Hortonworks Data Platform

Data Governance

(Nov 30, 2016)

docs.hortonworks.com

Hortonworks Data Platform: Data Governance

Copyright © 2012-2016 Hortonworks, Inc. Some rights reserved.

The Hortonworks Data Platform, powered by Apache Hadoop, is a massively scalable and 100% open source platform for storing, processing and analyzing large volumes of data. It is designed to deal with data from many sources and formats in a very quick, easy and cost-effective manner. The Hortonworks Data Platform consists of the essential set of Apache Hadoop projects including MapReduce, Hadoop Distributed File System (HDFS), HCatalog, Pig, Hive, HBase, ZooKeeper and Ambari. Hortonworks is the major contributor of code and patches to many of these projects. These projects have been integrated and tested as part of the Hortonworks Data Platform release process and installation and configuration tools have also been included.

Unlike other providers of platforms built using Apache Hadoop, Hortonworks contributes 100% of our code back to the Apache Software Foundation. The Hortonworks Data Platform is Apache-licensed and completely open source. We sell only expert technical support, training and partner-enablement services. All of our technology is, and will remain, free and open source.

Please visit the Hortonworks Data Platform page for more information on Hortonworks technology. For more information on Hortonworks services, please visit either the Support or Training page. Feel free to contact us directly to discuss your specific needs.

Except where otherwise noted, this document is licensed under
Creative Commons Attribution ShareAlike 3.0 License.
http://creativecommons.org/licenses/by-sa/3.0/legalcode

Table of Contents

1. HDP Data Governance 1
1.1. Apache Atlas Features 1
1.2. Atlas Architecture 3
1.2.1. Core 3
1.2.2. Integration 4
1.2.3. Metadata Sources 4
1.3. Atlas-Ranger Integration 4
2. Installing and Configuring Apache Atlas 7
2.1. Installing and Configuring Apache Atlas Using Ambari 7
2.1.1. Apache Atlas Prerequisites 7
2.1.2. Authentication Settings 7
2.1.3. Authorization Settings 12
2.2. Configuring Atlas Tagsync in Ranger 14
2.3. Configuring Atlas High Availability 14
2.4. Configuring Atlas Security 14
2.4.1. Additional Requirements for Atlas with Ranger and Kerberos 14
2.4.2. Enabling Atlas HTTPS 15
2.4.3. Hive CLI Security 15
2.5. Installing Sample Atlas Metadata 16
2.6. Updating the Atlas Ambari Configuration 16
2.7. Using Distributed HBase as the Atlas Metastore 16
3. Searching and Viewing Assets 20
3.1. Using Text and DSL Search 20
3.2. Viewing Asset Data Lineage 22
3.3. Viewing Asset Details 24
4. Working with Atlas Tags 30
4.1. Creating Atlas Tags 30
4.2. Associating Tags with Assets 31
4.3. Searching for Assets Associated with Tags 34
5. Managing the Atlas Business Taxonomy (Technical Preview) 36
5.1. Enabling the Atlas Taxonomy Technical Preview 36
5.2. Creating Taxonomy Terms 40
5.3. Associating Taxonomy Terms with Assets 48
5.4. Navigating the Atlas Taxonomy 51
5.4.1. Navigation Arrows 51
5.4.2. Breadcrumb Trail 52
5.4.3. Search Terms 53
5.4.4. Back Button 53
5.5. Searching for Assets Associated with Taxonomy Terms 54
6. Apache Atlas REST API Reference 56
6.1. Data Model 56
6.2. AdminResource 56
6.3. DataSetLineageResource 57
6.4. EntityService 58
6.5. LineageResource 62
6.6. MetadataDiscoveryResource 62
6.7. TaxonomyService 64
6.8. TypesResource 67

List of Figures

1.1. Atlas Overview 2

List of Tables

2.1. Apache Atlas File-based Configuration Settings 9
2.2. Apache Atlas LDAP Configuration Settings 10
2.3. Apache Atlas AD Configuration Settings 11
2.4. Apache Atlas Simple Authorization 12

1. HDP Data Governance

Apache Atlas provides governance capabilities for Hadoop that use both prescriptive and forensic models enriched by business taxonomical metadata. Atlas is designed to exchange metadata with other tools and processes within and outside of the Hadoop stack, thereby enabling platform-agnostic governance controls that effectively address compliance requirements.

Apache Atlas enables enterprises to effectively and efficiently address their compliance requirements through a scalable set of core governance services. These services include:

- Search and Proscriptive Lineage – facilitates pre-defined and *ad hoc* exploration of data and metadata, while maintaining a history of data sources and how specific data was generated.
- Metadata-driven data access control.
- Flexible modeling of both business and operational data.
- Data Classification – helps you to understand the nature of the data within Hadoop and classify it based on external and internal sources.
- Metadata interchange with other metadata tools.

1.1. Apache Atlas Features

Apache Atlas is a low-level service in the Hadoop stack that provides core metadata services. Atlas currently provides metadata services for the following components:

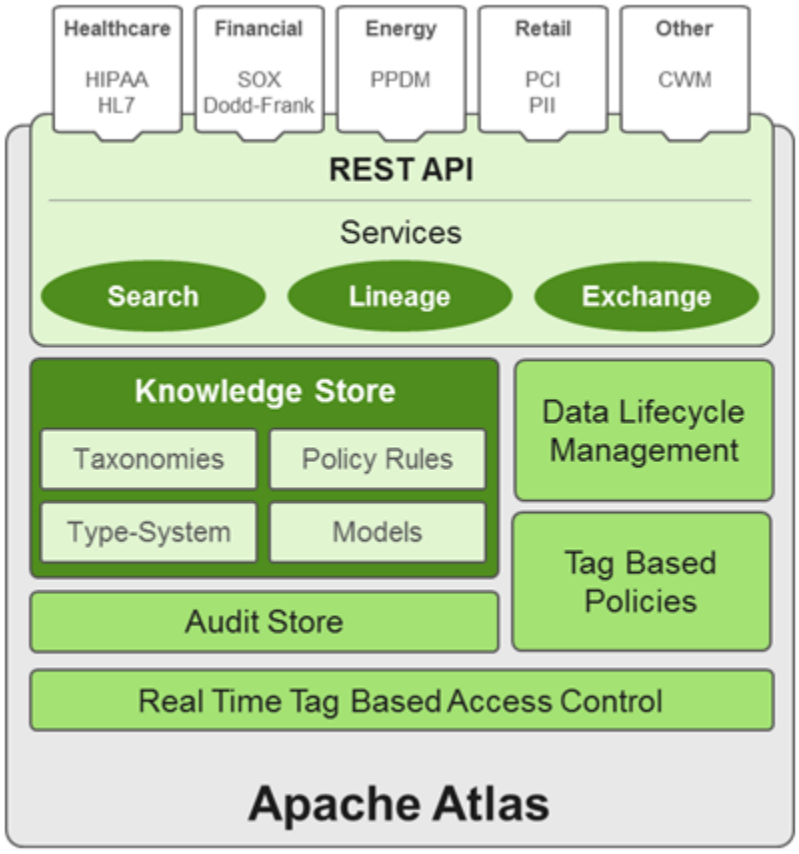
- Hive
- Ranger
- Sqoop
- Storm/Kafka (limited support)
- Falcon (limited support)

Apache Atlas provides the following features:

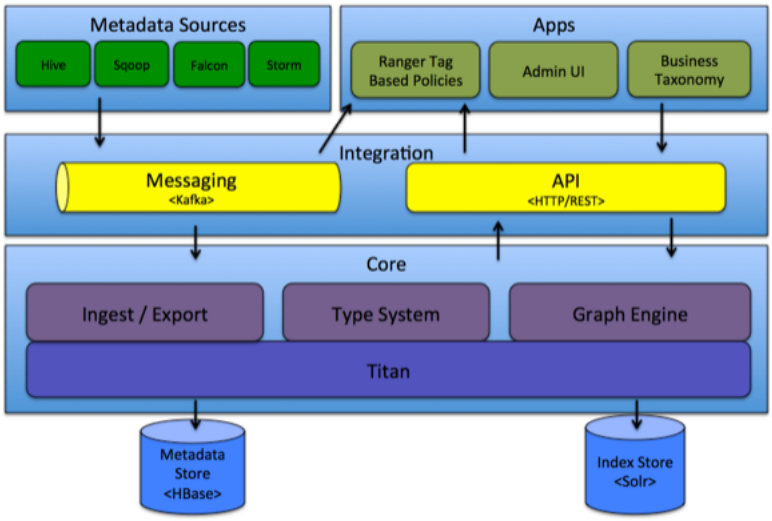
- **Knowledge store that leverages existing Hadoop metastores:** Categorized into a business-oriented taxonomy of data sets, objects, tables, and columns. Supports the exchange of metadata between HDP foundation components and third-party applications or governance tools.
- **Data lifecycle management:** Leverages existing investment in Apache Falcon with a focus on provenance, multi-cluster replication, data set retention and eviction, late data handling, and automation.
- **Audit store:** Historical repository for all governance events, including security events (access, grant, deny), operational events related to data provenance and metrics. The Atlas audit store is indexed and searchable for access to governance events.

- **Security:** Integration with HDP security that enables you to establish global security policies based on data classifications and that leverages Apache Ranger plug-in architecture for security policy enforcement.
- **Policy engine:** Fully extensible policy engine that supports metadata-based, geo-based, and time-based rules that rationalize at runtime.
- **RESTful interface:** Supports extensibility by way of REST APIs to third-party applications so you can use your existing tools to view and manipulate metadata in the HDP foundation components.

Figure 1.1. Atlas Overview



1.2. Atlas Architecture



1.2.1. Core

This category contains the components that implement the core of Atlas functionality, including:

Type System: Atlas allows you to define a model for metadata objects. The model is composed of "types" definitions. "Entities" are instances of Types that represent the actual metadata objects. The Type system allows you to define and manage types and entities. All metadata objects managed by Atlas out-of-the-box (such as Hive tables) are modelled using types and represented as entities.

One key point to note is that the generic nature of the modelling in Atlas allows data stewards and integrators to define both technical metadata and business metadata. It is also possible to use Atlas to define rich relationships between technical and business metadata.

Ingest / Export: The Ingest component allows metadata to be added to Atlas. Similarly, the Export component exposes metadata changes detected by Atlas to be raised as events. Consumers can use these change events to react to metadata changes in real time.

Graph Engine: Internally, Atlas represents metadata objects using a Graph model. This facilitates flexibility and rich relationships between metadata objects. The Graph Engine is a component that is responsible for translating between types and entities of the Type System, as well as the underlying Graph model. In addition to managing the Graph objects, The Graph Engine also creates the appropriate indices for the metadata objects to facilitate efficient searches.

Titan: Currently, Atlas uses the Titan Graph Database to store the metadata objects. Titan is used as a library within Atlas. Titan uses two stores. The Metadata store is configured to use HBase by default, and the Index store is configured to use Solr. It is also possible to use BerkeleyDB as the Metadata store, and ElasticSearch as the Index store, by building with those corresponding profiles. The Metadata store is used for storing the metadata

objects, and the Index store is used for storing indices of the Metadata properties to enable efficient search.

1.2.2. Integration

You can manage metadata in Atlas using the following methods:

API: All functionality of Atlas is exposed to end users via a REST API that allows types and entities to be created, updated, and deleted. It is also the primary mechanism to query and discover the types and entities managed by Atlas.

Atlas Admin UI: This component is a web-based application that allows data stewards and scientists to discover and annotate metadata. Of primary importance here is a search interface and SQL-like query language that can be used to query the metadata types and objects managed by Atlas. The Admin UI is built using the Atlas REST API.

Messaging: In addition to the API, you can integrate with Atlas using a messaging interface that is based on Kafka. This is useful both for communicating metadata objects to Atlas, and also to transmit metadata change events from Atlas to applications. The messaging interface is particularly useful if you would like to use a more loosely coupled integration with Atlas that could allow for better scalability and reliability. Atlas uses Apache Kafka as a notification server for communication between hooks and downstream consumers of metadata notification events. Events are written by the hooks and Atlas to different Kafka topics.

1.2.3. Metadata Sources

Currently, Atlas supports ingesting and managing metadata from the following sources:

- Hive
- Sqoop
- Storm/Kafka (limited support)
- Falcon (limited support)

As a result of his integration:

- There are metadata models that Atlas defines natively to represent objects of these components.
- Atlas provides mechanisms to ingest metadata objects from these components (in real time, or in batch mode in some cases).

1.3. Atlas-Ranger Integration

Atlas provides data governance capabilities and serves as a common metadata store that is designed to exchange metadata both within and outside of the Hadoop stack. Ranger provides a centralized user interface that can be used to define, administer and manage security policies consistently across all the components of the Hadoop stack. The Atlas-

Ranger unites the data classification and metadata store capabilities of Atlas with security enforcement in Ranger.

You can use Atlas and Ranger to implement dynamic classification-based security policies, in addition to role-based security policies. Ranger's centralized platform empowers data administrators to define security policy based on Atlas metadata tags or attributes and apply this policy in real-time to the entire hierarchy of assets including databases, tables, and columns, thereby preventing security violations.

Ranger-Atlas Access Policies

- **Classification-based access controls:** A data asset such as a table or column can be marked with the metadata tag related to compliance or business taxonomy (such as "PCI"). This tag is then used to assign permissions to a user or group. This represents an evolution from role-based entitlements, which require discrete and static one-to-one mapping between user/group and resources such as tables or files. As an example, a data steward can create a classification tag "PII" (Personally Identifiable Information) and assign certain Hive table or columns to the tag "PII". By doing this, the data steward is denoting that any data stored in the column or the table has to be treated as "PII". The data steward now has the ability to build a security policy in Ranger for this classification and allow certain groups or users to access the data associated with this classification, while denying access to other groups or users. Users accessing any data classified as "PII" by Atlas would be automatically enforced by the Ranger policy already defined.

- **Data Expiry-based access policy:** For certain business use cases, data can be toxic and have an expiration date for business usage. This use case can be achieved with Atlas and Ranger. Apache Atlas can assign expiration dates to a data tag. Ranger inherits the expiration date and automatically denies access to the tagged data after the expiration date.

- **Location-specific access policies:** Similar to time-based access policies, administrators can now customize entitlements based on geography. For example, a US-based user might be granted access to data while she is in a domestic office, but not while she is in Europe. Although the same user may be trying to access the same data, the different geographical context would apply, triggering a different set of privacy rules to be evaluated.

- **Prohibition against dataset combinations:** With Atlas-Ranger integration, it is now possible to define a security policy that restricts combining two data sets. For example, consider a scenario in which one column consists of customer account numbers, and another column contains customer names. These columns may be in compliance individually, but pose a violation if combined as part of a query. Administrators can now apply a metadata tag to both data sets to prevent them from being combined.

Cross Component Lineage

Apache Atlas now provides the ability to visualize cross-component lineage, delivering a complete view of data movement across a number of analytic engines such as Apache Storm, Kafka, Falcon, and Hive.

This functionality offers important benefits to data stewards and auditors. For example, data that starts as event data through a Kafka bolt or Storm Topology is also analyzed as an aggregated dataset through Hive, and then combined with reference data from a

RDBMS via Sqoop, can be governed by Atlas at every stage of its lifecycle. Data stewards, Operations, and Compliance now have the ability to visualize a data set's lineage, and then drill down into operational, security, and provenance-related details. As this tracking is done at the platform level, any application that uses these engines will be natively tracked. This allows for extended visibility beyond a single application view.

2. Installing and Configuring Apache Atlas

2.1. Installing and Configuring Apache Atlas Using Ambari

To install Apache Atlas using Ambari, follow the procedure in Adding a Service to your Hadoop cluster in the Ambari User's Guide. On the Choose Services page, select the Atlas service. When you reach the Customize Services step in the Add Service wizard, set the following Atlas properties, then complete the remaining steps in the Add Service wizard. The Atlas user name and password are set to `admin`/`admin` by default.

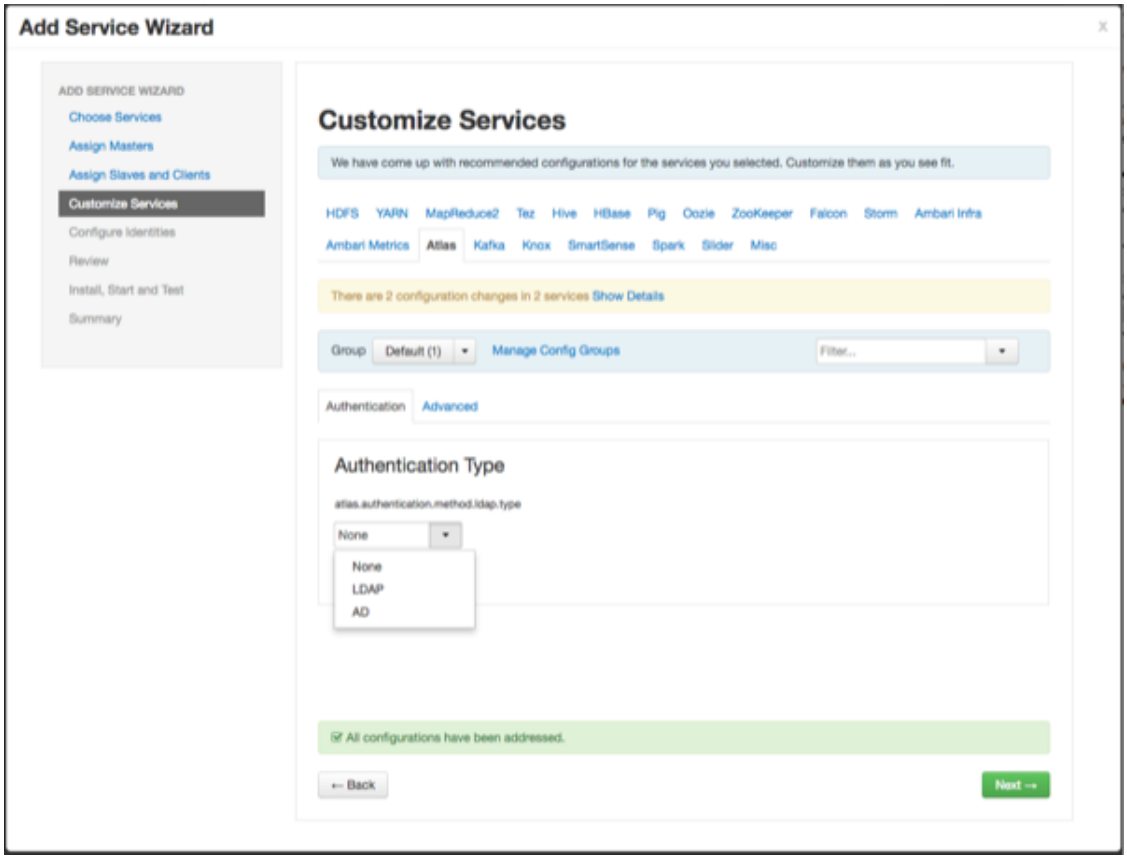
2.1.1. Apache Atlas Prerequisites

Apache Atlas requires the following components:

- Ambari Infra (which includes an internal HDP Solr Cloud instance) or an externally managed Solr Cloud instance.
- HBase (used as the Atlas Metastore).
- Kafka (provides a durable messaging bus).

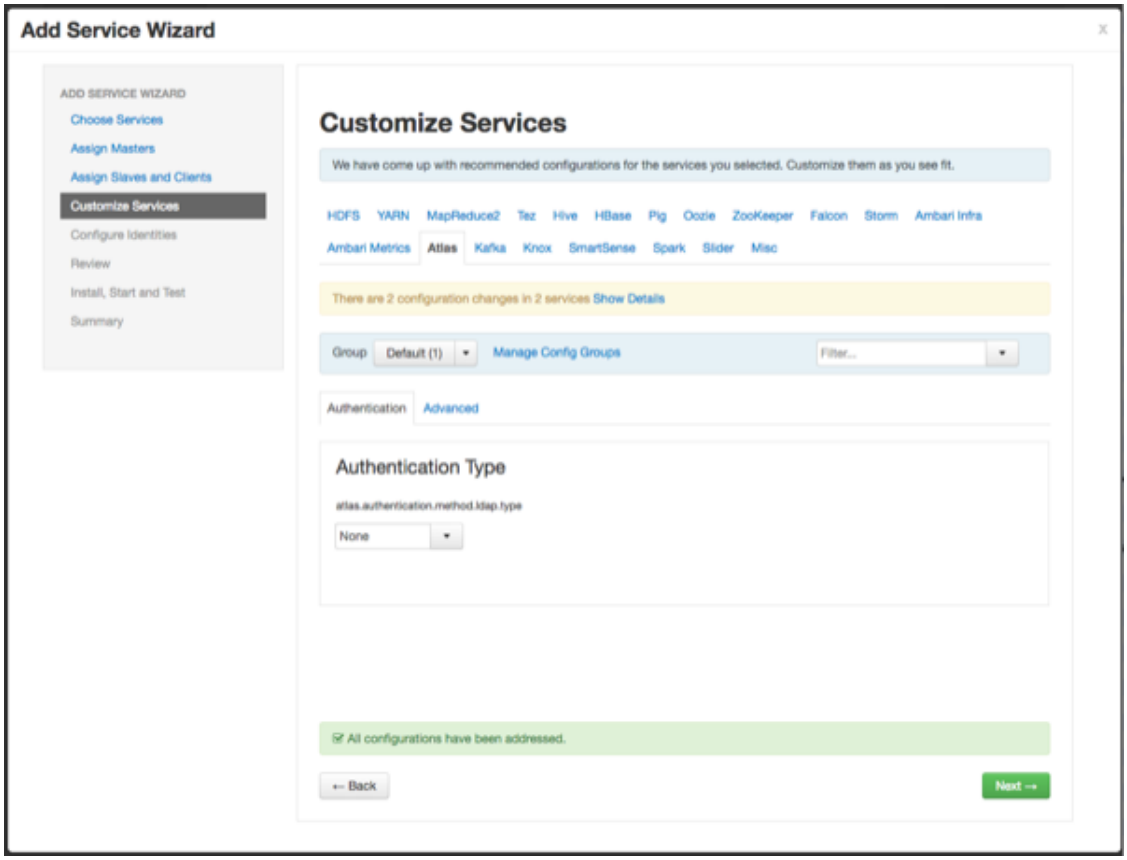
2.1.2. Authentication Settings

You can set the Authentication Type to None, LDAP, or AD. If authentication is set to None, file-based authentication is used.



2.1.2.1. File-based Authentication

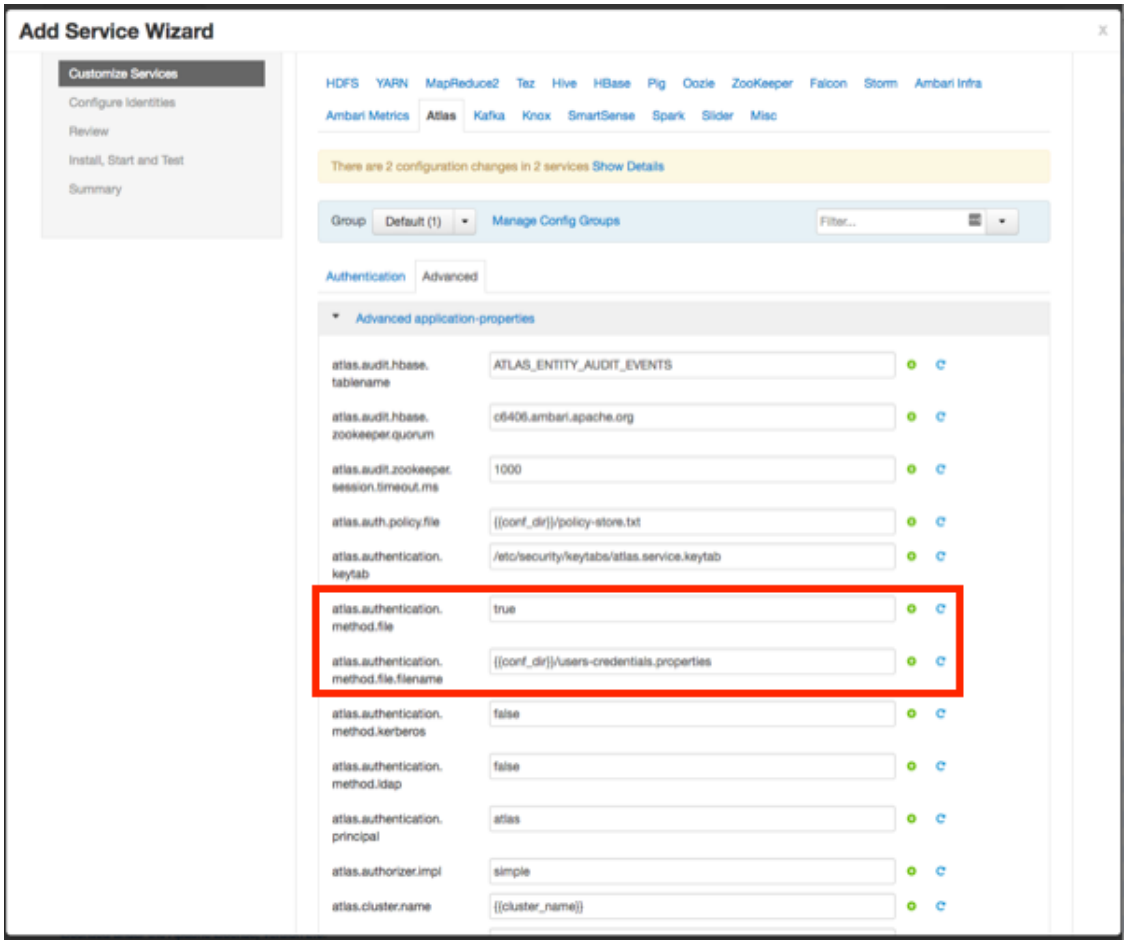
Select **None** to default to file-based authentication.



When file-based authentication is selected, the following properties are automatically set under **Advanced application-properties** on the Advanced tab.

Table 2.1. Apache Atlas File-based Configuration Settings

Property	Value
atlas.authentication.method.file	true
atlas.authentication.method.file.filename	{{conf_dir}}/users-credentials.properties



The `users-credentials.properties` file should have the following format:

```
username=group::sha256password
admin=ADMIN::e7cf3ef4f17c3999a94f2c6f612e8a888e5b1026878e4e19398b23bd38ec221a
```

The user group can be `ADMIN`, `DATA_STEWARD`, or `DATA_SCIENTIST`.

