Hortonworks Data Platform

Data Governance

(April 3, 2017)

docs.hortonworks.com

Hortonworks Data Platform: Data Governance

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.

Table of Contents

List of Figures

List of Tables

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:

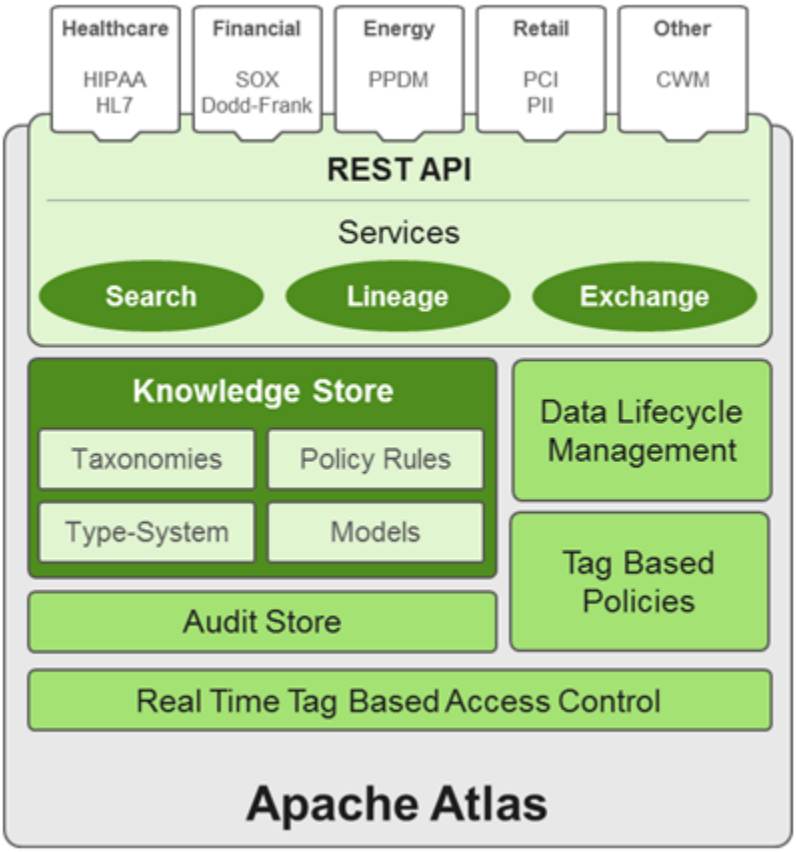
- Hive
- Ranger
- Sqoop
- Storm/Kafka (limited support)
- Falcon (limited support)

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-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 entities including databases, tables, and columns, thereby preventing security violations.

Ranger-Atlas Access Policies

- **Classification-based access controls:** A data entity 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.

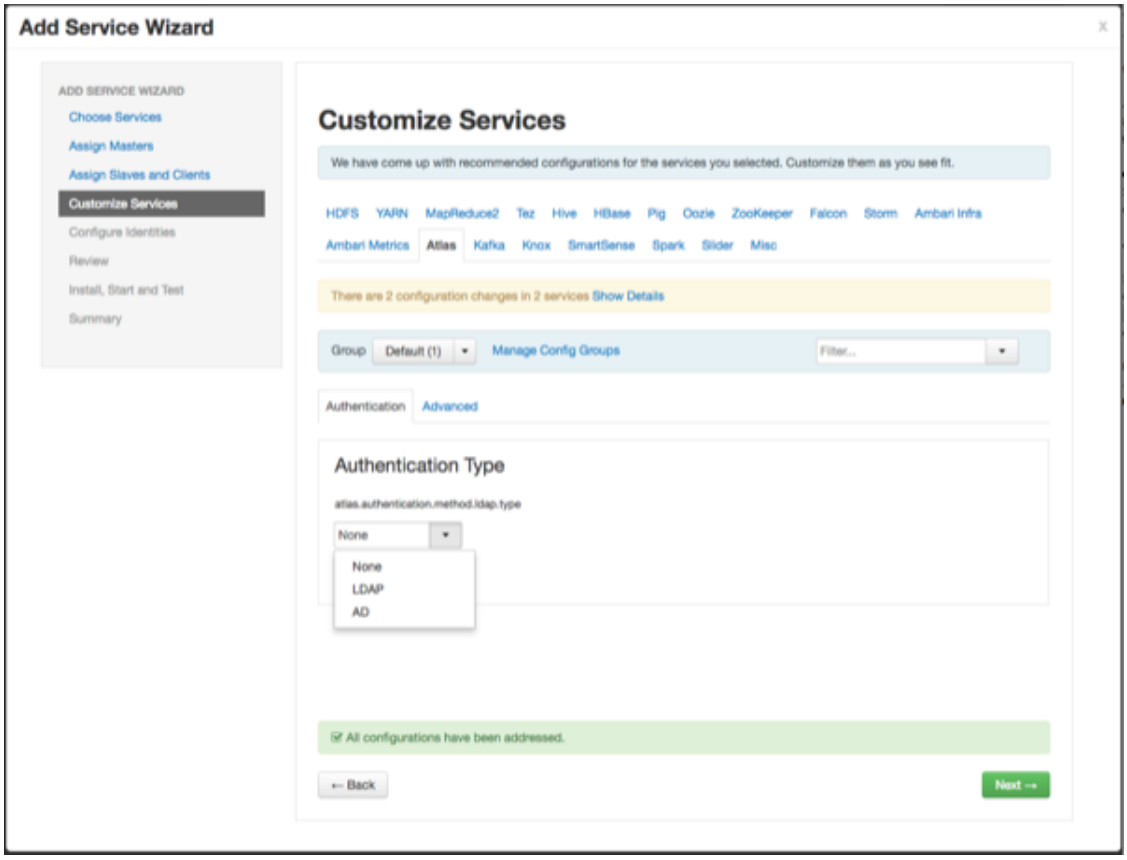
2.1.1. Apache Atlas Prerequisites

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).

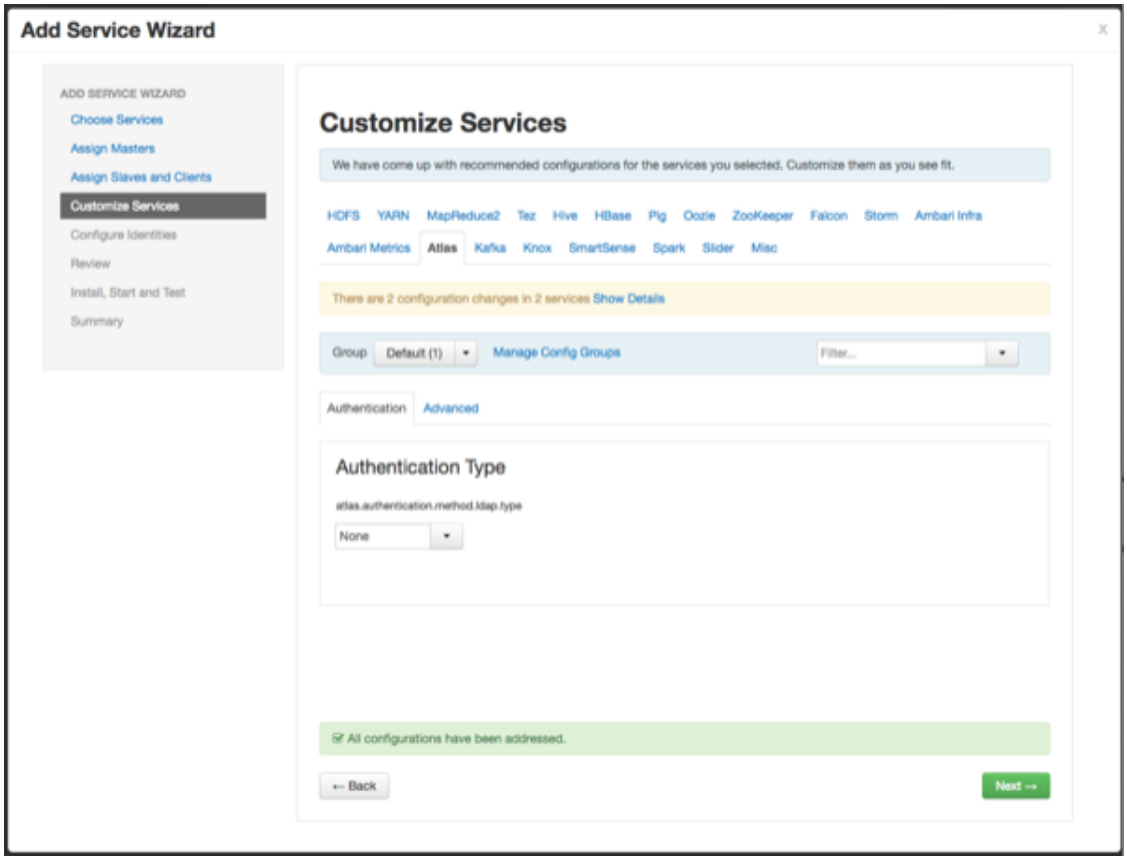
2.1.2. Authentication Settings

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

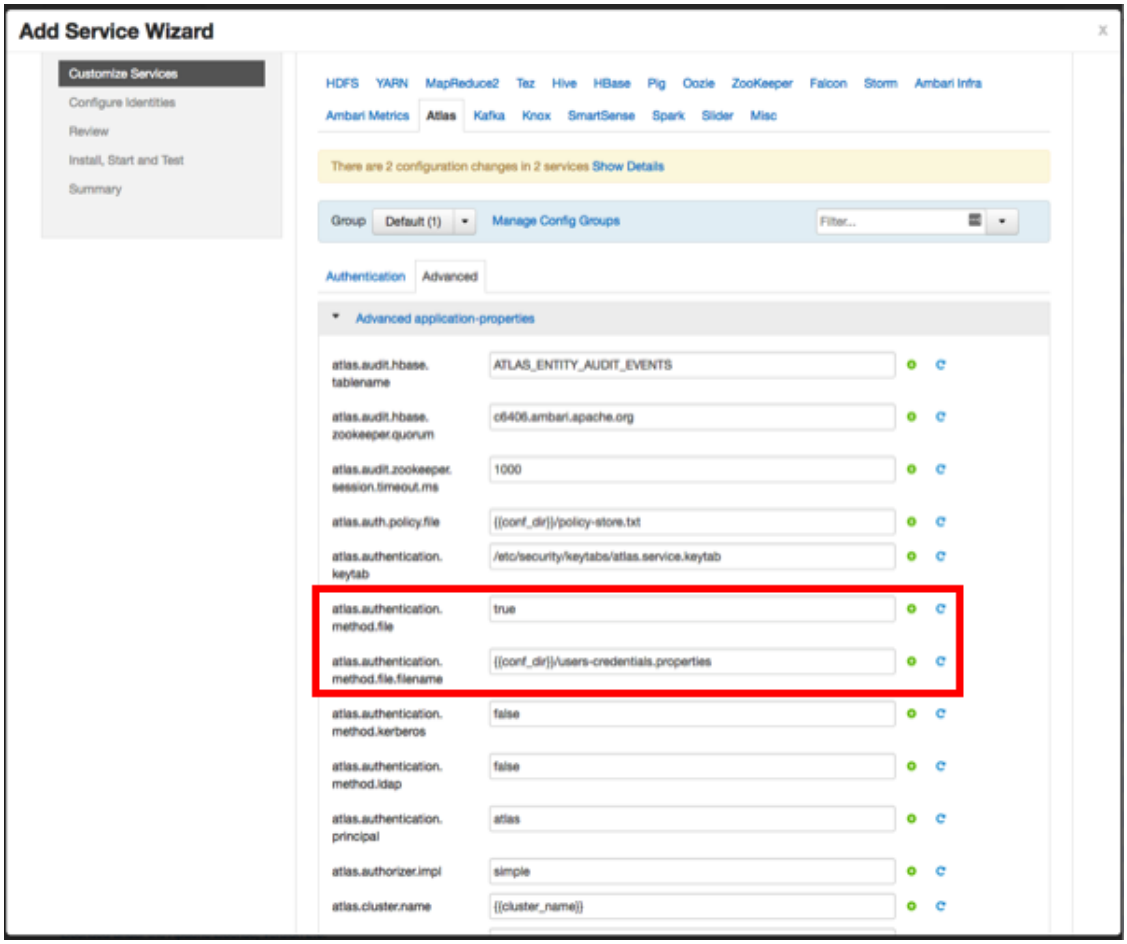
Select **None** to default to file-based authentication.



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 -
```

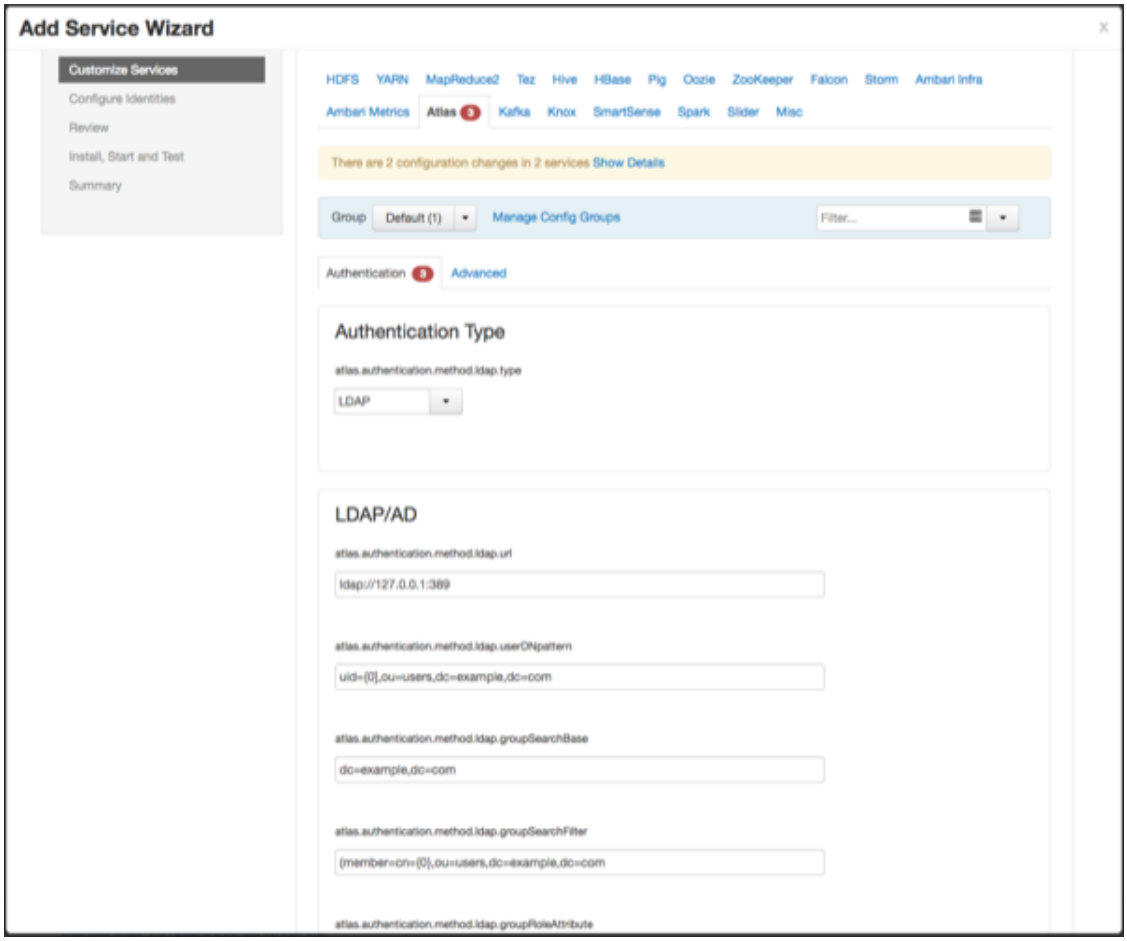
2.1.2.2. LDAP Authentication

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



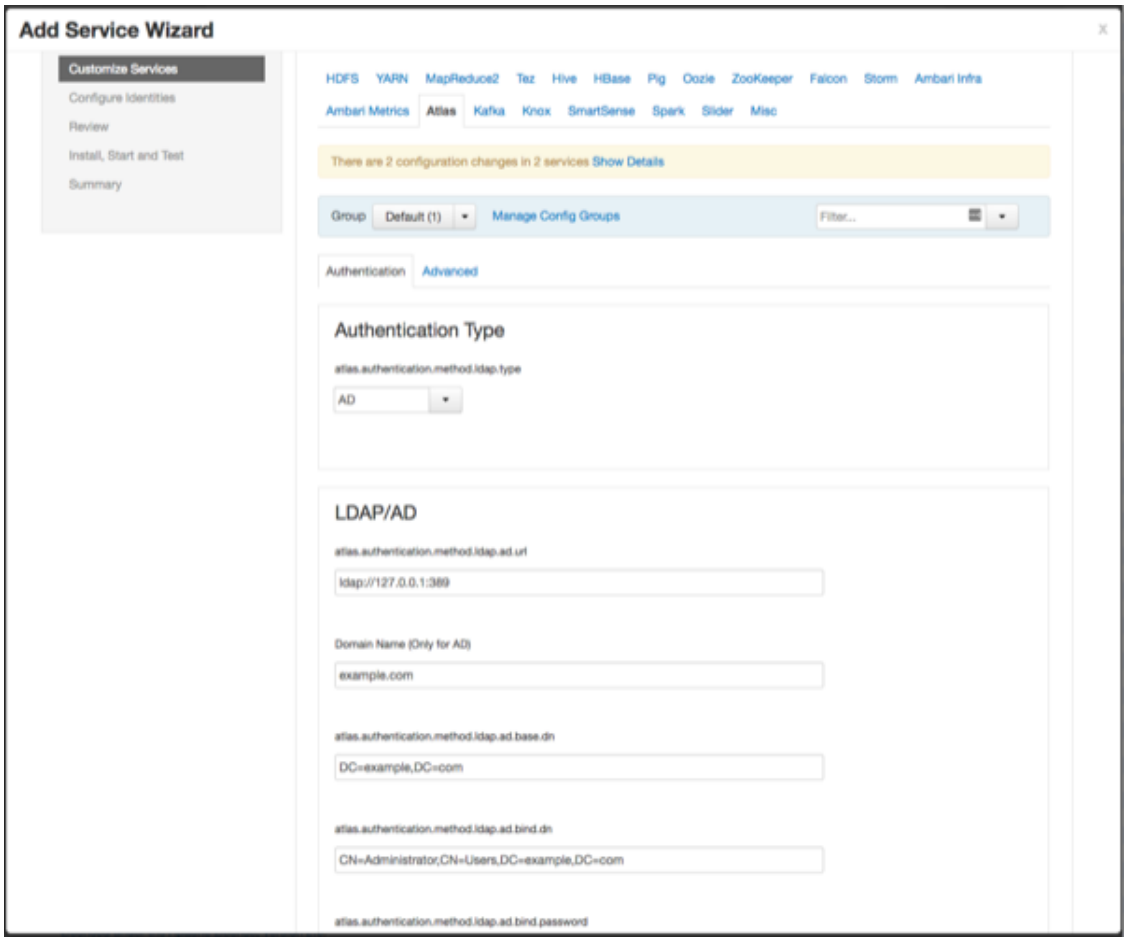
2.1.2.3. AD Authentication

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

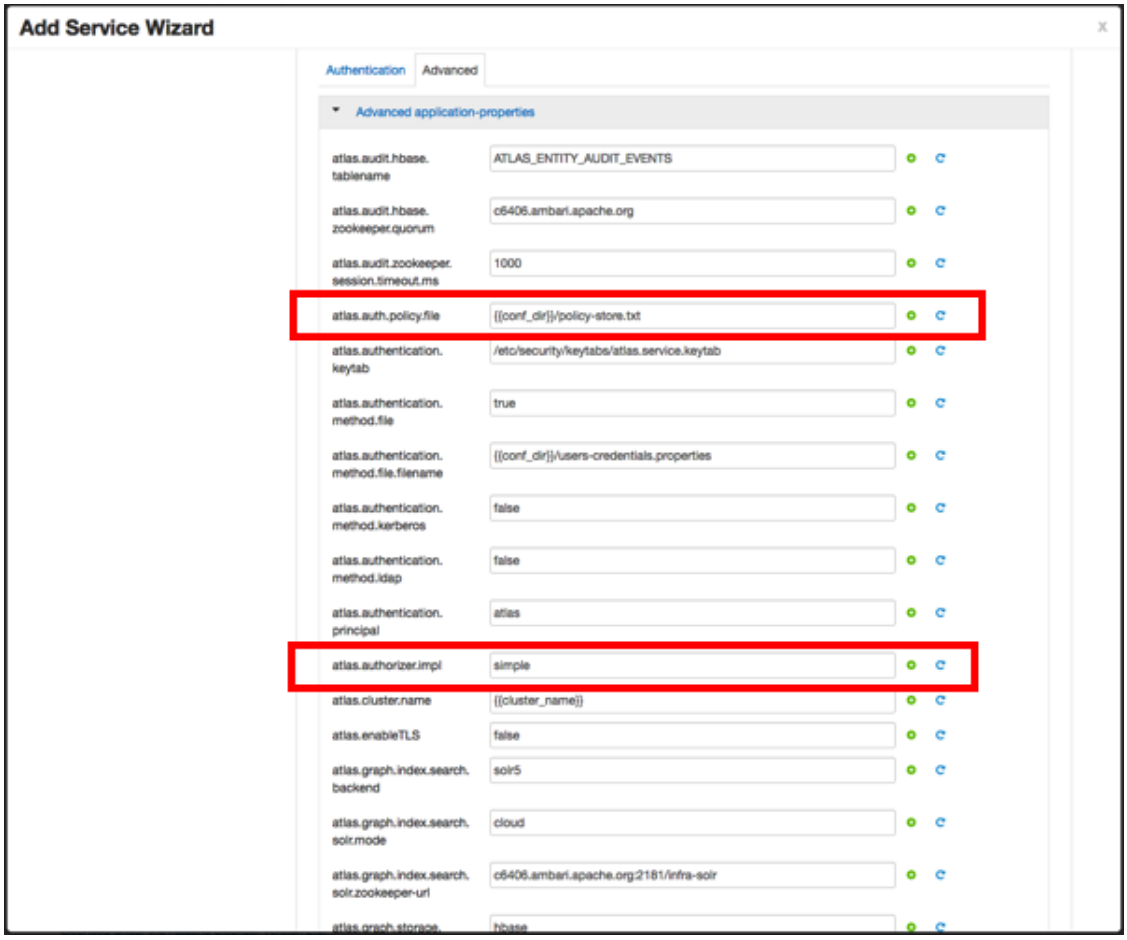
Two authorization methods are available for Atlas: Simple and Ranger.

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.

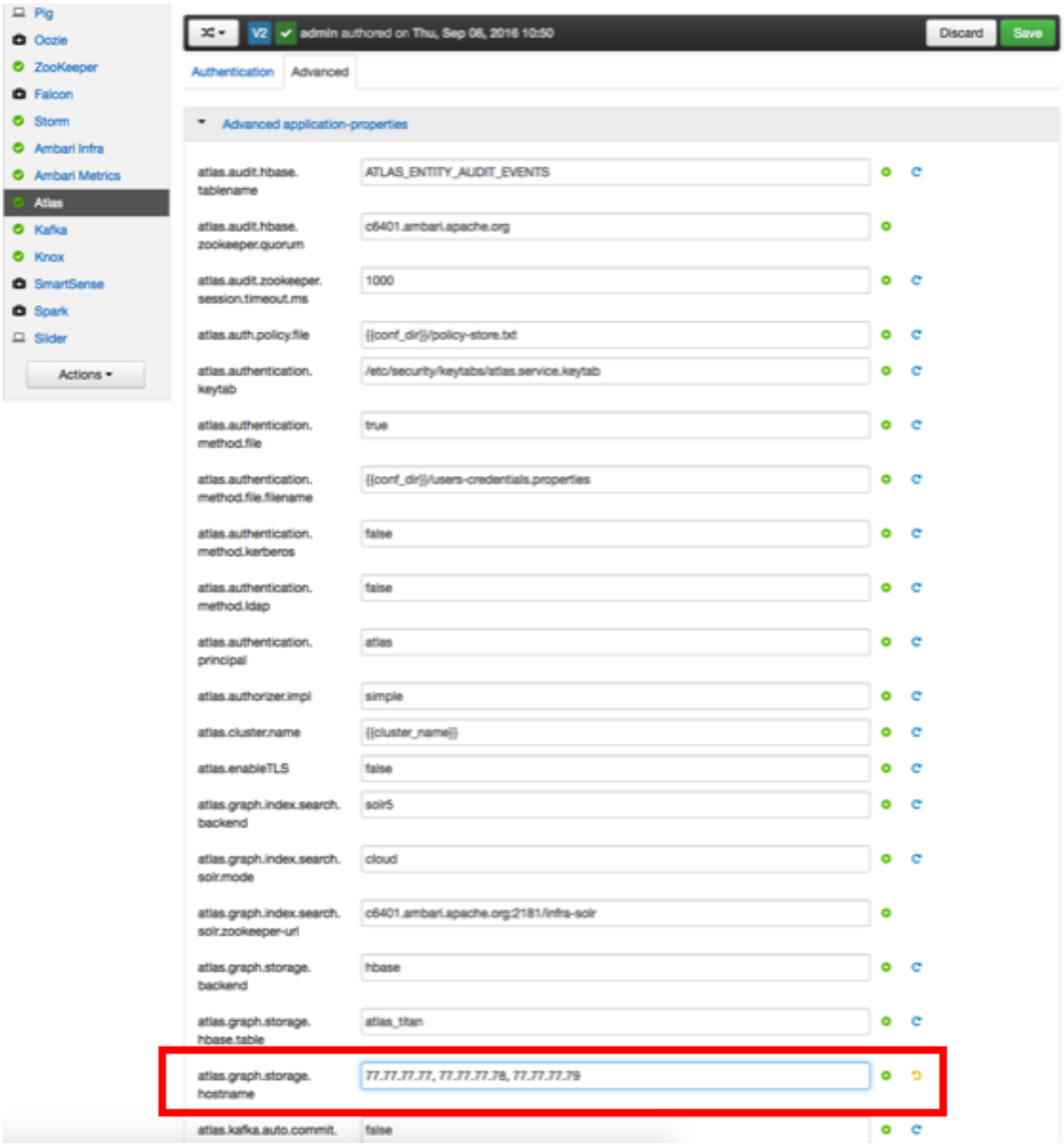
Note

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**.

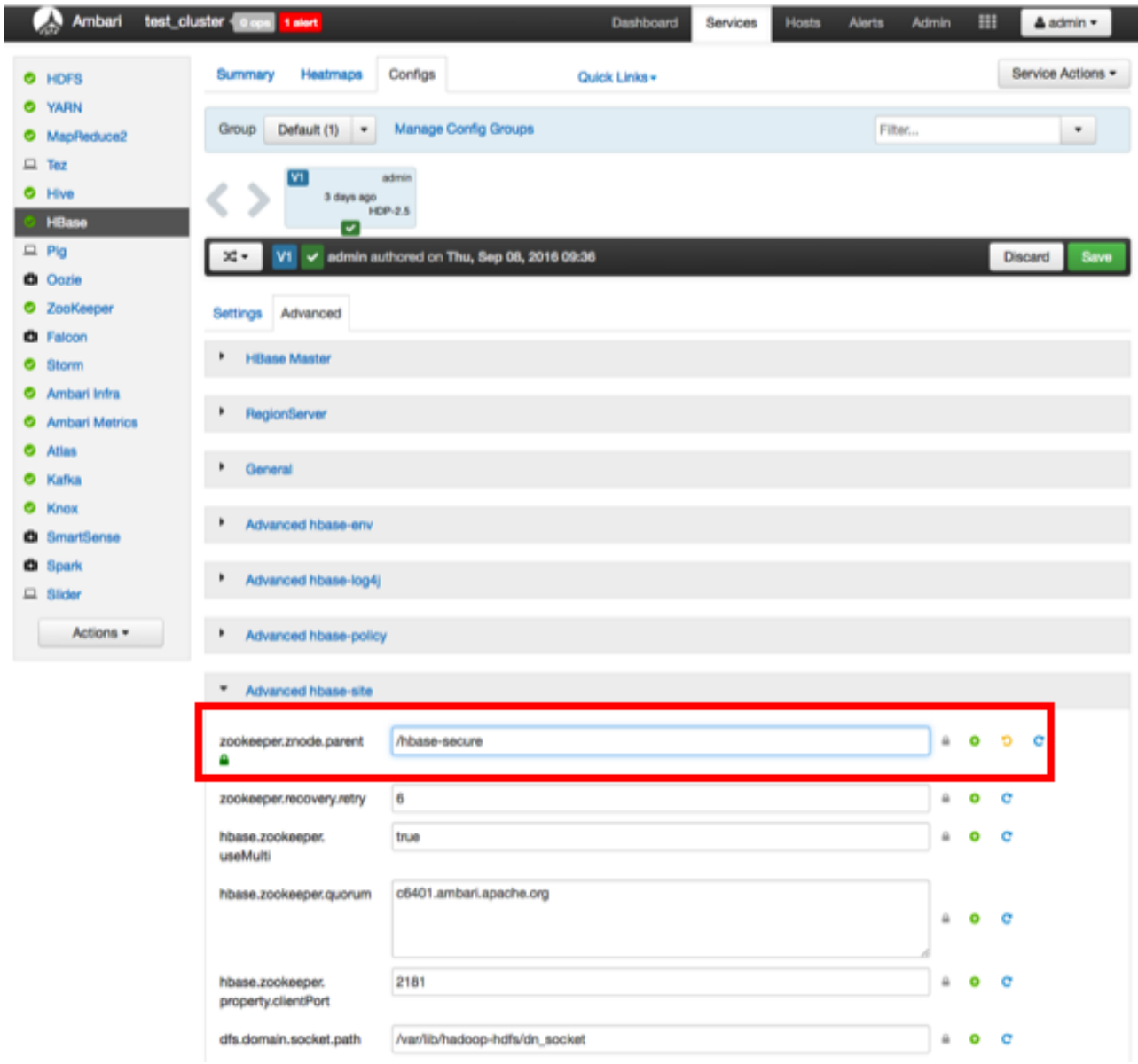
2. Set the value of the `atlas.graph.storage.hostname` property to the value of the distributed HBase ZooKeeper quorum. This value is a comma-separated list of the servers in the distributed HBase ZooKeeper quorum:

```
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 Entities

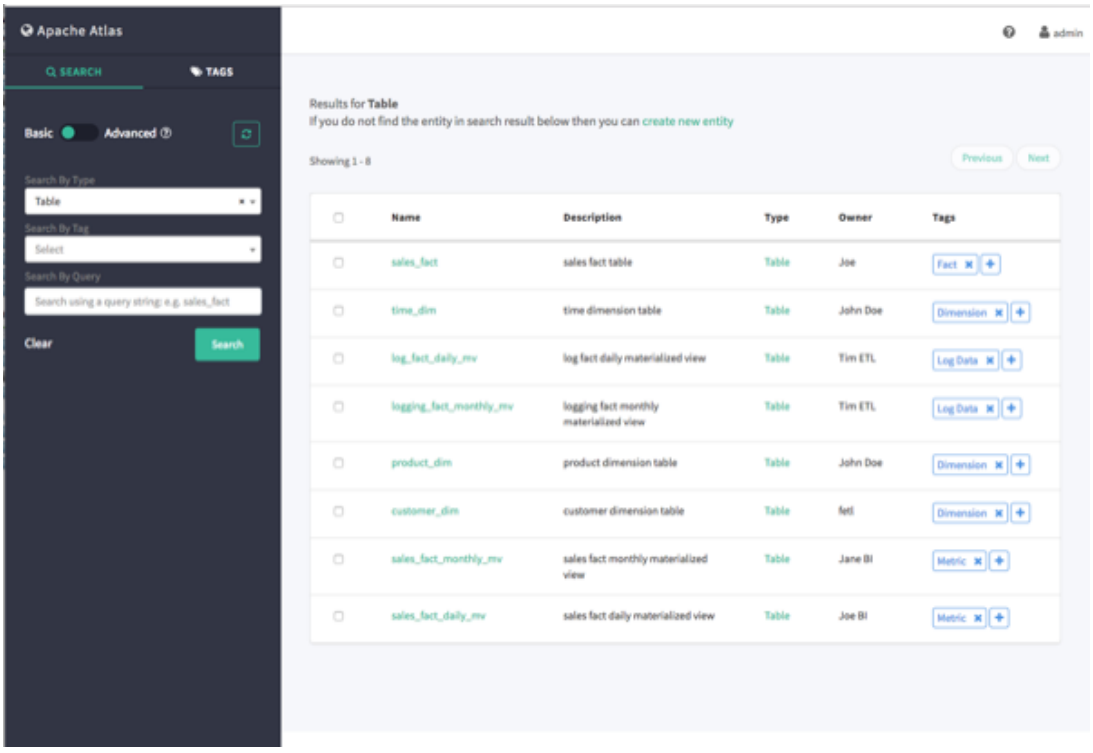
3.1. Using Text and DSL Search

You can search for entities using three search modes:

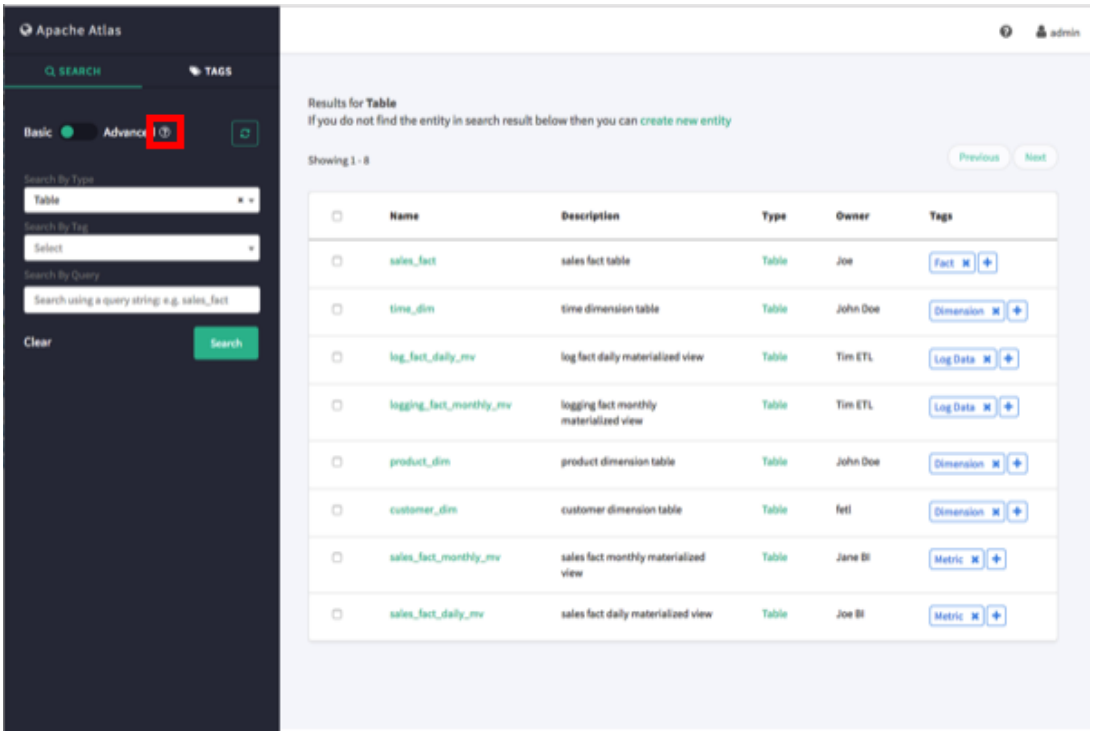
- Search by Type – search based on a selected Entity type.
- Search by Tag – search based on a selected Atlas tag.
- Search by Query – search using an Apache Atlas DSL query. Atlas DSL (Domain-Specific Language) is a SQL-like query language that enables you to search metadata using complex queries.

1. To search for entities, click **SEARCH** on the Atlas web UI. Select an entity type, an Atlas tag, or enter an Atlas DSL search query, then click **Search** to display a list of the entities associated with the specified search criteria.

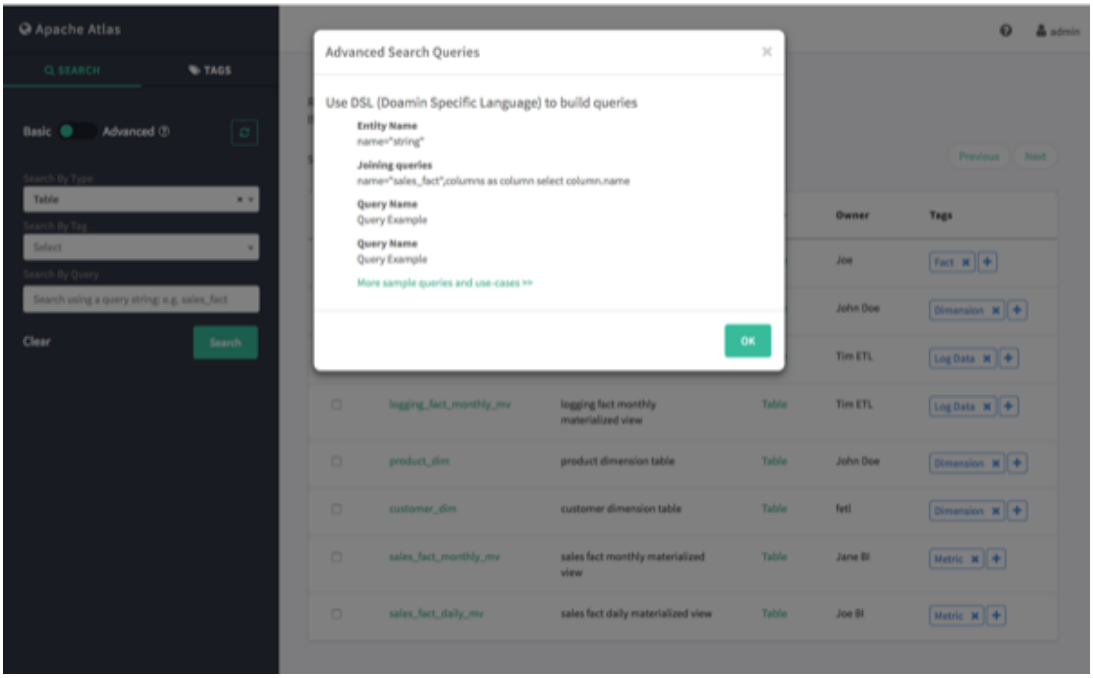
   You can also combine search criteria (search for a type with a specific name, for example). In the example below, we searched for the Table entity type.



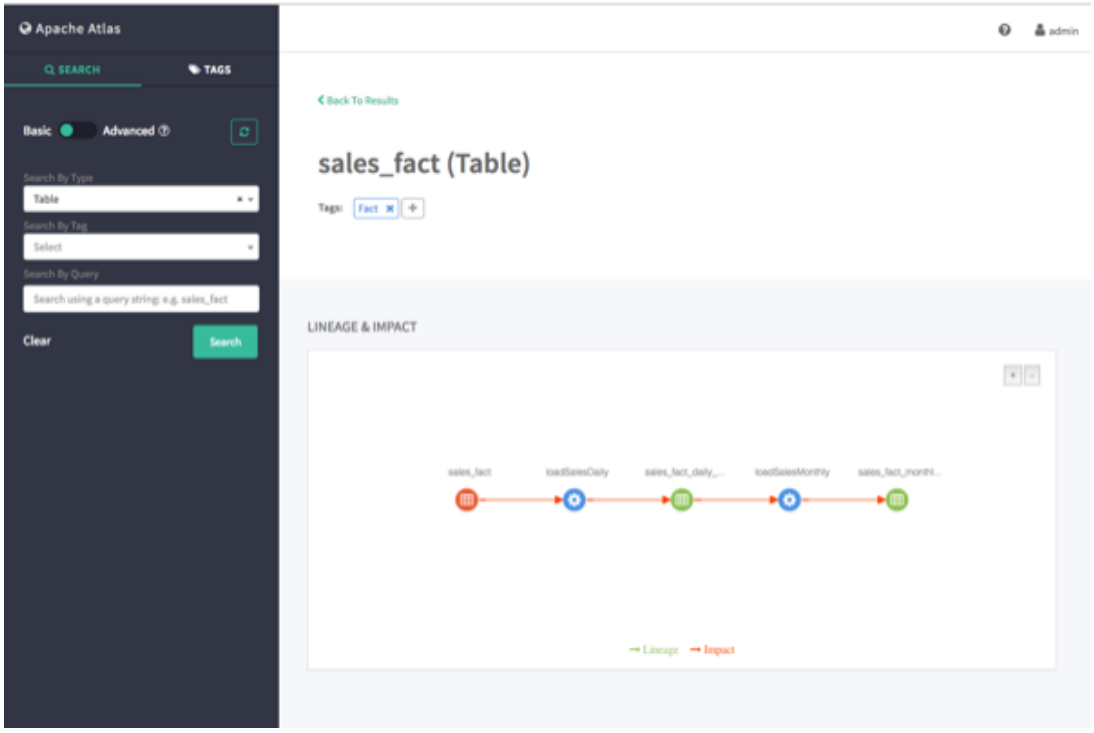
   To display more information about Atlas DSL queries, click the question mark symbol next to the **Advanced** label above the search boxes.



