Key Concepts

Let's take a look at some concepts and terminology that you need to know in order to follow the use cases.

API Manager components

A component is made up of one or more OSGi bundles. A bundle is the modularization unit in OSGi, similar to a JAR file in Java. The component-based architecture of all WSO2 products gives developers flexibility to remove or add features with minimum dependencies.

The API Manager consists of the following high-level components as illustrated in the diagram and listed below:

API Publisher

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 simple Web interface called WSO2 API Publisher for API development 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, and promoting.

The API Publisher URL is https://<YourHostName>:9443/publisher and it is accessible on HTTPS only. The default credentials are admin/admin.

The diagram below shows the common lifecycle activities of an API developer/manager:
**API Store (Developer Portal)**

The API Store Web application provides a collaborative interface for API publishers to host and advertise their APIs and for API consumers to self register, discover, evaluate, subscribe to and use secured, protected, authenticated APIs.

The API Store URL is https://<YourHostName>:9443/store and it is accessible on HTTPS only. The default credentials are admin/admin.

The diagram below shows common API consumer lifecycle activities:

**API Gateway**

A runtime, backend component (an API proxy) developed using WSO2 ESB. API Gateway secures, protects, manages, and scales API calls. It intercepts API requests, applies policies such as throttling and security using handlers, and manages API statistics. Upon validation of a policy, the Gateway passes web service calls to the actual backend. If the service call is a token request, the Gateway passes it directly to the Key Manager.

When WSO2 API Manager is running, you can access the Gateway using the following URL: https://localhost:9443/carbon. You integrate a monitoring and analytics component to the API Manager by configuring WSO2 API Manager Analytics. This component provides reports, statistics and graphs on the APIs deployed in WSO2 API Manager. You can then configure alerts to monitor these APIs and detect unusual activity, manage locations via geo location statistics and, carry out detailed analysis of the logs.
Key Manager

Manages all clients, security and access token-related operations. The Gateway connects with the Key Manager to check the validity of OAuth tokens, subscriptions and API invocations. When a subscriber creates an application and generates an access token to the application using the API Store, the Store makes a call to the API Gateway, which in turn connects with the Key Manager to create an OAuth client and obtain an access token. Similarly, to validate a token, the API Gateway calls the Key Manager, which fetches and validates the token details from the database.

The Key Manager also provides a token API to generate OAuth tokens that can be accessed via the Gateway. All tokens used for validation are based on the 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.

The Key Manager properly decouples the operations for creating OAuth applications and validating access tokens so that you can even plug in a third-party authorization server for key validations.

You can avoid making the Gateway connect with the Key Manager every time it receives an API invocation call, by enabling API Gateway caching. When caching is not enabled, a verification call happens every time the Gateway receives an API invocation call. For this verification, the Gateway passes an access token, the API, and API version to the Key Manager. Communication between the API Gateway and the Key Manager happens in either of the following ways:

- Through a Web service call
- Through a Thrift call (Thrift is the default communication protocol and is much faster than SOAP over HTTP)

If your setup has a cluster of multiple Key Manager nodes that are fronted by a load balancer that does not support Thrift, change the key management protocol from Thrift to WSClient using the `<KeyValidatorClientType>` element in the `<API-M_HOME>/repository/conf/api-manager.xml` file. Thrift uses TCP load balancing.

In a typical production environment, you can configure one of the following setups:

- Configure a WSO2 API Manager instance as the Key Manager in a separate server. See Product Profiles.
- Configure an instance of WSO2 Identity Server as the Key Manager. See Configuring WSO2 Identity Server as the Key Manager.
- Configure a third-party authorization server for key validations and an API Manager instance for the rest of the key management operations. See Configuring a Third-Party Key Manager.

Traffic Manager
The Traffic Manager helps users to regulate API traffic, make APIs and applications available to consumers at different service levels, and secure APIs against security attacks. The Traffic Manager features a dynamic throttling engine to process throttling policies in real-time, including rate limiting of API requests. For more information, see Working with Throttling.

Analytics

Additionally, monitoring and analytics are provided by the analytics component, WSO2 API Manager Analytics. This component provides a host of statistical graphs and an alerting mechanism on pre-determined events. For more information, see Working with Analytics.

Handlers

When an API is created, a file with its synapse configuration is added to the API Gateway. You can find it in the <API-M_HOME>/repository/deployer folder. It has a set of handlers, each of which is executed on the APIs in the same order they appear in the configuration. You find the default handlers in any API's Synapse definition.