The password is encoded with the `sha256` encoding method and can be generated using the UNIX tool:

```
echo -n "Password" | sha256sum
e7cf3ef4f17c3999a94f2c6f612e8a888e5b1026878e4e19398b23bd38ec221a -
```

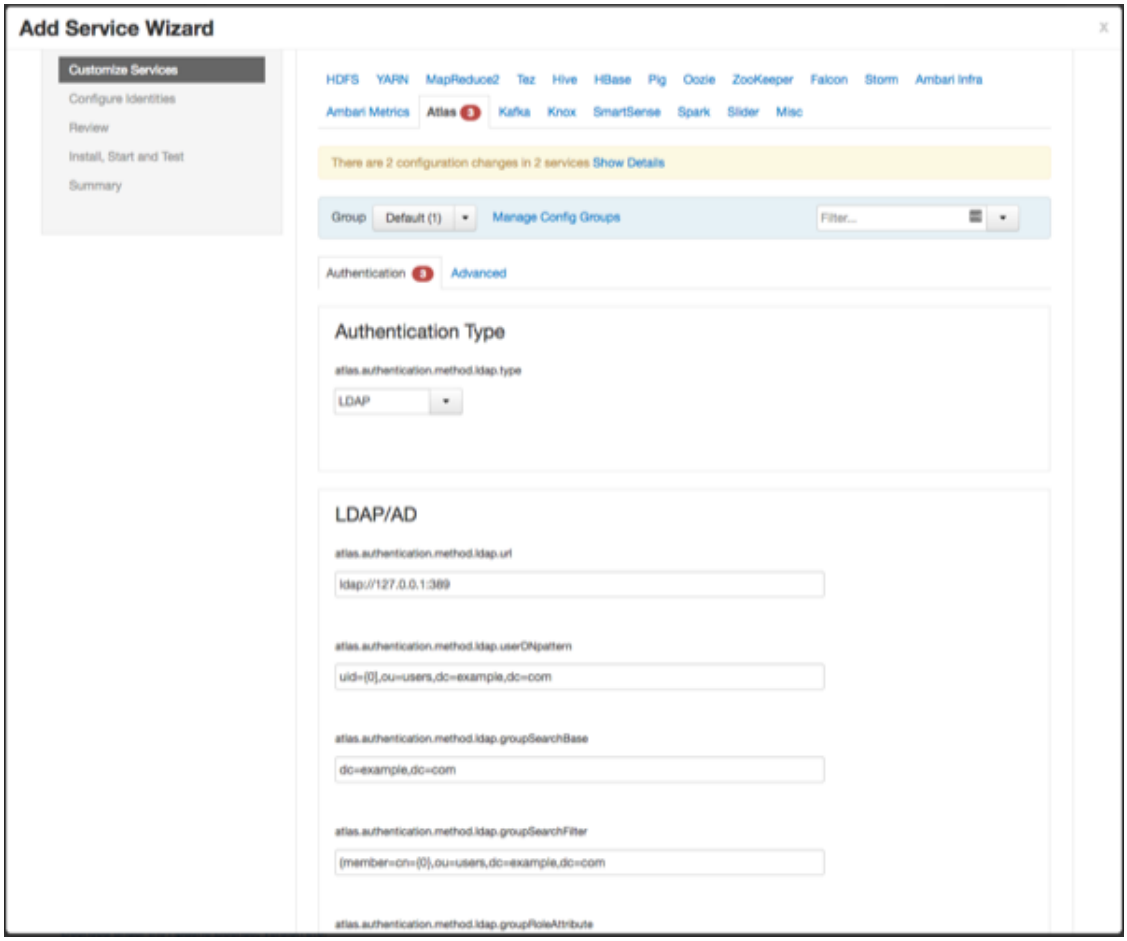
2.1.2.2. LDAP Authentication

To enable LDAP authentication, select **LDAP**, then set the following configuration properties.

Table 2.2. Apache Atlas LDAP Configuration Settings

Property	Sample Values
atlas.authentication.method.ldap.url	ldap://127.0.0.1:389
atlas.authentication.method.ldap.userDNpattern	uid={0],ou=users,dc=example,dc=com

Property	Sample Values
atlas.authentication.method.ldap.groupSearchBase	dc=example,dc=com
atlas.authentication.method.ldap.groupSearchFilter	(member=cn={0},ou=users,dc=example,dc=com
atlas.authentication.method.ldap.groupRoleAttribute	cn
atlas.authentication.method.ldap.base.dn	dc=example,dc=com
atlas.authentication.method.ldap.bind.dn	cn=Manager,dc=example,dc=com
atlas.authentication.method.ldap.bind.password	PassW0rd
atlas.authentication.method.ldap.referral	ignore
atlas.authentication.method.ldap.user.searchfilter	(uid={0})
atlas.authentication.method.ldap.default.role	ROLE_USER



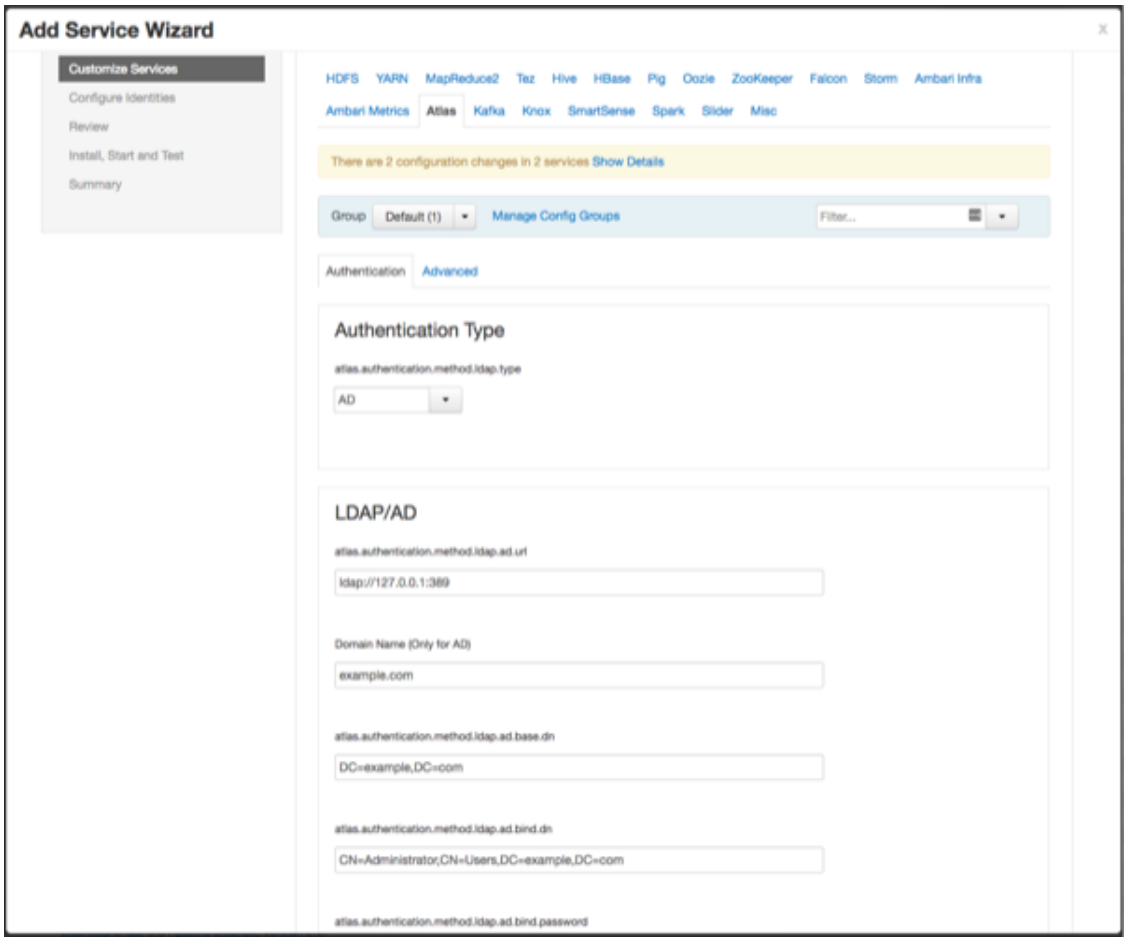
2.1.2.3. AD Authentication

To enable AD authentication, select **AD**, then set the following configuration properties.

Table 2.3. Apache Atlas AD Configuration Settings

Property	Sample Values
atlas.authentication.method.ldap.ad.url	ldap://127.0.0.1:389
Domain Name (Only for AD)	example.com
atlas.authentication.method.ldap.ad.base.dn	DC=example,DC=com

Property	Sample Values
atlas.authentication.method.ldap.ad.bind.dn	CN=Administrator,CN=Users,DC=example,DC=com
atlas.authentication.method.ldap.ad.bind.password	PassW0rd
atlas.authentication.method.ldap.ad.referral	ignore
atlas.authentication.method.ldap.ad.user.searchfilter	(sAMAccountName={0})
atlas.authentication.method.ldap.ad.default.role	ROLE_USER



2.1.3. Authorization Settings

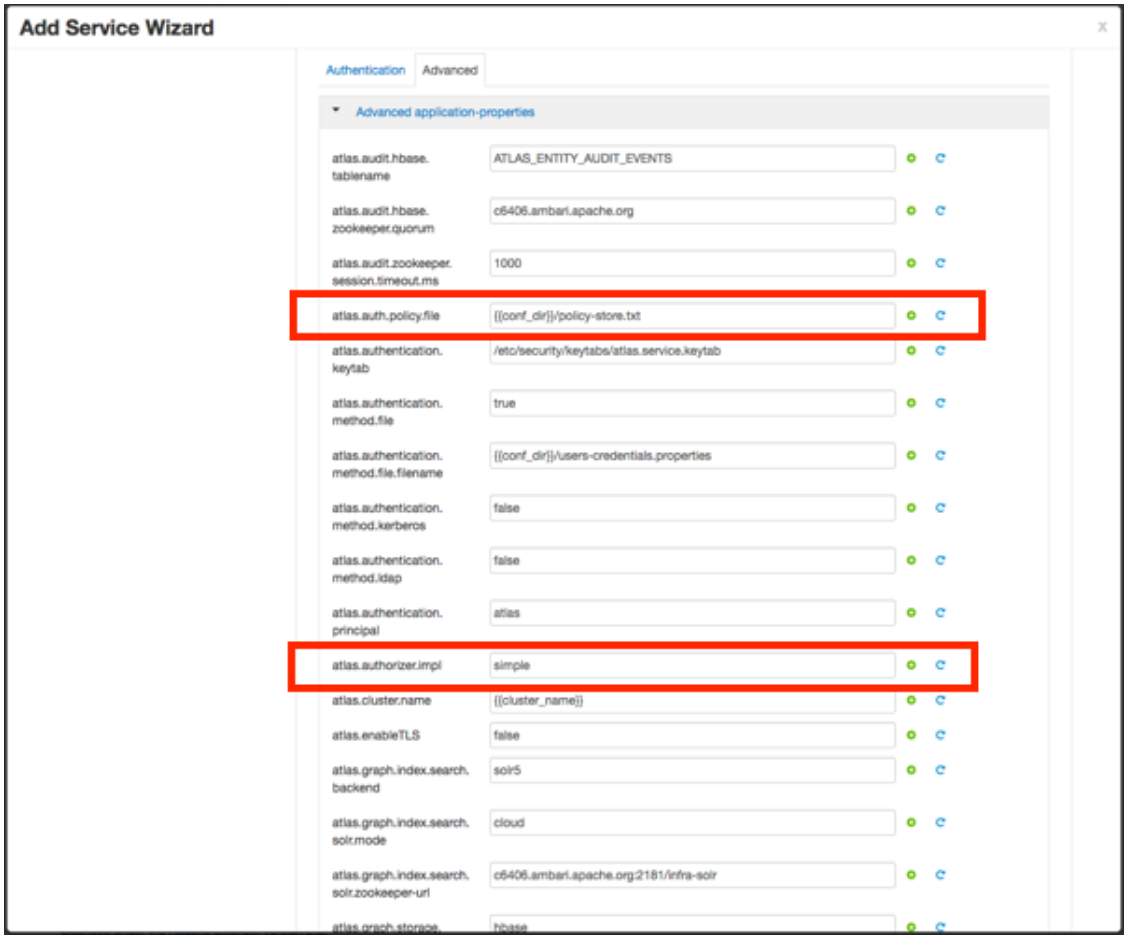
Two authorization methods are available for Atlas: Simple and Ranger.

2.1.3.1. Simple Authorization

The default setting is Simple, and the following properties are automatically set under **Advanced application-properties** on the Advanced tab.

Table 2.4. Apache Atlas Simple Authorization

Property	Value
atlas.authorizer.impl	simple
atlas.auth.policy.file	{{conf_dir}}/policy-store.txt



The `policy-store.txt` file has the following format:

```
Policy_Name;;User_Name:Operations_Allowed;;Group_Name:Operations_Allowed;;Resource_Type:Reso
```

For example:

```
adminPolicy;;admin:rwud;;ROLE_ADMIN:rwud;;type:*,entity:*,operation:*,
taxonomy:*,term:*
userReadPolicy;;readUser1:r,readUser2:r;;DATA_SCIENTIST:r;;type:*,entity:*,
operation:*,taxonomy:*,term:*
userWritePolicy;;writeUser1:rwu,writeUser2:rwu;;BUSINESS_GROUP:rwu,
DATA_STEWARD:rwud;;type:*,entity:*,operation:*,taxonomy:*,term:*
```

In this example `readUser1`, `readUser2`, `writeUser1` and `writeUser2` are the user IDs, each with its corresponding access rights. The `User_Name`, `Group_Name` and `Operations_Allowed` are comma-separated lists.

Authorizer Resource Types:

- Operation
- Type
- Entity
- Taxonomy

- Term
- Unknown

`Operations_Allowed` are `r` = read, `w` = write, `u` = update, `d` = delete

2.1.3.2. Ranger Authorization

Ranger Authorization is activated by enabling the Ranger Atlas plug-in in Ambari.

2.2. Configuring Atlas Tagsync in Ranger

Note

Before configuring Atlas Tagsync in Ranger, you must enable Ranger Authorization in Atlas by enabling the Ranger Atlas plug-in in Ambari.

For information about configuring Atlas Tagsync in Ranger, see Configure Ranger Tagsync.

2.3. Configuring Atlas High Availability

For information about configuring High Availability (HA) for Apache Atlas, see Apache Atlas High Availability.

2.4. Configuring Atlas Security

2.4.1. Additional Requirements for Atlas with Ranger and Kerberos

Currently additional configuration steps are required for Atlas with Ranger and in Kerberized environments.

2.4.1.1. Additional Requirements for Atlas with Ranger

When Atlas is used with Ranger, perform the following additional configuration steps:

- Create the following HBase policy:
  - table: atlas_titan, ATLAS_ENTITY_AUDIT_EVENTS

    user: atlas

    permission: Read, Write, Create, Admin
- Create following Kafka policies:
  - topic=ATLAS_HOOK

    permission=publish, create; group=public

    permission=consume, create; user=atlas (for non-kerberized environments, set group=public)

- topic=ATLAS_ENTITIES

  permission=publish, create; user=atlas (for non-kerberized environments, set group=public)

  permission=consume, create; group=public

2.4.1.2. Additional Requirements for Atlas with Kerberos without Ranger

When Atlas is used in a Kerberized environment without Ranger, perform the following additional configuration steps:

- Start the HBase shell with the user identity of the HBase admin user ('hbase')
- Execute the following command in HBase shell, to enable Atlas to create necessary HBase tables:
  - grant 'atlas', 'RWXCA'
- Start (or restart) Atlas, so that Atlas would create above HBase tables
- Execute the following command in HBase shell, to revoke global permissions granted to 'atlas' user:
  - revoke 'atlas'
- Execute the following commands in HBase shell, to enable Atlas to access necessary HBase tables:
  - grant 'atlas', 'RWXCA', 'atlas_titan'
  - grant 'atlas', 'RWXCA', 'ATLAS_ENTITY_AUDIT_EVENTS'
- Kafka – To grant permissions to a Kafka topic, run the following commands as the Kafka user:

```
/usr/hdp/current/kafka-broker/bin/kafka-acls.sh --topic ATLAS_HOOK --allow-
principals * --operations All --authorizer-properties "zookeeper.connect=
hostname:2181"
/usr/hdp/current/kafka-broker/bin/kafka-acls.sh --topic ATLAS_ENTITIES --
allow-principals * --operations All --authorizer-properties "zookeeper.
connect=hostname:2181"
```

2.4.2. Enabling Atlas HTTPS

For information about enabling HTTPS for Apache Atlas, see Enable SSL for Apache Atlas.

2.4.3. Hive CLI Security

If you have Oozie, Storm, or Sqoop Atlas hooks enabled, the Hive CLI can be used with these components. You should be aware that the Hive CLI may not be secure without taking additional measures.

2.5. Installing Sample Atlas Metadata

You can use the `quick_start.py` Python script to install sample metadata to view in the Atlas web UI. Use the following steps to install the sample metadata:

1. Log in to the Atlas host server using a command prompt.
2. Run the following command as the Atlas user:

   ```
   su atlas -c '/usr/hdp/current/atlas-server/bin/quick_start.py'
   ```

   When prompted, type in the Atlas user name and password. When the script finishes running, the following confirmation message appears:

   ```
   Example data added to Apache Atlas Server!!!
   ```

   If Kerberos is enabled, `kinit` is required to execute the `quick_start.py` script.

After you have installed the sample metadata, you can explore the Atlas web UI.

Note

If you are using the HDP Sandbox, you do not need to run the Python script to populate Atlas with sample metadata.

2.6. Updating the Atlas Ambari Configuration

When you update the Atlas configuration settings in Ambari, Ambari marks the services that require restart, and you can select **Actions > Restart All Required** to restart all services that require a restart.

Important

Apache Oozie requires a restart after an Atlas configuration update, but may not be included in the services marked as requiring restart in Ambari. Select **Oozie > Service Actions > Restart All** to restart Oozie along with the other services.

2.7. Using Distributed HBase as the Atlas Metastore

Apache HBase can be configured to run in stand-alone and distributed mode. The Atlas Ambari installer uses the stand-alone Ambari HBase instance as the Atlas Metastore by default. The default stand-alone HBase configuration should work well for POC (Proof of Concept) deployments, but you should consider using distributed HBase as the Atlas Metastore for production deployments. Distributed HBase also requires a ZooKeeper quorum.

Use the following steps to configure Atlas for distributed HBase.

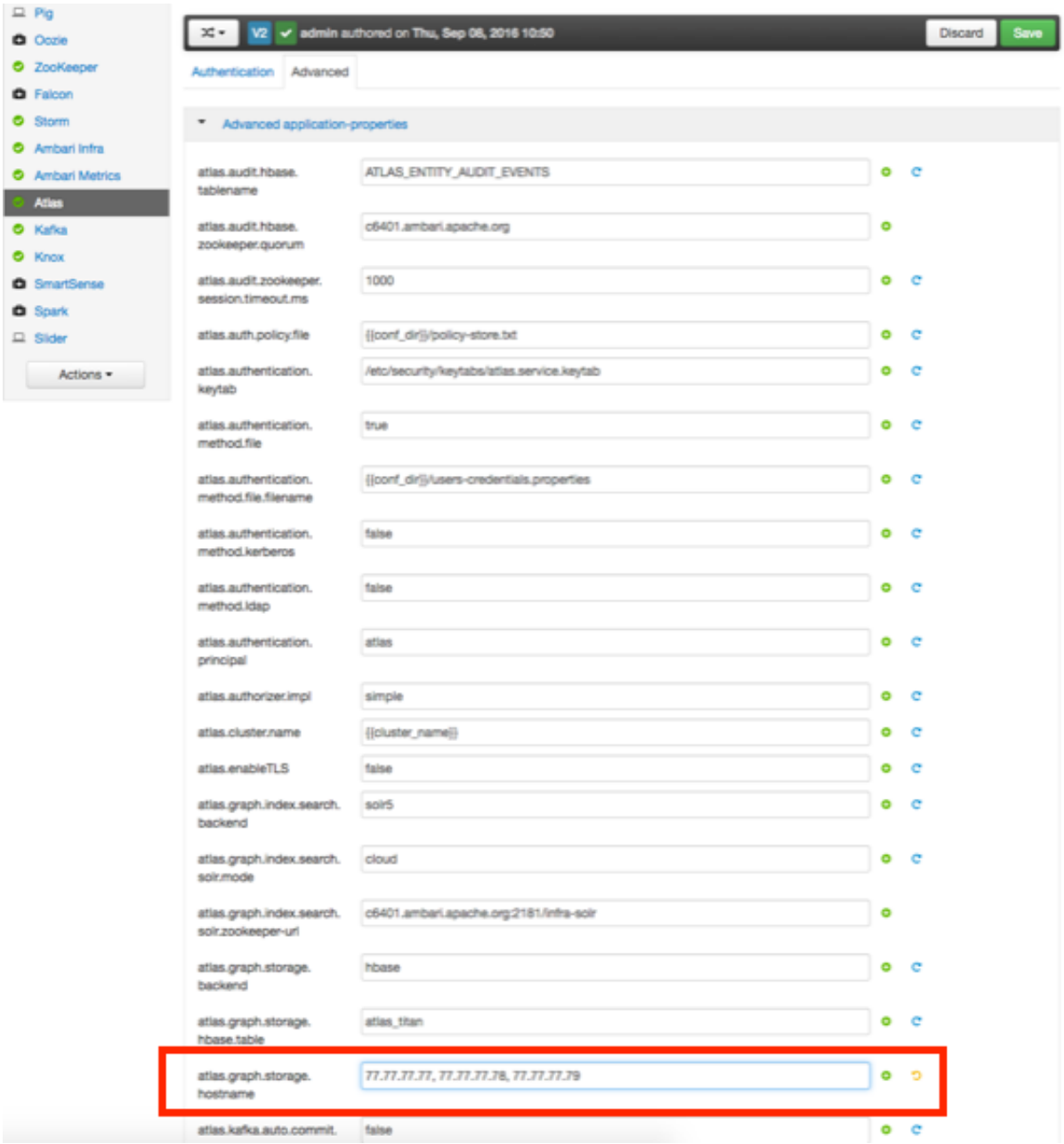
Note

This procedure does not represent a migration of the Graph Database, so any existing lineage reports will be lost.

1. On the Ambari dashboard, select **Atlas > Configs > Advanced**, then select **Advanced application-properties**.

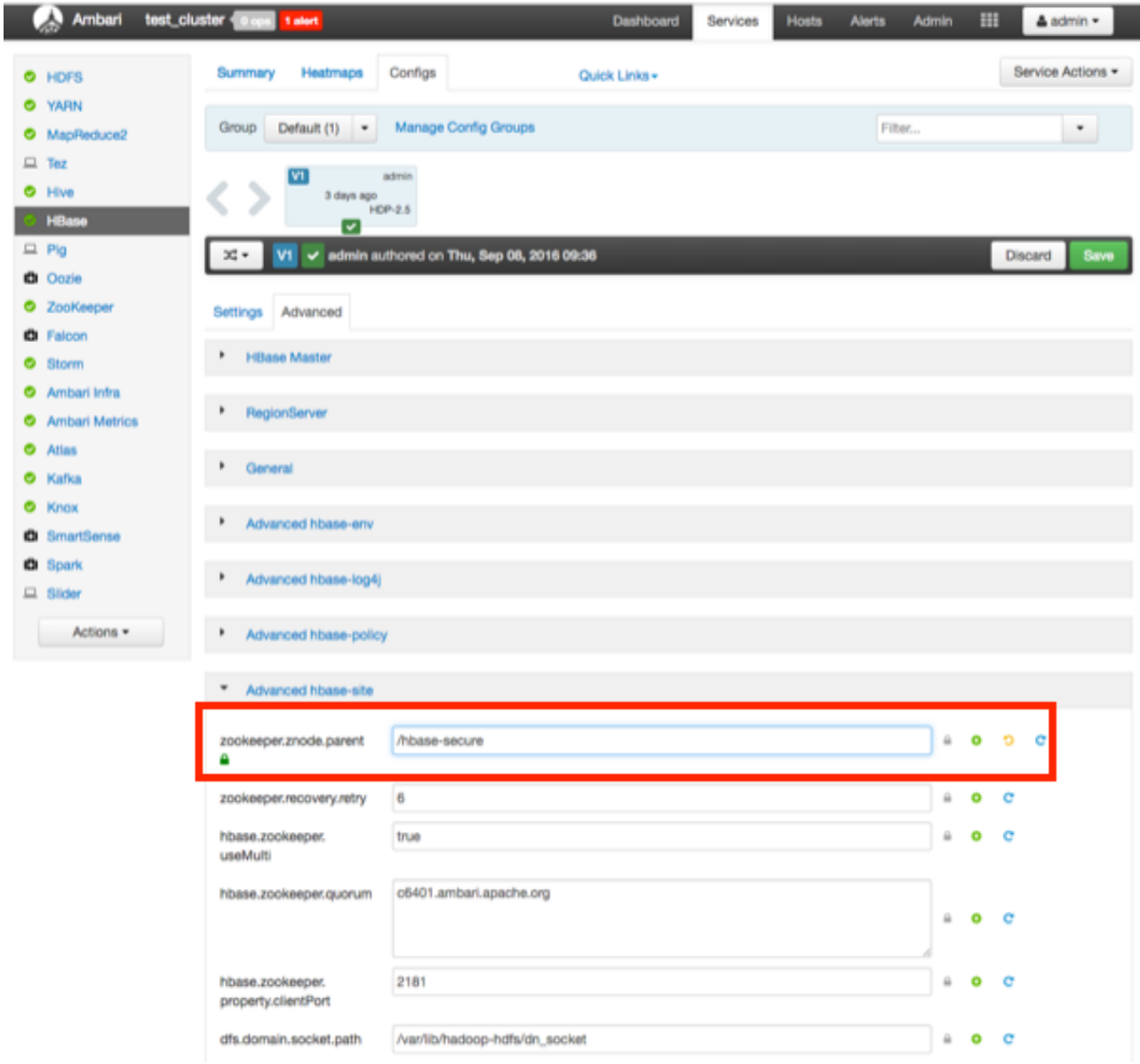
2. Set the value of the `atlas.graph.storage.hostname` property to the value of the distributed HBase ZooKeeper quorum. This value is a comma-separated list of the servers in the distributed HBase ZooKeeper quorum:

```
host1.mydomain.com,host2.mydomain.com,host3.mydomain.com
```



3. Click **Save** to save your changes, then restart Atlas and all other services that require a restart. As noted previously, Oozie requires a restart after an Atlas configuration change (even if it is not marked as requiring a restart).

