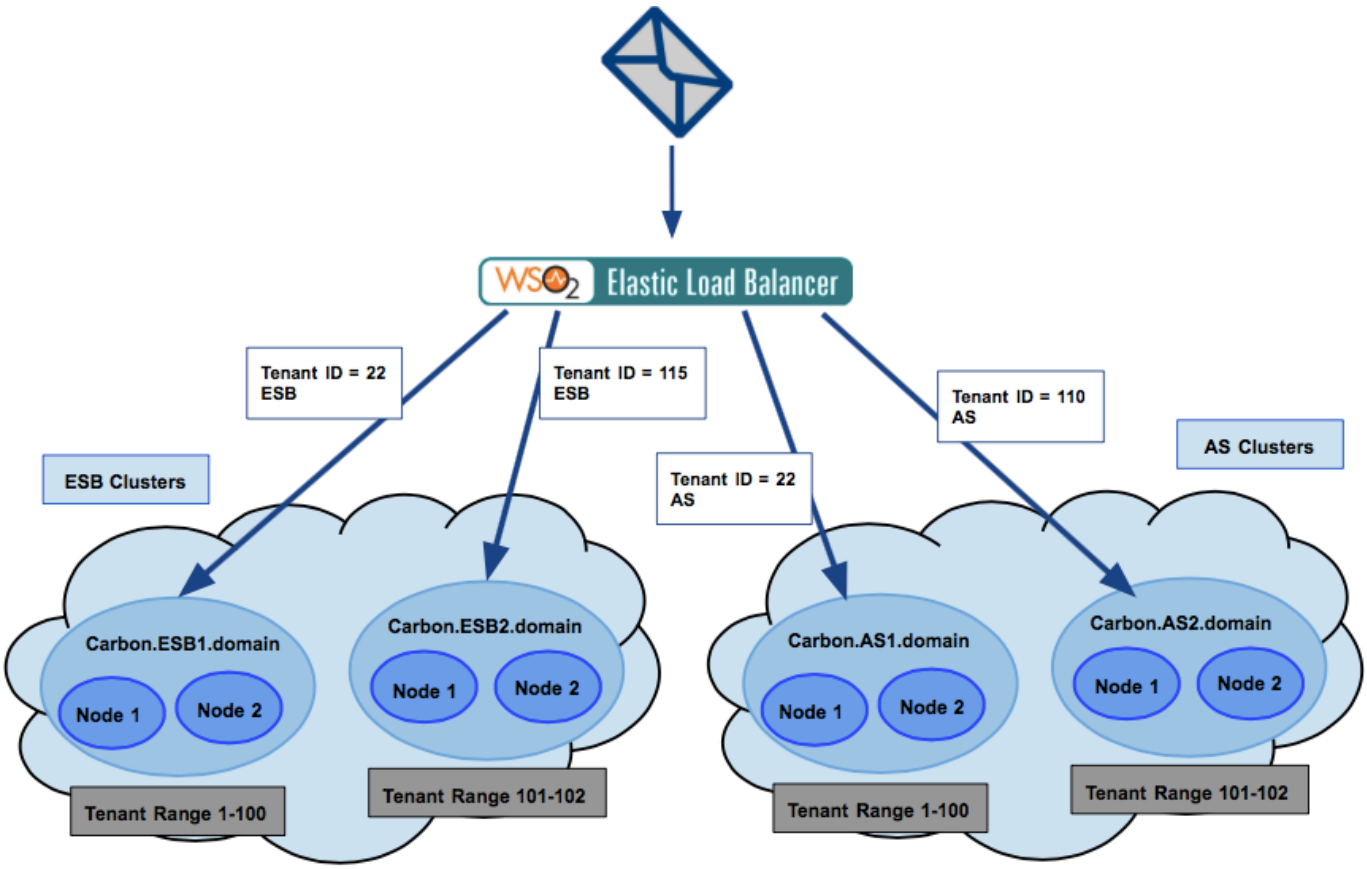
   This is called the public store. Each icon here is linked to the API Store of a registered tenant, including the super tenant, which is `carbon.super`. That is, the super tenant is also considered a tenant.
2. Click the **Visit Store** link associated with a given store to open it.
3. Anonymous users can browse all stores and all public APIs that are published to them. However, in order to subscribe to an API, the user must log in.

   For example, if you are a user in the `domain1.com` tenant domain,
   - You can access the public store (`https://localhost:9443/store`), go to the `domain1.com` store, log in to it and subscribe to its APIs.
   - You can also browse the other tenant stores listed in the public store. But, within other tenant stores, you can only subscribe to the APIs to which your tenant domain is permitted to subscribe to. At the time an API is created, the API creator can specify which tenants are allowed to subscribe to the API . For information, see API Subscriptions.

Tenant-Aware Load Balancing using WSO2 ELB

Tenant partitioning is required in a clustered deployment to be able to scale to large numbers of tenants. There can be multiple clusters for a single service and each cluster would have a subset of tenants as illustrated in the diagram below. In such situations, the load balancers need to be tenant aware in order to route the requests to the required tenant clusters. They also need to be service aware since it is the service clusters which are partitioned according to the clients.

The following example further illustrates how this is achieved in WSO2 Elastic Load Balancer (ELB).



A request sent to a load balancer has the following host header to identify the cluster domain:

`https://appserver.cloud-test. wso2.com/carbon.as1.domain/carbon/admin/login.jsp`

In this URL:

- `appserver.cloud-test.wso2.com` is the service domain which allows the load balancer to identify the service.
- `carbon.as1.domain.com` is the tenant domain which allows the load balancer to identify the tenant.

Services are configured with their cluster domains and tenant ranges in the in `ELB_HOME/repository/conf/loadbalancer.conf` file. These cluster domains and tenant ranges are picked by the load balancer when it loads.

The following is a sample configuration of the loadbalancer.conf file.

```
appserver {
# multiple hosts should be separated by a comma.
hosts appserver.cloud-test.wso2.com;

domains {
carbon.as1.domain {
tenant_range 1-100;
}
carbon.as2.domain {
tenant_range 101-200;
}
}
}
```

In the above configuration, there is a host address which maps to the application server service. If required, you can enter multiple host addresses separated by commas.

There are two cluster domains defined in the configuration. The cluster domain named `carbon.as1.domain` is used to load the range of tenants with IDs 1-100. The other cluster domain named `carbon.as2.domain` is used to load the tenants with IDs 101-200.

If the tenant ID of `abc.com` is 22, the request will be directed to the `Carbon.AS1.domain` cluster.

Samples

The WSO2 API Manager comes with a set of working samples that demonstrate some of its basic features and capabilities. The following topics provide information on executing these samples and evaluating their results.

- Setting up the Samples
- Deploying and Testing YouTube API
- Generating Billing Data
- Invoking APIs using a Web App Deployed in WSO2 AS
- Deploying and Testing Wikipedia API

Setting up the Samples

The API Manager binary distribution comes with a number of samples to demonstrate API Manager's basic functionality. These samples are located in `<APIM_HOME>/samples` folder. Inside this directory, there are sub directories for each sample. Each sub directory contains the relevant configurations, scripts and instructions required to run the a sample. Each sample contains an `APIPopulator` script, which drives the API Manager via a REST API.

The sections below describe the generic setup instructions and prerequisites to run API Manager samples:

- Prerequisites
- Setting up samples

> Executing these steps only once is enough to try multiple samples in a single API Manager installation.

Prerequisites

- Java Development Kit/JRE version 1.6.* or 1.7.*
- Apache Ant 1.6.x or later
- An HTTP client tool such as cURL (http://curl.haxx.se)
- A JavaScript compatible web browser
- An active Internet connection

Setting up samples

1. Download and install the API Manager according to the instructions given in Getting Started.
2. Before installing samples, you must configure libraries. Go to `<APIM_HOME>/bin` directory using a command prompt (on Windows) or text Linux console (on Linux) and type `ant`command. This step populates master data required for the API Manager to start up. For example, on Windows:

```
C:\wso2\wso2am-1.0.0\wso2am-1.0.0\bin>ant
Buildfile: C:\wso2\wso2am-1.0.0\wso2am-1.0.0\bin\build.xml

setup:
    [mkdir] Created dir: C:\wso2\wso2am-1.0.0\wso2am-1.0.0\repository\lib
     [copy] Copying 76 files to C:\wso2\wso2am-1.0.0\wso2am-1.0.0\repository\lib
    [mkdir] Created dir: C:\wso2\wso2am-1.0.0\wso2am-1.0.0\tmp\setup
    [unzip] Expanding: C:\wso2\wso2am-1.0.0\wso2am-1.0.0\repository\components\plugins\org.wso2.ca
so2am-1.0.0\tmp\setup
    [unzip] Expanding: C:\wso2\wso2am-1.0.0\wso2am-1.0.0\repository\components\plugins\org.wso2.ca
2am-1.0.0\tmp\setup
    [unzip] Expanding: C:\wso2\wso2am-1.0.0\wso2am-1.0.0\repository\components\plugins\org.wso2.ca
-1.0.0\tmp\setup
   [delete] Deleting directory C:\wso2\wso2am-1.0.0\wso2am-1.0.0\tmp\setup
    [unzip] Expanding: C:\wso2\wso2am-1.0.0\wso2am-1.0.0\repository\components\plugins\h2-database
1.0.0\repository\lib
    [unzip] Expanding: C:\wso2\wso2am-1.0.0\wso2am-1.0.0\repository\components\plugins\org.wso2.ca
repository\lib
     [copy] Copying 1 file to C:\wso2\wso2am-1.0.0\wso2am-1.0.0\repository\lib
     [move] Moving 39 files to C:\wso2\wso2am-1.0.0\wso2am-1.0.0\repository\lib
   [delete] Deleting directory C:\wso2\wso2am-1.0.0\wso2am-1.0.0\repository\lib\META-INF
   [delete] Deleting directory C:\wso2\wso2am-1.0.0\wso2am-1.0.0\repository\lib\org

BUILD SUCCESSFUL
Total time: 15 seconds
```

3. Start the API Manager by executing `<APIM_HOME>/bin/wso2server.bat` (on Windows) or `<APIM_HOME` Running the API Manager.`>/bin/wso2server.sh` (on Linux). For more information, see This step also populates more master data required for the server to start up.
4. Next, shut down the API Manager.

> It is a must to shut down the server before executing step 5 below.

5. Run the `ant` command inside `<APIM_HOME>/samples/Data` directory. An output similar to following appears:

```
C:\wso2\wso2am-1.0.0-Beta3\wso2am-1.0.0-Beta2\samples\Data>ant
Buildfile: C:\wso2\wso2am-1.0.0-Beta3\wso2am-1.0.0-Beta2\samples\Data\build.xml

init:

populate-user-database:
      [sql] Executing resource: C:\wso2\wso2am-1.0.0-Beta3\wso2am-1.0.0-Beta2\samples\Data\UserPopulator.sql
      [sql] 10 of 10 SQL statements executed successfully

BUILD SUCCESSFUL
Total time: 1 second
```

   It executes the `UserPopulator.sql`, which creates two user accounts as `provider1` and `subscriber1`. You can use them to log in to the API Publisher and API Store respectively.
6. Start the API Manager again and log in to the API Publisher (`http://localhost:9763/publisher`) using username/password as provider1/provider1. Similarly, log in to the API Store (`https://localhost:9443/store`) using username/password as subscriber1/subscriber1.

Next, proceed to executing the samples as described from the next section onwards.

Deploying and Testing YouTube API

- Introduction
- Prerequisites
- Building the Sample
- Executing the Sample

Introduction

This sample demonstrates how to subscribe to a published API and consume its functionality using the API Store Web application. The API used here provides YouTube feeds.

