Atlas Technical User Guide

Flow (will be removed once we have the entire doc in place)

- Introduction - We can update ASF wiki, but we have some of this section completed in various places.   The docs team should own this.
    - What is Atlas
    - Who is Atlas for
    - Use cases Atlas solves
        - Metadata based policy governance
        - Impact analysis via provenance / lineage tracking

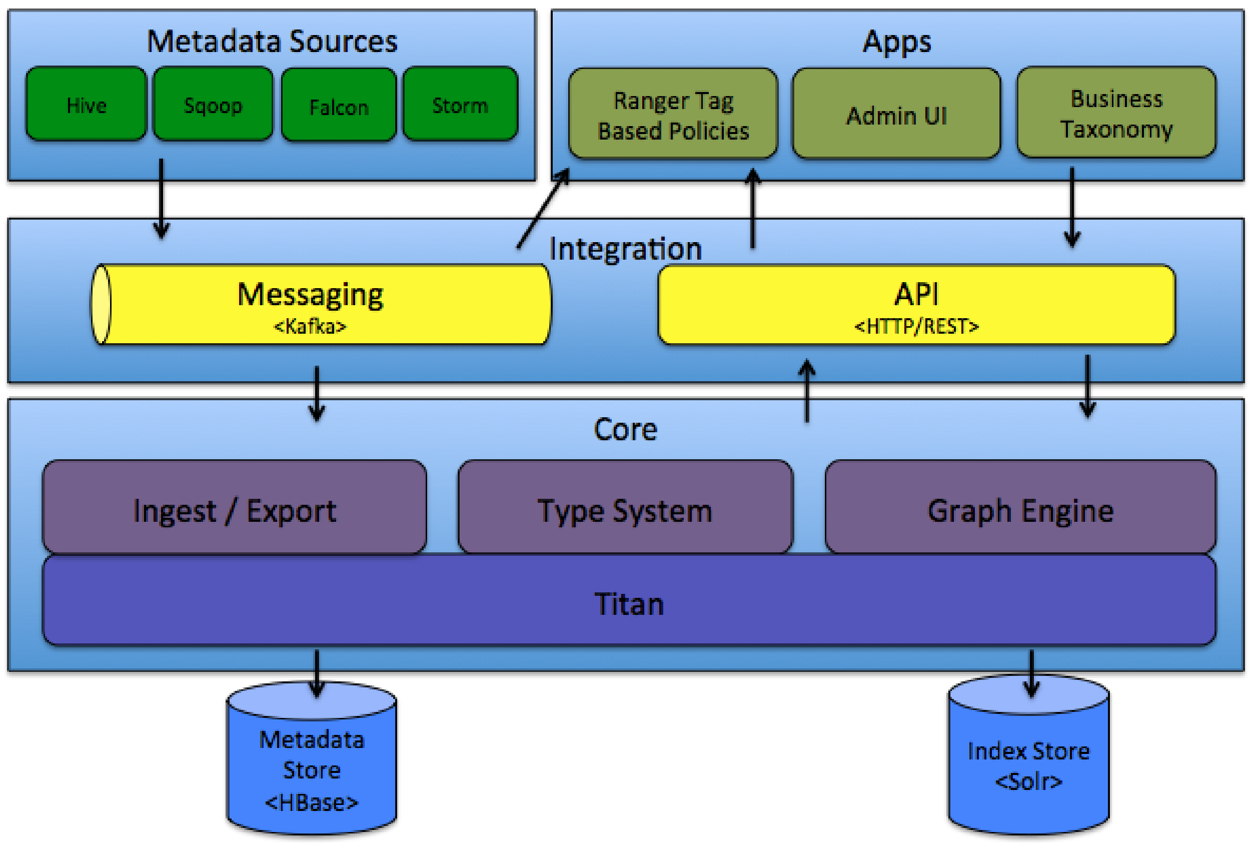
            - Data discovery
- Integrating with Atlas
    - Importing metadata into Atlas - Hooks & Bridges
        - Why we need them
        - Existing hooks
        - Implementing a new hook
            - Integrating via Kafka
            - Message types / formats
        - Importing data into Atlas via Bridges
        - Implementation details to take care of
    - Tracking metadata changes in real time
        - Use case (Ranger)
        - Integrating via Kafka
        - Message types / formats
        - Existing consumers
        - Writing a new consumer
    - Walk through sample user scenario of metadata based management
        - How Ranger policies work?

Introduction to Apache Atlas

TODO >> Update wiki and then recycle here -- AA

Architectural Overview

The following picture describes the components forming the Atlas system.



The components of Atlas can be grouped under the following major categories:

Core

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

Type System: Atlas allows users to define a model for the metadata objects they want to manage. The model is composed of definitions called 'types'. Instances of 'types' called 'entities' represent the actual metadata objects that are managed. The Type System is a component that allows users to define and manage the types and entities. All metadata objects managed by Atlas out of the box (like Hive tables, for e.g.) are modelled using types and represented as entities. To store new types of metadata in Atlas, one needs to understand the concepts of the type system component.

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

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 consume these change events to react to metadata changes in real time.

Graph Engine: Internally, Atlas represents metadata objects it manages using a Graph model. It does this to achieve great flexibility and rich relations between the metadata objects. The Graph Engine is a component that is responsible for translating between types and entities of the Type System, and the underlying Graph model. In addition to managing the Graph objects, The Graph Engine also creates the appropriate indices for the metadata objects so that they can be searched for efficiently.

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 HBase by default and the Index store is configured to Solr.

Integration

Users can manage metadata in Atlas using two 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, users can choose to 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 consume metadata change events from Atlas using which applications can be built. The messaging interface is particularly useful if one wishes to use a more loosely coupled integration with Atlas that could allow for better scalability, reliability etc.

Metadata sources

Atlas supports integration with many sources of metadata out of the box. More integrations will be added in future as well. Currently, Atlas supports ingesting and managing metadata from the following source: Hive, Sqoop, Falcon and Storm. The integration implies two things:

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

Applications

The metadata managed by Atlas is consumed by a variety of applications for satisfying many governance use cases.

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 uses the REST API of Atlas for building its functionality.

Ranger Tag Based Policies: Apache Ranger is an advanced security management solution for the Hadoop ecosystem having wide integration with a variety of Hadoop components. By integrating with Atlas, Ranger allows security administrators to define metadata driven security policies for effective governance. Ranger is a consumer to the metadata change events notified by Atlas.

Business Taxonomy: The metadata objects ingested into Atlas from the Metadata sources are primarily a form of technical metadata. To enhance the discoverability and governance capabilities, Atlas comes with a Business Taxonomy interface that allows users to first, define a hierarchical set of business terms that represent their business domain and associate them to the metadata entities Atlas manages. Business Taxonomy is a web application that is part of the Atlas Admin UI currently and integrates with Atlas using the REST API.

With this birds eye view of the various components that make up Atlas, we can now move to understanding the key concepts of managing metadata with Atlas.

Creating Metadata: The Atlas Type System

Types

A ‘Type’ in Atlas is a definition of how a particular type of metadata objects are stored and accessed. A type represents one or a collection of attributes that define the properties for the metadata object. Users with a development background will recognize the similarity of a type to a ‘Class’ definition of object oriented programming languages, or a ‘table schema’ of relational databases.

An example of a type that comes natively defined with Atlas is a Hive table. A Hive table is defined with these 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
```

The following points can be noted from the above example:

- A type in Atlas is identified uniquely by a 'name'
- A type has a metatype. A metatype represents the type of this model in Atlas. Atlas has the following metatypes:
    - `Basic metatypes: E.g. Int, String, Boolean etc.`
    - `Enum metatypes: TODO`
    - `Collection metatypes: E.g. Array, Map`
    - `Composite metatypes: E.g. Class, Struct, Trait`

- A type can 'extend' from a parent type called 'supertype' - by virtue of this, it will get to include the attributes that are defined in the supertype as well. This allows modellers to define common attributes across a set of related types etc. This is again similar to the concept of how Object Oriented languages define super classes for a class.
    - In this example, every hive table extends from a pre-defined supertype called a 'DataSet'. More details about this pre-defined types will be provided later.

  It is also possible for a type in Atlas to extend from multiple super types.

- Types which have a metatype of 'Class', 'Struct' or 'Trait' can have a collection of attributes. Each attribute has a name (e.g. 'name') and some other associated properties. A property can be referred to using an expression type_name.attribute_name. It is also good to note that attributes themselves are defined using Atlas metatypes. The difference between ‘Classes’ and ‘Structs’ is explain in the context of Entities in the next section. ‘Traits’ will be discussed in a separate section on “Cataloging Metadata in Atlas”.

    ○ 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 called hive_db` and so on.

● Type references in attributes, (like hive_table.db) are particularly interesting. Note that using such an attribute, we can define arbitrary relationships between two types defined in Atlas and thus build rich models. Note that one can also collect a list of references as an attribute type (e.g. hive_table.cols which represents a list of references from hive_table to the hive_column type)

Entities

An ‘entity’ in Atlas is a specific value or instance of a Class ‘type’ and thus represents a specific metadata object in the real world. Referring back to our analogy of Object Oriented Programming languages, an ‘instance’ is an ‘Object’ of a certain ‘Class’.

An example of an entity will be a specific Hive Table. Say Hive has a table called ‘customers’ in the ‘default’ database. This table will be an ‘entity’ in Atlas of type hive_table. By virtue of being an instance of a class type, it will have values for every attribute that are a part of the Hive table ‘type’, such as:

```
id: "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
```

The following points can be noted from the example above:

- Every entity that is an instance of a Class type is identified by a unique identifier, 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. At any point in time, this particular entity can be accessed using 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 the attribute names and their values for attributes that are defined in the hive_table type definition.
- Attribute values will be according to the metatype of the attribute.
    - Basic metatypes: integer, String, boolean values.
        - `E.g. 'name' = 'customers'`
        - `'Temporary' = 'false'`
    - Collection metatypes: An array or map of values of the contained metatype.
        - `E.g. parameters = { "transient_lastDdlTime": "1466403208"`
    - Composite metatypes: For classes, the value will be an entity with which this particular entity will have a relationship.
        - `E.g. The hive table "customers" is present in a database called "default". The relationship between the table and database are captured via the "db" attribute. Hence, the value of the "db" attribute will be a GUID that uniquely identifies the hive_db entity called "default"`

With this idea on entities, 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 Id attribute (with a GUID value) and can be referenced from other entities (like a hive_db entity 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.

Attributes

We already saw that attributes are defined inside composite metatypes like Class and Struct. But we simplistically referred to attributes as having a name and a metatype value. However, attributes in Atlas have some more properties that define more concepts related to the type system.

An attribute has the following properties:

```
name: string,
dataTypeName: string,
isComposite: boolean,
isIndexable: boolean,
isUnique: boolean,
multiplicity: enum,
reverseAttributeName: string
```

The properties above have the following meanings:

- Name - the name of the attribute
- dataTypeName - the metatype name of the attribute (native, collection or composite)
- isComposite -
  - This flag indicates an aspect of modelling. If an attribute is defined as composite, it means that it cannot have a lifecycle independent of the entity it is contained in. A good example of this concept is the set of columns that make a part of a hive table. Since the columns do not have meaning outside of the hive table, they are defined as composite attributes.
  - A composite attribute must be created in Atlas along with the entity it is contained in. i.e. A hive column must be created along with the hive table.
- isIndexable -
  - This flag indicates whether this property should be indexed on, so that look ups can be performed using the attribute value as a predicate and can be performed efficiently.
- isUnique -
  - This flag is again related to indexing. If specified to be unique, it means that a special index is created for this attribute in Titan that allows for equality based look ups.
  - Any attribute with a true value for this flag is treated like a primary key to distinguish this entity from other entities. Hence care should be taken ensure that this attribute does model a unique property in real world.
    - For e.g. 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 amongst multiple clusters. Only a cluster location, database name and table name can be deemed unique in the physical world.

- Multiplicity - indicates whether this attribute is required, optional, or could be multi-valued. If an entity's definition of the attribute value does not match the multiplicity declaration in the type definition, this would be a constraint violation and the entity addition will fail. This field can therefore be used to define some constraints on the metadata information.
- reverseNameAttribute: TODO

Using the above, let us expand on the attribute definition of one of the attributes of the hive table below. Let us look at the attribute called 'db' which represents the database to which the hive table belongs:

```
db:
      "dataTypeName": "hive_db",
      "isComposite": false,
      "isIndexable": true,
      "isUnique": false,
      "multiplicity": "required",
      "name": "db",
      "reverseAttributeName": null
```

Note the "required" constraint on multiplicity. A table entity cannot be sent without a db reference.