4. If HBase is running in secure mode, select **HBase > Configs > Advanced** on the Ambari dashboard, then select **Advanced hbase-site**. Set the value of the `zookeeper.znode.parent` property to `/hbase-secure` (if HBase is not running in secure mode, you can leave this property set to the default `/hbase-unsecure` value).



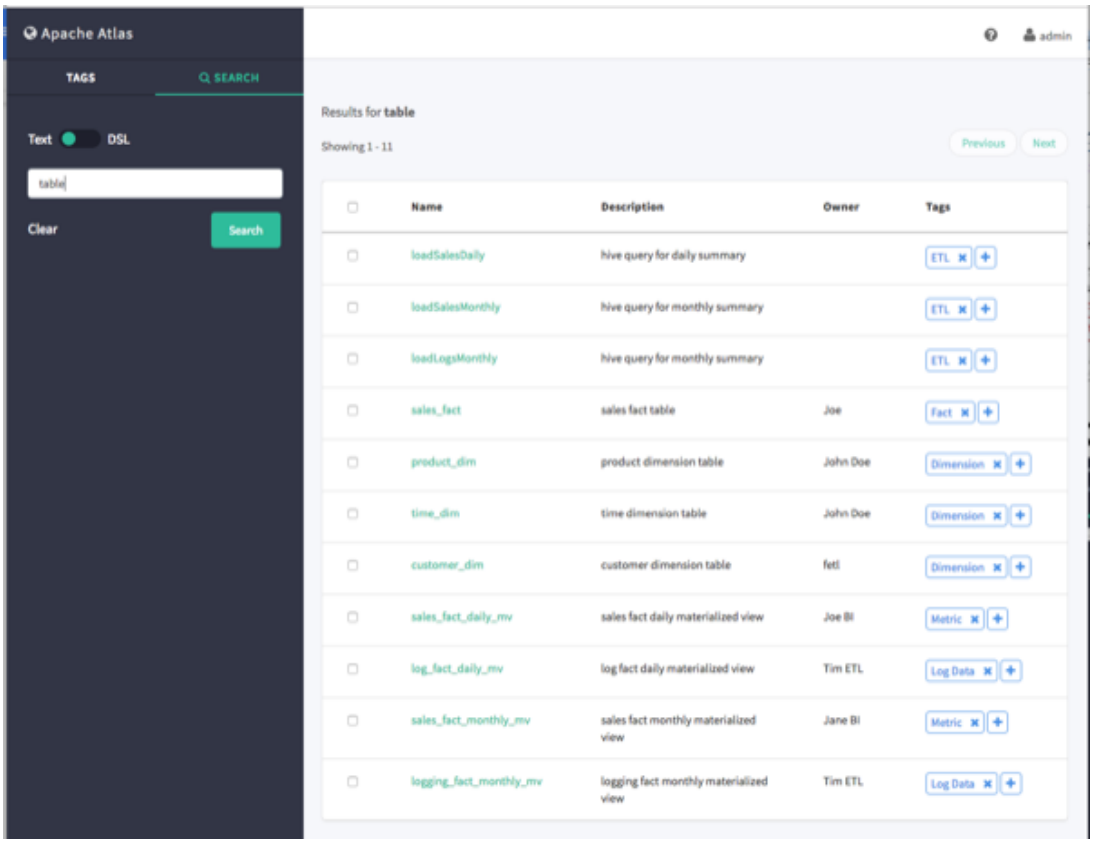
5. Click **Save** to save your changes, then restart HBase and all other services that require a restart.

3. Searching and Viewing Assets

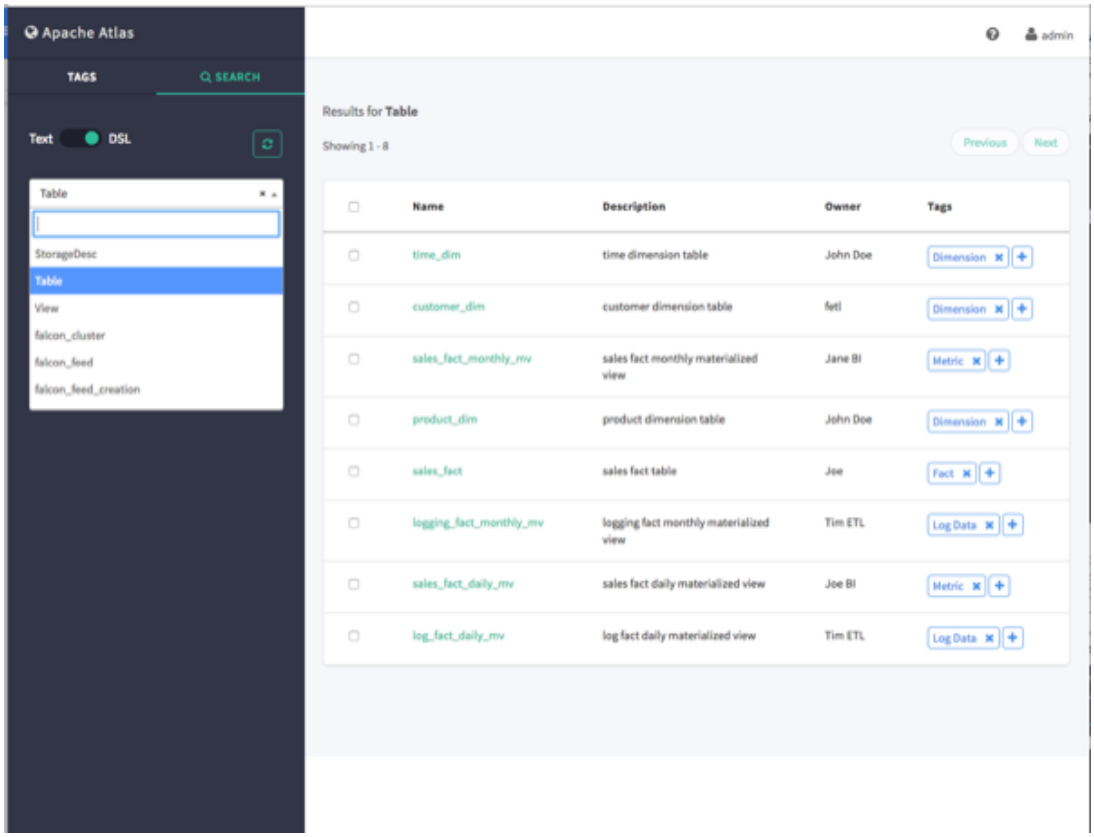
3.1. Using Text and DSL Search

You can search for assets using two search modes: Text or DSL. Text is a full-text search, and DSL enables you to search using Apache Atlas DSL. Atlas DSL (Domain-Specific Language) is a SQL-like query language that enables you to search metadata using complex queries.

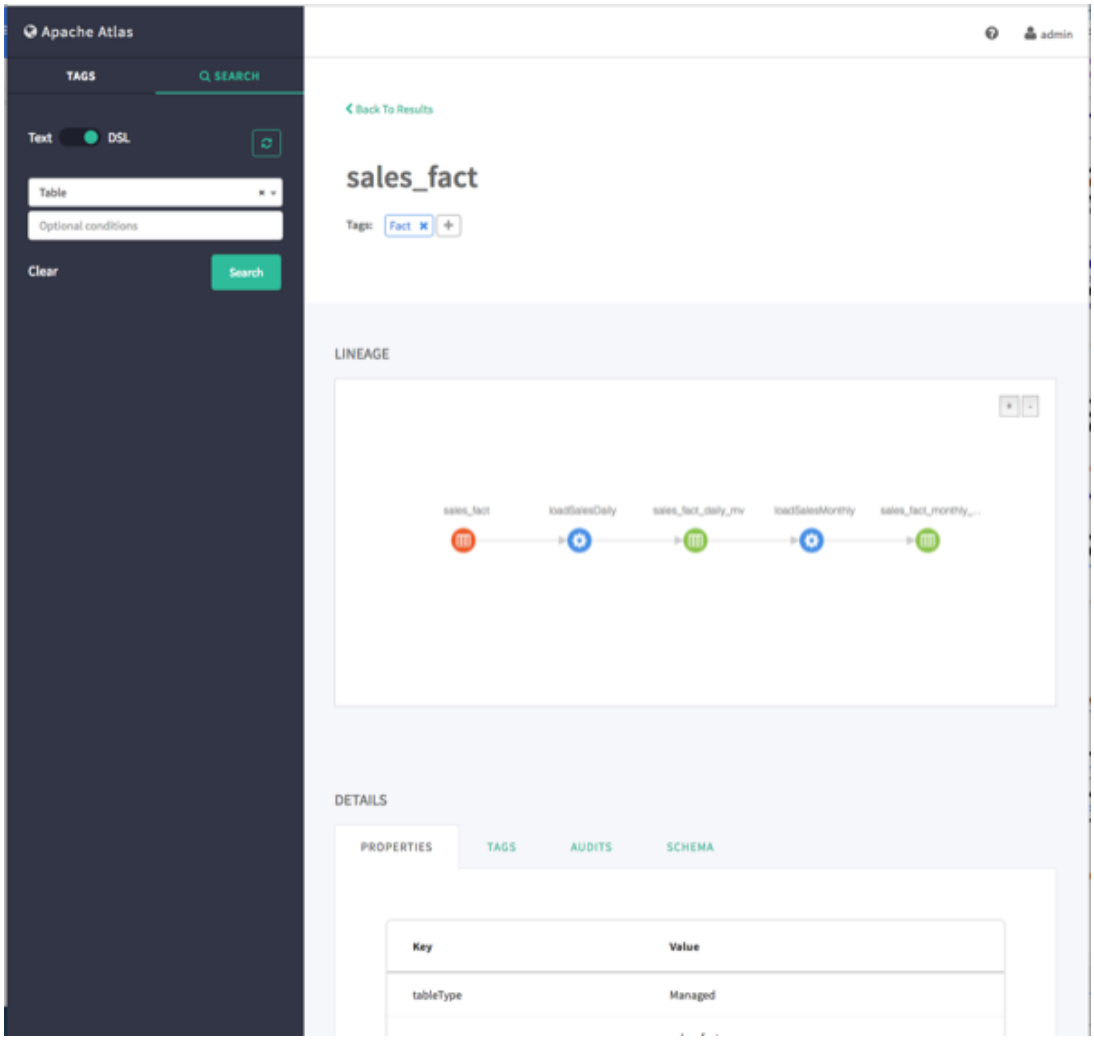
1. To search for assets, click **SEARCH** on the Atlas web UI, select **Text**, type in a search string, then click **Search** to display a list of the assets associated with that tag. In the example below, we searched for the text string "table".



You can also select the **DSL** search option and use the Search box to select a pre-configured DSL query. You can use the Optional Conditions box to enter additional DSL query parameters. In the example below, we selected the `Table` DSL query:

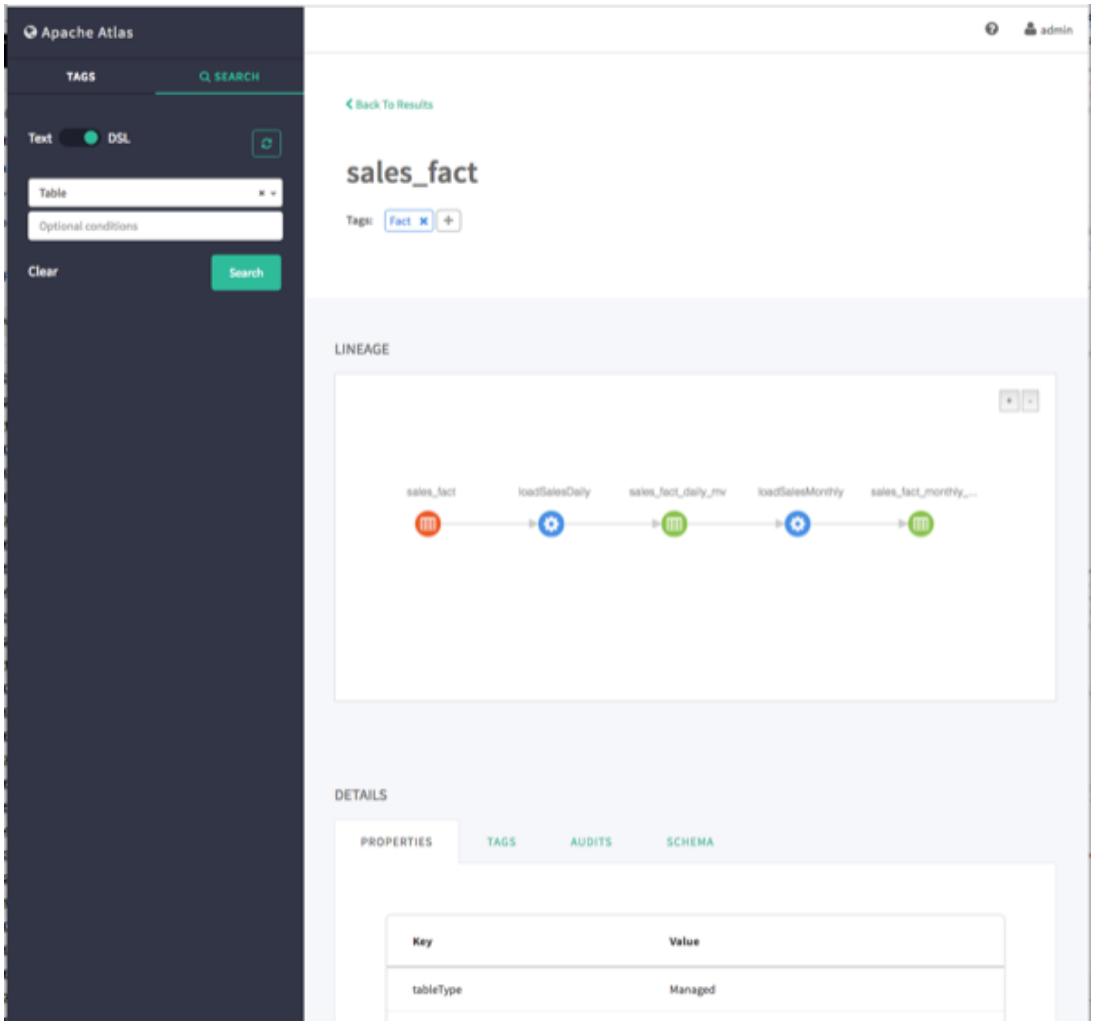


2. To view detailed information about an asset, click the asset in the search results list. In the example below, we selected the "sales_fact" table from the list of search results.

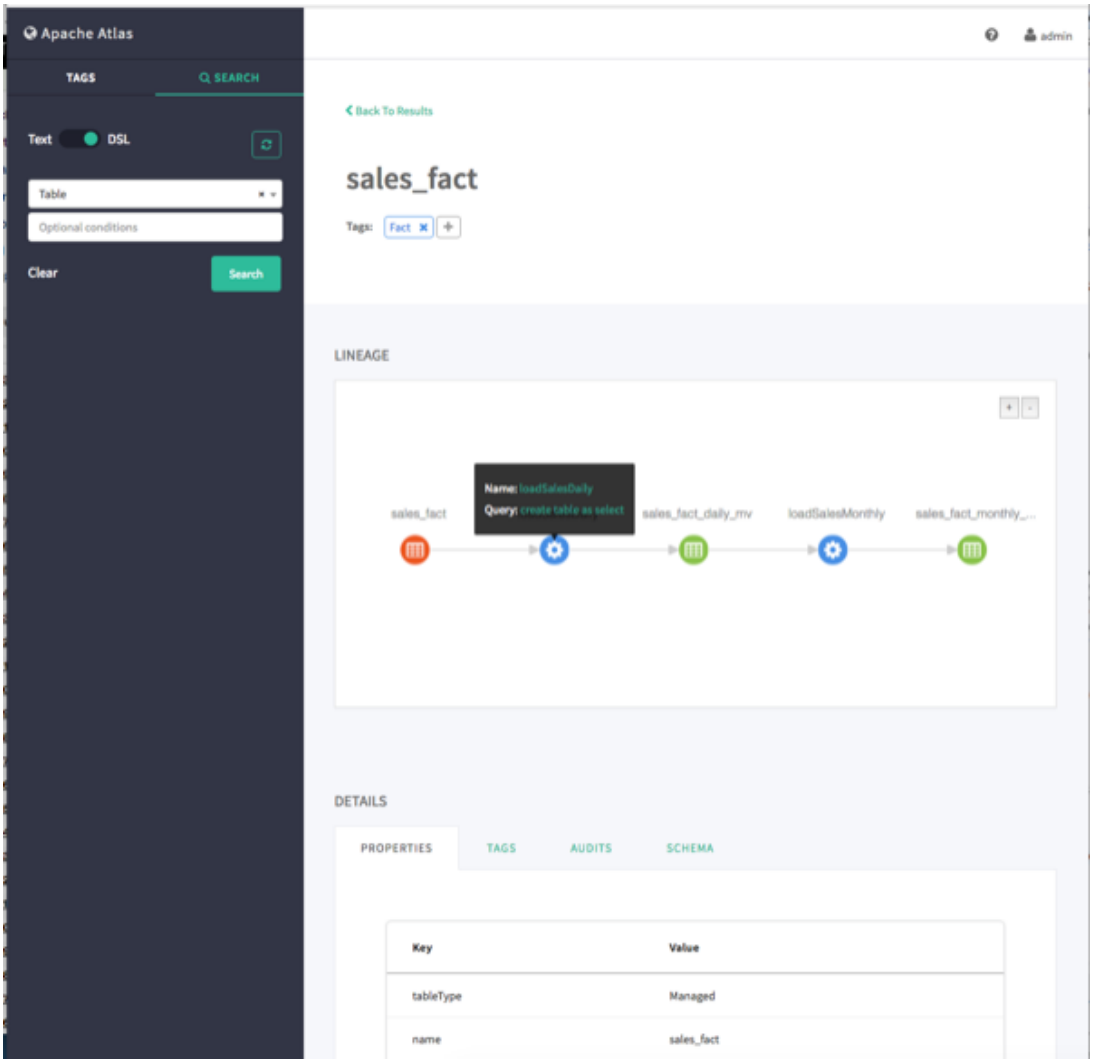


3.2. Viewing Asset Data Lineage

1. Data lineage is displayed when you select an asset. In the following example, we ran a DSL search for `Table`, and then selected the "sales_fact" asset. Data lineage is displayed graphically, with each icon representing an action. You can use the + and - buttons to zoom in and out, and you can also click and drag to move the image.



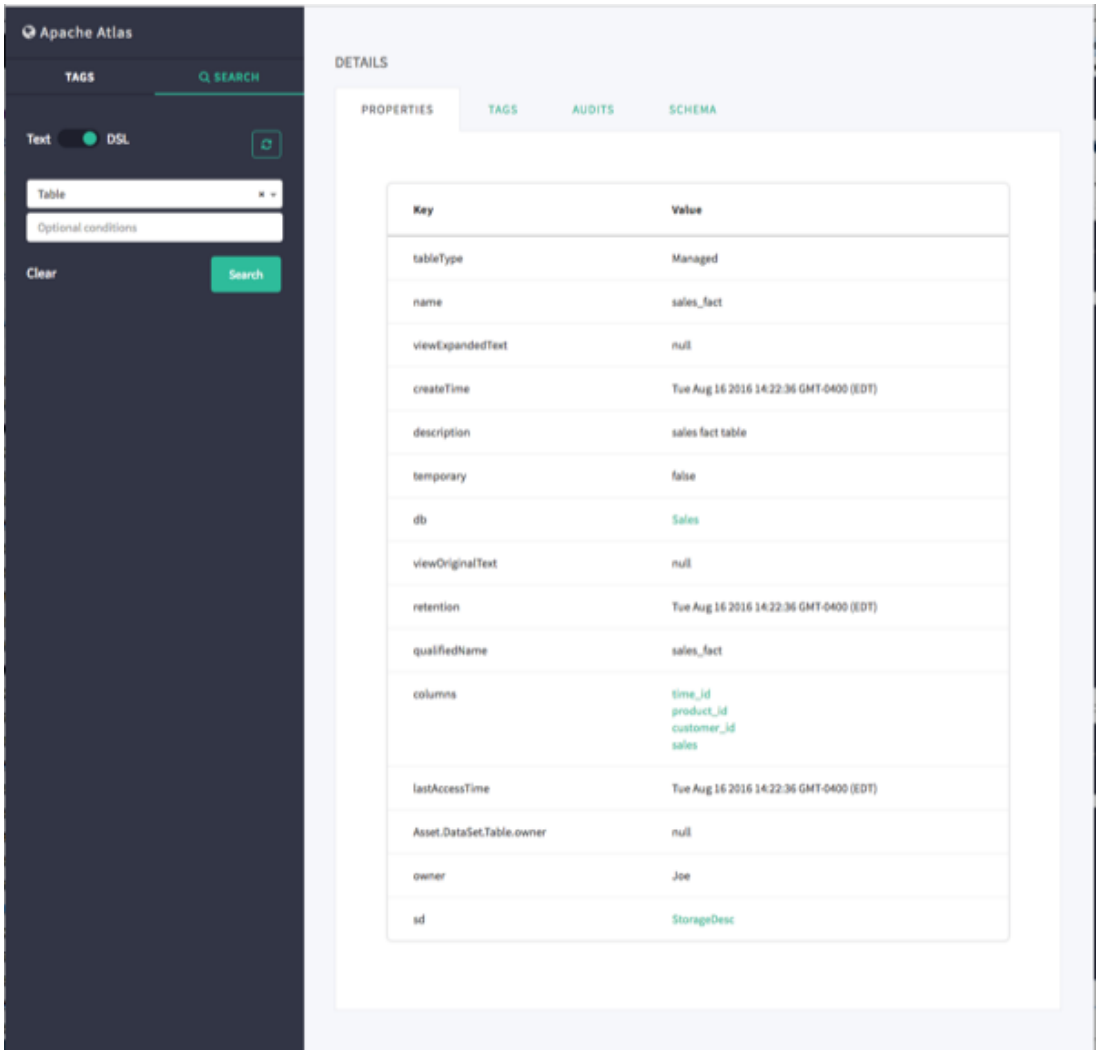
2. Moving the cursor over an icon displays a pop-up with more information about the action that was performed. In the following example, we can see that a query was used to create the "loadSalesDaily" table from the "sales_fact" table.



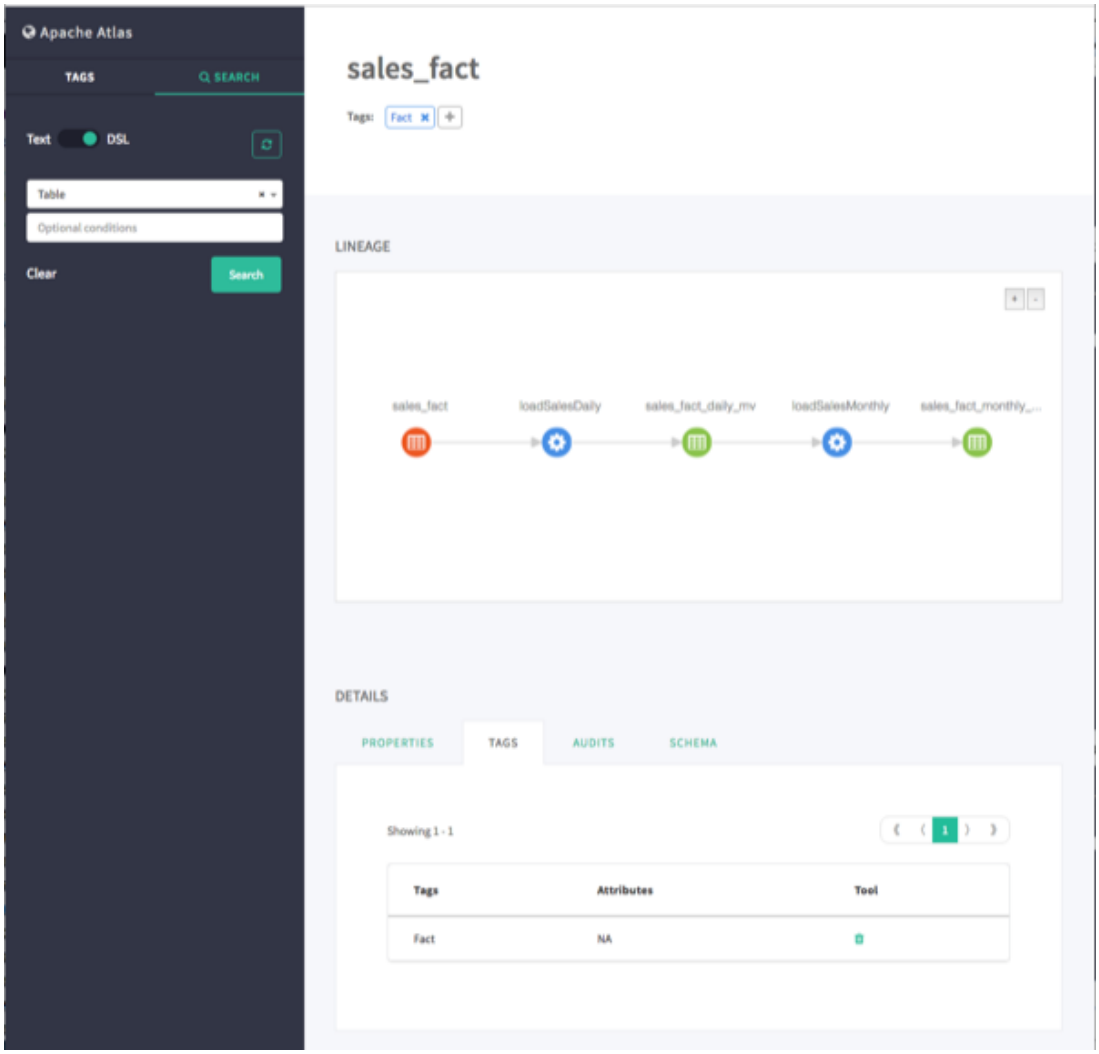
3.3. Viewing Asset Details

When you select an asset, detailed information about the asset is displayed under DETAILS.