The Advanced Search Queries lists example queries, along with a link to the Apache Atlas DSL query documentation:

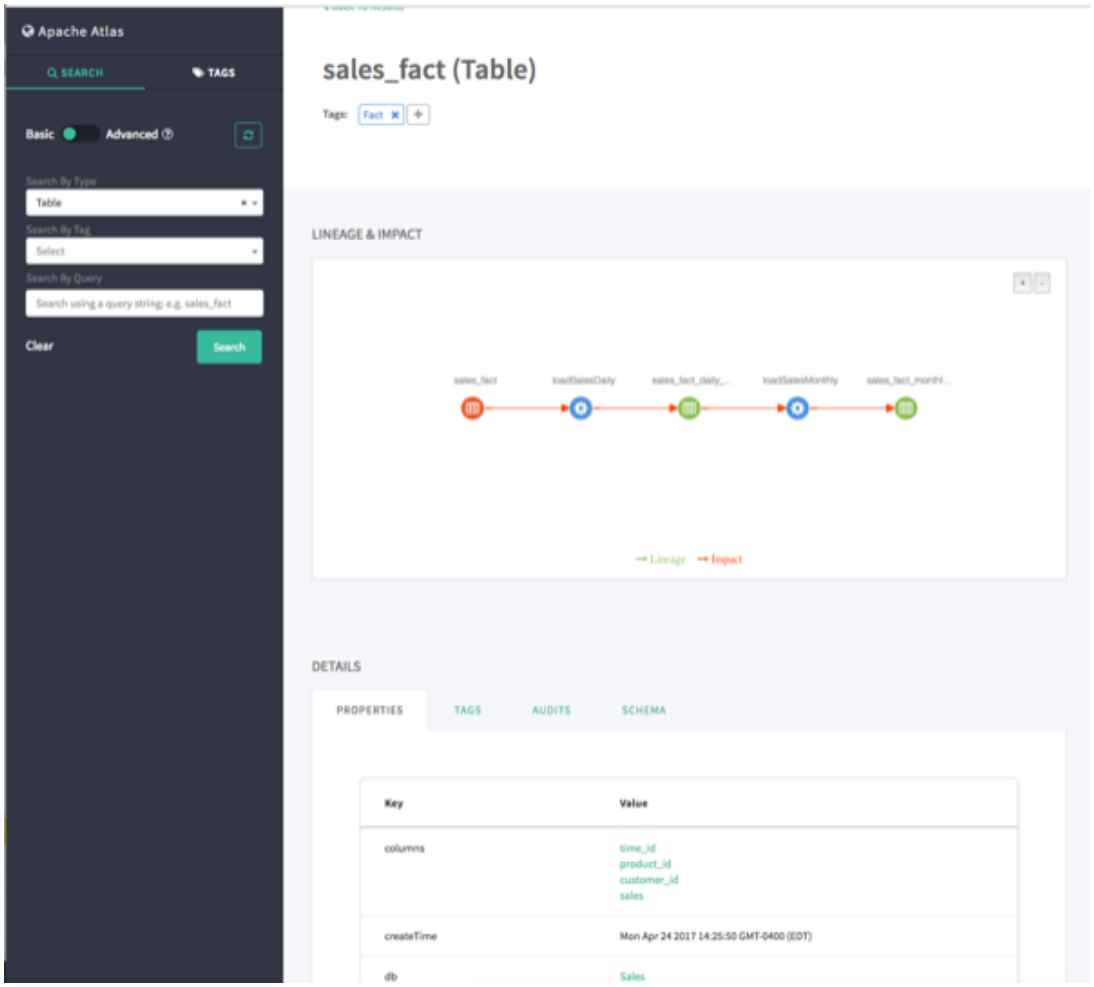


2. To view detailed information about an entity, click the entity in the search results list. In the example below, we selected the "sales_fact" table from the list of search results.

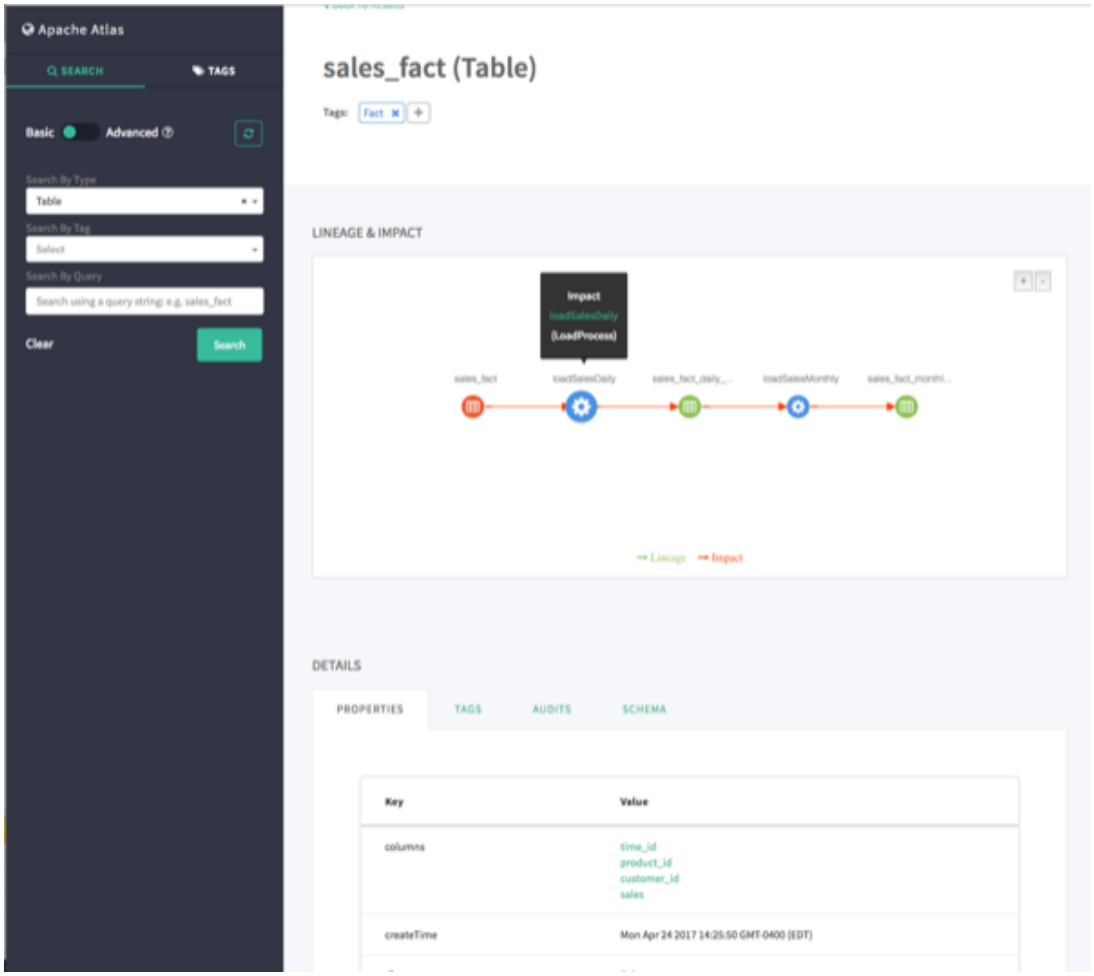


3.2. Viewing Entity Data Lineage & Impact

1. Data lineage and impact is displayed when you select an entity. In the following example, we ran a Type search for `Table`, and then selected the "sales_fact" entity. Data lineage and image 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.



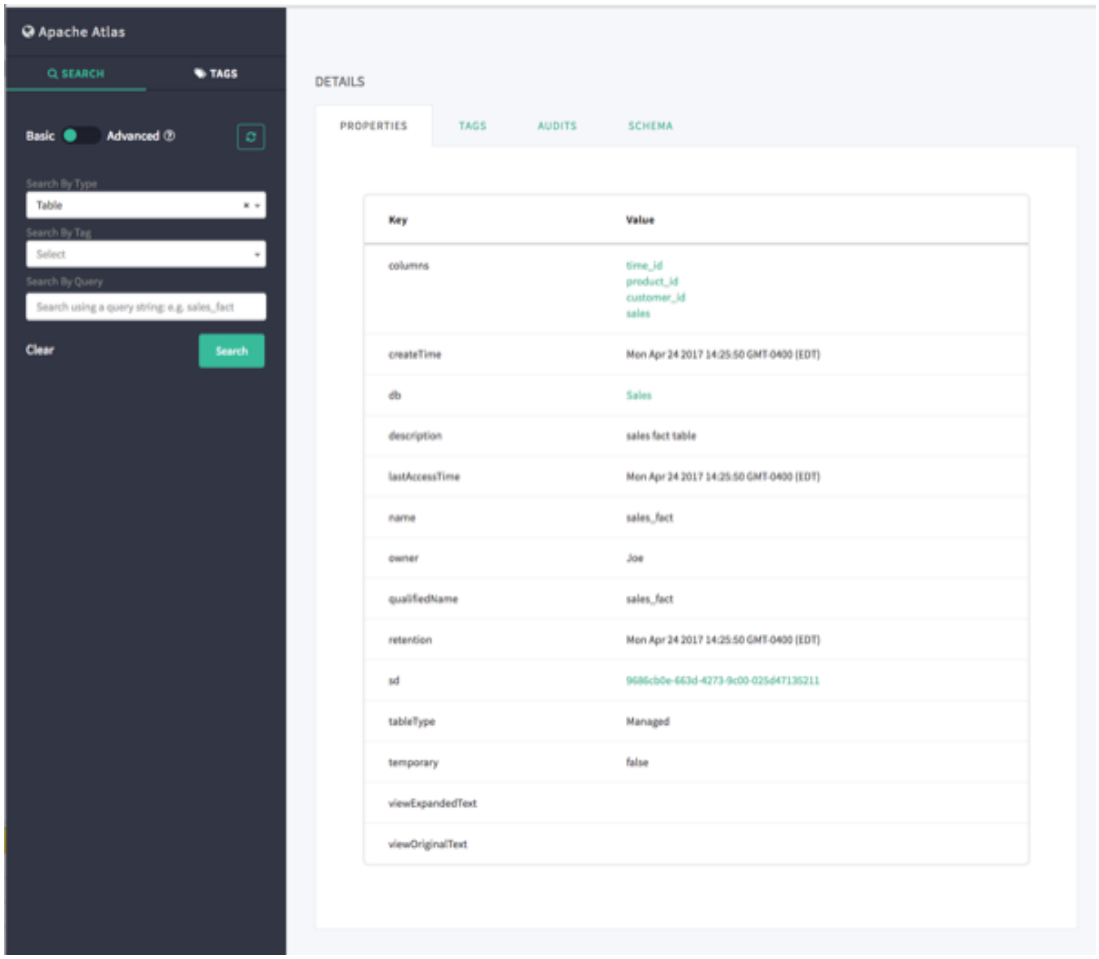
2. Moving the cursor over an icon displays a pop-up with more information about the action that was performed. In the following example, we can see that a query was used to create the "loadSalesDaily" table from the "sales_fact" table.



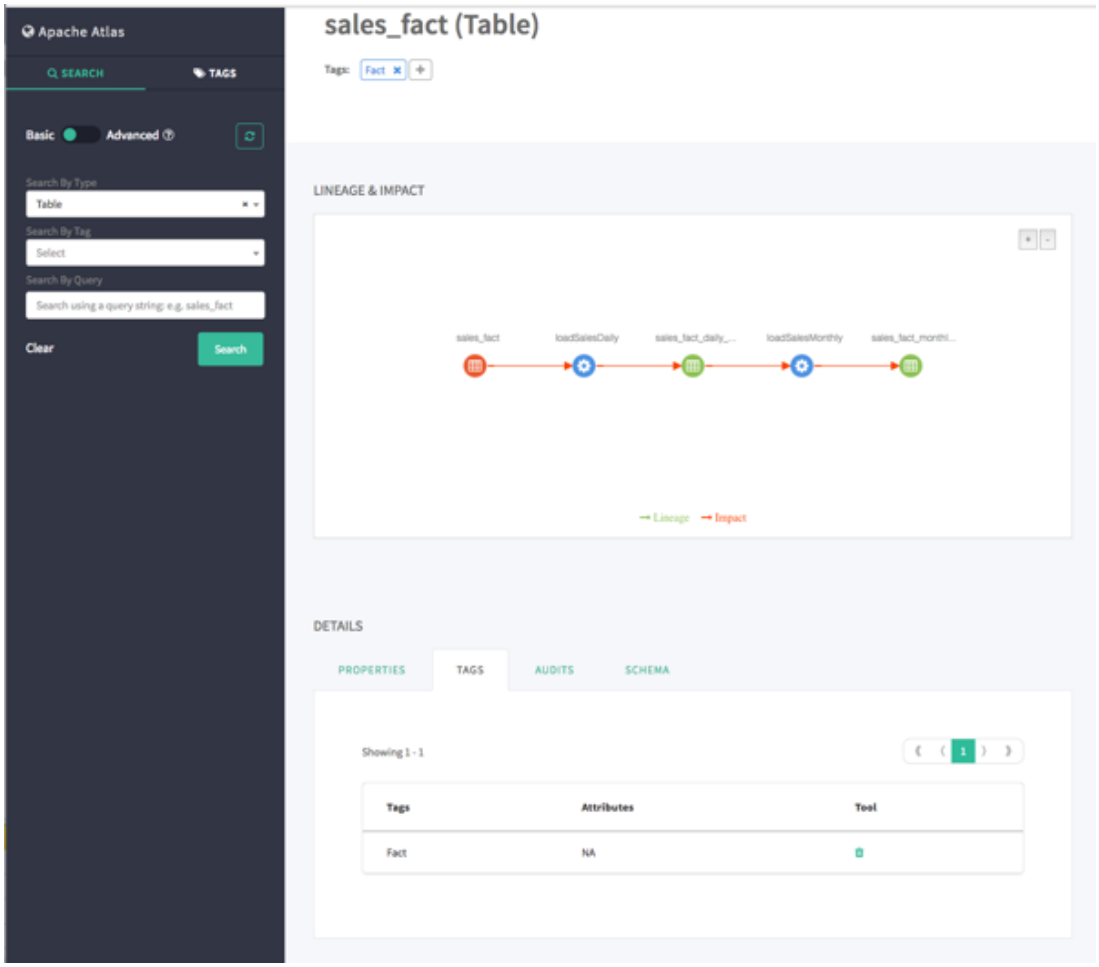
3.3. Viewing Entity Details

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

- The Properties tab displays all of the entity properties.

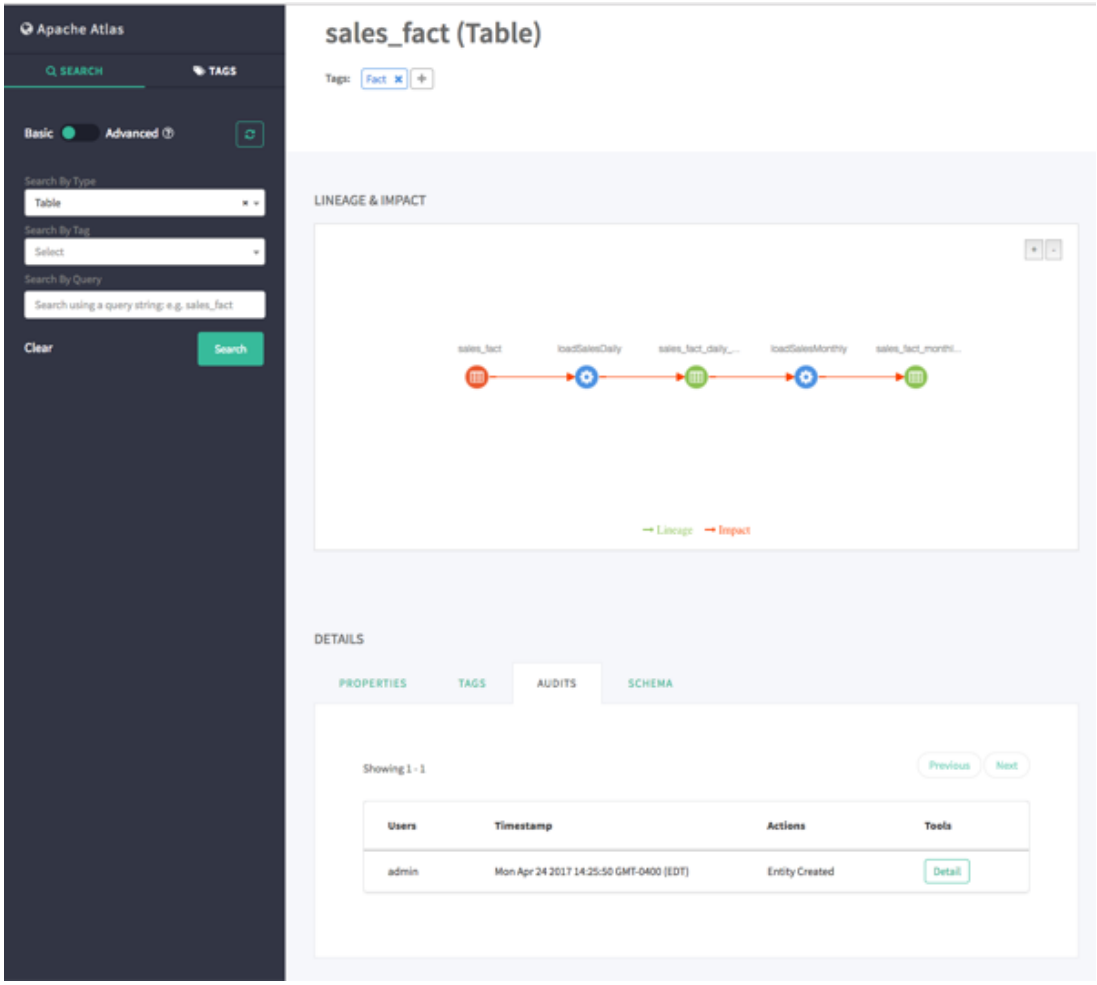


- Click the Tags tab to display the tags associated with the entity. In this case, the "fact" tag has been associated with the "sales_fact" table.

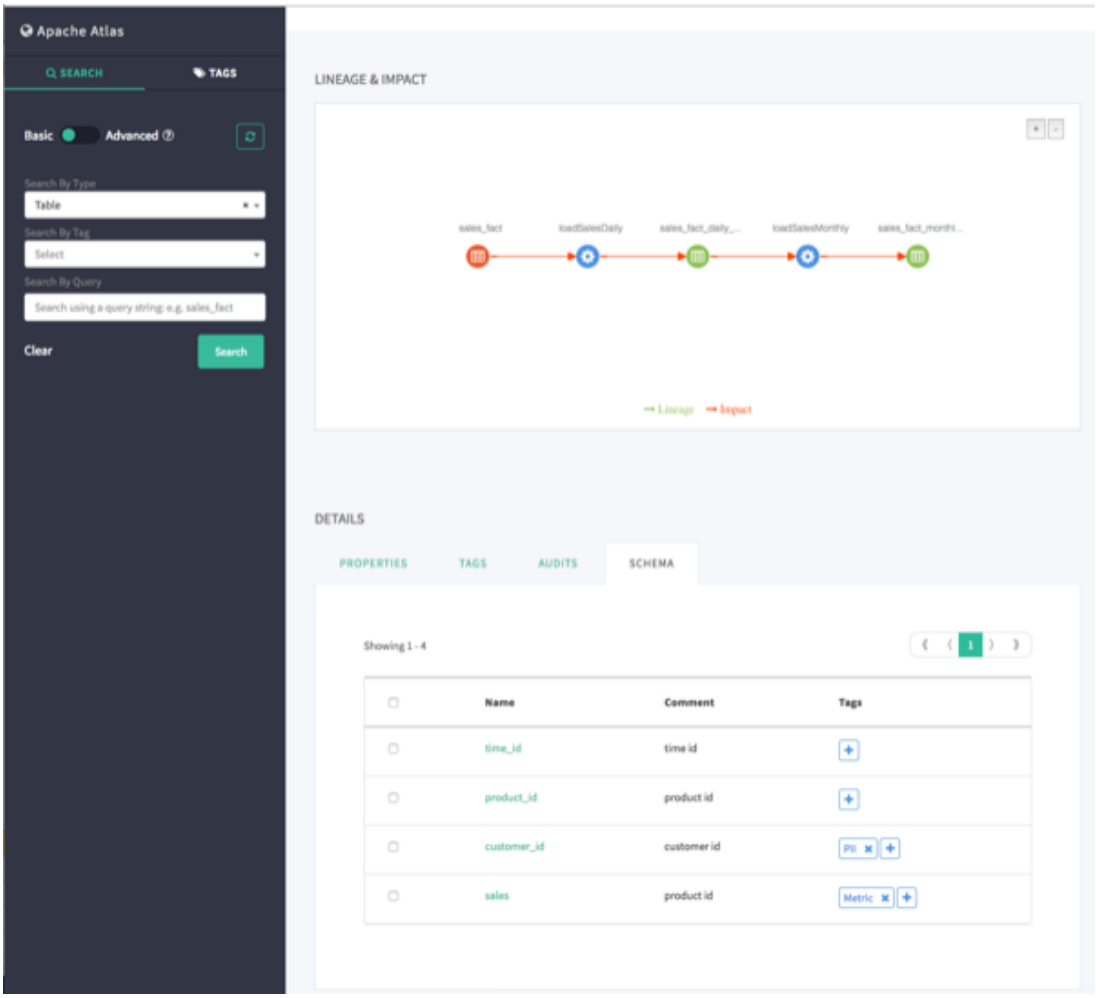


- If the Atlas Taxonomy has been enabled, the Terms tab lists the taxonomy terms that have been associated with the entity. The Terms tab is not displayed if the Taxonomy has not been enabled.

- The Audits tab provides a complete audit trail of all events in the entity history. You can use the Detail button next to each action to view more details about the event.



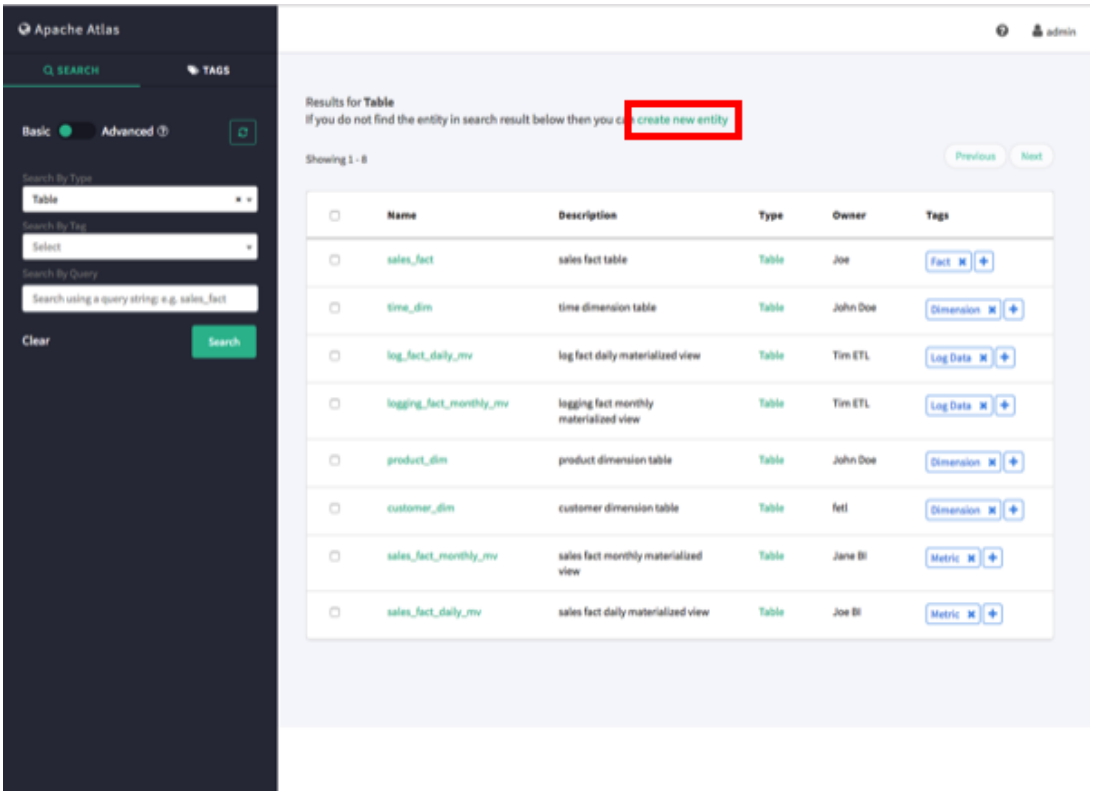
- The Schema tab shows schema information, in this case the columns for the table. We can also see that a PII tag has been associated with the "customer_id" column.



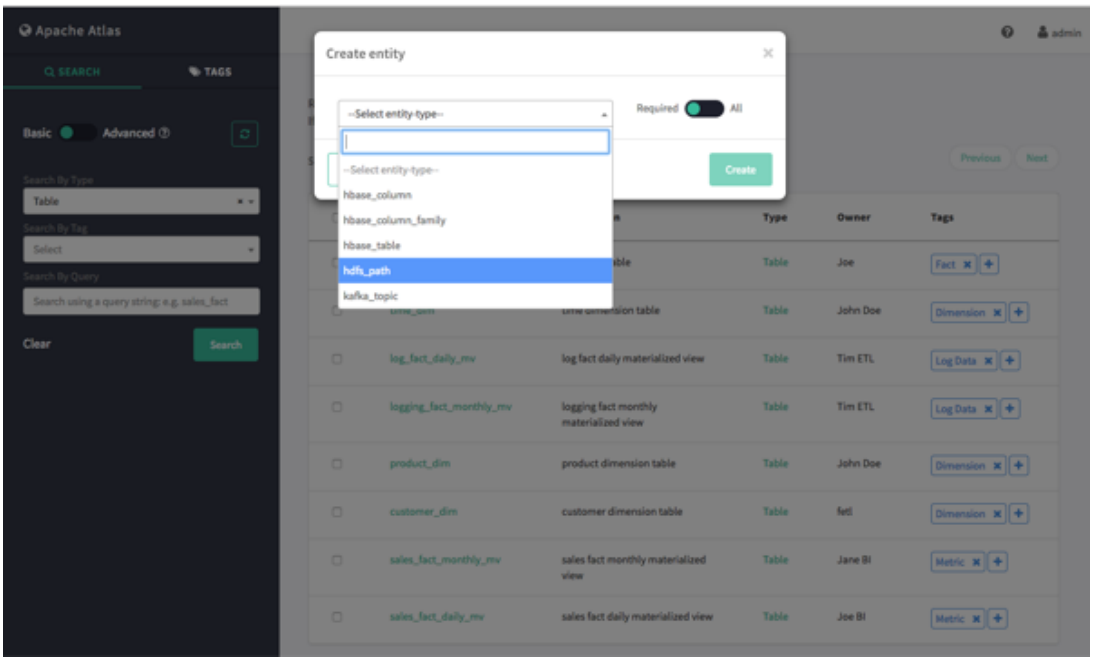
3.4. Manually Creating Entities

Currently there is no Atlas hook for HBase, HDFS, or Kafka. For these components, you must manually create entities in Atlas. You can then associate tags with these entities and control access using Ranger tag-based policies.

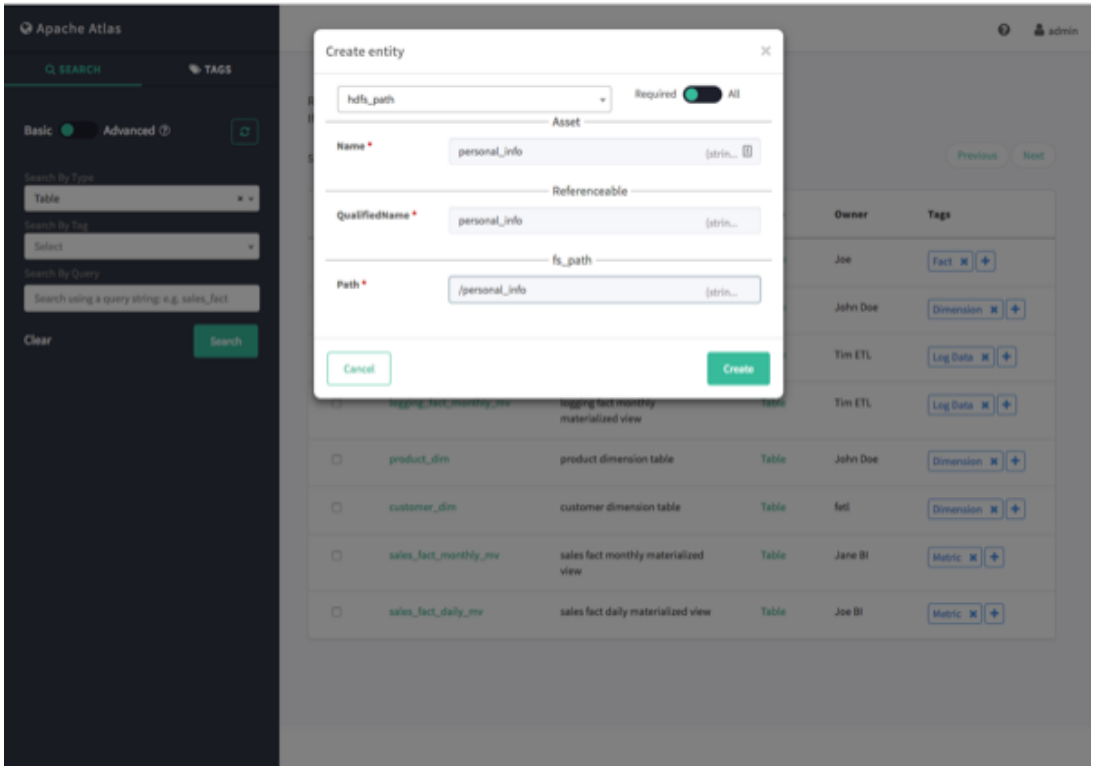
1. On the Atlas web UI Search page, click the **create new entity** link at the top of the page.



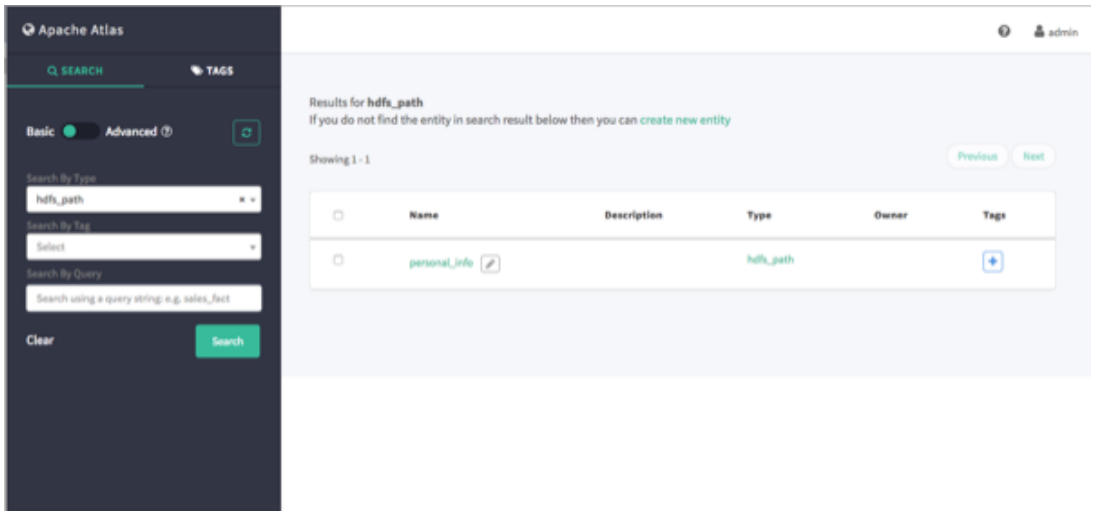
2. On the Create Entity pop-up, select an entity type.



3. Enter the required information for the new entity. Click **All** to display both required and non-required information. Click **Create** to create the new entity.



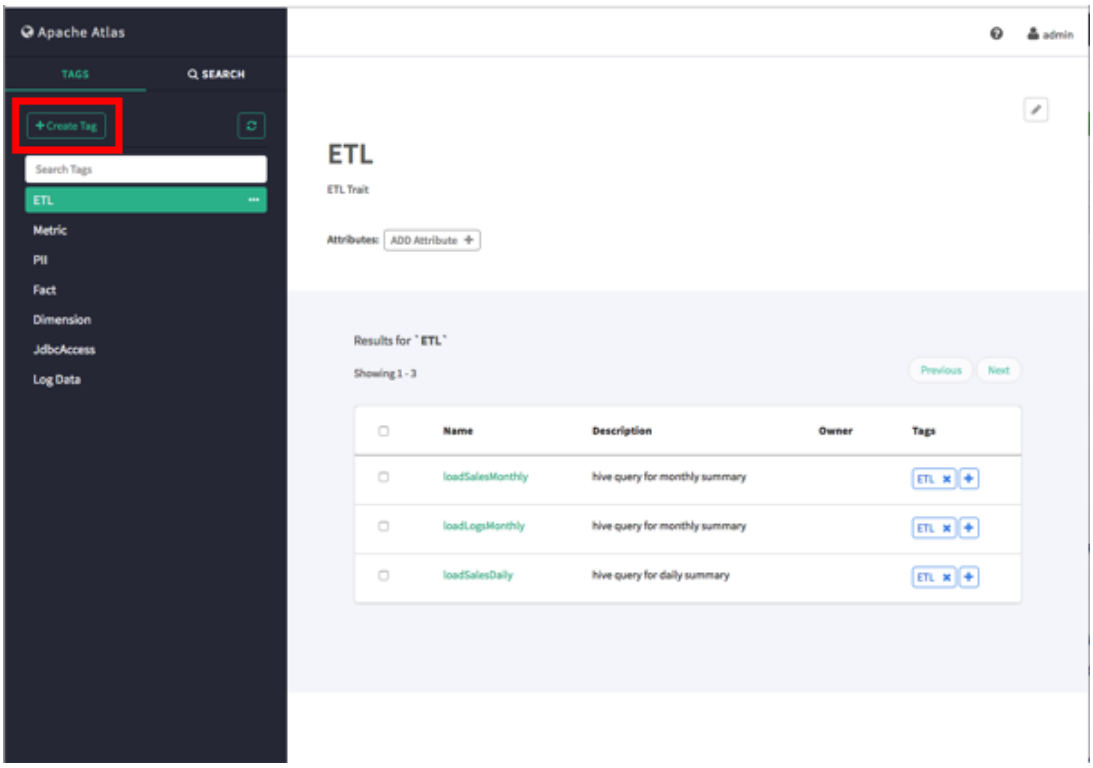
4. The entity is created and returned in search results for the applicable entity type. You can now associate tags with the new entity and control access to the entity with Ranger tag-based policies.



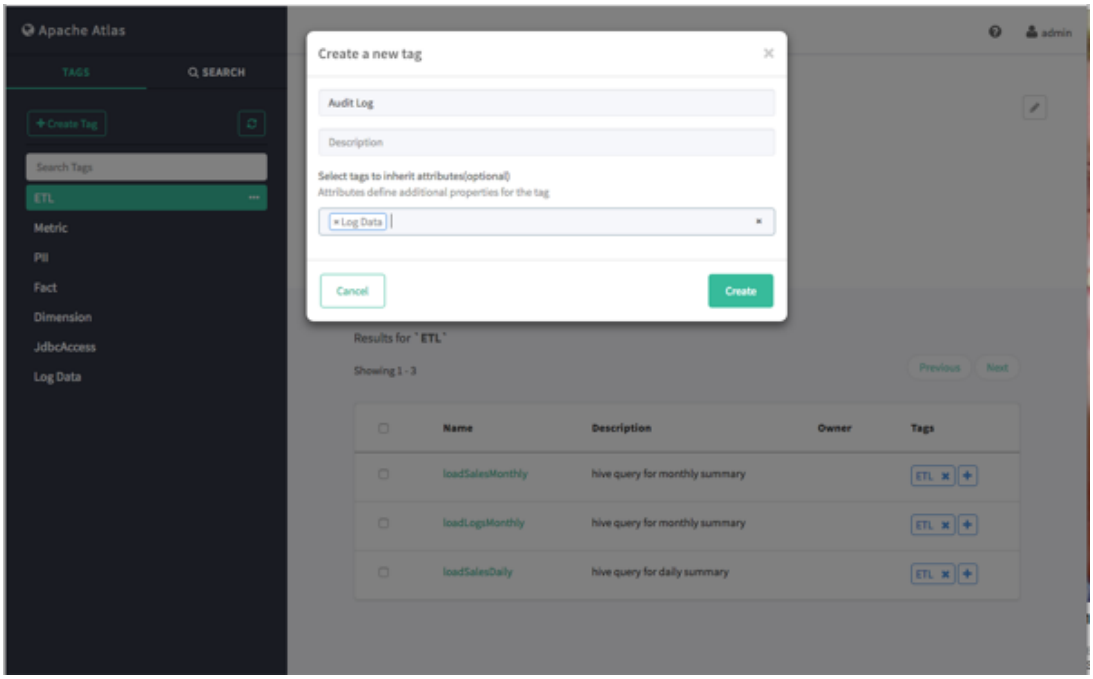
4. Working with Atlas Tags

4.1. Creating Atlas Tags

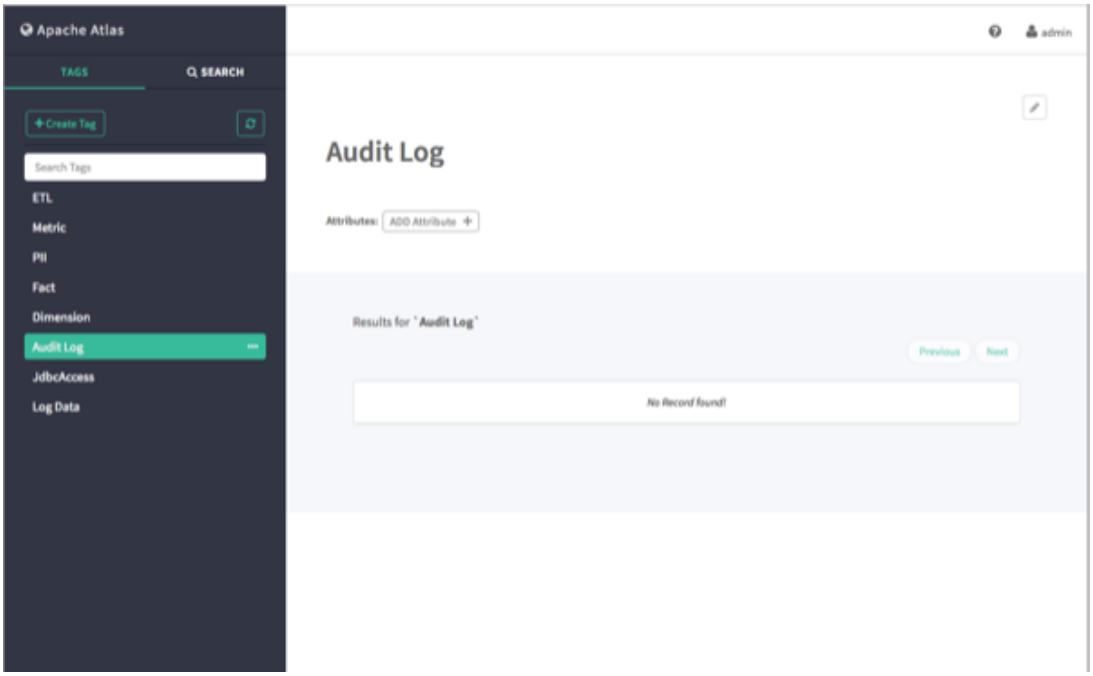
1. On the Atlas web UI, click **TAGS**, then click **Create Tag**.



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.

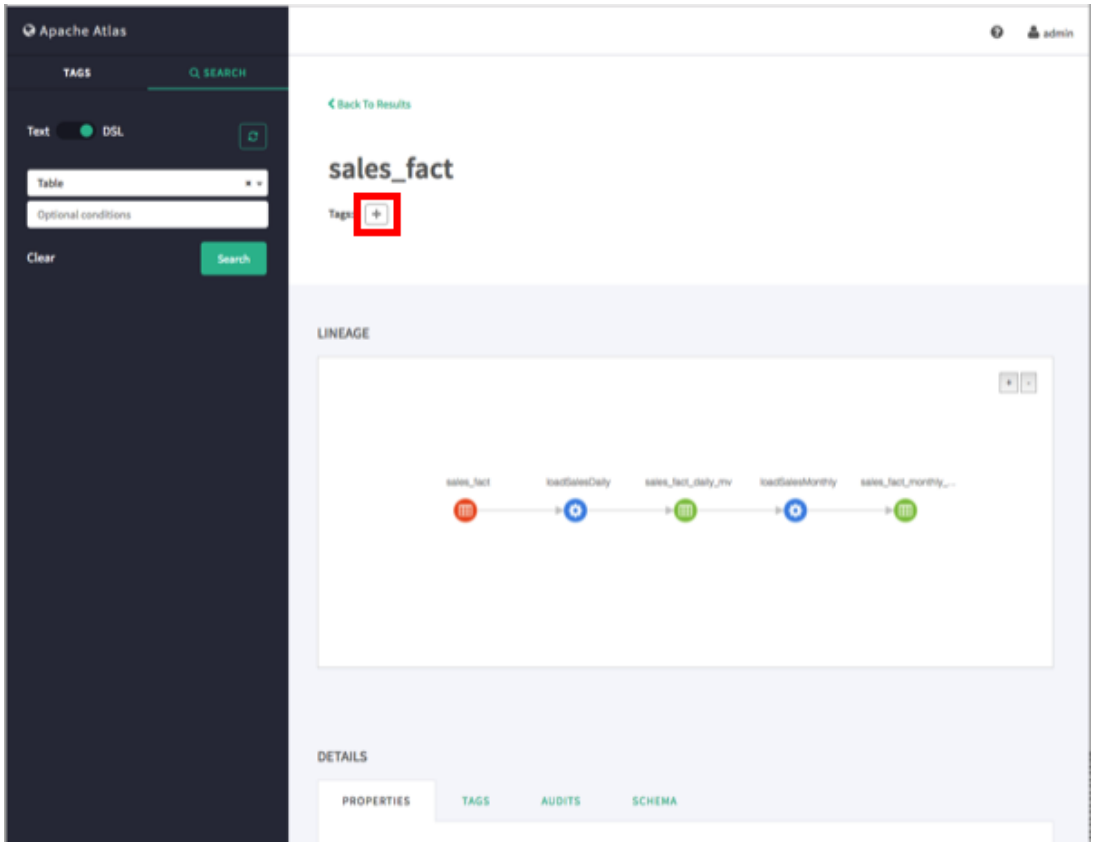


3. The new tag appears in the Tags list.

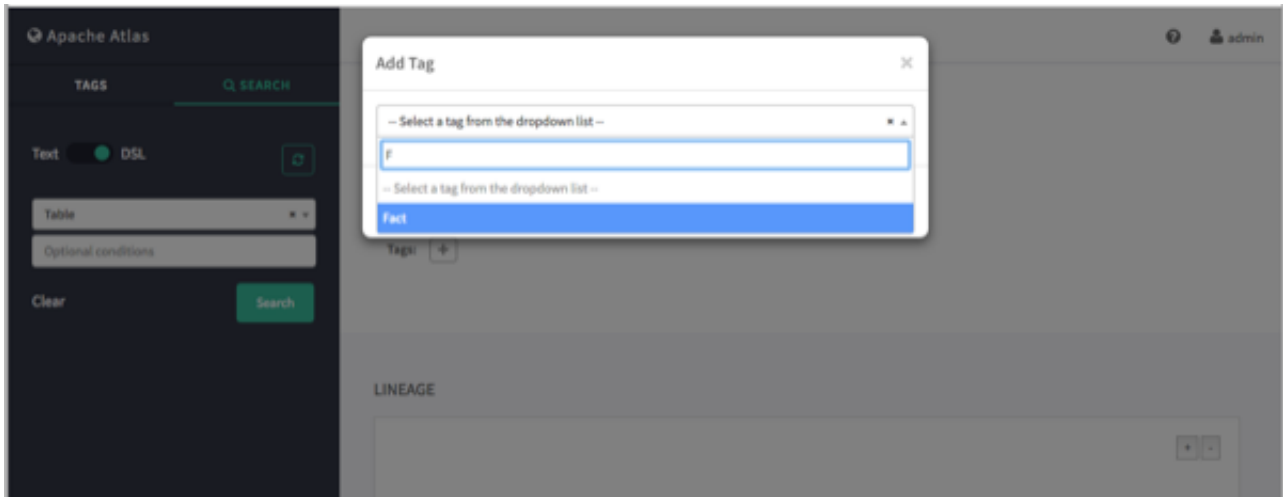


4.2. Associating Tags with Entities

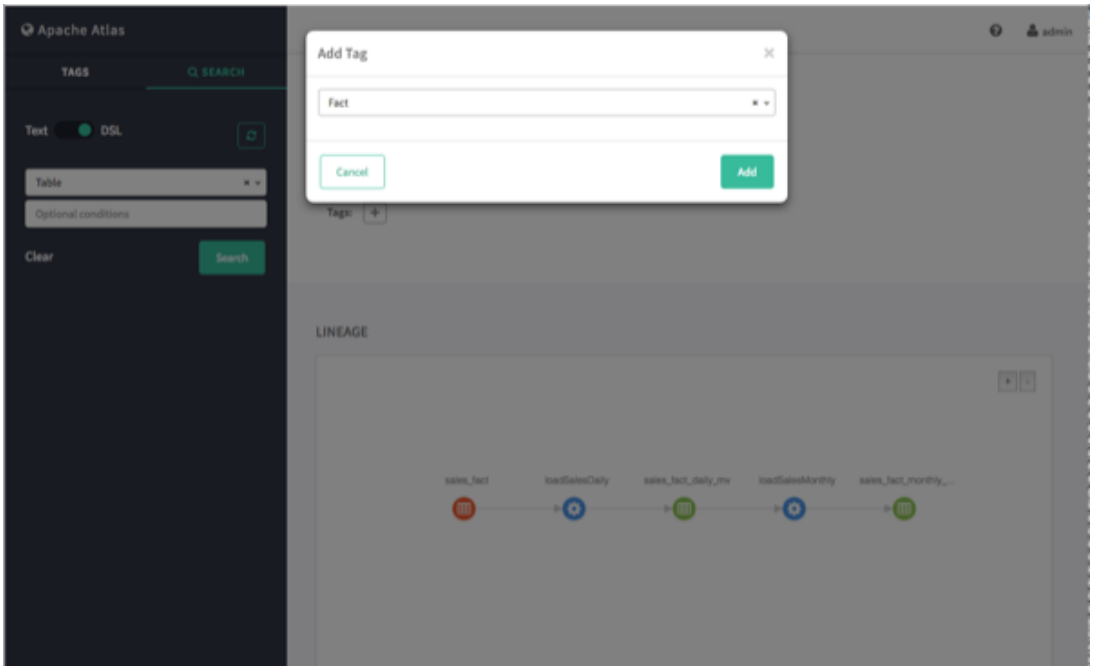
1. Select an asset. In the example below, we searched for all `Table` entities, and then selected the "sales_fact" table from the list of search results. To associate a tag with an asset, click the + icon next to the **Tags:** label.



