Defining a Data Source

Data Source Function List

A connector data source is an implementation of the abstract java class AbstractDataSource. When defining a data source the following functions require implementation:

- public abstract String getDataSourceName();
- public abstract Collection<AbstractDataSet> getDataSets();
- public abstract JDBCMetaData getDataSourceMetaData();
- public abstract boolean authenticate() throws Exception;
- public abstract void disconnect();
- public abstract Map<String, Object> testConnection() throws Exception

The following functions can optionally be overwritten:

- public ScheduleDefinition getScheduleDefinition();
- public boolean autoRun() { return true; };

The following functions are available as utility functions:

- protected final byte[] loadBlob(String key);
- protected final boolean saveBlob(String key, byte[] data);
- protected final boolean areBlobsAvailable();
- public final Object getAttribute(String key);
- public final Integer getSourceId();

Data Source Function Definitions

public abstract String getDataSourceName();

Return a DataSourceName as a String.

top

---

public abstract Collection<AbstractDataSet> getDataSets();

Return the collection of data sets that are available in this data source. See the Data Set section for defining a data set.

top

---

public abstract JDBCMetaData getDataSourceMetaData();

Return the connection metadata required for this data source. See the Metadata section for defining metadata for a data source.

top

---

public abstract boolean authenticate() throws Exception;

Return true or false depending on whether authentication against the data source was successful. This can return true if this is not required.

top

---

public abstract void disconnect();

disconnect() is called when the connection to the connector is closed. Perform any clean up here. This function can do nothing if it is not required.

top

---

public abstract Map<String, Object> testConnection() throws Exception

Return a map of text entries to be displayed on a successful connection test. The key to the Map is the description shown on the connection test within the Yellowfin UI.

An error message can be displayed if the connection test was not successful. The key to the Map in this case should be “ERROR”, with a description of the error stored in the Map value.

top

---

public ScheduleDefinition getScheduleDefinition();

Return a ScheduleDefinition for when the background task should run for this connector. A ScheduleDefinition is instantiated with:

```
public ScheduleDefinition(FrequencyTypeCode frequencyTypeCode, String frequencyCode, Integer frequencyUnit);
```

Each **frequencyTypeCode** is defined below. For any type that requires **n**, that value is defined in the frequencyUnit.

frequencyUnit	Description
MINUTES	Run every **n** minutes.
DAILY	Run every day.
WEEKLY	Run once a week, on the **n**th day of the week.
FORTNIGHTLY	Run once a fortnight, where frequencyCode is **ONE** or **TWO**, specifying the week in the fortnight, and the **n**th day of that week.
MONTHLY	Run once a month on the **n**th day of the month.
ENDOFMONTH	Run at the end of the month.
QUARTERLY	Run once a quarter, where frequencyCode is **ONE**, **TWO**, or **THREE**, specifying the month within the quarter, and the **n**th day of the month.

BIANNUAL	Run once every six months, where frequencyCode is **ONE**, **TWO**, **THREE**, **FOUR**, **FIVE**, or **SIX**, specifying the month within the half year, and the **n**th day of that month.
ANNUAL	Run once a quarter, where frequencyCode is **JANUARY**, **FEBRUARY**, **MARCH**, **APRIL**, **MAY**, **JUNE**, **JULY**, **AUGUST**, **SEPTEMBER**, **OCTOBER**, **NOVEMBER**, **DECEMBER**, specifying the month of the year, and the **n**th day of the month.

For example, to create a schedule for running background tasks once a week on Sunday:

```
public ScheduleDefinition getScheduleDefinition() {

                return new ScheduleDefinition("WEEKLY", null, 1);

}
```

To create a schedule for running background tasks every hour:

```
public ScheduleDefinition getScheduleDefinition() {

                return new ScheduleDefinition("MINUTES", null, 60);

}
```

top

---

public boolean autoRun();

autoRun is the call to perform any background tasks. This function is called based on the getScheduleDefinition(). This could be used to download and cache data locally.

top

---

protected final byte[] loadBlob(String key);

loadBlob() will load a blob (byte[]) that was previously saved by the connector, usually in a background task. The parameter key is a unique identifier for the data to load. Blobs can only be loaded on data sources that have been saved. areBlobsAvailable() can be used to see if blob access is available.