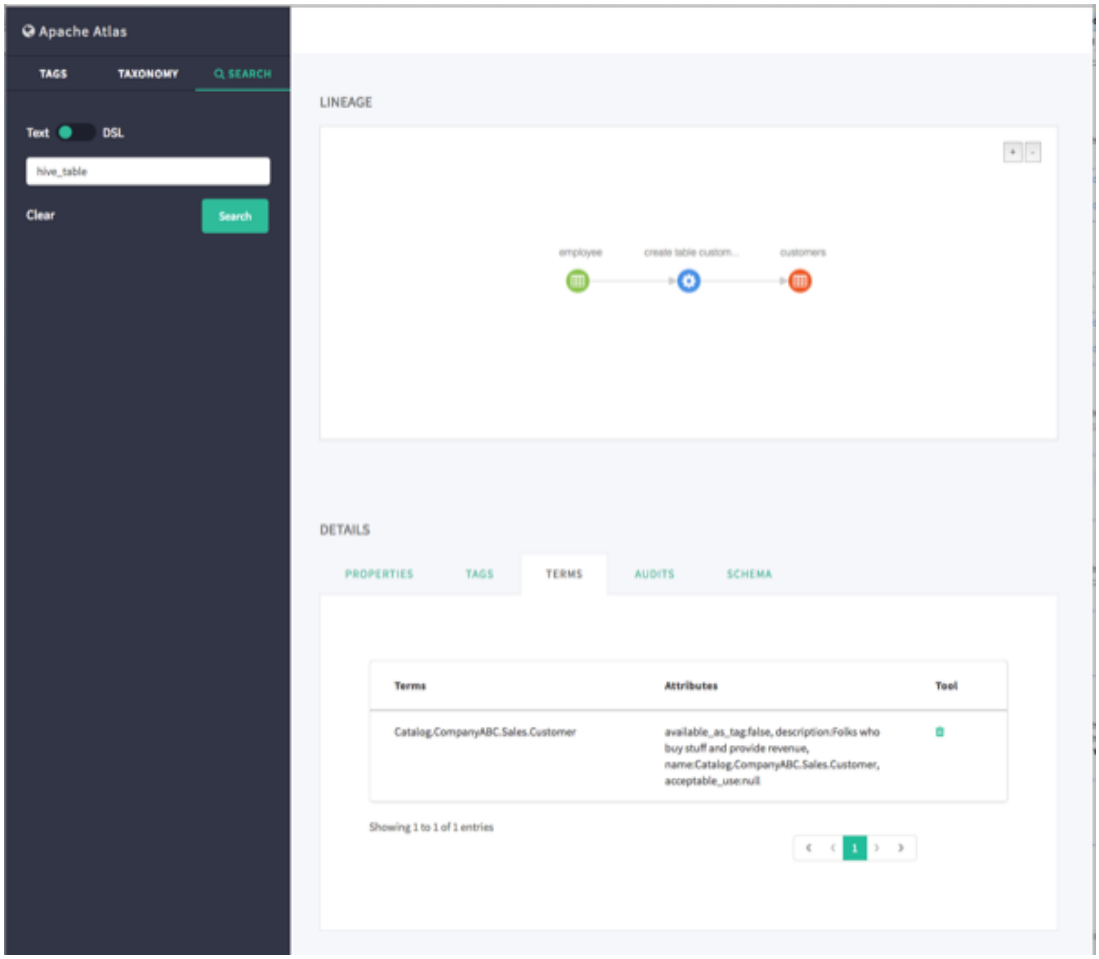
- The Properties tab displays all of the asset properties.



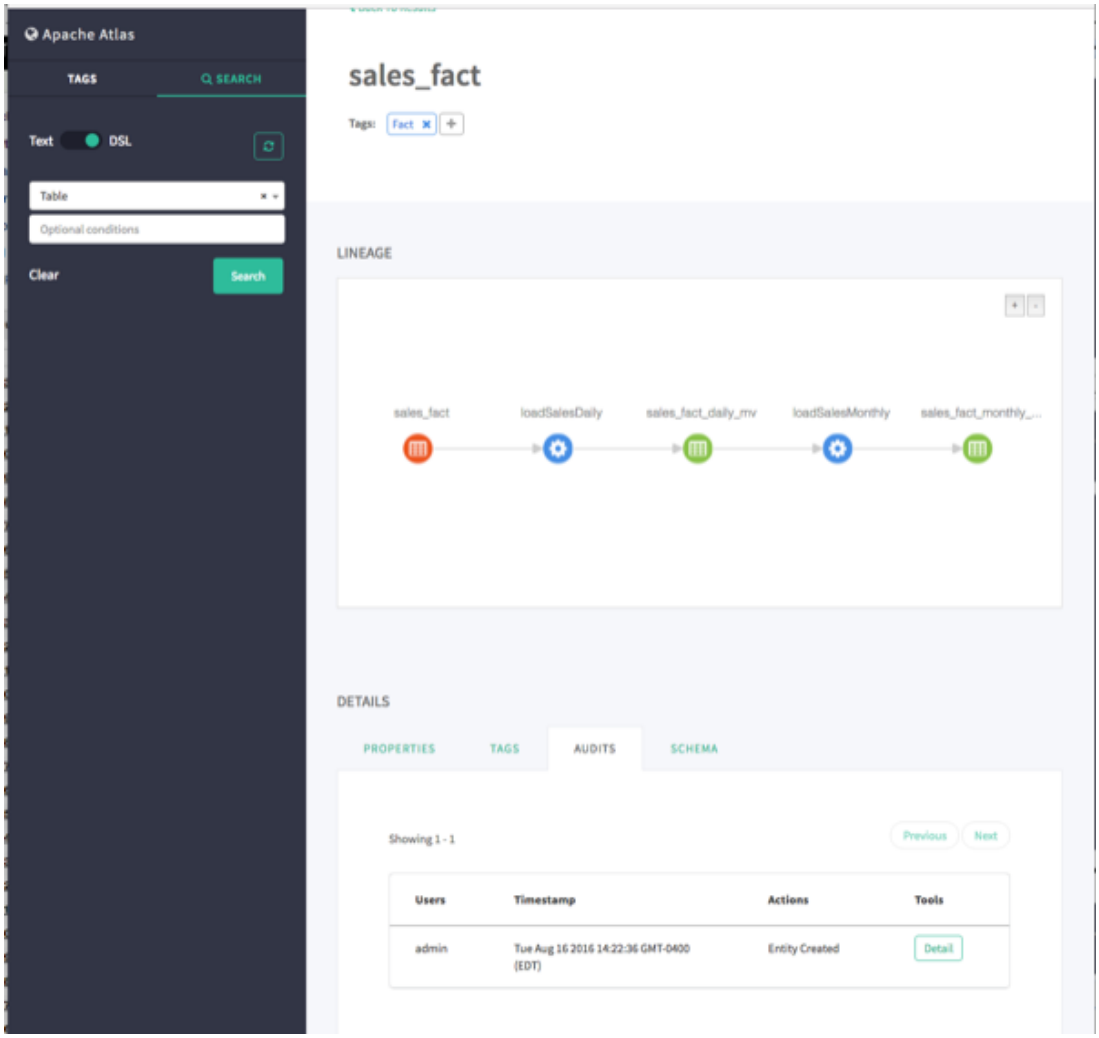
- Click the Tags tab to display the tags associated with the asset. In this case, the "fact" tag has been associated with the "sales_fact" table.



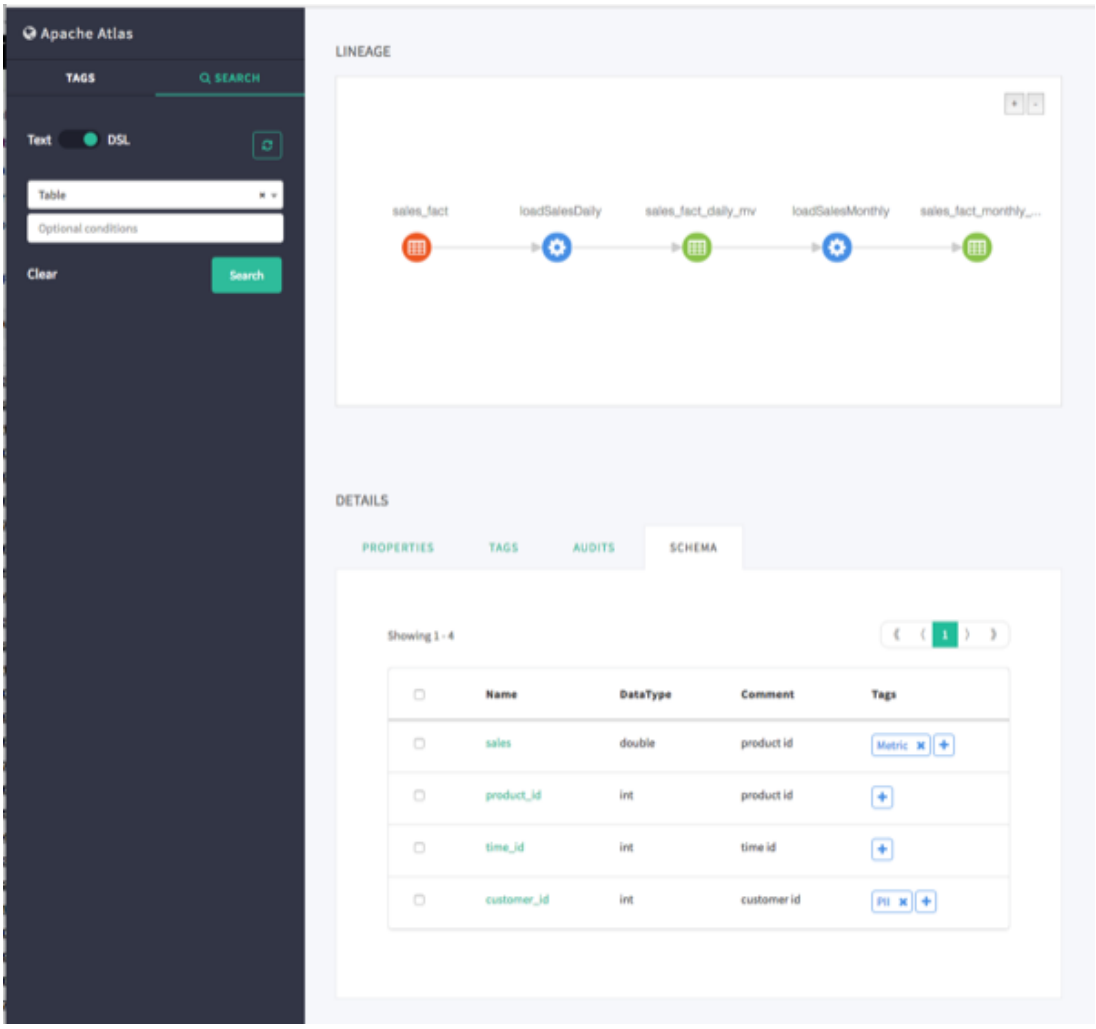
- If the Atlas Taxonomy has been enabled, the Terms tab lists the taxonomy terms that have been associated with the asset. The Terms tab is not displayed if the Taxonomy has not been enabled.



- The Audits tab provides a complete audit trail of all events in the asset's history (assets are also sometimes referred to as entities). You can use the Detail button next to each action to view more details about the event.



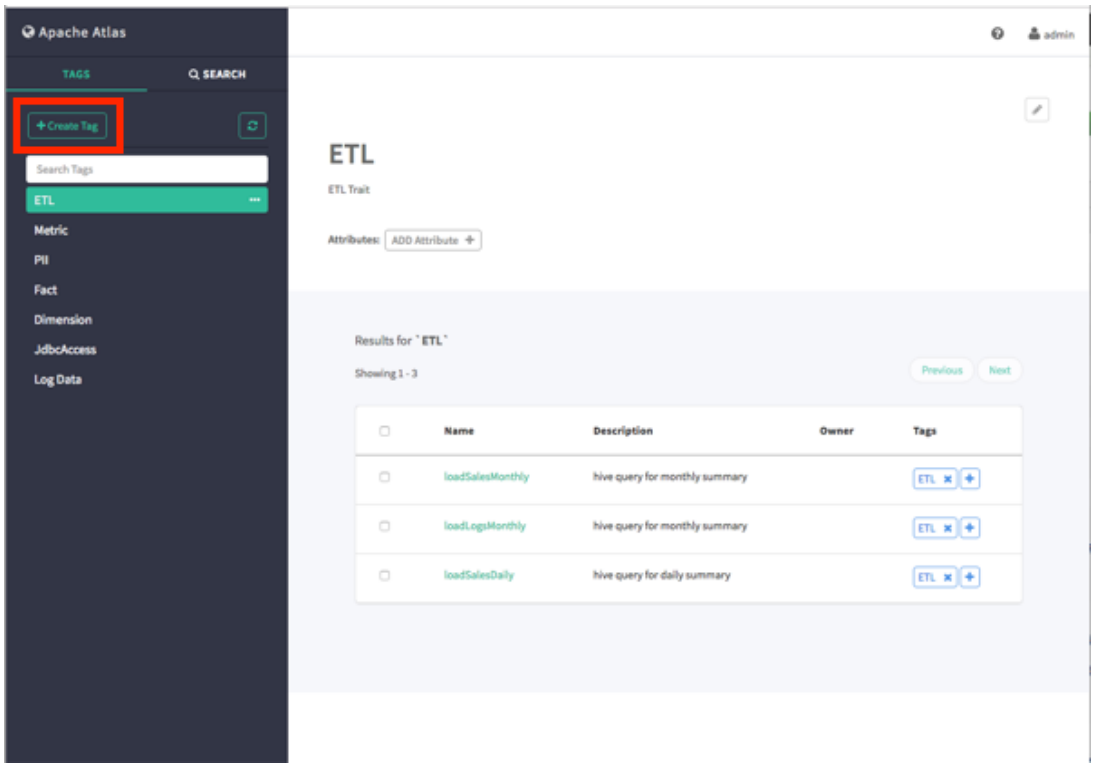
- The Schema tab shows schema information, in this case the columns for the table. We can also see that a PII tag has been associated with the "customer_id" column.


Apache Atlas
TAGS
SEARCH
Text
DSL
Table
Optional conditions
Clear
Search
LINEAGE
sales_fact
loadSalesDaily
sales_fact_daily_mv
loadSalesMonthly
sales_fact_monthly_...
DETAILS
PROPERTIES
TAGS
AUDITS
SCHEMA
Showing 1 - 4
Name
DataType
Comment
Tags
sales
double
product id
Metric
product_id
int
product id
time_id
int
time id
customer_id
int
customer id
PII

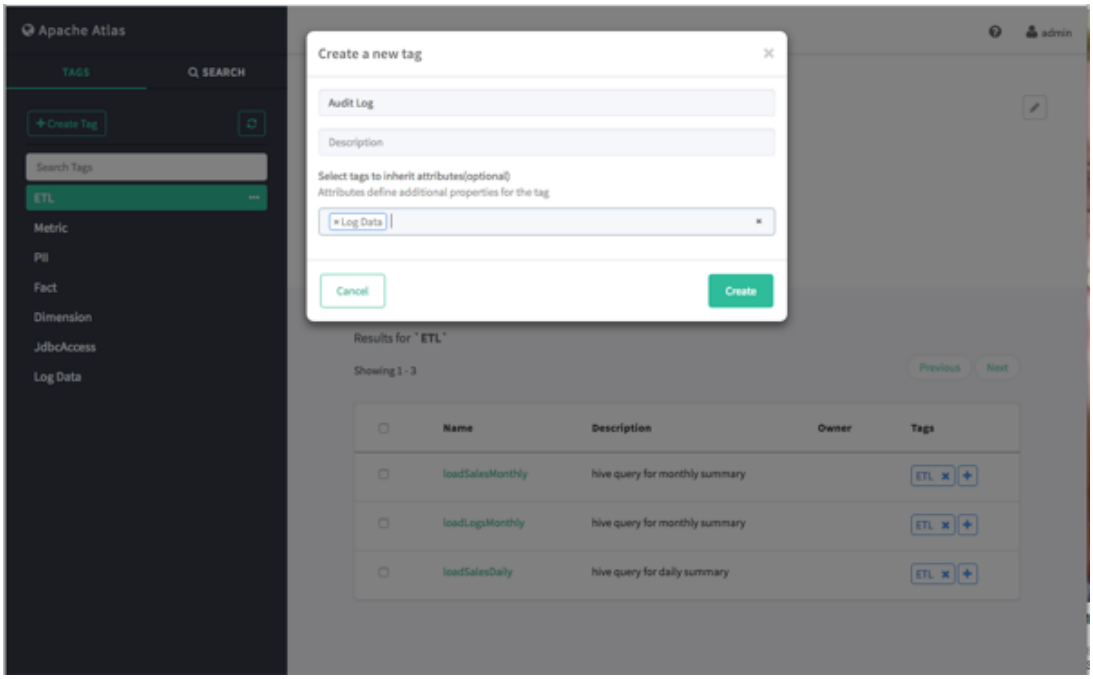
4. Working with Atlas Tags

4.1. Creating Atlas Tags

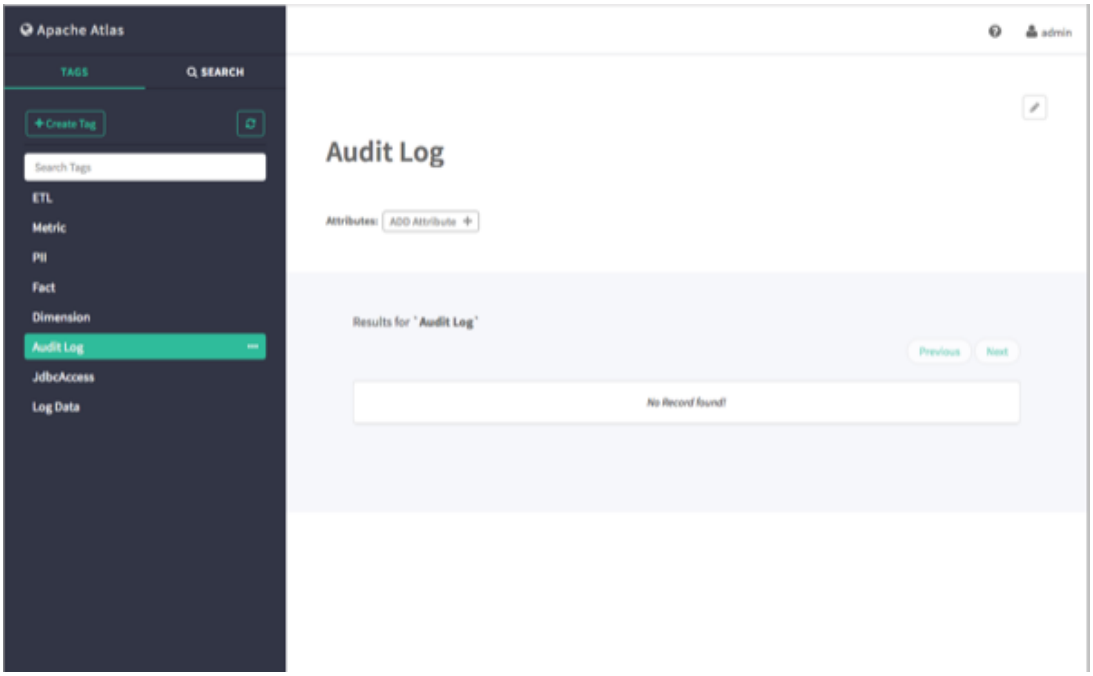
1. On the Atlas web UI, click **TAGS**, then click **Create Tag**.



2. On the Create a New Tag pop-up, type in a name and an optional description for the tag. You can also use the **Select tags to inherit attributes** box to inherit attributes from other tags. Click **Create** to create the new Tag.

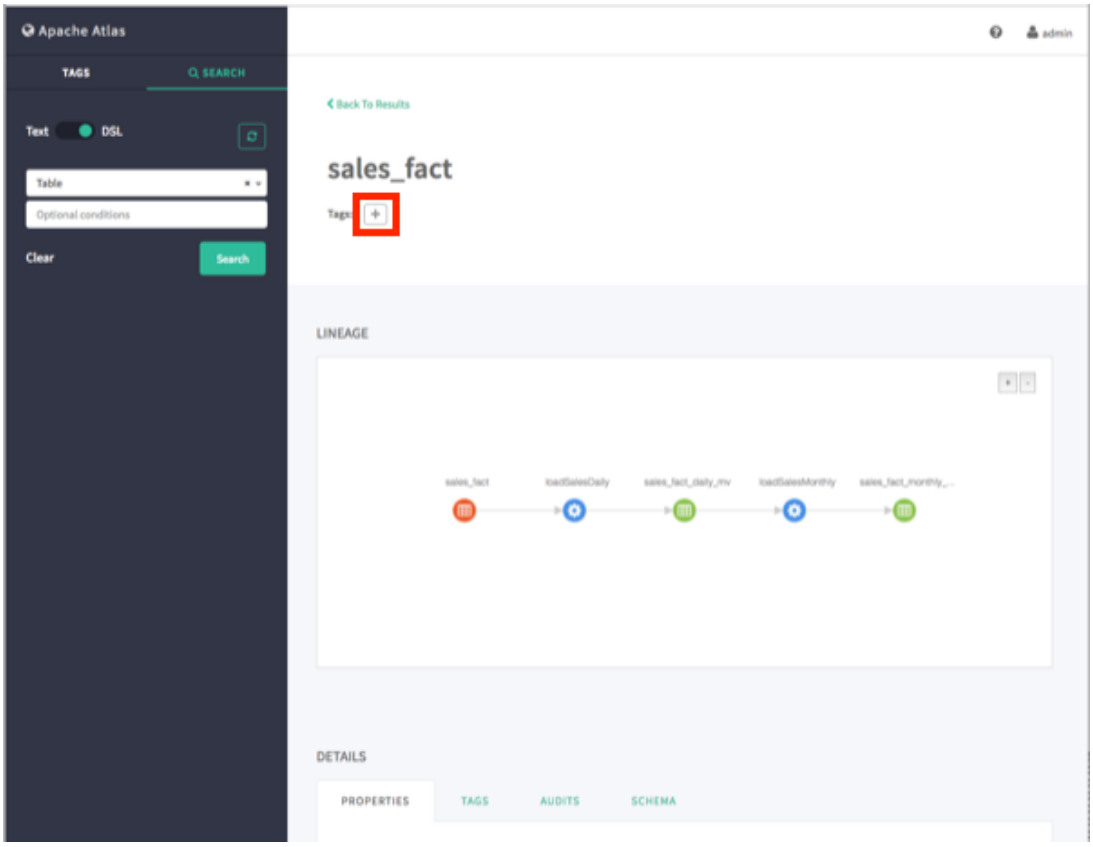


3. The new tag appears in the Tags list.

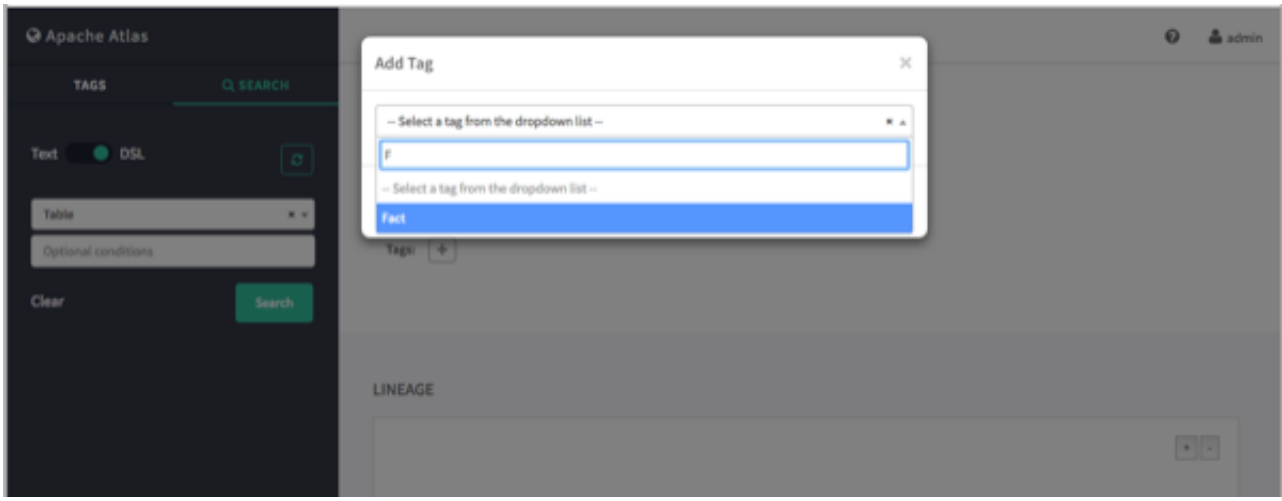


4.2. Associating Tags with Assets

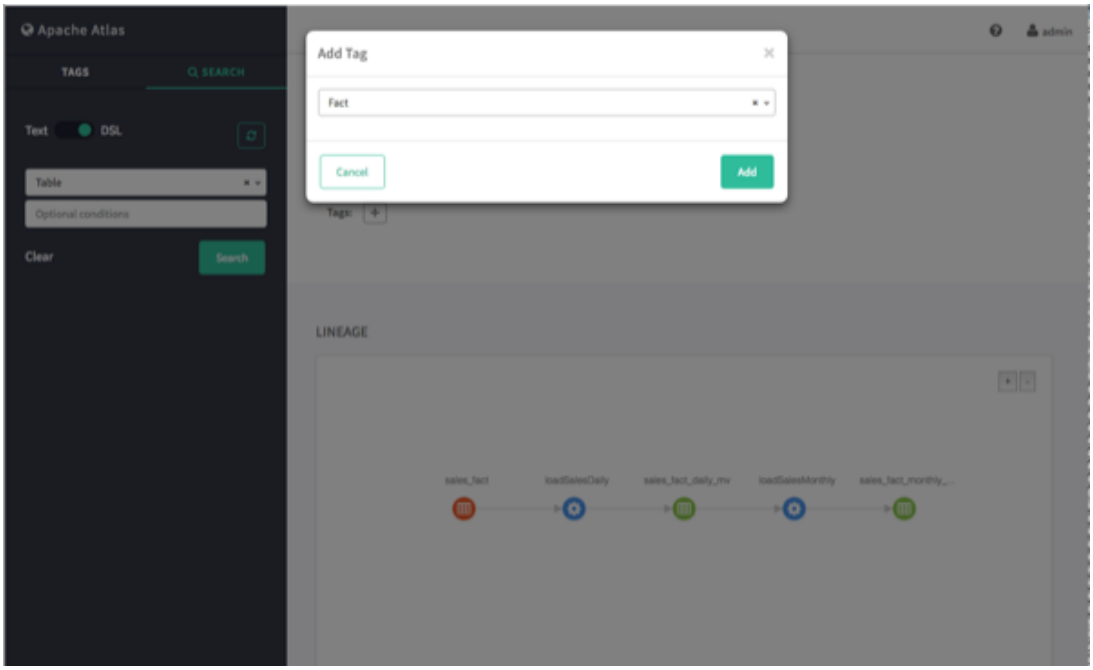
1. Select an asset. In the example below, we searched for all `Table` assets, and then selected the "sales_fact" table from the list of search results. To associate a tag with an asset, click the + icon next to the **Tags:** label.