For a detailed description of handlers and how to write a custom handler, see Writing Custom Handlers.
Users and roles

The API Manager offers four distinct community roles that are applicable to most enterprises:

- **Admin**: the API management provider who hosts and manages the API Gateway. S/he is responsible for creating user roles in the system, assign them roles, managing databases, security etc. The Admin role is available by default with the credentials admin/admin.
- **Creator**: a creator is a person in a technical role who understands the technical aspects of the API (interfaces, documentation, versions etc.) 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.
- **Publisher**: a publisher manages a set of APIs across the enterprise or business unit and controls the API lifecycle, subscriptions and monetization aspects. The publisher is also interested in usage patterns for APIs and has access to all API statistics.
- **Subscriber**: a subscriber uses the API store to discover APIs, read the documentation and forums, rate/comment on the APIs, subscribes to APIs, obtain access tokens and invoke the APIs.

Tip: See Managing Users and Roles for more information.

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 lifecycle:

- **CREATED**: API metadata is added to the API Store, but it is not deployed in the API Gateway and therefore, is not visible to subscribers in the API Store.
- **PROTOTYPED**: The 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 can invoke the API without a subscription.
- **PUBLISHED**: The API is visible in the API Store and available for subscription.
- **DEPRECATED**: When an API is deprecated, new subscriptions are disabled. But the API is still deployed in the Gateway and is available at runtime to existing subscribers. Existing subscribers can continue to use it as usual until the API is retired.
Applications

An application is a logical collection of APIs. 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 tiers/Service Level Agreement (SLA) levels

You subscribe to APIs through an application. Applications are available at different SLA levels and have application-level throttling tiers engaged in them. A throttling tier determines the maximum number of calls you can make to an API during a given period of time.

The API Manager comes with a pre-created default application, which allows unlimited access by default. You can also create your own applications.

Callback URL

A callback URL is a URL that sends a callback to a specific server or program soon after your application request is sent. The callback URL is specified when generating or re-generating production or sandbox keys for an application. You can either provide a single application callback URL or a RegEx pattern as the callback URL. You use a RegEx pattern as the callback URL when you need to specify multiple callback URLs for an application.

For example, consider a situation where you have two (2) service providers that need to use the same application that have the following callback URLs:

- https://mytestapp.com/callback
- https://testapp:8000/callback

In this instance, your callback URL should have the following RegEx pattern:

```
```

You can configure any RegEx pattern to match the callback URLs that you need to register with the application. However, it is mandatory to have the prefix `regexp=` before the pattern.

Access tokens

An access token is a simple string that is passed as an HTTP header of a request. For example, "Authorization: Bearer NcBoKQoRE1u0H1a1fQ0DIfo6I4a." Access tokens authenticate API users and applications, and ensure better security (e.g., prevent certain types of DoS attacks). Note that DoS attacks made to the key manager with random access tokens can not be prevented. DoS attacks with the same fake access token can affect the Gateway as well. If a token that is passed with a request is invalid, the request is discarded at the first stage of processing. Access tokens work equally well for SOAP and REST calls.

For more information on different types of access tokens and how to generate them, see Working with Access Tokens.

Throttling tiers

Throttling allows you to limit the number of successful 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 certain types of Denial of Service (DoS) attacks
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 purposes

Multi tenanted API management

A tenant in WSO2 API Manager is a separate business level entity, such as an organizational unit or a department. Multi tenancy enables such organizational units/departments to share the same API Manager deployment and the respective resources, but function individually with an isolated view of the deployment. A tenant does not need to be aware of the other tenants in the system as by design the Multitenancy creates an isolated space for each tenant, although they are sharing the same deployment.

See the section Configuring Multiple Tenants for information on how to create tenants.

In WSO2 API Manager deployment, API Visibility and Subscription Availability are the two main applications of tenancy.

API visibility

API Manager allows users to control API visibility and subscription availability. API visibility can be one of the following options:

- Public
- Restricted by roles
- Visible to my domain

Subscription availability

Subscription availability has three options. Those options are as follows:

- Available to current Tenant Only
- Available to All the Tenants
- Available to Specific Tenants

See the next section on API Visibility and Subscription for more details on each category.

Additionally, it’s possible to configure a secondary userstore per tenant as well. For more information, see Configuring Secondary Userstores.

API visibility and subscription Availability

API visibility
Visibility settings prevent certain user roles from viewing and modifying APIs created by another user role.

- **Public**: The API is visible to all users who are registered and anonymous (who use APIs without login to the store, for example testing and demonstration), and can be advertised in multiple stores (central and non-WSO2 stores).
- **Restricted by Roles**: The API is visible to its tenant domain and only to the user roles that you specify. You should provide the roles separated by commas in the UI or as a cURL parameter when creating or editing the API.
- **Visible to my domain**: The API is visible to all users who are registered to the API's tenant domain. This option is available only in a multi-tenanted environment. It’s not applicable when there is only one active tenant (super tenant) in the system.