Prerequisites

Samples Setup1. Execute the steps in . When you are done, you will have the API Manager started and the relevant scripts run to create user accounts for API Publisher and API Store.

Building the Sample

1. First, we need to add an API in the API Publisher and publish it to the API Store. To do that, simply run the APIPopulator.sh (for Linux) or APIPopulator.bat (for Windows) files from folder, <AM_HOME>/samples/YoutubeFeeds.

2. The script will add an API to the API Publisher in Published state. This API can then be consumed by any user signed in to the API Store.

Executing the Sample

Subscribing to the API

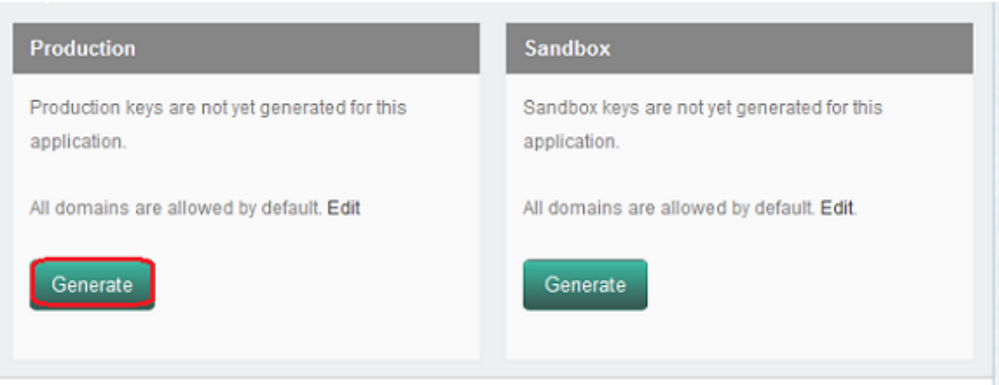
1. Log in to the API Store (https://localhost:9443/store) with credentials subscriber1/subscriber1.

2. Click the **APIs** tab at the top of the page and select the YoutubeFeeds API.

3. Next, subscribe to this API. Simply select the default application and throttling tier as **Gold**.

4. You will be asked to navigate to **My Subscriptions** tab.

5. Next, you can generate a key to the application. This key allows you to invoke APIs subscribed under a given application. Click on the **Generate** option to obtain an Application key. For example,



Invoking the API

6. Once you have obtained a key, you can invoke the API using a REST client of your choice. In this example, we use cURL (http://curl.haxx.se).

7. Copy and paste following into a new console window and execute it.

```
curl -H "Authorization :Bearer 9nEQnijLZ0Gi0gZ6a3pZICktVUca"
http://localhost:8280/youtube/1.0.0/most_popular
```

where, access token = `9nEQnijLZ0Gi0gZ6a3pZICktVUca`. Replace this value with the access token you generated through the API Store in step 5 above.

The access token is passed in the Authorization header as a value of "Bearer". The Authorization header of the message is prefixed by the string "Bearer". This is because, WSO2 API Manager enforces OAuth security on all the published APIs. Any consumer that talks to the API Manager should send their credential (application key) as per the OAuth bearer token profile. If you don't send an application key or send a wrong key, you will receive a 401 Unauthorized response in return.

8. You should be able to see results from YouTube on your console. For example,

```
<?xml version='1.0' encoding='UTF-8'?>
<feed xmlns='http://www.w3.org/2005/Atom' xmlns:app='http://purl.org/atom/app#'
xmlns:media='http://search.yahoo.com/mrss/'
xmlns:openSearch='http://a9.com/-/spec/opensearchrss/1.0/'
xmlns:gd='http://schemas.google.com/g/2005'
xmlns:yt='http://gdata.youtube.com/schemas/2007'>
<id>http://gdata.youtube.com/feeds/api/standardfeeds/most_popular</id>
<updated>2012-07-26T04:51:52.363-07:00</updated>
<category scheme='http://schemas.google.com/g/2005#kind'
term='http://gdata.youtube.com/schemas/2007#video'/>
<title type='text'>Most Popular</title>
<logo>http://www.youtube.com/img/pic_youtubelogo_123x63.gif</logo>
<link rel='alternate' type='text/html' href='http://www.youtube.com/browse?s=bzb'/>...
```

9. Access various other feeds in the YouTube API by changing the last segment of the invoked URL. For example,

```
curl -H "Authorization :Bearer 9nEQnijLZ0Gi0gZ6a3pZICktVUca"
http://localhost:8280/youtube/1.0.0/top_rated
curl -H "Authorization :Bearer 9nEQnijLZ0Gi0gZ6a3pZICktVUca"
http://localhost:8280/youtube/1.0.0/most_shared
curl -H "Authorization :Bearer 9nEQnijLZ0Gi0gZ6a3pZICktVUca"
http://localhost:8280/youtube/1.0.0/most_viewed
```

Replace `9nEQnijLZ0Gi0gZ6a3pZICktVUca` with the access token you generated through the API Store in step 5 above.

Generating Billing Data

- Introduction
- Prerequisites
- Building and running the sample

Introduction

This sample demonstrates how to setup WSO2 Business Activity Monitor (BAM) to collect and summarize runtime statistics from the WSO2 API Manager and generate bills for API consumers on usage.

Prerequisites

- Installation Prerequisites. Java Development Kit/JRE version 1.6.* or 1.7.*. Also see
- Download and install WSO2 BAM using the instructions given in BAM Installation Guide: docs.wso2.org/business-activity-monitor/Getting Started.

Building and running the sample

Configuring BAM

1. Open `<BAM_HOME>/repository/conf/carbon.xml` file where <BAM_HOME> is the BAM binary distribution folder that was downloaded as a prerequisite above. Change the carbon.xml file's port offset to 1. This is done to avoid any port conflicts of running two WSO2 Carbon instances in the same machine.

   ```
   <Offset>1</Offset>
   ```
2. Copy the `API_Manager_Analytics.tbox` in `<APIM_HOME>/samples/Billing` folder to `<BAM_HOME>/repository/deployment/server/bam-toolbox` folder. Create the `bam-toolbox` directory, if it doesn't exist already.

> ⚠ If you have copied `API_Manager_Analytics.tbox` to the `<BAM_HOME>/statistics` folder before, then you have to uninstall it first and install the new toolbox through the BAM Admin Console. Else, the Hive script used to summarize data on a monthly basis will not get executed.

The API Manager Analytic Toolbox : A toolbox is an installable archive, with a .tbox extension. It contains necessary artifacts that models a complete usecase, from collecting data, analyzing through defined Hive scripts to summarizing data through gadgets, Jaggery scripts and other dashboard components.

3. Connect the datasource to the database where the analytical data is stored using the `<BAM_HOME>/repository/conf/datasources/master-datasources.xml` file as follows. In the example, `WSO2AM_STATS_DB` is the datasource used to fetch the analytical data stored in an H2 database. If you want to use a different database, see Changing the statistics database.

```
<datasource>
    <name>WSO2AM_STATS_DB</name>
    <description>The datasource used for getting statistics to API
Manager</description>
    <jndiConfig>
          <name>jdbc/WSO2AM_STATS_DB</name>
    </jndiConfig>
    <definition type="RDBMS">
          <configuration>
               <!-- JDBC URL to query the database -->

<url>jdbc:h2:repository/database/APIMGTSTATS_DB;AUTO_SERVER=TRUE</url>
               <username>wso2carbon</username>
               <password>wso2carbon</password>
               <driverClassName>org.h2.Driver</driverClassName>
               <maxActive>50</maxActive>
               <maxWait>60000</maxWait>
               <testOnBorrow>true</testOnBorrow>
               <validationQuery>SELECT 1</validationQuery>
               <validationInterval>30000</validationInterval>
          </configuration>
    </definition>
</datasource>
```

4. Because you changed the default BAM port in step 2 above, you must change the Cassandra port given in JDBC connection url in the following datasource configuration found in master-datasources.xml file. Since the port offset is 1, the Cassandra port must be 9161.

   Default Ports of WSO2 ProductsFor a list of default ports used by WSO2 products, see .

```
<datasource>
           <name>WSO2BAM_CASSANDRA_DATASOURCE</name>
           <description>The datasource used for Cassandra data</description>
           <definition type="RDBMS">
               <configuration>
                   <url>jdbc:cassandra://localhost:9161/EVENT_KS</url>
                   <username>admin</username>
                   <password>admin</password>
               </configuration>
           </definition>
</datasource>
```

> If you run the Hive scripts before changing the default Cassandra port according to the BAM port offset, you keep getting an exception. To overcome this, add the following line at the beginning of the Hive script and rerun.
> drop table <hive_cassandra_table_name>;

5. Start WSO2 BAM server by running wso2server.bat (on Windows) and wso2server.sh (on Linux).

Configuring API Manager

1. To enable API statistics collection, configure the following properties in `<APIM_HOME>/repository/conf/api-manager.xml` file. Ensure that `<DataSourceName>` name is the same as JNDI config name in master-datasources.xml file in BAM.

```
<!-- Enable/Disable the API usage tracker. -->
<Enabled>true</Enabled>

<!-- JNDI name of the data source to be used for getting BAM statistics.This data
source should
 be defined in the master-datasources.xml file in conf/datasources directory. -->
<DataSourceName>jdbc/WSO2AM_STATS_DB</DataSourceName>

<!-- Enable/Disable Usage metering and billing for api usage -->
<EnableBillingAndUsage>true</EnableBillingAndUsage>
```

2. Configure the data source definition in `<APIM_HOME>/repository/conf/datasources/master-datasources.xml` file.

   Note: Replace <BAM_HOME> in the configuration below with the path to the actual BAM distribution location and the JNDI names must match the ones defined earlier in API Manager.

```
<datasource>
     <name>WSO2AM_STATS_DB</name>
     <description>The datasource used for getting statistics to API
Manager</description>
     <jndiConfig>
         <!-- This jndi name should be same as the DataSourceName defined in
api-manager.xml -->
         <name>jdbc/WSO2AM_STATS_DB</name>
     </jndiConfig>
     <definition type="RDBMS">
         <configuration>
             <!-- JDBC URL to query the database -->

<url>jdbc:h2:<BAM_HOME>/repository/database/APIMGTSTATS_DB;AUTO_SERVER=TRUE</url>
             <username>wso2carbon</username>
             <password>wso2carbon</password>
             <driverClassName>org.h2.Driver</driverClassName>
             <maxActive>50</maxActive>
             <maxWait>60000</maxWait>
             <testOnBorrow>true</testOnBorrow>
             <validationQuery>SELECT 1</validationQuery>
             <validationInterval>30000</validationInterval>
          </configuration>
      </definition>
</datasource>
```

3. Copy `<APIM_HOME>/samples/Billing/billing-conf.xml` file into `<APIM_HOME>/repository/conf` folder.

Viewing billing information

Once the above configurations are done, log in to API Store Web application (`https://<YourHostName>:9443/store`). You will see the menu items required for API Monetization in the menu bar at the top of the page.

> If you are a new user, there will not be any billing information at the beginning.

Invoking APIs using a Web App Deployed in WSO2 AS

Introduction

This sample demonstrates a pizza ordering scenario with backend services deployed in WSO2 Application Server (AS) to which we create APIs in WSO2 API Manager. Then, we invoke those APIs using a Web application deployed in WSO2 AS.

Prerequisites

Download and install WSO2 Application Server. For instructions, see Installation. Because you installed WSO2 AS on the same server as APIM, increment its default port to avoid port conflicts. To do this, go to `<AS_HOME>/repository/conf/carbon.xml` and change `<Offset>2</Offset>`.