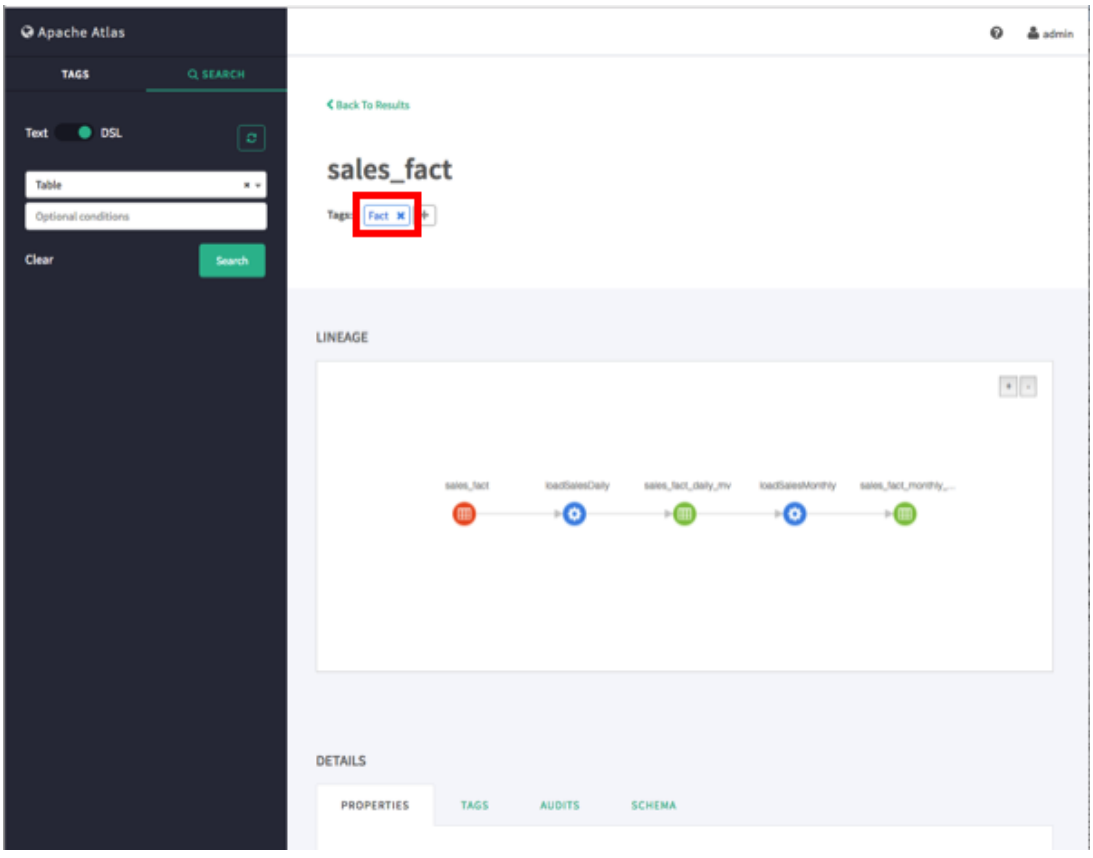
2. On the Add Tag pop-up, click **Select Tag**, then select the tag you would like to associate with the asset. You can filter the list of tags by typing text in the Select Tag box.



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.

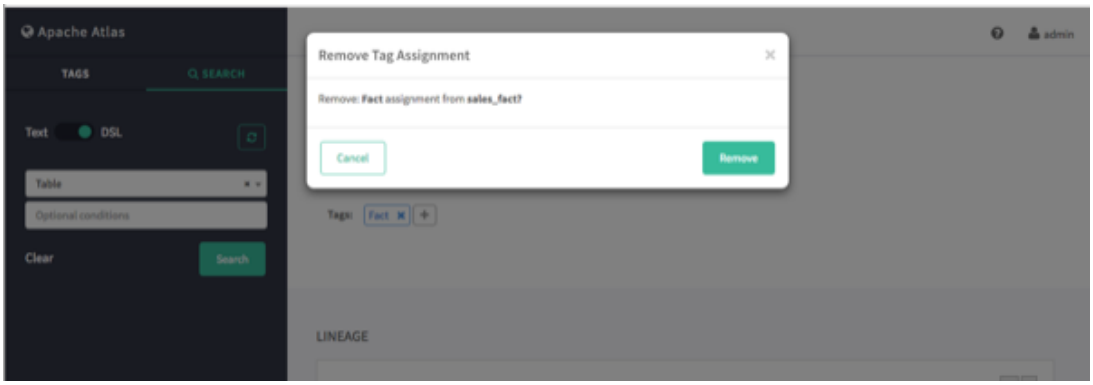


4. The new tag is displayed next to the **Tags:** label on the asset page.



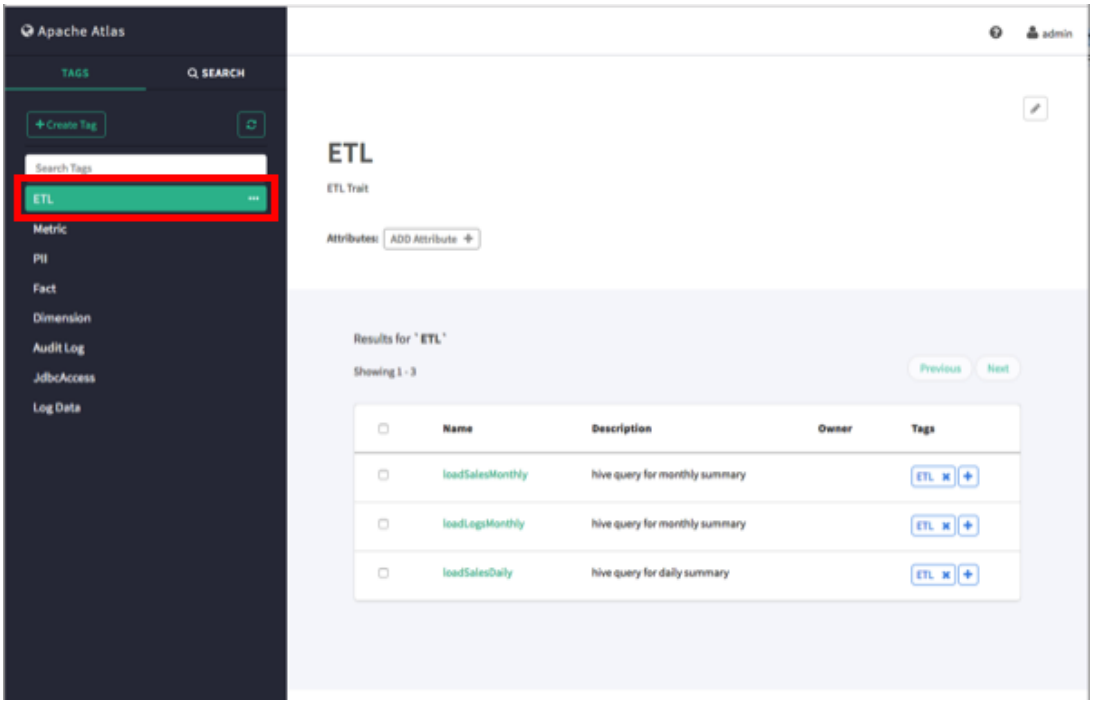
5. You can view details about a tag by clicking the tag name on the tag label.

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.

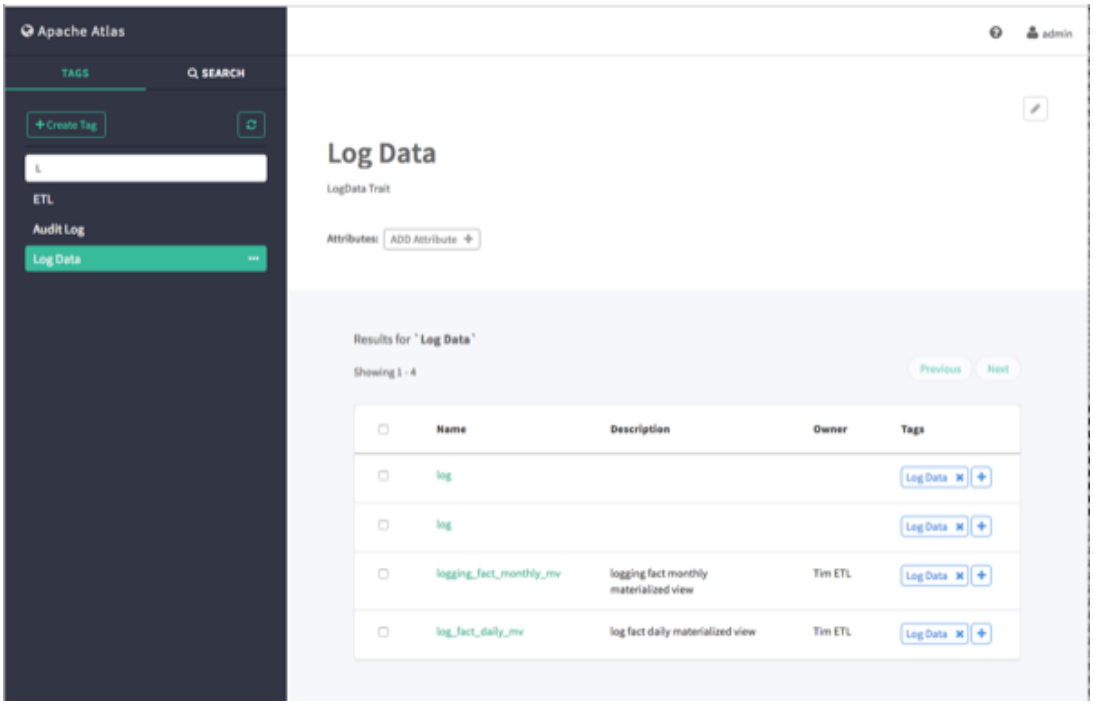


4.3. Searching for Entities Associated with Tags

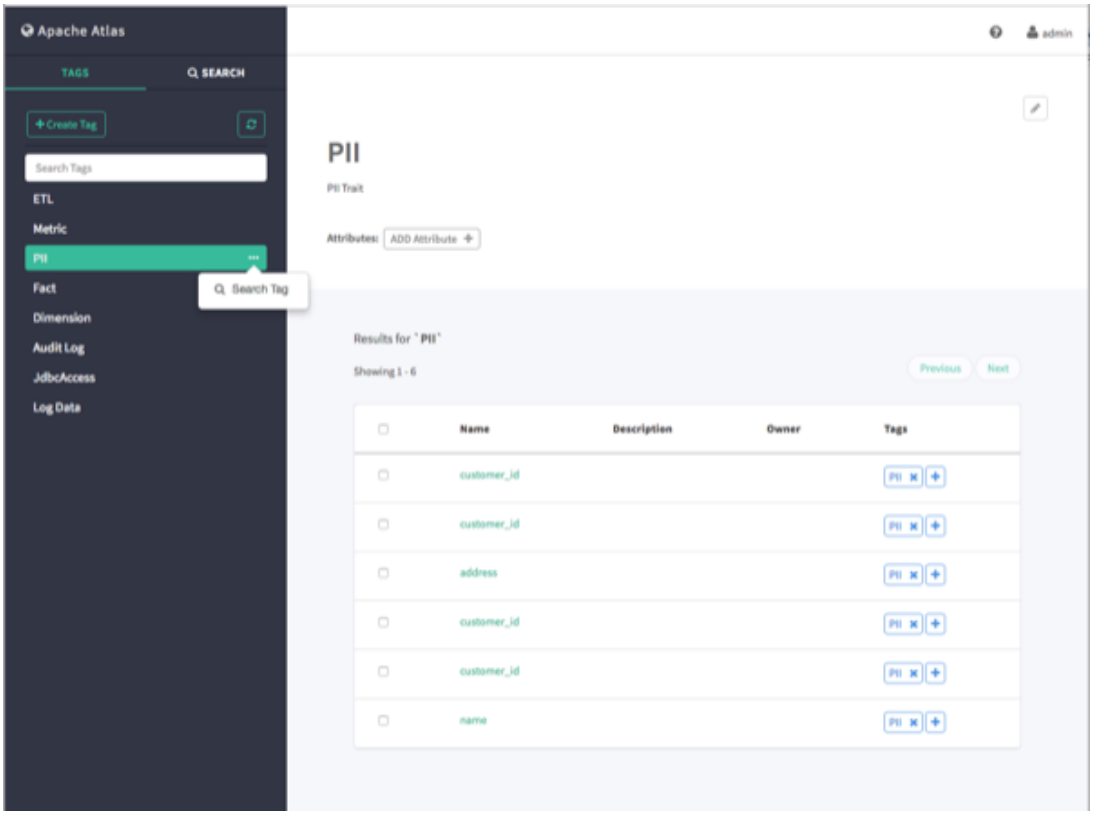
1. To display a list of all of the entities associated with a tag, click the tag name in the Atlas Tags list.



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 entities associated with that tag.



3. You can also search for entities 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 entities 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.

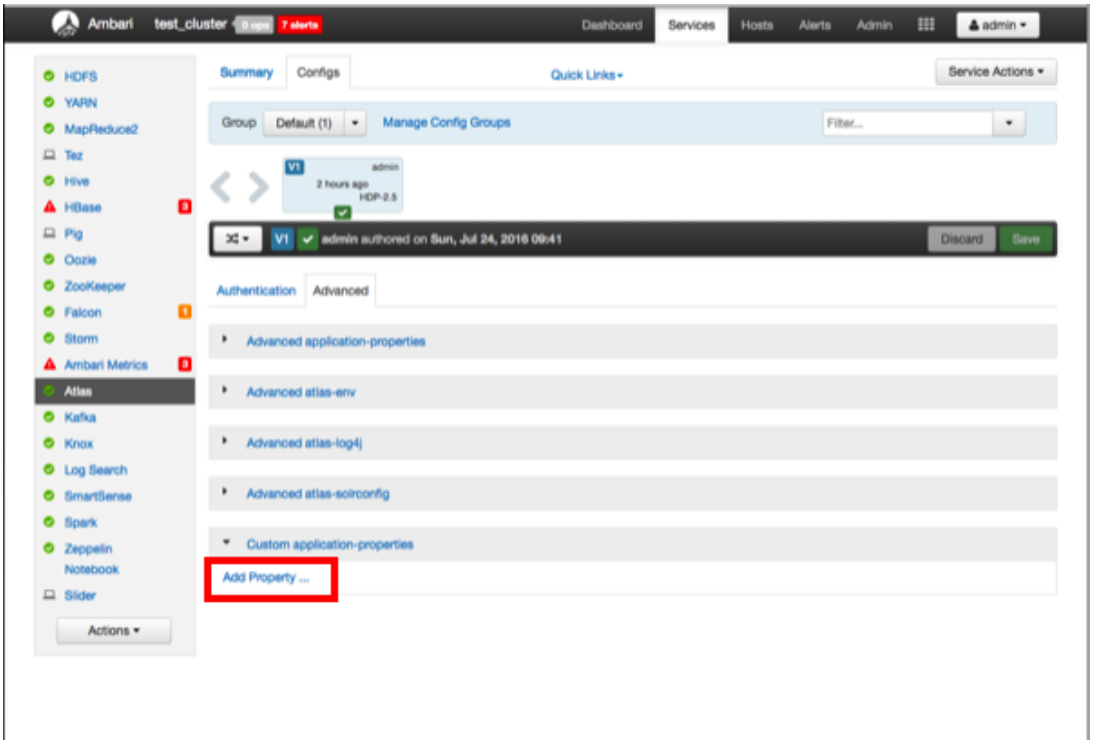
Note

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.

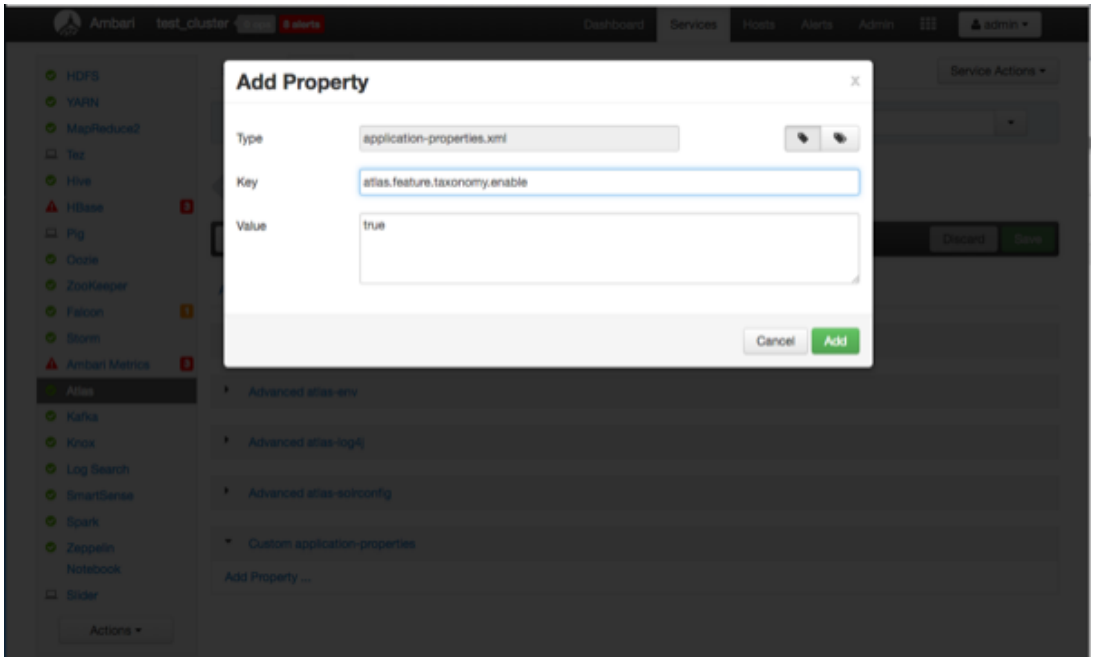
1. Select **Atlas > Configs > Advanced > Custom application-properties**, then click **Add Property**.



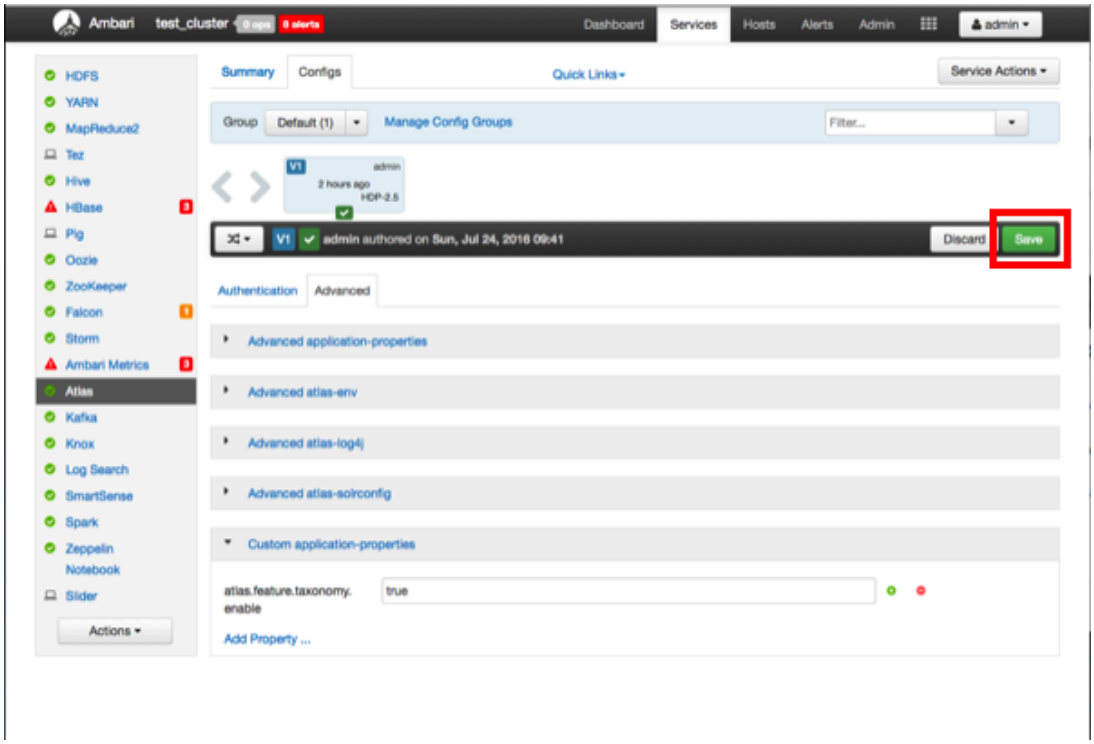
2. On the Add Property pop-up, add the following properties:

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

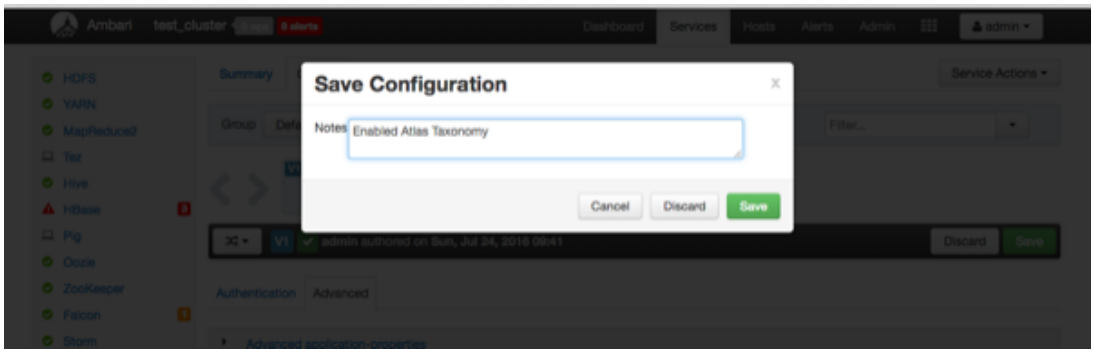
   Click **Add** to add the new property.



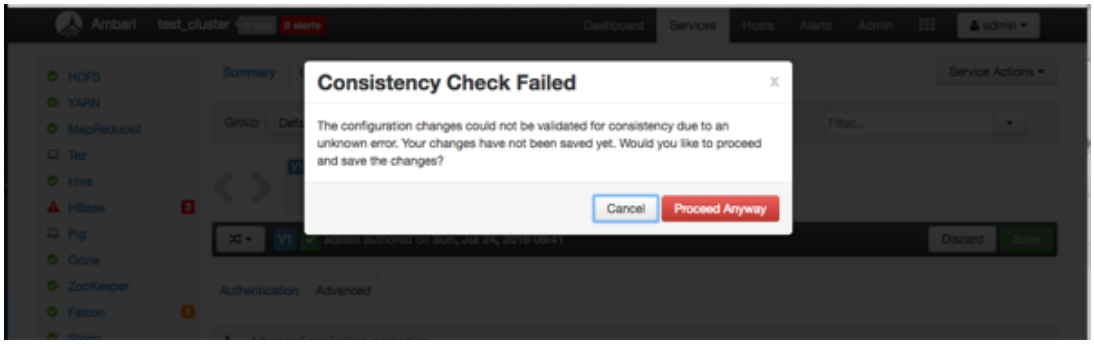
3. The Advanced tab is redisplayed with the new property. Click **Save** to save the new configuration.



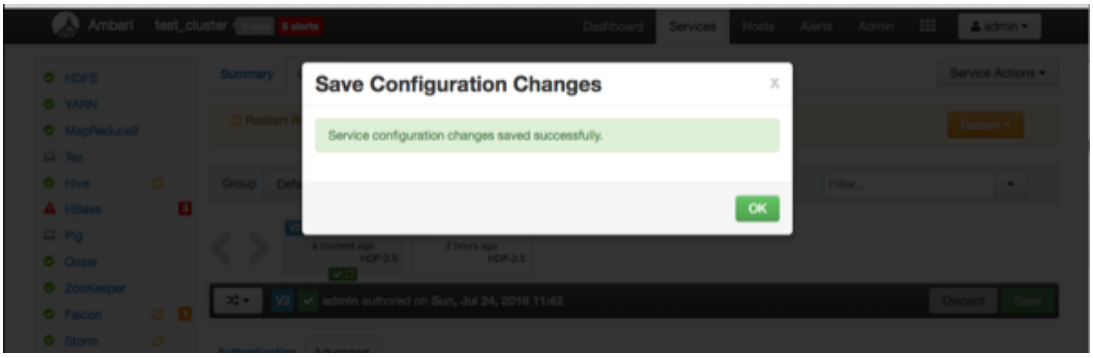
4. A Save Configuration pop-up appears. Type in a note describing the changes you just made, then click **Save**.



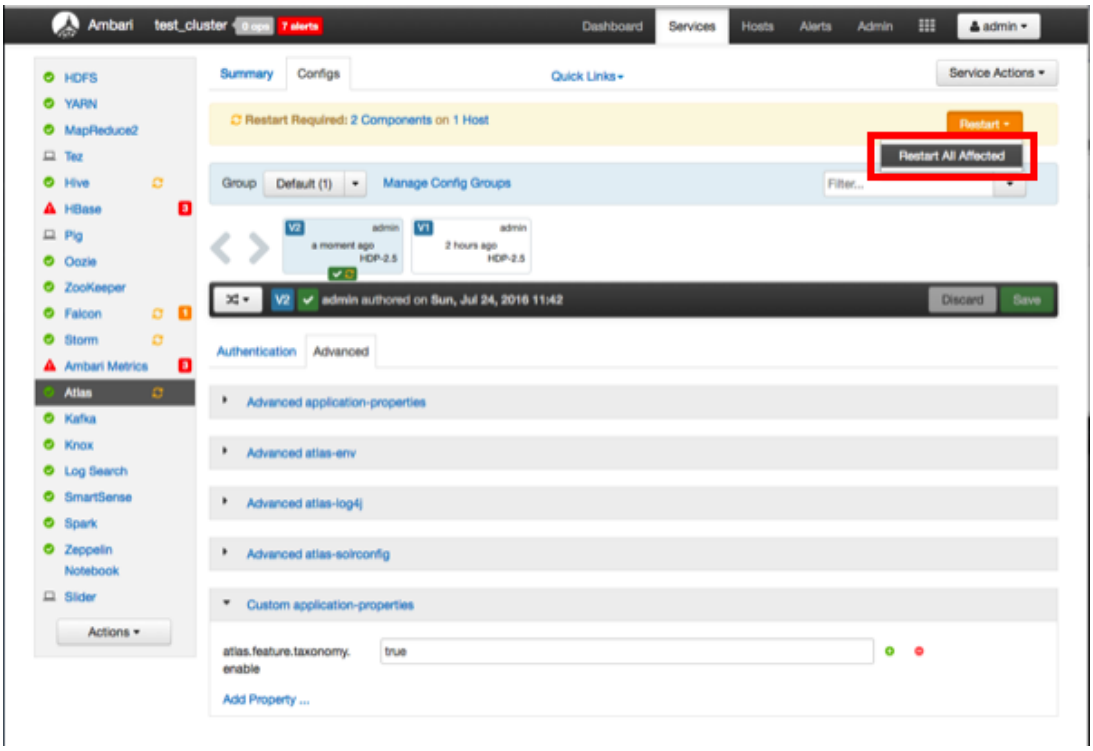
5. If a Consistency Check Failed pop-up appears, click **Proceed Anyway**.



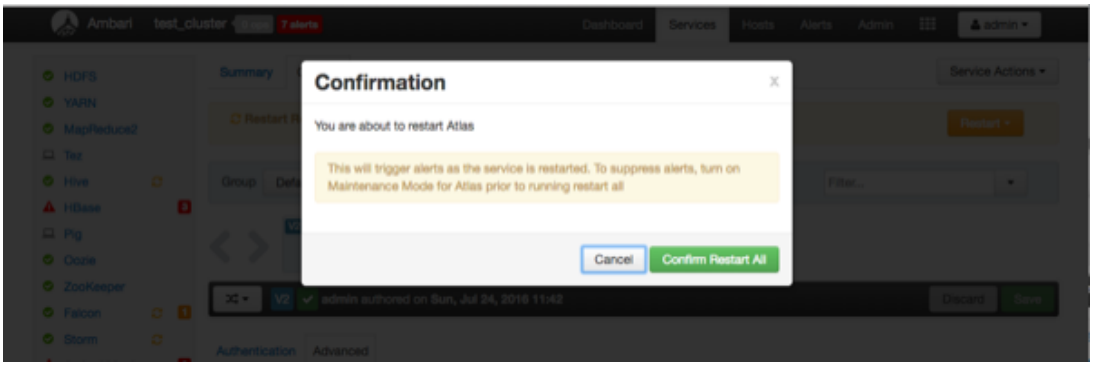
6. Click **OK** on the Save Configuration Changes pop-up.



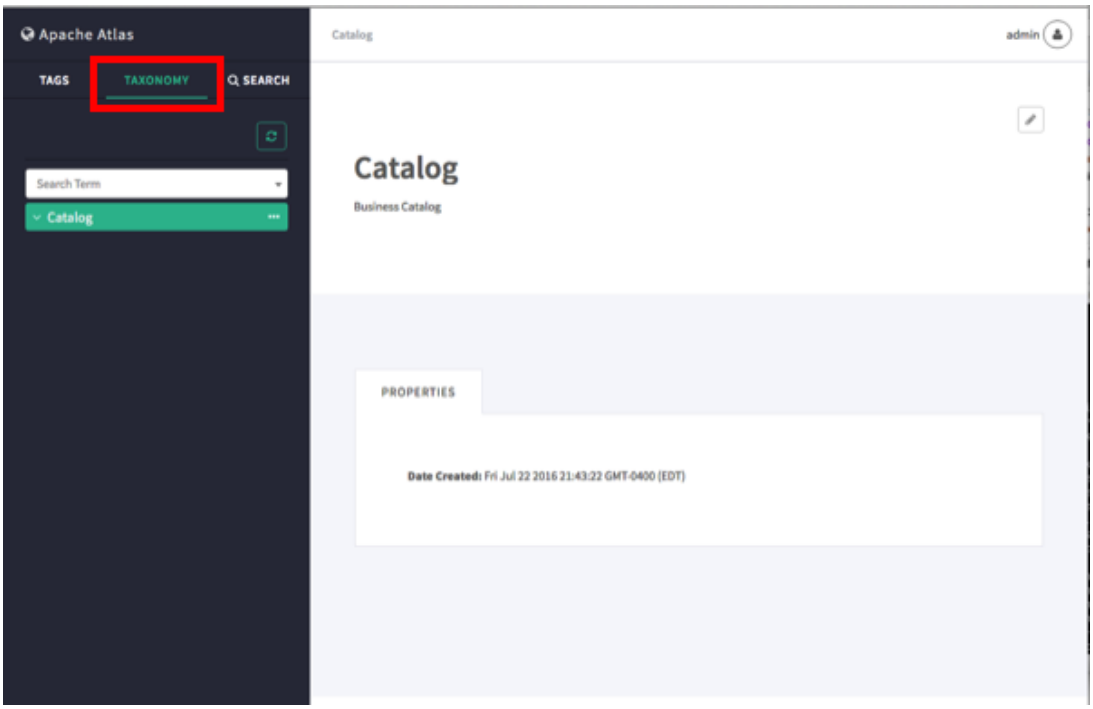
7. Select **Restart > Restart All Affected** to restart the Atlas service and load the new configuration.



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

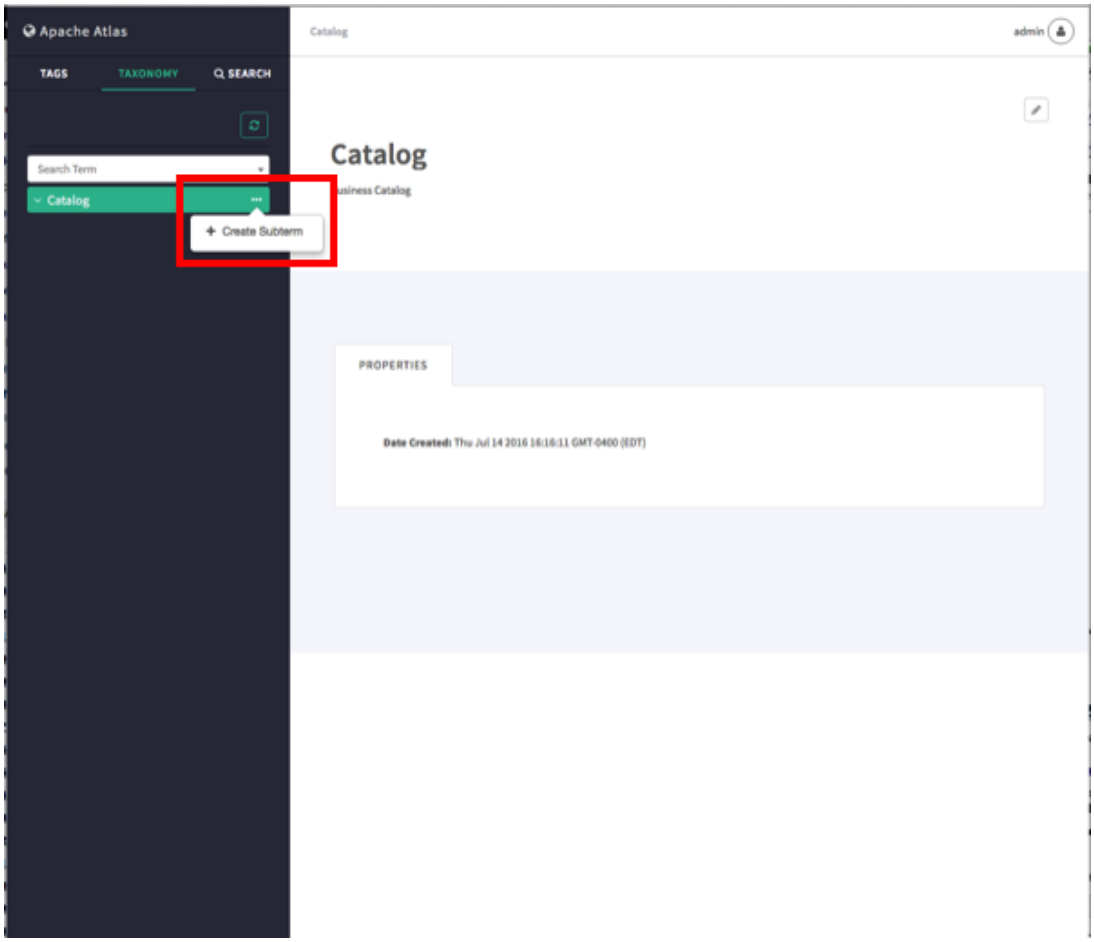


9. After Atlas restarts, the Taxonomy feature is enabled. Other components may also require a restart. To access the Atlas web UI, select **Atlas > Quick Links > Atlas Dashboard**.

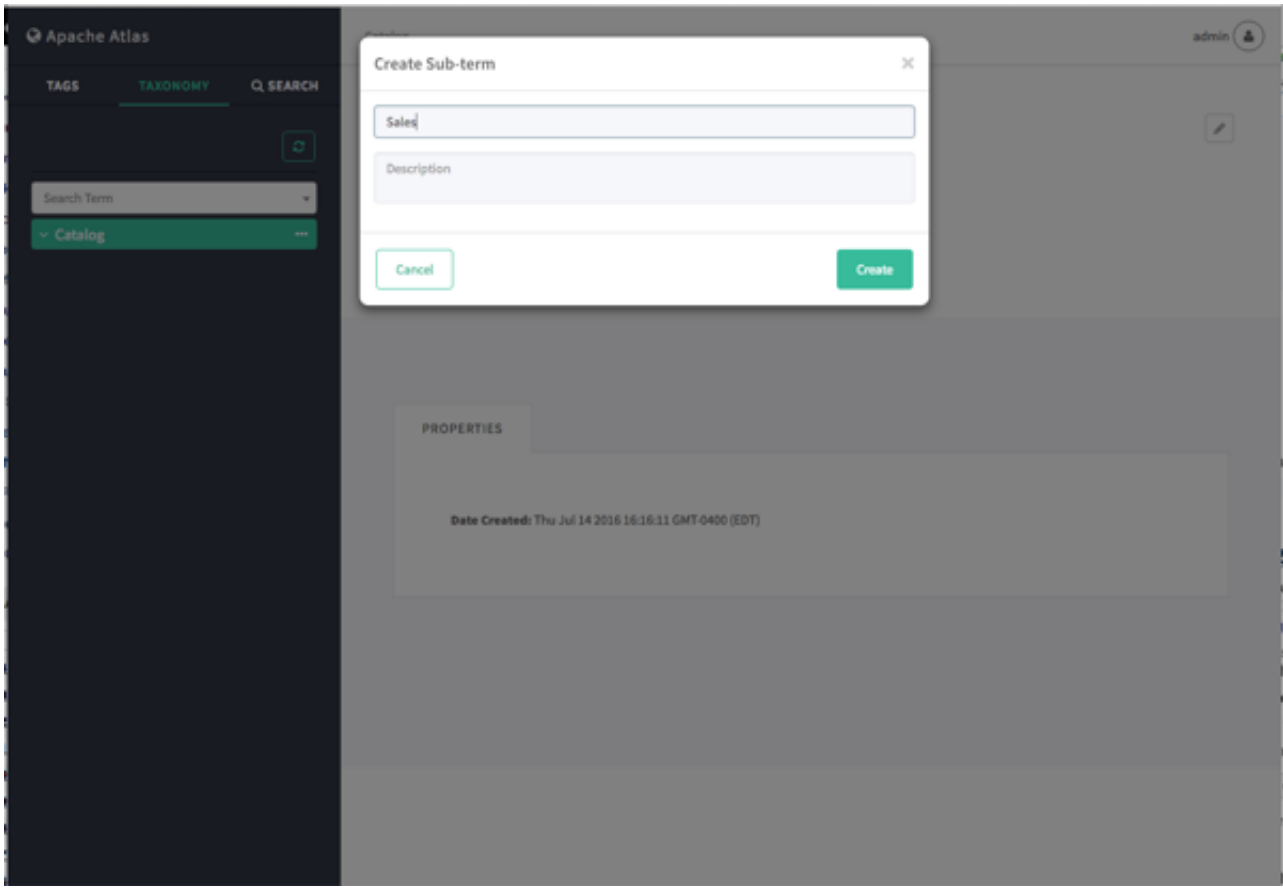


5.2. Creating Taxonomy Terms

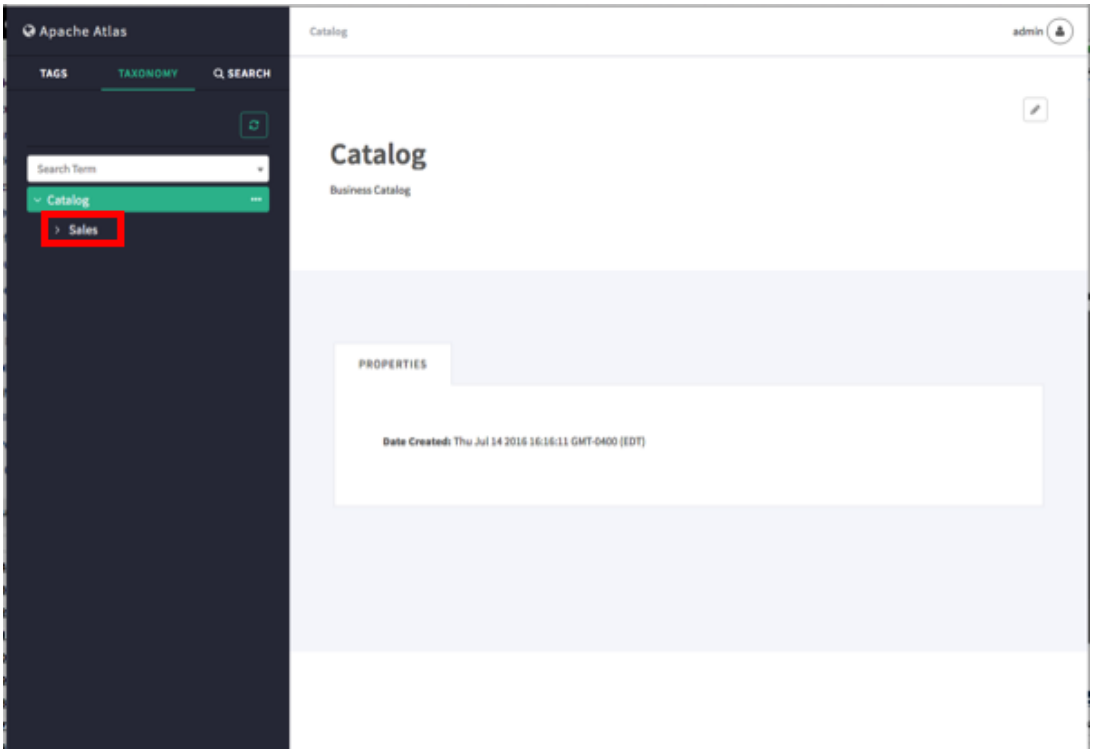
1. On the Atlas web UI, click **Taxonomy**. To create a new sub-term, click the ellipsis symbol at the top level of the Taxonomy, then click **Create Subterm**.