```
columns:
      "dataTypeName": "array<hive_column>",
      "isComposite": true,
      "isIndexable": true,
      “isUnique": false,
      "multiplicity": "optional",
      "name": "columns",
      "reverseAttributeName": null
```

Note the "isComposite" true value for columns. By doing this, we are indicating that the defined column entities should always be bound to the table entity they are defined with.

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

System types and their significance

Atlas comes with a few pre-defined system types. We saw one example (DataSet) in the preceding sections. In this section we will see all these types and understand their significance.

Referenceable: This type represents all entities that can be searched for using a unique attribute called `qualifiedName.`

Asset: This type contains attributes like `name, description and owner.` Name is a required attribute (multiplicity = required), the others are optional.

The purpose of Referenceable and Asset is to provide modellers with 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 attributes they can expect of types by default.

Infrastructure: This type extends Referenceable and Asset and typically can be used to be a common super type for infrastructural metadata objects like clusters, hosts etc.

DataSet: This type extends Referenceable and Asset. Conceptually, it can be used to represent an 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 e.g. the columns attribute in a hive_table. Also entities of types that extend DataSet 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.

Conceptual Example

To see the creation of types and entities in action, let us consider an example model. We shall try and model HBase metadata in Atlas. Since this is for the purpose of illustration, we will restrict the scope to only few concepts of HBase metadata itself and not do very extensive analysis into the modelling exercise. Also, please do not consider this as a real world example of how HBase metadata should be modelled.

In the example, we shall assume a cluster_name represents some identifier for the cluster where HBase is running.

For the purpose of the example, we will have the following types:

- `hbase_namespace`: A HBase namespace
    - `name`: Short name for the namespace
    - `qualifiedName`: A unique name that is of the form name@cluster_name
- `hbase_table`: A HBase table
    - `name`: Short name for the table
    - `namespace`: Namespace to which the table belongs
    - `qualifiedName`: A unique name that is of the form namespace.name@cluster_name
    - `description`: Description for the table
    - `isEnabled`: True if enabled, false otherwise
    - `owner`: Who created the table
    - `columnFamilies`: Array of column families for the table
- `hbase_column_family`: A HBase column family
    - `name`: Short name of the column family
    - `qualifiedName`: A unique name that is of the form namespace.name.cf_name@cluster_name
    - `versions`: number of versions to store
    - `inMemory`: boolean
    - `blockSize`: int
    - `compression`: String
    - `columns`: Array of columns in the column family
- `hbase_column`: A HBase column
    - `name`: short name of the column
    - `qualifiedName`: A unique name that is of the form namespace.name.cf_name.col_name@cluster_name
    - `type`: An application specific data type stored in the column
- `hbase_replication_process`: A Process that models replication of a HBase table from one cluster to another.
    - `name`: name for the replication, including what tables are replicated
    - `qualifiedName`: Same as name
    - `replicationSchedule`: A String, indicating how often replication happens
    - `replicationEnabled`: A boolean, indicating whether replication is enabled or not.

Using the Hive integration work as a model, let us define the following:

- All types above will extend from the `Asset` type so that they would get the conventional attributes like name, qualifiedName, description etc.
- `hbase_table` stores core data. So, let us make it a DataSet so that it can be used to capture lineage.
- For purpose of illustration, let us make `hbase_table` contain the `hbase_column_families` as a composite attribute
- A `hbase_column_family` will contain an array of `hbase_columns`. But for purpose of illustration, let us not make them composite, so as to show the difference in creating composite and non-composite attributes and other differences.
- `hbase_replication_process` will extend from the `Process` type so that lineage will be captured as tables are replicated from one cluster to another.

For entities, let us define the following:

- One `hbase_namespace` entity of name “`default`”
- One `hbase_table` entity with name “`webtable`”
- The `hbase_table` entity contains two `hbase_column_family` entities: `anchor` and `content`
- The `anchor` column family contains two `hbase_column` entities: `cssnsi, mylookca`.
- The `content` column family contains one `hbase_column` entity: `contents`.
- Two `hbase_table` instances are created in two different clusters - local and remote - to demonstrate how lineage is captured.
- An instance of the `hbase_replication_process` is created to capture this lineage information.

With this conceptual model in mind, let us now dive into interacting with Atlas through the API it exposes.

It is assumed that you have access to a functional Apache Atlas server for using the APIs. It is also assumed that you have access to the relevant authentication and authorization information. Other than this, a normal HTTP client is sufficient for running the example.

Atlas ***Types*** APIs

Summary

- **Base resource name:** `/types`
- **Full URL:** `http://<atlas-server-host:port>/api/atlas/types`
- **Operations allowed:**
    - GET: list all the types

  - GET: list a specific type: `http://<atlas-server-host:port>/api/atlas/types/{type_name}`
  - POST: Create new types
  - PUT: Update existing types (with some restrictions)

Listing all types

Request

```
GET http://<atlas-server-host:port>/api/atlas/types
GET
http://<atlas-server-host:port>/api/atlas/types?type=STRUCT|CLASS|TRA
IT
```

Response

```
{
  "results": [
    "Asset",
    "hive_column",
    "Process",
    "storm_node",
    "storm_bolt",
    "falcon_process",
    "falcon_feed_replication",
    "hive_serde",
    "kafka_topic",
    "hive_table",
    "hive_storagedesc",
    "sqoop_dbdatastore",
    "hive_principal_type",
    "fs_permissions",
    "jms_topic",
    "hive_process",
    "falcon_cluster",
    "storm_spout",
    "Referenceable",
    "falcon_feed_creation",
    "falcon_feed",
    "hdfs_path",
    "sqoop_process",
    "Infrastructure",
    "storm_topology",
    "hive_order",
    "DataSet",
    "fs_path",
    "hive_db",
    "file_action"
  ],
  "count": 34,
  "requestId": "qtp1803965313-15 -
b17554dc-f93d-4d7f-a6ed-e124388a5759"
}
```

Description

This API lists all the types that the Atlas server currently knows about. The return object is an array of type names that identify each type registered with the server.

If provided with a type filter (either STRUCT, TRAIT or CLASS), only those types of that metatype are returned.

Retrieving a type definition

Request

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

E.g. `GET http://<atlas-server-host:port>/api/atlas/types/hive_column`

Response

```
{
  "typeName": "hive_column",
  "definition": {
    "enumTypes": [],
    "structTypes": [],
    "traitTypes": [],
    "classTypes": [
      {
        "superTypes": [
          "Referenceable",
          "Asset"
        ],
        "hierarchicalMetaTypeName":
"org.apache.atlas.typesystem.types.ClassType",
        "typeName": "hive_column",
        "typeDescription": null,
        "attributeDefinitions": [
          {
            "name": "type",
            "dataTypeName": "string",
            "multiplicity": "required",
            "isComposite": false,
            "isUnique": false,
            "isIndexable": true,
            "reverseAttributeName": null
          },
          {
            "name": "comment",
            "dataTypeName": "string",
            "multiplicity": "optional",
            "isComposite": false,
            "isUnique": false,
            "isIndexable": true,
            "reverseAttributeName": null
          },
          {
            "name": "table",
            "dataTypeName": "hive_table",
            "multiplicity": "optional",
            "isComposite": false,
            "isUnique": false,
            "isIndexable": true,
            "reverseAttributeName": "columns"
          }
        ]
      }
    ]
  },
  "requestId": "qtp1803965313-15 -
36153858-294b-46a5-a41c-acd55ded0e38"
}
```

Description

The response to get a specific type definition contains the following attributes:

- **`typeName`**: Name of the type being defined
- **`definition`**: A `TypesDef` structure. Please refer to the `TypesDef` structure definition for a description of the fields.

Creating new types

Request

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

Body

The body of the above POST request is a `TypesDef` structure defined in the “Important Atlas API Datatypes” section.

E.g. Body

```
{
  "enumTypes":[

  ],
  "structTypes":[

  ],
  "traitTypes":[

  ],
  "classTypes":[
    {
      "superTypes":[
        "Referenceable",
        "Asset"
      ],

"hierarchicalMetaTypeName":"org.apache.atlas.typesystem.types.ClassTy
pe",
      "typeName":"hbase_namespace",
      "typeDescription":null,
      "attributeDefinitions":[

      ]
    },
    {
      "superTypes":[
        "Referenceable",
        "Asset"
      ],

"hierarchicalMetaTypeName":"org.apache.atlas.typesystem.types.ClassTy
pe",
      "typeName":"hbase_column",
      "typeDescription":null,
      "attributeDefinitions":[
        {
          "name":"type",
          "dataTypeName":"string",
          "multiplicity":"required",
          "isComposite":false,
          "isUnique":false,
          "isIndexable":true,
          "reverseAttributeName":null
        }
      ]
    },
    {
      "superTypes":[
        "Referenceable",
        "Asset"
      ],

"hierarchicalMetaTypeName":"org.apache.atlas.typesystem.types.ClassTy
pe",
      "typeName":"hbase_column_family",
      "typeDescription":null,
      "attributeDefinitions":[
        {
          "name":"versions",
          "dataTypeName":"int",
          "multiplicity":"optional",
          "isComposite":false,
          "isUnique":false,
          "isIndexable":true,
          "reverseAttributeName":null
        },
        {
          "name":"inMemory",
          "dataTypeName":"boolean",
          "multiplicity":"optional",
          "isComposite":false,
          "isUnique":false,
          "isIndexable":true,
          "reverseAttributeName":null
        },
        {
          "name":"blockSize",
          "dataTypeName":"int",
          "multiplicity":"required",
          "isComposite":false,
          "isUnique":false,
          "isIndexable":true,
          "reverseAttributeName":null
        },
        {
          "name":"compression",
          "dataTypeName":"string",
          "multiplicity":"optional",
          "isComposite":false,
          "isUnique":false,
          "isIndexable":true,
          "reverseAttributeName":null
        },
        {
          "name":"columns",
          "dataTypeName":"array<hbase_column>",
          "multiplicity":"collection",
          "isComposite":false,
          "isUnique":false,
          "isIndexable":true,
          "reverseAttributeName":null
        }
      ]
    },
    {
      "superTypes":[
        "DataSet",
        "Asset"
      ],