Building the sample

1. Go to `<APIM_HOME>/samples/PizzaShack` in command shell and run `mvn clean install` to build the sample.
2. Go to `<APIM_HOME>/samples/PizzaShack/pizza-shack-web` in command shell and run `mvn clean install`.

> **If you are rebuilding this sample after building it at least once before**, execute the following steps instead of the above:
>
> 1. Remove the following module from `<APIM_HOME>/samples/PizzaShack/pom.xml` file: `<module>pre-processor</module>`.
> 2. Delete the `PizzaShack.zip` file from `<APIM_HOME>/samples/PizzaShack`.
> 3. Go to `<APIM_HOME>/samples/PizzaShack` in command shell and run `mvn clean install`.

Executing the sample

1. Log in to the API Publisher (https://localhost:9443/publisher) and create the following APIs.

```
Delivery API

        API Name= pizzaShack
        Context = /pizzashack/delivery
        Version = 1.0.0
        Production Endpoint
URL=http://localhost:9765/pizzashack-api-1.0.0/api/delivery
        API Resources =Keep the default values

   Order API

        API Name= pizzashack-order
        Context = /pizzashack/order
        Version = 1.0.0
        Production Endpoint
URL=http://localhost:9765/pizzashack-api-1.0.0/api/order
        API Resources =Keep the default values

   Menu API

        API Name= pizzashack-menu
        Context = /pizzashack/menu
        Version = 1.0.0
        Production Endpoint
URL=http://localhost:9765/pizzashack-api-1.0.0/api/menu
        API Resources =Keep the default values
```

2. Navigate to the **Lifecycles** tab of each API and promote them to **PUBLISHED** state. This will push the APIs to the Gateway and they will be available for subscription in the API Store.
3. Log in to the API Store (https://localhost:9443/store) and click on each API created earlier. Next, subscribe to each of them using the default application.
4. After subscription, a message appears. Choose **Go to My Subscriptions**.
5. The **Subscriptions** page opens. Create a production key by clicking the **Generate** button associated with it. You also have the option to increase the default token validity period, which is 1 hour.
6. You get the access token, a consumer key and a consumer secret. Replace the consumer key and secret pair in `<APIM_HOME>/samples/PizzaShack/pizza-shack-web/src/main/webapp/WEB-INF/web.xml` with the newly generated ones. For example,

```
<context-param>
   <param-name>consumerKey</param-name>
   <param-value>szsHscDYLeKUcwA1GhPARQlflusa</param-value>
</context-param>
<context-param>
   <param-name>consumerSecret</param-name>
   <param-value>wJEfRDE3JeFnGMuwVNseNzsXM1sa</param-value>
</context-param>
```

   You now have three APIs subscribed under an application and an access token to the application. Next, we deploy a Web application in the Application Server and use it to invoke the APIs.
7. Start WSO2 AS (https://localhost:9445/console) and log into its management console. For instructions, see A S documentation (If the AS documentation link doesn't load, please clear your browser cache and retry).
8. Deploy the following into the Application Server.

   - `<APIM_HOME>/samples/PizzaShack/pizza-shack-web/target/pizzashack.war`
   - `<APIM_HOME>/samples/PizzaShack/pizza-shack-api/target/pizzashack-api-1.0.0.`

war

9. After deploying, access the application using `http://localhost:9765/pizzashack`. It opens the application in a Web browser.
10. You can use this application to order pizza. Internally, the APIs get invoked when you use the application.

Deploying and Testing Wikipedia API

- Introduction
- Building the Sample
- Executing the Sample

Introduction

This sample demonstrates how to subscribe to a published API and consume its functionality using the API Store Web application. We use the Wikipedia API here.

Building the Sample

Samples Setup. Execute the steps in When you are done, you will have the API Manager started and the relevant scripts run to create user accounts for API Publisher and API Store.

> ⚠ The scripts used for this sample do not work in Windows. Support for Windows will be added in an upcoming release.

Executing the Sample

1. If you haven't done so already, start the API Manager and log in to the API Publisher (`http://localhost:9763/publisher`) using credentials provider1/provider1.
2. There are no APIs created yet. To create one, run `<APIM_HOME>/samples/WikipediaAPI/APIPopulator.sh` (on Linux) or `<APIM_HOME>/samples/WikipediaAPI/APIPopulator.bat` (on Windows).
3. Refresh the API Publisher to see the Wikipedia API created.
4. Click on the API, go to its **Lifecycles** tab and publish the API by selecting its life cycle stage as PUBLISHED.
5. You can now access Wikipedia through this newly-deployed API. Log in to the API Store (`http://localhost:9763/store`) using credentials subscriber1/subscriber1.
6. Select the **Applications** tab at the top of the page, and create a new application. Provide any name you like.
7. Select the **APIs** tab at the top of the page, select the `WikipediaAPI` API and subscribe to it using the newly-created application.
8. Go to the **My Subscriptions** tab and select your application. Click the **Generate** button associated with the production system to obtain an application access token.
9. You are now ready to invoke the API. Copy and paste following into a new console and execute it. Be sure to replace the string '9nEQnijLZ0Gi0gZ6a3pZICktVUca' with the application access token you obtained earlier.

```
curl -H "Authorization :Bearer 9nEQnijLZ0Gi0gZ6a3pZICktVUca"
"http://10.100.5.20:8280/wikipedia/1.0.0?format=json&action=query&titles=MainPage
&prop=revisions&rvprop=content"
```

10. You must see the JSON result from the Wikipedia API on you console. For example,

```
{"query":{"pages":{"5982813":{"pageid":5982813,"ns":0,"title":"MainPage","revisio
ns":[{"contentformat":"text/x-wiki","contentmodel":"wikitext", "*":"#Redirect
[[Main Page]]\n\n{{Redr|mod|rcc}}"}]}}}}...
```

See http://www.mediawiki.org/wiki/API:Main_page for more information about the Wikipedia API. You can try out various API actions and features similar to step 9.

Published APIs

The following topics discuss the APIs exposed from the API Publisher and API Store Web applications using which you can create and manage APIs. You can consume APIs directly through their UIs or an external REST client like cURL or the WSO2 REST client. The Token APIs exposed in API Manager are also described here.

- Publisher APIs
- Store APIs
- Token API
- WSO2 Admin Services

Publisher APIs

Publisher APIs provide the following REST resources.
[Login] [Logout] [Add/Update API] [Get All APIs] [Get an API] [Remove an API] [Copy an API] [Check Older Version] [Change API Status] [Add/Update an API Document] [Remove an API Document]

> **Note**: When you access any API other than the login and logout APIs through an external REST client such as cURL, first invoke the login API to ensure that user is authenticated. When the login API is invoked, the system stores the generated session cookie in a file, which we use in the next API invocations.
>
> Alternatively, if you access these APIs from the API Publisher application itself, you do not have to invoke the login API first.

Login

Description	Log in to API Publisher web application.
URI	http://localhost:9763/publisher/site/blocks/user/login/ajax/login .jag
URI Parameters	action=login&username=xxx&password=xxx
HTTP Methods	POST
Example	curl -X POST -c cookies http://localhost:9763/publisher/site/blocks/user/login/ajax/login.jag -d 'action=login&username=admin&password=admin'

Logout

Description	Log out from API Publisher web application.
URI	http://localhost:9763/publisher/site/blocks/user/login/ajax/login .jag
URI Parameters	?action=logout
HTTP Methods	GET
Example	curl -b cookies http://localhost:9763/publisher/site/blocks/user/login/ajax/login.jag?action=logout

Add/Update API

Description	Add a new API or update an existing API.
URI	http://localhost:9763/publisher/site/blocks/item-add/ajax/add.jag

URI Parameters	**A d d** "action=addAPI&name=xxx&visibility=public&version=x.x.x&description=xxx&endpointType=nonsecu -d'endpoint_config={"production_endpoints":{"url":"<URL>","conf **Update API**: "action=updateAPI&visibility=public& thumbUrl=<File> &description=xxx&tags=x,y,z& e -d'endpoint_config={"production_endpoints":{"url":"<URL>","config":null},"endpoint_ type":"http"}'; **Tip**: If you want to set only the HTTP transport, leave the `https_checked` parameter empty a **Tip**: To add a thumbnail image, create a file object of that thumbnail and pass it with the `thumb`
HTTP Methods	POST
Example	**A d d** curl -X POST -b cookies http://10.100.1.71:9763/publisher/site/blocks/item-add/ajax/add.jag -d "actio -thumbnail-youtube_logo.jpg&context=/youtube&tiersCollection=Gold&resourceCount=0&resourceMe **U p d a t e** curl -X POST -b cookies http://10.100.1.71:9763/publisher/site/blocks/item-add/ajax/add.jag -d "actior ext=/youtube&tiersCollection=Gold&resourceCount=0&resourceMethod-0=GET&resourceMethodAut From APIM 1.6.0 onwards, this service accepts endpoint configuration data as a JSON value. I Add the argument `subscriptions=all_tenants` to enable subscription to this API by all te curl -X POST -b cookies http://localhost:9763/publisher/site/blocks/item-add/ajax/add.jag -d "ac es/pf-thumbnail-youtube_logo.jpg&context=/youtube&tiersCollection=Gold&resourceCount=0& point_type":"http"}';

Get All APIs

Description	Lists all the created APIs.
URI	http://localhost:9763/publisher/site/blocks/listing/ajax/item-list.jag
URI Parameters	?action=getAllAPIs
HTTP Methods	GET
Example	curl -b cookies http://localhost:9763/publisher/site/blocks/listing/ajax/item-list .jag ?action=getAllAPIs

Get an API

Description	Get details of a specific API.
URI	http://localhost:9763/publisher/site/blocks/listing/ajax/item-list.jag
URI Parameters	action=getAPI&name=xxx&version=xxx&provider=xxx
HTTP Methods	POST

Example	curl -X POST -b cookies http://localhost:9763/publisher/site/blocks/listing/ajax/item-list.jag -d "action=getAPI&name=API1&version=1.0.0&provider=user1"

Remove an API

Description	Remove an API.
URI	http://localhost:9763/publisher/site/blocks/item-add/ajax/remove.jag
URI Parameters	action=removeAPI&name=xxx&version=xxx&provider=xxx
HTTP Methods	POST
Example	curl -X POST -b cookies http://localhost:9763/publisher/site/blocks/item-add/ajax/remove.jag -d "action=removeAPI&name=API1&version=1.0.0&provider=user1"

Copy an API

Description	Copy an API to a newer version.
URI	http://localhost:9763/publisher/site/blocks/overview/ajax/overvi ew.jag
URI Parameters	action=createNewAPI&provider=xxx&apiName=xxx&version=xxx&newVersion=xxx
HTTP Methods	POST
Example	curl -X POST -b cookies http://localhost:9763/publisher/site/blocks/overview/ajax/overview.jag -d "action=createNewAPI&provider=user1&apiName=API1&version=1.0.0&newVersion=2.0.0&isDefault

Check Older Version

Description	Does older version of API exist.
URI	http://localhost:9763/publisher/site/blocks/life-cycles/ajax/life-cycles .jag
URI Parameters	?action=isAPIOlderVersionExist&provider=xxx&name=xxx&version=xxx
HTTP Methods	GET
Example	curl -X POST -b cookies http://localhost:9763/publisher/site/blocks/life-cycles/ajax/life-cycles .jag ?action=isAPIOlderVersion Exist&provider=user1&name=API1&version=1.0.0

Change API Status

Description	Change the API's status.
URI	http://localhost:9763/publisher/site/blocks/life-cycles/ajax/life-cycles .jag
URI Parameters	action=updateStatus&name=xxx&version=1.0.0&provider=apiCreateName&status=PUBLISHED&pub
HTTP Methods	POST