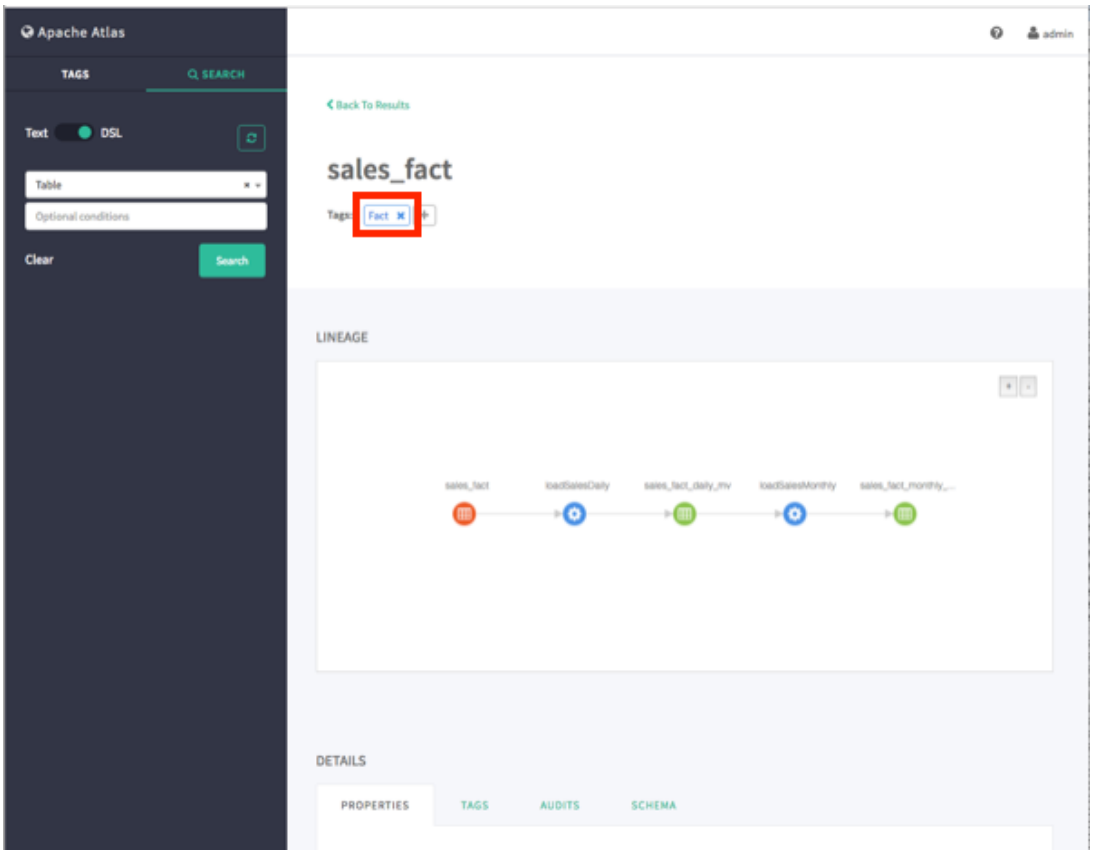
2. On the Add Tag pop-up, click **Select Tag**, then select the tag you would like to associate with the asset. You can filter the list of tags by typing text in the Select Tag box.



3. After you select a tag, the Add Tag pop-up is redisplayed with the selected tag. Click **Save** to associate the tag with the asset.

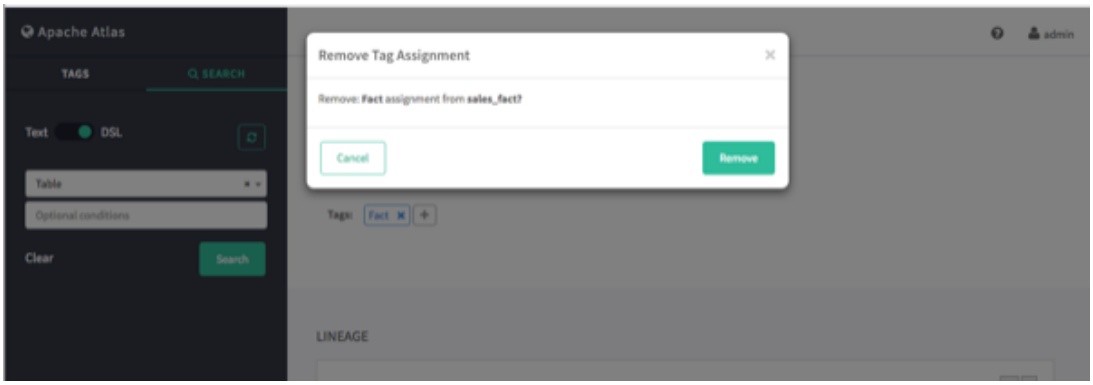


4. The new tag is displayed next to the **Tags:** label on the asset page.



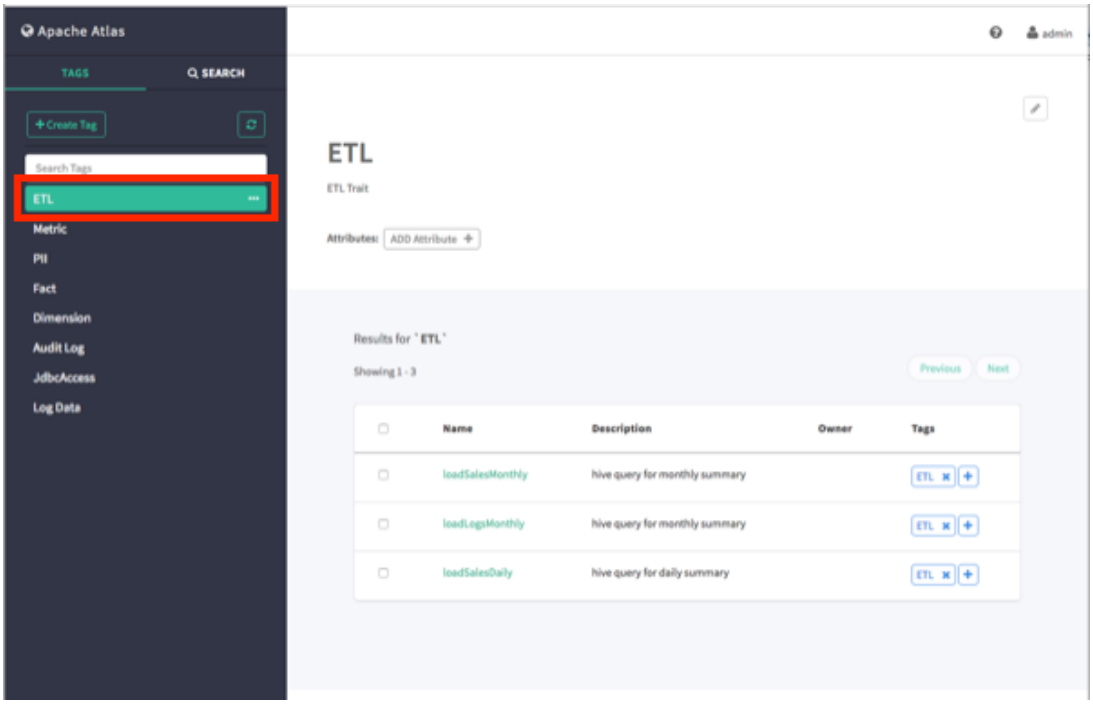
5. You can view details about a tag by clicking the tag name on the tag label.

To remove a tag from an asset, click the **x** symbol on the tag label, then click **Remove** on the confirmation pop-up. This removes the tag association with the asset, but does not delete the tag itself.

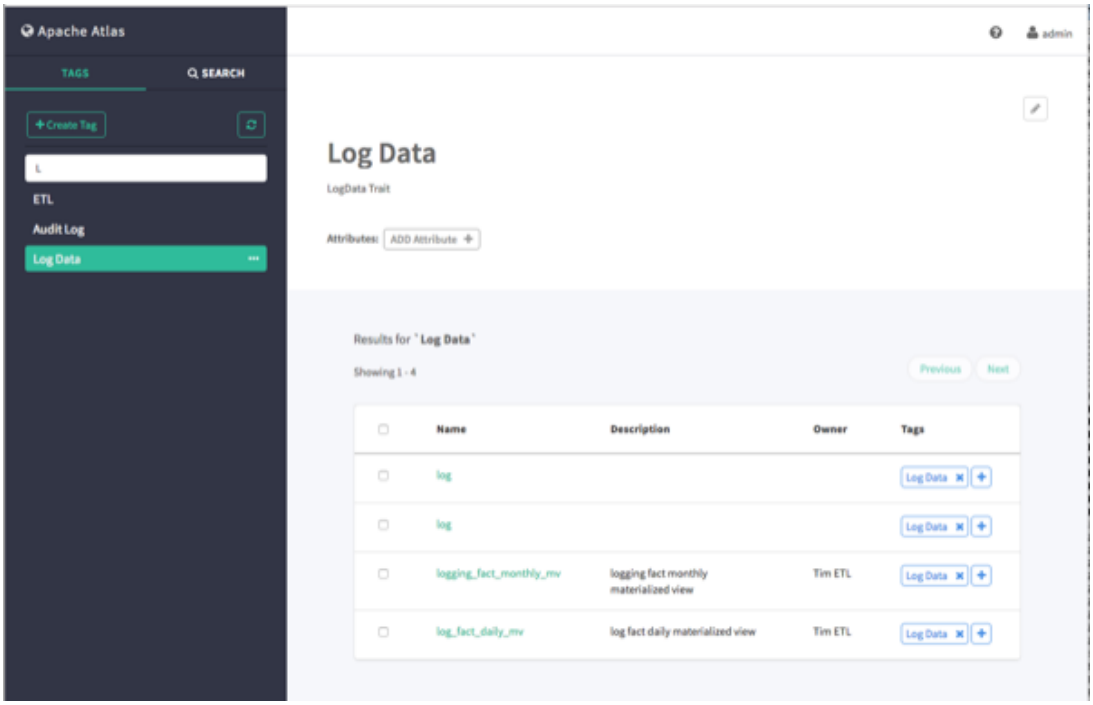


4.3. Searching for Assets Associated with Tags

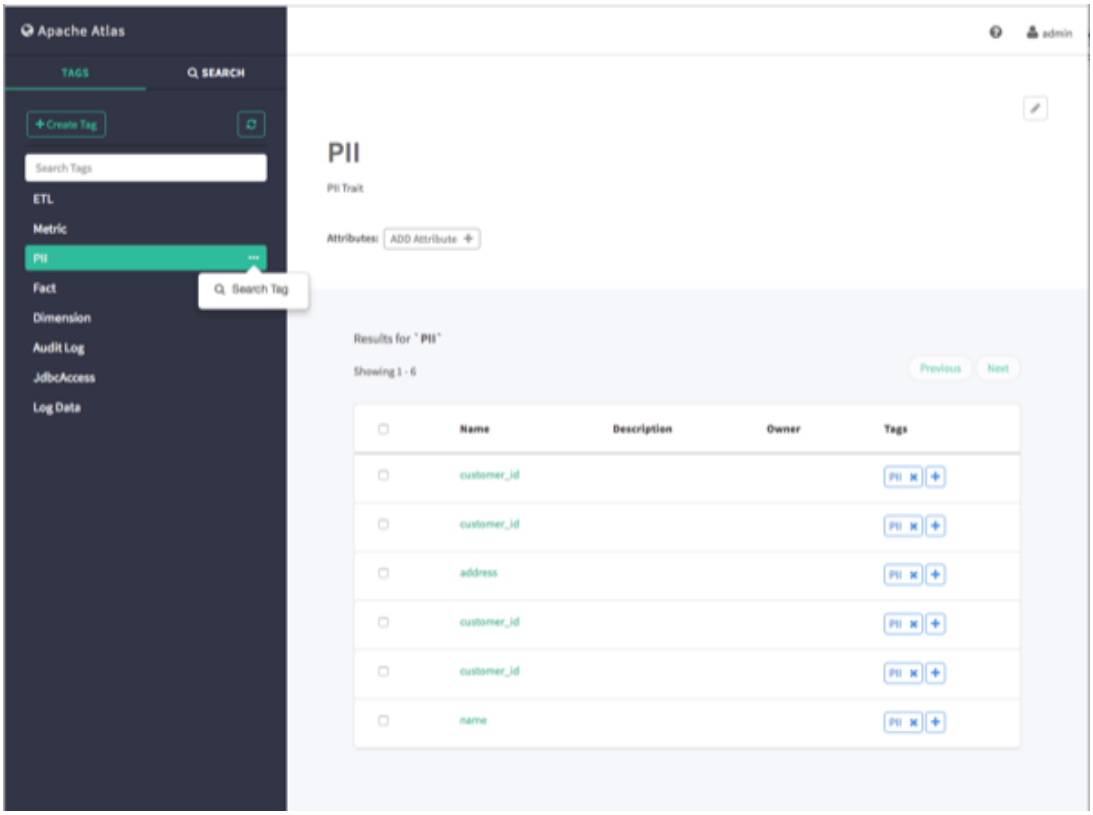
1. To display a list of all of the assets associated with a tag, click the tag name in the Atlas Tags list.



2. To filter the Tags list based on a text string, type the text in the Search Tags box. The list is filtered dynamically as you type to display the tags that contain that text string. You can then click a tag in the filtered list to display the assets associated with that tag.



3. You can also search for assets associated with a tag by clicking the ellipsis symbol for the tag and selecting **Search Tag**. This launches a DSL search query that returns a list of all assets associated with the tag.



5. Managing the Atlas Business Taxonomy (Technical Preview)

The taxonomy feature in Apache Atlas enables you to define a hierarchical set of business terms that represents your business domain. You can then associate these taxonomy terms with the metadata entities that Atlas manages. This hierarchical business catalog makes it easier to organize and discover data stored in Hadoop.

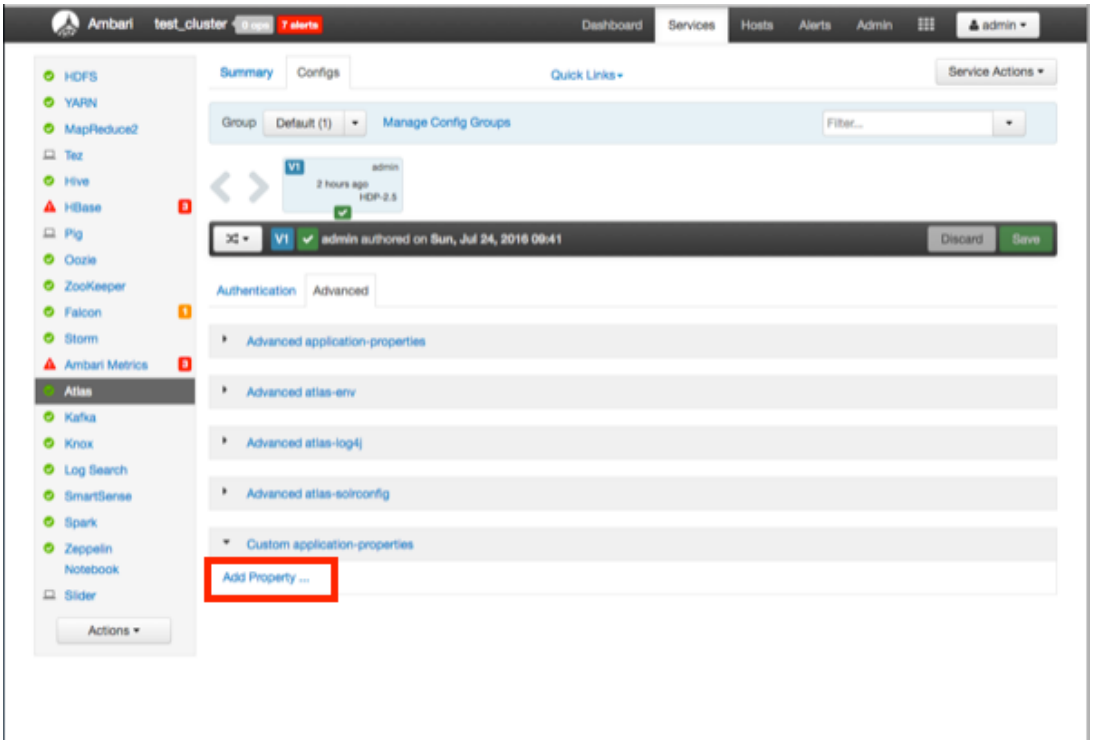
Note

The Apache Atlas Taxonomy feature is a Technical Preview and is considered under development. Do not use this feature in your production systems. If you have questions regarding this feature, contact Support by logging a case on our Hortonworks Support Portal at https://support.hortonworks.com.

5.1. Enabling the Atlas Taxonomy Technical Preview

Because the Atlas Taxonomy feature is a Technical Preview, it is not enabled by default and does not appear on the Atlas web UI. Use the following steps to enable the Atlas Taxonomy feature.

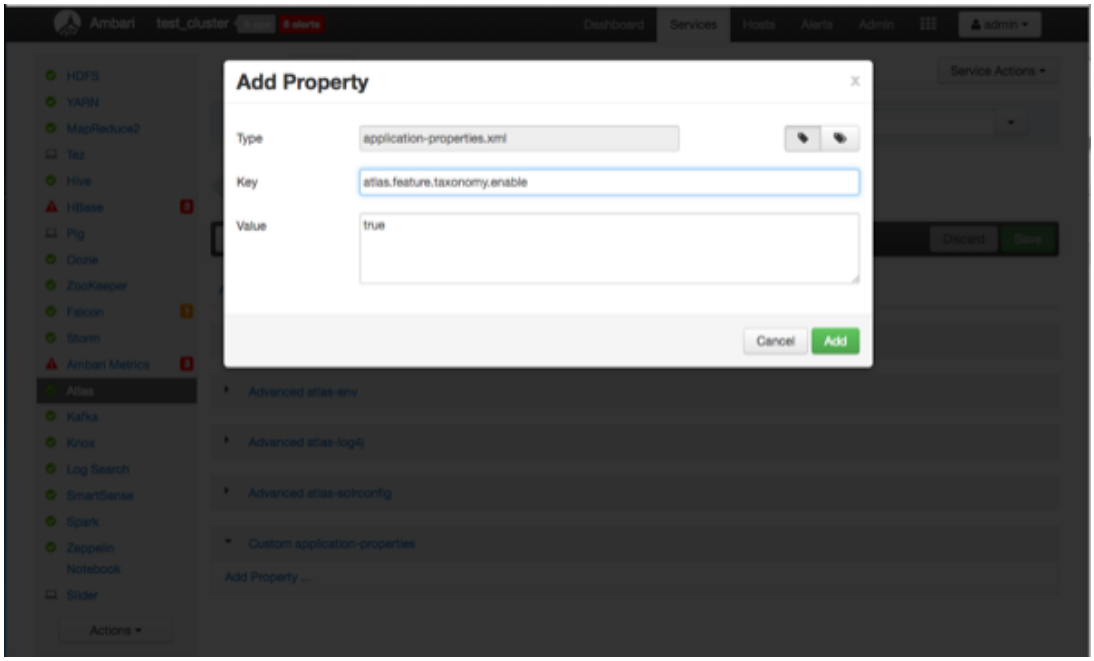
1. Select **Atlas > Configs > Advanced > Custom application-properties**, then click **Add Property**.



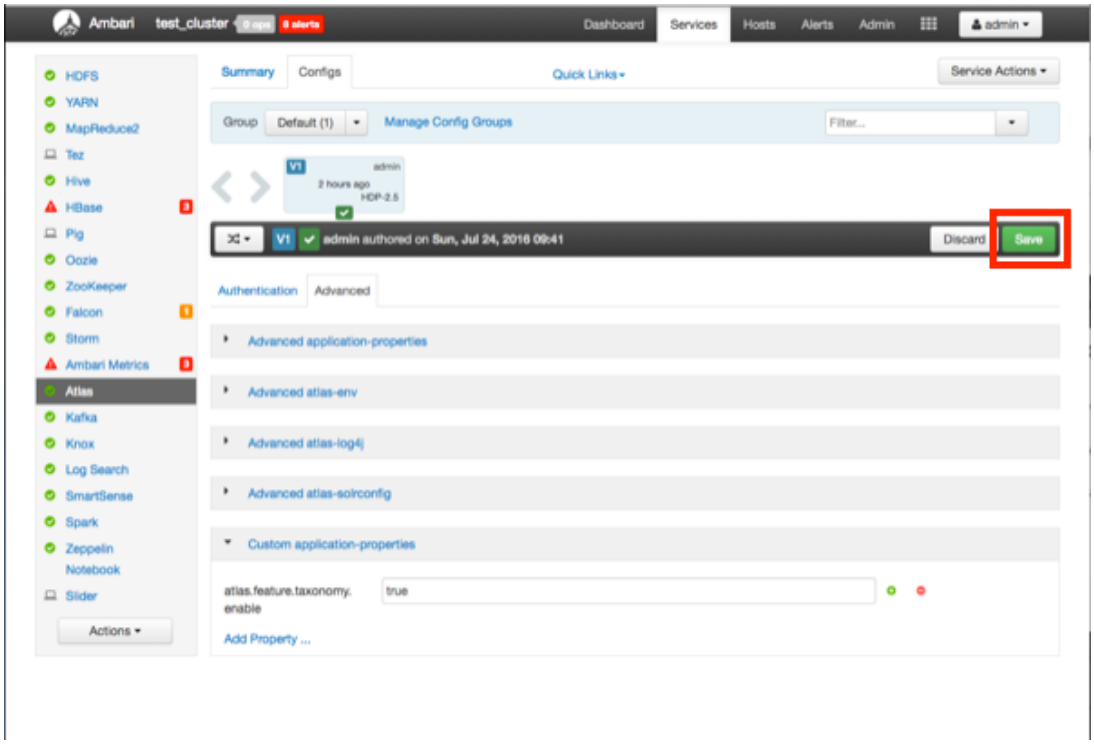
2. On the Add Property pop-up, add the following properties:

   - **Key** – `atlas.feature.taxonomy.enable`
   - **Value** – `true`

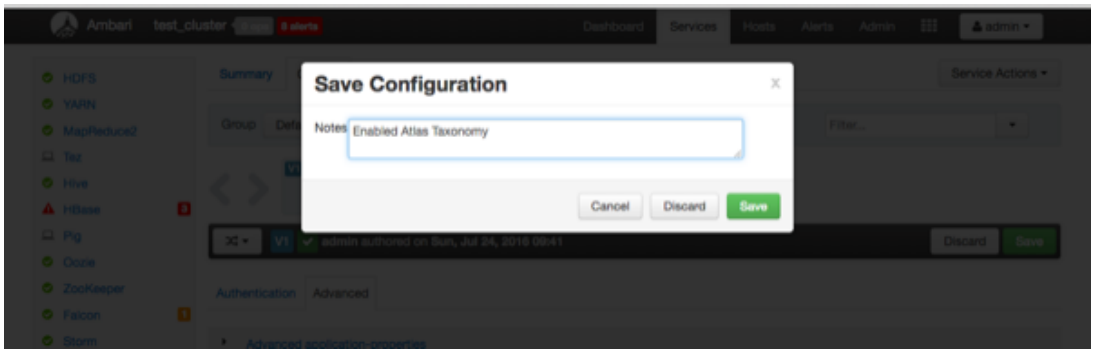
   Click **Add** to add the new property.



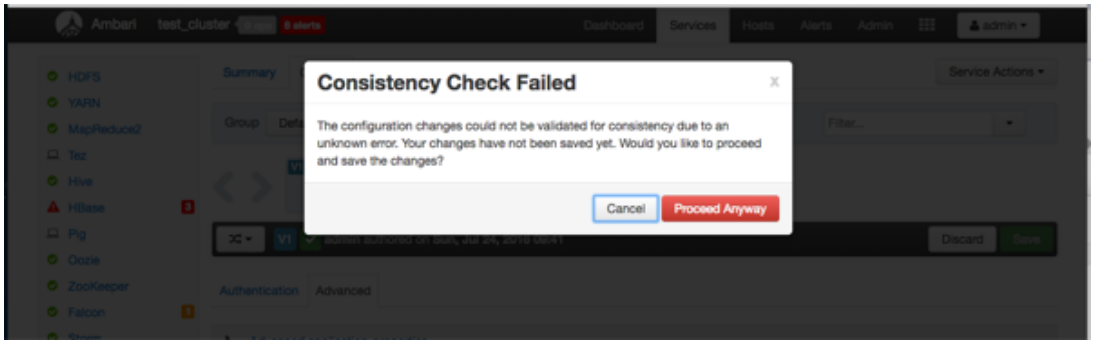
3. The Advanced tab is redisplayed with the new property. Click **Save** to save the new configuration.