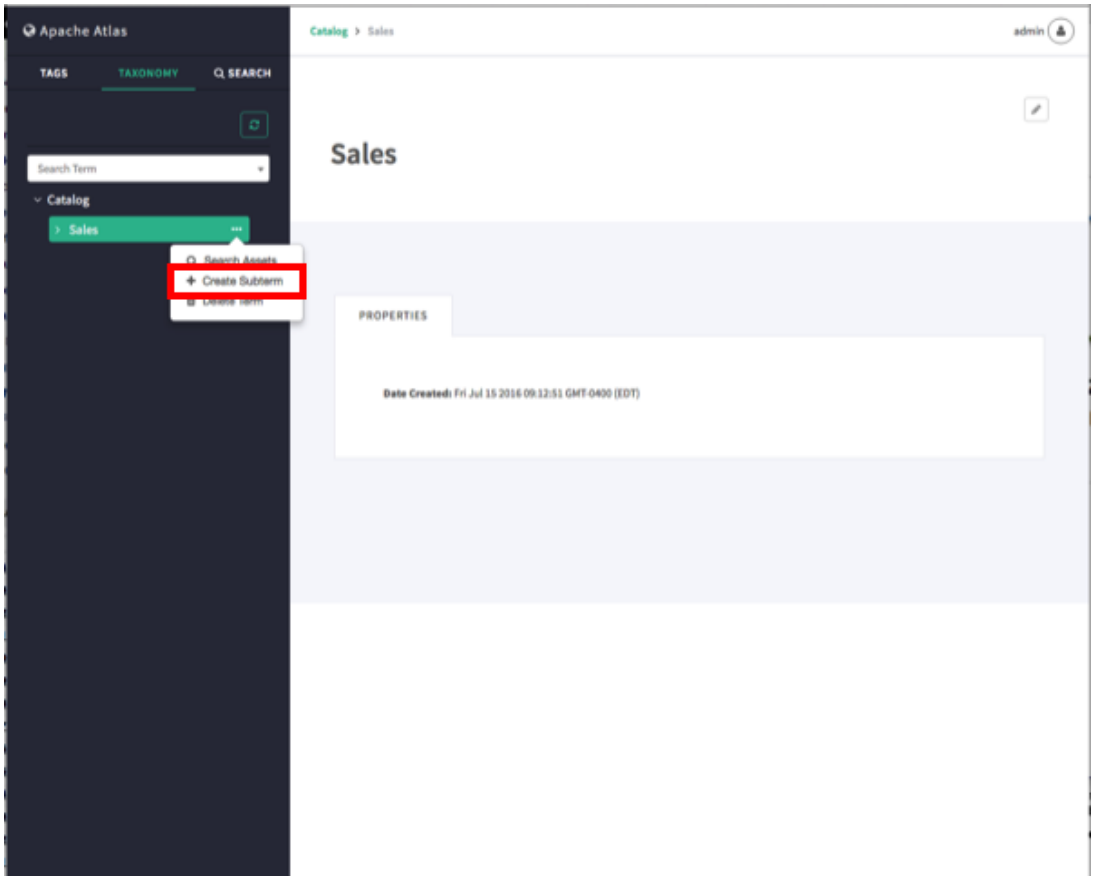
2. On the Create Sub-term pop-up, type in a name and an optional description for the sub-term, then click **Create**.



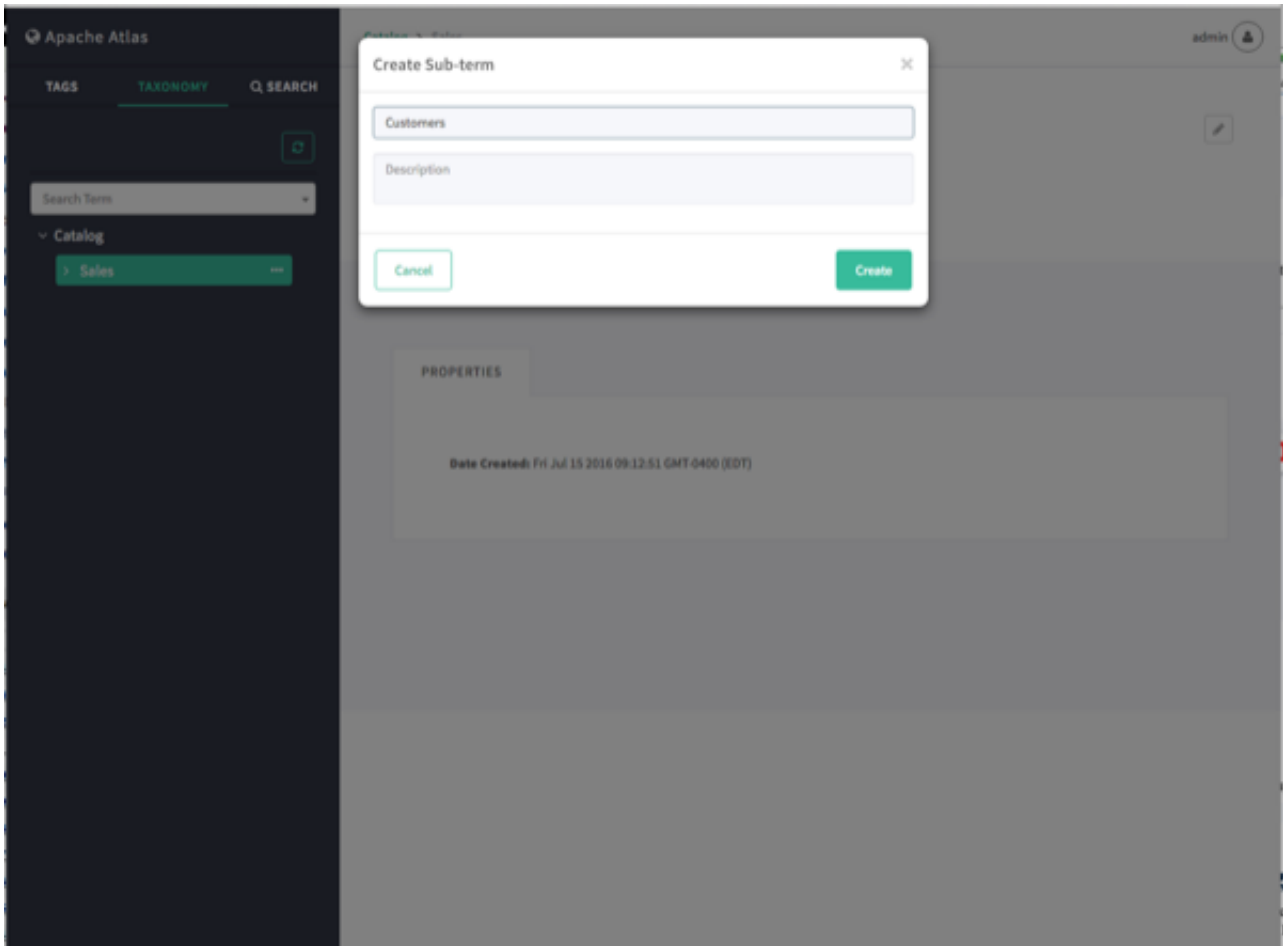
3. The new sub-term appears in the Taxonomy below the top level.



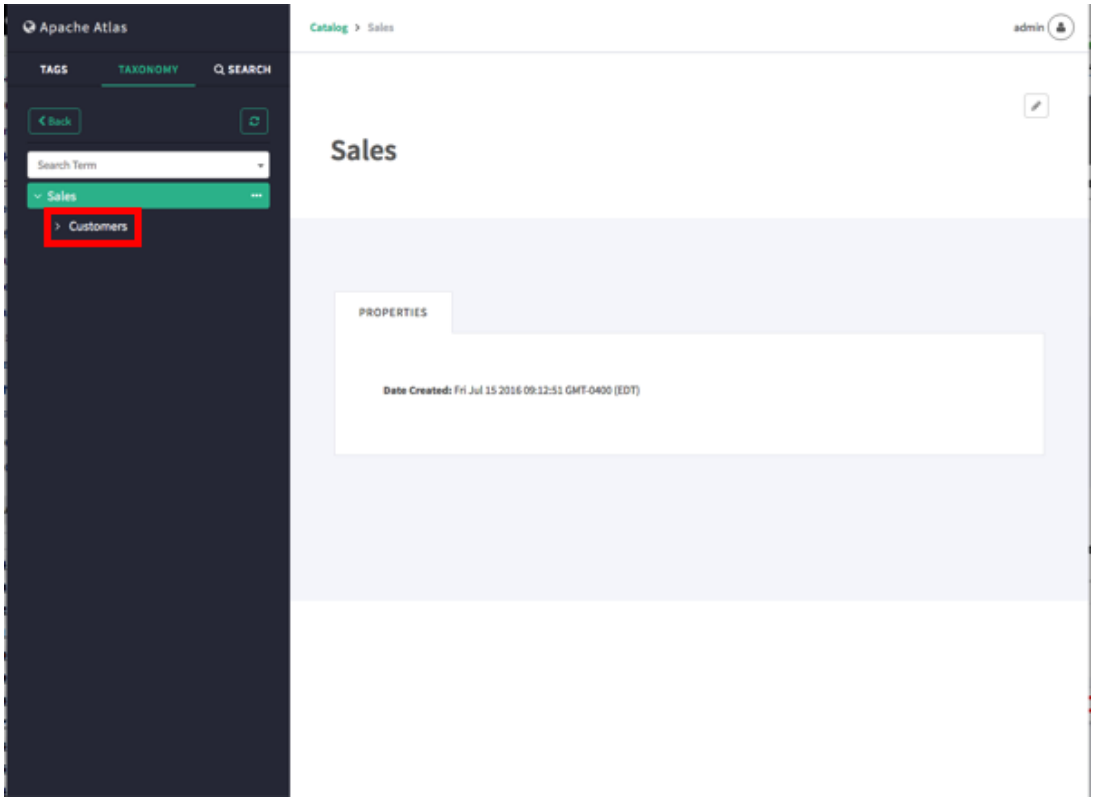
4. To create a new sub-term another level down in the taxonomy hierarchy, select the sub-term, click the ellipsis symbol, then click **Create Subterm**.



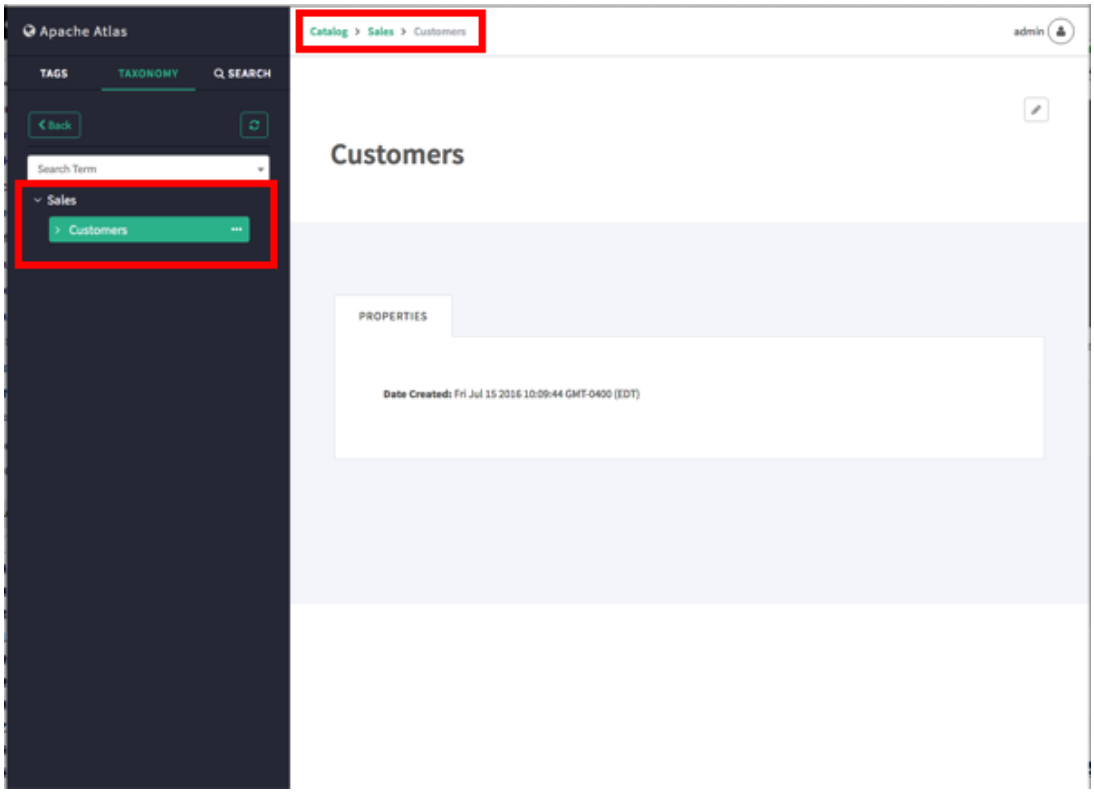
5. On the Create Sub-term pop-up, type in a name and an optional description for the new second-level sub-term, then click **Create**.



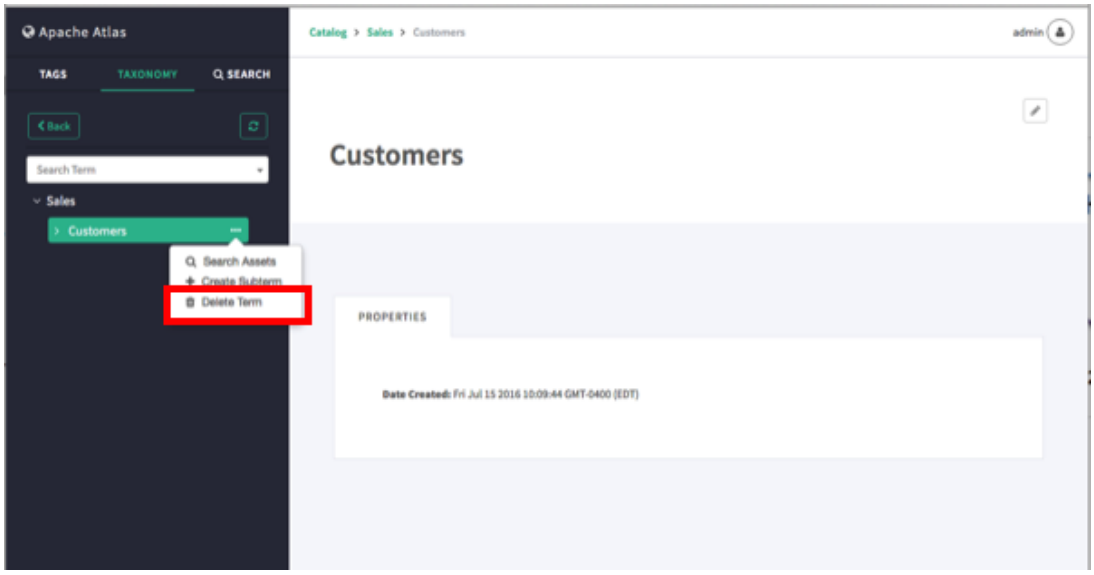
6. The new second-level sub-term appears in the Taxonomy.



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.

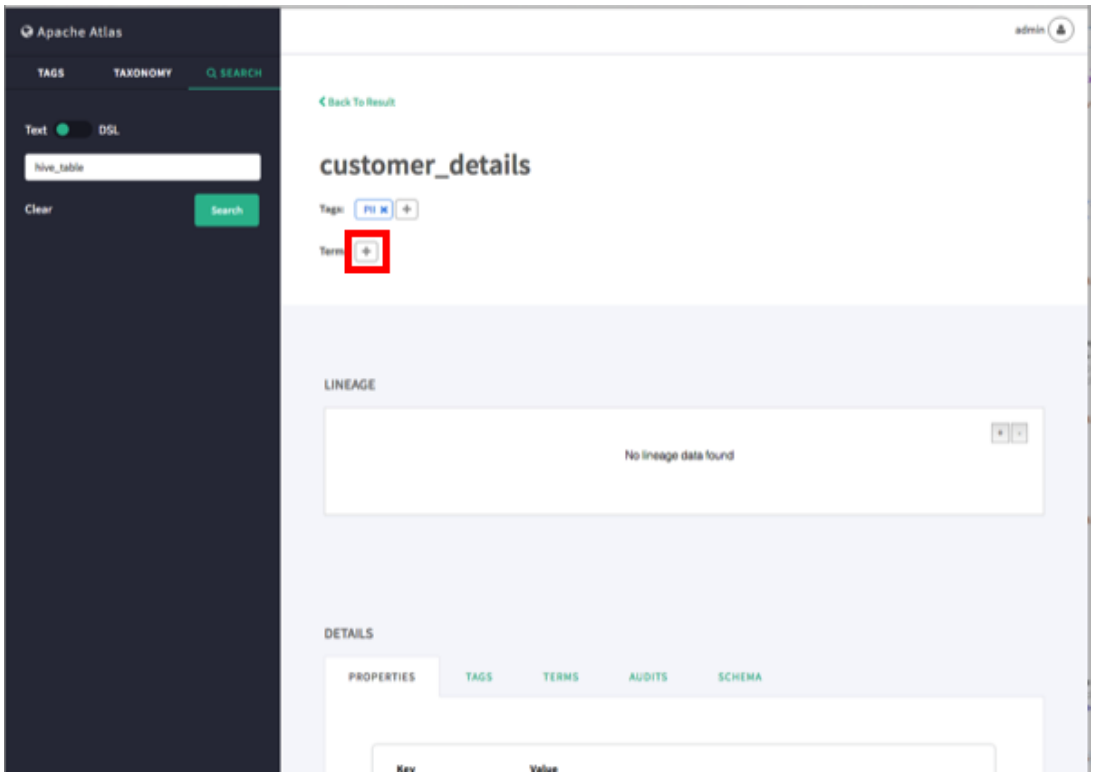


8. To delete a taxonomy term, click the ellipsis symbol for the term, then select **Delete Term**. When you delete a term it is also removed from all entities that are currently associated with the term.

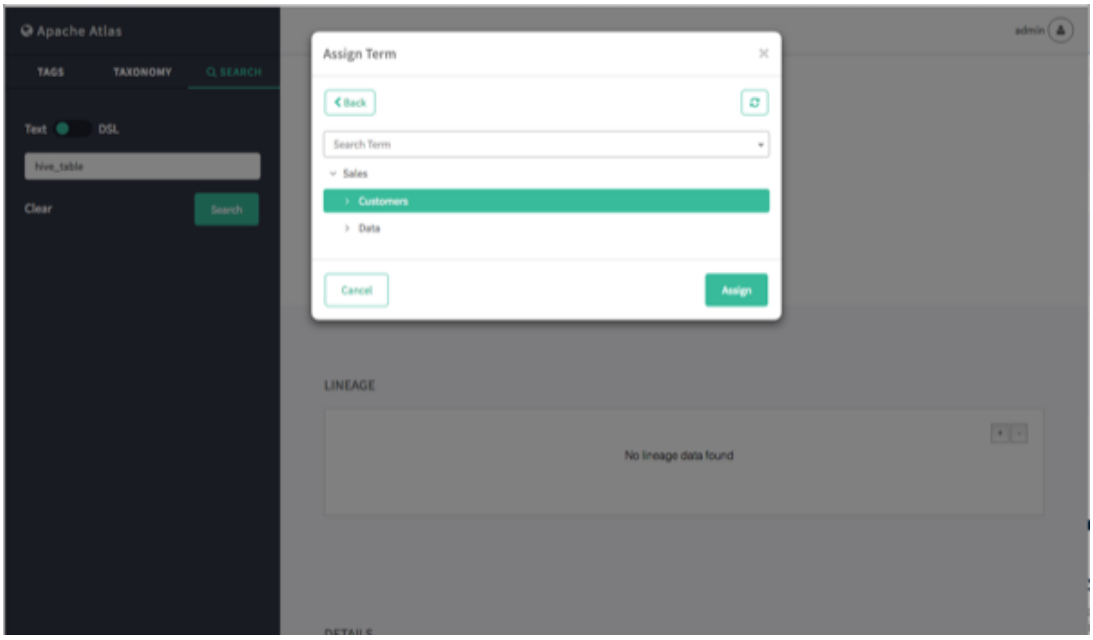


5.3. Associating Taxonomy Terms with Entities

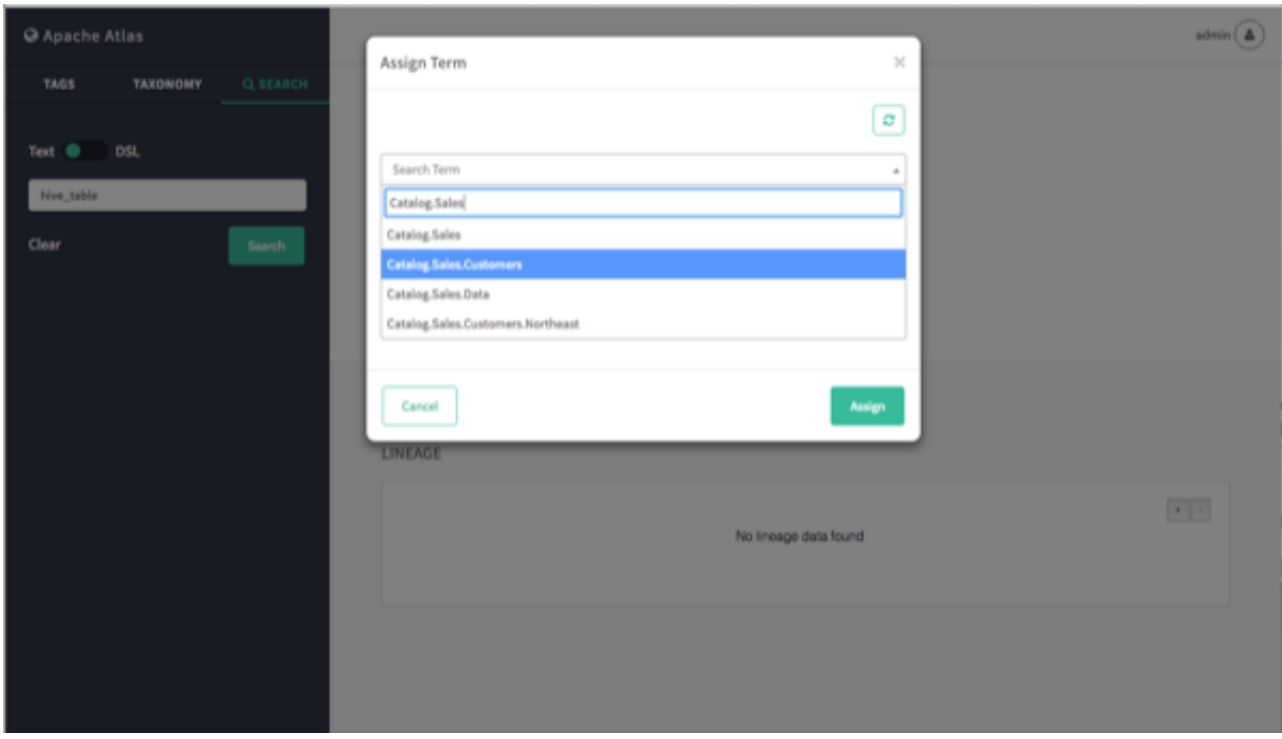
1. Select an entity. In the example below, we searched for all `hive_table` entities, and then selected the "customer_details" table from the list of search results. To associate a taxonomy term with an asset, click the + icon next to the **Terms:** label.



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.

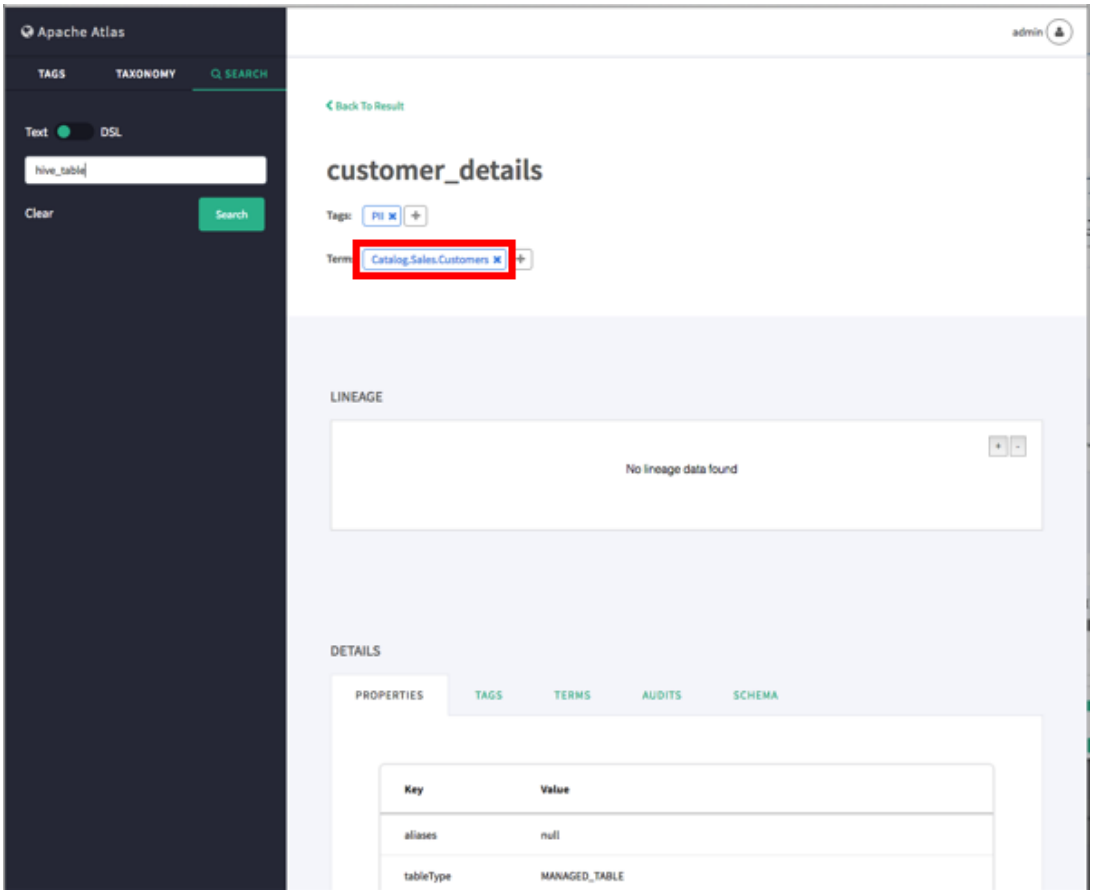


Note

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,

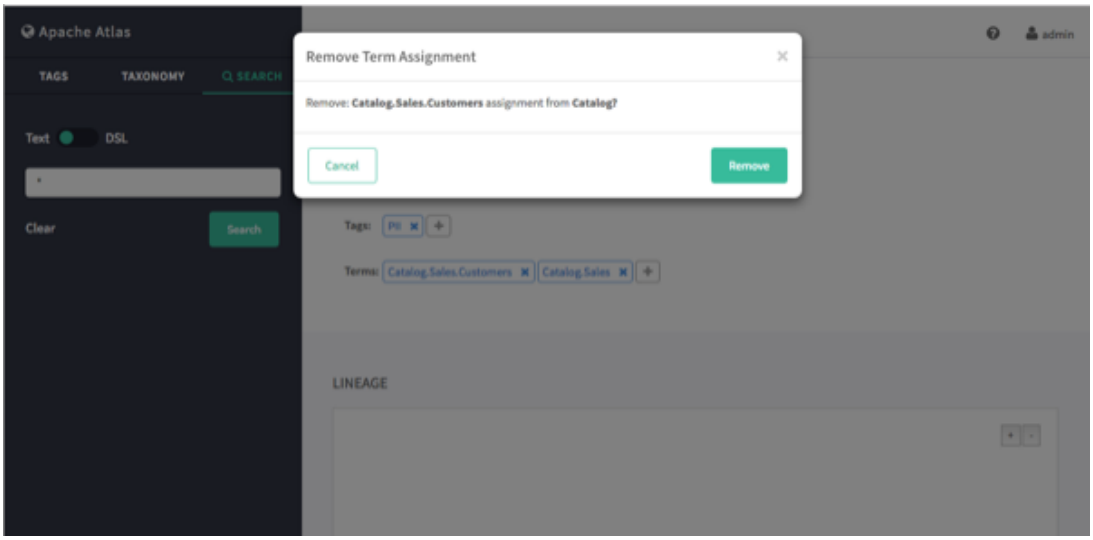
Catalog.Sales.Customers represents the Catalog > Sales > Customers taxonomy level.

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.

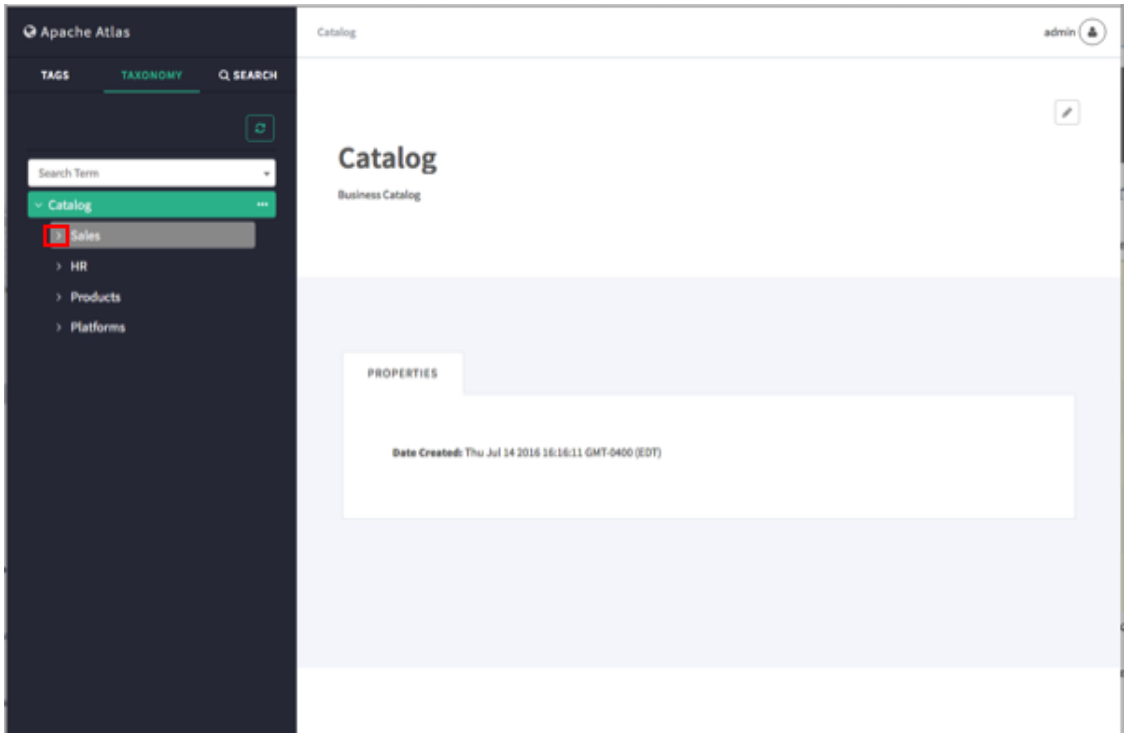


5.4. Navigating the Atlas Taxonomy

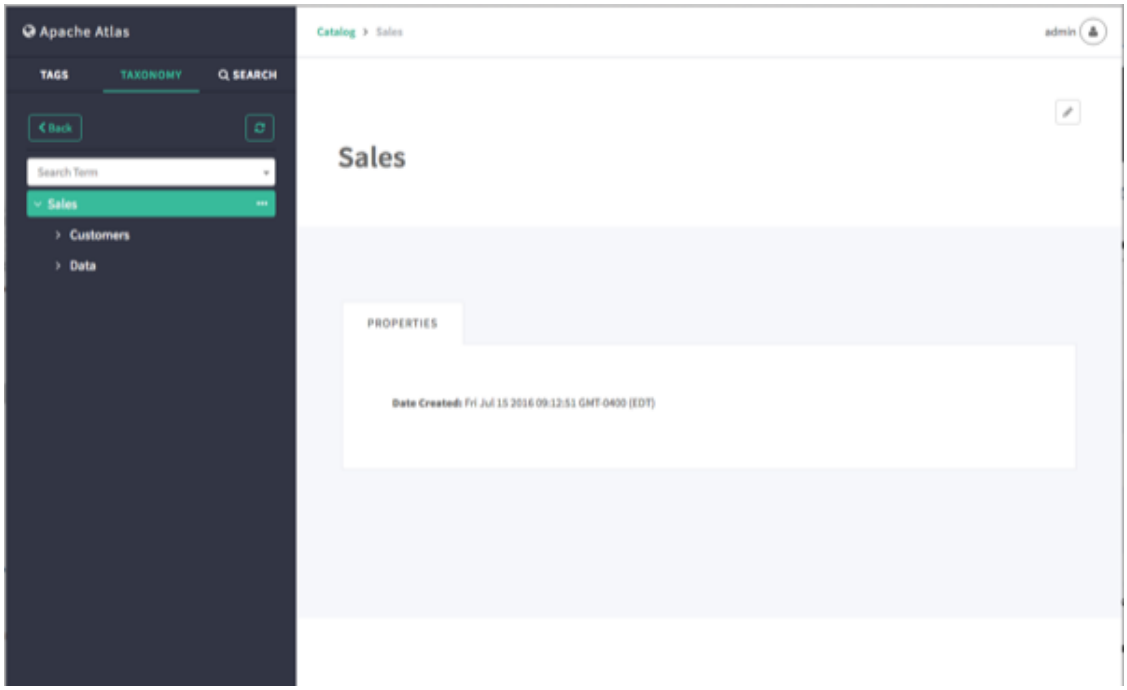
Only two levels at a time are displayed in the Taxonomy list, but you can use the following methods to navigate the Atlas Taxonomy.

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:



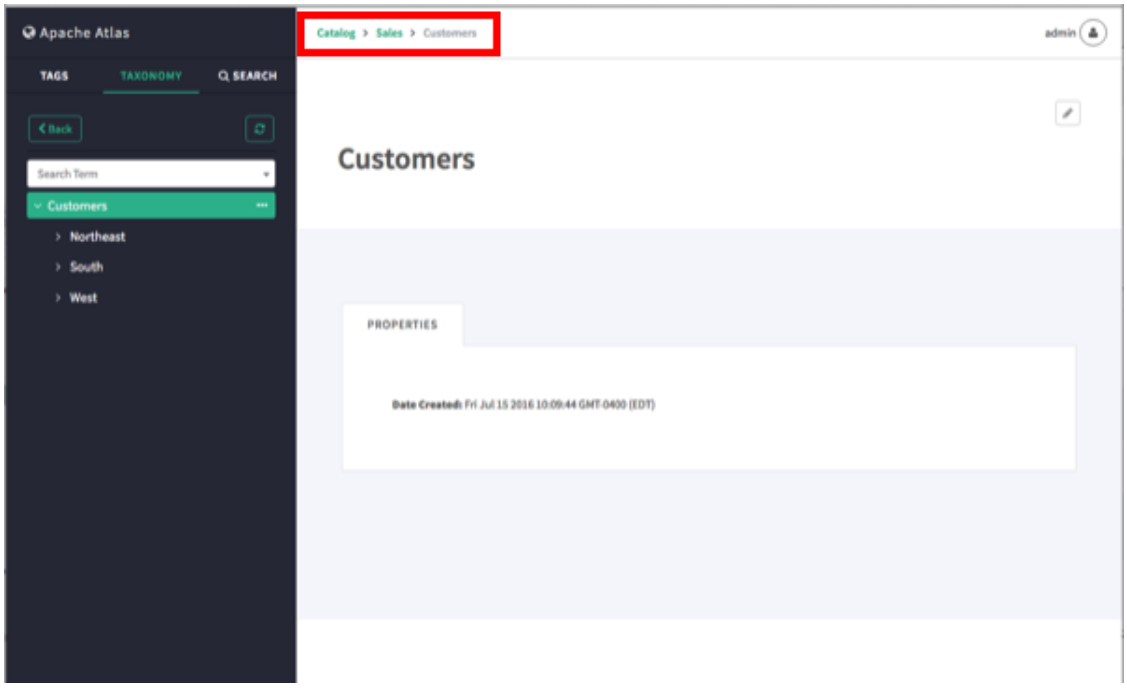
The child terms for Sales (Customers and Data) are displayed:



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

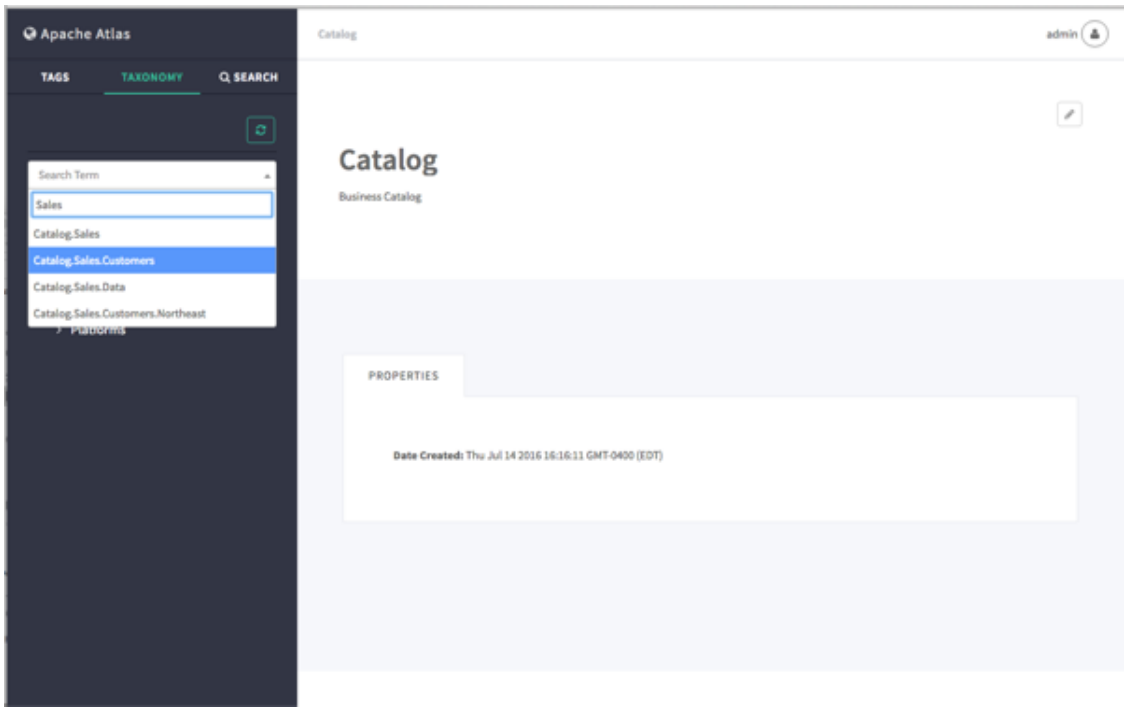
5.4.2. Breadcrumb Trail

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.

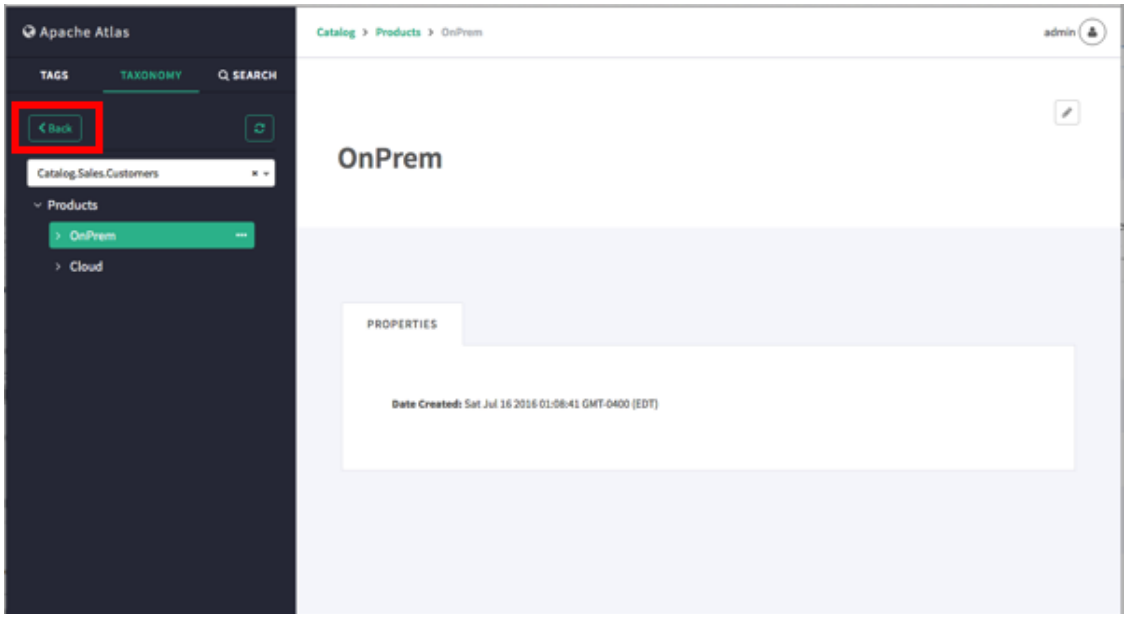


Note

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.

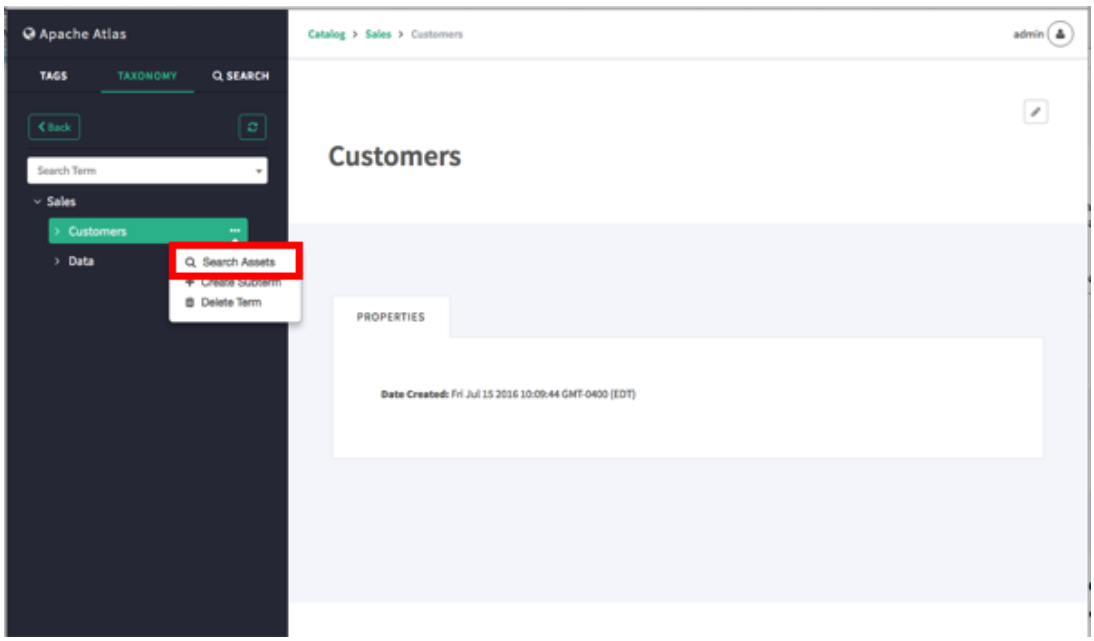
5.4.4. Back Button

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.