top

---

protected final boolean saveBlob(String key, byte[] data);

saveBlob() allows for saving a blob (byte[]) for later use. This is a way of saving data from background tasks for later use. The parameter key is the unique identifier for the data to be saved. Data is the byte[] to be associated with that key. Writing null to data will delete the saved data for the specified key. Blobs can only be loaded on data sources that have been saved. areBlobsAvailable() can be used to see if blob access is available.

top

---

protected final boolean areBlobsAvailable();

Blobs can only be loaded on data sources that have been saved. areBlobsAvailable() can be used to see if blob access is available. Blob access will not be available if a connector is tested prior to being saved.

top

---

public final Object getAttribute(String key);

getAttribute() allows for fetching attributes from the connection metadata. For example, a username may be specified for the connection through the Yellowfin UI. Using the key of the parameter, the contents of the username metadata field can be fetched for use when retrieving data from external APIs.

top

---

public final Integer getSourceId();

getSourceId() can be used to fetch the unique internal ID of source that this connector is associated with. This may be helpful for segregating data by connection in some kind of external cache or database.

top

Recommendations for using saveBlob() and loadBlob()

Minimize Stored/Cached Data

It is recommended to only store data that cannot be retrieved from an external source reliably. This could be in the case of “sliding window” access to data, where only a limited amount of historical data is available, and this needs to be downloaded prior to it becoming unavailable. Extremely slow data sets can also use locally stored data sets to improve query speed.

If significant amounts of data are stored in the blob system, it is recommended to truncate the data after a certain period. This might mean deleting all data when it reaches a certain age, or storing less granular information for older data. For instance, store raw data for three months, daily aggregated data for one year, and weekly aggregated data for older data. To achieve this, a background job would need to reaggregate and restore the data.

top

Minimize Blob Size

There is significant load induced on the Yellowfin database and server when storing and loading large sized blobs. If possible, distribute stored data across multiple blobs.

For instance, there may be an instance where 100,000 tweets are stored in the blob storage system. This might be stored with the key “ALL_TWEETS”. However, to minimize loading times of blobs, and to not overload the caching system, this could be split and stored in smaller chunks.

One way to do this would be to split up tweets by month:

"201601_TWEETS"

"201602_TWEETS"

"201603_TWEETS"

"201604_TWEETS"

When a query is requested from the connector, filters can be used to determine which blobs need to be used, and thus loaded from the blob storage system. For example, a query with the specified date range of 2016-02-05 to 2016-03-05 would just need to load the data for February and March, “201602_TWEETS” and “201603_TWEETS”.

There may be significant overhead required to join data sets from blobs. It is recommended that this be taken into consideration and to compare the performance of multiple smaller blobs versus larger ones.

There is no ideal size for blobs. The loading speed of blobs from the Yellowfin database will be dependent on the hardware and DBMS used. Public connectors will be used on Yellowfin installations of all sizes, so smaller, less powerful systems should be taken into consideration.

top

Application-Level Filtering & Aggregations

Yellowfin supports application-level filtering and aggregation. This allows Yellowfin to aggregate and filter data after receiving a result set from a connector.

Application-level aggregation is toggled based on the capabilities of the connector. If any data set columns (as returned by getColumns()) support native aggregations (where the connector returns aggregated data), then aggregations will be available.

Application-level filtering is also toggled based on the capabilities of columns in the connector. If any data set columns (as returned by getColumns()) support native filtering (where the connector applies its own filters), then the application-level filtering will be disabled, and only connector column filters will be available. Connector filters (as returned by getFilters()) can coexist with application-level filters.

top

Custom Error Messages

Custom messages can be returned to the Yellowfin UI if an error occurs whilst running a connector report. This can be done by throwing a ThirdPartyException() with a custom message from the connector plug-in.

```
throw new ThirdPartyException("Unable to connect to the Twitter API at this time.");
```

The custom error message will be shown as a standard “Oh-No” error where the report would usually be rendered. This would usually be thrown from the execute() function on a data set.

top

Previous topic: Define connector metadata
Next topic: Define data sets