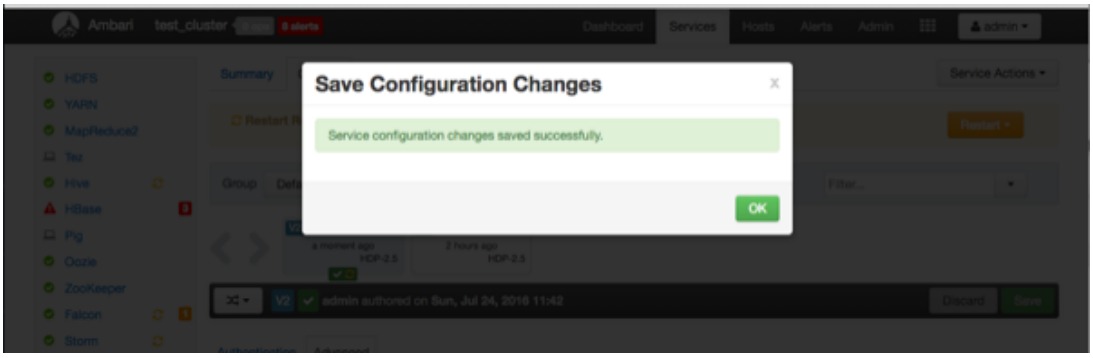
4. A Save Configuration pop-up appears. Type in a note describing the changes you just made, then click **Save**.



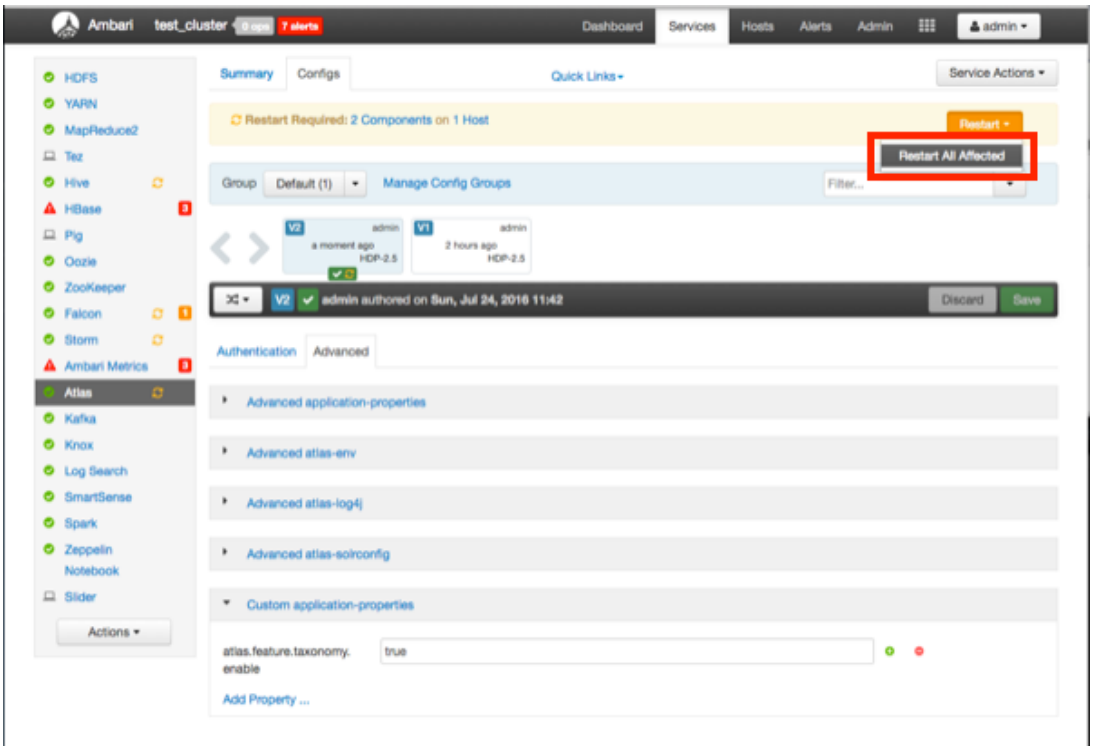
5. If a Consistency Check Failed pop-up appears, click **Proceed Anyway**.



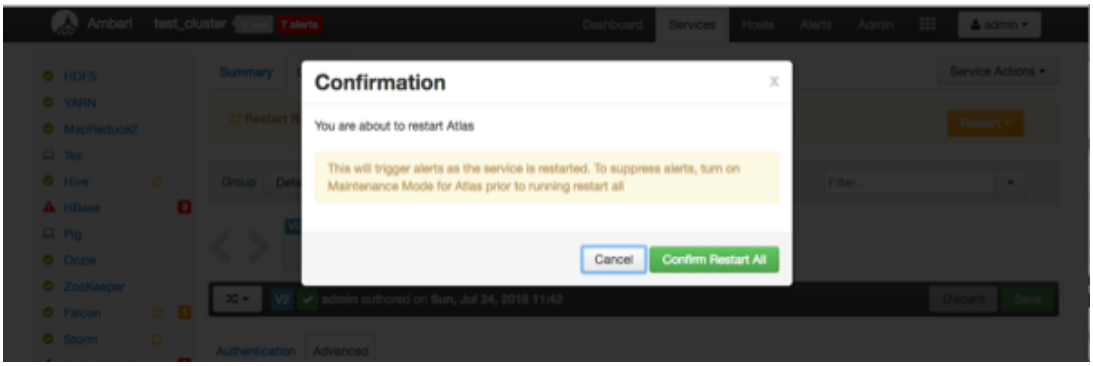
6. Click **OK** on the Save Configuration Changes pop-up.



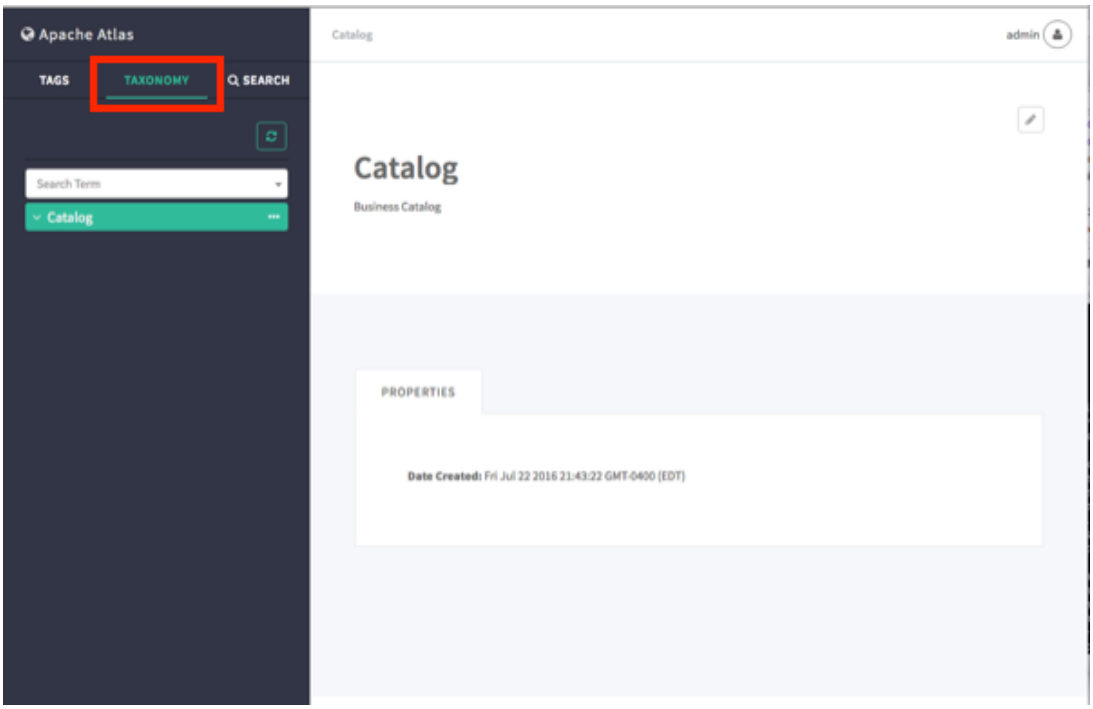
7. Select **Restart > Restart All Affected** to restart the Atlas service and load the new configuration.



8. Click **Confirm Restart All** on the confirmation pop-up to confirm the Atlas restart.

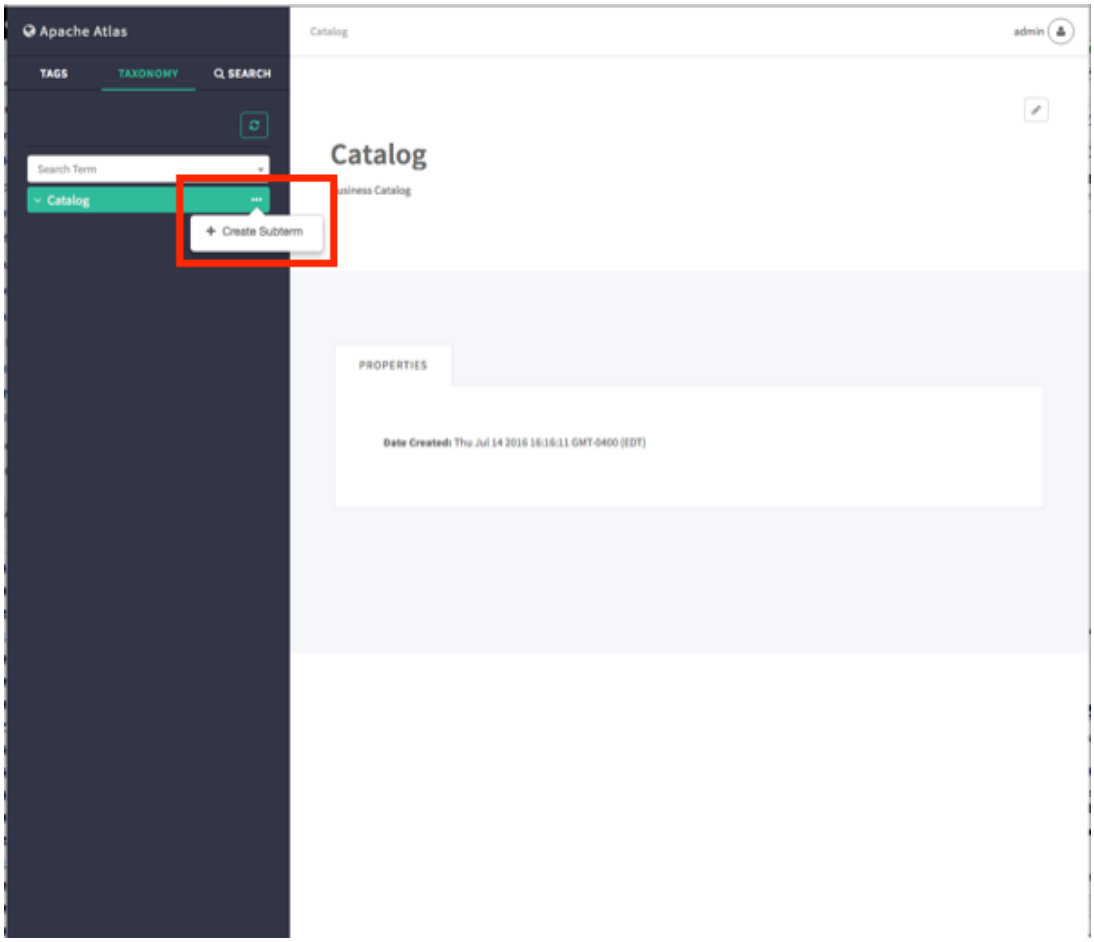


9. After Atlas restarts, the Taxonomy feature is enabled. Other components may also require a restart. To access the Atlas web UI, select **Atlas > Quick Links > Atlas Dashboard**.

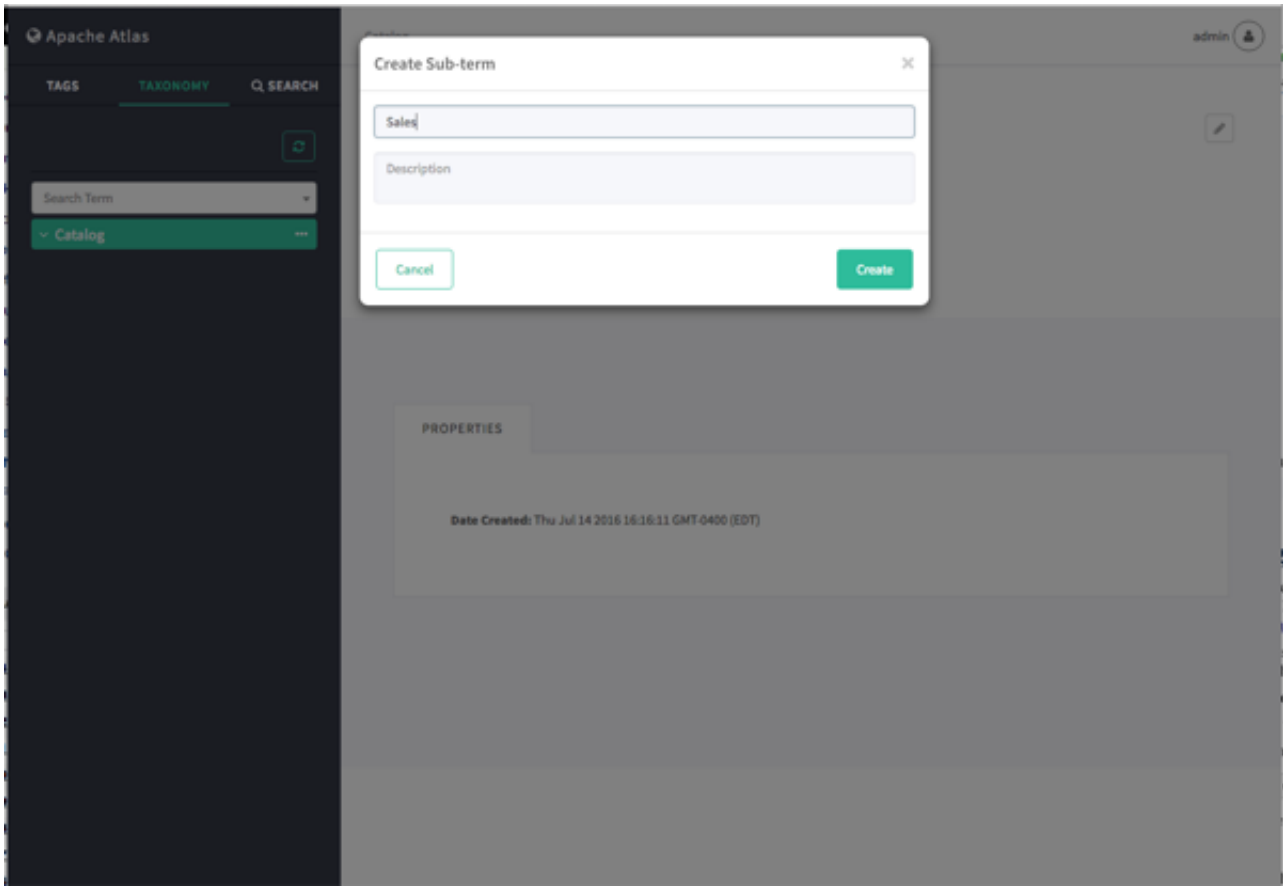


5.2. Creating Taxonomy Terms

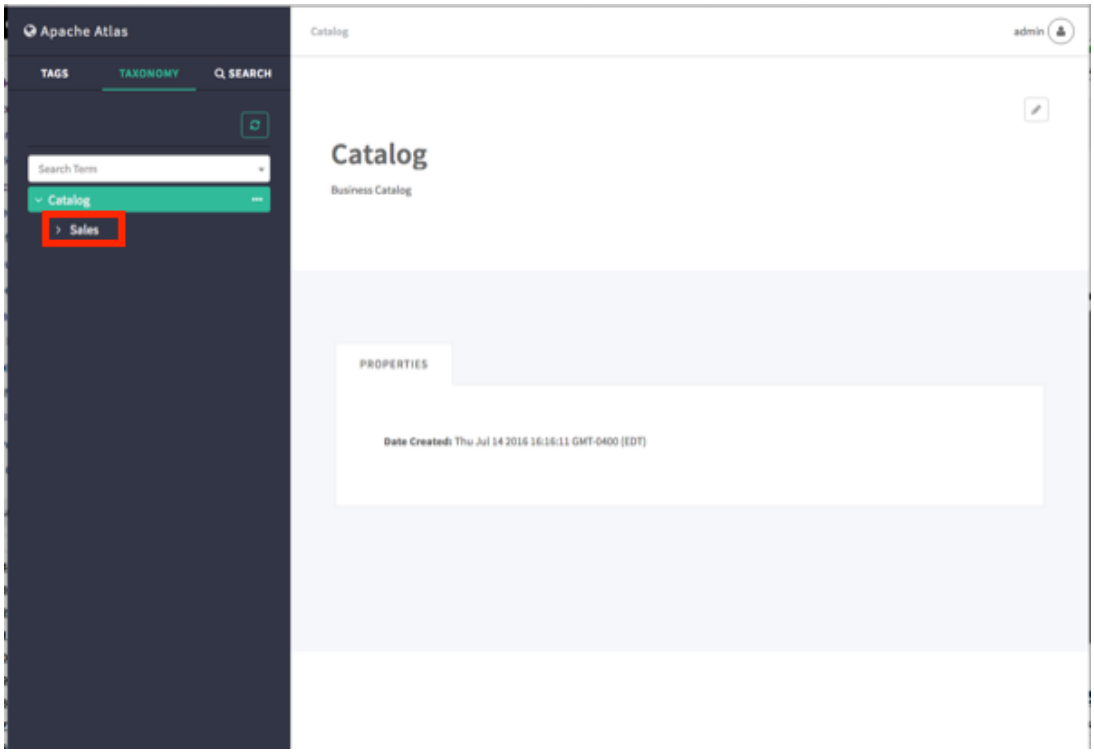
1. On the Atlas web UI, click **Taxonomy**. To create a new sub-term, click the ellipsis symbol at the top level of the Taxonomy, then click **Create Subterm**.



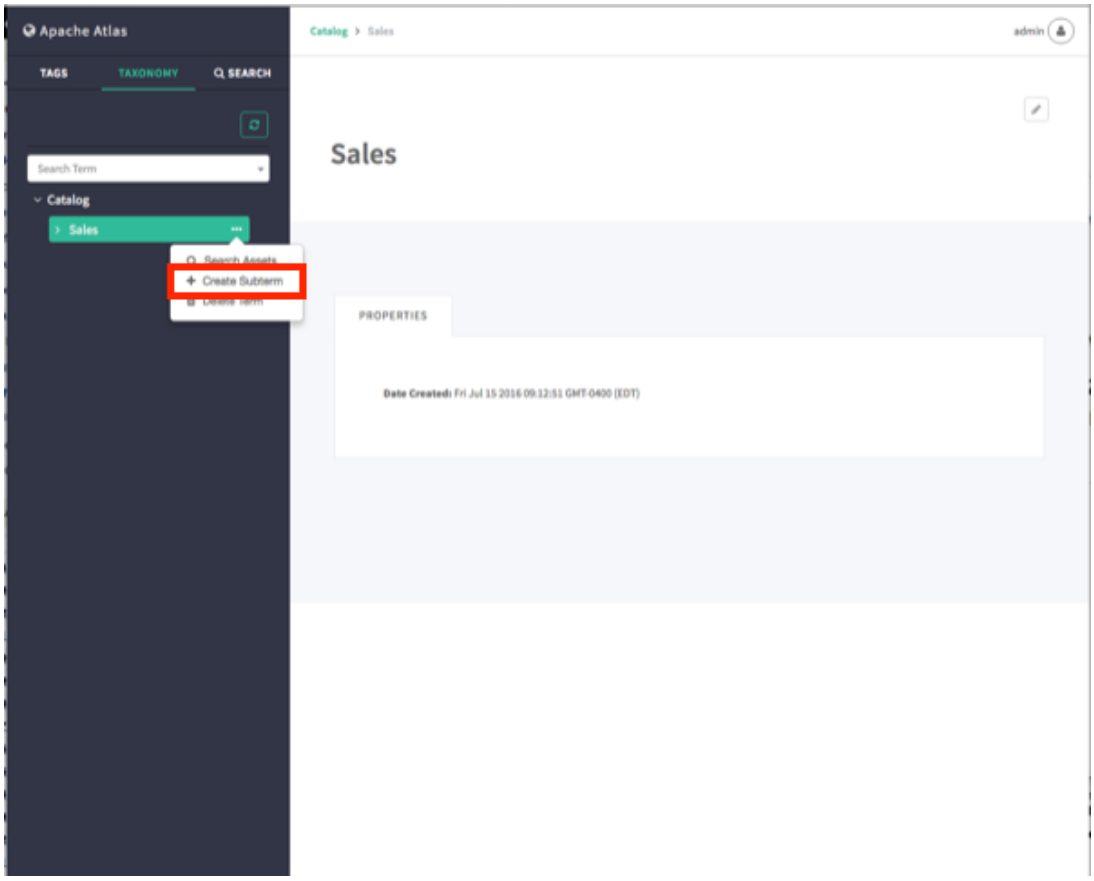
2. On the Create Sub-term pop-up, type in a name and an optional description for the sub-term, then click **Create**.



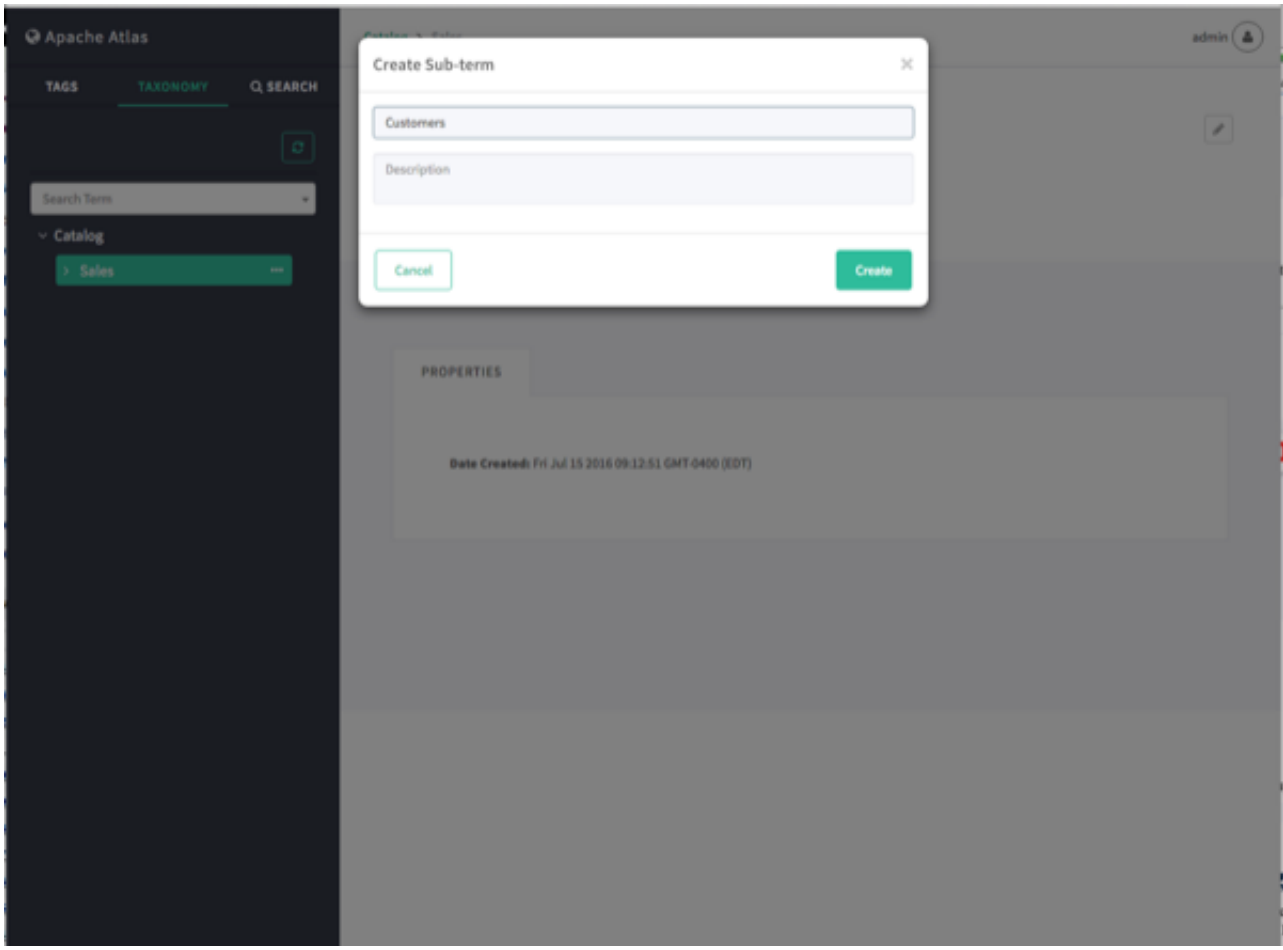
3. The new sub-term appears in the Taxonomy below the top level.



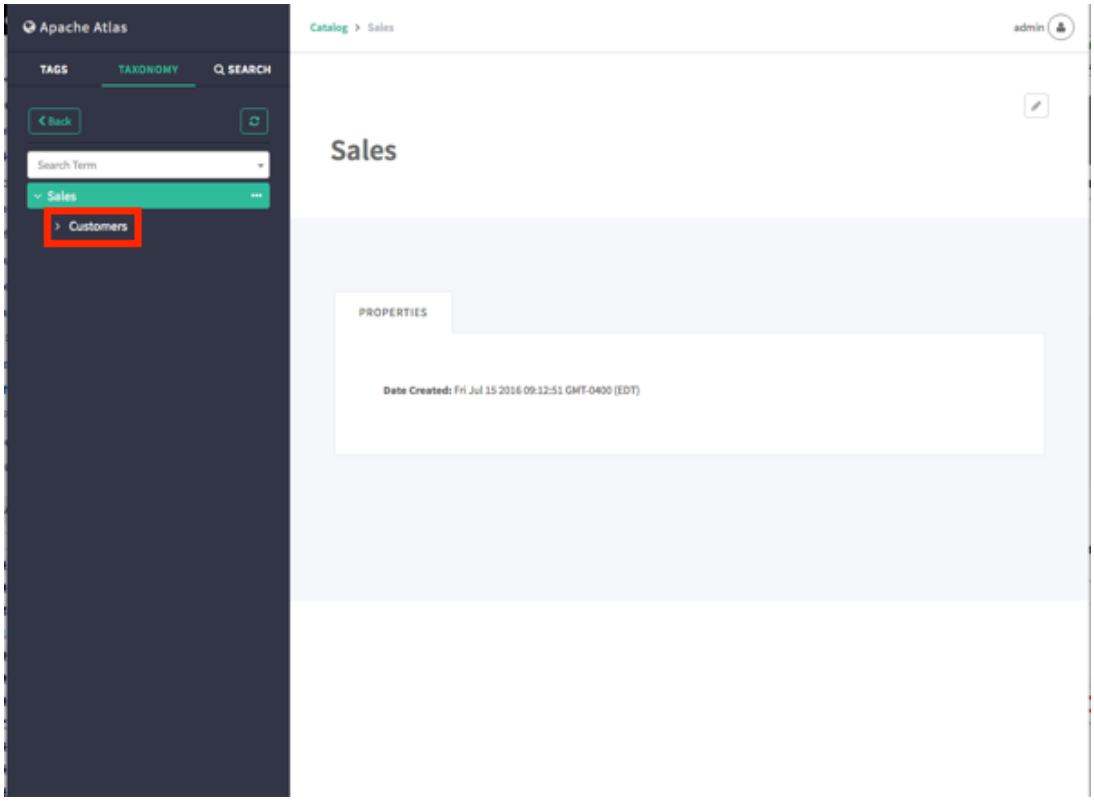
4. To create a new sub-term another level down in the taxonomy hierarchy, select the sub-term, click the ellipsis symbol, then click **Create Subterm**.