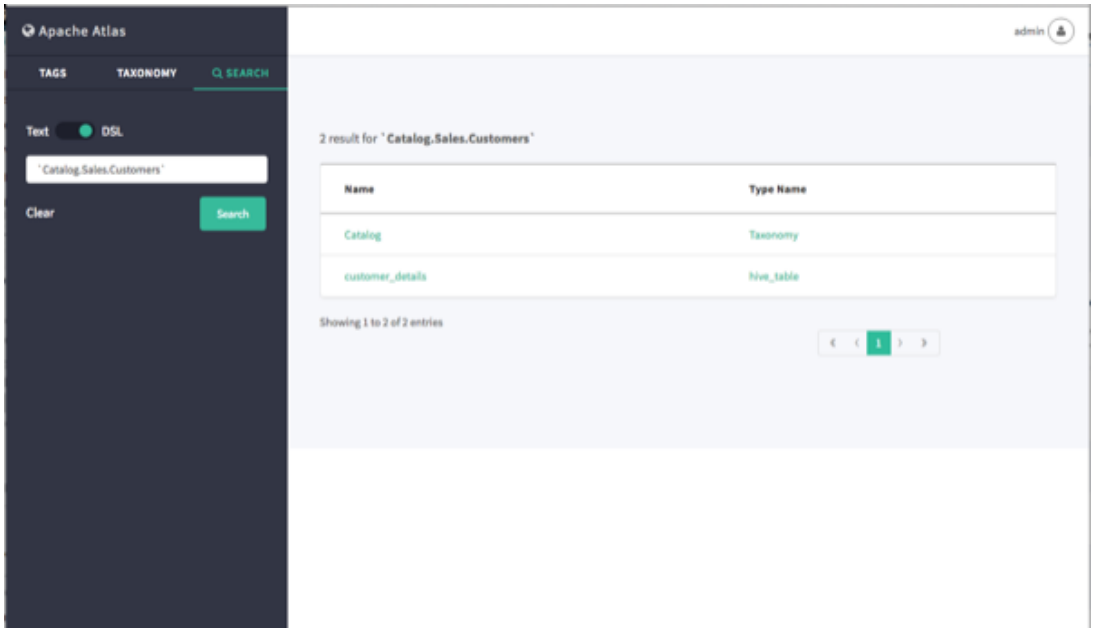


5.5. Searching for Entities Associated with Taxonomy Terms

1. To search for entities 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 entities associated with the term.

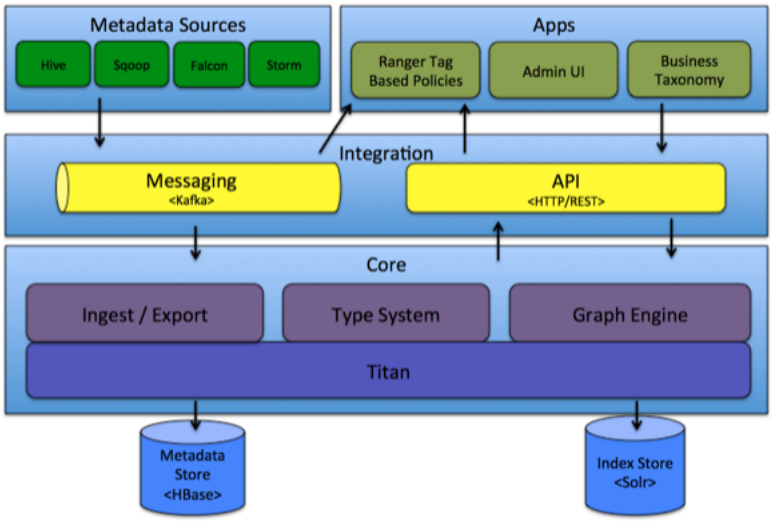

Apache Atlas
admin
TAGS
TAXONOMY
SEARCH
Text
DSL
'Catalog.Sales.Customers'
Clear
Search
2 result for 'Catalog.Sales.Customers'
Name
Type Name
Catalog
Taxonomy
customer_details
hive_table
Showing 1 to 2 of 2 entries

6. Apache Atlas Technical Reference

Apache Atlas provides scalable and extensible governance capabilities for Hadoop. Atlas enables enterprises to effectively and efficiently meet their compliance requirements within Hadoop, and allows integration with the entire enterprise data ecosystem.

6.1. Apache Atlas Architecture

The following image shows the Atlas components.



Atlas components can be grouped under the following categories:

- Core
- Integration
- Metadata Sources
- Applications

6.1.1. 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. This model is composed of "types" definitions. "Entities" are instances of types that represent the actual metadata objects. All metadata objects managed by Atlas (such as Hive tables) are modeled using types, and represented as entities.

One key point to note is that the generic nature of the modeling 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.

6.1.2. 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.

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.

6.1.3. 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 this 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).

6.1.4. Applications

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.

Ranger Tag-based Policies: 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.

Business Taxonomy: The metadata objects ingested into Atlas from metadata sources are primarily a form of technical metadata. To enhance the discoverability and governance capabilities, Atlas includes a Business Taxonomy interface that allows users to define a hierarchical set of business terms that represent their business domain, and then associate these terms with Atlas metadata entities Atlas. The Business Taxonomy is included in the Atlas Admin UI, and integrates with Atlas using the REST API.

6.2. Creating Metadata: The Atlas Type System

Atlas allows you to define a model for metadata objects. This model is composed of "types" definitions. "Entities" are instances of types that represent the actual metadata objects. All metadata objects managed by Atlas (such as Hive tables) are modelled using types, and represented as entities.

6.2.1. Atlas Types

In Atlas, a "type" is a definition of how a particular type of metadata object is stored and accessed. A type represents one or more attributes that define the properties for the metadata object. Users with a development background will recognize the similarity of a type to classes used in object-oriented programming languages, or a table schema in a relational database.

An example of a type that is natively defined within Atlas is a Hive table. A Hive table is defined with the following attributes:

```
Name: hive_table
MetaType: Class
SuperTypes: DataSet
Attributes:
 name: String (name of the table)
 db: Database object of type hive_db
```

```
owner: String
createTime: Date
lastAccessTime: Date
comment: String
retention: int
sd: Storage Description object of type hive_storagedesc
partitionKeys: Array of objects of type hive_column
aliases: Array of strings
columns: Array of objects of type hive_column
parameters: Map of String keys to String values
viewOriginalText: String
viewExpandedText: String
tableType: String
temporary: Boolean
```

This example helps illustrate the following points:

- An Atlas type is identified uniquely by a name.
- A type has a metatype. A metatype represents the type of a model in Atlas. Atlas contains the following metatypes:
  - Basic metatypes – Int, String, Boolean, etc.
  - Enum metatypes
  - Collection metatypes – Array, Map
  - Composite metatypes – Class, Struct, Trait
- A type can "extend" from a parent "superType", and therefore inherits the super type attributes. This allows modellers to define common attributes across a set of related types. This is similar to classes inheriting properties of super classes in object-oriented programming.

  In this example, every hive table extends from a pre-defined "DataSet" super type. More details about pre-defined types will be provided in subsequent sections.

  It is also possible for an Atlas type to extend from multiple super types.
- Types that have a Class, Struct, or Trait metatype can have a collection of attributes. Each attribute has a name along with other associated properties. A property can be referred to using the format `type_name.attribute_name`. You should also note that attributes themselves are defined using Atlas metatypes. The difference between Classes and Structs is explained in the context of Entities in the next section. Traits will be discussed in the “Cataloging Metadata in Atlas” section.

  In this example, `hive_table.name` is a String, `hive_table.aliases` is an array of Strings, `hive_table.db` refers to an instance of a type named `hive_db`, and so on.
- You can use type references in attributes (such as `hive_table.db`) to define arbitrary relationships between two types defined in Atlas, which enables you to build rich models. You can also collect a list of references as an attribute type (for example, `hive_table.cols`, which represents a list of references from `hive_table` to the `hive_column` type).

6.2.2. Atlas Entities

In Atlas, an "entity" is a specific value or instance of a type, and thus represents a specific metadata object. Using the object-oriented programming language analogy, an instance (entity) is an object of a particular class (type).

Note

In the Atlas UI, entities are sometimes referred to as "assets".

An example of an entity is a specific Hive table. Consider a Hive "customers" table in the "default" Hive database. In Atlas this table is an entity of the type `hive_table`. As an instance of a class type, it has values for all of the `hive_table` type attributes. For example:

```
guid: "9ba387dd-fa76-429c-b791-ffc338d3c91f"
typeName: “hive_table”
values:
 name: “customers”
 db: "b42c6cfc-c1e7-42fd-a9e6-890e0adf33bc"
 owner: “admin”
 createTime: "2016-06-20T06:13:28.000Z"
 lastAccessTime: "2016-06-20T06:13:28.000Z"
 comment: null
 retention: 0
 sd: "ff58025f-6854-4195-9f75-3a3058dd8dcf"
 partitionKeys: null
 aliases: null
 columns: ["65e2204f-6a23-4130-934a-9679af6a211f", "d726de70-faca-46fb-9c99-
cf04f6b579a6", ...]
 parameters: {"transient_lastDdlTime": "1466403208"}
 viewOriginalText: null
 viewExpandedText: null
 tableType: “MANAGED_TABLE”
 temporary: false
```

This example helps illustrate the following points:

- Every entity that is an instance of a class type is identified by a unique identifier, referred to as a GUID. This GUID is generated by the Atlas server when the object is defined, and remains constant for the entire lifetime of the entity. Each entity can be accessed by referencing its GUID.

  In this example, the "customers" table in the default database is uniquely identified by the GUID "9ba387dd-fa76-429c-b791-ffc338d3c91f".

- An entity is of a given type, and the name of the type is provided with the entity definition.

  In this example, the "customers" table is a `hive_table`.

- The values of this entity are a map of all of the attribute names and values for attributes that are defined in the `hive_table` type definition.

- Attribute values follow the metatype of the attribute.

- Basic metatypes – Integer, string, or boolean values. For example:
  - `name: “customers”`
  - `temporary: false`
- Collection metatypes – An array or map of values of the contained metatype. For example:

  ```
  parameters: {"transient_lastDdlTime": "1466403208"}
  ```

- Composite metatypes – For classes, the value is an entity with which this particular entity will have a relationship.

  For example, the Hive “customers” table is present in the "default" database. The relationship between the table and database are captured via the `db` attribute. Therefore, the value of the `db` attribute is a GUID that uniquely identifies the `hive_db` "default" entity.

We can now see the difference between Class and Struct metatypes. Classes and Structs both compose attributes of other types. However, entities of Class types have the `guid` attribute, and can be referenced from other entities (such as a `hive_db` entity that is referenced from a `hive_table` entity). Instances of Struct types do not have an identity of their own. The value of a Struct type is a collection of attributes that are "embedded" inside the entity itself.

6.2.3. Atlas Attributes

We have noted that attributes are defined inside composite metatypes such as Class and Struct, and that attributes have a name and a metatype value. However, attributes in Atlas have additional properties that define more concepts related to the type system.

An attribute has the following properties:

```
name: string,
typeName: string,
constraintDefs: list<AtlasContraintDef>,
isIndexable: boolean,
isUnique: boolean,
isOptional : boolean,
cardinality: enum
```

- `name` – The attribute name.
- `typeName` – The metatype name of the attribute (native, collection, or composite).
- `constraintDefs` – This list indicates an aspect of modeling. If we want to impose custom constraints on the attributes of a type, we can specify those constraints using this field. Let’s take the example of type `hive_table`. Hive column is a dependent attribute of a Hive table, and does not have a lifecycle of its own. Therefore we can impose a constraint on a `hive_column` entity that whenever an entity of type `hive_table` is deleted, all entities of type `hive_column` contained in `hive_table` entities should be deleted.

Let's examine the attribute definitions in types `hive_table` and `hive_column`.

The `columns` attribute in type `hive_table`:

```
{
 "name": "columns",
 "typeName": "array<hive_column>",
 "cardinality": "SINGLE",
 "constraintDefs": [
  {
   "type": "mappedFromRef",
   "params": {
    "refAttribute": "table"
   }
  }
 ],
 "isIndexable": false,
 "isOptional": true,
 "isUnique": false
}
```

And the corresponding `table` attribute in type `hive_column`:

```
{
 "name": "table",
 "typeName": "hive_table",
 "cardinality": "SINGLE",
 "constraintDefs": [
  {
   "type": "foreignKey",
   "params": {
    "onDelete": "cascade"
   }
  }
 ],
 "isIndexable": false,
 "isOptional": true,
 "isUnique": false
}
```

`"type" : "foreignKey"` indicates that column entities are tied to a particular `hive_table` entity. We have defined an action `"onDelete": "cascade"` which indicates that if the `hive_table` entity is deleted, all of the `hive_column` entities should be deleted.

- `isIndexable` – This flag indicates whether this property should be indexed, so that look-ups can be performed using the attribute value as a predicate, which improves efficiency.
- `isUnique`
  - This flag is also related to indexing. If an attribute is specified as unique, a special index is created for the attribute in Titan that allows for equality-based look ups.
  - Any attribute with a `true` value for this flag is treated as a primary key to distinguish the entity from other entities. Therefore, care should be taken ensure that this attribute does model a unique property in the real world.

For example, consider the `name` attribute of a `hive_table`. In isolation, a name is not a unique attribute for a `hive_table`, because tables with the same name can exist in multiple databases. Even a pair of (database name, table name) is not unique if Atlas is storing metadata of Hive tables among multiple clusters. Only a cluster location, database name, and table name can be deemed unique in the physical world.

- `isOptional` – Indicates whether a value is optional or required.
- `cardinality` – Indicates whether this attribute is a singleton or could be multi-valued. Possible values are `SINGLE`, `LIST` and `SET`.

With this information, let us expand on the attribute definition of one of the attributes of the Hive table below. Let us look at the "db" attribute, which represents the database to which the Hive table belongs:

```
db:
    "dataTypeName": "hive_db",
    "isIndexable": true,
    “isOptional”: false,
    "isUnique": false,
    “Cardinality” : “SINGLE”,
    "name": "db",
```

Note the `false` value for `isOptional`. A table entity cannot be sent without a db reference.

From this description and examples, you can see that attribute definitions can be used to influence specific modeling behavior (constraints, indexing, etc.) to be enforced by the Atlas system.

6.2.4. Atlas System Types

This section describes the available pre-defined Atlas system types (super types).

- `Referenceable` – This type represents all entities that can be searched for using a unique `qualifiedName` attribute.
- `Asset` – This type contains attributes such as `name`, `description`, and `owner`. The `name` attribute is required (multiplicity = required), but the others are optional.

  The purpose of `Referenceable` and `Asset` is to provide modelers with a way to enforce consistency when defining and querying entities of their own types. Having these fixed set of attributes allows applications and User Interfaces to make convention-based assumptions about what default attributes they can expect from types.
- `Infrastructure` – This type extends `Referenceable` and `Asset`, and typically can be used as a common super type for infrastructure metadata objects such as clusters, hosts, etc.
- `DataSet` – This type extends `Referenceable` and `Asset`. Conceptually, it can be used to represent a type that stores data. In Atlas, Hive tables, Sqoop RDBMS tables, etc., are all types that extend from `DataSet`. Types that extend `DataSet` can be expected to

have a Schema, in the sense that they would have an attribute that defines attributes of that dataset – for example, the `columns` attribute in a `hive_table`. Entities types that extend `DataSet` also participate in data transformation, and this transformation can be captured by Atlas via lineage (or provenance) graphs.

- `Process` – This type extends `Referenceable` and `Asset`. Conceptually, it can be used to represent any data transformation operation. For example, an ETL process that transforms a Hive table with raw data to another Hive table that stores some aggregate can be a specific type that extends the `Process` type. A `Process` type has two specific attributes: inputs and outputs. Both inputs and outputs are arrays of `DataSet` entities. Thus an instance of a `Process` type can use these inputs and outputs to capture how the lineage of a DataSet evolves.

6.2.5. Atlas Types API

Summary:

- Base resource name: `v2/types`
- Full URL: `http://<atlas-server-host:port>/api/atlas/v2/types`

6.2.5.1. Enum Type API

- Base resource name: `v2/types/enumdef`
- Full URL: `http://<atlas-server-host:port>/api/atlas/v2/types/enumdef`

6.2.5.1.1. Register Enum Type

Request:

```
POST /v2/types/enumdef
```

Description:

This request is used to register one particular Enum type with the Atlas type system.

Method Signature:

```
@POST
@Path("/enumdef")
@Consumes(Servlets.JSON_MEDIA_TYPE)
@Produces(Servlets.JSON_MEDIA_TYPE)
public AtlasEnumDef createEnumDef(AtlasEnumDef enumDef) throws
 AtlasBaseException {
```

Example Request:

```
POST /v2/types/enumdef
```

Example Request Body:

```
{
 "name" : "creation_order",
 "typeVersion" : "1.1",
 "elementDefs" : [
  {
   "ordinal" : 1,
   "value" : "PRE"
  },
  {
   "ordinal" : 2,
   "value" : "POST"
  }
  ]
}
```

Example Response:

```
{
  "category": "ENUM",
  "guid": "48bb2f42-745c-4b1b-9491-248326dbe997",
  "createTime": 1481872144316,
  "updateTime": 1481872144316,
  "version": 1,
  "name": "creation_order",
  "description": "creation_order",
  "typeVersion": "1.1",
  "elementDefs": [
    {
      "value": "PRE",
      "ordinal": 1
    },
    {
      "value": "POST",
      "ordinal": 2
    }
  ]
}
```

6.2.5.1.2. Update Enum Type Using Name

Request:

```
PUT /v2/types/enumdef/name/{name}
```

Description:

This request is used to update an Enum type that is already registered with the Atlas type system. {name} refers to the original name of the Enum type with which it was registered.

Method Signature:

```
@PUT
@Path("/enumdef/name/{name}")
@Consumes(Servlets.JSON_MEDIA_TYPE)
@Produces(Servlets.JSON_MEDIA_TYPE)
public AtlasEnumDef updateEnumDefByName(@PathParam("name") String name,
 AtlasEnumDef enumDef) throws AtlasBaseException {
```

Example Request:

```
PUT /v2/types/enumdef/name
```

Example Request Body:

```
{
 "name" : "creation_order",
 "typeVersion" : "1.1",
 "elementDefs" : [
  {
   "ordinal" : 1,
   "value" : "PRE"
  },
  {
   "ordinal" : 2,
   "value" : "POST"
  },
  {
   "ordinal" : 3,
   "value" : "UNKNOWN"
  }
  ]
}
```

Example Response:

```
{
  "category": "ENUM",
  "guid": "48bb2f42-745c-4b1b-9491-248326dbe997",
  "createTime": 1481872144316,
  "updateTime": 1481872144316,
  "version": 1,
  "name": "creation_order",
  "description": "creation_order",
  "typeVersion": "1.1",
  "elementDefs": [
    {
      "value": "PRE",
      "ordinal": 1
    },
    {
      "value": "POST",
      "ordinal": 2
    },
    {
      "value": "UNKNOWN",
      "ordinal": 3
    }
  ]
}
```

6.2.5.1.3. Update Enum Type Using GUID

Request:

```
PUT v2/types/enumdef/guid/{guid}
```

Description:

This request is used to update an Enum type that is already registered with the Atlas type system by referencing its GUID.

Method Signature:

```
@PUT
@Path("/enumdef/guid/{guid}")
@Consumes(Servlets.JSON_MEDIA_TYPE)
@Produces(Servlets.JSON_MEDIA_TYPE)
public AtlasEnumDef updateEnumDefByGuid(@PathParam("guid") String guid,
 AtlasEnumDef enumDef) throws AtlasBaseException {
```

Example Request:

```
PUT v2/types/enumdef/guid/48bb2f42-745c-4b1b-9491-248326dbe997
```

Example Request Body:

```
{
 "name" : "creation_order",
 "typeVersion" : "1.1",
 "elementDefs" : [
  {
   "ordinal" : 1,
   "value" : "PRE"
  },
  {
   "ordinal" : 2,
   "value" : "POST"
  },
  {
   "ordinal" : 3,
   "value" : "UNKNOWN"
  }
  ]
}
```

Example Response:

```
{
  "category": "ENUM",
  "guid": "48bb2f42-745c-4b1b-9491-248326dbe997",
  "createTime": 1481872144316,
  "updateTime": 1481872144316,
  "version": 1,
  "name": "creation_order",
  "description": "creation_order",
  "typeVersion": "1.1",
  "elementDefs": [
    {
      "value": "PRE",
      "ordinal": 1
    },
    {
      "value": "POST",
      "ordinal": 2
    },
    {
      "value": "UNKNOWN",
      "ordinal": 3
    }
  ]
}
```

6.2.5.1.4. Get Enum Type Definition Using Name

Request:

```
GET /v2/types/enumdef/name/{name}
```

Description:

Returns the definition of the Enum type associated with the given name.

Method Signature:

```
@GET
@Path("/enumdef/name/{name}")
@Produces(Servlets.JSON_MEDIA_TYPE)
public AtlasEnumDef getEnumDefByName(@PathParam("name") String name) throws
 AtlasBaseException {
```

Example Request:

```
GET v2/types/enumdef/name/creation_order
```

Example Response:

```
{
  "category": "ENUM",
  "name": "creation_order",
  "typeVersion": "1.1",
  "elementDefs": [
    {
      "value": "PRE",
      "ordinal": 1
    },
    {
      "value": "POST",
      "ordinal": 2
    },
    {
      "value": "UNKNOWN",
      "ordinal": 3
    }
  ]
}
```

6.2.5.1.5. Get Enum Type Definition Using GUID

Request:

```
GET v2/types/enumdef/guid/{guid}
```

Description:

Returns the definition of the Enum type by referencing its GUID.

Method Signature:

```
@GET
@Path("/enumdef/guid/{guid}")
@Produces(Servlets.JSON_MEDIA_TYPE)
public AtlasEnumDef getEnumDefByGuid(@PathParam("guid") String guid) throws
 AtlasBaseException {
```

Example Request:

```
GET v2/types/enumdef/guid/d0c902bf-3872-4192-9208-1e9f21e641dc
```

Example Response:

```
{
  "category": "ENUM",
  "name": "creation_order",
  "typeVersion": "1.1",
  "elementDefs": [
    {
      "value": "PRE",
      "ordinal": 1
    },
    {
      "value": "POST",
      "ordinal": 2
    },
    {
      "value": "UNKNOWN",
      "ordinal": 3
    }
  ]
}
```

6.2.5.1.6. Delete Enum Type Definition Using Name

Request:

```
DELETE /v2/types/enumdef/name/{name}
```

Description:

Deletes an Enum type from the Atlas repository by referencing its name.

Method Signature:

```
@DELETE
@Path("/enumdef/name/{name}")
@Produces(Servlets.JSON_MEDIA_TYPE)
public void deleteEnumDefByName(@PathParam("name") String name) throws
 AtlasBaseException {
```

Example Request:

```
DELETE /v2/types/enumdef/name/creation_order
```

6.2.5.1.7. Delete Enum Type Definition Using GUID

Request:

```
DELETE /v2/types/enumdef/guid/{guid}
```

Description:

Deletes an Enum type from the Atlas repository by referencing its GUID.

Method Signature:

```
@DELETE
@Path("/enumdef/guid/{guid}")
@Produces(Servlets.JSON_MEDIA_TYPE)
public void deleteEnumDefByGuid(@PathParam("guid") String guid) throws
 AtlasBaseException {
```

Example Request:

```
DELETE v2/types/enumdef/guid/d0c902bf-3872-4192-9208-1e9f21e641dc
```

6.2.5.1.8. Get All Enum Type Definitions

Request:

```
GET /v2/types/enumdef
```

Description:

Returns all Enum type definitions.

Method Signature:

```
@GET
@Path("/enumdef")
@Produces(Servlets.JSON_MEDIA_TYPE)
public AtlasEnumDefs searchEnumDefs() throws AtlasBaseException {
```

Example Request:

```
GET /v2/types/enumdef
```

Example Response:

```
{
  "list": [
    {
      "category": "ENUM",
      "guid": "b3fd4d06-2a10-4722-9419-8210b00fddd0",
      "createTime": 1480496684400,
      "updateTime": 1480496684400,
      "version": 1,
      "name": "hive_principal_type",
      "description": "hive_principal_type",
      "typeVersion": "1.0",
      "elementDefs": [
        {
          "value": "USER",
          "ordinal": 1
        },
        {
          "value": "ROLE",
          "ordinal": 2
        },
        {
          "value": "GROUP",
          "ordinal": 3
        }
      ]
    },
    {
      "category": "ENUM",
```

```
      "guid": "2ba8a2e0-63e6-4d5c-bc26-140fa95eb241",
      "createTime": 1481884990372,
      "updateTime": 1481884990372,
      "version": 1,
      "name": "creation_order",
      "description": "creation_order",
      "typeVersion": "1.1",
      "elementDefs": [
        {
          "value": "PRE",
          "ordinal": 1
        },
        {
          "value": "POST",
          "ordinal": 2
        },
        {
          "value": "UNKNOWN",
          "ordinal": 3
        }
      ]
    },
    {
      "category": "ENUM",
      "guid": "011fe1e9-78a0-41bb-b7cf-6091a3e40424",
      "createTime": 1480496681754,
      "updateTime": 1480496681754,
      "version": 1,
      "name": "file_action",
      "description": "file_action",
      "typeVersion": "1.0",
      "elementDefs": [
        {
          "value": "NONE",
          "ordinal": 0
        },
        {
          "value": "EXECUTE",
          "ordinal": 1
        },
        {
          "value": "WRITE",
          "ordinal": 2
        },
        {
          "value": "WRITE_EXECUTE",
          "ordinal": 3
        },
        {
          "value": "READ",
          "ordinal": 4
        },
        {
          "value": "READ_EXECUTE",
          "ordinal": 5
        },
        {
          "value": "READ_WRITE",
          "ordinal": 6
        },
```

```
        {
          "value": "ALL",
          "ordinal": 7
        }
      ]
    }
  ],
  "startIndex": 0,
  "pageSize": 3,
  "totalCount": 3,
  "sortType": "NONE"
}
```

6.2.5.2. Struct Type API

- Base resource name: `v2/types/structdef`
- Full URL: `http://<atlas-server-host:port>/api/atlas/v2/types/structdef`

6.2.5.2.1. Register Struct Type

Request:

```
POST /v2/types/structdef
```

Description:

This request is used to register a single Struct type with the Atlas type system.

Method Signature:

```
@POST
@Path("/structdef")
@Consumes(Servlets.JSON_MEDIA_TYPE)
@Produces(Servlets.JSON_MEDIA_TYPE)
public AtlasStructDef createStructDef(AtlasStructDef structDef) throws
 AtlasBaseException {
```

Example Request:

```
POST /v2/types/structdef
```

Example Request Body:

```
{
    "name": "table_creation_order",
    "typeVersion": "1.0",
    "attributeDefs": [
        {
            "name": "order",
            "typeName": "int",
            "cardinality": "SINGLE",
            "isIndexable": false,
            "isOptional": false,
            "isUnique": false
        },
        {
            "name": "tablename",
            "typeName": "string",
```

```
            "cardinality": "SINGLE",
            "isIndexable": false,
            "isOptional": false,
            "isUnique": false
        }
    ]
}
```

Example Response:

```
{
  "category": "STRUCT",
  "guid": "9527b5c2-49f7-4e25-bab0-a352d58fc2bf",
  "createTime": 1481887151201,
  "updateTime": 1481887151201,
  "version": 1,
  "name": "table_creation_order",
  "description": "table_creation_order",
  "typeVersion": "1.0",
  "attributeDefs": [
    {
      "name": "order",
      "typeName": "int",
      "isOptional": false,
      "cardinality": "SINGLE",
      "valuesMinCount": 1,
      "valuesMaxCount": 1,
      "isUnique": false,
      "isIndexable": false
    },
    {
      "name": "tablename",
      "typeName": "string",
      "isOptional": false,
      "cardinality": "SINGLE",
      "valuesMinCount": 1,
      "valuesMaxCount": 1,
      "isUnique": false,
      "isIndexable": false
    }
  ]
}
```

6.2.5.2.2. Get Struct Type Definition Using Name

Request:

```
GET /v2/types/structdef/name/{name}
```

Description:

Returns the definition of the Struct type with the given name.

Method Signature:

```
@GET
@Path("/structdef/name/{name}")
@Produces(Servlets.JSON_MEDIA_TYPE)
public AtlasStructDef getStructDefByName(@PathParam("name") String name)
 throws AtlasBaseException {
```

Example Request:

```
GET v2/types/structdef/name/table_creation_order
```

Example Response:

```
{
  "category": "STRUCT",
  "guid": "9527b5c2-49f7-4e25-bab0-a352d58fc2bf",
  "createTime": 1481887151201,
  "updateTime": 1481887151201,
  "version": 1,
  "name": "table_creation_order",
  "typeVersion": "1.0",
  "attributeDefs": [
    {
      "name": "order",
      "typeName": "int",
      "isOptional": false,
      "cardinality": "SINGLE",
      "valuesMinCount": 1,
      "valuesMaxCount": 1,
      "isUnique": false,
      "isIndexable": false
    },
    {
      "name": "tablename",
      "typeName": "string",
      "isOptional": false,
      "cardinality": "SINGLE",
      "valuesMinCount": 1,
      "valuesMaxCount": 1,
      "isUnique": false,
      "isIndexable": false
    }
  ]
}
```

6.2.5.2.3. Get Struct Type Definition Using GUID

Request:

```
GET v2/types/structdef/guid/{guid}
```

Description:

Returns the definition of the Struct type by referencing its GUID.

Method Signature:

```
@GET
@Path("/structdef/guid/{guid}")
@Produces(Servlets.JSON_MEDIA_TYPE)
public AtlasStructDef getStructDefByGuid(@PathParam("guid") String guid)
 throws AtlasBaseException {
```

Example Request:

```
GET v2/types/structdef/guid/9527b5c2-49f7-4e25-bab0-a352d58fc2bf
```

Example Response:

```
{
  "category": "STRUCT",
  "guid": "9527b5c2-49f7-4e25-bab0-a352d58fc2bf",
  "createTime": 1481887151201,
  "updateTime": 1481887151201,
  "version": 1,
  "name": "table_creation_order",
  "typeVersion": "1.0",
  "attributeDefs": [
    {
      "name": "order",
      "typeName": "int",
      "isOptional": false,
      "cardinality": "SINGLE",
      "valuesMinCount": 1,
      "valuesMaxCount": 1,
      "isUnique": false,
      "isIndexable": false
    },
    {
      "name": "tablename",
      "typeName": "string",
      "isOptional": false,
      "cardinality": "SINGLE",
      "valuesMinCount": 1,
      "valuesMaxCount": 1,
      "isUnique": false,
      "isIndexable": false
    }
  ]
}
```

6.2.5.2.4. Update Struct Type Using Name

Request:

```
PUT /v2/types/structdef/name/{name}
```

Description:

This request is used to update a Struct type definition by referencing its name.

Method Signature:

```
@PUT
@Path("/structdef/name/{name}")
@Consumes(Servlets.JSON_MEDIA_TYPE)
@Produces(Servlets.JSON_MEDIA_TYPE)
public AtlasStructDef updateStructDefByName(@PathParam("name") String name,
 AtlasStructDef structDef) throws AtlasBaseException {
```

Example Request:

```
PUT v2/types/structdef/name/table_creation_order
```

Example Request Body:

```
{
```

```
    "name": "table_creation_order",
    "typeVersion": "1.0",
    "attributeDefs": [
        {
            "name": "order",
            "typeName": "int",
            "cardinality": "SINGLE",
            "isIndexable": false,
            "isOptional": false,
            "isUnique": false
        },
        {
            "name": "tablename",
            "typeName": "string",
            "cardinality": "SINGLE",
            "isIndexable": false,
            "isOptional": false,
            "isUnique": false
        },
        {
         "name": "before_tablename_guid",
            "typeName": "string",
            "cardinality": "SINGLE",
            "isIndexable": false,
            "isOptional": true,
            "isUnique": false
        }
    ]
}
```

Example Response:

```
{
  "category": "STRUCT",
  "guid": "9527b5c2-49f7-4e25-bab0-a352d58fc2bf",
  "createTime": 1481887151201,
  "updateTime": 1481887614746,
  "version": 2,
  "name": "table_creation_order",
  "description": "table_creation_order",
  "typeVersion": "1.0",
  "attributeDefs": [
    {
      "name": "order",
      "typeName": "int",
      "isOptional": false,
      "cardinality": "SINGLE",
      "valuesMinCount": 1,
      "valuesMaxCount": 1,
      "isUnique": false,
      "isIndexable": false
    },
    {
      "name": "tablename",
      "typeName": "string",
      "isOptional": false,
      "cardinality": "SINGLE",
      "valuesMinCount": 1,
      "valuesMaxCount": 1,
      "isUnique": false,
```

```
      "isIndexable": false
    },
    {
      "name": "before_tablename_guid",
      "typeName": "string",
      "isOptional": true,
      "cardinality": "SINGLE",
      "valuesMinCount": 0,
      "valuesMaxCount": 1,
      "isUnique": false,
      "isIndexable": false
    }
  ]
}
```

6.2.5.2.5. Update Struct Type Using GUID

Request:

```
PUT /v2/types/structdef/guid/{guid}
```

Description:

This request is used to update a Struct type by referencing its GUID.

Method Signature:

```
@PUT
@Path("/structdef/guid/{guid}")
@Consumes(Servlets.JSON_MEDIA_TYPE)
@Produces(Servlets.JSON_MEDIA_TYPE)
public AtlasStructDef updateStructDefByGuid(@PathParam("guid") String guid,
 AtlasStructDef structDef) throws AtlasBaseException {
```

Example Request:

```
PUT v2/types/structdef/guid/9527b5c2-49f7-4e25-bab0-a352d58fc2bf
```

Example Response:

```
{
  "category": "STRUCT",
  "guid": "9527b5c2-49f7-4e25-bab0-a352d58fc2bf",
  "createTime": 1481887151201,
  "updateTime": 1481887614746,
  "version": 2,
  "name": "table_creation_order",
  "description": "table_creation_order",
  "typeVersion": "1.0",
  "attributeDefs": [
    {
      "name": "order",
      "typeName": "int",
      "isOptional": false,
      "cardinality": "SINGLE",
      "valuesMinCount": 1,
      "valuesMaxCount": 1,
      "isUnique": false,
      "isIndexable": false
```

```
    },
    {
      "name": "tablename",
      "typeName": "string",
      "isOptional": false,
      "cardinality": "SINGLE",
      "valuesMinCount": 1,
      "valuesMaxCount": 1,
      "isUnique": false,
      "isIndexable": false
    },
    {
      "name": "before_tablename_guid",
      "typeName": "string",
      "isOptional": true,
      "cardinality": "SINGLE",
      "valuesMinCount": 0,
      "valuesMaxCount": 1,
      "isUnique": false,
      "isIndexable": false
    }
  ]
}
```

6.2.5.2.6. Delete Struct Type Definition Using Name

Request:

```
DELETE /v2/types/structdef/name/{name}
```

Description:

Deletes a Struct type definition by referencing its name.

Method Signature:

```
@DELETE
@Path("/structdef/name/{name}")
@Produces(Servlets.JSON_MEDIA_TYPE)
public void deleteStructDefByName(@PathParam("name") String name) throws
 AtlasBaseException {
```

Example Request:

```
DELETE v2/types/structdef/name/table_creation_order
```

6.2.5.2.7. Delete Struct Type Definition Using GUID

Request:

```
DELETE /v2/types/structdef/guid/{guid}
```

Description:

Deletes a Struct type definition by referencing its GUID.

Method Signature:

```
@DELETE
@Path("/structdef/guid/{guid}")
@Produces(Servlets.JSON_MEDIA_TYPE)
public void deleteStructDefByGuid(@PathParam("guid") String guid) throws
 AtlasBaseException {
```

Example Request:

```
DELETE v2/types/structdef/guid/9527b5c2-49f7-4e25-bab0-a352d58fc2bf
```

6.2.5.3. Classification Type API

- Base resource name: `v2/types/classificationdef`
- Full URL: `http://<atlas-server-host:port>/api/atlas/v2/types/classificationdef`

6.2.5.3.1. Register Classification Type

Request:

```
POST /v2/types/classificationdef
```

Description:

This request is used to register a Classification type with the Atlas type system.

Method Signature:

```
@POST
@Path("/classificationdef")
@Consumes(Servlets.JSON_MEDIA_TYPE)
@Produces(Servlets.JSON_MEDIA_TYPE)
public AtlasClassificationDef createClassificationDef(AtlasClassificationDef
 classificationDef) throws AtlasBaseException {
```

Example Request:

```
POST v2/types/classificationdef
```

Example Request Body:

```
{
    "name": "Secured_Data",
    "typeVersion": "1.0",
    "superTypes" : [],
    "attributeDefs": [
        {
            "name": "allowed_groups",
            "typeName": "array<string>",
            "cardinality": "LIST",
            "isIndexable": true,
            "isOptional": true,
            "isUnique": false
        }
    ]
}
```

Example Response:

```
{
  "category": "CLASSIFICATION",
  "guid": "c8011ad1-bf60-4e9c-a77c-b1f947435ece",
  "createTime": 1481979168876,
  "updateTime": 1481979168876,
  "version": 1,
  "name": "Secured_Data",
  "description": "Secured_Data",
  "typeVersion": "1.0",
  "attributeDefs": [
    {
      "name": "allowed_groups",
      "typeName": "array<string>",
      "isOptional": true,
      "cardinality": "LIST",
      "valuesMinCount": 0,
      "valuesMaxCount": 2147483647,
      "isUnique": false,
      "isIndexable": true
    }
  ],
  "superTypes": []
}
```

6.2.5.3.2. Get Classification Type Definition Using Name

Request:

```
GET /v2/types/classificationdef/name/{name}
```

Description:

Returns the definition of the Classification type with the given name.

Method Signature:

```
@GET
@Path("/classificationdef/name/{name}")
@Produces(Servlets.JSON_MEDIA_TYPE)
public AtlasClassificationDef getClassificationDefByName(@PathParam("name")
 String name) throws AtlasBaseException {
```

Example Request:

```
GET v2/types/classificationdef/name/Secured_Data
```

Example Response:

```
{
  "category": "CLASSIFICATION",
  "guid": "c8011ad1-bf60-4e9c-a77c-b1f947435ece",
  "createTime": 1481979168876,
  "updateTime": 1481979168876,
  "version": 1,
  "name": "Secured_Data",
  "typeVersion": "1.0",
  "attributeDefs": [
```

```
    {
      "name": "allowed_groups",
      "typeName": "array<string>",
      "isOptional": true,
      "cardinality": "LIST",
      "valuesMinCount": 1,
      "valuesMaxCount": 1,
      "isUnique": false,
      "isIndexable": true
    }
  ],
  "superTypes": []
}
```

6.2.5.3.3. Get Classification Type Definition Using GUID

Request:

```
GET /v2/types/classificationdef/guid/{guid}
```

Description:

Returns the definition of the Classification type by referencing its GUID.

Method Signature:

```
@GET
@Path("/classificationdef/guid/{guid}")
@Produces(Servlets.JSON_MEDIA_TYPE)
public AtlasClassificationDef getClassificationDefByGuid(@PathParam("guid")
 String guid) throws AtlasBaseException {
```

Example Request:

```
GET v2/types/classificationdef/guid/c8011ad1-bf60-4e9c-a77c-b1f947435ece
```

Example Response:

```
{
  "category": "CLASSIFICATION",
  "guid": "c8011ad1-bf60-4e9c-a77c-b1f947435ece",
  "createTime": 1481979168876,
  "updateTime": 1481979168876,
  "version": 1,
  "name": "Secured_Data",
  "typeVersion": "1.0",
  "attributeDefs": [
    {
      "name": "allowed_groups",
      "typeName": "array<string>",
      "isOptional": true,
      "cardinality": "LIST",
      "valuesMinCount": 1,
      "valuesMaxCount": 1,
      "isUnique": false,
      "isIndexable": true
    }
  ],
  "superTypes": []
}
```

6.2.5.3.4. Update Classification Type Using Name

Request:

```
PUT /v2/types/classificationdef/name/{name}
```

Description:

This request is used to update a Classification type definition by referencing its name.

Method Signature:

```
@PUT
@Path("/classificationdef/name/{name}")
@Consumes(Servlets.JSON_MEDIA_TYPE)
@Produces(Servlets.JSON_MEDIA_TYPE)
public AtlasClassificationDef updateClassificationDefByName(@PathParam("name")
 String name, AtlasClassificationDef classificationDef)throws
 AtlasBaseException {
```

Example Request:

```
PUT /v2/types/name/Secured_Data
```

Example Request Body:

```
{
    "name": "Secured_Data",
    "typeVersion": "1.0",
    "superTypes" : [],
    "attributeDefs": [
        {
            "name": "allowed_groups",
            "typeName": "array<string>",
            "cardinality": "LIST",
            "isIndexable": true,
            "isOptional": true,
            "isUnique": false
        },
        {
            "name": "partial_access_group",
            "typeName": "array<string>",
            "cardinality": "LIST",
            "isIndexable": true,
            "isOptional": true,
            "isUnique": false
        }
    ]
}
```

Example Response:

```
{
  "category": "CLASSIFICATION",
  "guid": "c8011ad1-bf60-4e9c-a77c-b1f947435ece",
  "createTime": 1481979168876,
  "updateTime": 1482124162350,
  "version": 3,
  "name": "Secured_Data",
```

```
  "description": "Secured_Data",
  "typeVersion": "1.0",
  "attributeDefs": [
    {
      "name": "allowed_groups",
      "typeName": "array<string>",
      "isOptional": true,
      "cardinality": "LIST",
      "valuesMinCount": 0,
      "valuesMaxCount": 2147483647,
      "isUnique": false,
      "isIndexable": true
    },
    {
      "name": "partial_access_group",
      "typeName": "array<string>",
      "isOptional": true,
      "cardinality": "LIST",
      "valuesMinCount": 0,
      "valuesMaxCount": 2147483647,
      "isUnique": false,
      "isIndexable": true
    }
  ],
  "superTypes": []
}
```

6.2.5.3.5. Update Classification Type Using GUID

Request:

```
PUT /v2/types/classificationdef/guid/{guid}
```

Description:

This request is used to update a Classification type by referencing its GUID.