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

- The API **Creator** and **Publisher** roles can see all APIs in their tenant store even if you restrict access to them. This is because those roles have permission to view and edit all APIs in the API Publisher, and therefore, does not have to be restricted in the Store.
- Anonymous users can only see APIs that have the visibility set as **Public**.
- Registered users can see all APIs in all tenant domains.
- The API is not restricted to a role that the user is assigned to.

By default, an API has public visibility. You can set the API visibility in the **Design** tab of the API Publisher at the time the API is created or updated.

### Design API

When using the REST API directly, these visibility options are available as **public**, **private** and **restricted**.

<table>
<thead>
<tr>
<th>API visibility level specified in the UI</th>
<th>API visibility level specified in the REST API</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public</td>
<td>public i.e. &quot;visibility&quot;: &quot;PUBLIC&quot;</td>
</tr>
<tr>
<td>Visible to my domain</td>
<td>private i.e. &quot;visibility&quot;: &quot;PRIVATE&quot;</td>
</tr>
<tr>
<td>Restricted by roles</td>
<td>restricted i.e. &quot;visibility&quot;: &quot;RESTRICTED&quot;, visibleRoles: [&quot;role1&quot;,&quot;role2&quot;,&quot;role3&quot;]</td>
</tr>
</tbody>
</table>

### Subscription availability

The subscription availability option has three values as follows. You can set subscription availability to an API through the API Publisher’s **Manage** tab.

- **Available to current tenant only**: only users in the current organization/tenant domain can subscribe to the API.
- **Available to all the tenants**: users of all organizations/tenant domains can subscribe to the API.
- **Available to specific tenants**: users of the organizations/tenant domains that you specify, as well as the current tenant domain, can subscribe to the API.
Subscription is only available to the current tenant in the following instances:

- When there is only one tenant in your system.
- Even if there are multiple tenants in your system, when you select **Visible to my domain** or **Restricted by roles** as the API's visibility in the previous step.

The diagram below depicts the relationship between the API's visibility and subscription availability:

Refer the article [Multi Tenant API Management with WSO2 API Manager](#) for examples and real world usage of the above concepts.

### API documentation visibility

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 the `<API-M_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, click the Docs tab of an API, and click Add New Document to see a new drop-down list added to select visibility from:

PhoneVerification - 1.0.0

You set visibility in the following ways:

- **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 domain.
- **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.

### API resources

An API is made up of one or more resources, each of which handles a particular type of request. A resource has a set of methods that operate on it. The methods are analogous to a method or a function, and a resource is analogous to an object instance or a class in an object-oriented programming language. There are a few standard methods defined for a resource (corresponding to the standard HTTP GET, POST, PUT and DELETE methods.)

The diagram below shows a resource by the name CheckPhoneNumber added with four HTTP methods.

When you add resources to an API, you define a URL pattern and HTTP methods. A resource can also have a list of OAuth scopes.
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.state` and `uri.var.town` properties. For more information on how to use these properties, see Introduction to REST API and the HTTP Endpoint of the WSO2 ESB documentation.


Once a request is accepted by a resource, it will be mediated through an in-sequence. Any response from the backend is handled through the out-sequence. Fault sequences are used to mediate errors that might occur in either sequence. The diagram below shows the flow of a request sent to an API resource.

![API Resource Flow Diagram](image)

The default in-sequence, out-sequence and fault sequences are generated when the API is published.

**HTTP methods**

HTTP methods specify the desired action to be performed on an API's resource. You can select multiple methods from GET, POST, PUT, DELETE, PATCH, HEAD and OPTIONS. A method has attributes such as an OAuth scope, authentication type, response content type, parameters etc. as the diagram below shows:
The main attributes of a method are described below:

**Response content type**
You can use the `Produces` attribute to document in what type the backend sends the response back to the API Manager. **Note that this attribute does not do any message type conversion**, it is only used as a way of letting the user know the type of response they will receive. You can define the response content type in the **Design** tab when creating an API.

**Parameters**
Parameters of an HTTP method are analogous to arguments of a function in an object-oriented programming language. A resource's parameters are cached in the **resource cache** at the API Gateway. You can define the parameters in the **Design** tab when creating an API.