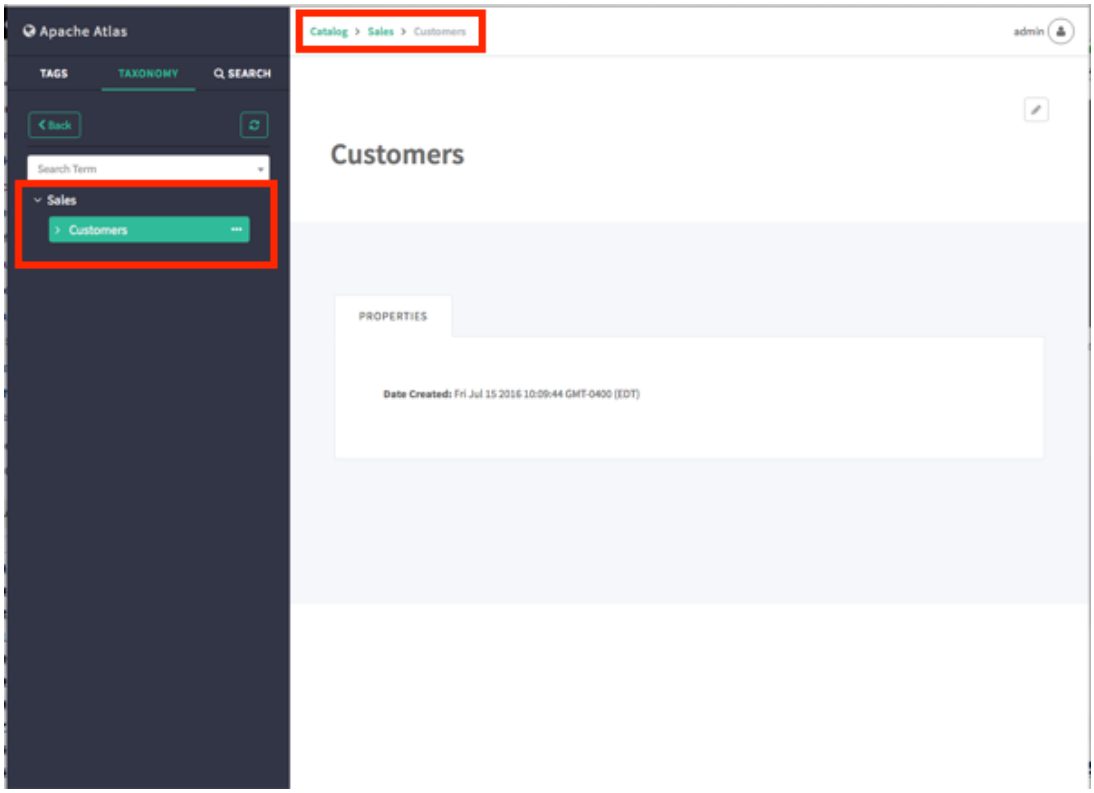
5. On the Create Sub-term pop-up, type in a name and an optional description for the new second-level sub-term, then click **Create**.



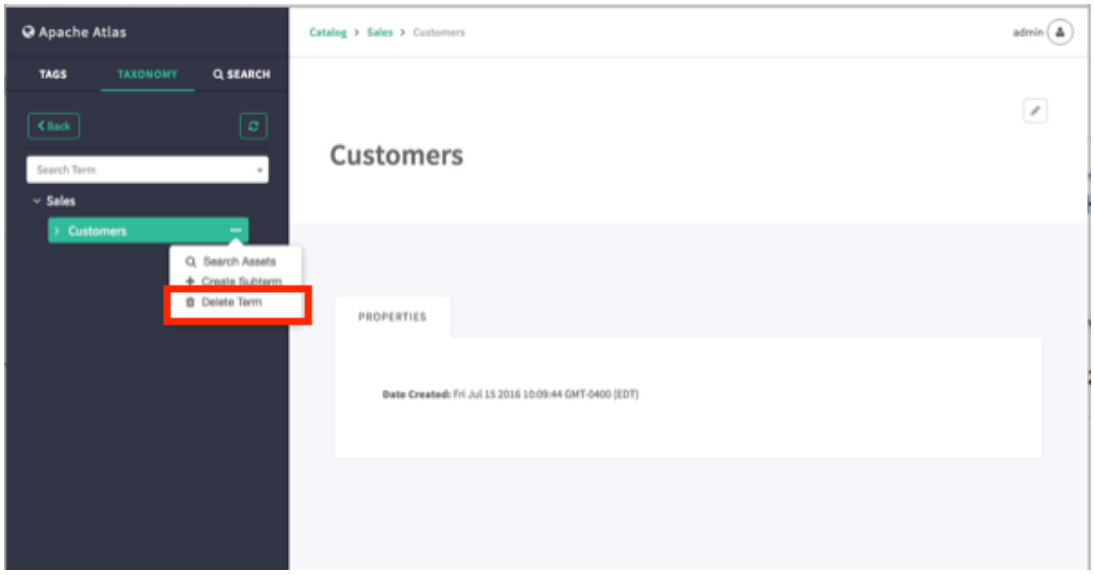
6. The new second-level sub-term appears in the Taxonomy.



7. You can repeat this process to create multiple taxonomy levels. Only two levels at a time are displayed in the navigation bar, but you can use the breadcrumb trail at the top of the page to navigate the taxonomy hierarchy, and you can use the Back button to return to the previously selected level. You can also use the Search Term box to search for taxonomy terms.

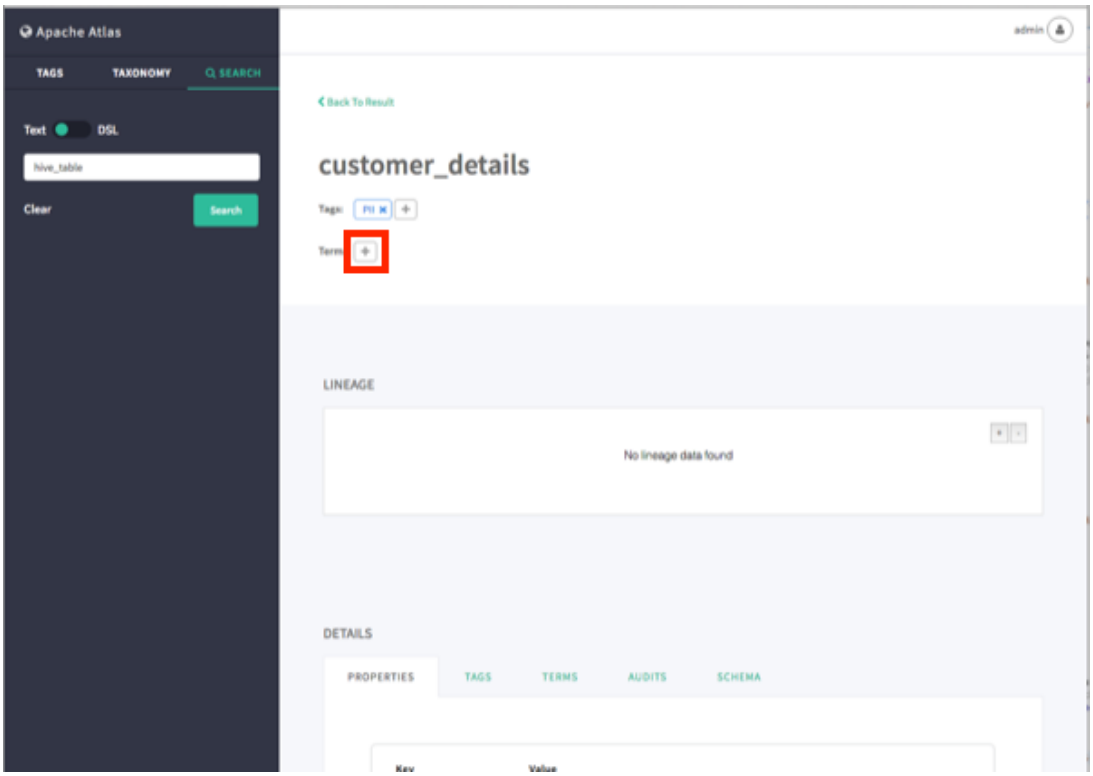


8. To delete a taxonomy term, click the ellipsis symbol for the term, then select **Delete Term**. When you delete a term it is also removed from all assets that are currently associated with the term.

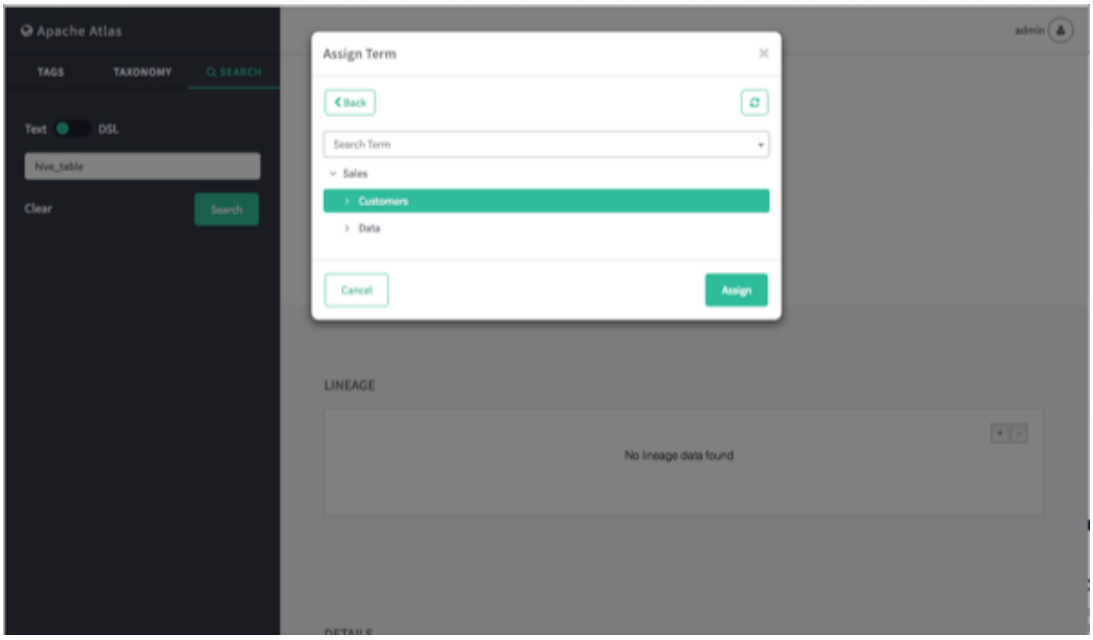


5.3. Associating Taxonomy Terms with Assets

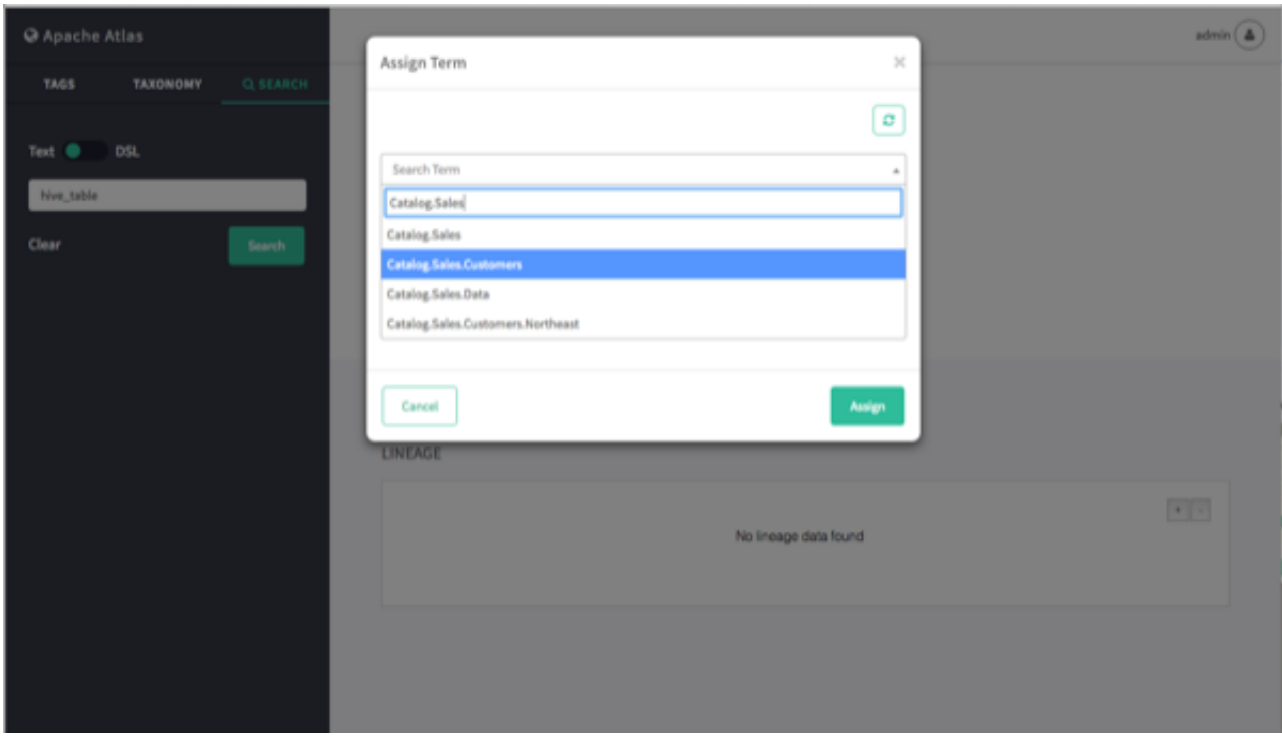
1. Select an asset. In the example below, we searched for all `hive_table` assets, and then selected the "customer_details" table from the list of search results. To associate a taxonomy term with an asset, click the + icon next to the **Terms:** label.



2. On the Assign Term pop-up, browse to select a taxonomy term. Here we have selected the term "Customers".



You can also filter the list of tags by typing text in the Search Term box, and then click to select a term.

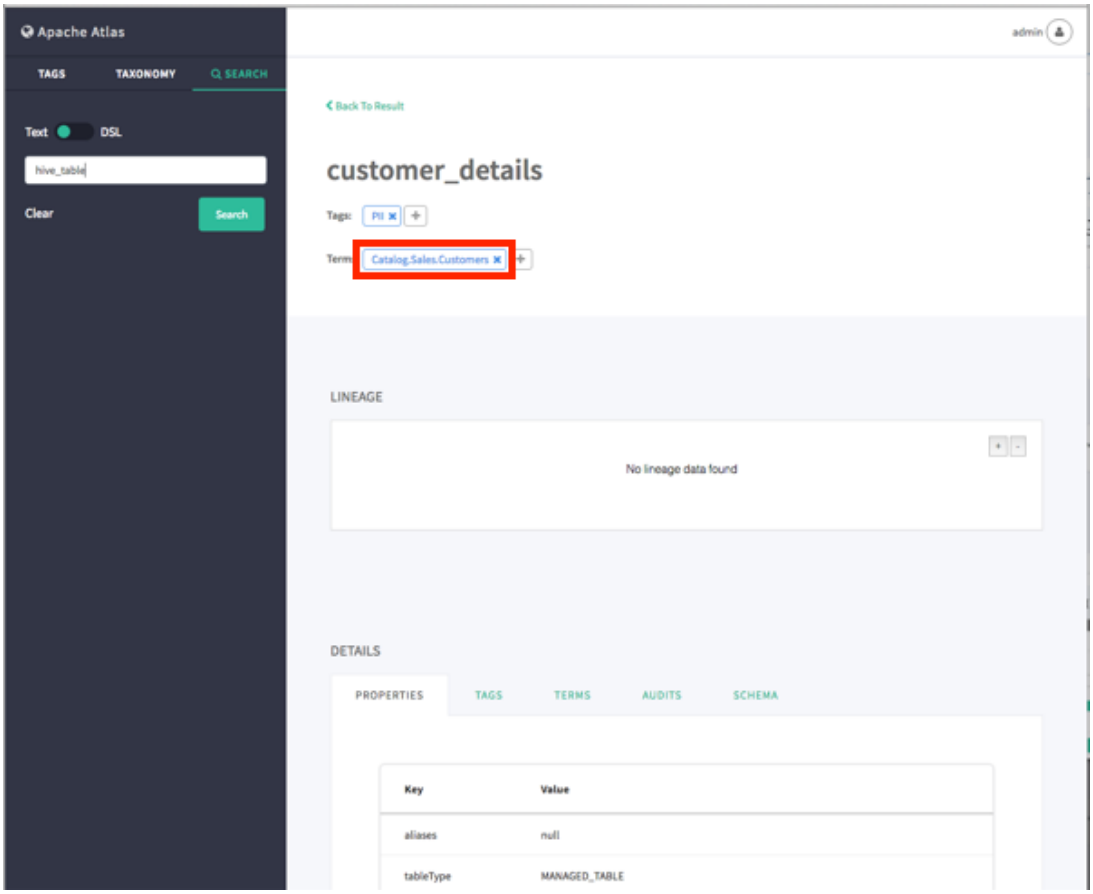


Note

In the Search Term list (and elsewhere in the UI), a period symbol is used as a separator to indicate taxonomy hierarchy levels. For example,

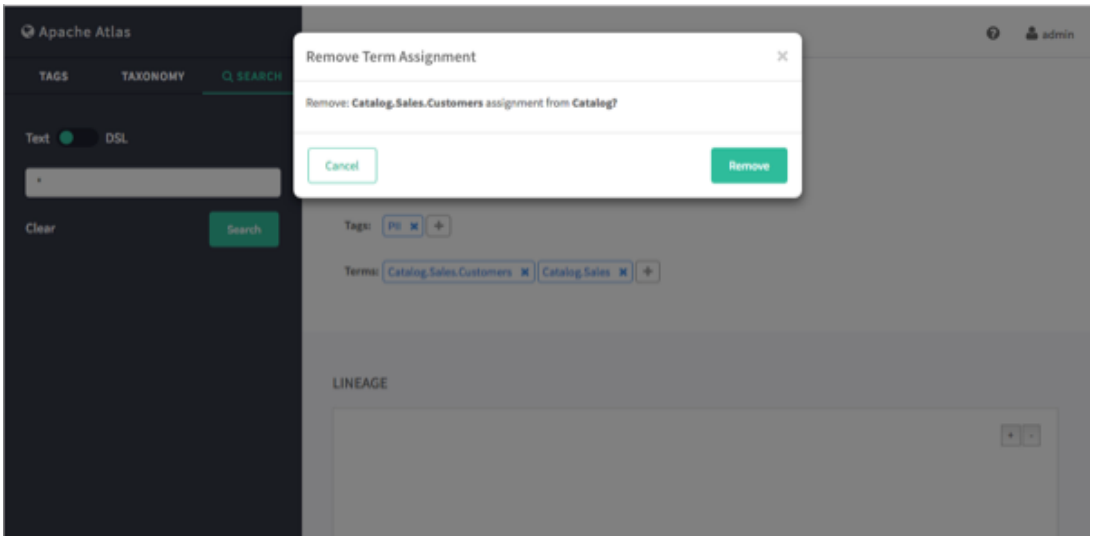
Catalog.Sales.Customers represents the Catalog > Sales > Customers taxonomy level.

3. After you select a term, click **Assign**. The new term is displayed next to the **Terms:** label on the asset page.



4. You can view details about a taxonomy term by clicking the term name on the term label.

   To remove a term from an asset, click the **x** symbol on the term label, then click **Remove** on the confirmation pop-up. This removes the term association with the asset, but does not delete the term itself.

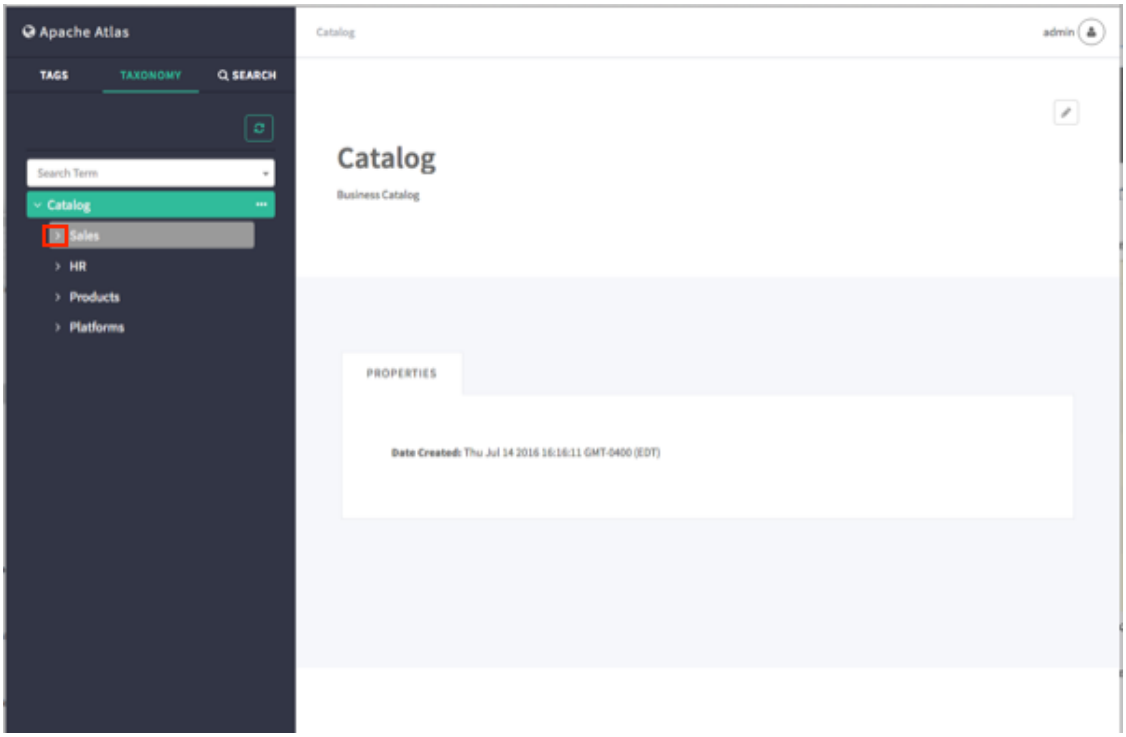


5.4. Navigating the Atlas Taxonomy

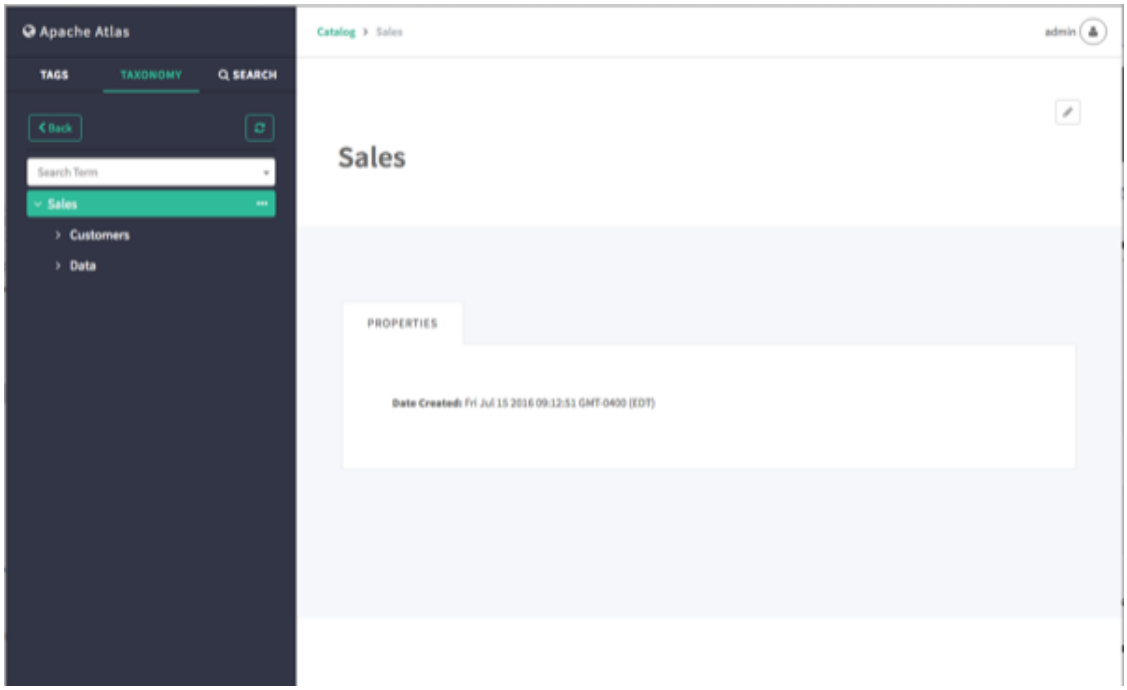
Only two levels at a time are displayed in the Taxonomy list, but you can use the following methods to navigate the Atlas Taxonomy.