Example	curl -X POST -b cookies 'http://localhost:9763/publisher/site/blocks/life-cycles/ajax/life-cycles.jag' -d 'action=updateStatus&name=TwitterAPI&version=1.0.0&provider=provider&status=PUBLISHED&pub

Add/Update an API Document

Description	Add a new API document.
URI	http://localhost:9763/publisher/site/blocks/documentation/ajax/docs.jag
URI Parameters	**Add** action=addDocumentation&mode=Add&provider=xxx&apiName=xxx&version=xxx&docName=xxx&d Note that docVisibility is applicable only if you have enabled it. See API documentation visibility. **Update Document**: action=addDocumentation&mode=Update&provider=xxx&apiName=xxx&version
HTTP Methods	POST
Example	**Add Document**: curl -X POST -b cook "action=addDocumentation&provider=admin&apiName=api1&version=1.0.0&docName=test&docType **Update Document**: curl -X POST -b cookies http://localhost:9763/publisher/site/blocks/documentatio &docName=test&docType=how to&sourceType=inline&docUrl=&summary=new summary&docLocati

Remove an API Document

Description	Remove an API document.
URI	http://localhost:9763/publisher/site/blocks/documentation/ajax/docs.jag
URI Parameters	action=removeDocumentation&provider=xxx&apiName=xxx&version=xxx&docName=xxx&docType=
HTTP Methods	POST
Example	curl -X POST -b cookies http://localhost:9763/publisher/site/blocks/documentation/ajax/docs.jag -d 'action=removeDocume ntation&provider=admin&apiName=API1&version=1.0.0&docName=doc1&d Type=How To'

Store APIs

Store APIs provide the following REST resources.
[Login] [Logout] [User Signup] [Get all Paginated Published APIs] [Get Published APIs by Application] [Add an Application] [Update an Application] [Get Applications] [Remove an Application] [Add a Subscription] [List Subscriptions] [Remove a Subscription] [Add an API Comment]

> **Note**: When you access any API other than the login and logout APIs through an external REST client such as cURL, first invoke the login API to ensure that user is authenticated. When the login API is invoked, the system stores the generated session cookie in a file, which we use in the next API invocations.
>
> Alternatively, if you access these APIs from the API Store application itself, you do not have to invoke the login API first.

Login

Description	Log in to API Store.

URI	http://localhost:9763/store/site/blocks/user/login/ajax/login.jag
URI Parameters	action=login&username=xxx&password=xxx
HTTP Methods	POST
Example	curl -X POST -c cookies http://localhost:9763/store/site/blocks/user/login/ajax/login.jag -d 'action=login&username=user1&password=xxx'

Logout

Description	Log out from API Store.
URI	http://localhost:9763/store/site/blocks/user/login/ajax/login.jag?action=logout
URI Parameters	?action=logout
HTTP Methods	GET
Example	curl -b cookies http://localhost:9763/publisher/site/blocks/user/login/ajax/login.jag?action=logout

User Signup

Description	Add a new API Consumer.
URI	http://localhost:9763/store/site/blocks/user/sign-up/ajax/user-add.jag
URI Parameters	action=addUser&username=xxx&password=xxx&allFieldsValues=firstname|lastname|email
HTTP Methods	POST
Example	curl -X POST -b cookies http://localhost:9763/store/site/blocks/user/sign-up/ajax/user-add.jag -d “action=addUser&username=user2&password=xxx&allFieldsValues=firstname|lastname|email”

Get all Paginated Published APIs

Description	Get a list of all published APIs in paginated form so that browsing is easier.
URI	http://localhost:9763/store/site/blocks/api/listing/ajax/list.jag
URI Parameters	action=*getAllPaginatedPublishedAPIs*, tenant, start, end The `start` and `end` parameters determine from which API to which you want to retrieve. For example, if start=1 and end=10, the first 10 APIs that appear on the API Store will be returned.
HTTP Methods	GET
Example	To get the first 5 APIs: curl -b cookies "http://localhost:9763/store/site/blocks/api/listing/ajax/list.jag?action=getAllPaginatedPublishedAPIs&tenant=carbon.super&start=1&end=5"

Please note that the `getAllPublishedAPIs` API is now deprecated. You can get the same functionality from `getAllPaginatedPublishedAPIs`.

Get Published APIs by Application

Description	Get a list of published APIs filtered by the subscribed Application. Login API needs be called prior to calling this API.
URI	http://localhost:9763/store/site/blocks/subscription/subscription-list/ajax/subscription-list.jag
URI Parameters	action=getSubscriptionByApplication&app=App1
HTTP Methods	GET
Example	curl -b cookies 'http://localhost:9763/store/site/blocks/subscription/subscription-list/ajax/subscription-list.jag ?action=getSubscriptionByApplication&app=App1 '

Add an Application

Description	Add a new application.
URI	http://localhost:9763/store/site/blocks/application/application-add/ajax/application-add.jag
URI Parameters	action=addApplication&application=xxx&tier=xxx&description=xxx&callbackUrl
HTTP Methods	POST
Example	curl -X POST -b cookies http://localhost:9763/store/site/blocks/application/application-add/ajax/application-add.jag -d 'action=addApplication&application=app1&tier=Unlimited&description=&callbackUrl='

Update an Application

Description	Update an existing application.
URI	http://localhost:9763/store/site/blocks/application/application-update/ajax/application-update.jag
URI Parameters	action=updateApplication&applicationOld=xxx&applicationNew=xxx&callbackUrlNew=xxx&descriptior
HTTP Methods	POST
Example	curl -X POST -b cookies http://localhost:9763/store/site/blocks/application/application-update/ajax/ag -d 'action=updateApplication&applicationOld=app1&applicationNew=app2&tier=Unlimited&descriptionNe

Get Applications

Description	Get list of applications.
URI	http://localhost:9763/store/site/blocks/application/application-list/ajax/application-list.jag
URI Parameters	?action=getApplications
HTTP Methods	GET

Example	curl -b cookies http://localhost:9763/store/site/blocks/application/application-list/ajax/application-list.jag ?action=getApplications

Remove an Application

Description	Remove an existing application.
URI	http://localhost:9763/store/site/blocks/application/application-remove/ajax/application-remove.jag
URI Parameters	action=removeApplication&application=xxx
HTTP Methods	POST
Example	curl -X POST -b cookies http://localhost:9763/store/site/blocks/application/application-remove/ajax/application-remove.jag -d "action=removeApplication&application=app2"

Add a Subscription

Description	Add a new API subscription.
URI	http://localhost:9763/store/site/blocks/subscription/subscription-add/ajax/subscription-add.jag
URI Parameters	• To add a subscription by application ID: action=**addSubscription**&name=xxx&version=xxx&provider=xxx&**applicationId**=xxx • To add a subscription by application name: action=**addAPISubscription**&name=xxx&version=xxx=xxx&tier=xxx&**applicationName**=xxx
HTTP Methods	POST
Example	• curl -X POST -b cookies http://localhost:9763/store/site/blocks/subscription/subscription-add/ajax/on-add.jag -d 'action=addSubscription&name=API1&version=1.0.0&provider=user1&tier=gold&applicationId=1' • curl -X POST -b cookies http://localhost:9763/store/site/blocks/subscription/subscription-add/ajax/on-add.jag -d 'action=addAPISubscription&name=API1&version=1.0.0&provider=user1&tier=gold&applicationN

List Subscriptions

Description	List all API subscriptions.
URI	http://localhost:9763/store/site/blocks/subscription/subscription-list/ajax/subscription-list.jag
URI Parameters	action=getAllSubscriptions
HTTP Methods	GET
Example	curl -b cookies http://localhost:9763/store/site/blocks/subscription/subscription-list/ajax/subscription-list.jag?action=getAllSubscriptions

Remove a Subscription

Description	Remove an API subscription.

URI	http://localhost:9763/store/site/blocks/subscription/subscription-remove/ajax/subscription-remove.jag
URI Parameters	action=removeSubscription&name=xxx&version=xxx&provider=xxx&applicationId=xxx
HTTP Methods	POST
Example	curl -X POST -b cookies http://localhost:9763/store/site/blocks/subscription/subscription-remove/ajax/subscription-remove.jag -d 'action=removeSubscription&name=API1&version=1.0.0&provider=user1&applicationId=1'

Add an API Comment

Description	Add a comment for an API.
URI	http://localhost:9763/store/site/blocks/comment/comment-add/ajax/comment-add.jag
URI Parameters	action=addComment&name=xxx&version=xxx&provider=xxx&comment=xxx
HTTP Methods	POST
Example	curl -X POST -b cookies http://localhost:9763/store/site/blocks/comment/comment-add/ajax/comment-add.jag -d 'action=addComment&name=API1&version=1.0.0&provider=user1&comment=Hello'

Token API

Users need access tokens to invoke APIs subscribed under an application. Access tokens are passed in the HTTP header when invoking APIs. The API Manager provides a Token API that you can use to generate and renew user and application access tokens. The response of the Token API is a JSON message. You extract the token from the JSON and pass it with an HTTP Authorization header to access the API.

Let's take a look at how to generate/renew access tokens and authorize them. WSO2 API Manager supports the four most common authorization grant types and you can also define additional types such as SAML.

- Generating access tokens with user credentials (password grant type)
- Generating access tokens with authorization code (authorization code grant type)
- Exchanging SAML2 bearer tokens with OAuth2 (SAML extension grant type)
- Renewing access tokens
- Revoking access tokens

Generating access tokens with user credentials (password grant type)

You can obtain an access token by providing the resource owner's username and password as an authorization grant. It requires the base64 encoded string of the `consumer-key:consumer-secret` combination. You need to meet the following prerequisites before using the Token API to generate a token.

Prerequisites

- Signing up to API Store. A valid user account in the API Store. See
- A valid consumer key and consumer secret pair. Initially, these keys must be generated through the management console by clicking the **Generate** link on **My Subscriptions** page. You can find more details in Working with Access Tokens.
- A running API Gateway instance (typically an API Manager instance should be running). For instructions on API Gateway, see Architecture.
- If you have multiple Carbon servers (such as API Manager and WSO2 Application Server) running on the same computer, you must change the port offset to avoid port conflicts. Setting the port offset causes API

Manager to run on a different port from the default. Therefore, when you change the port offset, you must also update the port for the endpoint defined inside the Send mediator of the token API in `<APIM_HOME>/repository/deployment/server/synapse-configs/default/api/_TokenAPI_.xml`. For example, if you set the port offset to 1, and the default port is 9443, you must change the port in the endpoint to 9444, as follows:

```
<send>
  <endpoint>
    <address uri="https://localhost:9444/oauth2/token"/>
  </endpoint>
</send>
```

If you have upgraded from a previous release of API Manager, you should also update the endpoint in the deprecated API file `_LoginAPI_.xml`

- If the Key Manager is running on a different server from the API Gateway instance, change the host and port of the token API endpoint (see above) to the correct address of the Key Manager.

Invoking the Token API to generate tokens

1. Combine the consumer key and consumer secret keys in the format **consumer-key:consumer-secret** and encode the combined string using base64. Encoding to base64 can be done using the URL: http://base64encode.org.
   Here's an example consumer key and secret combination : `wU62DjlyDBnq87GlBwplfqvmAbAa:ksdSdoefDDP7wpaElfqvmjDue.`
2. Access the Token API by using a REST client such as the WSO2 REST Client or Curl, with the following parameters.
   - Assuming that both the client and the API Gateway are run on the same server, the token API url is https://localhost:8243/token
   - payload - `"grant_type=password&username=<username>&password=<password>&scope=<scope>"`. Replace the `<username>` and `<password>` values as appropriate. <scope> is optional, you can leave it off if necessary
   - headers - `Authorization: Basic <base64 encoded string>, Content-Type: application/x-www-form-urlencoded`. Replace the `<base64 encoded string>` as appropriate.

