connecting people and their data

Third-Party Connectors

May, 2016.

Contents

Introduction

Third-Party Connectors allow for Yellowfin to connector to any data source and return tabular data that can be consumed and displayed as tables and charts.

Data can be retrieved from any source via file access, API access, or data that is embedded in the connector itself. Most implementations are expected to make use of web service APIs, third-party web applications, and SAAS providers.

Implementation Overview

Implementation is done by extending Yellowfin's third-party java interface.

Each implementation consists of a single `AbstractDataSource` class. This contains details about how the connection is made, and prompts the user for required connection information. The `DataSource` can also contain a reference to a background job that can perform tasks on a scheduled basis. This schedule is defined in an implementation of a `ScheduleDefinition` class.

Each `DataSource` can have many `DataSets` available within it. Each `DataSet` is an implementation of an `AbstractDataSet` class. These can be thought of as views. A report can only be written against a single `DataSet`. A `DataSet` will have several columns and filters defined and will return a result set to Yellowfin.

A metadata class is also required. This will allow for a connection wizard to be defined, and prompt the end user for the required parameters to make a connection to the third-party source. This is an implementation of a `JDBCMetaData` class.

IDE Integration / Classpath

The definitions for the required Yellowfin classes are available in the following libraries from a standard Yellowfin installation. These are located in the `/appserver/webapps/ROOT/WEB-INF/lib/` directory of the installation.

- `i4-cor.jar`
- `i4-mi.jar`

These are dependencies that are required to compile a connector plugin for Yellowfin. Referencing these libraries in an IDE will also allow for code-completion and function lookups while editing.

Java Version

Yellowfin 7.2 has backwards compatibility to Java 6. Any connectors made for public consumption should be compiled with Java 6, unless a connector's dependencies force the use of a newer Java implementation. The plugin loader will display an error reporting that the version of Java is not sufficient if the loaded `.jar` file is compiled with a newer version that what is in use by Yellowfin.

Connector Functional Restrictions

There are several functional restrictions when using Third-Party Connectors compared to the features available for relational SQL sources.

As of Yellowfin 7.2 the limitations to Third-Party Connector reports include;

- Multiple DataSets cannot be joined. Reports can be written against a single DataSet only.
- Calculated Fields and Custom Functions are unavailable.
- Sub Query functionality, both basic and advanced, are unavailable.
- Drill Down functionality is unavailable.
- Grouped Value functionality is unavailable.
- Complex Connector Filter logic is unavailable. Multiple connector-level filters are all applied with logical ANDs. Bracketing and logical ORs cannot be used to change filter logic on a report. Application-level filters do not have these restrictions.

Many of these features can be enabled with Yellowfin 7.2's Report From Report functionality, which allows for the creation of a view based on a Third-Party connector report results. Reports written against this view will have access to all report-level functionality available to relational sources.

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 DataSet 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 DataSet 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 co-exist with application-level filters.

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 plugin.

```
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 DataSet.

Defining a Data Source

A connector DataSource is an implementation of the abstract java class AbstractDataSource. When defining a DataSource 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();
```

DataSource Function Definitions

public abstract String getDataSourceName();

Return a DataSourceName as a String.

public abstract Collection<AbstractDataSet> getDataSets();

Return the collection of DataSets that are available in this DataSource. See the DataSet section for defining a DataSet.

public abstract JDBCMetaData getDataSourceMetaData();

Return the connection meta-data required for this DataSource. See the MetaData section for defining Meta Data for a Data Source.

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.

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.

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.