"hierarchicalMetaTypeName":"org.apache.atlas.typesystem.types.ClassTy
pe",
      "typeName":"hbase_table",
      "typeDescription":null,
      "attributeDefinitions":[
        {
          "name":"namespace",
          "dataTypeName":"hbase_namespace",
          "multiplicity":"required",
          "isComposite":false,
          "isUnique":false,
          "isIndexable":true,
          "reverseAttributeName":null
        },
        {
          "name":"isEnabled",
          "dataTypeName":"boolean",
          "multiplicity":"optional",
          "isComposite":false,
          "isUnique":false,
          "isIndexable":true,
          "reverseAttributeName":null
        },
        {
          "name":"columnFamilies",
          "dataTypeName":"array<hbase_column_family>",
          "multiplicity":"collection",
          "isComposite":true,
          "isUnique":false,
          "isIndexable":true,
          "reverseAttributeName":null
        }
      ]
    },
    {
      "superTypes":[
        "Process"
      ],

"hierarchicalMetaTypeName":"org.apache.atlas.typesystem.types.ClassTy
pe",
      "typeName":"hbase_replication_process",
      "typeDescription":null,
      "attributeDefinitions":[
        {
          "name":"replicationSchedule",
          "dataTypeName":"string",
          "multiplicity":"required",
          "isComposite":false,
          "isUnique":false,
          "isIndexable":true,
          "reverseAttributeName":null
        },
        {
          "name":"replicationEnabled",
          "dataTypeName":"boolean",
          "multiplicity":"required",
          "isComposite":false,
          "isUnique":false,
          "isIndexable":true,
          "reverseAttributeName":null
        }
      ]
    }
  ]
}
```

The above example is the TypesDef structure for all the HBase types that we defined in the section “Conceptual Example”. Since all of those types are classes, we have an array in the `classTypes` attribute, one element for each type being defined.

Response

```
{
  "requestId": "qtp221036634-204 -
30245c0e-cb2a-4c0e-96fd-78394413e466",
  "types": [
    {
      "name": "hbase_replication_process"
    },
    {
      "name": "hbase_column_family"
    },
    {
      "name": "hbase_column"
    },
    {
      "name": "hbase_table"
    },
    {
      "name": "hbase_namespace"
    }
  ]
}
```

The response of a Create Types request contains a single element `types`. This is an array of structures, one for each type that is defined. The structure contains, one attribute `name`, which is the name of the type being defined.

Updating a type

TODO

Atlas ***Entities*** APIs

In the previous section, we saw the different APIs to creating and querying types in Atlas. In this section, we shall look at the APIs to create, read, update and delete entities of those types.

Summary

- **Base resource name:** `/entities`
- **Full URL:** `http://<atlas-server-host:port>/api/atlas/entities`
- **Operations supported:**
    - POST: Create new entities
    - POST: Update an entity using a unique key (TODO: Why do we need the property and value params):
        - URL: `http://<atlas-server-host:port>/api/atlas/entities/{qualifiedName}?type=type_name&property=unique_propery_name&value=unique_property_value`
    - POST: Update an entity by guid (TODO: What's the use case for this API):
        - URL: `http://<atlas-server-host:port>/api/atlas/entities/{guid}?property=propery_name_to_update`
    - PUT: Update entity
    - DELETE: Delete entities either by GUIDs or by unique property name and value
        - URL: `http://<atlas-server-host:port>/api/atlas/entities?guid=list_of_guids (OR) http://<atlas-server-host:port>/api/atlas/entities?type=type_name&property=unique_propery_name&value=unique_property_value`
    - GET: Get an entity by GUID: `http://<atlas-server-host:port>/api/atlas/entities/{guid}`
    - GET: Get an entity by unique attribute: `http://<atlas-server-host:port>/api/atlas/entities?type=type_name&property=unique_propery_name&value=unique_property_value`

Creating new entities

Request

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

Body

The body for this POST request is one or more `EntityDefinition` structures that is described in the “Important Atlas API Datatypes” section below.

Response

The response for the POST request contains the following important attributes:

- `entities`: This contains following arrays:
  - `created`: List of GUIDs for entity created as part of handling the request.
  - Others: TODO
- `definition:` The `EntityDefinition` structure sent, but with the right GUID for the entity generated by the server.

Example - creating single entity

Referring to our running example, let us say we need to define a hbase_namespace instance. The body for the same will be as follows:

```
{

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Reference",
  "id":{

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Id",
    "id":"-1466683608564093000",
    "version":0,
    "typeName":"hbase_namespace",
    "state":"ACTIVE"
  },
  "typeName":"hbase_namespace",
  "values":{
    "qualifiedName":"default@cluster1",
    "owner":"hbase_admin",
    "description":"Default HBase namespace",
    "name":"default"
  },
  "traitNames":[
  ],
  "traits":{

  }
}
```

Note:

- Since we are creating a new entity, we have specified the ID of the entity to be a long negative number. Refer to “Important Atlas API Datatypes” for details.
- The qualifiedName attribute is being set to a combination of the HBase namespace and the cluster name which we have defined as cluster1. By virtue of this, we hope this will form a uniquely identified value for this entity in the global physical world.

Response Body - Creating single entity

```
{
  "requestId": "qtp221036634-17 -
a65f68bd-db95-4e65-ae63-ac61ca831601",
  "entities": {
    "created": [
      "139b47b2-b911-47d4-b43c-0493607b4b89"
    ]
  },
  "definition": {
    "jsonClass":
"org.apache.atlas.typesystem.json.InstanceSerialization$_Reference",
    "id": {
      "jsonClass":
"org.apache.atlas.typesystem.json.InstanceSerialization$_Id",
      "id": "139b47b2-b911-47d4-b43c-0493607b4b89",
      "version": 0,
      "typeName": "hbase_namespace",
      "state": "ACTIVE"
    },
    "typeName": "hbase_namespace",
    "values": {
      "qualifiedName": "default@cluster1",
      "owner": "hbase_admin",
      "description": "Default HBase namespace",
      "name": "default"
    },
    "traitNames": [],
    "traits": {}
  }
}
```

Example - Creating multiple entities

It is also possible to send multiple entities to create in the same POST request to the entities resource. This is sometimes necessary, if there are composite attributes defined for a type whose entity is being created (e.g. columnFamilies attribute in hbase_table). Sometimes, it is more logical to define entities together, because they are all components of the same entity.

Referring to our running example, now let us try to create a HBase table entity. Note that a table entity, refers to HBase column families and HBase column families, in turn, refer to HBase columns. We define all of these entities now as an array of EntityDefinition structures as below:

```
[{

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Reference",
  "id":{

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Id",
    "id":"-1466771970995057000",
    "version":0,
    "typeName":"hbase_column",
    "state":"ACTIVE"
  },
  "typeName":"hbase_column",
  "values":{
    "qualifiedName":"default.webtable.anchor.cssnsi@cluster1",
    "type":"string",
    "owner":"crawler",
    "name":"cssnsi"
  },
  "traitNames":[],
  "traits":{}
},
{
```

```
"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Reference",
  "id":{

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Id",
    "id":"-1466771970995105000",
    "version":0,
    "typeName":"hbase_column",
    "state":"ACTIVE"
  },
  "typeName":"hbase_column",
  "values":{
    "qualifiedName":"default.webtable.anchor.mylookca@cluster1",
    "type":"string",
    "owner":"crawler",
    "name":"mylookca"
  },
  "traitNames":[],
  "traits":{}
},
{

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Reference",
  "id":{

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Id",
    "id":"-1466771970995127000",
    "version":0,
    "typeName":"hbase_column",
    "state":"ACTIVE"
  },
  "typeName":"hbase_column",
  "values":{
    "qualifiedName":"default.webtable.contents.html@cluster1",
    "type":"byte[]",
    "owner":"crawler",
    "name":"html"
  },
  "traitNames":[],
  "traits":{}
},
{

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Reference",
  "id":{

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Id",
    "id":"-1466771970995148000",
    "version":0,
    "typeName":"hbase_column_family",
    "state":"ACTIVE"
  },
  "typeName":"hbase_column_family",
  "values":{
    "name":"anchor",
    "inMemory":true,
    "description":"The anchor column family that stores all links",
    "versions":3,
    "compression":"zip",
    "blockSize":128,
    "qualifiedName":"default.webtable.anchor@cluster1",
    "columns":[
      {

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Id",
        "id":"-1466771970995057000",
        "version":0,
        "typeName":"hbase_column",
        "state":"ACTIVE"
      },
      {

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Id",
        "id":"-1466771970995105000",
        "version":0,
        "typeName":"hbase_column",
        "state":"ACTIVE"
      }
    ],
    "owner":"crawler"
  },
  "traitNames":[],
  "traits":{}
},
{

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Reference",
  "id":{

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Id",
    "id":"-1466771970995232000",
    "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@cluster1",
    "columns":[
      {

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Id",
        "id":"-1466771970995127000",
        "version":0,
        "typeName":"hbase_column",
        "state":"ACTIVE"
      }
    ],
    "owner":"crawler"
  },
  "traitNames":[],
  "traits":{}
},
{

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Reference",
  "id":{

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Id",
    "id":"-1466771970995263000",
    "version":0,
    "typeName":"hbase_table",
    "state":"ACTIVE"
  },
  "typeName":"hbase_table",
  "values":{
    "columnFamilies":[
      {

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Reference",
        "id":{

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Id",
          "id":"-1466771970995148000",
          "version":0,
          "typeName":"hbase_column_family",
          "state":"ACTIVE"
        },
        "typeName":"hbase_column_family",
        "values":{
          "name":"anchor",
          "inMemory":true,
          "description":"The anchor column family that stores all
links",
          "versions":3,
          "compression":"zip",
          "blockSize":128,
          "qualifiedName":"default.webtable.anchor@cluster1",
          "columns":[
            {
```

```
"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Id",
              "id":"-1466771970995057000",
              "version":0,
              "typeName":"hbase_column",
              "state":"ACTIVE"
            },
            {

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Id",
              "id":"-1466771970995105000",
              "version":0,
              "typeName":"hbase_column",
              "state":"ACTIVE"
            }
          ],
          "owner":"crawler"
        },
        "traitNames":[],
        "traits":{}
      },
      {

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Reference",
        "id":{

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Id",
          "id":"-1466771970995232000",
          "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@cluster1",
          "columns":[
            {

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Id",
              "id":"-1466771970995127000",
              "version":0,
              "typeName":"hbase_column",
              "state":"ACTIVE"
            }
          ],
          "owner":"crawler"
        },
        "traitNames":[],
        "traits":{}
      }
    ],
    "name":"webtable",
    "description":"Table that stores crawled information",
    "qualifiedName":"default.webtable@cluster1",
    "isEnabled":true,
    "namespace":{

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Id",
      "id":"71cbc76e-d1bf-4e92-865a-13624cac7a2a",
      "version":0,
      "typeName":"hbase_namespace",
      "state":"ACTIVE"
    }
  },
  "traitNames":[],
  "traits":{}
}]
```

Clearly, the above is a very verbose structure and appears daunting at first glance. However, it is easy to understand once we break it down.

The array contains 6 elements: one for each of 3 hbase_columns, 2 hbase_column_families, and 1 hbase_table that we defined in the “Conceptual Example” section.

The 3 `hbase_column` entity definitions are straightforward and follow the structure which contain the standard entity definition attributes like `jsonClass, id, typeName` and the values is a map of attribute names and values pertaining to the `hbase_column` definition. (shrunk for brevity):

```
{
  "jsonClass":..,
  "id":{
    "jsonClass":..,
    "id":"-1466690195705126000",
    "version":0,
    "typeName":"hbase_column",
    "state":"ACTIVE"
  },
  "typeName":"hbase_column",
  "values":{
    "qualifiedName":"default.webtable.anchor.cssnsi@cluster1",
      ...
  },
  "traitNames":[],
  "traits":{}
}
```

Next come the 2 hbase_column_family entity definitions, shrunk for brevity. The important point here is to note how the “`columns`” attribute is encoded. Note that the `columns` attribute is an array of `hbase_column` entities. These refer to the entity definitions of the `hbase_columns` being defined in the same request. However, since `columns` is not a composite attribute, only the ID part of the column entity definitions is copied into the “columns” attribute.