**OAuth scopes**
You can define a list of OAuth scopes to an API's resource and assign one of them to each HTTP method. You can define the OAuth scopes in the **Manage** tab when creating an API as shown above.
Authentication type

You can set the response content type by navigating to the Manage tab, when creating an API, and select the corresponding authentication from the "Application & Application user" drop-down list, which corresponds to the respective resource.

The authentication type can be one of the following:

- **None**: You can select this option to create a non-authenticated API. This will remove the authentication requirements for regular managed APIs, so that the API Gateway skips the authentication process. This is useful in instances when you want to still have the API listed on the API Store home screen and/or when you want to disable authentication requirement for individual resources of an API.

- **Application**: The authentication is done by the application and the resource accepts application access tokens. Therefore, if you select this option, the API will require OAuth tokens generated with a client grant type that produces tokens specific to the subscription, but not the end-users. Therefore, if there is a web or mobile application that has subscribed to this API and it has multiple end-users, then they will all share the same token and the API backend will not know which end-user invoked the API.

- **Application User**: Authentication is done by the application user and the resource accepts user access tokens. Therefore, if you select this option, the API accepts the OAuth tokens that are generated with the password grant type. These tokens are specific to the end-user, because they not just require the application key, but also the end-user’s username and password. In this case, each end-user gets their own OAuth tokens even though they are using the same API subscription. The API Gateway then generates a JWT token and uses it to pass application and user information to the API backend.

- **Application and Application User**: Both application and application user level authentication is applied. Thereby, both kinds of tokens are acceptable by the API.

Note that if you select this option in the UI, it appears as in the API Manager's internal data storage and will appear in the response messages as well.

Note that for the resources that have HTTP verbs (GET, POST etc.) requiring authentication (i.e., Auth Type is not NONE), set None as the Auth type of OPTIONS. This is to support CORS (Cross Origin Resource Sharing) between the API Store and Gateway.

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.

Cross-origin resource sharing

Cross-origin resource sharing (CORS) is a mechanism that allows restricted resources (e.g., fonts, JavaScript) of a Web page to be requested from another domain outside the domain from which the resource originated.

The Swagger API Console that is integrated in the API Manager runs as a JavaScript client in the API Store and makes calls from the Store to the API Gateway. Therefore, if you have the API Store and Gateway running on different ports, enable CORS between them.

The CORS configuration is in the <APIM_HOME>/repository/conf/api-manager.xml file. Given below is a sample code.

```xml
<CORSConfiguration>
  <Enabled>true</Enabled>
  <Access-Control-Allow-Methods>GET,PUT,POST,DELETE,PATCH,OPTIONS</Access-Control-Allow-Methods>
  <Access-Control-Allow-Headers>authorization,Access-Control-Allow-Origin,Content-Type,SOAPAction</Access-Control-Allow-Headers>
  <Access-Control-Allow-Credentials>false</Access-Control-Allow-Credentials>
</CORSConfiguration>
```

The elements are described below:

<table>
<thead>
<tr>
<th>Header</th>
<th>Description</th>
<th>Sample values</th>
</tr>
</thead>
</table>
Access-Control-Allow-Origin | Determines whether a resource can be shared with the resource of a given origin. The API Gateway validates the origin request header value against the list of origins defined under the Access Control Allow Origins configuration (this can be All Allow Origins or a specific value like localhost). If the host is in the allowed origin list, it will be set as the Access-Control-Allow-Origin response header in the response. | All Allow Origins(*), localhost

Access-Control-Allow-Headers | Determines, as part of the response to a preflight request (a request that checks to see if the CORS protocol is understood), which header field names can be used during the actual request. The gateway will set the header values defined under Access Control Allow Headers configurations. | authorization, Access-Control-Allow-Origin, Content-type, SOAPAction

Access-Control-Allow-Methods | This header specifies the method(s) allowed when accessing the resource in response to a preflight request. Required methods can be defined under the Access Control Allow Method configuration. | GET, PUT, POST, DELETE, PATCH, OPTIONS

Access-Control-Allow-Credentials | Determines whether or not the response to the request can be exposed to the page. It can be exposed when the header value is true. The header value can be set to true/false by enabling/disabling the Access Control Allow Credentials configuration. | true, false

If you try to invoke an API with inline endpoints, you add the CORS Handler in the <handlers> section of the API's configuration as follows. You can find the API's configuration in the <APIM_HOME>/repository/deployment/server/synapse-configs/default/api folder. Change your code according to the sample given here.

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

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.

OAuth provides a method for clients to access a protected resource on behalf of a resource owner. OAuth 2 bearer token is a security token that any party in possession of it can use the token for authentication. Refer OAuth 2.0 Specification of Bearer Token Usage for more information.