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);
}
```

public boolean autoRun();

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

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.

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.

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.

public final Object getAttribute(String key);

getAttribute() allows for fetching attributes from the connection meta-data. 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 meta-data field can be fetched for use when retrieving data from external APIs.

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.

Recommendations for using saveBlob() and loadBlob()

Minimise 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 re-aggregate and re-store the data.

Minimise 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. 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 datasets 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.

Defining a Data Set

A connector DataSet is an implementation of abstract java class `AbstractDataSet`. When defining a DataSet the following functions require implementation:

```
public abstract String getDataSetName();
public abstract List<ColumnMetaData> getColumns();
public abstract List<FilterMetaData> getFilters();
public abstract Object[][] execute(List<ColumnMetaData> columns,
List<FilterData> filters);
public abstract boolean getAllowsDuplicateColumns();
public abstract boolean getAllowsAggregateColumns();
```

Other functions that can be overridden:

```
public boolean isFilterValueEnabled(String filter);
public List<Object> getFilterValues(String filter, HashMap<String,
FilterData> appliedFilters);
```

DataSet Function Definitions

public abstract String getDataSetName();

Return a DataSetName as a String. This should not be internationalised.

public abstract List<ColumnMetaData> getColumns();

Return a collection of ColumnMetaData objects that define the columns that are available in the DataSet. A column can also be defined to be used as a filter.

ColumnMetaData objects require the following meta-data to be defined:

`Attribute`	Description
`columnName`	Name of the column. This should be unique and should not be internationalised. User friendly names and internationalisation can be applied at the Yellowfin metadata level.
`columnType`	DataType of the column. See DataType in the appendix for more information.
`fieldType`	FieldType of the column. See FieldType in the appendix for more information.
`availableAggregations`	Array of AggregationType. This defines the aggregations that can be applied to this column. See AggregationType in the appendix for more information.
`availableFilterOperators`	Array of FilterOperator. This defines the operators that can be applied to this column if it is used as a filter. If this column cannot be used as a filter is should be set as **null**. See FilterOperator in the appendix for more information.

There are multiple constructors for ColumnMetaData, that allow for defining this object with a single line of Java code.

public abstract List<FilterMetaData> getFilters();

Return a collection of FilterMetaData objects that define the filters that are available in the DataSet. A filter in this context is a parameter that can be used for a report, it does not return data like a column does.

FilterMetaData objects require the following meta-data to be defined:

`Attribute`	Description
`filterName`	Name of Filter. This should be unique and should not be internationalised. User friendly names and internationalisation can be applied at the Yellowfin metadata level.
`filterType`	FilterType of the column. See FilterType in the appendix for more information.
`Mandatory`	Define whether this filter is mandatory or not. Data cannot be returned from this DataSet without this filter being set.
`availableAggregations`	Array of AggregationType. This defines the aggregations that can be applied to this filter. Currently unsupported. See AggregationType in the appendix for more information.
`availableOperators`	Array of FilterOperator. This defines the operators that can be applied to this filter. See FilterOperator in the appendix for more information.

There are multiple constructors for FilterMetaData, that allow for defining this object with a single line of Java code.

public abstract Object[][] execute(List<ColumnMetaData> columns, List<FilterData> filters);

Returns the result set from this DataSet. This is the main function for processing the query. The parameter column specifies the columns that have been chosen for the query from Yellowfin. The parameter filters specifies the filters that have been assigned to the query from Yellowfin. The function must return a 2 dimensional object array with the contents of the columns requested.

The parameter column is of the type ColumnMetaData. This has information about the columns selected and any aggregation modifiers that have been made to those columns.

The ColumnMetaData function `getSelectedAggregation()` returns the selected aggregation applied to a column. This is of type AggregationType.

The parameter filters is of the type FilterData. This has information about the filters selected and the values that have been assigned to them. FilterData has the following attributes:

`Attribute`	Description
`filterName`	Name of the filter or column that this filter represents.
`metaData`	FilterMetaData of filter. This will be a link to the FilterMetaData object that was defined in `getFilters()` or FilterMetaData object that is created automatically to wrap a column that supports filtering.
`filterValue`	The value of the filter. This is a java Object that holds the data for the datatype of the filter. Filters that hold multiple values will be returned in a Java List. This is for filters using Between, Not Between and In List and Not In List.
`filterOperator`	Instance of type FilterOperator. This defines the operator that has been applied to this filter.
`aggregationType`	Instance of type AggregationOperator. This defines the operator that has been applied to this filter.

A custom error message can be show if an error occurs by throwing a `ThirdPartyException()`. See **Custom Error Messages**.

public abstract boolean getAllowsDuplicateColumns();

Return whether or not this Data Set supports the same column being selected more than once.

If this returns true, then the Data Set will be sent duplicate columns via the execute function, if they are required.

If this is set to false, then only a single instance of each column will be sent to the execute function and Yellowfin will duplicate repeated columns after receiving the data.

public abstract boolean getAllowsAggregateColumns();

Return whether or not this Data Set supports native aggregations.

If this is false, then Yellowfin will enable application level aggregations. This will allow for Yellowfin to aggregate data once it has been returned from the DataSet.

If this is true, then the DataSet must return aggregated data when requested. Columns can be defined with what aggregations can be applied to them during report creation. This is defined in `getColumns()`.

public boolean isFilterValueEnabled(String filter);

Return whether filter values can be returned for a particular filter (or column filter). Returning true will mean that calling `getFilterValues()` will return a list of values for this filter.

public List<Object> getFilterValues(String filter, HashMap<String, FilterData> appliedFilters);

If `isFilterValueEnabled()` returns true for a given filter name, then this function will be called to return a Java List of available filter options for user selection. The parameter appliedFilters will hold a Map (keyed by filtername) that holds the values of other filters that are currently set. This information can be used to further restrict the values returned by this function.

Defining Connector Metadata

The Connector MetaData is an implementation of abstract java class JDBCMetaData. This defines what connection details need to be prompted to the user for creating a connection to a third-party source. This may include parameters like usernames, tokens, hostnames, ports, account names etc.

Basically, the JDBCMetaData class is used for building a connection wizard for a DataSource. The followfing functions need to be implemented to create a basic connection wizard:

```
public JDBCMetaData();
public void initialiseParameters();
public String buttonPressed(String buttonName) throws Exception;
```

Helper functions that are also accessible in JDBCMetaData:

```
protected final void addParameter(Parameter p);
public void setParameterValue(String key, Object value);
public final Object getParameterValue(String key);
public boolean isParameterRequired(String key);
public boolean hasDependentParameters(String key);
```

MetaData Function Definitions

public JDBCMetaData(); (Constructor)

The following attributes should be set in the constructor:

Attribute	Description
sourceName	Text name for the DataSource. For example “Twitter Connector”.
sourceCode	A unique text code for the DataSource. For example “TWITTER_CONNECTOR”.
driverName	The text class name of the DataSource. For example “com.code.TwitterConnector”
sourceType	This should always be DBType.THIRDPARTY

Example implementation:

```
	public SkiTeamMetaData() {

			super();

			sourceName = "Ski Team Source";
			sourceCode = "SKI_DATA_SOURCE";
			driverName = SkiTeamDataSource.class.getName();
			sourceType = DBType.THIRD_PARTY;
	}
```

public void initialiseParameters();

This function is where parameters should be registered. Registered parameters will be displayed to the user when creating a connection with this DataSource. Use the function `addParameter()` to add required parameters.

Example implementation:

```
public  void initialiseParameters() {

        super.initialiseParameters();

        addParameter(new Parameter("HELP", "Connection Details",  "Text",
        TYPE_NUMERIC, DISPLAY_STATIC_TEXT, null, true));

        Parameter p = new Parameter("URL", "1. Request Access PIN", "Connect
        to twitter to receive a PIN for data access",TYPE_UNKNOWN,
        DISPLAY_URLBUTTON,  null, true);
        p.addOption("BUTTONTEXT", "Request URL");
        p.addOption("BUTTONURL", "http://google.com");
        addParameter(p);

        addParameter(new Parameter("PIN", "2. Enter PIN",  "Enter the PIN
        recieved from Twitter", TYPE_NUMERIC, DISPLAY_TEXT_MED, null, true));

        p = new Parameter("POSTPIN", "3. Validate Pin",  "Validate the PIN",
        TYPE_TEXT, DISPLAY_BUTTON, null, true);
        p.addOption("BUTTONTEXT", "Validate PIN");
        addParameter(p);

        addParameter(new Parameter("ACCESSTOKEN", "Access Token",
        "AccessToken that allows access to the Twitter API", TYPE_TEXT,
        DISPLAY_PASSWORD, null, true));

        addParameter(new Parameter("ACCESSTOKENSECRET", "Access Token
        Secret",  "AccessToken Password that allows access to the Twitter
        API", TYPE_TEXT, DISPLAY_PASSWORD, null, true));