```
{
  "jsonClass":"...",
  "id":{
    "jsonClass":"...",
    "id":"-1466690195705260000",
    "version":0,
    "typeName":"hbase_column_family",
    "state":"ACTIVE"
  },
  "typeName":"hbase_column_family",
  "values":{
    "name":"contents",
    "inMemory":false,
    ...
    "columns":[
        "jsonClass":"...",
        "id":"-1466771970995057000",
        "version":0,
        "typeName":"hbase_column",
        "state":"ACTIVE"

    ],
    "owner":"crawler"
  },
  "traitNames":[],
  "traits":{}
}
```

Next comes the definition for the hbase_table, again shrunk for brevity. The important point here is to note how the "`columnFamilies`" attribute is encoded. Note that the `columnFamilies` attribute is an array of `hbase_column_family` entities. These refer to the entity definitions of the `hbase_column_family` being defined in the same request. However, since `columnFamilies` is defined as a composite attribute, the entire entity definition and not just the ID are encoded in the value of the attribute.

```
{
  "jsonClass":"...",
  "id":{
    "jsonClass":"...",
    "id":"-1466771970995263000",
    "version":0,
    "typeName":"hbase_table",
    "state":"ACTIVE"
  },
  "typeName":"hbase_table",
  "values":{
    "columnFamilies":[
      {
        "jsonClass":"...",
        "id":{
          "jsonClass":"...",
          "id":"-1466771970995148000",
          ...
        },
        "typeName":"hbase_column_family",
        "values":{
          "name":"anchor",
          ...
          "columns":[
            {
              "jsonClass":"...",
              "id":"-1466771970995057000",
              ...
            },
            ...
          ],
          "owner":"crawler"
        },
        "traitNames":[],
        "traits":{}
      },
      {
        "jsonClass":"...",
        "id":{
          "jsonClass":"...",
          "id":"-1466771970995232000",
          ...
        },
        "typeName":"hbase_column_family",
        "values":{
          "name":"contents",
          ...
          "columns":[
            {
              "jsonClass":"...",
              "id":"-1466771970995127000",
              ...
            }
          ],
          "owner":"crawler"
        },
        "traitNames":[],
        "traits":{}
      }
    ],
    "name":"webtable",
    "description":"Table that stores crawled information",
    "qualifiedName":"default.webtable@cluster1",
    "isEnabled":true,
    "namespace":{
      "jsonClass":"...",
      "id":"71cbc76e-d1bf-4e92-865a-13624cac7a2a",
      ...
    }
  },
  "traitNames":[],
  "traits":{}
}]
```

Response body for multiple entity request

```
{
  "requestId": "qtp221036634-12 -
edc360bf-0b3b-45ea-9bfc-b0116e778ef8",
  "entities": {
    "created": [
      "f3655669-930f-4202-877e-a8d83d2d3b30",
      "3defc36d-2088-422d-ac6f-eb775be6adbb",
      "73f7716a-9821-47d2-acb4-3c9ceda1fb7e",
      "62ff1e88-92fa-4ec7-a223-5eba6284e568",
      "01105e4f-bbd5-4ea6-b988-b332ade2a451",
      "774a3bee-2aa6-4b6f-81c4-fa00ac40c506"
    ]
  },
  "definition": {
    "jsonClass":
"org.apache.atlas.typesystem.json.InstanceSerialization$_Reference",
    "id": {
      "jsonClass":
"org.apache.atlas.typesystem.json.InstanceSerialization$_Id",
      "id": "f3655669-930f-4202-877e-a8d83d2d3b30",
      "version": 0,
      "typeName": "hbase_column",
      "state": "ACTIVE"
    },
    "typeName": "hbase_column",
    "values": {
      "name": "cssnsi",
      "description": null,
      "qualifiedName": "default.webtable.anchor.cssnsi@cluster1",
      "owner": "crawler",
      "type": "string"
    },
    "traitNames": [],
    "traits": {}
  }
}
```

Note how the `entities.created` array contains 6 GUIDs. These are in the order of the entities defined.

Retrieving an entity definition

In this section, we will see how to retrieve an entity that has already been created.

Request

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

Response

The response to a GET request is an `EntityDefinition` structure that is defined in the "Important Atlas API Datatypes" section below.

Example

For the hbase_table we created above, say the GUID is 4939dca8-29ca-41a0-a920-4537c17d061c. The request will be

```
GET
http://<atlas-server-host:port>/api/atlas/entities/4939dca8-29ca-41a0
-a920-4537c17d061c
```

The response will be:

```
{

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Reference",
  "id":{

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Id",
    "id":"d81e7763-92f8-4a90-8d73-f1a97b10af8e",
    "version":0,
    "typeName":"hbase_table",
    "state":"ACTIVE"
  },
  "typeName":"hbase_table",
  "values":{
    "columnFamilies":[
      {

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Reference",
        "id":{

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Id",
          "id":"6c43e1c2-1251-4e97-bcdc-bfbe1be4698b",
          "version":0,
          "typeName":"hbase_column_family",
          "state":"ACTIVE"
        },
        "typeName":"hbase_column_family",
        "values":{
          "name":"anchor",
          "inMemory":true,
          "description":"The anchor column family that stores all
links",
          "versions":3,
          "compression":"zip",
          "blockSize":128,
          "qualifiedName":"default.webtable.anchor@cluster1",
          "columns":[
            {

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Id",
              "id":"fef85bf1-a903-4cf2-984d-b6613e84c29e",
              "version":0,
              "typeName":"hbase_column",
              "state":"ACTIVE"
            },
            {

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Id",
              "id":"e061416b-3727-4a41-b218-5e2e5b47c439",
              "version":0,
              "typeName":"hbase_column",
              "state":"ACTIVE"
            }
          ],
          "owner":"crawler"
        },
        "traitNames":[

        ],
        "traits":{

        }
      },
      {

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Reference",
        "id":{

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Id",
          "id":"7896ad81-c6b8-4d96-ad5b-13c30919aa09",
          "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@cluster1",
          "columns":[
            {

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Id",
              "id":"00bedbb4-aea1-43ef-97bd-cddbea56b324",
              "version":0,
              "typeName":"hbase_column",
              "state":"ACTIVE"
            }
          ],
          "owner":"crawler"
        },
        "traitNames":[

        ],
        "traits":{

        }
      }
    ],
    "name":"webtable",
    "description":"Table that stores crawled information",
    "qualifiedName":"default.webtable@cluster1",
    "isEnabled":true,
    "namespace":{

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Id",
      "id":"4939dca8-29ca-41a0-a920-4537c17d061c",
      "version":0,
      "typeName":"hbase_namespace",
      "state":"ACTIVE"
    }
  },
  "traitNames":[

  ],
  "traits":{

  }
}
```

The response is mostly self explanatory, except to point out that since the `columnFamilies` is defined as a ‘composite’ attribute in the type definition of `hbase_table`, the full entity definition of the `hbase_column_family` entities are embedded inline in the `hbase_table` entity definition. However, since `hbase_namespace` is not defined as a ‘composite’ attribute , only the ID part of the namespace is included in the `hbase_table` entity definition.

Retrieving an entity definition by unique attribute

It is easier to refer to an entity by a user defined attribute, rather than a system generated GUID. Atlas allows for this capability with the restriction that the query can be done only using an attribute declared as ‘unique’ in the attribute definition.

Request

```
GET
http://<atlas-server-host:port>/api/atlas/entities?type={type_name}&p
roperty={unique_attribute_name}&value={unique_attribute_value}
```

Response

The response for this request is the `EntityDefinition` structure for the entity, defined in the “Important Atlas API Datatypes” section.

Example

In our running example, remember that all our HBase data types extend from the System defined type `Referenceable` which has a unique attribute called `qualifiedName`. So, we can use this to query for a given entity.

```
GET
http://<atlas-server-host:port>/api/atlas/entities?type=hbase_table&p
roperty=qualifiedName&value=default.webtable@cluster1
```

Updating an attribute of an entity

Request

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

Body:

The request body is the same as the `EntityDefinition` structure used in the creation of entities. However, in the 'values' map, it only needs to contain the attributes whose values should change. The rest of the attributes are retained unchanged after this operation by Atlas.

Response

The response is a structure as follows:

- **`entities`**: A structure that contains information about entities that are updated.
    - **`updated`**: An array of GUIDs of entities that were updated.
- **`definition`**: The `EntityDefinition` structure of the entity whose GUID is specified in the request.

Example

Suppose we want to set the isEnabled flag of our HBase table to false (say because it was disabled in HBase), this can be done using this request.

Request Body:

```
{

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Reference",
  "id":{

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Id",
    "id":"-1466773356539553000",
    "version":0,
    "typeName":"hbase_table",
    "state":"ACTIVE"
  },
  "typeName":"hbase_table",
  "values":{
    "isEnabled":false
  },
  "traitNames":[],
  "traits":{}
}
```

Note the similarity of the structure to the `EntityDefinition` in create request. However, note that the `values` attribute contains only the `isEnabled` flag, with the value false.

Response body: The definition has been truncated for brevity. It actually will be the complete `EntityDefinition` of the `hbase_table` entity.