Method Signature:

```
@PUT
@Path("/classificationdef/guid/{guid}")
@Consumes(Servlets.JSON_MEDIA_TYPE)
@Produces(Servlets.JSON_MEDIA_TYPE)
public AtlasClassificationDef updateClassificationDefByGuid(@PathParam("guid")
 String guid, AtlasClassificationDef classificationDef) throws
 AtlasBaseException {
```

Example Request:

```
PUT /v2/types/guid/c8011ad1-bf60-4e9c-a77c-b1f947435ece
```

Example Response:

```
{
  "category": "CLASSIFICATION",
  "guid": "c8011ad1-bf60-4e9c-a77c-b1f947435ece",
  "createTime": 1481979168876,
  "updateTime": 1482124162350,
  "version": 3,
```

```
  "name": "Secured_Data",
  "description": "Secured_Data",
  "typeVersion": "1.0",
  "attributeDefs": [
    {
      "name": "allowed_groups",
      "typeName": "array<string>",
      "isOptional": true,
      "cardinality": "LIST",
      "valuesMinCount": 0,
      "valuesMaxCount": 2147483647,
      "isUnique": false,
      "isIndexable": true
    },
    {
      "name": "partial_access_group",
      "typeName": "array<string>",
      "isOptional": true,
      "cardinality": "LIST",
      "valuesMinCount": 0,
      "valuesMaxCount": 2147483647,
      "isUnique": false,
      "isIndexable": true
    }
  ],
  "superTypes": []
}
```

6.2.5.3.6. Delete Classification Type Definition Using Name

Request:

```
DELETE /v2/types/classificationdef/name/{name}
```

Description:

Deletes a Classification type definition by referencing its name.

Method Signature:

```
@DELETE
@Path("/classificationdef/name/{name}")
@Produces(Servlets.JSON_MEDIA_TYPE)
public void deleteClassificationDefByName(@PathParam("name") String name)
 throws AtlasBaseException {
```

Example Request:

```
DELETE /v2/types/classificationdef/name/Secured_Data
```

6.2.5.3.7. Delete Classification Type Definition Using GUID

Request:

```
DELETE /v2/types/classificationdef/guid/{guid}
```

Description:

Deletes a Classification type definition by referencing its GUID.

Method Signature:

```
@DELETE
@Path("/classificationdef/guid/{guid}")
@Produces(Servlets.JSON_MEDIA_TYPE)
public void deleteClassificationDefByGuid(@PathParam("guid") String guid)
 throws AtlasBaseException {
```

Example Request:

```
DELETE /v2/types/classificationdef/guid/c8011ad1-bf60-4e9c-a77c-b1f947435ece
```

6.2.5.3.8. Get All Classification Type Definitions

Request:

```
GET /v2/types/classificationdef
```

Description:

Returns all Classification type definitions.

Method Signature:

```
@GET
@Path("/classificationdef")
@Consumes(Servlets.JSON_MEDIA_TYPE)
@Produces(Servlets.JSON_MEDIA_TYPE)
public AtlasClassificationDefs searchClassificationDefs() throws
 AtlasBaseException {
```

Example Request:

```
GET /v2/types/classificationdef
```

Example Response:

```
{
  "list": [
    {
      "category": "CLASSIFICATION",
      "guid": "4697914e-fa53-4776-a54a-1deadf8f477f",
      "createTime": 1480595851386,
      "updateTime": 1480595851386,
      "version": 1,
      "name": "Data",
      "description": "kmf",
      "typeVersion": "1.0",
      "attributeDefs": [],
      "superTypes": []
    },
    {
      "category": "CLASSIFICATION",
      "guid": "c8011ad1-bf60-4e9c-a77c-b1f947435ece",
      "createTime": 1481979168876,
      "updateTime": 1482124162350,
      "version": 3,
      "name": "Secured_Data",
      "description": "Secured_Data",
      "typeVersion": "1.0",
```

```
      "attributeDefs": [
        {
          "name": "allowed_groups",
          "typeName": "array<string>",
          "isOptional": true,
          "cardinality": "LIST",
          "valuesMinCount": 0,
          "valuesMaxCount": 2147483647,
          "isUnique": false,
          "isIndexable": true
        },
        {
          "name": "partial_access_group",
          "typeName": "array<string>",
          "isOptional": true,
          "cardinality": "LIST",
          "valuesMinCount": 0,
          "valuesMaxCount": 2147483647,
          "isUnique": false,
          "isIndexable": true
        }
      ],
      "superTypes": []
    },
    {
      "category": "CLASSIFICATION",
      "guid": "0c2edbcc-c993-4071-95b0-4e831d1cca49",
      "createTime": 1480595843527,
      "updateTime": 1480595843527,
      "version": 1,
      "name": "PII",
      "description": "secure",
      "typeVersion": "1.0",
      "attributeDefs": [],
      "superTypes": []
    }
  ],
  "startIndex": 0,
  "pageSize": 3,
  "totalCount": 3,
  "sortType": "NONE"
}
```


6.2.5.4. Entity Type API

- Base resource name: `v2/types/entitydef`
- Full URL: `http://<atlas-server-host:port>/api/atlas/v2/types/entitydef`

6.2.5.4.1. Register Entity Type

Request:

```
POST /v2/types/entitydef
```

Description:

This request is used to register an Entity type with the Atlas type system.

Method Signature:

```
@POST
@Path("/entitydef")
@Consumes(Servlets.JSON_MEDIA_TYPE)
@Produces(Servlets.JSON_MEDIA_TYPE)
public AtlasEntityDef createEntityDef(AtlasEntityDef entityDef) throws
 AtlasBaseException {
```

Example Request:

```
POST /v2/types/entitydef
```

Example Request Body:

```
{
    "name": "spark_dataframe",
    "superTypes": [
        "DataSet"
    ],
    "typeVersion": "1.0",
    "attributeDefs": [
        {
            "name": "source",
            "typeName": "string",
            "cardinality": "SINGLE",
            "isIndexable": false,
            "isOptional": false,
            "isUnique": false
        },
        {
            "name": "destination",
            "typeName": "string",
            "cardinality": "SINGLE",
            "isIndexable": false,
            "isOptional": true,
            "isUnique": false
        }
    ]
}
```

Example Response:

```
{
  "category": "ENTITY",
  "guid": "fd47c0e9-7a06-488a-9831-8ebc92d7f332",
  "createTime": 1482130414242,
  "updateTime": 1482130414242,
  "version": 1,
  "name": "spark_dataframe",
  "typeVersion": "1.0",
  "attributeDefs": [
    {
      "name": "source",
      "typeName": "string",
      "isOptional": false,
      "cardinality": "SINGLE",
      "valuesMinCount": 1,
      "valuesMaxCount": 1,
      "isUnique": false,
```

```
      "isIndexable": false
    },
    {
      "name": "destination",
      "typeName": "string",
      "isOptional": true,
      "cardinality": "SINGLE",
      "valuesMinCount": 1,
      "valuesMaxCount": 1,
      "isUnique": false,
      "isIndexable": false
    }
  ],
  "superTypes": [
    "DataSet"
  ]
}
```

6.2.5.4.2. Get Entity Type Definition Using Name

Request:

```
GET /v2/types/entitydef/name/{name}
```

Description:

Returns the definition of the Entity type with the given name.

Method Signature:

```
@GET
@Path("/entitydef/name/{name}")
@Produces(Servlets.JSON_MEDIA_TYPE)
public AtlasEntityDef getEntityDefByName(@PathParam("name") String name)
 throws AtlasBaseException {
```

Example Request:

```
GET /v2/types/entitydef/name/spark_dataframe
```

Example Response:

```
{
  "category": "ENTITY",
  "guid": "fd47c0e9-7a06-488a-9831-8ebc92d7f332",
  "createTime": 1482130414242,
  "updateTime": 1482130414242,
  "version": 1,
  "name": "spark_dataframe",
  "typeVersion": "1.0",
  "attributeDefs": [
    {
      "name": "source",
      "typeName": "string",
      "isOptional": false,
      "cardinality": "SINGLE",
      "valuesMinCount": 1,
      "valuesMaxCount": 1,
      "isUnique": false,
```

```
      "isIndexable": false
    },
    {
      "name": "destination",
      "typeName": "string",
      "isOptional": true,
      "cardinality": "SINGLE",
      "valuesMinCount": 1,
      "valuesMaxCount": 1,
      "isUnique": false,
      "isIndexable": false
    }
  ],
  "superTypes": [
    "DataSet"
  ]
}
```

6.2.5.4.3. Get Entity Type Definition Using GUID

Request:

```
GET /v2/types/entitydef/guid/{guid}
```

Description:

Returns the definition of the Entity type by referencing its GUID.

Method Signature:

```
@GET
@Path("/entitydef/guid/{guid}")
@Produces(Servlets.JSON_MEDIA_TYPE)
public AtlasEnityDef getEntityDefByGuid(@PathParam("guid") String guid) throws
 AtlasBaseException {
```

Example Request:

```
GET /v2/types/entitydef/guid/fd47c0e9-7a06-488a-9831-8ebc92d7f332
```

Example Response:

```
{
  "category": "ENTITY",
  "guid": "fd47c0e9-7a06-488a-9831-8ebc92d7f332",
  "createTime": 1482130414242,
  "updateTime": 1482130414242,
  "version": 1,
  "name": "spark_dataframe",
  "typeVersion": "1.0",
  "attributeDefs": [
    {
      "name": "source",
      "typeName": "string",
      "isOptional": false,
      "cardinality": "SINGLE",
      "valuesMinCount": 1,
      "valuesMaxCount": 1,
      "isUnique": false,
```

```
      "isIndexable": false
    },
    {
      "name": "destination",
      "typeName": "string",
      "isOptional": true,
      "cardinality": "SINGLE",
      "valuesMinCount": 1,
      "valuesMaxCount": 1,
      "isUnique": false,
      "isIndexable": false
    }
  ],
  "superTypes": [
    "DataSet"
  ]
}
```

6.2.5.4.4. Update Entity Type Using Name

Request:

```
PUT /v2/types/entitydef/name/{name}
```

Description:

This request is used to update an Entity type definition by referencing its name.

Method Signature:

```
@PUT
@Path("/entitydef/name/{name}")
@Consumes(Servlets.JSON_MEDIA_TYPE)
@Produces(Servlets.JSON_MEDIA_TYPE)
@Experimental
public AtlasEntityDef updateEntityDefByName(@PathParam("name") String name,
 AtlasEntityDef entityDef) throws Exception {
```

Example Request:

```
PUT /v2/types/entitydef/name/spark_dataframe
```

Example Request Body:

```
{
    "name": "spark_dataframe",
    "superTypes": [
        "DataSet"
    ],
    "typeVersion": "1.0",
    "attributeDefs": [
        {
            "name": "source",
            "typeName": "string",
            "cardinality": "SINGLE",
            "isIndexable": false,
            "isOptional": false,
            "isUnique": false
        },
        {
```

```
            "name": "destination",
            "typeName": "string",
            "cardinality": "SINGLE",
            "isIndexable": false,
            "isOptional": true,
            "isUnique": false
        },
        {
         "name": "num_partitions",
            "typeName": "int",
            "cardinality": "SINGLE",
            "isIndexable": false,
            "isOptional": true,
            "isUnique": false
        }
    ]
}
```

Example Response:

```
{
  "category": "ENTITY",
  "guid": "fd47c0e9-7a06-488a-9831-8ebc92d7f332",
  "createTime": 1482130414242,
  "updateTime": 1482135739983,
  "version": 2,
  "name": "spark_dataframe",
  "description": "spark_dataframe",
  "typeVersion": "1.0",
  "attributeDefs": [
    {
      "name": "source",
      "typeName": "string",
      "isOptional": false,
      "cardinality": "SINGLE",
      "valuesMinCount": 1,
      "valuesMaxCount": 1,
      "isUnique": false,
      "isIndexable": false
    },
    {
      "name": "destination",
      "typeName": "string",
      "isOptional": true,
      "cardinality": "SINGLE",
      "valuesMinCount": 0,
      "valuesMaxCount": 1,
      "isUnique": false,
      "isIndexable": false
    },
    {
      "name": "num_partitions",
      "typeName": "int",
      "isOptional": true,
      "cardinality": "SINGLE",
      "valuesMinCount": 0,
      "valuesMaxCount": 1,
      "isUnique": false,
      "isIndexable": false
    }
```

```
  ],
  "superTypes": [
    "DataSet"
  ]
}
```

6.2.5.4.5. Update Entity Type Using GUID

Request:

```
PUT /v2/types/entitydef/guid/{guid}
```

Description:

This request is used to update an Entity type by referencing its GUID.

Method Signature:

```
@PUT
@Path("/entitydef/guid/{guid}")
@Consumes(Servlets.JSON_MEDIA_TYPE)
@Produces(Servlets.JSON_MEDIA_TYPE)
@Experimental
public AtlasEntityDef updateEntityDefByGuid(@PathParam("guid") String guid,
 AtlasEntityDef entityDef) throws Exception {
```

Example Request:

```
PUT /v2/types/entitydef/guid/fd47c0e9-7a06-488a-9831-8ebc92d7f332
```

Example Response:

```
{
  "category": "ENTITY",
  "guid": "fd47c0e9-7a06-488a-9831-8ebc92d7f332",
  "createTime": 1482130414242,
  "updateTime": 1482135739983,
  "version": 2,
  "name": "spark_dataframe",
  "description": "spark_dataframe",
  "typeVersion": "1.0",
  "attributeDefs": [
    {
      "name": "source",
      "typeName": "string",
      "isOptional": false,
      "cardinality": "SINGLE",
      "valuesMinCount": 1,
      "valuesMaxCount": 1,
      "isUnique": false,
      "isIndexable": false
    },
    {
      "name": "destination",
      "typeName": "string",
      "isOptional": true,
      "cardinality": "SINGLE",
      "valuesMinCount": 0,
      "valuesMaxCount": 1,
      "isUnique": false,
```

```
      "isIndexable": false
    },
    {
      "name": "num_partitions",
      "typeName": "int",
      "isOptional": true,
      "cardinality": "SINGLE",
      "valuesMinCount": 0,
      "valuesMaxCount": 1,
      "isUnique": false,
      "isIndexable": false
    }
  ],
  "superTypes": [
    "DataSet"
  ]
}
```

6.2.5.4.6. Delete Entity Type Definition Using Name

Request:

```
DELETE /v2/types/entitydef/name/{name}
```

Description:

Deletes an Entity type definition by referencing its name.

Method Signature:

```
@DELETE
@Path("/entitydef/name/{name}")
@Produces(Servlets.JSON_MEDIA_TYPE)
@Experimental
public void deleteEntityDef(@PathParam("name") String name) throws Exception
 {
```

Example Request:

```
DELETE /v2/types/entitydef/name/spark_dataframe
```

6.2.5.4.7. Delete Entity Type Definition Using GUID

Request:

```
DELETE /v2/types/entitydef/guid/{guid}
```

Description:

Deletes an Entity type definition by referencing its GUID.

Method Signature:

```
@DELETE
@Path("/entitydef/guid/{guid}")
@Produces(Servlets.JSON_MEDIA_TYPE)
@Experimental
public void deleteEntityDefByGuid(@PathParam("guid") String guid) throws
 Exception {
```

Example Request:

```
DELETE /v2/types/entitydef/guid/fd47c0e9-7a06-488a-9831-8ebc92d7f332
```

6.2.5.4.8. Get All Entity Type Definitions

Request:

```
GET /v2/types/entitydef
```

Description:

Returns all Entity type definitions.

Method Signature:

```
@GET
@Path("/entitydef")
@Produces(Servlets.JSON_MEDIA_TYPE)
public AtlasEntityDefs searchEntityDefs() throws AtlasBaseException {
```

Example Request:

```
GET /v2/types/entitydef
```

Example Response:

This request returns an extremely long response – not shown here due to space constraints.

6.2.5.5. Bulk Type System API

- Base resource name: `v2/types/typedefs`
- Full URL: `http://<atlas-server-host:port>/api/atlas/v2/types/typedefsf`

6.2.5.5.1. Get Entity Type Definition Headers

Request:

```
GET /v2/types/typedefs/headers
```

Description:

Returns headers (with minimal information) for all type definitions.

Method Signature:

```
@GET
@Path("/typedefs/headers")
@Produces(Servlets.JSON_MEDIA_TYPE)
public List<AtlasTypeDefHeader> getTypeDefHeaders() throws AtlasBaseException
 {
```

Example Request:

```
GET v2/types/typedefs/headers
```

Example Response:


```
[
  {
    "guid": "b3fd4d06-2a10-4722-9419-8210b00fddd0",
    "name": "hive_principal_type",
    "category": "ENUM"
  },
  {
    "guid": "c5642d55-9f8e-45b1-b4a9-709c97b46233",
    "name": "creation_order1",
    "category": "ENUM"
  },
  {
    "guid": "2ba8a2e0-63e6-4d5c-bc26-140fa95eb241",
    "name": "creation_order",
    "category": "ENUM"
  },
  {
    "guid": "011fe1e9-78a0-41bb-b7cf-6091a3e40424",
    "name": "file_action",
    "category": "ENUM"
  },
  {
    "guid": "a9547d7a-bee1-49af-a4f0-fcd8a128091f",
    "name": "hive_order",
    "category": "STRUCT"
  },
  {
    "guid": "18114653-958d-41c2-ac2c-5f7f5b5d181d",
    "name": "hive_serde",
    "category": "STRUCT"
  },
  {
    "guid": "191c0a7a-e4e7-4fac-9f60-d08b54c34ecb",
    "name": "fs_permissions",
    "category": "STRUCT"
  },
  {
    "guid": "63d0755e-8e54-4f5f-9266-fa8d5df20969",
    "name": "creation_order2",
    "category": "STRUCT"
  },
  {
    "guid": "4697914e-fa53-4776-a54a-1deadf8f477f",
    "name": "Data",
    "category": "CLASSIFICATION"
  },
  {
    "guid": "c8011ad1-bf60-4e9c-a77c-b1f947435ece",
    "name": "Secured_Data",
    "category": "CLASSIFICATION"
  },
  {
    "guid": "0c2edbcc-c993-4071-95b0-4e831d1cca49",
    "name": "PII",
    "category": "CLASSIFICATION"
  },
```

```
{
  "guid": "be6f7de1-73f6-41c4-b73d-96bc7840c0a4",
  "name": "hive_column_lineage",
  "category": "ENTITY"
},
{
  "guid": "1c4699e8-d441-45f4-a5ef-49663d7bdaf1",
  "name": "Asset",
  "category": "ENTITY"
},
{
  "guid": "0402c8c7-b052-4299-a38b-337b3a16aa8c",
  "name": "DataSet",
  "category": "ENTITY"
},
{
  "guid": "8e9478a6-9d07-4775-a3af-5090367caba4",
  "name": "hive_process",
  "category": "ENTITY"
},
{
  "guid": "30a85b88-ea84-4cc7-b8e2-e63f6ca053cd",
  "name": "storm_bolt",
  "category": "ENTITY"
},
{
  "guid": "9694b576-b93a-4a06-90b5-84df684d9387",
  "name": "hdfs_path",
  "category": "ENTITY"
},
{
  "guid": "e703d827-9974-4e83-a310-b412992f56f1",
  "name": "falcon_cluster",
  "category": "ENTITY"
},
{
  "guid": "f88a3b11-0049-4fed-9c2a-7e453610e395",
  "name": "storm_spout",
  "category": "ENTITY"
},
{
  "guid": "fd47c0e9-7a06-488a-9831-8ebc92d7f332",
  "name": "spark_dataframe",
  "category": "ENTITY"
},
{
  "guid": "60c93610-36d4-4e93-8e0d-98c26f387ac8",
  "name": "sqoop_process",
  "category": "ENTITY"
},
{
  "guid": "5ed52a18-9053-47eb-808b-15078f0660fd",
  "name": "Infrastructure",
  "category": "ENTITY"
},
{
  "guid": "7b6e53bd-1fd7-4952-bb27-b81feb8e1b5f",
  "name": "Referenceable",
  "category": "ENTITY"
},
```

```
{
  "guid": "cac1bf5b-0212-4655-885e-0343ef716776",
  "name": "falcon_feed_replication",
  "category": "ENTITY"
},
{
  "guid": "fd9f7e57-5335-44b0-acd0-85fb89d97616",
  "name": "Process",
  "category": "ENTITY"
},
{
  "guid": "3f6671de-5fd9-4057-a2f4-1a4dc5e85674",
  "name": "falcon_feed_creation",
  "category": "ENTITY"
},
{
  "guid": "64f91e28-9194-4146-8894-415677a0a12b",
  "name": "storm_topology",
  "category": "ENTITY"
},
{
  "guid": "3cacd545-d31a-4c95-ae76-ebb674a27486",
  "name": "kafka_topic",
  "category": "ENTITY"
},
{
  "guid": "a8514f62-9bd6-4457-960e-95a32b92757d",
  "name": "hive_table",
  "category": "ENTITY"
},
{
  "guid": "1280de4a-6187-43b2-b32b-2f742cbd5ffa",
  "name": "hive_storagedesc",
  "category": "ENTITY"
},
{
  "guid": "a26eec40-d6df-456a-9dbc-06c942822a4e",
  "name": "sqoop_dbdatastore",
  "category": "ENTITY"
},
{
  "guid": "6a417c47-1f3e-4f7e-b0b5-f26c85a238c5",
  "name": "hbase_table",
  "category": "ENTITY"
},
{
  "guid": "904f79d3-725a-4385-8124-75c0b7d937ec",
  "name": "hive_db",
  "category": "ENTITY"
},
{
  "guid": "b15ff0e3-da16-489d-b0ae-7ca903f7421b",
  "name": "jms_topic",
  "category": "ENTITY"
},
{
  "guid": "441137db-4758-4c00-ae1b-5a5e5079a8ef",
  "name": "hbase_namespace",
  "category": "ENTITY"
},
```

```
  {
    "guid": "bba56230-ed46-4311-a481-d4686d06b1d2",
    "name": "storm_node",
    "category": "ENTITY"
  },
  {
    "guid": "ee9cb57d-66b1-4568-9294-c9e779ecb054",
    "name": "fs_path",
    "category": "ENTITY"
  },
  {
    "guid": "d9c84e2b-d014-48a5-b294-93b10b9cb68a",
    "name": "hive_column",
    "category": "ENTITY"
  },
  {
    "guid": "9bfce300-1b70-4b6c-9a7c-edf9cedee0d2",
    "name": "falcon_feed",
    "category": "ENTITY"
  },
  {
    "guid": "b57449c9-cd6f-4969-b334-9337739b69b8",
    "name": "falcon_process",
    "category": "ENTITY"
  }
]
```

6.2.5.5.2. Get All Type Definitions

Request:

```
GET /v2/types/typedefs
```

Description:

Returns the complete definition of all types.

Method Signature:

```
@GET
@Path("/typedefs")
@Produces(Servlets.JSON_MEDIA_TYPE)
public AtlasTypesDef getAllTypeDefs() throws AtlasBaseException {
```

Example Request:

```
GET /v2/types/typedefs
```

Example Response:

This request returns an extremely long response – not shown here due to space constraints.

6.2.5.5.3. Bulk Create Type Definitions

Request:

```
POST /v2/types/typedefs
```

Description:

Bulk creates Atlas type definitions. Only new definitions are created. Any changes to existing definitions are ignored.

Method Signature:

```
@POST
@Path("/typedefs")
@Consumes(Servlets.JSON_MEDIA_TYPE)
@Produces(Servlets.JSON_MEDIA_TYPE)
public AtlasTypesDef createAtlasTypeDefs(final AtlasTypesDef typesDef) throws
 AtlasBaseException {
```

Example Request:

```
POST /v2/types/typedefs
```

Example Request Body:

```
{
 "enumDefs" : [],
 "structDefs" : [],
 "classificationDefs": [],
 "entityDefs" : [{
    "name": "spark_mysql_dataframe",
    "superTypes": [
        "DataSet",
        "spark_dataframe"
    ],
    "typeVersion": "1.0",
    "attributeDefs": [
        {
            "name": "database_url",
            "typeName": "string",
            "cardinality": "SINGLE",
            "isIndexable": false,
            "isOptional": false,
            "isUnique": false
        }
    ]
}]
}
```

Example Response:

```
{
  "enumDefs": [],
  "structDefs": [],
  "classificationDefs": [],
  "entityDefs": [
    {
      "category": "ENTITY",
      "guid": "30690807-ddec-432d-8641-8199e8cd57de",
      "createTime": 1482138649298,
      "updateTime": 1482138649298,
      "version": 1,
      "name": "spark_mysql_dataframe",
      "description": "spark_mysql_dataframe",
      "typeVersion": "1.0",
      "attributeDefs": [
```

```
        {
          "name": "database_url",
          "typeName": "string",
          "isOptional": false,
          "cardinality": "SINGLE",
          "valuesMinCount": 1,
          "valuesMaxCount": 1,
          "isUnique": false,
          "isIndexable": false
        }
      ],
      "superTypes": [
        "DataSet",
        "spark_dataframe"
      ]
    }
  ]
}
```

6.2.5.5.4. Bulk Update Type Definitions

Request:

```
PUT /v2/types/typedefs
```

Description:

Bulk updates all Atlas type definitions. Changes to existing definitions are persisted.

Method Signature:

```
@PUT
@Path("/typedefs")
@Consumes(Servlets.JSON_MEDIA_TYPE)
@Produces(Servlets.JSON_MEDIA_TYPE)
@Experimental
public AtlasTypesDef updateAtlasTypeDefs(final AtlasTypesDef typesDef) throws
 Exception {
```

Example Request:

In this example we add one more attribute to the `spark_mysql_dataframe` type that we registered in the POST request in the previous section.

```
PUT /v2/types/typedefs
```

Example Request Body:

```
{
"enumDefs" : [],
"structDefs" : [],
"classificationDefs": [],
"entityDefs" : [{
"name": "spark_mysql_dataframe",
"superTypes": [
    "DataSet",
    "spark_dataframe"
],
"typeVersion": "1.0",
```

```
"attributeDefs": [
    {
        "name": "database_url",
        "typeName": "string",
        "cardinality": "SINGLE",
        "isIndexable": false,
        "isOptional": false,
        "isUnique": false
    },
    {
     "name" : "instance_size",
     "typeName": "string",
        "cardinality": "SINGLE",
        "isIndexable": false,
        "isOptional": true,
        "isUnique": false
    }
]
}]
}
```

Example Response:

```
{
  "enumDefs": [],
  "structDefs": [],
  "classificationDefs": [],
  "entityDefs": [
    {
      "category": "ENTITY",
      "guid": "30690807-ddec-432d-8641-8199e8cd57de",
      "createTime": 1482138649298,
      "updateTime": 1482140154378,
      "version": 2,
      "name": "spark_mysql_dataframe",
      "description": "spark_mysql_dataframe",
      "typeVersion": "1.0",
      "attributeDefs": [
        {
          "name": "database_url",
          "typeName": "string",
          "isOptional": false,
          "cardinality": "SINGLE",
          "valuesMinCount": 1,
          "valuesMaxCount": 1,
          "isUnique": false,
          "isIndexable": false
        },
        {
          "name": "instance_size",
          "typeName": "string",
          "isOptional": true,
          "cardinality": "SINGLE",
          "valuesMinCount": 0,
          "valuesMaxCount": 1,
          "isUnique": false,
          "isIndexable": false
        }
      ],
      "superTypes": [
```

```
        "DataSet",
        "spark_dataframe"
      ]
    }
  ]
}
```

6.2.5.5.5. Bulk Delete Type Definitions

Request:

```
DELETE /v2/types/typedefs
```

Description:

Bulk deletes all Atlas type definitions provided in the request body.

Method Signature:

```
@DELETE
@Path("/typedefs")
@Consumes(Servlets.JSON_MEDIA_TYPE)
@Produces(Servlets.JSON_MEDIA_TYPE)
@Experimental
public void deleteAtlasTypeDefs(final AtlasTypesDef typesDef) {
```

Example Request:

```
DELETE /v2/types/typedefs
```

Example Request Body:

```
{
"enumDefs" : [],
"structDefs" : [],
"classificationDefs": [],
"entityDefs" : [{
"name": "spark_mysql_dataframe",
"superTypes": [
    "DataSet",
    "spark_dataframe"
],
"typeVersion": "1.0",
"attributeDefs": [
    {
        "name": "database_url",
        "typeName": "string",
        "cardinality": "SINGLE",
        "isIndexable": false,
        "isOptional": false,
        "isUnique": false
    },
    {
     "name" : "instance_size",
     "typeName": "string",
        "cardinality": "SINGLE",
        "isIndexable": false,
        "isOptional": true,
        "isUnique": false
```

```
    }
]
}]
}
```

6.2.6. Atlas Entity API

Summary:

These API endpoints can be used to create, read, update, and delete a single entity.

- Base resource name: `v2/entity`
- Full URL: `http://<atlas-server-host:port>/api/atlas/v2/entity`

6.2.6.1. Create or Update a Single Entity

Request:

```
POST v2/entity
```

Description:

Create or update a single entity.

Method Signature:

```
@POST
@Consumes({Servlets.JSON_MEDIA_TYPE, MediaType.APPLICATION_JSON})
@Produces(Servlets.JSON_MEDIA_TYPE)
public EntityMutationResponse createOrUpdate(final AtlasEntity entity) throws
 AtlasBaseException {
```

Example Request:

This example request creates a `spark_dataframe` entity type.

```
PUT /v2/entity/
```

Example Request Body:

```
{
  "typeName": "spark_dataframe",
  "attributes" : {

     "qualifiedName" : "spark_dataframe_qualifiedName",
     "name" : "spark_dataframe_entity",
    "source" : "hdfs://vimal-fenton-4-1.openstacklocal:8020/apps/hive/
warehouse/source",
    "destination" : "hdfs://vimal-fenton-4-1.openstacklocal:8020/apps/hive/
warehouse/destination"
  },
  "classifications": [

  ]

}
```

Example Response:

```
{
  "entitiesMutated": {
    "CREATE_OR_UPDATE": [
      {
        "guid": "b42350c8-665b-4ff9-bdb5-16cf432412f7"
      }
    ]
  }
}
```

6.2.6.2. Update Entity Using GUID

Request:

```
PUT /v2/entity/guid/{guid}
```

Description:

This request is used to update an Entity by referencing its GUID.

Method Signature:

```
@PUT
@Path("guid/{guid}")
@Consumes({Servlets.JSON_MEDIA_TYPE, MediaType.APPLICATION_JSON})
@Produces(Servlets.JSON_MEDIA_TYPE)
public EntityMutationResponse updateByGuid(@PathParam("guid") String guid,
 AtlasEntity entity, @DefaultValue("false") @QueryParam("partialUpdate")
 boolean partialUpdate) throws AtlasBaseException {
```

Example Request:

```
PUT /v2/entity/guid/b42350c8-665b-4ff9-bdb5-16cf432412f7
```

Example Request Body:

```
{
  "typeName": "spark_dataframe",
  "attributes" : {

     "qualifiedName" : "spark_dataframe_qualifiedName",
     "name" : "spark_dataframe_entity",
    "source" : "/new/source/path",
    "destination" : "/new/destination/path"
  },
  "classifications": [

  ]

}
```

Example Response:

```
{
  "entitiesMutated": {
    "CREATE_OR_UPDATE": [
      {
        "guid": "b42350c8-665b-4ff9-bdb5-16cf432412f7"
      }
    ]
  }
}
```

6.2.6.3. Get Entity Definition Using GUID

Request:

```
GET v2/entity/guid/{guid}
```

Description:

Returns an Entity definition by referencing its GUID.

Method Signature:

```
@GET
@Path("/guid/{guid}")
@Produces(Servlets.JSON_MEDIA_TYPE)
public AtlasEntity getById(@PathParam("guid") String guid) throws
 AtlasBaseException {
```

Example Request:

```
GET v2/entity/guid/b42350c8-665b-4ff9-bdb5-16cf432412f7
```

Example Response:

```
{
  "typeName": "spark_dataframe",
  "attributes": {
    "source": "/new/source/path",
    "description": null,
    "qualifiedName": "spark_dataframe_qualifiedName",
    "name": "spark_dataframe_entity",
    "owner": null,
    "destination": "/new/destination/path"
  },
  "guid": "b42350c8-665b-4ff9-bdb5-16cf432412f7"
}
```

6.2.6.4. Get Entity Definition and Associations Using GUID

Request:

```
GET v2/entity/guid/{guid}/associations
```

Description:

Returns an Entity definition and all of its associations, such as classifications and terms, by referencing its GUID.

Method Signature:

```
@GET
@Path("/guid/{guid}/associations")
@Produces(Servlets.JSON_MEDIA_TYPE)
public AtlasEntityWithAssociations
 getWithAssociationsByGuid(@PathParam("guid") String guid) throws
 AtlasBaseException {
```

Example Request:

```
GET v2/entity/guid/b42350c8-665b-4ff9-bdb5-16cf432412f7/associations
```

Example Response:

```
{
  "typeName": "spark_dataframe",
  "attributes": {
    "source": "/new/source/path",
    "description": null,
    "qualifiedName": "spark_dataframe_qualifiedName",
    "name": "spark_dataframe_entity",
    "owner": null,
    "destination": "/new/destination/path"
  },
  "guid": "b42350c8-665b-4ff9-bdb5-16cf432412f7"
}
```

6.2.6.5. Delete Entity Using GUID

Request:

```
DELETE /v2/entity/guid/{guid}
```

Description:

Deletes an Entity by referencing its GUID.

Method Signature:

```
@DELETE
@Path("guid/{guid}")
@Consumes({Servlets.JSON_MEDIA_TYPE, MediaType.APPLICATION_JSON})
@Produces(Servlets.JSON_MEDIA_TYPE)
public EntityMutationResponse deleteByGuid(@PathParam("guid") final String
 guid) throws AtlasBaseException {
```

Example Request:

```
DELETE v2/entity/guid/b42350c8-665b-4ff9-bdb5-16cf432412f7
```

Example Response:

```
{
  "entitiesMutated": {
    "DELETE": [
      {
        "guid": "b42350c8-665b-4ff9-bdb5-16cf432412f7"
      }
    ]
  }
}
```

6.2.6.6. Update a Subset of Entity Attributes

Request:

```
PUT /v2/entity/uniqueAttribute/type/{typeName}/attribute/{attrName}
```

Description:

Updates a subset of attributes based on Entity type and a unique attribute, such as `Referenceable.qualifiedName`. Null updates are not allowed.

Method Signature:

```
@Deprecated
@PUT
@Consumes(Servlets.JSON_MEDIA_TYPE)
@Produces(Servlets.JSON_MEDIA_TYPE)
@Path("/uniqueAttribute/type/{typeName}/attribute/{attrName}")
public EntityMutationResponse
 partialUpdateByUniqueAttribute(@PathParam("typeName") String entityType,
    @PathParam("attrName") String attribute,
    @QueryParam("value") String value, AtlasEntity entity) throws Exception {
```

Example Request:

```
PUT v2/entity/uniqueAttribute/type/spark_dataframe/attribute/qualifiedName?
value=spark_dataframe_qualifiedName
```

Example Request Body:

```
{
  "typeName": "spark_dataframe",
  "attributes" : {

    "qualifiedName" : "spark_dataframe_qualifiedName",
    "name" : "spark_dataframe_entity",
    "source" : "/new/source/path",
    "destination" : "/new/destination/path"
  },
  "classifications": [

  ]

}
```

Example Response:

```
{
  "entitiesMutated": {
    "CREATE_OR_UPDATE": [
      {
        "guid": "3942dc63-cdae-4f67-b7a3-e33723ffae3e"
      }
    ]
  }
}
```

6.2.6.7. Delete Entity Using Type and Unique Attribute

Request:

```
DELETE /v2/entity/uniqueAttribute/type/{typeName}/attribute/{attrName}
```

Description:

Deletes an Entity referenced by an Entity type and a unique attribute, such as `Referenceable.qualifiedName`.

Method Signature:

```
@Deprecated
@DELETE
@Consumes(Servlets.JSON_MEDIA_TYPE)
@Produces(Servlets.JSON_MEDIA_TYPE)
@Path("/uniqueAttribute/type/{typeName}/attribute/{attrName}")
public EntityMutationResponse deleteByUniqueAttribute(@PathParam("typeName")
 String entityType,
    @PathParam("attrName") String attribute,
    @QueryParam("value") String value) throws Exception {
```

Example Request:

```
DELETE v2/entity/uniqueAttribute/type/spark_dataframe/attribute/qualifiedName?
value=spark_dataframe_qualifiedName
```

Example Response:

```
{
  "entitiesMutated": {
    "DELETE": [
      {
        "guid": "3942dc63-cdae-4f67-b7a3-e33723ffae3e"
      }
    ]
  }
}
```

6.2.6.8. Get Entity Definition Using Type and Unique Attribute

Request:

```
GET /v2/entity/uniqueAttribute/type/{typeName}/attribute/{attrName}
```

Description:

Returns an Entity definition by referencing its Entity type and a unique attribute, such as `Referenceable.qualifiedName`.

Method Signature:

```
@Deprecated
@GET
@Consumes({Servlets.JSON_MEDIA_TYPE, MediaType.APPLICATION_JSON})
@Produces(Servlets.JSON_MEDIA_TYPE)
@Path("/uniqueAttribute/type/{typeName}/attribute/{attrName}")
public AtlasEntity getByUniqueAttribute(@PathParam("typeName") String
 entityType,
    @PathParam("attrName") String attribute,
    @QueryParam("value") String value) throws AtlasBaseException {
```

Example Request:

```
GET v2/entity/uniqueAttribute/type/spark_dataframe/attribute/qualifiedName?
value=spark_dataframe_qualifiedName
```

Example Response:

```
{
  "typeName": "spark_dataframe",
  "attributes": {
    "source": "hdfs://vimal-fenton-4-1.openstacklocal:8020/apps/hive/
warehouse/source",
    "description": null,
    "qualifiedName": "spark_dataframe_qualifiedName",
    "name": "spark_dataframe_entity",
    "owner": null,
    "destination": "hdfs://vimal-fenton-4-1.openstacklocal:8020/apps/hive/
warehouse/destination"
  },
  "guid": "6fe47044-04f8-4bb1-9abf-765d7a5ada2a"
}
```

6.2.6.9. Add Classifications to an Entity Referenced by GUID

Request:

```
POST v2/entity/guid/{guid}/classifications
```

Description:

Adds classifications to an Entity referenced by its GUID.

Method Signature:

```
@POST
@Path("/guid/{guid}/classifications")
@Consumes({Servlets.JSON_MEDIA_TYPE, MediaType.APPLICATION_JSON})
@Produces(Servlets.JSON_MEDIA_TYPE)
public void addClassifications(@PathParam("guid") final String guid,
 List<AtlasClassification> classifications) throws AtlasBaseException {
```

Example Request:

```
POST v2/entity/guid/6fe47044-04f8-4bb1-9abf-765d7a5ada2a/classifications
```

Example Request Body:

```
[
    {
     "typeName" : "PII"
    }
]
```

6.2.6.10. Update Entity Classifications Referenced by GUID

Request:

```
PUT v2/entity/guid/{guid}/classifications
```

Description:

Updates one or more classifications of an Entity referenced by its GUID.

Method Signature:

```
@PUT
@Path("/guid/{guid}/classifications")
@Consumes({Servlets.JSON_MEDIA_TYPE, MediaType.APPLICATION_JSON})
@Produces(Servlets.JSON_MEDIA_TYPE)
public void updateClassifications(@PathParam("guid") final String guid,
 List<AtlasClassification> classifications) throws AtlasBaseException {
```

Example Request:

```
PUT v2/entity/guid/6fe47044-04f8-4bb1-9abf-765d7a5ada2a/classifications
```

Example Request Body:

```
[
    {
     "typeName" : "PII"
    },
    {
     "typeName" : "Secure"
    }
]
```

6.2.6.11. Get Entity Classifications Using GUID

Request:

```
GET v2/entity/guid/{guid}/classifications
```

Description:

Returns the classifications of an Entity referenced by its GUID.

Method Signature:

```
@GET
@Path("/guid/{guid}/classifications")
@Produces(Servlets.JSON_MEDIA_TYPE)
public AtlasClassification.AtlasClassifications
 getClassifications(@PathParam("guid") String guid) throws AtlasBaseException
 {
```

Example Request:

```
GET v2/entity/guid/6fe47044-04f8-4bb1-9abf-765d7a5ada2a/classifications
```

Example Response:

```
GET v2/entity/guid/6fe47044-04f8-4bb1-9abf-765d7a5ada2a/classifications
```

```
Response
{
  "list": [
    {
      "typeName": "PII"
    },
    {
      "typeName": "Secure"
    }
  ],
  "startIndex": 0,
  "pageSize": 0,
  "totalCount": 0
}
```

6.2.6.12. Delete Entity Classification Using GUID

Request:

```
DELETE v2/entity/guid/{guid}/classification/PII
```

Description:

Deletes a given classification of an Entity referenced by its GUID.

Method Signature:

```
@DELETE
@Path("/guid/{guid}/classification/{classificationName}")
@Consumes({Servlets.JSON_MEDIA_TYPE, MediaType.APPLICATION_JSON})
@Produces(Servlets.JSON_MEDIA_TYPE)
public void deleteClassification(@PathParam("guid") String guid,
    @PathParam("classificationName") String classificationName) throws
 AtlasBaseException {
```

Example Request:

```
DELETE v2/entity/guid/6fe47044-04f8-4bb1-9abf-765d7a5ada2a/classification/PII
```

Example Response:

After running the DELETE, the following GET:

```
GET v2/entity/guid/6fe47044-04f8-4bb1-9abf-765d7a5ada2a/classifications
```

Returns the following response:

```
{
  "list": [
    {
      "typeName": "Secure"
    }
  ],
  "startIndex": 0,
  "pageSize": 0,
  "totalCount": 0
}
```

6.2.7. Atlas Entities API

You can use the Atlas Entities API to create, read, update, and delete multiple entities with a single request.

Summary:

- Base resource name: `v2/entities`
- Full URL: `http://<atlas-server-host:port>/api/atlas/v2/entities`

6.2.7.1. Create or Update Entities

Request:

```
POST /v2/entities
```

Description:

Creates new entities or updates existing entities. An existing entity is referenced by its GUID or by a unique attribute such as `qualifiedName`. Any associations such as Classifications or Business Terms must be assigned using the applicable API.

Method Signature:

```
@POST
@Consumes(Servlets.JSON_MEDIA_TYPE)
@Produces(Servlets.JSON_MEDIA_TYPE)
public EntityMutationResponse createOrUpdate(List<AtlasEntity> entities)
 throws AtlasBaseException {
```

Example Request:

```
POST /v2/entities
```

Example Request Body:

The body for this POST request registers multiple `spark_dataframe` entities.

```
[{
  "typeName": "spark_dataframe",
  "attributes" : {
     "qualifiedName" : "dataframe_jobID_1234",
     "name" : "dataframe_1",
    "source" : "hdfs://vimal-fenton-4-1.openstacklocal:8020/apps/hive/
warehouse/source_1",
    "destination" : "hdfs://vimal-fenton-4-1.openstacklocal:8020/apps/hive/
warehouse/destination_1"
  },
  "classifications": [

  ]

},
{
```

```
  "typeName": "spark_dataframe",
  "attributes" : {

     "qualifiedName" : "dataframe_jobID_9349",
     "name" : "dataframe_2",
    "source" : "hdfs://vimal-fenton-4-1.openstacklocal:8020/apps/hive/
warehouse/source_2",
    "destination" : "hdfs://vimal-fenton-4-1.openstacklocal:8020/apps/hive/
warehouse/destination_2"
  },
  "classifications": [

  ]

}
]
```

Example Response:

```
{
  "entitiesMutated": {
    "CREATE_OR_UPDATE": [
      {
        "guid": "4d6e5368-609a-4ba9-9153-6dcf4759de0a"
      },
      {
        "guid": "fbe78388-0eed-4439-b738-5fd1a2f9db68"
      }
    ]
  }
}
```

6.2.7.2. Update Entities

Request:

```
PUT /v2/entities
```

Description:

Updates the specified entities. Any associations such as Classifications or Business Terms must be assigned using the applicable API. Null updates are supported, for example, setting optional attributes to Null.