}
```

protected final void addParameter(Parameter p);

Parameter objects require the following meta-data to be defined:

Attribute	Description
uniqueKey	Text Unique Key for this parameter.
displayName	Text description. This can be internationalised.
description	Parameter description. This can be internationalised.
defaultValue	Object to be assigned as the default value for this parameter.
displayType	DisplayType for this parameter. See DisplayType in appendix for more information.
dataType	DataType for this parameter. See Parameter DataType in appendix for more information.

There are multiple constructors for Class Parameter, that allow for defining this object with a single line of Java code.

Some parameter display types require additional options, such as dropdown boxes and radio buttons. These need to be added to the parameter object after instantiation. For example:

```
Parameter p = new Parameter("URL", " Access PIN", "Connect to twitter to
receive a PIN for data access",TYPE_UNKNOWN, DISPLAY_URLBUTTON,  null,
true);


p.addOption("BUTTONTEXT", "Request URL");
p.addOption("BUTTONURL", "http://google.com");
addParameter(p);
```

public String buttonPressed(String buttonName) throws Exception;

This is a call-back for button UI elements. The function parameter buttonName holds the unique key for the button that called the callback function.

A button callback may be used to change the values of other parameters programmatically. A parameter can be set with setParameterValue(String key, String value).

public void setParameterValue(String key, Object value);

Set the value of the parameter. Where function parameter key is the unique key of the parameter to be set and value is the value to assign to it.

public final Object getParameterValue(String key);

Get the value of a parameter. Where the function parameter key is the unique key of the parameter value to fetch.

public boolean isParameterRequired(String key);

To implement dependent filters the isParameterRequired() function can be overridden. Based on the values of other parameters, logic can determine whether the parameter with unique key should be shown.

For example:

```
public boolean isParameterRequired(String key) {

        if ("DOMAIN".equals(key)) {
              if ("SQL".equals(getParameterValue("WINDOWSAUTH"))) {
                     return false;
              }
        }
       return true;
}
```

public boolean hasDependentParameters(String key);

Return true for parameter with unique key if it has dependent parameters. This function is used to determine whether other parameter’s visibility needs to be updated based on modification of this parameter’s value.

Packaging a Connector for Yellowfin

Yellowfin's plug-in loader will accept standard JAR files, however any dependencies will need to be loaded individually into the same plug-in group.

An alternative to loading all JAR files individually is to create a YFP package. A YFP package is just a standard zip file renamed with a "yfp" extension. This will allow the user to load all dependencies with a single file upload into Yellowfin.

When creating a YFP package all JAR files should be placed in the root of the YFP zip file. Only external dependencies should be included in the YFP file. Do not include Yellowfin build dependencies like

Additionally, out-of-the-box content can also be embedded in a YFP file. The out-of-the-box content is contained in a standard Yellowfin export XML file. This should also be placed in the root of the YFP zip file, and be renamed to content.xml. Yellowfin will detect this and import this content when a new data source is created with this DataSource.

Appendix

DataType

`DataType` is used for defining the data type for a column or filter.

Type	Description
`TEXT`	Text value. A DataSet should return a Java String for this type.
`NUMERIC`	Numeric value. A DataSet should return a Java String for this type.
`INTEGER`	Integer value. A DataSet should return a Java Long or Integer for this type.
`DATE`	Date value. A DataSet should return a Java java.sql.Date for this type.
`TIMESTAMP`	Timestamp value. A DataSet should return a Java java.sql.Timestamp for this type.
`SHORT`	Short integer value. A DataSet should return a Java Long or Integer for this type.
`ARRAY`	Array. Currently unsupported.
`BOOLEAN`	Boolean value. A DataSet should return a Java Boolean for this type.
`BLOB`	Blob/Binary value. A DataSet should return a Java byte[] for this type.

FieldType

`FieldType` is used for defining the field type for a column or filter.

Type	Description
`DIMENSION`	Dimensional Value. The report builder will use dimensional filter paradigm for this column, and restrict aggregation types to COUNT and COUNT DISTINCT.
`METRIC`	Metric/Measure value. The report builder will use metric filter paradigm for this column, and allow all available aggregation types.
`UNKNOWN`	Unknown column type. This should not be used.

AggregationType

`AggregationType` is used for defining an aggregation that can be applied to a column or filter.

Type	Description
`COUNT`	Count all values of a column.
`COUNTDISTINCT`	Count all distinct values of a column.
`AVG`	Average the values of a column.
`MAX`	Return the maximum value of a column.
`MIN`	Return the minimum value of a column.
`SUM`	Return the sum of the values in the column.

FilterOperator

`FilterOperator` is used for defining operators for a column or filter.

Type	Description
`EQUAL`	Values are equal.
`NOTEQUAL`	Values are not equal.
`INLIST`	Value is within a defined list of values.
`NOTINLIST`	Value is not withing a defined list of values.
`GREATER`	Value is greater than a value.
`GREATEREQUAL`	Value is greater than or equal to a value.
`LESS`	Value is less than a value.
`LESSEQUAL`	Value is less than or equal to a value.
`BETWEEN`	Value is between two bounds.
`NOTBETWEEN`	Value is not between two bounds.
`ISNULL`	Value is null.
`ISNOTNULL`	Value is not null.

STARTSWITH	Text starts with a value.
NOTSTARTSWITH	Text does not start with a value.
ENDSWITH	Text ends with a value.
NOTENDSWITH	Text does not end with a value.
CONTAINS	Text contains a value.
NOTCONTAINS	Text does not contain a value.
ISEMPTYSTRING	Text is empty string.
ISNOTEMPTYSTRING	Text is not an empty string.

Parameter DataType

Parameter DataType is used for defining the data stored for a given meta-data parameter. These are defined as final static integers in the UserInputParameter Class.

Type	Description
TYPE_UNKNOWN	Unknown Value.
TYPE_NUMERIC	Numeric Value.
TYPE_TEXT	Text Value.
TYPE_PASSWORD	Password Value.
TYPE_BOOLEAN	Boolean Value.

Display Type

DisplayType is used for defining how data will be shown in the connection meta-data wizard. These are defined as final static integers in the UserInputParameter Class.

Type	Description
DISPLAY_TEXT_TINY	Short text input box.
DISPLAY_TEXT_MED	Medium length text input box.
DISPLAY_TEXT_MED_LONG	Medium-long text input box.
DISPLAY_TEXT_LONG	Long text input box.

`DISPLAY_SELECT`	Select dropdown input box. This type of parameter requires additional options. Options are added to Parameter with `addOption(String Key, String Description)`. The key is stored as the parameter value when selected. The description is shown as a selectable option in a dropdown box.
`DISPLAY_RADIO`	Radio selection input box. This type of parameter requires additional options. Additional options are rendered as selectable elements in the UI. Options are added to Parameter with `addOption(String Key, String Description)`. The key is stored as the parameter value when selected. The description is shown next to the selectable radio button.
`DISPLAY_HIDDEN`	Hidden input box.
`DISPLAY_BUTTON`	Button. This type of parameter requires additional options. This UI element makes a callback to the `buttionPressed()` function in the `JDBCMetaData` Class. Additional option "BUTTONTEXT" is used for setting the text shown on the button.
`DISPLAY_STATIC_TEXT`	Displays static text. The long description of the parameter will be displayed as static text.
`DISPLAY_URLBUTTON`	Button with a link to an external URL. This type of parameter requires additional options. Additional option "BUTTONTEXT" is used for setting the text shown on the button. Additional option "BUTTONURL" is used for setting the URL that the button links to.
`DISPLAY_PASSWORD`	Password input box.