```
{
  "requestId": "qtp221036634-17 -
cbe51ccd-9464-4025-9d76-281c72ea561c",
  "entities": {
    "updated": [
      "af0f04b1-7f37-4e87-9300-c590435beaf9"
    ]
  },
  "definition": {
      ...
      "isEnabled": true,
      ...
}
```

Create Lineage amongst data sets

One important use case Atlas caters to is the tracking of lineage / provenance of data sets as they continuously evolve across data stores and data pipelines. In this section, we shall see how this can be tracked using APIs exposed by Atlas.

In our discussion in the section “System types and their significance”, we introduced the two system types “DataSet” and “Process” which we shall use here to create lineage between data sets.

The general steps to creating lineage are as follows:

- Define a type to represent a data asset and mark this as a subclass of the system type DataSet.

- Define a type to represent a data transformation process that modifies a data asset into another and mark this as a subclass of the system type Process.
- Create entities for the data assets, as described above.
- Create an entity of the data transformation process and fill in the inputs and outputs attributes as an array of DataSet references. Also fill in any other attributes that are defined as part of the data transformation type definition.

As can be seen, the steps to create a lineage do not require any new API calls, only creating the right instances using APIs we have already seen.

Example

- **Step 1**: We have already defined types for the HBase Table in the section “Creating new types”, with the name `hbase_table`. Note that it extended “`DataSet`” as a super type.
- **Step 2:** We have already defined types for the HBase Replication process in the section “Creating new types”, with the name `hbase_replication_process`. Note that it extended “`Process`” as a super type.
- **Step 3:** We have already shown how to create hbase_namespace and hbase_table entities in the section “Creating new entities”. We could use the same entity JSONs to create another set of entities but change the cluster name to cluster2. An example of the hbase_table JSON is replicated below, in condensed form, showing only the changes in name for brevity. Note that the ID used for the namespace attribute in the new hbase_table should be the ID of the new namespace instance created in this step:

```
[{
  "jsonClass":"...",
  "id":{
    "jsonClass":"...",
    ...
  },
  "typeName":"hbase_column",
  "values":{
    "qualifiedName":"default.webtable.anchor.cssnsi@cluster2",
    ...
  },
},
{
  "jsonClass":"...",
  "id":{
    "jsonClass":"...",
  },
  "typeName":"hbase_column",
  "values":{
    "qualifiedName":"default.webtable.anchor.mylookca@cluster2",
    ...
  },
},
{
  "jsonClass":"...",
  "id":{
    "jsonClass":"...",
    ...
  },
  "typeName":"hbase_column",
  "values":{
    "qualifiedName":"default.webtable.contents.html@cluster2",
    ...
  },
},
{
  "jsonClass":"...",
  "id":{
    "jsonClass":"...",
    ...
  },
  "typeName":"hbase_column_family",
  "values":{
    "qualifiedName":"default.webtable.anchor@cluster2",
    ...
  },
},
{
  "jsonClass":"...",
  "id":{
    "jsonClass":"...",
    ...
  },
  "typeName":"hbase_column_family",
  "values":{
    "qualifiedName":"default.webtable.contents@cluster2",
    ...
},
{
  "jsonClass":"...",
  "id":{
    "jsonClass":"...",
    ...
  },
  "typeName":"hbase_table",
  "values":{
    "qualifiedName":"default.webtable@cluster2",
    ...
    "namespace":{
      "jsonClass":"...",
      ...
    }
  },
}]
```

- **Step 4**: Now, we have all the information to create a `hbase_replication_process` instance. This will also be an instance of type `hbase_replication_process`. Since this is simply an Entity of type `hbase_replication_process`, it will have the same form as an `EntityDefinition` structure:

```
{

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Reference",
  "id":{

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Id",
    "id":"-1467009423967137000",
    "version":0,
    "typeName":"hbase_replication_process",
    "state":"ACTIVE"
  },
  "typeName":"hbase_replication_process",
  "values":{
    "name":"Replication:
default.webtable@cluster1->default.webtable@cluster2",
    "outputs":[
      {

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Id",
        "id":"6e86cdc6-a136-459f-9231-3eeae60ec56a",
        "version":0,
        "typeName":"hbase_table",
        "state":"ACTIVE"
      }
    ],
    "replicationSchedule":"daily",
    "replicationEnabled":true,
    "inputs":[
      {

"jsonClass":"org.apache.atlas.typesystem.json.InstanceSerialization$_
Id",
        "id":"6f89adfb-b29b-45d4-bd7c-0044095a2dde",
        "version":0,
        "typeName":"hbase_table",
        "state":"ACTIVE"
      }
    ],
    "qualifiedName":"Replication:
default.webtable@cluster1->default.webtable@cluster2"
  },
  "traitNames":[

  ],
  "traits":{

  }
}
```

Note how we have

- Given the `inputs` and `outputs` attribute to be an ID referring to the `hbase_table`. Since these are not composite attributes, we have to pre-create these are described in Step 3 above and use only the ID portion of the structure. For details, refer to the `EntityDefinition` structure in “Important Atlas API Datatypes”
- Given other attribute values for attributes of the `hbase_replication_process` type like `qualifiedName, name, replicationSchedule and replicationEnabled`.

Retrieve lineage for a dataset

By using the GUID or unique attribute qualifiedName of a Process, one can get the details of the inputs and outputs it refers to. However, Atlas provides a richer API to retrieve a more complete lineage information. We shall describe that API in this section.

Take an example of a data pipeline where a RawHiveTable is transformed into an AggregateHiveTable, and the AggregateHiveTable is loaded into a OracleSummaryTable using Sqoop. A good way to capture this lineage would be:

`RawHiveTable ->` **`AggregationQuery`** `-> AggregateHiveTable ->`
`SqoopProcess` `-> OracleSummaryTable`

In the above lineage 'graph', the bold entries are entities of the Process class and the others are entities of 'DataSets'. Querying for a Process entity will only give one level of input and output process.

For e.g. Querying on the SqoopProcess entity ID, will only give this view:

`AggregateHiveTable ->` **`SqoopProcess`** `-> OracleSummaryTable`

And querying on the AggregationQuery entity ID will only give this view:

`RawHiveTable ->` **`AggregationQuery`** `-> AggregateHiveTable`
However, it would be ideal to see the entire lineage graph at one shot, where the first level of inputs / outputs are recursively expanded to their inputs and outputs and so on, until we see the very beginning set of inputs and the very end set of outputs. Atlas provides a Lineage API for just such a requirement.

Request

```
GET
http://<atlas-server-host:port>/api/atlas/lineage/{guid}/inputs/graph
```

And

```
GET
http://<atlas-server-host:port>/api/atlas/lineage/{guid}/outputs/grap
h
```

The GUID to be used here is the GUID of a DataSet entity.

Response

The Response is a structure are follows:

```
{
  "requestId": string_value,
  "results": {
    "jsonClass":
"org.apache.atlas.typesystem.json.InstanceSerialization$_Struct",
    "typeName": string_value,
    "values": {
      "vertices": {
        guid_1: {
          "jsonClass":
"org.apache.atlas.typesystem.json.InstanceSerialization$_Struct",
          "typeName": string_value,
          "values": {
            "vertexId": {
              "jsonClass":
"org.apache.atlas.typesystem.json.InstanceSerialization$_Struct",
              "typeName": "__IdType",
              "values": {
                "guid": guid_1,
                "typeName": data_set_type
              }
            },
            "name": data_set_name
          }
        },
        guid_2: {
           Same as above
        }
      },
      "edges": {
        guid_1: [array of guids],
        guid_2: [array of guids],
       ...
      }
    }
  }
}
```

This structure is a little complicated, but we can try and understand this structure with the information below:

- For an `inputs` lineage query, the return JSON contains the GUIDs of every `DataSet` and every `Process` entity that contributed to the input of the GUID in the request, and the GUIDs of the datasets and processes that contributed to the inputs of those GUIDs recursively.
- For an `outputs` query, the return JSON contains the GUIDs of every `DataSet` and every `Process` entity to which the GUID in the request was one of the inputs, and the GUIDs of the datasets and the processes that were in turn generated by those GUIDs recursively.

How this information is arranged is what we need to see, that is described below:

The details of this structure are as follows (only the important attributes are explained)

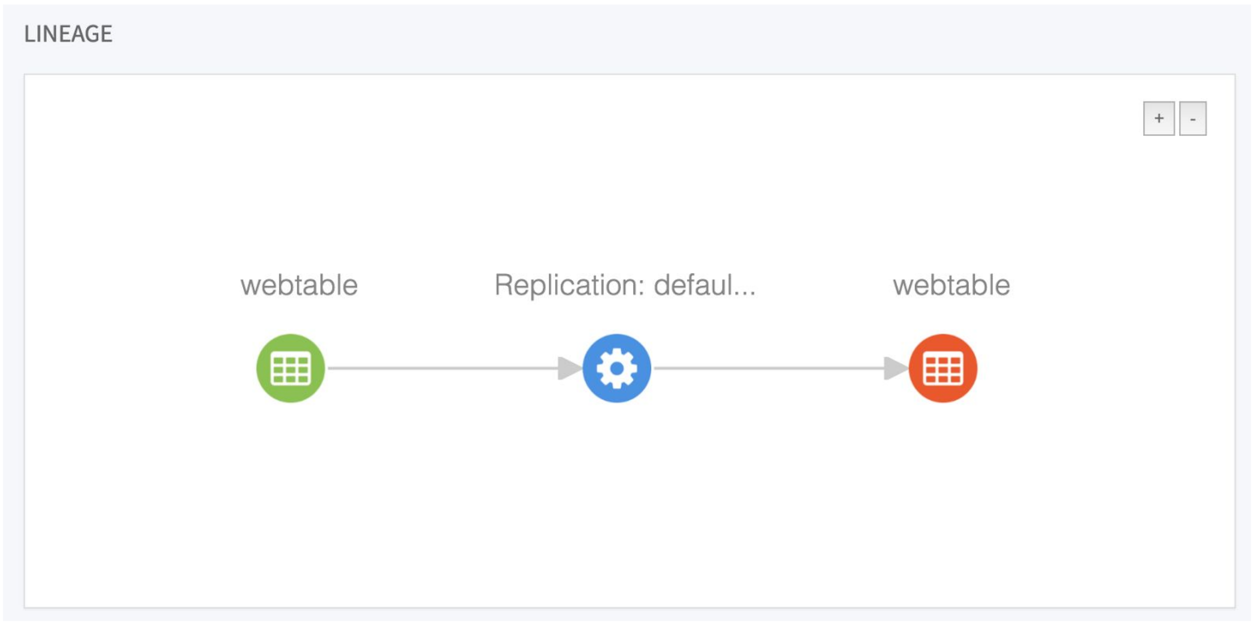
- `results`: contains the information JSON we need to interpret
- `results.values`: Contains two important structures as below
    - `vertices`: A Map of key-structure pairs (with as many entries as there are `DataSets` in the lineage graph), where
        - Key is a GUID of a DataSet
        - Value is a structure with the following important attributes:
            - `values.vertexId.name`: Name of the DataSet that can be used for display
            - `values.vertexId.values.typeName`: Type name of the DataSet that can be used for display.
    - `edges`: A Map of key-array pairs, where
        - Key is a GUID of a either a `DataSet` or `Process`
        - Value is an array of GUIDs that are:
            - For `inputs` lineage query: The incoming edges to the vertex denoted by the Key
            - For `outputs` lineage query: The outgoing edges to the vertex denoted by the Key

To those familiar with a Graph data structure, the `edges` structure is a typical “Adjacency list” of a graph. By following the adjacency list, one can plot the structure of the graph easily.

All GUIDs in the edges that are also present in the vertices structure are `DataSets`. The others are `Processes`. By issuing a GET request on those, one can get details about the `Process` information as well.

Example

For the HBase replication example, Atlas Admin UI shows a lineage graph as below:



This lineage graph is shown when browsing the entity with GUID: 6e86cdc6-a136-459f-9231-3eeae60ec56a, which points to the HBase table `default.webtable@cluster2`.

So, the example requests and responses are as given below. Note some of the extraneous information is omitted for brevity.

Request:

```
GET
http://<atlas-server-host:port>/api/atlas/lineage/6e86cdc6-a136-459f-
9231-3eeae60ec56a/inputs/graph
```

Response:

```
{
  "requestId": "...",
  "results": {
    "jsonClass": "...",
    "typeName": "...",
    "values": {
      "vertices": {
        "6f89adfb-b29b-45d4-bd7c-0044095a2dde": {
          "jsonClass": "...",
          "typeName": "...",
          "values": {
            "vertexId": {
              "jsonClass": "...",
              "typeName": "__IdType",
              "values": {
                "guid": "6f89adfb-b29b-45d4-bd7c-0044095a2dde",
                "typeName": "hbase_table"
              }
            },
            "name": "webtable"
          }
        },
        "6e86cdc6-a136-459f-9231-3eeae60ec56a": {
          "jsonClass": "...",
          "typeName": "...",
          "values": {
            "vertexId": {
              "jsonClass": "...",
              "typeName": "__IdType",
              "values": {
                "guid": "6e86cdc6-a136-459f-9231-3eeae60ec56a",
                "typeName": "hbase_table"
              }
            },
            "name": "webtable"
          }
        }
      },
      "edges": {
        "6e86cdc6-a136-459f-9231-3eeae60ec56a":
["230d8a60-fceb-4780-8988-58f7b0615f1e"],
        "230d8a60-fceb-4780-8988-58f7b0615f1e":
["6f89adfb-b29b-45d4-bd7c-0044095a2dde"]
      }
    }
  }
}
```

Since 6e86cdc6-a136-459f-9231-3eeae60ec56a and 6f89adfb-b29b-45d4-bd7c-0044095a2dde are present in the vertices list and thus are DataSets of type `hbase_table`. The GUID 230d8a60-fceb-4780-8988-58f7b0615f1e is a process.

Request:

```
GET
http://<atlas-server-host:port>/api/atlas/lineage/6e86cdc6-a136-459f-
9231-3eeae60ec56a/outputs/graph
```

Response:

```
{
  "requestId": "...",
  "results": {
    "jsonClass": "...",
    "typeName": "...",
    "values": {
      "vertices": {},
      "edges": {}
    }
  }
}
```

Since the HBase table `default.webtable@cluster2` contributes to inputs of no other data sets this is an empty response.

Deleting an entity

Atlas supports the deletion of entities. By default, the delete is a “**soft delete**” in the sense that it is not really deleted from the store, but its state is marked as “DELETED”.

Request

```
DELETE http://<atlas-server-host:port>/api/atlas/entities?guid={guid}
```

Response

The response to a DELETE request contains the following fields:

- **`entities`**: A structure that contains information about entities that are deleted.
    - **`deleted`**: An array of GUIDs of entities that were deleted. Note that this can be more than one even if only one GUID is sent for deletion. This happens in case the entity has composite attributes. Then, the entities that are composite attributes are deleted along with the entity being deleted itself.
- **`definition`**: The `EntityDefinition` structure of the entity whose GUID is specified in the request. The `state` attribute in the ID structure for every deleted entity will be defined as `DELETED`.

Example

Suppose we want to delete the HBase table, with GUID
91e2c596-703e-4980-b9b1-27f8d0b6dc89

The request will be as follows:

```
DELETE
http://<atlas-server-host:port>/api/atlas/entities?guid=91e2c596-703e
-4980-b9b1-27f8d0b6dc89
```

The response body will be as follows. The definition has been truncated for brevity. It actually will be the complete `EntityDefinition` of the `hbase_table` entity.

```
{
  "requestId": "qtp221036634-16 -
f7cac02b-b9c6-44a7-a9d9-001bb3089cb2",
  "entities": {
    "deleted": [
      "91e2c596-703e-4980-b9b1-27f8d0b6dc89",
      "bfa078d9-b6fe-4edc-be48-360f28eff338",
      "16919e52-69e5-4eee-a266-8b8c7fd27a0b"
    ]
  },
  "definition": {
    "jsonClass":
"org.apache.atlas.typesystem.json.InstanceSerialization$_Reference",
    "id": {
      "jsonClass":
"org.apache.atlas.typesystem.json.InstanceSerialization$_Id",
      "id": "91e2c596-703e-4980-b9b1-27f8d0b6dc89",
      "version": 0,
      "typeName": "hbase_table",
      "state": "DELETED"
    },
    "typeName": "hbase_table",
    "values": {
      "columnFamilies": [
        {
          "jsonClass":
"org.apache.atlas.typesystem.json.InstanceSerialization$_Reference",
          "id": {
            "jsonClass":
"org.apache.atlas.typesystem.json.InstanceSerialization$_Id",
            "id": "bfa078d9-b6fe-4edc-be48-360f28eff338",
            "version": 0,
            "typeName": "hbase_column_family",
            "state": "DELETED"
          },
         ...
        },
        {
          "jsonClass":
"org.apache.atlas.typesystem.json.InstanceSerialization$_Reference",
          "id": {
            "jsonClass":
"org.apache.atlas.typesystem.json.InstanceSerialization$_Id",
            "id": "16919e52-69e5-4eee-a266-8b8c7fd27a0b",
            "version": 0,
            "typeName": "hbase_column_family",
            "state": "DELETED"
          },
          ...
        }
      ],
      "name": "webtable",
      ...
}
```
Note how the `entities.deleted` attribute contains 3 GUIDs. These are the GUIDs corresponding to the deleted HBase table and the 2 HBase column families it has defined as composite attributes.

Note also how the `state` attribute of the deleted entities comes with value `DELETED`.

Cataloging Metadata: Traits / Tags and Business Taxonomy

Purpose of cataloging

As seen in previous sections, metadata is added to Atlas as entities (instances) of types (model definitions). Typically, the models are defined by whoever understands the metadata best. For e.g. the Hive data types are typically defined by someone who understands hive types best.

However, more effective usage of this metadata for purposes of data discovery and governance comes when it can be annotated and cataloged according to concepts that are closer to the business terminology and processes, rather than just technical metadata. This can be performed by data stewards or data scientists who act as a bridge between the technical metadata and the business concepts defined.

Another advantage of having this capability is that metadata that is similar from a business standpoint, but different from a technical management standpoint, can be cataloged using similar business terminology. By doing so, applications can be built that apply the same governance policies to similar metadata irrespective of their sources of origin or who has modelled it. Also, using the search capabilities of Atlas, all similar business metadata can be found very easily.

For instance, in a 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 cataloged similarly, Credit related policies can be applied to all data assets cataloged with this concept.

Atlas provides two ways of cataloging metadata: Traits / Tags and Business Taxonomy. Loosely speaking, while the former is 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, controlled vocabulary that has specific meaning in a domain and is uniformly understood within a certain context.

This section will describe ways of defining these concepts and using them to catalog metadata. In the next section, when we look at searching metadata in Atlas, we shall also see ways of searching using these concepts.

Traits

Firstly, let’s get some terminology out of the way: Traits and Tags are used interchangeably in Atlas to mean the same thing. This document will stick to using Traits, as that is how the APIs define them.

Traits were introduced in the section on “Types” 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 for now.

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 as a consequence, they cannot be referenced from attributes in other types. So, how a trait instance is defined and used becomes different from how an entity is defined and used.

Trait instances also have one other special significance in Atlas. They can be associated to any entity in Atlas without prior declaration of this fact in the Type definition of the entity. Note that, in contrast, to define an entity reference in a type, this has to be declared a priori (for e.g. HBase table refers to HBase namespace should be declared upfront). The Atlas type system recognises traits specially and has specific APIs to associate traits to entities. We shall look at these APIs in the following sections.

Business Taxonomy

Business Taxonomy is a hierarchical collection of objects called “**Terms**”. Each “Term” has two important attributes, name and description. Terms can be defined either under a predefined Taxonomy object, or under another Term. The predefined Taxonomy object is named “**Catalog**”. This will be used in the APIs related to Business Taxonomy.

A Term, once created, can be associated to any entity in Atlas. This is similar to how Traits instances can be associated to any entity. In fact, beneath the hood, Terms are implemented using the Traits APIs. However, this is a detail that users MUST NOT use, as it is subject to change in future. Also unlike Traits, Terms can be deleted.

There are a different set of APIs that users should use to create terms, list them, associate them to entities and so on. These are defined after the APIs for Traits are completed.

Creating Traits

Since Traits are types in Atlas, the APIs to create a Trait are the same as the APIs to create Types, except that Attribute definitions cannot refer to non-native metatypes.

Request

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

Body

The Body is the same structure as the `TypesDef` defined in the section "Important Atlas API Datatypes". The Traits should be defined under the `traitTypes` attribute.

Response

The response is the same as the response for a Type Definition request which will contain the trait names of the traits that have been defined.

Example

For our running example, let us say we want to define two Traits:

- **`PublicData`**: Any metadata marked with this means that this data was collected from publicly available sources. Hence, any policies applicable to publicly collected data can be applied to all such data.
- **`Retainable`**: This Trait indicates that any metadata associated with this trait should be retained for some period of time. The periodicity is maintained in an attribute called `retentionPeriod` which is a duration in days.

Body for POST request:

```
{
  "enumTypes":[],
  "structTypes":[],
  "traitTypes":[
    {
      "superTypes":[],

"hierarchicalMetaTypeName":"org.apache.atlas.typesystem.types.TraitTy
pe",
      "typeName":"PublicData",
      "typeDescription":null,
      "attributeDefinitions":[]
    },
    {
      "superTypes":[],
"hierarchicalMetaTypeName":"org.apache.atlas.typesystem.types.TraitTy
pe",
      "typeName":"Retainable",
      "typeDescription":null,
      "attributeDefinitions":[
        {
          "name":"retentionPeriod",
          "dataTypeName":"int",
          "multiplicity":"required",
          "isComposite":false,
          "isUnique":false,
          "isIndexable":true,
          "reverseAttributeName":null
        }
      ]
    }
  ],
  "classTypes":[]
}
```

Note how the `traitTypes` attribute is filled with the Traits we are defining. The rest of the metatypes - structs, enums, classes are empty. The attribute `retentionPeriod` is defined for the Retainable Trait as an int.

Response BODY:

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

Listing Traits

Since Traits are a specific metatype like Classes, the same API to listing a specific metatype can be used to list Traits:

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

The response to this API is a list of Trait names as described in the section on “Listing all types”.

Example

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

Retrieving a Trait

Since Traits are a specific metatype like Classes, the same API to retrieve a specific metatype can be used to retrieve a Trait:

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

The response is the same `TypesDef` structure defined in “Important Atlas API Datatypes”. The `traitTypes` attribute will be filled in the response with the type definition of the Trait requested for.

Example

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

Response Body:

```
{
  "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"
}
```

Associating Trait Instances to Entities

To catalog entities using Traits, we need to associate an entity with an instance of a Trait definition. To do this, Atlas provides APIs in the `/entities` resource.

Request

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

Body

The Body is a Trait InstanceDefinition structure that is defined in “Important Atlas API Datatypes”.

Response

No data is returned as response. A status of 201 indicates success.

Example

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

The following are the requests to send:

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

Body

```
{

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

  }
}
```

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

Body

```
{

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

Reading Trait Instances associated to entities

When Trait Instances are associated to entities, the attributes `traitNames` and `traits` are filled up according to the description in the structure `EntityDefinition` described in "Important Atlas API Datatypes".

Example

Let us get the Entity definition for the HBase table to which we associated the trait instances. Only the traitNames and traits values are shown below.

Request

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

Response

```
{
    ...
    "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": {}
      }
    }
  }
}
```

Disassociating Trait Instances Associated To Entities

This is a simple DELETE operation as follows.