Method Signature:

```
@PUT
@Consumes(Servlets.JSON_MEDIA_TYPE)
@Produces(Servlets.JSON_MEDIA_TYPE)
public EntityMutationResponse update(List<AtlasEntity> entities) throws
 AtlasBaseException {
```

Example Request:

```
PUT /v2/entities
```

Example Request Body:

```
[{
  "typeName": "spark_dataframe",
  "attributes" : {
     "qualifiedName" : "dataframe_jobID_1234",
     "name" : "updated_dataframe_1",
    "source" : "hdfs://vimal-fenton-4-1.openstacklocal:8020/apps/hive/
warehouse/source_1",
    "destination" : "hdfs://vimal-fenton-4-1.openstacklocal:8020/apps/hive/
warehouse/destination_1"
  },
  "classifications": [

  ]

},
{
  "typeName": "spark_dataframe",
  "attributes" : {

     "qualifiedName" : "dataframe_jobID_9349",
     "name" : "updated_dataframe_2",
    "source" : "hdfs://vimal-fenton-4-1.openstacklocal:8020/apps/hive/
warehouse/source_2",
    "destination" : "hdfs://vimal-fenton-4-1.openstacklocal:8020/apps/hive/
warehouse/destination_2"
  },
  "classifications": [

  ]

}
]
```

Example Response:

```
{
  "entitiesMutated": {
    "CREATE_OR_UPDATE": [
      {
        "guid": "4d6e5368-609a-4ba9-9153-6dcf4759de0a"
      },
      {
        "guid": "fbe78388-0eed-4439-b738-5fd1a2f9db68"
      }
    ]
  }
}
```

6.2.7.3. Get Entities Definitions

Request:

```
GET /v2/entities/guids?guid=list<guid>
```

Description:

Returns the Entity definitions referenced by the GUIDs passed in the request.

Method Signature:

```
@GET
@Path("/guids")
@Consumes(Servlets.JSON_MEDIA_TYPE)
@Produces(Servlets.JSON_MEDIA_TYPE)
public AtlasEntity.AtlasEntities getById(@QueryParam("guid") List<String>
 guids) throws AtlasBaseException {
```

Example Request:

```
GET v2/entities/guids?guid=fbe78388-0eed-4439-b738-5fd1a2f9db68&guid=
fbe78388-0eed-4439-b738-5fd1a2f9db68
```

Example Response:

```
{
  "list": [
    {
      "typeName": "spark_dataframe",
      "attributes": {
        "source": "hdfs://vimal-fenton-4-1.openstacklocal:8020/apps/hive/
warehouse/source_1",
        "description": null,
        "qualifiedName": "dataframe_jobID_1234",
        "name": "updated_dataframe_1",
        "owner": null,
        "destination": "hdfs://vimal-fenton-4-1.openstacklocal:8020/apps/hive/
warehouse/destination_1"
      },
      "guid": "4d6e5368-609a-4ba9-9153-6dcf4759de0a"
    },
    {
      "typeName": "spark_dataframe",
      "attributes": {
        "source": "hdfs://vimal-fenton-4-1.openstacklocal:8020/apps/hive/
warehouse/source_2",
        "description": null,
        "qualifiedName": "dataframe_jobID_9349",
        "name": "updated_dataframe_2",
        "owner": null,
        "destination": "hdfs://vimal-fenton-4-1.openstacklocal:8020/apps/hive/
warehouse/destination_2"
      },
      "guid": "fbe78388-0eed-4439-b738-5fd1a2f9db68"
    }
  ],
  "startIndex": 0,
  "pageSize": 0,
  "totalCount": 0
}
```

6.2.7.4. Delete Entities

Request:

```
DELETE /v2/entities/guids?guid=list<guid>
```

Description:

Deletes the entities referenced by the GUIDs passed in the request.

Method Signature:

```
@DELETE
@Path("/guids")
@Consumes(Servlets.JSON_MEDIA_TYPE)
@Produces(Servlets.JSON_MEDIA_TYPE)
public EntityMutationResponse deleteById(@QueryParam("guid") final
 List<String> guids) throws AtlasBaseException {
```

Example Request:

```
DELETE /v2/entities/guids?guid=4d6e5368-609a-4ba9-9153-6dcf4759de0a&guid=
fbe78388-0eed-4439-b738-5fd1a2f9db68
```

Example Response:

```
{
  "entitiesMutated": {
    "DELETE": [
      {
        "guid": "4d6e5368-609a-4ba9-9153-6dcf4759de0a"
      },
      {
        "guid" : "fbe78388-0eed-4439-b738-5fd1a2f9db68"
      }
    ]
  }
}
```

6.2.8. Atlas Lineage API

In the previous section we registered a `spark_dataframe` type. In this section we will use a `spark_dataframe` type in order to describe the Atlas Lineage API.

We will start by registering a `spark_transformation` type. As the name suggests, a `spark_transformation` symbolizes a transformation of a `spark_dataframe`. The `spark_transformation` type extends the `Process` type.

```
POST /v2/types/entitydef
```

```
{
  "category" : "ENTITY",
  "name" : "spark_transformation",
  "superTypes" : [
    "Process"
  ],
  "typeVersion" : "1.0",
  "attributeDefs" : [
    {
      "name" : "parallelism",
      "typeName" : "int",
      "cardinality" : "SINGLE",
      "isIndexable" : false,
      "isOptional" : true,
      "isUnique" : false
    }
  ]
}
```

Next we will create two `spark_dataframe` entities:

```
POST /v2/entity/
```

```
{
  "typeName": "spark_dataframe",
  "attributes" : {
    "qualifiedName" : "source_dataframe@clusterName",
    "name" : "source_dataframe",
    "source" : "hdfs://vimal-fenton-4-1.openstacklocal:8020/apps/hive/
warehouse/source",
    "destination" : "hdfs://vimal-fenton-4-1.openstacklocal:8020/apps/hive/
warehouse/destination"
  },
  "classifications": [

  ]
}
```

```
POST /v2/entity/
```

```
{
  "typeName": "spark_dataframe",
  "attributes" : {
    "qualifiedName" : "destination_dataframe@clusterName",
    "name" : "destination_dataframe",
    "source" : "hdfs://vimal-fenton-4-1.openstacklocal:8020/apps/hive/
warehouse/source_2",
    "destination" : "hdfs://vimal-fenton-4-1.openstacklocal:8020/apps/hive/
warehouse/destination_2"
  },
  "classifications": [

  ]
}
```

Next we will register a `spark_transformation` entity, which will establish a relationship between the two `spark_dataframe` entities:

```
POST /v2/entity/
```

Request Body:

```
{
  "typeName": "spark_transformation",
  "attributes" : {
    "qualifiedName" : "spark_process_id_24343324",
    "name" : "spark_process",
  "parallelism" : "10",
  "inputs" : [
      {
        "typeName": "spark_dataframe",
        "attributes": {
          "source": "hdfs://vimal-fenton-4-1.openstacklocal:8020/apps/hive/
warehouse/source",
          "description": null,
          "qualifiedName": "source_dataframe@clusterName",
          "name": "source_dataframe",
          "owner": null,
          "destination": "hdfs://vimal-fenton-4-1.openstacklocal:8020/apps/
hive/warehouse/destination"
        },
        "guid": "c94d8450-6d59-4cd1-8732-863286387c7d"
```

```
      }
    ],
    "outputs" : [
    {
      "typeName": "spark_dataframe",
      "attributes": {
        "source": "hdfs://vimal-fenton-4-1.openstacklocal:8020/apps/hive/
warehouse/source_2",
        "description": null,
        "qualifiedName": "destination_dataframe@clusterName",
        "name": "destination_dataframe",
        "owner": null,
        "destination": "hdfs://vimal-fenton-4-1.openstacklocal:8020/apps/hive/
warehouse/destination_2"
      },
      "guid": "86d5cb10-538a-40cd-80c9-22fc363224d0"
    }
  ]
  },
  "classifications": [

  ]
}
```

Response:

```
{
  "entitiesMutated": {
    "CREATE_OR_UPDATE": [
      {
        "guid": "58a3ee5d-827f-4aa4-98e1-ccebd5851c76"
      }
    ]
  }
}
```

Now we can use the Lineage API to retrieve the lineage information of the `spark_dataframe`:

Method Signature:

```
@GET
@Path("/{guid}")
@Consumes(Servlets.JSON_MEDIA_TYPE)
@Produces(Servlets.JSON_MEDIA_TYPE)
public AtlasLineageInfo getLineageGraph(@PathParam("guid") String guid,
@QueryParam("direction") @DefaultValue(DEFAULT_DIRECTION)  LineageDirection
 direction,
@QueryParam("depth") @DefaultValue(DEFAULT_DEPTH) int depth) throws
 AtlasBaseException {
```

Example Request:

```
GET v2/lineage/c94d8450-6d59-4cd1-8732-863286387c7d
```

Example Response:

```
{
  "baseEntityGuid": "c94d8450-6d59-4cd1-8732-863286387c7d",
  "lineageDirection": "BOTH",
  "lineageDepth": 3,
```

```
  "guidEntityMap": {
    "58a3ee5d-827f-4aa4-98e1-ccebd5851c76": {
      "typeName": "spark_transformation",
      "guid": "58a3ee5d-827f-4aa4-98e1-ccebd5851c76",
      "status": "STATUS_ACTIVE",
      "displayText": "spark_process_id_24343324"
    },
    "c94d8450-6d59-4cd1-8732-863286387c7d": {
      "typeName": "spark_dataframe",
      "guid": "c94d8450-6d59-4cd1-8732-863286387c7d",
      "status": "STATUS_ACTIVE",
      "displayText": "source_dataframe@clusterName"
    },
    "86d5cb10-538a-40cd-80c9-22fc363224d0": {
      "typeName": "spark_dataframe",
      "guid": "86d5cb10-538a-40cd-80c9-22fc363224d0",
      "status": "STATUS_ACTIVE",
      "displayText": "destination_dataframe@clusterName"
    }
  },
  "relations": [
    {
      "fromEntityId": "c94d8450-6d59-4cd1-8732-863286387c7d",
      "toEntityId": "58a3ee5d-827f-4aa4-98e1-ccebd5851c76"
    },
    {
      "fromEntityId": "58a3ee5d-827f-4aa4-98e1-ccebd5851c76",
      "toEntityId": "86d5cb10-538a-40cd-80c9-22fc363224d0"
    }
  ]
}
```

We can specify the direction of the lineage information in the API call. There are three possible values for direction: INPUT, OUTPUT, and BOTH. The direction is BOTH by default. If we specify direction as INPUT in the API call, the result set contains only the input entities from which the given entity has been derived. Similarly, if we specify the direction as OUTPUT, the result set contains all of the entities derived from the given entity.

We can also specify the depth of the lineage results. If we specify depth, Atlas fetches all of the entities that lie within the given depth in the entity lineage diagram .

To demonstrate the query parameters in the Lineage API call, we will register another entity of type `hdfs_path`:

Request:

```
POST /v2/entity
```

Request Body:

```
{
 "typeName" : "hdfs_path",
 "attributes" : {
  "path" : "hdfs://vimal-fenton-4-1.openstacklocal:8020/apps/hive/warehouse/
result",
  "qualifiedName" : "result_directory@clusterName",
  "name" : "spark_transformaion_result"
 }
}
```

Response:

```
          {
            "typeName": "hdfs_path",
            "attributes": {
              "clusterName": null,
              "createTime": null,
              "qualifiedName": "result_directory@clusterName",
              "modifiedTime": null,
              "posixPermissions": null,
              "fileSize": 0,
              "numberOfReplicas": 0,
              "description": null,
              "isFile": false,
              "name": "spark_transformaion_result",
              "owner": null,
              "path": "hdfs://vimal-fenton-4-1.openstacklocal:8020/apps/hive/
warehouse/result",
              "group": null,
              "extendedAttributes": null,
              "isSymlink": false
            },
            "guid": "17a657d3-e72a-4501-8bde-a2843bed84ac"
          }
```

Next we will register another `spark_transformation` entity:

Request:

```
POST /v2/entity
```

Request Body:

```
{
  "typeName": "spark_transformation",
  "attributes" : {
    "qualifiedName" : "spark_process_id_32454545",
    "name" : "spark_process_2",
  "parallelism" : "10",
  "inputs" : [
      {
        "typeName": "spark_dataframe",
        "attributes": {
          "source": "hdfs://vimal-fenton-4-1.openstacklocal:8020/apps/hive/
warehouse/source_2",
          "description": null,
          "qualifiedName": "destination_dataframe@clusterName",
          "name": "destination_dataframe",
          "owner": null,
          "destination": "hdfs://vimal-fenton-4-1.openstacklocal:8020/apps/
hive/warehouse/destination_2"
        },
        "guid": "86d5cb10-538a-40cd-80c9-22fc363224d0"
      }
    ],
    "outputs" : [
          {
            "typeName": "hdfs_path",
            "attributes": {
              "clusterName": null,
```

```
              "createTime": null,
              "qualifiedName": "result_directory@clusterName",
              "modifiedTime": null,
              "posixPermissions": null,
              "fileSize": 0,
              "numberOfReplicas": 0,
              "description": null,
              "isFile": false,
              "name": "spark_transformaion_result",
              "owner": null,
              "path": "hdfs://vimal-fenton-4-1.openstacklocal:8020/apps/hive/
warehouse/result",
              "group": null,
              "extendedAttributes": null,
              "isSymlink": false
            },
            "guid": "17a657d3-e72a-4501-8bde-a2843bed84ac"
          }
  ]
  },
  "classifications": [

  ]
}
```

Response:

```
{
  "typeName": "spark_transformation",
  "attributes": {
    "inputs": [
      {
        "typeName": "DataSet",
        "guid": "86d5cb10-538a-40cd-80c9-22fc363224d0"
      }
    ],
    "description": null,
    "qualifiedName": "spark_process_id_32454545",
    "name": "spark_process_2",
    "owner": null,
    "outputs": [
      {
        "typeName": "DataSet",
        "guid": "17a657d3-e72a-4501-8bde-a2843bed84ac"
      }
    ],
    "parallelism": 10
  },
  "guid": "7cedee1c-c75a-4be8-b383-6079013ee095"
}
```

Now that we have the following lineage relationship:

source_dataframe@clusterName # spark_process_id_24343324 # destination_dataframe@clusterName # spark_process_id_32454545 # result_directory@clusterName

We can experiment with the query parameters in the lineage API. The GUID corresponding to `destination_dataframe@clusterName` is `86d5cb10-538a-40cd-80c9-22fc363224d0`.

Example Request:

```
GET v2/lineage/86d5cb10-538a-40cd-80c9-22fc363224d0
```

Example Response:


```
{
  "baseEntityGuid": "86d5cb10-538a-40cd-80c9-22fc363224d0",
  "lineageDirection": "BOTH",
  "lineageDepth": 3,
  "guidEntityMap": {
    "7cedee1c-c75a-4be8-b383-6079013ee095": {
      "typeName": "spark_transformation",
      "guid": "7cedee1c-c75a-4be8-b383-6079013ee095",
      "status": "STATUS_ACTIVE",
      "displayText": "spark_process_id_32454545"
    },
    "58a3ee5d-827f-4aa4-98e1-ccebd5851c76": {
      "typeName": "spark_transformation",
      "guid": "58a3ee5d-827f-4aa4-98e1-ccebd5851c76",
      "status": "STATUS_ACTIVE",
      "displayText": "spark_process_id_24343324"
    },
    "c94d8450-6d59-4cd1-8732-863286387c7d": {
      "typeName": "spark_dataframe",
      "guid": "c94d8450-6d59-4cd1-8732-863286387c7d",
      "status": "STATUS_ACTIVE",
      "displayText": "source_dataframe@clusterName"
    },
    "17a657d3-e72a-4501-8bde-a2843bed84ac": {
      "typeName": "hdfs_path",
      "guid": "17a657d3-e72a-4501-8bde-a2843bed84ac",
      "status": "STATUS_ACTIVE",
      "displayText": "result_directory@clusterName"
    },
    "86d5cb10-538a-40cd-80c9-22fc363224d0": {
      "typeName": "spark_dataframe",
      "guid": "86d5cb10-538a-40cd-80c9-22fc363224d0",
      "status": "STATUS_ACTIVE",
      "displayText": "destination_dataframe@clusterName"
    }
  },
  "relations": [
    {
      "fromEntityId": "c94d8450-6d59-4cd1-8732-863286387c7d",
      "toEntityId": "58a3ee5d-827f-4aa4-98e1-ccebd5851c76"
    },
    {
      "fromEntityId": "86d5cb10-538a-40cd-80c9-22fc363224d0",
      "toEntityId": "7cedee1c-c75a-4be8-b383-6079013ee095"
    },
    {
      "fromEntityId": "58a3ee5d-827f-4aa4-98e1-ccebd5851c76",
      "toEntityId": "86d5cb10-538a-40cd-80c9-22fc363224d0"
    },
    {
      "fromEntityId": "7cedee1c-c75a-4be8-b383-6079013ee095",
      "toEntityId": "17a657d3-e72a-4501-8bde-a2843bed84ac"
    }
  ]
```

```
}
```

To find the INPUT entities from which `destination_dataframe@clusterName` is derived, we can specify the query parameter direction in the request:

Example Request:

```
GET v2/lineage/86d5cb10-538a-40cd-80c9-22fc363224d0?direction=INPUT
```

Example Response:

```
{
  "baseEntityGuid": "86d5cb10-538a-40cd-80c9-22fc363224d0",
  "lineageDirection": "INPUT",
  "lineageDepth": 3,
  "guidEntityMap": {
    "58a3ee5d-827f-4aa4-98e1-ccebd5851c76": {
      "typeName": "spark_transformation",
      "guid": "58a3ee5d-827f-4aa4-98e1-ccebd5851c76",
      "status": "STATUS_ACTIVE",
      "displayText": "spark_process_id_24343324"
    },
    "c94d8450-6d59-4cd1-8732-863286387c7d": {
      "typeName": "spark_dataframe",
      "guid": "c94d8450-6d59-4cd1-8732-863286387c7d",
      "status": "STATUS_ACTIVE",
      "displayText": "source_dataframe@clusterName"
    },
    "86d5cb10-538a-40cd-80c9-22fc363224d0": {
      "typeName": "spark_dataframe",
      "guid": "86d5cb10-538a-40cd-80c9-22fc363224d0",
      "status": "STATUS_ACTIVE",
      "displayText": "destination_dataframe@clusterName"
    }
  },
  "relations": [
    {
      "fromEntityId": "c94d8450-6d59-4cd1-8732-863286387c7d",
      "toEntityId": "58a3ee5d-827f-4aa4-98e1-ccebd5851c76"
    },
    {
      "fromEntityId": "58a3ee5d-827f-4aa4-98e1-ccebd5851c76",
      "toEntityId": "86d5cb10-538a-40cd-80c9-22fc363224d0"
    }
  ]
}
```

Similarly, we can specify the parameter depth to fetch entities within a depth limit.

6.3. Cataloging Atlas Metadata: Traits and Business Taxonomy

As discussed previously, metadata is added to Atlas as entities (instances) of types (model definitions). Typically, the models are defined by whoever best understands the metadata. For example, the Hive data types are typically defined by someone with a good understanding of Hive types.

Data discovery and governance can be enhanced when metadata use cases are expanded to include business terminology and processes, rather than just technical metadata. This business cataloging can be performed by data stewards or data scientists who act as a bridge between technical and business metadata.

Business metadata can be cataloged using a common business terminology, even if the metadata may not be closely related from a technical standpoint. Using a common business taxonomy enables you to build applications that apply the same governance policies to similar metadata irrespective of their sources of origin. Also, Atlas search capabilities allow you to easily find similar business metadata.

For example, in the finance industry, all data sets that deal with “credit” as a concept can be cataloged as such irrespective of whether they originate from Hive, HBase, or any other data stores. Once similarly cataloged, credit-related policies can be applied to all data assets (entities) cataloged with this concept.

Atlas provides two ways of cataloging metadata: Traits and Business Taxonomy. Loosely speaking, while Traits represent a more free-form way of cataloging or annotating metadata (think of how tags are added to documents in a document management system), Business Taxonomy should relate to a more clearly defined and controlled vocabulary that has specific meanings in a domain, and that is uniformly understood within a certain context.

Note

In the Atlas UI and elsewhere, traits are sometimes referred to as "tags". This document will use the term "traits", as that is the terminology used in the Atlas API.

6.3.1. Atlas Traits

Traits were introduced in the Atlas Types section as one of the composite metatypes, along with Classes and Structs. Traits share similarities with these other composite metatypes in that they define a Type and have a uniquely identifiable name in the type system. They can also have a set of attributes, although these attributes can only be of native types.

Like Classes, Traits can extend from other super traits, and thus inherit attributes defined in those super traits. However, unlike Classes, trait instances are not entities. They do not have a uniquely-identifiable GUID, and consequently they cannot be referenced from attributes in other types. Therefore, the way in which a trait instance is defined and used is different than the way in which an entity is defined and used.

Trait instances also have one other special significance in Atlas. They can be associated with any entity in Atlas without prior declaration of this fact in the Type definition of the entity. Note that, in contrast, defining an entity reference in a type must declared a priori (for example, HBase table references to an HBase namespace should be declared up-front). The Atlas type system recognizes traits, and includes specific APIs that can be used to associate traits with entities.

Note

Atlas Traits cannot be deleted.

6.3.1.1. Create Traits

Description:

Because Traits are Atlas Types, the same APIs used to create Types are used to create Traits, except that Attribute definitions cannot refer to non-native metatypes.

Request:

```
POST http://<atlas-server-host:port>/api/atlas/types
```

Request Body:

The body for this request is the same structure as the `TypesDef` structure that is defined in Important Atlas API Data Types. The Traits should be defined under the `traitTypes` attribute.

Response:

The response is the same as the response for a Type Definition request, and contains the names of the defined Traits.

Example:

Using our running example, we will define two Traits:

- `PublicData` – Any metadata marked with this Trait indicates that this data was collected from publicly available sources. Therefore, any policies applicable to publicly collected data can be applied to this data.
- `Retainable` – This Trait indicates that any metadata associated with this Trait should be retained for a period of time. The time period is maintained in an a `retentionPeriod` attribute, which is the duration in days.

Example Request Body:

```
{
  "enumTypes":[],
  "structTypes":[],
  "traitTypes":[
    {
      "superTypes":[],
      "hierarchicalMetaTypeName":"org.apache.atlas.typesystem.types.
TraitType",
      "typeName":"PublicData",
      "typeDescription":null,
      "attributeDefinitions":[]
    },
    {
      "superTypes":[],
      "hierarchicalMetaTypeName":"org.apache.atlas.typesystem.types.
TraitType",
      "typeName":"Retainable",
      "typeDescription":null,
      "attributeDefinitions":[
        {
          "name":"retentionPeriod",
          "dataTypeName":"int",
```

```
          "multiplicity":"required",
          "isComposite":false,
          "isUnique":false,
          "isIndexable":true,
          "reverseAttributeName":null
        }
      ]
    }
  ],
  "classTypes":[]
}
```

Note that the `traitTypes` attribute contains the defined Traits. The rest of the metatypes – structs, enums, and classes – are empty. The `retentionPeriod` attribute is defined as an `int` in the `Retainable` Trait.

Example Response:

```
{
  "requestId": "qtp221036634-18 - 59cbed8a-3637-496f-8b40-80ec829ce493",
  "types": [
    {
      "name": "Retainable"
    },
    {
      "name": "PublicData"
    }
  ]
}
```

6.3.1.2. List Traits

Description:

Because Traits are a specific metatype (like Classes), the same API used to list a specific metatype can be used to list Traits.

Request:

```
GET http://<atlas-server-host:port>/api/atlas/types?type=TRAIT
```

Response:

The response is a list of Trait names.

Example Response:

```
{
  "results": [
    "Retainable",
    "PublicData"
  ],
  "count": 2,
  "requestId": "qtp221036634-16 - 423d9f90-79ae-4b29-b9bf-2d2a1d05c2bd"
}
```

6.3.1.3. Retrieve a Trait

Description:

Because Traits are a specific metatype (like Classes), the same API used to retrieve a specific metatype can be used to retrieve a Trait.

Request:

```
GET http://<atlas-server-host:port>/api/atlas/types/{trait_name}
```

Response:

The response for this request is the same structure as the `TypesDef` structure that is defined in Important Atlas API Data Types. The `traitTypes` attribute contains the type definition of the Trait specified in the request.

Example Request:

```
GET http://<atlas-server-host:port>/api/atlas/types/Retainable
```

Example Response:

```
{
  "typeName": "Retainable",
  "definition": {
    "enumTypes": [],
    "structTypes": [],
    "traitTypes": [
      {
        "superTypes": [],
        "hierarchicalMetaTypeName": "org.apache.atlas.typesystem.types.
TraitType",
        "typeName": "Retainable",
        "typeDescription": null,
        "attributeDefinitions": [
          {
            "name": "retentionPeriod",
            "dataTypeName": "int",
            "multiplicity": "required",
            "isComposite": false,
            "isUnique": false,
            "isIndexable": true,
            "reverseAttributeName": null
          }
        ]
      }
    ],
    "classTypes": []
  },
  "requestId": "qtp221036634-204 - b9f43388-49d8-452b-8901-d05581d2b442"
}
```

6.3.1.4. Associate Trait Instances with Entities

Description:

To catalog entities using Traits, we must associate Trait instances with entities.

Request:

```
POST http://<atlas-server-host:port>/api/atlas/entities/{entity_guid}/traits
```

Request Body:

The request body is a Trait `InstanceDefinition` structure that is defined in Important Atlas API Data Types.

Response:

No data is returned in the response. A 201 status code indicates success.

Example:

In this example, we annotate our webtable (GUID f4019a65-8948-46f1-afcf-545baa2df99f) with `PublicData` Trait to indicate that it is a data asset that is created by crawling public sites. We also set a `Retainable` Trait on the column family contents (GUID 9e6308c6-1006-48f8-95a8-a605968e64d2) with a retention period of 100 days.

The following requests would be sent:

Example Request:

```
POST http://<atlas-server-host:port>/api/atlas/entities/f4019a65-8948-46f1-
afcf-545baa2df99f/traits
```

Example Request Body:

```
{
  "jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization
$_Struct",
  "typeName":"PublicData",
  "values":{

  }
}
```

Example Request:

```
POST http://<atlas-server-host:port>/api/atlas/entities/
9e6308c6-1006-48f8-95a8-a605968e64d2/traits
```

Example Request Body:

```
{
  "jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization
$_Struct",
  "typeName":"Retainable",
  "values":{
    "retentionPeriod":"100"
  }
}
```

6.3.1.5. Read Trait Instances Associated with Entities

Description:

When Trait instances are associated with entities according to structure that is defined in Important Atlas API Data Types, the `EntityDefinition` includes the `traitNames` and `traits` attributes. A request for an `EntityDefinition` returns a response that includes the `traitNames` and `trait` values.

Request:

```
GET http://<atlas-server-host:port>/api/atlas/entities/{entity_guid}
```

Example Request:

This is a request for an HBase table `EntityDefinition`.

```
GET http://<atlas-server-host:port>/api/atlas/entities/f4019a65-8948-46f1-
afcf-545baa2df99f
```

Example Response:

For the sake of brevity, only the `traitNames` and `trait` values are shown below.

```
{
    ...
    "typeName": "hbase_table",
    "values": {
      ...
      "columnFamilies": [
        {
          "typeName": "hbase_column_family",
          "values": {
            "qualifiedName": "default.webtable.contents@cluster2",
          },
          "traitNames": [
            "Retainable"
          ],
          "traits": {
            "Retainable": {
              "jsonClass": "org.apache.atlas.typesystem.json.
InstanceSerialization$_Struct",
              "typeName": "Retainable",
              "values": {
                "retentionPeriod": 100
              }
            }
          }
        }
      ],
      "qualifiedName": "default.webtable@cluster2",
      ...
       "traitNames": [
      "PublicData"
    ],
    "traits": {
      "PublicData": {
        "jsonClass": "org.apache.atlas.typesystem.json.InstanceSerialization
$_Struct",
        "typeName": "PublicData",
        "values": {}
      }
    }
  }
}
```

6.3.1.6. Disassociate Trait Instances Associated with Entities

Description:

This is a simple DELETE operation.

Request:

```
DELETE http://<atlas-server-host:port>/api/atlas/entities/{entity_guid}/
traits/{trait_name}
```

Response:

No data is returned.

Example Request:

```
DELETE http://<atlas-server-host:port>/api/atlas/entities/f4019a65-8948-46f1-
afcf-545baa2df99f/traits/PublicData
```

6.3.2. Atlas Business Taxonomy

The Atlas Business Taxonomy is a hierarchical collection of "Terms" objects. Each Term has two important attributes: name and description. Terms can be defined under a predefined Taxonomy object, or under another Term. The predefined Taxonomy object is referred to as a “Catalog”.

Once created, a Term can be associated with any entity in Atlas. This is similar to how Trait instances can be associated with any entity. Unlike Traits, Terms can be deleted.

6.3.2.1. Create a Term

Description:

To create a Term, you must provide a Term name. You can also provide an optional description.

Request:

```
POST http://<atlas-server-host:port>/api/atlas/v1/taxonomies/Catalog/terms/
{term_name}
```

Request Body:

The body for this request contains a single element: `{"description":string}`.

Response:

The response contains the following map:

```
{"href":<url_for_created_term_resource>,"Status":"201"}
```

The "href" attribute contains the resource URL for the new Term.

Example Request:

```
POST http://<atlas-server-host:port>/api/atlas/v1/taxonomies/Catalog/terms/
term1
```

Example Request Body:

```
{"description":"This is term1"}
```

Example Response:

```
{
"href":"http://localhost:21000/api/atlas/v1/taxonomies/Catalog/terms/term1",
"Status":"201"
}
```

6.3.2.2. Create a Term Under Another Term

Description:

To create a Term under another Term, you must first determine the URL of the created Term. This is a recursive URL that takes the following form:

```
http://<atlas-server-host:port>/api/atlas/v1/taxonomies/Catalog/terms/
{term_name}/terms/.../terms/{term_name}
```

Then the request takes the form:

```
POST {resource_url_of_parent_term}/terms/{term_name}
```

Example Request:

```
POST http://<atlas-server-host:port>/api/atlas/v1/taxonomies/Catalog/terms/
term1
```

Example Request Body:

```
{"description":"This is term1"}
```

Example Response:

```
{
"href":"http://localhost:21000/api/atlas/v1/taxonomies/Catalog/terms/term1",
"Status":"201"
}
```

6.3.2.3. Retrieve a Term Definition

Description:

Retrieve the definition for a specific Term.

Request:

```
GET http://<atlas-server-host:port>/api/atlas/v1/taxonomies/Catalog/terms/
{term_name}/terms/.../terms/{term_name}
```

Response:

The response has the following structure:

```
[
  {
    "href": url_of_term,
    "name": fully_qualified_name_of_term,
    "description": description_of_term,
    "available_as_tag": true,
    "creation_time": timestamp,
    "hierarchy": {
      ...
    },
```

```
    "terms": {
      "href": url_of_terms_under_this_term
    }
  }
]
```

Response field descriptions:

- `href` – The resource URL for the Term.
- `name` – The fully qualified name of this Term. By fully qualified, we mean the entire path starting from the Business Taxonomy name, along with all intermediate parent terms above the Term. These components are separated by a "." character.
- `description` – The description provided when the Term was created.
- System-defined properties such as `creation_time` are also included.
- `terms` – Contains a single `href` element with the URL to use to retrieve the Terms under this Term.

Example Request:

```
GET http://<atlas-server-host:port>/api/atlas/v1/taxonomies/Catalog/terms/
term1/terms/term11
```

Example Response:

```
[
  {
    "href": "http://localhost:21000/api/atlas/v1/taxonomies/Catalog/terms/
term1/terms/term11",
    "name": "Catalog.term1.term11",
    "description": "This is term11",
    "available_as_tag": true,
    "creation_time": "2016-06-20:03:14:42",
    "hierarchy": {
      "path": "/term1",
      "short_name": "term11",
      "taxonomy": "d"
    },
    "terms": {
      "href": "http://localhost:21000/api/atlas/v1/taxonomies/Catalog/terms/
term1/terms/term11/terms"
    }
  }
]
```

6.3.2.4. List All Terms

Description:

List all Terms in the Catalog Taxonomy hierarchy.

Request:

```
GET http://<atlas-server-host:port>/api/atlas/v1/taxonomies/Catalog/terms
```

Response:

The response is an array of results:

```
[
  {
    "href": url_of_term,
    "name": fully_qualified_name_of_term,
    "description": description_of_term
  },
  ...
]
```

Each element in the array is a descendant of the Catalog hierarchy of business Terms.

Response field descriptions:

- `href` – The resource URL for the Term.
- `name` – The fully qualified name of this Term. By fully qualified, we mean the entire path starting from the Business Taxonomy name, along with all intermediate parent terms above the Term. These components are separated by a "." character.
- `description` – The description provided when the Term was created.

Example Request:

```
GET http://<atlas-server-host:port>/api/atlas/v1/taxonomies/Catalog/terms
```

Example Response:

```
[
  {
    "href": "http://localhost:21000/api/atlas/v1/taxonomies/Catalog/terms/
term1",
    "name": "Catalog.term1",
    "description": "This is Term1"
  },
  {
    "href": "http://localhost:21000/api/atlas/v1/taxonomies/Catalog/terms/
term1/terms/term11",
    "name": "Catalog.term1.term11",
    "description": "This is term11"
  },
  {
    "href": "http://localhost:21000/api/atlas/v1/taxonomies/Catalog/terms/
term1/terms/term11/terms/term111",
    "name": "Catalog.term1.term11.term111"
  },
  {
    "href": "http://localhost:21000/api/atlas/v1/taxonomies/Catalog/terms/
term1/terms/term12",
    "name": "Catalog.term1.term12"
  }
]
```

6.3.2.5. List All Terms Under a Term

Description:

List all Terms under a specified Term.

Request:

```
GET http://<atlas-server-host:port>/api/atlas/v1/taxonomies/Catalog/terms/
{term_name}/terms/.../terms/{term_name}/terms
```

Response:

The response is an array of results:

```
[
  {
    "href": url_of_term,
    "name": fully_qualified_name_of_term,
    "description": description_of_term
  },
  ...
]
```

Each element in the array is a descendant of the Term specified in the request.

Response field descriptions:

- `href` – The resource URL for the Term.
- `name` – The fully qualified name of this Term. By fully qualified, we mean the entire path starting from the Business Taxonomy name, along with all intermediate parent terms above the Term. These components are separated by a "." character.
- `description` – The description provided when the Term was created.

Example Request:

```
GET http://<atlas-server-host:port>/api/atlas/v1/taxonomies/Catalog/terms/
term1/terms/term11/terms
```

Example Response:

```
[
  {
    "href": "http://localhost:21000/api/atlas/v1/taxonomies/Catalog/terms/
term1/terms/term11/terms/term111",
    "name": "Catalog.term1.term11.term111"
  },
]
```

6.3.2.6. Associate a Term with an Entity

Description:

To associate a Term with an entity, you must provide the entity GUID and the fully qualified name of the Term.

Request:

```
POST http://<atlas-server-host:port>/api/atlas/v1/entities/{entity_guid}/tags/
{fully_qualified_name_of_term}
```

The request body is an empty map.

Response:

```
{"href":url_for_created_resource,"status":"201"}
```

Example Request:

```
POST http://<atlas-server-host:port>/api/atlas/v1/entities/f4019a65-8948-46f1-
afcf-545baa2df99f/tags/Catalog.term1.term12
```

6.3.2.7. Disassociate a Term from an Entity

Description:

To disassociate a Term with an entity, you must provide the entity GUID and the fully qualified name of the Term.

Request:

```
DELETE http://<atlas-server-host:port>/api/atlas/v1/entities/{entity_guid}/
tags/{fully_qualified_name_of_term}
```

Response:

```
{"href":url_for_deleted_resource,"status":"200"}
```

Example Request:

```
DELETE http://<atlas-server-host:port>/api/atlas/v1/entities/
f4019a65-8948-46f1-afcf-545baa2df99f/tags/Catalog.term1.term12
```

6.3.2.8. Delete a Term

Description:

Deleting a Term removes it from the Business Catalog Taxonomy and also removes all entity associations with the Term. In addition, all child Terms under the deleted Term in the Catalog hierarchy are also deleted. Note that depending on the depth of the hierarchy and the number of associations for the term and its sub-terms, this could be an expensive operation in terms of system resources.

Request:

```
DELETE http://<atlas-server-host:port>/api/atlas/v1/taxonomies/Catalog/terms/
{term_name}/terms/.../terms/{term_name}
```

Response:

```
{"href":url_for_deleted_resource,"status":"200"}
```

Example Request:

```
DELETE http://<atlas-server-host:port>/api/atlas/v1/taxonomies/Catalog/terms/
term1
```

6.3.2.9. Update a Term

Description:

Currently, the only Term attribute that can be updated is the description. Updating a Term description also updates this property in all associated entities.

Request:

```
PUT http://<atlas-server-host:port>/api/atlas/v1/taxonomies/Catalog/terms/
{term_name}/terms/.../terms/{term_name}
```

Request Body:

```
{"description":"updated_description"}
```

Response:

```
{"href":url_for_updated_resource,"status":"200"}
```

Example Request:

```
PUT http://<atlas-server-host:port>/api/atlas/v1/taxonomies/Catalog/terms/
term2
```

6.4. Discovering Metadata: The Atlas Search API

In previous sections, we saw how to add metadata to Atlas, and how to catalog this metadata using traits and Business Catalog terms. We also discussed how to use the Atlas API to retrieve a particular metadata entity using its GUID or a unique attribute.

As more and more metadata is added to Atlas, it becomes difficult or impossible to remember all of the unique attribute values. Atlas provides the following methods to search metadata:

- **DSL Search** – Atlas DSL (Domain-Specific Language) is a SQL-like query language that enables you to search metadata using complex queries based on type and attribute names. This DSL query can be passed to a Search API. Internally, the query is translated to a Graph look-up query using Gremlin and fired against the metadata store. The results are then translated into entity and type system objects and returned.

  The DSL search is useful if you are aware of the specific metadata model (type names, attributes, etc.) of the entities you would like to retrieve. This generally results in very specific search queries and relevant results. Using the type system API (listing types, retrieving a type definition), you can obtain the model of an entity, and then use the Atlas DSL to search for entities of that type.

- **Full-text Search** – When entities are added to Atlas, a search indexing system (Solr) indexes their attribute values. These indexed attributes can be used to retrieve entities using a full-text search. The Atlas Search API can be used for both DSL and full-text search.

  Full-text search is useful if you are not familiar with the metadata model, or if you would like to query across different models (types). For example, full-text search can be used to find all assets related to customer data, irrespective of the storage used (Hive, HBase, etc.). However, because full text search is based on an index that is not aware of type or model information, the results are likely to be broader than with a DSL search.

- **Catalog-based Search** – Atlas enables data stewards to make data more discoverable by annotating metadata entities with traits (also referred to as "tags") and Business Catalog terms. DSL search enables you to search metadata based on specific traits and terms.

This provides highly relevant search results, provided that the metadata is annotated correctly.

6.4.1. DSL Search

DSL enables you to search using Apache Atlas DSL. In this section, we proved an example-driven approach to help you understand DSL syntax and capabilities. The syntax used here will not be as exact as the grammar described on the Apache Atlas DSL Search page, but these simplified examples should help explain the key concepts.

You can use the Atlas web UI to test these examples. Select **Search > DSL**, then enter the query in the **Search For** box.

6.4.1.1. Discovering Entities Using Attribute Values

Search Query Format:

```
type_name where attribute_name OP attribute_value
```

Where:

- `type_name` is the name of a predefined type.
- `attribute_name` is the name of an attribute in that type. This does not need to be a unique attribute (unlike in "Retrieve an Entity Definition Using a Unique Attribute" shown previously in the Entities API).
- OP is an operator: =, !=, <, >, <=, >=
- `attribute_value` is the value of an attribute.

The search results are entire entity definitions that match the search criteria.

Search Query Examples:

- `hbase_table where name = 'webtable'`
- `hbase_column_family where name != 'contents'`
- `hbase_column_family where versions > 1`
- `hbase_column_family where blockSize < 1000`

6.4.1.2. Discovering Entities Using Combinations of Attribute Values

Search Query Format:

```
type_name where attribute_name OP attribute_value AND_OR_OP attribute_name OP
 attribute_value [AND|OR …]
```

Where:

- `type_name` is the name of a predefined type.
- `attribute_name` is the name of an attribute in that type. This does not need to be a unique attribute (unlike in "Retrieve an Entity Definition Using a Unique Attribute" shown previously in the Entities API).

- OP is an operator: =, !=, <, >, <=, >=
- AND_OR_OP is and or or.
- attribute_value is the value of an attribute.

Note the following:

- It is possible to provide any number of expressions of the form attribute_name OP attribute_value combining them with an and or an or.
- It is also possible to impose an ordering of evaluation by enclosing the expressions within parentheses, for example:

  ```
  type_name where attribute_name OP attribute_value AND_OR_OP
  (attribute_name OP attribute_value AND_OR_OP attribute_name OP
  attribute_value)
  ```

The search results are entire entity definitions that match the search criteria.

Search Query Examples:

- `hbase_column_family where blockSize < 1000 and versions >= 2`
- `hbase_column_family where compression != 'lzo' and versions > 1 and blockSize > 1000`
- `hbase_column_family where compression = 'lzo' and (versions > 1 or blockSize > 1000)`

6.4.1.3. Selecting Native Attributes in Searches

As described above, search query results include the entire entity definitions. You can also use a SELECT clause in the search query to return specific attributes.

Search Query Format:

```
search_expression select attribute_name [, attribute_name…]
```

Where:

- search_expression is one of the previously described search expressions.
- attribute_name [list] specifies the attributes to return in the search results.

Search Query Example:

- `hbase_table where name='webtable' select name, qualifiedName, isEnabled`

6.4.1.4. Selecting References in Searches

The previous section showed how we can select any attributes which are of native types. But there are also more complex attribute types, such as collections, references to other entities, etc. For example, hbase_tables contain columnFamilies which are references to entities of type hbase_column_family. To help address this issue, DSL allows search queries to be combined as follows:

Search Query Format:

```
search_expression, reference_attribute_name
```

Where:

- `search_expression` is one of the previously described search expressions.
- `reference_attribute_name` is an attribute name in the entity being selected in `search_expression` that contains references to other attributes.