   For example, use the following cURL command to access the Token API. It generates two tokens as an access token and a refresh token. You can use the refresh token at the time a token is renewed .

```
curl -k -d "grant_type=password&username=<username>&password=<password>" -H
"Authorization: Basic
SVpzSWk2SERiQjVlOFZLZFpBblVpX2ZaM2Y4YTpHbTBiSjZvV1Y4ZkM1T1FMTGxDNmpzbEFDVzhh,
Content-Type: application/x-www-form-urlencoded" https://localhost:8243/token
```

CuRL command with Scopes

```
curl -k -d
"grant_type=password&username=<username>&password=<password>&scope=<scope1>
<scope2>" -H "Authorization: Basic
SVpzSWk2SERiQjVlOFZLZFpBblVpX2ZaM2Y4YTpHbTBiSjZvV1Y4ZkM1T1FMTGxDNmpzbEFDVzhh,
Content-Type: application/x-www-form-urlencoded" https://localhost:8243/token
```

A note about scopes

When defining an API, the API creator is able to specify a scope for an API Resource. This is so that the API Resource can only be accessed through a token that had been issued for at least the scope belonging to the API Resource. For example if a Resource had been defined for a scope named 'update' and if the token had been issued for the scopes 'read' and 'update', the token will be allowed to access the resource. If the token had been issued for a scope named 'read', the request bearing the particular token will be blocked.

The Token API endpoint is specified in `<APIM_HOME>/repository/deployment/server/synapse-configs/default/api/_TokenAPI_.xml` file. When running the server on a different port from the default (i.e., 9443), or if your Key Manager is running on a different machine from your API Gateway, you must update the endpoint inside the `_TokenAPI_.xml` file as described in the prerequisites.

User access tokens have a fixed expiration time, which is set to 60 minutes by default. Before deploying the API manager to users, extend the default expiration time by editing the `<AccessTokenDefaultValidityPeriod>` tag in `<PRODUCT_HOME>/repository/conf/identity.xml`.

When a user access token expires, the user can try regenerating the token as explained in the Renew user tokens section.

Instead of using the Token API, you can generate access tokens from the API Store UI. See Working with Access Tokens for information.

Generating access tokens with authorization code (authorization code grant type)

Instead of requesting authorization directly from the resource owner (resource owner's credentials), in this grant type, the client directs the resource owner to an authorization server. The authorization server works as an intermediary between the client and resource owner to issues an authorization code, authenticate the resource owner and obtain authorization. As this is a redirection-based flow, the client must be capable of interacting with the resource owner's user-agent (typically a Web browser) and receiving incoming requests (via redirection) from the authorization server.

The client initiates the flow by directing the resource owner's user-agent to the authorization endpoint (you can use the `/authorize` endpoint for the authorization code grant type of OAuth2.0). It includes the client identifier, response_type, requested scope, and a redirection URI to which the authorization server sends the user-agent back after granting access. The authorization server authenticates the resource owner (via the user-agent) and establishes whether the resource owner granted or denied the client's access request. Assuming the resource owner grants access, the authorization server then redirects the user-agent back to the client using the redirection URI provided earlier. The redirection URI includes an authorization code.

The client then requests an access token from the authorization server's `/token` endpoint by including the authorization code received in the previous step. When making the request, the client authenticates with the authorization server. It then includes the redirection URI used to obtain the authorization code for verification. The authorization server authenticates the client, validates the authorization code, and ensures that the redirection URI matches the URI used to redirect the client from the /authorize endpoint in the previous response. If valid, the authorization server responds back with an access token and, optionally, a refresh token.

Invoking the Token API to generate tokens

Assuming that both the client and the API Gateway are run on the same server, the Authorization API url is `https://localhost:8243/authorize`.

- query component - `response_type=code&client_id=<consumer_key>&scope=PRODUCTION&redirect_uri=<application_callback_url>`

- headers - `Content-Type: application/x-www-form-urlencoded`

For example, the client directs the user-agent to make the following HTTP request using TLS.

```
GET
/authorize?response_type=code&client_id=wU62DjlyDBnq87GlBwplfqvmAbAa&scope=PRODUCTION&
redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
HTTP/1.1
Host: server.example.com
Content-Type:
application/x-www-form-urlencoded
```

The authorization server redirects the user-agent by sending the following HTTP response:

```
HTTP/1.1 302 Found
Location:
https://client.example.com/cb?code=SplxlOBeZQQYbYS6WxSbIA
```

Now the client makes the following HTTP request using TLS to the /token endpoint.

```
POST /token HTTP/1.1
Host: server.example.com
Authorization: Basic
SVpzSWk2SERiQjVlOFZLZFpBblVpX2ZaM2Y4YTpHbTBiSjZvV1Y4ZkM1T1FMTGxDNmpzbEFDVzhh
Content-Type:
application/x-www-form-urlencoded
grant_type=authorization_code&code=SplxlOBeZQQYbYS6WxSbIA&redirect_uri=https%3A%2F%2Fc
lient%2Eexample%2Ecom%2Fcb
```

The /token endpoint responds in the same way like in password grant type.

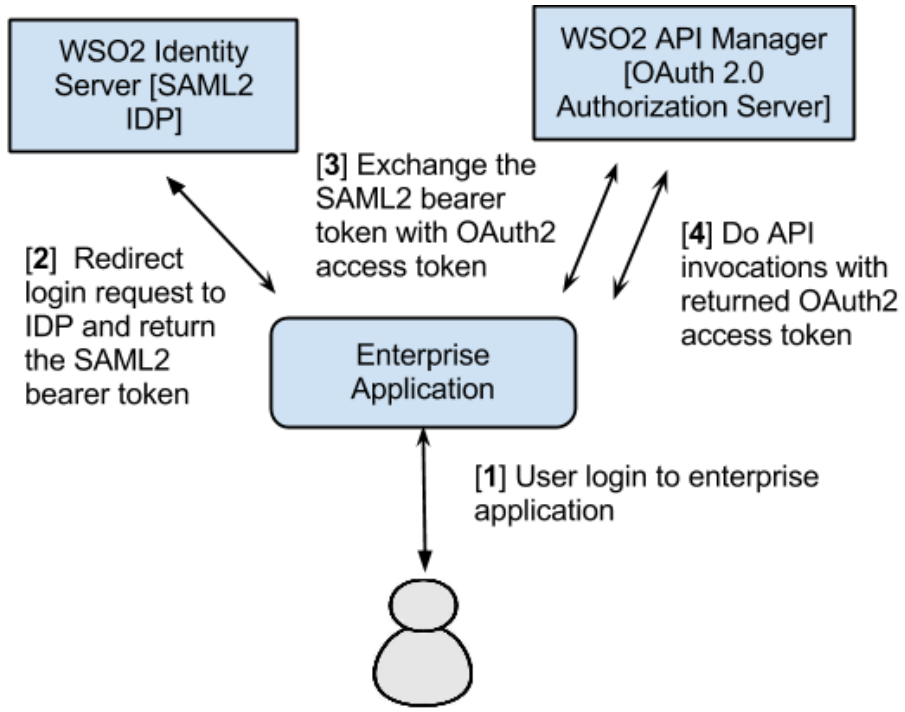
Exchanging SAML2 bearer tokens with OAuth2 (SAML extension grant type)

SAML 2.0 is an XML-based protocol. It uses security tokens containing assertions to pass information about an enduser between a SAML authority and a SAML consumer. A SAML authority is an identity provider (IDP) and a SAML consumer is a service provider (SP).

A lot of enterprise applications use SAML2 to engage a third-party identity provider to grant access to systems that are only authenticated against the enterprise application. These enterprise applications might need to consume OAuth-protected resources through APIs, after validating them against an OAuth2.0 authentication server. However, an enterprise application that already has a working SAML2.0 based SSO infrastructure between itself and the IDP prefers to use the existing trust relationship, even if the OAuth authorization server is entirely different from the IDP. The SAML2 Bearer Assertion Profile for OAuth2.0 helps leverage this existing trust relationship by presenting the SAML2.0 token to the authorization server and exchanging it to an OAuth2.0 access token.

WSO2 API Manager provides SAML2 Bearer Assertion Profile Support with the OAuth 2.0 feature. **WSO2 Identity Server (version 5.0.0** is used here but you can use any version from 4.5.0 onwards) or any other SAML2 Identity provider can act as an identity service provider for the systems enabled with SSO. WSO2 API Manager acts as the OAuth authorization server. This way, an enterprise application can exchange the SAML2.0 bearer token that it retrieves when authenticating against an IDP (e.g., WSO2 Identity Server) with an OAuth2.0 access token from an OAuth authorization server (e.g., WSO2 API Manager). It can then use the OAuth2 token in API invocations.

The diagram below depicts this scenario:



The scenarios of the above diagram are explained below:

Scenario [1]: User initiates login call to an enterprise application **.**

Scenario [2]:

- As the application is a SAML SP, it redirects the user to the SAML2.0 IDP to log in.
- The user provides credentials at the IDP and is redirected back to the SP with a SAML2.0 token signed by the IDP.
- The SP verifies the token and logs the user to the application.
- The SAML 2.0 token is stored in the user's session by the SP.

Scenario [3]:

- The enterprise application (SP) wants to access an OAuth2 protected API resource through WSO2 API Manager.
- The application makes a request to the API Manager to exchange the SAML2 bearer token for an OAuth2.0 access token.
- The API Manager validates the assertion and returns the access token.

Scenario [4]: User does API invocations through the API Manager by setting it as an Authorization header with the returned OAuth2 access token.

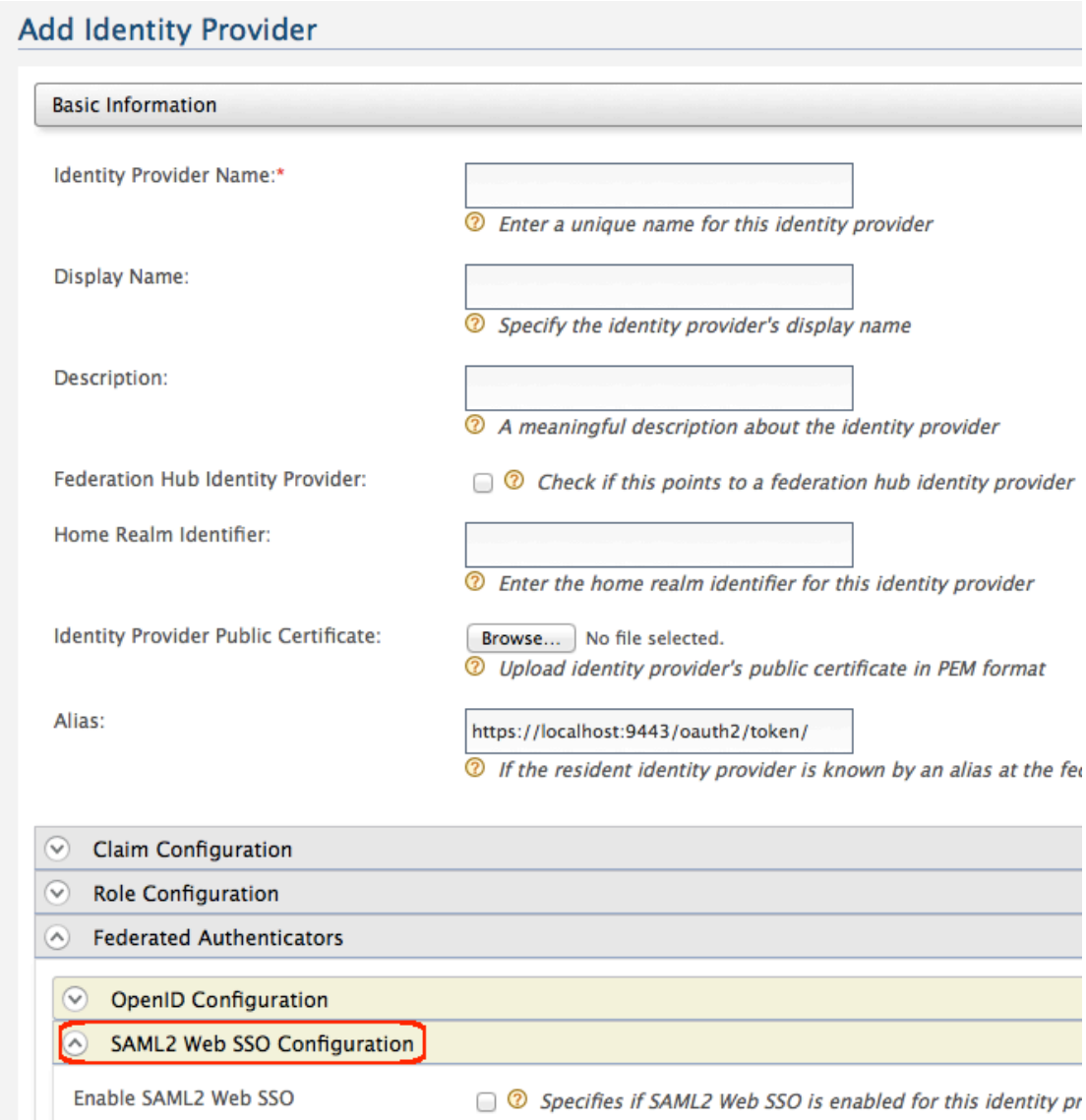
Let's see how to configure the token exchange.