Request

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

Response

No data is returned as response.

Example

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

Creating Terms

To create a new term, one has to give a name and an optional description. One must also determine where this new term will occur in the Business Taxonomy hierarchy.

Request - Create term under Catalog

To create a Term directly under the Business Taxonomy "Catalog":

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

Body

Body contains a simple single element name: `{"description":string}`

Response

Response contains a map as follows:

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

The “href” contains the resource URL for the newly created term.

Request - Create Term under another Term

To create a Term under another term, first we need to know the URL of the created term. This will be a recursive URL of form

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

Then the request is

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

Response is same as before.

Example

Request:

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

Body:

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

Response:

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

Request for Term under a Term

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

Retrieving a Term definition

To retrieve a specific Term, use the following API

Request

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

Response

The response is a structure as follows:

```
[
  {
    "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
    }
  }
]
```

The fields have the following meaning:

- **`href`**: The URL to refer to this Term resource
- **`name`**: A fully qualified name of this Term. By fully qualified, we mean the entire path starting from the Business Taxonomy name, all intermediate parent terms leading upto this term. These components are separated by a '.' character.
- **`description`**: The description provided when defining the term.
- Other properties are system defined properties like creation time of term etc.
- **`terms`**: This contains a single element href that refers to the URL to use to fetch the terms under this term.

Example

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

Response

```
[
  {
    "href":
"http://localhost:21000/api/atlas/v1/taxonomies/Catalog/terms/term1/t
erms/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/t
erms/term11/terms"
    }
  }
]
```

Listing the Term hierarchy

Request - Listing all terms under the Catalog Taxonomy

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

Request - Listing all terms under a given Term

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

Response

The response in both cases is an array of results:

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

Each element in the above Array is a descendant in the hierarchy of business terms rooted under the Catalog (in the first example) or terminal term_name (in the second example). The fully qualified name of the term is as described in the section "Retrieving a Term Definition".

Example

Request

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

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

Associating A Term to Entity

To associate a term to an entity, we need the GUID of the entity, just like with Associating Trait instances.

Request

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

Body is an empty map.

Response

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

Example

```
POST
http://<atlas-server-host:port>/api/atlas/v1/entities/f4019a65-8948-4
6f1-afcf-545baa2df99f/tags/d.term1.term12
```

Disassociating A Term from an entity

To disassociate a term to an entity, we need the GUID of the entity, just like with disassociating Trait instances.

Request

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

Response

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

Example

```
DELETE
http://<atlas-server-host:port>/api/atlas/v1/entities/f4019a65-8948-4
6f1-afcf-545baa2df99f/tags/d.term1.term12
```

Deleting a Term

Deleting a Term removes the Term from the Catalog Business Taxonomy and also removes all associations the Term might have had to any entities thus far. In addition, deleting a Term has a recursive effect in that it deletes all the terms are that under its hierarchy. Note that given the depth of the hierarchy and the number of associations the term and its sub-terms have, this could be an expensive operation.

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

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

Updating a Term

Currently, the only attribute that can be updated for a Term is the description. Updating the description of a Term updates this property in all associated Terms as well.

Request

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

Body

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

Response

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

Example

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

Body

```
{"description":”New description”}
```

Discovering metadata - the Atlas Search API

In the previous sections, we saw how to add metadata to Atlas and how to catalog it using traits and business catalog terms. We also saw APIs to retrieve a particular metadata entity by looking up by its GUID or a unique attribute that we defined for it.

But when a lot of metadata is added to Atlas, it is inconvenient or impossible to remember all the unique attribute values. The ability to search is imperative for any system that deals with lot of metadata. Atlas provides the following ways of looking up metadata:

A Search Domain Specific Language (DSL): Atlas defines a SQL like domain specific language that uses the type and attribute names defined for metadata. This DSL query can be given to a Search API. Internally, the query is translated to a Graph lookup query using a language called Gremlin and fired against the metadata store. The results are again translated into the entity and type system objects and returned back to users.

The DSL search is useful if one is aware of the specific metadata model (type names, attributes, etc) of the entities they want to retrieve. This generally results in very specific search queries and relevant results. Using the type system APIs (listing types, retrieving a type definition), one can learn the model of the entity and then use the DSL for searching amongst entities of that type.

A full text search: When entities are added to Atlas, it indexes the attribute values of entities defined using a Search indexing system (Solr). These indexed attributes can be searched to retrieve entities using a full text search capability. The Search API can be used to fire either a full text search or DSL search as the case may be.

The full text search is useful if one is not aware of the metadata model, or one wants to query across different models (types). For e.g. one wants to find all data assets that have to do with customer data, irrespective of the storage used for the same (Hive, HBase etc). However, since full text search is based on an index that is not aware of the type / model information, the results are likely to be broader in nature.

Catalog based search: Atlas empowers data stewards to make data more discoverable by annotating metadata entities with Traits / Tags and Business Catalog Terms. The DSL provides a specific way to search metadata using these annotations. This results in highly relevant discovery capabilities, provided the metadata is annotated correctly.

Using any of the above mechanisms, the search APIs allow users to retrieve entities using user friendly queries and then drill down into details of the retrieved results. In the following sections, we will understand the DSL query syntax and then look at how to use them with the search API.

DSL based search

The full grammar for the DSL is documented in the search Wiki page of Apache Atlas documentation. In this section, we will take a more example driven approach for understanding the syntax and also capabilities of the language. The syntax used will not be as exact as the grammar defined in the link, but hopefully will be easy enough to explain the key concepts.

To try these out, one can first use the Admin UI > Search tab. Remember to select the search type to DSL.

Discovering entities by attribute values

`type_name` **where** `attribute_name` *OP* `attribute_value`

- `type_name` is name of a predefined type
- `attribute_name` is a name of an attribute in that type. This need not be a unique attribute, unlike in the APIs we saw in “Retrieving an entity definition by unique attribute”.
- *OP* is an operator including ‘=’, ‘!=’, ‘<’, ‘>’, ‘<=’, ‘>=’
- `attribute_value` is value of attribute.

Results are entire entity definitions matching the search criteria.

Examples

- **hbase_table where name = ‘webtable’**
- **hbase_column_family where name != ‘contents’**
- **hbase_column_family where versions > 1**
- **hbase_column_family where blockSize < 1000**

Discovering entities by combinations of attribute values

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

- `type_name` is name of a predefined type
- `attribute_name` is a name of an attribute in that type. This need not be a unique attribute, unlike in the APIs we saw in “Retrieving an entity definition by unique attribute”.
- *OP* is an operator including ‘=’, ‘!=’, ‘<’, ‘>’, ‘<=’, ‘>=’
- *AND_OR_OP* is ‘and’ or ‘or’
- `attribute_value` is value of attribute.

Note the following:

- It is possible to provide any number of expressions of form `attribute_name` *OP* `attribute_value` combining them with an ‘and’ or ‘or’
- It is also possible to impose an ordering of evaluation by enclosing the expressions within parentheses like this

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

Results are entire entity definitions matching the search criteria.

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)**

Selecting native attributes in searches

As described above, the results of search queries return the entire entity definitions matching the search criteria. Sometimes we want to just fetch some attributes of the selected results. In such cases, a SELECT clause can be used in the search expression.

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

- `search_expression` is one of the forms of expression described in sections above
- `attribute_name` list: The attributes we want to select for matched results

Examples

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

Selecting references in searches

The previous section showed how we can select any attributes which are of native types. But remember that attributes can be of more complex types like collections, references to other entities etc. For e.g. `hbase_tables` contain `columnFamilies` which are references to entities of type `hbase_column_family`. To help with this, the DSL allows search queries to be combined as below:

```
search_expression, reference_attribute_name
```

- `search_expression` is one of the forms of expressions described above.
- `reference_attribute_name` is an attribute name in the entity being selected in `search_expression` which has references to other attributes.

A variation to the above is where the reference_attribute_name can be expanded to just select specific attributes of the reference_attribute's type, as below:

```
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, as below:

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

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[]'**

Search API

Atlas uses a search resource where the queries described above can be included.

Request - queries which return entities as results

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

The dsl_query_string should be encoded using normal URL encoding criteria.

Response

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

A description of the most important attributes:

- **`query`**: This is the query that was fired in the search API. This is the unencoded version.
- **`queryType`**: whether "`dsl`" or "`fulltext`" search was used for the results.
- **`count`**: Number of results returned.
- **`results`**: An array of search results. Each search result follows the `EntityDefinition` structure defined in the "Important Atlas API Datatypes" with just a few minor 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 type of the search result. The attribute definitions of the TypesDef is not complete.

Example

Request

Say the query is **hbase_column_family, columns where type='byte[]',** the request is

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

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
            }]
      }
}
```

Since the results for the query are hbase_column instances, we see that the entity definition matches that of a hbase_column.

Request - queries which select specific attributes

Same as above

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
                  }
}
```

The response is very similar as well, with the following notable differences:

- The result elements will only contain the attribute names and values that are selected in the SELECT clause of the query.
- The result will *not* include the GUID or any attribute that is not specified in the select clause.
- The dataType will include the attribute definitions only for the selected attributes of the result.

Example

Say the query is **hbase_table where name='webtable' select name, qualifiedName, isEnabled**

Request:

```
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
```

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 how the response result set contains only the attributes `name, qualifiedName and isEnabled`. Also note how the attribute definitions are defined only for these 3 attributes.

Full text search

As described in the introductory section on "Discovering metadata", Atlas indexes attribute values when metadata entities are added to it. The index will map the text value to the entity GUID that the attribute belongs to. Due to this, we can lookup queries using simple textual strings. These strings can be values of any of the attributes of any entities added to Atlas.

Request

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

The query_string should be encoded using normal 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
      }, ...]
}
```

- **`queryType`**: set to "full-text"
- **`results`**: Each result row contains the following:
  - **`guid`**: The ID of the entity which matched the search query.
  - **`typeName`**: The type of the entity which matched the search query.
  - **`score`**: The floating point score of how relevant the entity is to the search query. The higher the score, the more relevant the result is.

Example

Request

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

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”** contents tokens **“crawled”** that comes in both `hbase_column_family` attributes and `hbase_table`. However, since the “crawled content” comes as a sub string in the description for hbase_column_family, it has a lot more score than hbase_table results.

Searching entities associated to Traits / Terms

In the section on “Cataloging Metadata”, we saw how Trait instances and Business Taxonomy terms can be associated to entities. One of the purposes of doing this is to annotate these entities with additional information that the Trait or Term conveys. Once the association is made, we can also search for entities associated with a given Trait or Term to aid discovery.

The main advantage of using Trait / Term assisted discovery is that it can give precise results, but need not be restricted to a specific type (as required by DSL search).

For example, in our case, we can search which among all assets in our metadata repository, including HBase tables, contain content filled from crawling public data, using the Trait PublicData.

The Trait / Term search is a special form of DSL search as described below.

Searching among all entities

Request

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

The %60 encoding stands for the back-tick character "`".