One variation is where the `reference_attribute_name` can be expanded to only select specific attributes of the `reference_attribute` type:

```
search_expression, reference_attribute_name
 select reference_attribute_type_attribute_name [,
 reference_attribute_type_attribute_name]
```

Another variation is where the `reference_attribute_name` can be filtered to include only those references which satisfy some predicate:

```
search_expression, reference_attribute_name
 where reference_attribute_type_attribute_name OP
 reference_attribute_type_attribute_value
```

Search Query Examples:

- `hbase_table, columnFamilies`
- `hbase_column_family where name='anchor', columns`
- `hbase_columns_family where name=’anchor’, columns select name, type`
- `hbase_column_family, columns where type='byte[]'`

6.4.2. DSL Search API

6.4.2.1. DSL Search that Returns Entities

Request:

```
GET http://<atlas-server-host:port>/api/atlas/discovery/search/dsl?query=
{dsl_query_string}
```

The `dsl_query_string` should be encoded using standard URL encoding criteria.

Response:

```
{
 "requestId": string,
 "query": dsl_query_string,
 "queryType": "dsl",
 "count": int,
 "results": array_of_search_results,
 "dataType": TypesDef struct
}
```

Response field descriptions:

- `query` – The unencoded version of the dsl_query_string passed in the request.
- `queryType` – The query type (`dsl`).
- `count` – The number of results returned.
- `results` – An array of search results. Each search result follows the `EntityDefinition` structure defined in Important Atlas API Datatypes, with the following differences:
  - The `typeName` attribute is changed to `$typeName$`
  - The `id` attribute is changed to `$id$`
- `dataType` – A partial `TypesDef` Struct (defined in Important Atlas API Datatypes) that describes the search result type. The attribute definitions of the `TypesDef` are not complete.

Example Request:

The following example searches for an `hbase_column_family` where `type='byte[]'`.

```
GET http://<atlas-server-host:port>/api/atlas/discovery/search/dsl?
hbase_column_family%2C+columns+where+type%3D%27byte%5B%5D%27
```

Example Response:

```
{
 "requestId": "qtp221036634-903 - 98091bba-9ea1-4482-9355-4dca396d9657",
 "query": "hbase_column_family, columns where type='byte[]'",
 "queryType": "dsl",
 "count": 2,
 "results": [{
  "$typeName$": "hbase_column",
  "$id$": {
   "id": "fc711cee-185f-4f09-a2f9-a96e0173f51b",
   "$typeName$": "hbase_column",
   "version": 0,
   "state": "ACTIVE"
  },
  "qualifiedName": "default.webtable.contents.html@cluster2",
  "type": "byte[]",
  "owner": "crawler",
  "description": null,
  "name": "html"
 }, {
  "$typeName$": "hbase_column",
  "$id$": {
   "id": "3a76cb82-544c-49d8-9f8c-eb12bcbc4584",
   "$typeName$": "hbase_column",
   "version": 0,
   "state": "ACTIVE"
  },
  "qualifiedName": "default.webtable.contents.html@cluster1",
  "type": "byte[]",
```

```
  "owner": "crawler",
  "description": null,
  "name": "html"
 }],
 "dataType": {
  "superTypes": ["Referenceable", "Asset"],
  "hierarchicalMetaTypeName": "org.apache.atlas.typesystem.types.ClassType",
  "typeName": "hbase_column",
  "typeDescription": null,
  "attributeDefinitions": [{
   "name": "type",
   "dataTypeName": "string",
   "multiplicity": {
    "lower": 1,
    "upper": 1,
    "isUnique": false
   },
   "isComposite": false,
   "isUnique": false,
   "isIndexable": true,
   "reverseAttributeName": null
  }]
 }
}
```

Because the results for the query are `hbase_column` instances, we see that the entity definition is an `hbase_column`.

6.4.2.2. DSL Search that Returns Specific Attributes

Request:

```
GET http://<atlas-server-host:port>/api/atlas/discovery/search/dsl?query=
{dsl_query_string}
```

The `dsl_query_string` should be encoded using standard URL encoding criteria.

Response:

```
{
 "requestId": string,
 "query": dsl_query_string,
 "queryType": "dsl",
 "count": int,
 "results": [
   {
    “$typeName$”: “__tempQueryResultStruct..”,
    “selected_attribute_1”: “value”,
    ...
   },
   ...
],
 "dataType": {
    “typeName”: “__tempQueryResultStruct..”,
    "typeDescription": null,
    "attributeDefinitions": array of attributeDefinitions only for selected
 attributes
   }
}
```

Response field descriptions:

- `query` – The unencoded version of the dsl_query_string passed in the request.
- `queryType` – The query type (`dsl`).
- `count` – The number of results returned.
- `results` – An array of search results. Each search result follows the `EntityDefinition` structure defined in Important Atlas API Datatypes, with the following differences:
  - The `typeName` attribute is changed to `$typeName$`
  - The `id` attribute is changed to `$id$`
  - The result elements will contain only the attribute names and values specified in the query `select` clause.
  - The result elements will not include the GUID or any attribute that is not specified in the query `select` clause.
- `dataType` – A partial `TypesDef` Struct (defined in Important Atlas API Datatypes) that describes the search result type. The attribute definitions of the `TypesDef` are not complete. The `dataType` includes only the attribute definitions for the attributes specified in the query `select` clause.

Example Request:

The following example searches for an `hbase_table` where `name='webtable' select name, qualifiedName, isEnabled`.

```
GET http://<atlas-server-host:port>/api/atlas/discovery/search/dsl?query=
hbase_table+where+name%3D%27webtable%27+select+name%2C+qualifiedName%2C
+isEnabled
```

Example Response:

```
{
 "requestId": "qtp221036634-963 - e8d615ee-1604-44db-8344-579c2fc3bbfe",
 "query": "hbase_table where name='webtable' select name, qualifiedName,
 isEnabled",
 "queryType": "dsl",
 "count": 2,
 "results": [{
  "$typeName$": "__tempQueryResultStruct89",
  "qualifiedName": "default.webtable@cluster2",
  "isEnabled": true,
  "name": "webtable"
 }, {
  "$typeName$": "__tempQueryResultStruct89",
  "qualifiedName": "default.webtable@cluster1",
  "isEnabled": false,
  "name": "webtable"
 }],
 "dataType": {
  "typeName": "__tempQueryResultStruct89",
  "typeDescription": null,
```

```
  "attributeDefinitions": [{
   "name": "name",
   "dataTypeName": "string",
   "multiplicity": {
    "lower": 0,
    "upper": 1,
    "isUnique": false
   },
   "isComposite": false,
   "isUnique": false,
   "isIndexable": true,
   "reverseAttributeName": null
  }, {
   "name": "qualifiedName",
   "dataTypeName": "string",
   "multiplicity": {
    "lower": 0,
    "upper": 1,
    "isUnique": false
   },
   "isComposite": false,
   "isUnique": false,
   "isIndexable": true,
   "reverseAttributeName": null
  }, {
   "name": "isEnabled",
   "dataTypeName": "boolean",
   "multiplicity": {
    "lower": 0,
    "upper": 1,
    "isUnique": false
   },
   "isComposite": false,
   "isUnique": false,
   "isIndexable": true,
   "reverseAttributeName": null
  }]
 }
}
```

Note that the response contains only the attributes `name`, `qualifiedName` and `isEnabled`. Also note that only these three attribute definitions are included.

6.4.3. Full-text Search API

As described previously in the introduction of Discovering Metadata, Atlas indexes attribute values as metadata entities are added. The index maps the text value to the entity GUID that the attribute belongs to, which enables lookup queries using simple text strings. These strings can be attribute values of any Atlas entities.

Request:

```
GET http://<atlas-server-host:port>/api/atlas/discovery/search/fulltext?query=
{query_string}
```

The `query_string` should be encoded using standard URL encoding criteria.

Response:

```
{
 "requestId": string,
 "query": query_string,
 "queryType": "full-text",
 "count": int,
 "results": [{
  "guid": guid_of_matching_entity,
  "typeName": typename_of_matching_entity,
  "score": relevance_score in indexing
 }, ...]
}
```

Response field descriptions:

- `query` – The unencoded version of the query_string passed in the request.
- `queryType` – The query type (`fulltext`).
- `count` – The number of results returned.
- `results` – Each result row contains the following:
  - `guid` – The GUID of the entity.
  - `typeName` – The entity type.
  - `score` – The floating point score of how relevant the entity is to the search query. The higher the score, the more relevant the result.
- `dataType` – A partial `TypesDef` Struct (defined in Important Atlas API Datatypes) that describes the search result type. The attribute definitions of the `TypesDef` are not complete.

Example Request:

```
GET http://<atlas-server-host:port>/api/atlas/discovery/search/fulltext?query=
crawled+content
```

Example Response:

```
{
 "requestId": "qtp221036634-867 - 5344fa1e-e6f3-486b-ab95-2abc66641226",
 "query": "crawled content",
 "queryType": "full-text",
 "count": 4,
 "results": [{
  "guid": "48406281-f6be-4689-a55b-237e8911c356",
  "typeName": "hbase_column_family",
  "score": 0.63985527
 }, {
  "guid": "959a3b0e-5c14-4927-bc42-fd99146107d4",
  "typeName": "hbase_column_family",
  "score": 0.63985527
 }, {
  "guid": "f96c3641-d266-40ae-867e-52357cbcd7c3",
  "typeName": "hbase_table",
  "score": 0.11449061
 }, {
  "guid": "a8984af2-4a4e-4281-a14d-f58ecaa8a76e",
```

```
  "typeName": "hbase_table",
  "score": 0.11449061
 }]
}
```

Note how the results are ranked with varying scores. The query string “crawled content” returns both `hbase_column_family` and `hbase_table` attributes. However, because the “crawled content” is a sub-string in the description for `hbase_column_family`, it has a higher score than the `hbase_table` results.

6.4.4. Searching for Entities Associated with Traits

In Cataloging Metadata, we saw how Business Taxonomy Trait instances can be associated with entities, and thereby add the additional metadata information that the Trait conveys. Once the association is made, we can also search for entities associated with a given Trait.

The main advantage of using Trait-based search is that it can provide precise results, but the results are not restricted to a specific Type (as is the case with DSL search).

For example, you can use the `PublicData` Trait to search among all assets in a metadata repository, including HBase tables, for assets that contain content obtained by crawling public data.

Trait search is a special form of DSL search.

6.4.4.1. Search Among All Entities

Request:

```
GET http://<atlas-server-host:port>/api/atlas/discovery/search/dsl?query=
%60trait_name%60
```

The `dsl_query_string` should be encoded using standard URL encoding criteria. The `%60` encoding represents the back-tick character.

Response:

```
{
 "requestId": string,
 "query": trait_name_within_backticks,
 "queryType": "dsl",
 "count": int,
 "results": [{
  "$typeName$": "__tempQueryResultStruct...",
  "instanceInfo": {
   "$typeName$": "__IdType",
   "guid": guid of entity associated to trait,
   "typeName": type of entity associated to trait
  },
  "traitDetails": null
 }, ....],
 "dataType": {
  "typeName": "__tempQueryResultStruct...",
  "typeDescription": null,
  "attributeDefinitions": [{
   "name": "traitDetails",
   "dataTypeName": trait name,
   "multiplicity": {
```

```
    "lower": 0,
    "upper": 1,
    "isUnique": false
   },
   "isComposite": false,
   "isUnique": false,
   "isIndexable": true,
   "reverseAttributeName": null
  }, {
   "name": "instanceInfo",
   "dataTypeName": "__IdType",
   "multiplicity": {
    "lower": 0,
    "upper": 1,
    "isUnique": false
   },
   "isComposite": false,
   "isUnique": false,
   "isIndexable": true,
   "reverseAttributeName": null
  }]
 }
}
```

Response field descriptions:

- `query` – The unencoded version of the dsl_query_string passed in the request.
- `queryType` – The query type (`dsl`).
- `count` – The number of results returned.
- `results` – Each result element contains an instanceInfo map, which in turn contains the following attributes:
  - `guid` – Contains the GUID of the entity associated with the trait.
  - `typeName` – Contains the Type of the entity associated with the trait.
- `dataType` – A partial `TypesDef` Struct (defined in Important Atlas API Datatypes) that describes the search result type. The `dataType` structure contains the `traitDetails` and `instanceInfo` attribute definitions.

Example Request:

```
GET http://<atlas-server-host:port>/api/atlas/discovery/search/dsl?query=
%60PublicData%60
```

Example Response:

```
{
 "requestId": "qtp221036634-965 - a42d6e7f-a6c7-494d-8560-915e2b055ec2",
 "query": "`PublicData`",
 "queryType": "dsl",
 "count": 1,
 "results": [{
  "$typeName$": "__tempQueryResultStruct132",
  "instanceInfo": {
   "$typeName$": "__IdType",
```

```
   "guid": "a8984af2-4a4e-4281-a14d-f58ecaa8a76e",
   "typeName": "hbase_table"
  },
  "traitDetails": null
 }],
 "dataType": {
  "typeName": "__tempQueryResultStruct132",
  "typeDescription": null,
  "attributeDefinitions": [{
   "name": "traitDetails",
   "dataTypeName": "PublicData",
   "multiplicity": {
    "lower": 0,
    "upper": 1,
    "isUnique": false
   },
   "isComposite": false,
   "isUnique": false,
   "isIndexable": true,
   "reverseAttributeName": null
  }, {
   "name": "instanceInfo",
   "dataTypeName": "__IdType",
   "multiplicity": {
    "lower": 0,
    "upper": 1,
    "isUnique": false
   },
   "isComposite": false,
   "isUnique": false,
   "isIndexable": true,
   "reverseAttributeName": null
  }]
 }
}
```

6.4.4.2. Search Among a Specific Type

Definition:

You can use the DSL `isa` operator to restrict a search to only a given type of assets.

Request Structure:

<DSL Query>: <type_name**isa** <tag_or_trait_name>

Request Format:

```
GET http://<atlas-server-host:port>/api/atlas/discovery/search/dsl?query=
{type_name}+isa+%60{tag_or_trait_name}%60
```

The `dsl_query_string` should be encoded using standard URL encoding criteria. The `%60` encoding represents the back-tick character.

Response:

The response takes the same form as in the previous “Search API” section.

Example Request:

```
GET http://<atlas-server-host:port>/api/atlas/discovery/search/dsl?query=
hbase_table+isa+%60PublicData%60
```

Example Response:

```
{
 "requestId": "qtp221036634-867 - 3448a43c-bc07-4b5d-abdd-247acac687f4",
 "query": "hbase_table isa `PublicData`",
 "queryType": "dsl",
 "count": 1,
 "results": [{
  "$typeName$": "hbase_table",
  "$id$": {
   "id": "a8984af2-4a4e-4281-a14d-f58ecaa8a76e",
   "$typeName$": "hbase_table",
   "version": 0,
   "state": "ACTIVE"
  },
  "namespace": {
   "id": "d3eb90fa-53c8-473b-bc41-37e46c250bf0",
   "$typeName$": "hbase_namespace",
   "version": 0,
   "state": "ACTIVE"
  },
  "qualifiedName": "default.webtable@cluster2",
  "isEnabled": true,
  "description": "Table that stores crawled information",
  "columnFamilies": [{
   "$typeName$": "hbase_column_family",
   "$id$": {
    "id": "00b16f6d-ee04-4587-b68e-1ee70dac6b11",
    "$typeName$": "hbase_column_family",
    "version": 0,
    "state": "ACTIVE"
   },
   "qualifiedName": "default.webtable.anchor@cluster2",
   "blockSize": 128,
   "columns": [{
    "id": "f7ce9fbb-7242-4304-b42a-f65309bad8b0",
    "$typeName$": "hbase_column",
    "version": 0,
    "state": "ACTIVE"
   }, {
    "id": "80708552-56b8-4989-9ba7-281bcc97a1a6",
    "$typeName$": "hbase_column",
    "version": 0,
    "state": "ACTIVE"
   }],
   "owner": "crawler",
   "compression": "zip",
   "versions": 3,
   "description": "The anchor column family that stores all links",
   "name": "anchor",
   "inMemory": true
  }, {
   "$typeName$": "hbase_column_family",
   "$id$": {
    "id": "959a3b0e-5c14-4927-bc42-fd99146107d4",
    "$typeName$": "hbase_column_family",
    "version": 0,
```

```
   "state": "ACTIVE"
  },
  "qualifiedName": "default.webtable.contents@cluster2",
  "blockSize": 1024,
  "columns": [{
   "id": "fc711cee-185f-4f09-a2f9-a96e0173f51b",
   "$typeName$": "hbase_column",
   "version": 0,
   "state": "ACTIVE"
  }],
  "owner": "crawler",
  "compression": "lzo",
  "versions": 1,
  "description": "The contents column family that stores the crawled
content",
  "name": "contents",
  "inMemory": false,
  "$traits$": {
   "Retainable": {
    "$typeName$": "Retainable",
    "retentionPeriod": 100
   }
  }
 }],
 "name": "webtable",
 "$traits$": {
  "Catalog.term2": {
   "$typeName$": "Catalog.term2",
   "available_as_tag": false,
   "description": "Changing description for term",
   "name": "Catalog.term2",
   "acceptable_use": null
  },
  "PublicData": {
   "$typeName$": "PublicData"
  }
 }
}],
"dataType": {
 "superTypes": ["DataSet"],
 "hierarchicalMetaTypeName": "org.apache.atlas.typesystem.types.ClassType",
 "typeName": "hbase_table",
 "typeDescription": null,
 "attributeDefinitions": [{
  "name": "namespace",
  "dataTypeName": "hbase_namespace",
  "multiplicity": {
   "lower": 1,
   "upper": 1,
   "isUnique": false
  },
  "isComposite": false,
  "isUnique": false,
  "isIndexable": true,
  "reverseAttributeName": null
 }, {
  "name": "isEnabled",
  "dataTypeName": "boolean",
  "multiplicity": {
   "lower": 0,
```

```
    "upper": 1,
    "isUnique": false
   },
   "isComposite": false,
   "isUnique": false,
   "isIndexable": true,
   "reverseAttributeName": null
  }, {
   "name": "columnFamilies",
   "dataTypeName": "array<hbase_column_family>",
   "multiplicity": {
    "lower": 1,
    "upper": 2147483647,
    "isUnique": false
   },
   "isComposite": true,
   "isUnique": false,
   "isIndexable": true,
   "reverseAttributeName": null
  }]
 }
}
```

6.5. Integrating Messaging with Atlas

So far we have covered integrating with Atlas using its REST API. In the Architecture section, we mentioned another method of integration: a Messaging interface.

The Atlas Messaging interface uses Apache Kafka. Apache Kafka is a scalable, reliable, high-performance messaging system. It provides a mechanism for integrating between Atlas and metadata sources that generate a high volume of metadata events.

Kafka also provides a durable message store. This way, metadata change events can be written to Kafka by sources even if Atlas is not available to process them at the moment. This allows for a very loosely coupled integration, and therefore generally more reliability in a distributed architecture. This is also true for consumers of metadata change events that Atlas communicates via Kafka.

While this durability offers safety guarantees, when Atlas is active (along with the other metadata sources and consumers) it can process metadata events in real time.

Kafka is used for two types of messages. Each message type is written to a specific topic in Kafka.

Note also that the Messaging integration is restricted to Entity notifications. Type-related changes are still handled via the API layer. This is acceptable because Types are created much less frequently than Entities.

The following sections provide the formats for messages that are written to ATLAS_HOOK and ATLAS_ENTITIES.

6.5.1. Publishing Entity Changes to Atlas

Metadata sources can communicate the following forms of entity changes to Atlas: creation, updates, and deletions of entities. These messages are referred to as

`HookNotification` messages in the Atlas source code. There are four types of these messages described in the following sections. The sources can publish these messages to the ATLAS_HOOK topic, and the Atlas server will pick these up and process them. The format of publishing should be using String encoding of Kafka. Any Kafka producer client compatible with the Kafka broker version can be used for this purpose.

6.5.1.1. ENTITY_CREATE Message

ENTITY_CREATE notification messages are used to add one or more entities to Atlas. An ENTITY_CREATE message has the following format:

```
{
 "version": {
  "version": version_string
 },
 "message": {
  "entities": [array of entity_definition_structure],
  "type": "ENTITY_CREATE",
  "user": user_name
 }
}
```

Attribute Definitions:

- `version` – This structure has one field version, which is of the form `major.minor.revision`. This has been introduced to allow Atlas to evolve message formats while still allowing compatibility with older messages. In the 0.7-incubating release, the supported version number is `1.0.0`.
- `message` – This structure contains the details of the message.
  - `entities` – This is an array of entities that must be added to Atlas. Each element in the array is an `EntityDefinition` structure that is defined in Important Atlas API Datatypes.
  - `type` – The type of this message is `ENTITY_CREATE`.
  - `user` – This is the name of the user on whose behalf the entity is being added. Typically it will be the service through which metadata is generated.

Example :

The following example is an `hbase_namespace` message that is being added to Atlas. Note that it is a single element array, and the element structure matches the entity definition of an `hbase_namespace` entity.

```
{
 "version": {
  "version": "1.0.0"
 },
 "message": {
  "entities": [{
   "jsonClass": "org.apache.atlas.typesystem.json.InstanceSerialization
$_Reference",
   "id": {
    "jsonClass": "org.apache.atlas.typesystem.json.InstanceSerialization$_Id",
    "id": "-146729056513524600O",
```

```
    "version": 0,
    "typeName": "hbase_namespace",
    "state": "ACTIVE"
   },
   "typeName": "hbase_namespace",
   "values": {
    "qualifiedName": "default@cluster3",
    "owner": "hbase_admin",
    "description": "Default HBase namespace",
    "name": "default"
   },
   "traitNames": [],
   "traits": {}
  }],
  "type": "ENTITY_CREATE",
  "user": "integ_user"
 }
}
```

6.5.1.2. ENTITY_FULL_UPDATE Message

There is one important difference between the API and the Messaging modes of communication. The API uses two-way communication that allows Atlas to communicate results back to the caller, while Messaging communication is one-way, and there is no notification from Atlas to the system generating the messages.

Consider how `hbase_table` entities are added using the API. When referring to the `hbase_namespace` a table belongs to, we could use the GUID of the previously added `hbase_namespace` entity. We could retrieve this GUID by using either the value returned by a create request, or by looking it up using a query. Both of these synchronous, two-way modes do not apply for the Messaging system. While it is still possible to make API calls, it defeats the purpose of trying to decouple connection between the metadata sources and Atlas.

To address this situation, Atlas provides an ENTITY_FULL_UPDATE, where you can give an entity definition in full, but mark it as an update request. Atlas uses the unique attribute definition of this entity to check to see if this entity already exists in the metadata store. If it does, the entity attributes are updated with values from the request. Otherwise, they are created.

Thus, to add an `hbase_table` entity and refer to an `hbase_namespace` entity in one of the attributes, you do not need to fetch the GUID using the API. You can simply include all of these entity definitions in an ENTITY_FULL_UPDATE message and Atlas handles this automatically.

The structure of an ENTITY_FULL_UPDATE message is as follows:

```
{
 "version": {
  "version": version_string
 },
 "message": {
  "entities": [array of entity_definition_structure],
  "type": "ENTITY_FULL_UPDATE",
  "user": user_name
 }
}
```

This structure is identical to the ENTITY_CREATE structure, except that the type is ENTITY_FULL_UPDATE.

Example :

In the following example we create an `hbase_table` entity along with `hbase_column_family` and `hbase_column` entities. To refer to the namespace, we include the `hbase_namespace` entity again in the array of entities at the beginning. The structure is given below, but details of all columns, column families, etc. are omitted for the sake of brevity. They follow the same structure as described in the Atlas Entities API.

```
{
 "version": {
  "version": "1.0.0"
 },
 "message": {
  "entities": [{
   "jsonClass": "org.apache.atlas.typesystem.json.InstanceSerialization
$_Reference",
   "id": {
    "id": "-146729056651945600 0",
                                                    ...
   },
   "typeName": "hbase_namespace",
   "values": {
    "qualifiedName": "default@cluster3",
                                                    ...
   },
   "traitNames": [],
   "traits": {}
  }, {
   "jsonClass": "org.apache.atlas.typesystem.json.InstanceSerialization
$_Reference",
   "id": {
    "id": "-146729056651949100 0",
                                                    ...
   },
  }, …, {
   "jsonClass": "org.apache.atlas.typesystem.json.InstanceSerialization
$_Reference",
   "id": {
    "id": "-146729056651961500 0",
                                                    ...
   },
   "typeName": "hbase_table",
   "values": {
    "qualifiedName": "default.webtable@cluster3",
                                                    ...
    "namespace": {
     "jsonClass": "org.apache.atlas.typesystem.json.InstanceSerialization
$_Id",
     "id": "-146729056651945600 0",
     "version": 0,
     "typeName": "hbase_namespace",
     "state": "ACTIVE"
    }
   },
   "traitNames": [],
   "traits": {}
```

```
  }],
  "type": "ENTITY_FULL_UPDATE",
  "user": "integ_user"
 }
}
```

- Note that the ID of the namespace entity (first in the array) is set to a negative number and not the real GUID even though this might already be created in Atlas.
- Note also that when the namespace attribute is defined for the table entity, the same negative ID (-1467290566519456000) is used.
- The `"qualifiedName": "default.webtable@cluster3"` will be what Atlas uses to lookup the namespace entity for updating, because it is defined as the unique attribute for the `hbase_namespace` type.

6.5.1.3. ENTITY_PARTIAL_UPDATE Message

When the entity being updated has already been added to Atlas, you can send a partial update message. This message has the following structure:

```
{
 "version": {
  "version": "1.0.0"
 },
 "message": {
  "typeName": type_name,
  "attribute": unique_attribute_name,
           "attributeValue": unique_attribute_value,
  "entity": {
   "jsonClass": "org.apache.atlas.typesystem.json.InstanceSerialization
$_Reference",
   "id": {
    "jsonClass": "org.apache.atlas.typesystem.json.InstanceSerialization$_Id",
    "id": temp_id,
    "version": 0,
    "typeName": type_name,
    "state": "ACTIVE"
   },
   "typeName": type_name,
   "values": {
    updated_attribute_name: updated_attribute_value
   },
   "traitNames": [],
   "traits": {}
  },
  "type": "ENTITY_PARTIAL_UPDATE",
  "user": user_name
 }
}
```

The structure is very similar to the ENTITY_CREATE and ENTITY_FULL_UPDATE messages, with the following differences:

- `typeName` – The name of the type being updated.
- `attribute` – The unique attribute name of the entity being updated.

- `attributeValue` – The value of the unique attribute.
- `entity` – This is a partial `EntityDefinition` structure with the following fields:
  - `id` – This is a typical ID structure as seen in an `EntityDefinition`, but the ID value can be a temporary value and not the actual GUID.
  - `values` – This is a map whose keys are the attributes of the type that is being updated along with the new values.

Using the `typeName`, `attribute` and `attributeValue`, Atlas can locate the entity that needs to be updated. These parameters are similar to the API parameters described in Update a Subset of Entity Attributes.

Example :

In the following example we update an `hbase_table` entity with `qualifiedNamedefault.webtable@cluster3`, and set the `isEnabled` attribute to `false`, we can add an ENTITY_PARTIAL_UPDATE message as follows:

```
{
 "version": {
  "version": "1.0.0"
 },
 "message": {
  "typeName": "hbase_table",
  "attribute": "qualifiedName",
  "entity": {
   "jsonClass": "org.apache.atlas.typesystem.json.InstanceSerialization
$_Reference",
   "id": {
    "jsonClass": "org.apache.atlas.typesystem.json.InstanceSerialization$_Id",
    "id": "-1467290566551498000",
    "version": 0,
    "typeName": "hbase_table",
    "state": "ACTIVE"
   },
   "typeName": "hbase_table",
   "values": {
    "isEnabled": false
   },
   "traitNames": [],
   "traits": {}
  },
  "attributeValue": "default.webtable@cluster3",
  "type": "ENTITY_PARTIAL_UPDATE",
  "user": "integ_user"
 }
}
```

6.5.1.4. ENTITY_DELETE Message

You can use ENTITY_DELETE to deleted an entity. This message has the following structure:

```
{
 "version": {
  "version": version_string
 },
 "message": {
  "typeName": type_name,
  "attribute": unique_attribute_name,
  "attributeValue": unique_attribute_value,
  "type": "ENTITY_DELETE",
  "user": user_name
 }
}
```

The message structure is a subset of the ENTITY_PARTIAL_UPDATE structure.

- `typeName` – The type name of the entity being deleted.
- `attribute` – The unique attribute name of the type being deleted.
- `attributeValue` – The value of the unique attribute of the type being deleted.

Note that these three attributes form the key through which Atlas can identify an entity to delete, similar to how it can reference an entity for a partial update.

Example :

The following message can be used to delete a `hbase_table` with a `qualifiedName` of `default.webtable@cluster3`.

```
{
 "version": {
  "version": "1.0.0"
 },
 "message": {
  "typeName": "hbase_table",
  "attribute": "qualifiedName",
  "attributeValue": "default.webtable@cluster3",
  "type": "ENTITY_DELETE",
  "user": "integ_user"
 }
}
```

6.5.2. Consuming Entity Changes from Atlas

For every entity that Atlas adds, updates (including association and disassociation of Traits), or deletes, an event is raised from Atlas into the Kafka topic ATLAS_ENTITIES. Applications can consume these events and build functionality that is based on metadata changes.

An excellent example of such an application is Apache Ranger's Tag Based policy management (http://hortonworks.com/hadoop-tutorial/tag-based-policies-atlas-ranger/).

This section describes the message formats for events that are notified from Atlas. Standard Kafka consumers compatible with the version of the Kafka broker Atlas uses can be used to consume these events.

The messages written to ATLAS_ENTITIES are referred to as `EntityNotification` messages in the Atlas source code. There are five types of these events.

6.5.2.1. ENTITY_CREATE Message

An ENTITY_CREATE message is sent when an entity is created in the Atlas metadata store. An ENTITY_CREATE message has the following format:

```
{
 "version": {
  "version": version_string
 },
 "message": {
  "entity": entity_definition_structure,
  "operationType": "ENTITY_CREATE",
  "traits": []
 }
}
```

The message structure is very similar to the ENTITY_CREATE message in Publishing Entity Changes to Atlas, but with the following differences:

- `version` – This structure has one field version, which is of the form `major.minor.revision`. This has been introduced to allow Atlas to evolve message formats while still allowing compatibility with older messages. In the 0.7-incubating release, the supported version number is `1.0.0`. The version number can be used by components to determine if the message is compatible with the structure they can decode.
- `message` – This structure contains the details of the entity.
  - `entity` – This is a single entity that is created. The structure of the entity is exactly the same as the `EntityDefinition` structure described in Important Atlas API Datatypes. This is a critical difference from the ENTITY_CREATE message in the previous publishing section, in that notifications from Atlas always contain only one entity at a time, and not an array. The other key difference is that because these are entities created by Atlas, the IDs assigned will be the actual GUIDs of the entities.
  - `operationType` – The type of this message is `ENTITY_CREATE`.
  - `traits` – This field is empty for this operation.

Example :

When an `hbase_table` is created, `hbase_column` and `hbase_column_family` entities are also created. In the API and messages we have seen thus far, we were creating all of these together. However, as described above, every `hbase_column` entity is notified in a unique separate message as shown below.

```
{
 "version": {
  "version": "1.0.0"
 },
 "message": {
  "entity": {
   "jsonClass": "org.apache.atlas.typesystem.json.InstanceSerialization
$_Reference",
   "id": {
```

```
    "jsonClass": "org.apache.atlas.typesystem.json.InstanceSerialization$_Id",
    "id": "9027517b-1644-4f64-bf2c-7b6b49ae9ef2",
    "version": 0,
    "typeName": "hbase_column",
    "state": "ACTIVE"
   },
   "typeName": "hbase_column",
   "values": {
    "name": "cssnsi",
    "qualifiedName": "default.webtable.anchor.cssnsi@cluster1",
    "owner": "crawler",
    "type": "string"
   },
   "traitNames": [],
   "traits": {}
  },
  "operationType": "ENTITY_CREATE",
  "traits": []
 }
}
```

6.5.2.2. ENTITY_UPDATE Message

This message is sent when an entity is updated by Atlas. The format of this message is as follows:

```
{
 "version": {
  "version": version_string
 },
 "message": {
  "entity": entity_definition_structure,
  "operationType": "ENTITY_UPDATE",
  "traits": []
 }
}
```

This structure is similar to the ENTITY_CREATE message described above. One point to note is that Atlas does not currently say what part of the entity has changed, but the entity field has the complete definition (including unchanged attributes).

Example :

Previously in Update a Subset of Entity Attributes we updated the `hbase_table` to set the `isEnabled` flag to `false`. This operation results in an ENTITY_UPDATE event as shown below. The details of all of the `columnFamilies`, etc. are omitted for the sake of brevity.

```
{
 "version": {
  "version": "1.0.0"
 },
 "message": {
  "entity": {
   "jsonClass": "org.apache.atlas.typesystem.json.InstanceSerialization
$_Reference",
   "id": {
    "jsonClass": "org.apache.atlas.typesystem.json.InstanceSerialization$_Id",
    "id": "de9c64bd-f7fc-4b63-96fa-52879b651efe",
```

```
    "version": 0,
    "typeName": "hbase_table",
    "state": "ACTIVE"
   },
   "typeName": "hbase_table",
   "values": {
    "columnFamilies": [...],
    "name": "webtable",
    "description": "Table that stores crawled information",
    "qualifiedName": "default.webtable@cluster1",
    "isEnabled": false,
    "namespace": {...}
   },
   "traitNames": [],
   "traits": {}
  },
  "operationType": "ENTITY_UPDATE",
  "traits": []
 }
}
```

6.5.2.3. ENTITY_DELETE Message

You can use ENTITY_DELETE to deleted an entity. This message has the following structure:

```
{
 "version": {
  "version": version_string
 },
 "message": {
  "entity": {
   "jsonClass": "org.apache.atlas.typesystem.json.InstanceSerialization
$_Reference",
   "id": id_structure containing GUID of the deleted entity,
   "typeName": typeName,
   "values": empty_map,
   "traitNames": empty_list,
   "traits": empty_map
  },
  "operationType": "ENTITY_DELETE",
  "traits": []
 }
}
```

The message structure is similar to the ENTITY_CREATE and ENTITY_UPDATE messages above. The key difference is that the `values` attribute does not contain any data.

You should also note that the deletion of an entity can result in multiple ENTITY_DELETE messages. This is because when an entity is deleted, any entities referred to in composite attributes of the entity are deleted as well, and these also trigger individual messages.

Example :

In the following example, when the `hbase_table` is deleted, the composite attributes referred in `columnFamilies` are deleted as well. This example shows the ENTITY_DELETE message of one such `hbase_column_family` entity.

```
{
```

```
 "version": {
  "version": "1.0.0"
 },
 "message": {
  "entity": {
   "jsonClass": "org.apache.atlas.typesystem.json.InstanceSerialization
$_Reference",
   "id": {
    "jsonClass": "org.apache.atlas.typesystem.json.InstanceSerialization$_Id",
    "id": "eef88491-8333-4538-8e39-5af1f56de9c5",
    "version": 0,
    "typeName": "hbase_column_family",
    "state": "ACTIVE"
   },
   "typeName": "hbase_column_family",
   "values": {},
   "traitNames": [],
   "traits": {}
  },
  "operationType": "ENTITY_DELETE",
  "traits": []
 }
}
```

6.5.2.4. TRAIT_ADD Message

This message is sent when a Trait instance is associated with an entity. The format of this message is as follows:

```
{
 "version": {
  "version": version_string
 },
 "message": {
  "entity": entity_definition_structure,
  "operationType": "TRAIT_ADD",
  "traits": [{
   "typeName": trait_name,
   "values": {
    trait_attribute: value,
    ...
   }
  }]

 }
}
```

The message structure is similar to an ENTITY_CREATE message.

- `entity` – Contains the entity definition to which the Trait instance is added.
- `traits` – An array containing information about the associated Traits. Each attribute is a structure with the following fields:
  - `typeName` – The name of the Trait being added.
  - `values` – A map whose keys are the attributes defined in the Trait definition, and the corresponding values defined for in the Trait instance that is associated with the entity.

Example :

When the Retainable trait is associated with an `hbase_column_family`, the following TRAIT_ADD message is generated:

```
{
 "version": {
  "version": "1.0.0"
 },
 "message": {
  "entity": {
   "jsonClass": "org.apache.atlas.typesystem.json.InstanceSerialization
$_Reference",
   "id": {
    "jsonClass": "org.apache.atlas.typesystem.json.InstanceSerialization$_Id",
    "id": "7d4575d1-97f9-4f70-aa15-d7c3aaba3352",
    "version": 0,
    "typeName": "hbase_column_family",
    "state": "ACTIVE"
   },
   "typeName": "hbase_column_family",
   "values": {
    "name": "contents",
    "inMemory": false,
    "description": "The contents column family that stores the crawled
 content",
    "versions": 1,
    "compression": "lzo",
    "blockSize": 1024,
    "qualifiedName": "default.webtable.contents@cluster2",
    "columns": [{
     "jsonClass": "org.apache.atlas.typesystem.json.InstanceSerialization
$_Id",
     "id": "340d841a-8682-4ff9-9b8d-797a7aa387e2",
     "version": 0,
     "typeName": "hbase_column",
     "state": "ACTIVE"
    }],
    "owner": "crawler"
   },
   "traitNames": ["Retainable"],
   "traits": {
    "Retainable": {
     "jsonClass": "org.apache.atlas.typesystem.json.InstanceSerialization
$_Struct",
     "typeName": "Retainable",
     "values": {
      "retentionPeriod": 100
     }
    }
   }
  },
  "operationType": "TRAIT_ADD",
  "traits": [{
   "typeName": "Retainable",
   "values": {
    "retentionPeriod": 100
   }
  }]
 }
```

```
}
```

Note that the `traits` attribute contains the Trait instance details.

6.5.2.5. TRAIT_DELETE Message

This message is sent when a Trait instance is disassociated from an entity. The format of this message is as follows:

```
{
 "version": {
  "version": version_string
 },
 "message": {
  "entity": entity_definition_structure,
  "operationType": "TRAIT_DELETE",
  "traits": []
 }
}
```

The message structure is similar to an ENTITY_CREATE message.

- `entity` – Contains the entity definition from which the Trait instance is disassociated.

6.6. Appendix

6.6.1. Important Atlas API Data Types

6.6.1.1. AtlasAttributeDef

The `AtlasAttributeDef` structure contains the details of an attribute that is contained in an Entity, Classification, or Struct. An `AtlasAttributeDef` has the following properties:

- `name` – The attribute name.
- `typeName` – The attribute type.
- `cardinality` – Denotes whether the attribute is a single value, a list, or a set of values. Possible values are SINGLE, LIST, or SET.
- `isIndexable` – Denotes whether the attribute can be indexed by Atlas metadata store.
- `isOptional` – Whether or not it is necessary to pass a value for the attribute.
- `isUnique` – Whether or not the attribute value is unique across the Atlas metadata store.

6.6.1.2. AtlasTypesDef

The `AtlasTypesDef` structure is used in the following APIs:

- Bulk Create Type Definitions

- Get All Type Definitions

The `AtlasTypesDef` structure has the following attributes:

- `enumDefs` – An array of types of metatype `AtlasEnumDef`. Will be empty if no Enums are being queried or defined.
- `structDefs` – An array of types of metatype `AtlasStructDef`. Will be empty if no Structs are being queried or defined.
- `entityDefs` – An array of types of metatype `AtlasEntityDef`. Will be empty if no Classes are being queried or defined.
- `classificationDefs` – An array of types of metatype `AtlasClassificationDef`. Will be empty if no Traits are being queried or defined.
- For each of the `AtlasStruct`, `AtlasEntity` or `AtlasClassification` types defined, the following attributes are found:
  - `name` – The name of the specific `AtlasStruct`, `AtlasEntity`, or `AtlasClassification` being defined.
  - `typeVersion` – The version of the type.
  - `superTypes` – If the type being defined is a `AtlasEntity` or `AtlasClassification`, this will be an array of Strings, each for a super class of the `AtlasEntity` or `AtlasClassification` being defined.
  - `attributeDefs` – An array defining attributes that are part of the `AtlasStruct`, `AtlasEntity` or `AtlasClassification` being defined. Each attribute is an instance of `AtlasAttributeDef`.

6.6.1.3. AtlasEntity

The `AtlasEntity` structure is used in the following APIs:

- Create or Update a Single Entity
- Get Entity Definition Using GUID (and related topics).

The `AtlasEntity` structure has the following attributes:

- `typeName` – The name of the type of which this entity is an instance.
- `guid` – The system-specific identifier.
  - When being used to create an entity, this ID value for the entity MUST be a negative long number. For example, Atlas Java code uses "" + (-System.nanoTime()) as the value for this. An identifier specified like this has a special meaning in Atlas, that it is as yet unassigned in the data stores. Atlas then generates a GUID for this entity and stores it – which becomes the true identifier.
  - When being used to refer to an existing entity, this ID value is the system-generated GUID (in standard GUID format. For example: `139b47b2-b911-47d4-b43c-0493607b4b89`.

- `attributes` – A JSON map of the attribute names and values for the attributes defined in the type definition of the entity. This is obviously the most important part of the entity definition. To set or retrieve the attribute values of the entity, this map will need to be iterated. The encoding of the values in the map is according to the metatype of each attribute, as follows:
  - Basic Metatypes – `Int`, `String`, `Boolean`, etc. Encoding is corresponding string representation.
  - Enum Metatypes – TODO
  - Collection Metatypes – `Arrays`, `Maps` (TODO for Maps). Arrays are encoded as a JSON array, where each element is recursively encoded according to these rules.
  - Composite Metatypes – `AtlasEntity`, `AtlasStruct`, `AtlasClassification` (TODO for AtlasStruct /AtlasClassification). For classes: If the system ID of the entity being referenced is known, only the guid attribute needs to be filled. If the system ID of the entity being referenced is not known, then the full encoding of the entity according to its AtlasEntity should be specified.
- `classifications` – A Map of String to Classification instance definitions. Each entry has the key as the classification name and the value is an `AtlasClassification`.

6.6.1.4. TraitInstanceDefinition

The `TraitInstanceDefinition` is used in the following APIs:

- Associate Trait Instances with Entities
- • In the `traits` Map of the `EntityDefinition` structure.

The `TraitInstanceDefinition` has the following structure:

```
{
  "jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization
$_Struct",
  "typeName":name_of_trait,
  "values":{
    “attribute_name”: “attribute_value”,
    ...
  }
}
```

The `TraitInstanceDefinition` structure has the following properties:

- `jsonClass` – Points to the metatype `org.apache.atlas.typesystem.json.InstanceSerialization$_Struct`
- `typeName` – Refers to the name of the Trait being added.
- `values` – A map of key-value pairs. Key is attribute name defined when defining the Trait. Value is attribute value.

7. Apache Atlas REST API

Apache Atlas exposes a variety of REST endpoints that enable you to work with types, entities, lineage, and data discovery. The following resources provide detailed information about the Apache Atlas REST API:

- Apache Atlas REST API
- Apache Atlas Swagger interactive Atlas REST API interface
- Apache Atlas Technical Reference