How scopes work

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

![Diagram](https://via.placeholder.com/150)

Scope: news_read
Roles: employee and manager roles

Scope: news_write
Roles: manager role only

Resource: /read (GET)
Resource: /write (POST)
Assume there are two users Tom and John. Tom is assigned the employee role. John is assigned both employee and manager roles.

Tom requests a token through the Token API as `grant_type=password&username=tom&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":"8579fcb65d1d3ebe74a395a2e78dd6",
"access_token":"eb51eff0b4d85cdalebd312c5b6a3b8"
```

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":"4ca244fb321bd55b5d555df39315",
"access_token":"42a377a0101877dd9e29c5f30857e"
```

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: eb51eff0b4d85cdalebd312c5b6a3b8</ams:description>
</ams:fault>
```

### Applying a scope

You 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 under Resources.

On the screen that appears, enter a scope key, scope name and optionally, allowed roles and a description. Click Add Scope.
Define Scope

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.

Note that the role name is case sensitive in the DBMSs that are case sensitive, such as PostgreSQL.

When the role you specify is in a secondary user store, you have to give the role as <userstore name>/role name.

Description
E.g., This scope will group all the administration APIs

To apply the scope, you add the scope to a resource, save and publish the API.
A scope is not always used for controlling access to a resource. You can also use it to simply mark an access token. There are scopes that cannot be associated to roles (e.g., openid, device_). Such scopes do not have to have roles associated with them. Skipping role validation for scopes is called scope whitelisting.

If you do not want a role validation for a scope in an API's request, add the scope under the `<ScopeWhitelist>` element in the `<APIM_HOME>/repository/conf/api-manager.xml` file and restart the server. It will be whitelisted. For example,

```
<ScopeWhitelist>
  <Scope>device_.*</Scope>
  <Scope>some_random_scope</Scope>
</ScopeWhitelist>
```

Next, invoke the Token API to get a token for the scope that you just whitelisted. For example,
```bash
curl -k -d "grant_type=password&username=admin&password=admin&scope=some_random_scope" -H "Authorization: Basic WmRFUFBvZmZwYVFnR25ScG5iZldtcUtSS3IwYTpSaG5ocEVJYUVCMEN3T1FReWpiZTJwaDBzc1Vh" -H "Content-Type: application/x-www-form-urlencoded" https://10.100.0.3:8243/token
```

Note that the issued token has the scope you requested. You get the token without any role validation as the scope is whitelisted.

```json
{"scope":"some_random_scope","token_type":"bearer","expires_in":3600,"refresh_token":"59e6676db0addca46e68991e44f2b9b8","access_token":"48855d444db883171c347fa21ba77e8"}
```

### API templates

An API template is its XML representation, which is saved in `<APIM_HOME>/repository/resources/api_templates/velocity_template.xml` file. This file comes with the API Manager by default. You can edit this default template to 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.

### Endpoints

An endpoint is a specific destination for a message such as an address, WSDL, a failover group, a load-balance group etc.

WSO2 API Manager has support for a range of different endpoint types, allowing the API Gateway to connect with advanced types of backends. It supports HTTP endpoints, URL endpoints (also termed as address endpoint), WSDL endpoints, Failover endpoints, Load-balanced endpoints. For more information about endpoints, see Working with Endpoints.

### Sequences

The API Manager has a default mediation flow that is executed in each API invocation. There are 3 default sequences engaged as **in**, **out** and **fault** which perform following.

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>In-sequence is the first place that will be mediated through, once a request is dispatched to a resource of an API. At the end of the in-sequence the request can be forwarded to a back-end application for further processing.</td>
</tr>
<tr>
<td>out</td>
<td>Any responses coming from the back-end system are mediated through the out-sequence of the resource of the API.</td>
</tr>
<tr>
<td>fault</td>
<td>Fault sequence is there to handle any errors that may occur while mediating a message through a resource. When the sequence or the proxy service encounters an error during mediation or while forwarding a message, the message that triggered the error is delegated to the specified fault sequence. Using the available mediators it is possible to log the erroneous message, forward it to a special error-tracking service, and send a SOAP fault back to the client indicating the error. We need to configure the fault sequence with the correct error handling instead of simply dropping messages. For more information, see Error Handling.</td>
</tr>
</tbody>
</table>

### Caching

For information on configuring caching response messages and caching API calls at the Gateway and Key Manager server, see Configuring Caching.
1. Populate cache entry for token
2. Key validation happens
3. Validation information gets added to cache entry
4. JWT token also cached
5. Resource level details are cached
6. Call goes to key manager only if there is no cached entry