Prerequisites

- A signed SAML2 token (encoded assertion value), which you retrieve when authenticating against a SAML2 IDP. With the authentication request, you must pass attributes such as SAML2 issuer name, token endpoint and the restricted audience. To specify those attributes,

    1. Log in to the management console (https://localhost:9443/carbon) using admin/admin credentials and select **Add** under **Identity Providers** menu in the **Main** menu.

2. Provide the following values in the page that opens:
   - Under **Basic Information**
     - **Identity Provider Name:** Enter a unique name for IDP
     - **Identity Provider Public Certificate:** Upload Identity Provider public certificate
     - **Alias**: Give the name of the alias if the Identity Provider identifies this token endpoint by an alias
   - Under **Federated Authenticators -> SAML2 Web SSO Configuration**
     - **Identity Provider Entity Id**: The SAML2 issuer name specified when generating assertion token, which contains the unique identifier of the IDP
     - **Service Provider Entity Id**:
     - **SSO URL**: Enter the IDP's SAML2 Web SSO URL value



- Signing up to API Store.A valid user account in the API Store. See
- A valid consumer key and consumer secret. Initially, these keys must be generated through the management console by clicking the **Generate** link on **My Subscriptions** page. For more information, see Working with Access Tokens.
- A running API Gateway instance. See information on API Gateway in Architecture.
- If you have multiple Carbon servers (such as WSO2 API Manager and WSO2 Application Server) running on the same machine, you must change the port offset and update the token API endpoint. Additionally, if the key server is on a different server from the API Gateway, you must update the token API endpoint to use the correct host and port. For more information, see this prerequisite in the previous section.

Invoking Token API to generate tokens

Follow the steps below to invoke Token API to generate access tokens from SAML2 assertions.

1. Combine the consumer key and consumer secret keys as `consumer-key:consumer-secret` and encode the combined string using base64 using http://base64encode.org. Here's an example consumer key and secret combination: `wU62DjlyDBnq87GlBwplfqvmAbAa:ksdSdoefDDP7wpaElfqvmjDue`.
2. Access the Token API using a REST client such as the WSO2 REST Client or Curl. The parameters are explained below:
   - Assuming that both the client and the API Gateway run on the same server, the Token API URL is https://localhost:8243/token.
   - payload - `"grant_type=urn:ietf:params:oauth:grant-type:saml2-bearer&assertion=<SAML2_Encoded_Assertion_Token> &scope=PRODUCTION"`. Replace the `<SAML2_Encoded_Assertion_Token>` value as appropriate.
   - headers - `Authorization :Basic <base64 encoded string>, Content-Type: application/x-www-form-urlencoded`. Replace the `<base64 encoded string>` as appropriate.

   For example, use the following cURL command used to access the Token API generates an access token and a refresh token. You can use the refresh token at the time a token is renewed.

```
curl -k -d
"grant_type=urn:ietf:params:oauth:grant-type:saml2-bearer&assertion=<SAML2_Encode
d Assertion>&scope=PRODUCTION" -H "Authorization: Basic
SVpzSWk2SERiQjVlOFZLZFpBblVpX2ZaM2Y4YTpHbTBiSjZvV1Y4ZkM1T1FMTGxDNmpzbEFDVzhh,
Content-Type: application/x-www-form-urlencoded" https://localhost:8243/token
```

> The Token API endpoint is specified in `<APIM_HOME>/repository/deployment/server/synapse-configs/default/api/_TokenAPI_.xml` file. When running the server on a different port from the default (i.e., 9443), or if your Key Manager is running on a different server from your API Gateway, you must update the endpoint inside the `_TokenAPI_.xml` file as described here.

Renewing access tokens

After an access token is generated, sometimes you might have to renew the old token due to expiration or security concerns. You can renew an access token using a refresh token, by issuing a REST call to the Token API with the following parameters.

- The Token API URL is https://localhost:8243/token, assuming that both the client and the Gateway are run on the same server.
- payload: `"grant_type=refresh_token&refresh_token=<retoken>&scope=PRODUCTION"`. Replace the `<retoken>` value with the refresh token generated in the previous section.
- headers: `Authorization :Basic <base64 encoded string>, Content-Type: application/x-www-form-urlencoded`. Replace `<base64 encoded string>` as appropriate.

For example, the following cURL command can be used to access the Token API.

```
curl -k -d "grant_type=refresh_token&refresh_token=<retoken>&scope=PRODUCTION" -H
"Authorization: Basic
SVpzSWk2SERiQjVlOFZLZFpBblVpX2ZaM2Y4YTpHbTBiSjZvV1Y4ZkM1T1FMTGxDNmpzbEFDVzhh,
Content-Type: application/x-www-form-urlencoded" https://localhost:8243/token
```

The above REST message grants you a renewed access token along with a refresh token, which you can use the next time you renew the access token. A refresh token can be used only once. At the moment, a refresh token never expires, but we will provide a way to configure an expiration time in a future release.

Revoking access tokens

After issuing an access token, a user or an admin can revoke it in case of theft or a security violation. You can do this by calling Revoke API using a utility like cURL. The Revoke API's endpoint URL is http://localhost:8280/revoke.

Parameters required to invoke this API are as follows:

- The token to be revoked
- Consumer key and consumer secret key. Must be encoded using Base64 algorithm

For example, `curl -k -d "token=<ACCESS_TOKEN_TO_BE_REVOKED>" -H "Authorization: Basic Base64Encoded(Consumer key:consumer secret)" http://localhost:8280/revoke`

> When the API Gateway cache is enabled (it is enabled by default), even after revoking a token, it might still be available in the cache to consumers until the cache expires in approximately 15 minutes. You can clear the cache manually by restarting the server.

WSO2 Admin Services

WSO2 products are managed internally using SOAP Web services known as **admin services**. WSO2 products come with a management console UI, which communicates with these admin services to facilitate administration capabilities through the UI.

A service in WSO2 products is defined by the following components:

- Service component: provides the actual service
- UI component: provides the Web user interface to the service
- Service stub: provides the interface to invoke the service generated from the service WSDL

There can be instances where you want to call back-end Web services directly. For example, in test automation, to minimize the overhead of having to change automation scripts whenever a UI change happens, developers prefer to call the underlying services in scripts. The topics below explain how to discover and invoke these services from your applications.

Discovering the admin services

By default, the WSDLs of admin services are hidden from consumers. Given below is how to discover them.

1. Set the `<HideAdminServiceWSDLs>` element to false in the `<PRODUCT_HOME>/repository/conf/carbon.xml` file.
2. Restart the server.
3. Start the WSO2 product with the `-DosgiConsole` option, such as `sh <PRODUCT_HOME>/bin/wso2server.sh -DosgiConsole` in Linux.
4. When the server is started, hit the enter/return key several times to get the OSGI shell in the console.
5. In the OSGI shell, type: `osgi> listAdminServices`
6. The list of admin services of your product are listed. For example:

```
osgi> listAdminServices
Admin services deployed on this server:
1. ProvisioningAdminService, ProvisioningAdminService, https://192.168.219.1:8243/services/ProvisioningAdminService
2. SynapseApplicationAdmin, SynapseApplicationAdmin, https://192.168.219.1:8243/services/SynapseApplicationAdmin
3. CarbonAppUploader, CarbonAppUploader, https://192.168.219.1:8243/services/CarbonAppUploader
4. OperationAdmin, OperationAdmin, https://192.168.219.1:8243/services/OperationAdmin
5. SequenceAdminService, SequenceAdminService, https://192.168.219.1:8243/services/SequenceAdminService
6. MediationLibraryAdminService, MediationLibraryAdminService, https://192.168.219.1:8243/services/MediationLibraryAdminService
7. StatisticsAdmin, StatisticsAdmin, https://192.168.219.1:8243/services/StatisticsAdmin
8. LoggedUserInfoAdmin, LoggedUserInfoAdmin, https://192.168.219.1:8243/services/LoggedUserInfoAdmin
9. MediationStatisticsAdmin, MediationStatisticsAdmin, https://192.168.219.1:8243/services/MediationStatisticsAdmin
10. TopicManagerAdminService, TopicManagerAdminService, https://192.168.219.1:8243/services/TopicManagerAdminService
11. MessageProcessorAdminService, MessageProcessorAdminService, https://192.168.219.1:8243/services/MessageProcessorAdminService
12. ApplicationAdmin, ApplicationAdmin, https://192.168.219.1:8243/services/ApplicationAdmin
13. NDataSourceAdmin, NDataSourceAdmin, https://192.168.219.1:8243/services/NDataSourceAdmin
14. ServiceGroupAdmin, ServiceGroupAdmin, https://192.168.219.1:8243/services/ServiceGroupAdmin
15. ClassMediatorAdmin, ClassMediatorAdmin, https://192.168.219.1:8243/services/ClassMediatorAdmin
```

7. To see the service contract of an admin service, select the admin service's URL and then paste it in your

browser with **?wsdl** at the end. For example: `https://localhost:9443/services/UserAdmin?wsdl`

> In products like WSO2 ESB and WSO2 API Manager, the port is 8243 (assuming 0 port offset). However, you should be accessing the Admin Services via the management console port, which is 9443 when there is no port offset.

8. Note that the admin service's URL appears as follows in the list you discovered in step 6:

```
AuthenticationAdmin, AuthenticationAdmin, https://<host
IP>:8243/services/AuthenticationAdmin
```

Invoking an admin service

Admin services are secured using common types of security protocols such as HTTP basic authentication, WS-Security username token, and session based authentication to prevent anonymous invocations. For example, the `UserAdmin` Web service is secured with the HTTP basic authentication. To invoke a service, you do the following:

1. Authenticate yourself and get the session cookie.
2. Generate the client stubs to access the back-end Web services.

> To generate the stubs, you can write your own client program using the Axis2 client API or use an existing tool like SoapUI (4.5.1 or later) or wsdl2java.
>
> The wsdl2java tool, which comes with WSO2 products by default hides all the complexity and presents you with a proxy to the back-end service. The stub generation happens during the project build process within the Maven POM files. It uses the Maven ant run plug-in to execute the wsdl2java tool.
>
> You can also use the Java client program given here to invoke admin services. All dependency JAR files that you need to run this client are found in the `/lib` directory.