For Terms, one should use the fully qualified term name as defined in the section "Retrieving a Term definition"

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 or term,
                "typeName": type of entity associated to trait or
term
           },
           "traitDetails": null
      }, ....],
      "dataType": {
           "typeName": "__tempQueryResultStruct...",
           "typeDescription": null,
           "attributeDefinitions": [{
                "name": "traitDetails",
                "dataTypeName": trait or term 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
            }]
        }
}
```

As can be seen, the search result structure is very similar to other types of searches. The differences are as below:

- Each result element contains an instanceInfo map, which in turn contains the following attributes:
    - **`guid`**: contains GUID of the entity that has the term or trait associated
    - **`typeName`**: contains the type of the entity that has the term or trait associated.
- The **`dataType`** structure contains definitions about the **`traitDetails`** and **`instanceInfo`** attributes.

Example

Request

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

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
		}]
```

```
        }
}
```

Searching among a specific type

Sometimes we may want to restrict the search to only a given type of assets. This can be done by using a special DSL operator called **isa**.

Request

DSL Query: `type_name` **isa** `` `tag_or_trait_name` ``

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

Response

Response is the same as the search response documented under the section “Search API” above.

Example

Request

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

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
            }]
        }
}
```

Messaging Integration with Atlas

So far we have covered integrating with Atlas using the RESTful APIs it exposes. In the Architecture section, we mentioned another method of integration, using a “Messaging” interface.

The messaging interface uses Apache Kafka as the medium through which messages are passed to / from Atlas. Apache Kafka is a scalable, reliable, high performance messaging system. Therefore it forms an ideal system for integrating between metadata sources that generate high volume of metadata events and Atlas.

Kafka is also a durable store of messages. Hence, 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 very loosely coupled integration and thus are generally more reliable in a distributed architecture. This is also true for consumers of metadata change events that Atlas communicates via Kafka.

While the durability offers safe guarantees, when Atlas is active along with the other metadata sources and consumers, this allows for real time handling of metadata events both to and from Atlas.

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

- **Publishing entity changes to Atlas**: These messages are passed from the metadata sources where the metadata is originally created / updated or deleted to Atlas. These messages are written to a topic called **ATLAS_HOOK.** Typically, these metadata sources are other components in the Hadoop ecosystem. As of Atlas 0.7-incubating, there are integrations with Hive, Sqoop, Falcon and Storm with Atlas.
- **Consuming entity changes from Atlas:** These messages are passed from Atlas to external consumers who might be interested in changes to metadata. An example of such a source in the current Hadoop ecosystem is Apache Ranger. By capturing metadata change events in real time, Ranger provides policy driven security

management of Hadoop data assets. These messages are written to a topic called **ATLAS_ENTITIES.**

Note that the Messaging integration is restricted to Entity notifications. Type related changes are still handled via the API layer. This is all right because types are created much less frequently than entities are.

In the following sections, we shall see the formats for messages that are written to ATLAS_HOOK and ATLAS_ENTITIES.

Publishing Entity Changes to Atlas

Metadata sources can communicate the following forms of entity changes to Atlas: creation, updates, deletions of entities. These messages are called `HookNotification` messages in the Atlas source code. There are four types of these messages described below. The sources can publish these messages to the ATLAS_HOOK topic and 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.

ENTITY_CREATE message

ENTITY_CREATE notification messages are used to add one or more entities to the Atlas system. The structure of an ENTITY_CREATE message is:

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

The definition of the attributes is as follows:

- **`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 version number supported 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 the “Important Atlas API Datatypes” section below.
    - **`type`**: The type of the message will be **`ENTITY_CREATE`** for this message type.
    - **`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 that of a `hbase_namespace` message that is being added to Atlas. Note that it is a single element array in this case, and the element structure matches the entity definition of a 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": "-1467290565135246000",
                        "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"
      }
}
```

ENTITY_FULL_UPDATE message

There is one important difference between API and message mode of communication. While the former is two way communication and allows Atlas to communicate back results to the caller, the latter is one way and there is no notification that comes from Atlas to the system generating the messages.

Consider how we added hbase_table entities using API. When referring to the hbase_namespace a table belongs to, we could use the GUID of the hbase_namespace entity. We could retrieve this GUID either by using the return value from a create request or by looking it up using a query. Both 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. The other option of remembering such state information complicates metadata sources.

To help us out of this situation, Atlas allows an update called ENTITY_FULL_UPDATE, where one can give an entity definition in full, but mark this as an update request. Atlas uses the unique attribute definition of this entity to see if this entity already exists in the metadata store. If it does, this entity's attributes are updated with values from the request. Else, they are created.

Thus, to add a hbase_table and refer to an hbase_namespace, we need not remember the GUID of the added hbase_namespace or fetch it using APIs. We can simply include all of these entity definitions in a ENTITY_FULL_UPDATE message and Atlas would handle 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 exactly the same as the ENTITY_CREATE structure, except the type is ENTITY_FULL_UPDATE.

Example

Let us now create a hbase_table instance along with column family entities and column entities. To refer to the namespace, let us include the hbase_namespace entity again in the array of entities. The structure is given below, but details of all columns, column families etc are omitted for brevity. They follow the same structure as described in the section “Creating new entities” of APIs.

```
{
      "version": {
            "version": "1.0.0"
      },
      "message": {
            "entities": [{
                  "jsonClass":
"org.apache.atlas.typesystem.json.InstanceSerialization$_Reference",
                  "id": {
                        "id": "-1467290566519456000",
                                                  ...
                  },
                  "typeName": "hbase_namespace",
                  "values": {
                        "qualifiedName": "default@cluster3",
                                                  ...
                  },
                  "traitNames": [],
                  "traits": {}
            }, {
                  "jsonClass":
"org.apache.atlas.typesystem.json.InstanceSerialization$_Reference",
                  "id": {
                        "id": "-1467290566519491000",
                                                  ...
                  },
            }, …, {
                  "jsonClass":
"org.apache.atlas.typesystem.json.InstanceSerialization$_Reference",
                  "id": {
                        "id": "-1467290566519615000",
                                                      ...
                  },
                  "typeName": "hbase_table",
                  "values": {
                        "qualifiedName": "default.webtable@cluster3",
                                                      ...
                        "namespace": {
                              "jsonClass":
"org.apache.atlas.typesystem.json.InstanceSerialization$_Id",
                              "id": "-1467290566519456000",
                              "version": 0,
                              "typeName": "hbase_namespace",
                              "state": "ACTIVE"
                        }
                  },
                  "traitNames": [],
                  "traits": {}
            }],
            "type": "ENTITY_FULL_UPDATE",
            "user": "integ_user"
      }
}
```

The highlighted sections in the example are used to demonstrate the following points.

- 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@cluster3 will be what Atlas uses to lookup the namespace entity for updation because it is defined as the unique attribute for the `hbase_namespace` type.

ENTITY_PARTIAL_UPDATE message

When we know that the entity being updated has already been added to Atlas, we can send a partial update message. This has a structure as follows:

```
{
      "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. The differences are as follows:

- **`typeName:`** The name of the type being updated
- **`attribute:`** This should be the name of the attribute that is unique for the type of the entity being updated.
- **`attributeValue`**: This is the value of the unique attribute
- **`entity`**: This is a partial `EntityDefinition` structure, with only the following fields:
    - **`id`**: This is a typical ID structure as seen in `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 are being updated along with the new values.

Using the `typeName, attribute and attributeValue`, Atlas can find the entity that needs to be updated. It will be clear that these parameters are similar to the parameters of the API seen in the section “Updating an attribute of an entity”.

Example

Suppose we want to update the `hbase_table` entity with `qualifiedName default.webtable@cluster3`, and set the `isEnabled` attribute to false, we can add a 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"
      }
}
```

ENTITY_DELETE message

The last message in the list of messages is a way to delete an entity found in Atlas. The structure for this is as follows:

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

This structure is a subset of the ENTITY_PARTIAL_UPDATE structure.

- **`typeName`**: Name of the type of the entity being deleted.
- **`attribute`**: Name of the unique attribute of type being deleted.
- **`attributeValue`**: Value of the unique attribute of type being deleted.

Note that these 3 attributes form the key through which Atlas can identify the entity to delete, similar to how it can find the entity to do a partial update.

Example

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

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

Appendix

Important Atlas API Datatypes

TypesDef

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

- API to create types
- API to list a type

The `TypesDef` structure has the following attributes:

- **`enumTypes`**: An array of types of metatype Enum. Will be empty if no Enums are being queried or defined
- **`structTypes`**: An array of types of metatype Struct. Will be empty if no Structs are being queried or defined.
- **`classTypes`**: An array of types of metatype Class. Will be empty if no Classes are being queried or defined.
- **`traitTypes`**: An array of types of metatype Trait. Will be empty if no Traits are being queried or defined.
- For each of the Struct, Class or Trait types defined, the following attributes will be found:
    - **`typeName`**: Name of the specific Class/Struct/Trait being defined
    - **`typeDescription`**: Description of the specific Class/Struct/Trait if the type has one.
    - **`hierarchicalMetaTypeName:`** If the type being defined is a Class or Trait, this will be the class name of the Metatype, either `org.apache.atlas.typesystem.types.TraitType` or `org.apache.atlas.typesystem.types.ClassType`
    - **`superTypes:`** If the type being defined is a Class or Trait, this will be an array of Strings, each for a super class of the Class or Trait being defined.
    - **`attributeDefinitions:`** An array defining attributes that are part of the Struct, Class or Trait being defined. Each attributeDefinition will be as defined in the Attributes section above.

EntityDefinition

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

- API to create entities
- API to list an entity

The `EntityDefinition` has the following attributes:

- **`jsonClass`**: Refers to the internal Atlas class used to represent the entity object. Typically, it is `"org.apache.atlas.typesystem.json.InstanceSerialization$_Reference"`
- **`id`**: A Structure that refers to the system specific identifier details for this entity. The ID structure has the following attributes in turn:
    - **`jsonClass`**: Refers to the internal Atlas class used to represent the ID object. Typically, it is `"org.apache.atlas.typesystem.json.InstanceSerialization$_Id"`
    - **`id`**: The System specific Identifier.
        - When being used to create an entity, this ID value for the entity MUST be a negative long number. For e.g. 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. E.g."139b47b2-b911-47d4-b43c-0493607b4b89"
    - **`version`**: Always 0 (TODO: Check)
    - **`typeName`**: The name of the type of which this entity is an instance
    - **`state`**: One of ACTIVE (the entity is still live), or DELETED (the entity was deleted via an API. Any access to an entity MUST check this state value to determine if the entity should be considered as live or deleted. If required, deleted entities can be filtered using this.
- **`typeName`**: The name of the type of which this entity is an instance
- **`values`**: A JSON map of the attribute names and values for the attributes defined in the type definition of this 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: E.g. Int, String, Boolean etc. -` Encoding is corresponding string representation
  - `Enum metatypes: TODO`
  - `Collection metatypes: E.g. Arrays, Maps (TODO for maps)`
    - Arrays are encoded as a JSON array, where each element is recursively encoded according to these rules.
  - `Composite metatypes: E.g. Class, Struct, Trait. (TODO for Struct / Trait)`
    - For classes: If the system ID of the entity being referenced is known, then only the `jsonClass, id and typeName` attributes need 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 `EntityDefinition` should be specified.
- **`traitNames`**: Array of traits associated to the entity
- **`traits`**: A Map of String to trait instance definitions. Each entry has the key as the trait name and the value is a Trait InstanceDefinition.

Trait InstanceDefinition

The Trait InstanceDefinition is used in the following APIs:

- When associating a Trait instance to an entity
- In the `traits` Map of the EntityDefinition structure.

The structure is as follows:

```
{

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

The 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.