5.4.1. Navigation Arrows

To display the child terms that belong to a taxonomy term, click the right-arrow symbol next to the term. For example, if we click the arrow for the Sales term in the following Taxonomy list:



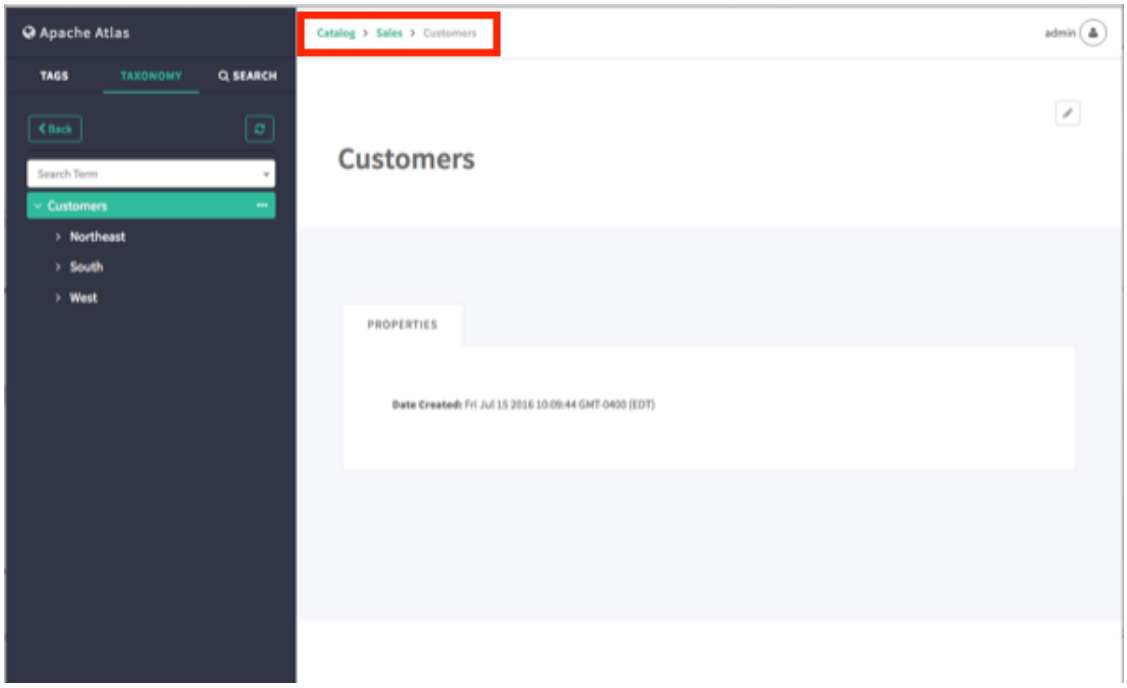
The child terms for Sales (Customers and Data) are displayed:



To hide the child terms, click the down-arrow next to the Sales term.

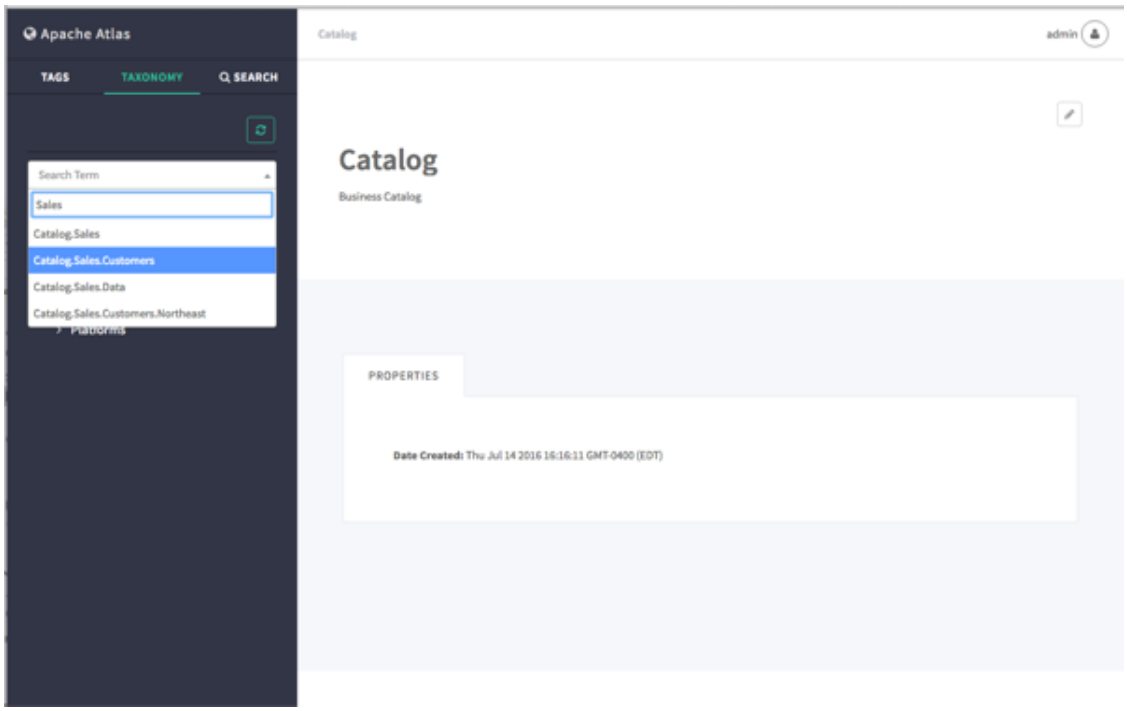
5.4.2. Breadcrumb Trail

As you navigate through the taxonomy, a breadcrumb trail at the top of the page tracks your position in the taxonomy hierarchy. You can use the links in the breadcrumb trail to navigate back to a higher level.



5.4.3. Search Terms

To filter the Taxonomy terms list based on a text string, type the text in the Search Term box. The list is filtered dynamically as you type to display the terms that contain that text string. You can then select a term from the filtered list.

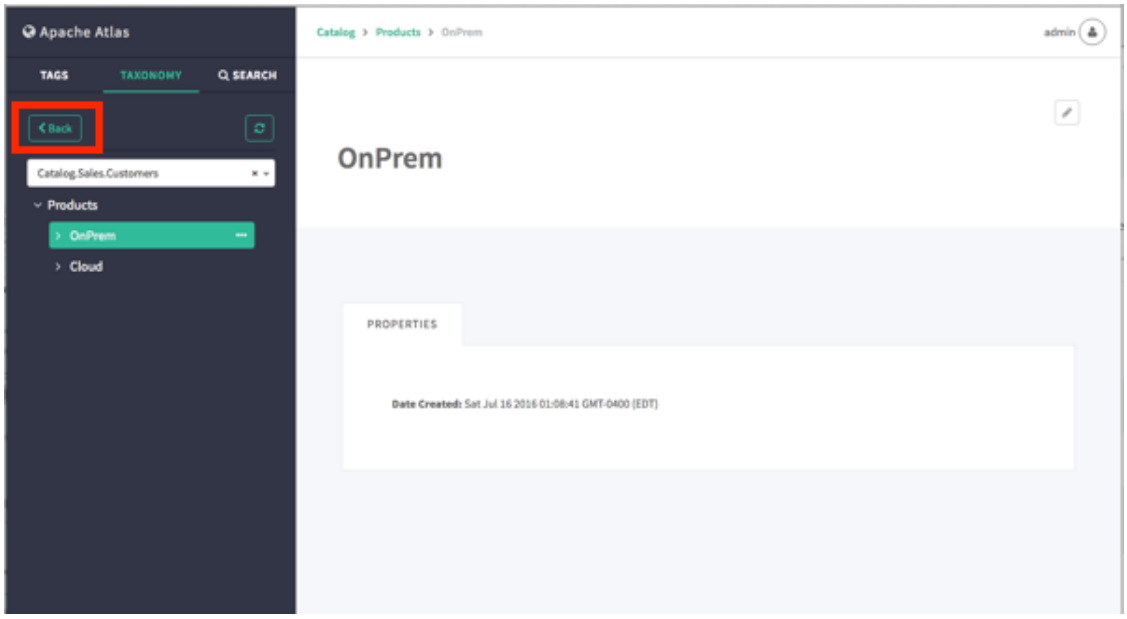


Note

In the Search Term list (and elsewhere in the UI), a period symbol is used as a separator to indicate taxonomy hierarchy levels. For example, Catalog.Sales.Customers represents the Catalog > Sales > Customers taxonomy level.

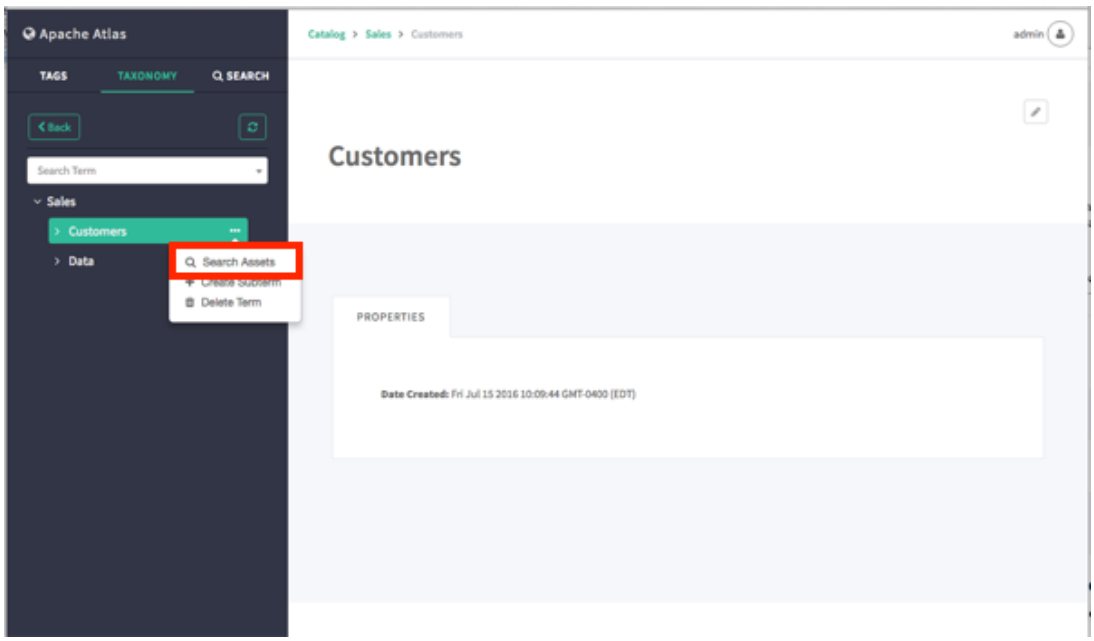
5.4.4. Back Button

You can also use the **Back** button on the Atlas web UI (or your browser's Back button) to return to the previous taxonomy page.

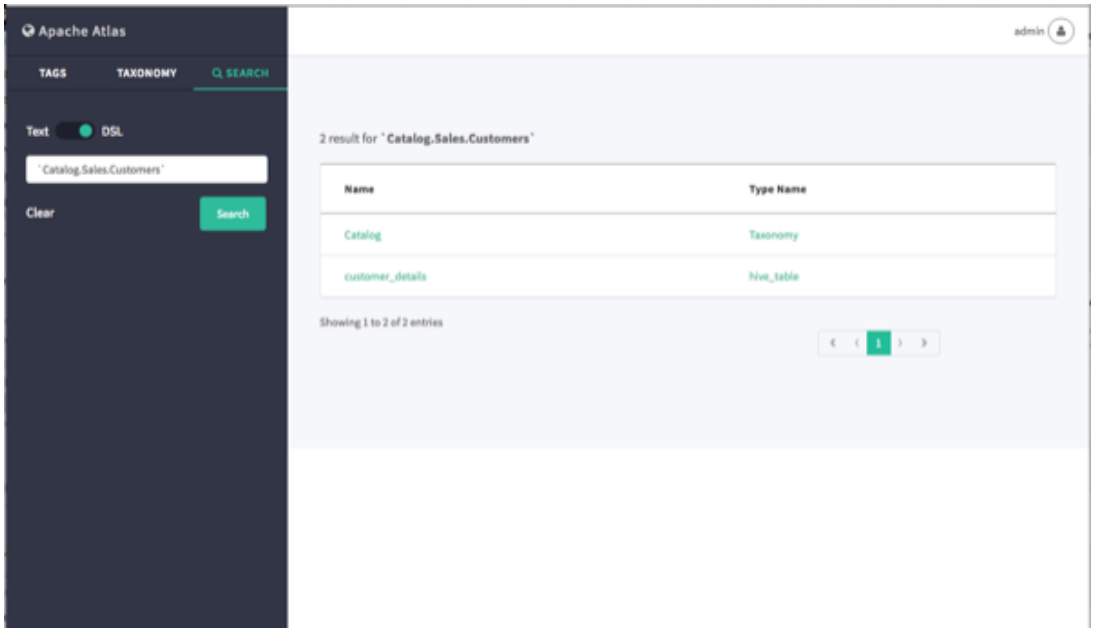


5.5. Searching for Assets Associated with Taxonomy Terms

1. To search for assets associated with a taxonomy term, select the term, click the ellipsis symbol, and then select **Search Assets**.



2. This launches a DSL search query that returns a list of all assets associated with the term.


Apache Atlas
admin
TAGS
TAXONOMY
SEARCH
Text
DSL
'Catalog.Sales.Customers'
Clear
Search
2 result for 'Catalog.Sales.Customers'
Name
Type Name
Catalog
Taxonomy
customer_details
hive_table
Showing 1 to 2 of 2 entries

6. Apache Atlas REST API Reference

This API supports a Representational State Transfer (REST) model for accessing a set of resources through a fixed set of operations. The following resources are accessible through the RESTful model:

- AdminResource
- DataSetLineageResource
- EntityService
- LineageResource
- MetadataDiscoveryResource
- TaxonomyService
- TypesResource

6.1. Data Model

All endpoints act on a common set of data. The data can be represented with difference media (i.e. "MIME") types, depending on the endpoint that consumes and/or produces the data. The data can be described by an XML Schema, which definitively describes the XML representation of the data, but is also useful for describing the other formats of the data, such as JSON.

This document describes the data using terms based on an XML Schema. Data can be grouped by namespace with a schema document describing the elements and types of the namespace. Types define the structure of the data and elements are instances of a type. For example, elements are usually produced by (or consumed by) a REST endpoint, and the structure of each element is described by its type.

6.2. AdminResource

Jersey Resource for administrative operations. The following resources are available:

- /admin/session [56]
- /admin/stack [57]
- /admin/status [57]
- /admin/version [57]

/admin/session

GET		
Response Body	element:	(custom)

	media types:	application/json

/admin/stack

GET	Fetches the thread stack dump for this application.	
Response Body	element:	(custom)
	media types:	text/plain
	JSON represents the thread stack dump.	

/admin/status

GET		
Response Body	element:	(custom)
	media types:	application/json

/admin/version

GET	Fetches the version for this application.	
Response Body	element:	(custom)
	media types:	application/json
	JSON represents the version.	

6.3. DataSetLineageResource

Jersey Resource for the Hive table lineage. The following resources are available:

- /lineage/hive/table/{tableName}/inputs/graph [57]
- /lineage/hive/table/{tableName}/outputs/graph [57]
- /lineage/hive/table/{tableName}/schema [58]

/lineage/hive/table/{tableName}/inputs/graph

GET	Fetches the inputs graph for an entity.			
Parameters	**name**	**description**	**type**	**default**
	tableName	The name of the table.	path	
Response Body	element:		(custom)	
	media types:		application/json	

/lineage/hive/table/{tableName}/outputs/graph

GET	Fetches the outputs graph for an entity.			
Parameters	**name**	**description**	**type**	**default**

	tableName	The name of the table.	path	
Response Body	element:		(custom)	
	media types:		application/json	

/lineage/hive/table/{tableName}/schema

GET	Fetches the schema for the table.			
Parameters	**name**	**description**	**type**	**default**
	tableName	The name of the table.	path	
Response Body	element:		(custom)	
	media types:		application/json	

6.4. EntityService

Entity management operations. An entity is an instance of a type. Entities conform to the definition of the type that they they correspond to. The following resources are available:

- /entities [58]
- /entities/{guid} [59]
- /entities/{guid}/audit [60]
- /entities/{guid}/traits [60]
- /entities/{guid}/traits/{traitName} [61]

/entities

POST	Submits the entity definitions (instances). The body contains the JSONArray of entity json. The service takes care of de-duping the entities based on any unique attribute for the give type.	
Response Body	element:	(custom)
	media types:	application/json

PUT	Completely updates a set of entities. Unspecified values are replaced with null and removed. Adds or updates the entities specified by GUID or a unique attribute.	
Response Body	element:	(custom)
	media types:	application/json

DELETE	Deletes entities from the repository identified by their GUIDs (including their composite references), or deletes a single entity identified by its type and unique attribute (including composite references).			
Parameters	**name**	**description**	**type**	**default**
	guid	A list of deletion	query	

		candidate GUIDs.		
	type	The entity type.	query	
	property	The unique attribute used to identify the entity.	query	
	value	The unique attribute value used to identify the entity.	query	

Response Body	
	element: (custom)
	media types: application/json
	Response payload as json – includes guids of entities (including composite references from that entity) that were deleted.

GET	Fetches the list of entities for an entity type.			
Parameters	**name**	**description**	**type**	**default**
	type	The name of a unique type.	query	
Response Body	element: (custom)			
	media types: application/json			

/entities/{guid}

GET	Fetches the complete definition of the entity identified by the GUID.			
Parameters	**name**	**description**	**type**	**default**
	GUID	The globally unique identifier of the entity.	path	
Response Body	element: (custom)			
	media types: application/json			

POST	Updates an entity identified by its GUID. Supports partial update of an entity – adds/updates any new values specified. Does not support removal of attribute values.			
Parameters	**name**	**description**	**type**	**default**
	GUID	The globally unique identifier of the entity.	path	
	property	The property that must be added.	query	
Response Body	element: (custom)			
	media types: application/json			

	Response payload as JSON.

/entities/{guid}/audit

GET	Returns the entity audit events for a given entity GUID. The events are returned in decreasing order based on timestamp.
Parameters	**name** / **description** / **type** / **default** guid / The globally unique identifier of the entity. / path / startKey / Used for pagination. Startkey is inclusive; the returned results contain the event with the given startkey. The first time getAuditEvents() is called for an entity, startKey should be null, with count = (number of events required + 1). The next time getAuditEvents() is called for the same entity, startKey should be equal to the entityKey of the last event returned in the previous call. / query / count / The number of events required / query / 100
Response Body	element: (custom) media types: application/json A list of trait names for the entity that is identified by the GUID.

/entities/{guid}/traits

GET	Gets the list of trait names for the entity that is represented by the GUID.

Parameters	**name** / **description** / **type** / **default** GUID / The globally unique identifier of the entity. / path /
Response Body	element: (custom) media types: application/json A list of trait names for the entity that is identified by the GUID.

POST	Submits a new trait to an existing entity that is represented by the GUID.
Parameters	**name** / **description** / **type** / **default** GUID / The globally unique identifier of the entity. / path /
Response Body	element: (custom) media types: application/json

/entities/{guid}/traits/{traitName}

DELETE	Deletes a trait from the entity that is represented by the GUID.
Parameters	**name** / **description** / **type** / **default** GUID / The globally unique identifier of the entity. / path / traitName / The name of the trait. / path /
Response Body	element: (custom) media types: application/json

/entities/qualifiedName

POST	Adds or Updates a given entity identified by its unique attribute (entityType, attributeName and value). Updates support only partial update of an entity – adds or updates any new values specified. Updates do not support removal of attribute values.
Parameters	**name** / **description** / **type** / **default** type / The entity type. / query / property / The unique attribute used to identify the entity / query / value / The unique attribute's value. / query /

Response Body	element: (custom) media types: application/json Response payload as json – The body contains the JSONArray of entity json. The service takes care of de-duping the entities based on any unique attribute for the give type.

6.5. LineageResource

The following resources are available:

- /lineage/{guid}/inputs/graph [62]
- /lineage/{guid}/outputs/graph [62]
- /lineage/{guid}/schema [62]

/lineage/{guid}/inputs/graph

GET	Returns an inputs lineage graph for the specified entity ID.			
Parameters	**name**	**description**	**type**	**default**
	guid	dataset entity ID	path	
Response Body	element: (custom) media types: application/json			

/lineage/{guid}/outputs/graph

GET	Returns an outputs lineage graph for the specified entity ID.			
Parameters	**name**	**description**	**type**	**default**
	guid	dataset entity ID	path	
Response Body	element: (custom) media types: application/json			

/lineage/{guid}/schema

GET	Returns the schema for the specified entity ID.			
Parameters	**name**	**description**	**type**	**default**
	guid	dataset entity ID	path	
Response Body	element: (custom) media types: application/json			

6.6. MetadataDiscoveryResource

Jersey Resource for metadata operations. The following resources are available:

- /discovery/search [63]
- /discovery/search/dsl [63]
- /discovery/search/fulltext [63]
- /discovery/search/gremlin [63]

Note

Only the Admin user is authorized to invoke the Gremlin search API.

/discovery/search

GET	Search by using a query.
Parameters	**name** / **description** / **type** / **default** query / The search query in raw Gremlin or DSL format that falls back to full text. / query /
Response Body	element: (custom) media types: application/json JSON represents the type and results.

/discovery/search/dsl

GET	Search by using the query DSL format.
Parameters	**name** / **description** / **type** / **default** query / The search query in DSL format. / query /
Response Body	element: (custom) media types: application/json JSON represents the type and results.

/discovery/search/fulltext

GET	Search by using full text search.
Parameters	**name** / **description** / **type** / **default** query / The full text search query. / query /
Response Body	element: (custom) media types: application/json JSON represents the type and results.

/discovery/search/gremlin

GET	Search by using the raw gremlin query format.

Parameters	**name** / **description** / **type** / **default** query / The search query in raw gremlin format. / query /
Response Body	element: (custom) media types: application/json JSON represents the type and results.

6.7. TaxonomyService

This service handles API requests for taxonomy and term resources.The following resources are available:

- /v1/taxonomies [64]
- /v1/taxonomies/{taxonomyName} [64]
- /v1/taxonomies/{taxonomyName}/terms [65]
- /v1/taxonomies/{taxonomyName}/terms/{rootTerm}/{remainder} [65]
- /v1/taxonomies/{taxonomyName}/terms/{termName} [65]
- /v1/taxonomies/{taxonomyName}/terms/{termName}/{remainder} [66]

v1/taxonomies

GET	
Response Body	element: (custom) media types: application/json

/v1/taxonomies/{taxonomyName}

GET	
Parameters	**name** / **description** / **type** / **default** taxonomyName / The taxonomy name. / path /
Response Body	element: (custom) media types: application/json

POST	
Parameters	**name** / **description** / **type** / **default** taxonomyName / The taxonomy name. / path /
Request Body	element: body media types: */*application/xml

Response Body	element:	(custom)
	media types:	application/json

PUT				
Parameters	**name**	**description**	**type**	**default**
	taxonomyName	The taxonomy name.	path	
Request Body	element:		body	
	media types:		*/*application/xml	
Response Body	element:		(custom)	
	media types:		application/json	

DELETE				
Parameters	**name**	**description**	**type**	**default**
	taxonomyName	The taxonomy name.	path	
Response Body	element:		(custom)	
	media types:		application/json	

/v1/taxonomies/{taxonomyName}/terms

GET				
Parameters	**name**	**description**	**type**	**default**
	taxonomyName	The taxonomy name.	path	
Response Body	element:		(custom)	
	media types:		application/json	

/v1/taxonomies/{taxonomyName}/terms/{rootTerm}/{remainder}

GET				
Parameters	**name**	**description**	**type**	**default**
	taxonomyName	The taxonomy name.	path	
	rootTerm		path	
	remainder		path	
Response Body	element:		(custom)	
	media types:		application/json	

/v1/taxonomies/{taxonomyName}/terms/{termName}

GET				
Parameters	**name**	**description**	**type**	**default**

	taxonomyName	The taxonomy name.	path	
	termName		path	
Response Body	element:		(custom)	
	media types:		application/json	

POST				
Parameters	**name**	**description**	**type**	**default**
	taxonomyName	The taxonomy name.	path	
	termName		path	
Request Body	element:		body	
	media types:		*/*application/xml	
Response Body	element:		(custom)	
	media types:		application/json	

PUT				
Parameters	**name**	**description**	**type**	**default**
	taxonomyName	The taxonomy name.	path	
	termName		path	
Request Body	element:		body	
	media types:		*/*application/xml	
Response Body	element:		(custom)	
	media types:		application/json	

DELETE				
Parameters	**name**	**description**	**type**	**default**
	taxonomyName	The taxonomy name.	path	
	termName		path	
Response Body	element:		(custom)	
	media types:		application/json	

/v1/taxonomies/{taxonomyName}/terms/{termName}/{remainder}

POST				
Parameters	**name**	**description**	**type**	**default**
	taxonomyName	The taxonomy name.	path	
	termName		path	
	remainder		path	

Request Body	element: body media types: */*application/xml
Response Body	element: (custom) media types: application/json

PUT				
Parameters	**name**	**description**	**type**	**default**
	taxonomyName	The taxonomy name.	path	
	termName		path	
	remainder		path	
Request Body	element: body media types: */*application/xml			
Response Body	element: (custom) media types: application/json			

DELETE				
Parameters	**name**	**description**	**type**	**default**
	taxonomyName	The taxonomy name.	path	
	termName		path	
	remainder		path	
Response Body	element: (custom) media types: application/json			

6.8. TypesResource

This class provides a RESTful API for types. A type is the description of any representable item, for example, a Hive table. You can represent any meta model of any domain using these types. The following resources are available:

- /types [67]
- /types/{typeName} [68]

/types

POST	Submits a type definition that corresponds to a type that represents a domain meta model. This method can represent objects like a Hive database, Hive table, and so on.
Response Body	element: (custom) media types: application/json

PUT	Updates existing types. If the specified type doesn't exist, a new type is created. Allowed updates are: 1. Add optional

	attribute 2. Change required to optional attribute 3. Add super types. Super types should not contain any required attributes.
Response Body	element: (custom) media types: application/json

GET	Fetches a list of trait type names that are registered in the type system.
Parameters	**name** / **description** / **type** / **default** type / The name of the enumerator `org.apache.atlas.typesystem.types.DataTypes.TypeCategory`. Typically, this can be one of: all, TRAIT, CLASS, ENUM, STRUCT. / query / all
Response Body	element: (custom) media types: application/json The entity names response payload represented as JSON.

/types/{typeName}

GET	Fetches the complete definition of a unique type name.
Parameters	**name** / **description** / **type** / **default** typename / The unique name of the type. / path /
Response Body	element: (custom) media types: application/json