Authenticate the user

The example code below authenticates the user and gets the session cookie:

```
import org.apache.axis2.AxisFault;
 import org.apache.axis2.transport.http.HTTPConstants;
 import org.wso2.carbon.authenticator.stub.AuthenticationAdminStub;
 import org.wso2.carbon.authenticator.stub.LoginAuthenticationExceptionException;
 import org.wso2.carbon.authenticator.stub.LogoutAuthenticationExceptionException;
 import org.apache.axis2.context.ServiceContext;
 import java.rmi.RemoteException;

 public class LoginAdminServiceClient {
   private final String serviceName = "AuthenticationAdmin";
     private AuthenticationAdminStub authenticationAdminStub;
     private String endPoint;

     public LoginAdminServiceClient(String backEndUrl) throws AxisFault {
       this.endPoint = backEndUrl + "/services/" + serviceName;
       authenticationAdminStub = new AuthenticationAdminStub(endPoint);
     }

     public String authenticate(String userName, String password) throws
RemoteException,
                                        LoginAuthenticationExceptionException {

       String sessionCookie = null;

       if (authenticationAdminStub.login(userName, password, "localhost")) {
         System.out.println("Login Successful");

         ServiceContext serviceContext = authenticationAdminStub.
             _getServiceClient().getLastOperationContext().getServiceContext();
         sessionCookie = (String)
serviceContext.getProperty(HTTPConstants.COOKIE_STRING);
         System.out.println(sessionCookie);
       }

       return sessionCookie;
     }

     public void logOut() throws RemoteException,
LogoutAuthenticationExceptionException {
       authenticationAdminStub.logout();
     }
 }
```

To resolve dependency issues, if any, add the following dependency JARs location to the class path: `<PRODUCT_HOME>/repository/components/plugins`.

The the `AuthenticationAdminStub` class requires `org.apache.axis2.context.ConfigurationContext` as a parameter. You can give a null value there.

Generate the client stubs

After authenticating the user, give the retrieved admin cookie with the service endpoint URL as shown in the sample below. The service management service name is ServiceAdmin. You can find its URL (e.g., `https://localhost:9443/services/ServiceAdmin`) in the `service.xml` file in the `META-INF` folder in the respective bundle that you find in `<PRODUCT_HOME>/repository/components/plugins`.

```
import org.apache.axis2.AxisFault;
 import org.apache.axis2.client.Options;
 import org.apache.axis2.client.ServiceClient;
 import org.wso2.carbon.service.mgt.stub.ServiceAdminStub;
 import org.wso2.carbon.service.mgt.stub.types.carbon.ServiceMetaDataWrapper;
 import java.rmi.RemoteException;

 public class ServiceAdminClient {
   private final String serviceName = "ServiceAdmin";
   private ServiceAdminStub serviceAdminStub;
   private String endPoint;

   public ServiceAdminClient(String backEndUrl, String sessionCookie) throws AxisFault
{
     this.endPoint = backEndUrl + "/services/" + serviceName;
     serviceAdminStub = new ServiceAdminStub(endPoint);
     //Authenticate Your stub from sessionCooke
     ServiceClient serviceClient;
     Options option;

     serviceClient = serviceAdminStub._getServiceClient();
     option = serviceClient.getOptions();
     option.setManageSession(true);
     option.setProperty(org.apache.axis2.transport.http.HTTPConstants.COOKIE_STRING,
sessionCookie);
   }

   public void deleteService(String[] serviceGroup) throws RemoteException {
     serviceAdminStub.deleteServiceGroups(serviceGroup);

   }

   public ServiceMetaDataWrapper listServices() throws RemoteException {
     return serviceAdminStub.listServices("ALL", "*", 0);
   }
 }
```

The following sample code lists the back-end Web services:

```
import org.wso2.carbon.authenticator.stub.LoginAuthenticationExceptionException;
 import org.wso2.carbon.authenticator.stub.LogoutAuthenticationExceptionException;
 import org.wso2.carbon.service.mgt.stub.types.carbon.ServiceMetaData;
 import org.wso2.carbon.service.mgt.stub.types.carbon.ServiceMetaDataWrapper;

 import java.rmi.RemoteException;

 public class ListServices {
   public static void main(String[] args)
       throws RemoteException, LoginAuthenticationExceptionException,
           LogoutAuthenticationExceptionException {
     System.setProperty("javax.net.ssl.trustStore",
"$ESB_HOME/repository/resources/security/wso2carbon.jks");
     System.setProperty("javax.net.ssl.trustStorePassword", "wso2carbon");
     System.setProperty("javax.net.ssl.trustStoreType", "JKS");
     String backEndUrl = "https://localhost:9443";

     LoginAdminServiceClient login = new LoginAdminServiceClient(backEndUrl);
     String session = login.authenticate("admin", "admin");
     ServiceAdminClient serviceAdminClient = new ServiceAdminClient(backEndUrl,
session);
     ServiceMetaDataWrapper serviceList = serviceAdminClient.listServices();
     System.out.println("Service Names:");
     for (ServiceMetaData serviceData : serviceList.getServices()) {
       System.out.println(serviceData.getName());
     }

     login.logOut();
   }
 }
```

Reference Guide

The following topics provide reference information for working with WSO2 API Manager:

- Default Ports of WSO2 Products
- WSO2 Patch Application Process
- Error Handling

Default Ports of WSO2 Products

This page describes the default ports that are used for each WSO2 product when the port offset is 0.

- Common ports
- Product-specific ports

Common ports

The following ports are common to all WSO2 products that provide the given feature. Some features are bundled in the WSO2 Carbon platform itself and therefore are available in all WSO2 products by default.

Management console ports

WSO2 products that provide a management console use the following servlet transport ports:

- 9443 - HTTPS servlet transport (the default URL of the management console is https://localhost:9443/carbon)
- 9763 - HTTP servlet transport

LDAP server ports

Provided by default in the WSO2 Carbon platform.

- 10389 - Used in WSO2 products that provide an embedded LDAP server

KDC ports

- 8000 - Used to expose the Kerberos key distribution center server

JMX monitoring ports

WSO2 Carbon platform uses TCP ports to monitor a running Carbon instance using a JMX client such as JConsole. By default, JMX is enabled in all products. You can disable it using `<PRODUCT_HOME>/repository/conf/etc/jmx.xml` file.

- 11111 - RMIRegistry port. Used to monitor Carbon remotely
- 9999 - RMIServer port. Used along with the RMIRegistry port when Carbon is monitored from a JMX client that is behind a firewall

Clustering ports

To cluster any running Carbon instance, either one of the following ports must be opened.

- 45564 - Opened if the membership scheme is multicast
- 4000 - Opened if the membership scheme is wka

Random ports

Certain ports are randomly opened during server startup. This is due to specific properties and configurations that become effective when the product is started. Note that the IDs of these random ports will change every time the server is started.

- A random TCP port will open at server startup because of the `-Dcom.sun.management.jmxremote` property set in the server startup script. This property is used for the JMX monitoring facility in JVM.
- A random UDP port is opened at server startup due to the log4j appender (`SyslogAppender`), which is

configured in the `<PRODUCT_HOME>/repository/conf/log4j.properties` file.

Product-specific ports

Some products open additional ports.

API Manager | BAM | BPS | Complex Event Processor | Elastic Load Balancer | ESB | Identity Server | Message Broker | Storage Server | Enterprise Mobility Manager

API Manager

- 10397 - Thrift client and server ports
- 8280, 8243 - NIO/PT transport ports
- 7711 - Thrift SSL port for secure transport, where the client is authenticated to BAM/CEP: stat pub

> If you change the default API Manager ports with a port offset, most of its ports will be changed automatically according to the offset except a few exceptions described in the APIM Manager documentation.

BAM

- 9160 - Cassandra port using which Thrift listens to clients
- 7711 - Thrift SSL port for secure transport, where the client is authenticated to BAM
- 7611 - Thrift TCP port to receive events from clients to BAM
- 21000 - Hive Thrift server starts on this port

BPS

- 2199 - RMI registry port (datasources provider port)

Complex Event Processor

- 9160 - Cassandra port on which Thrift listens to clients
- 7711 - Thrift SSL port for secure transport, where the client is authenticated to CEP
- 7611 - Thrift TCP port to receive events from clients to CEP
- 11224 - Thrift TCP port for HA management of CEP

Elastic Load Balancer

- 8280, 8243 - NIO/PT transport ports

ESB

Non-blocking HTTP/S transport ports: Used to accept message mediation requests. If you want to send a request to an API or a proxy service for example, you must use these ports. ESB_HOME}/repository/conf/axis2/axis2.xml file.

- 8243 - Passthrough or NIO HTTPS transport
- 8280 - Passthrough or NIO HTTP transport

Identity Server

- 8000 - KDCServerPort. Port which KDC (Kerberos Key Distribution Center) server runs
- 10500 - ThriftEntitlementReceivePort

Message Broker

Message Broker uses the following JMS ports to communicate with external clients over the JMS transport.

- 5672 - Port for listening for messages on TCP when the AMQP transport is used.

- 8672 - Port for listening for messages on TCP/SSL when the AMQP Transport is used.
- 1883 - Port for listening for messages on TCP when the MQTT transport is used.
- 8833 - Port for listening for messages on TCP/SSL when the MQTT Transport is used.
- 7611 - The port for Apache Thrift Server.

Storage Server

Cassandra:

- 7000 - For Inter node communication within cluster nodes
- 7001 - For inter node communication within cluster nodes vis SSL
- 9160 - For Thrift client connections
- 7199 - For JMX

HDFS:

- 54310 - Port used to connect to the default file system.
- 54311 - Port used by the MapRed job tracker
- 50470 - Name node secure HTTP server port
- 50475 - Data node secure HTTP server port
- 50010 - Data node server port for data transferring
- 50075 - Data node HTTP server port
- 50020 - Data node IPC server port

Enterprise Mobility Manager

The following ports need to be opened for Android and iOS devices, so that it can connect GCM (Google Cloud Message) and APNS (Apple Push Notification Service) and enroll to WSO2 EMM.

Android:
The ports to open are 5228, 5229 and 5230. GCM typically only uses 5228, but it sometimes uses 5229 and 5230. GCM does not provide specific IPs, so it is recommended to allow the firewall to accept outgoing connections to all IP addresses contained in the IP blocks listed in Google's ASN of 15169.

iOS:

- 5223 - TCP port used by devices to communicate to APNs servers
- 2195 - TCP port used to send notifications to APNs
- 2196 - TCP port used by the APNs feedback service
- 443 - TCP port used as a fallback on Wifi only when devices are unable to communicate to APNs on port 5223
  The APNs servers use load balancing. The devices will not always connect to the same public IP address for notifications. The entire 17.0.0.0/8 address block is assigned to Apple, so it is best to allow this range in the firewall settings.

API Manager:

> The following WSO2 API Manager ports are only applicable to WSO2 EMM 1.1.0 onwards.

- 10397 - Thrift client and server ports
- 8280, 8243 - NIO/PT transport ports

WSO2 Patch Application Process

You apply patches to WSO2 products either as individual patches or through a service pack. A service pack is recommended when the number of patches increase. The following sections explain the WSO2 patch application process:

- Applying service packs to the Kernel
- Applying individual patches to the Kernel
- Verifying the patch application
- Overview of the patch application process

Before you begin

- You can download all WSO2 Carbon Kernel patches from here.
- Before you apply a patch, check its `README.txt` file for any configuration changes required.

Applying service packs to the product

Carbon 4.2.0 Kernel supports service packs. A service pack is a collection of patches in a single pack. It contains two elements:

- The `lib` directory: contains all the JARs relevant to the service pack.
- The `servicepack_patches.txt` text file: contains the list of JARs in the service pack.

Follow the steps below to apply service packs to your product.

1. Copy the service pack file to the `<PRODUCT_HOME>/repository/components/servicepacks/` directory. For example, the image below shows how a new service pack named `servicepack001` is added to this directory.

```
servicepacks
└── servicepack001
    ├── lib
    │   ├── org.wso2.carbon.base_4.2.0.SNAPSHOT.jar
    │   ├── org.wso2.carbon.bridge_4.2.0.SNAPSHOT.jar
    │   ├── org.wso2.carbon.core.bootup.validator_4.2.0.SNAPSHOT.jar
    │   ├── org.wso2.carbon.core.common_4.2.0.SNAPSHOT.jar
    │   └── org.wso2.carbon.registry.api_4.2.0.SNAPSHOT.jar
    └── servicepack_patches.txt
```

2. Start your product. The following steps will be executed:
   a. Before applying any patches, the process first creates a backup folder named `patch0000` inside the `<PRODUCT_HOME>/repository/components/patches/` directory, which will contain the original content of the `<PRODUCT_HOME>/repository/components/plugins/` directory. This step enables you to revert back to the previous state if something goes wrong during operations.
   b. The latest service pack in the `<PRODUCT_HOME>/repository/components/servicepacks/` directory will be applied. That is, the patches in the service pack will be applied to the `<PRODUCT_HOME>/repository/components/plugins/` directory.
   c. In addition to the service pack, if there are individual patches added to the `<PRODUCT_HOME>/repository/components/patches/` directory, those will also be incrementally applied to the `plugins` directory.

   > The metadata file available in the service pack will maintain a list of the applied patches by service pack. Therefore, the metadata file information will be compared against the `<PRODUCT_HOME>/repository/components/patches/` directory, and only the patches that were not applied by the service pack will be incrementally applied to the `plugins` directory.

Applying individual patches to the product

You can apply each patch individually to your system as explained below. Alternatively, you can apply patches through service packs as explained above.

1. Copy the patches to the `<PRODUCT_HOME>/repository/components/patches/` directory.
2. Start the Carbon server. The patches will then be incrementally applied to the `plugins` directory.

> Before applying any patches, the process first creates a backup folder named `patch0000` inside the `<PRODUCT_HOME>/repository/components/patches/` directory, which will contain the original content of the `<PRODUCT_HOME>/repository/components/plugins/` directory. This step enables you to revert back to the previous state if something goes wrong during operations.

> Prior to Carbon 4.2.0, users were expected to apply patches by starting the server with `wso2server.sh -DapplyPatches`. Now, you do not have to issue a special command to trigger the patch application process. It starts automatically if there are changes in either the `<PRODUCT_HOME>/repository/components/servicepacks/` directory or the `<PRODUCT_HOME>/repository/components/patches/` directory. It verifies all the latest JARs in the `servicepacks` and `patches` directories against the JARs in the `plugins` directory by comparing MD5s of JARs.

Verifying the patch application

After the patch application process is completed, the patch verification process ensures that the latest service pack and other existing patches are correctly applied to the `<PRODUCT_HOME>/repository/components/plugins/` folder.

- All patch related logs are recorded in the `<PRODUCT_HOME>/repository/logs/patches.log` file.
- The `<PRODUCT_HOME>/repository/components/patches/.metadata/prePatchedJARs.txt` meta file contains the list of patched JARs and the md5 values.
- A list of all the applied service packs and patches are in the `<PRODUCT_HOME>/repository/components/default/configuration/prePatched.txt` file.

> Do not change the data in the `<PRODUCT_HOME>/repository/components/default/configuration/prePatched.txt` file. The patch application process gets the pre-patched list from this file and compares the list with the patches available in the `servicepack` and `patches` directories. If you change the data in this file, you will get a startup error when applying patches.

Overview of the patch application process

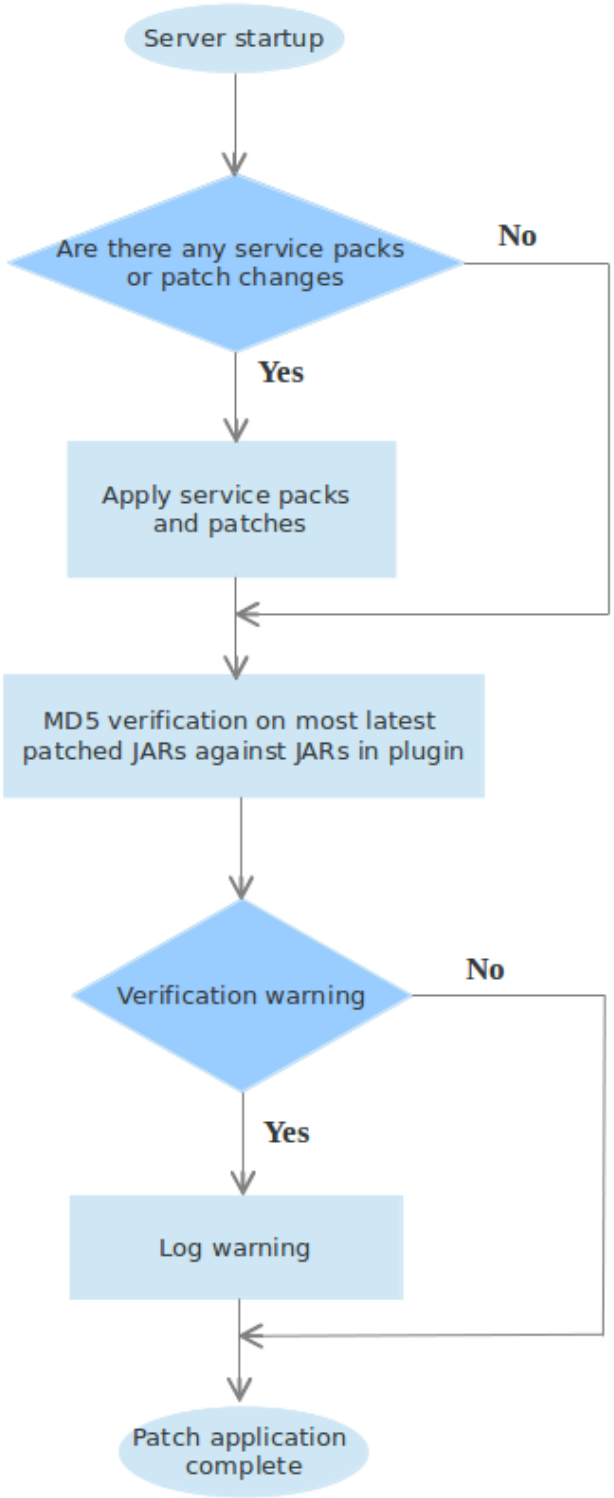
The diagram below shows how the patch application process is implemented when you start the server.

Error Handling

When errors/exception occur in the system, the API Manager throws XML-based error responses by default. To change the format of the error response that is sent to the client, you change the auth failure handler in the `<AM_HOME>/repository/deployment/server/synapse-configs/auth_failure_handler.xml` file. Given below is the default configuration:

```
<sequence name="auth_failure_handler">
   <property name="error_message_type" value="application/xml"/>
   <sequence key="build"/>
</sequence>
```

If you change `application/xml` to something like **`applicatoin/json`**, the error response will be sent in JSON format.



Given below are some error codes and their meanings.

API handlers error codes

Error code	Error Message	Description
900900	Unclassified Authentication Failure.	An unspecified error has occurred
900901	Invalid Credentials	Invalid Authentication information provided

900902	Missing Credentials	No authentication information provided
900903	Access Token Expired	Access Token has expired. Renew the access token.
900904	Access Token Inactive	Access token has become inactive. Generate new access token.
900905	Incorrect Access Token Type is provided	The access token type used is not supported when invoking the API. The supported access token types are Application Accesses Token and User Accesses Token. See Access Tokens.
900906	No matching resource found in the API for the given request	A resource with the name in the request can not be found in the API.
900907	The requested API is temporarily blocked	The status of the API has been changed to an inaccessible/unavailable state.
900908	Resource forbidden	The user invoking the API has not been granted access to the required resource.
900909	The subscription to the API is inactive	Happens when the API user is blocked.
900910	The access token does not allow you to access the requested resource	Can not access the required resource with the provided access token. Check the valid resources that can be accessed with this token.
900800	Message throttled out	The maximum number of requests that can be made to the API within a designated time period is reached and the API is throttled for the user.

Sequences error codes

Error code	Description
900901	Production/sandbox key offered to the API with no production/sandbox endpoint
403	No matching resource found in the API for the given request

In addition to the above error codes, we have engaged Synapse-level error codes to the default fault sequence and custom fault sequences (e.g.,_token_fault_.xml) of the API Manager. For information, see Error Handling in WSO2 ESB documentation.

Getting Support

In addition to this documentation, there are several ways to get help as you work on WSO2 products.

	**Explore learning resources**: For tutorials, articles, whitepapers, webinars, and other learning resources, look in the **Resources** menu on the WSO2 website. In products that have a visual user interface, click the Help link in the top right-hand corner to get help with your current task.
	**Try our support options**: WSO2 offers a variety of development and production support programs, ranging from web-based support during normal business hours to premium 24x7 phone support. For support information, see http://wso2.com/support/.
	**Ask questions in the user forums** at http://stackoverflow.com. Ensure that you tag your question with appropriate keywords such as *WSO2* and the product name so that our team can easily find your questions and provide answers. If you can't find an answer on the user forum, you can email the WSO2 development team directly using the relevant mailing lists described at http://wso2.org/mail.
	**Report issues**, submit enhancement requests, track and comment on issues using our public bug-tracking system, and contribute samples, patches, and tips & tricks (see the WSO2 Contributor License Agreement).

Glossary

Component | Endpoint | SOAP | Denial of Service

Component

Components in the Carbon platform add functionality to all WSO2 Carbon-based products. For example, the statistics component enables users to monitor system and service level statistics. A component in the Carbon platform is made up of one or more OSGi bundles, which is the modularization unit in OSGi similar to a JAR file in Java. For example, the statistics component contains two bundles: one is the back-end bundle that collects, summarizes, and stores statistics, and the other is the front-end bundle, which presents the data to the user through a user-friendly interface. This component-based architecture of the WSO2 Carbon platform gives developers flexibility to build efficient and lean products that best suit their unique business needs simply by adding and removing components.

Endpoint

An endpoint is a specific destination for a message. It may be specified as an Address endpoint, WSDL endpoint, a Failover group, a Loadbalance group, and more. Endpoints can be added, edited, and deleted.

SOAP

An XML-based, extensible message envelope format, with "bindings" to underlying protocols. The primary protocols are HTTP and HTTPS, although bindings for others, including SMTP and XMPP, have been written.

Denial of Service

In a Denial of Service (DOS) attack, the attacker tries to overload the backend services by sending invalid requests such as requests with false return addresses, so that the server cannot find the user when it tries to send the response back. The server gradually slows down when consuming CPU and memory in order to process multiple requests. When the server closes the connection due to failure, the attacker sends a new batch of forged requests, and the process begins again, stalling the services indefinitely.

One of the most common methods of blocking a DOS attack is to filter requests by noticing patterns of incoming traffic. If a pattern comes in frequently, the filter can block messages containing that pattern.

Site Map

Use this site map to quickly find the topic you're looking for by searching for a title on this page using your browser's search feature. You an also use the search box in the upper right corner of this window to search for a word or phrase in all the pages in this documentation.