



Connector Plugins

An Example

Connector Plugins

With the Tableau Connector Plugin SDK, you can add a new connector that you can use to visualize your data from any database through an ODBC or JDBC driver. When you create a connector plugin, you can add customizations to the connector, use the connectivity test harness to validate the connector behavior during the development process, and then package and distribute the connector plugin to users. This document describes the files that make up a connector plugin.

What is a connector plugin

A connector plugin is a set of files that describe:

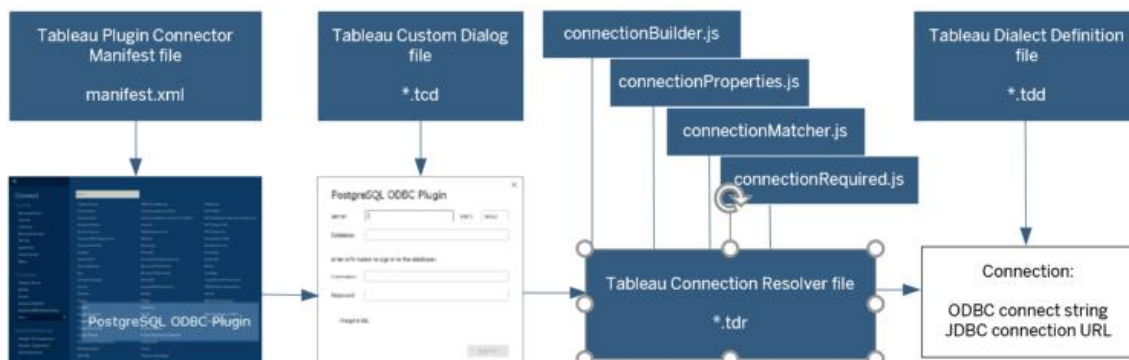
- UI elements needed to collect user input for creating a connection to a data source
- Any dialect or customizations needed for the connection

And include:

- A connection string builder
- A driver resolver

A connector plugin supports the same things any other connector supports, including publishing to a server if the server has the plugin, creating extracts, data sources, vizzes, and so on.

See the relationship between the connector plugin files (in blue) and the Tableau Connect pane and connection dialog:



Connector plugin example

A connector plugin, is a set of files that describe the UI elements needed to collect user input for creating a connection to a data source, any dialect or customizations needed, a connection string builder, driver resolver, and the ODBC- or JDBC-based driver. Starting with the set of base connector files, you can add customizations to each file, while using the connectivity test harness to validate the plugin behavior along the way. The base connector files are described below.

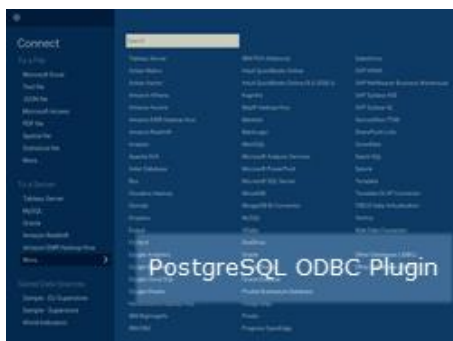


1 manifest.xml

The manifest.xml file informs Tableau about your connector plugin and displays the connector plugin name in the Tableau Connect pane. It's a required file that defines the plugin class and description. The **class** value is a unique key for your plugin and is used in other XML files to apply their customizations and in Tableau workbooks to match connection types.

Each connector plugin is typically based on a "class" such as ODBC or JDBC, and provides additional customizations beyond the class. The **name** value displays the connector name in the Tableau **Connect** pane.

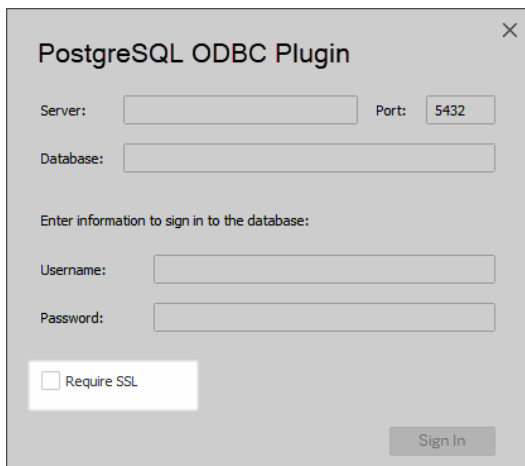
```
<connector-plugin class='postgres_odbc' superclass='odbc' plugin-
version='0.0.0' name='PostgreSQL ODBC Plugin' version='18.1'>
  <connection-customization class="postgres_odbc" enabled="true"
version="10.0">
    <vendor name="vendor"/>
    <driver name="driver"/>
    <customizations>
      <customization name="CAP_64BIT_CALCULATIONS" value="yes"/>
      <customization name="CAP_SELECT_INT0" value="yes"/>
      <customization name="CAP_SELECT_TOP_INT0" value="yes"/>
    </customizations>
  </connection-customization>
  <connection-dialog file='connection-dialog.tcd'> 2
  <dialect file='dialect.tdd'> 9
</connect-plugin>
```



2 *.tcd

(Optional) You can use the Tableau Custom Dialog (.tcd) file to customize the connection dialog, or your plugin can inherit a dialog from its parent. For example, if you include **show-ssl-check box** and set the value to "true", the **Require SSL** check box will display on the sign-in dialog.

```
<connection-dialog class='postgres_odbc'>
  <connection-config>
    <authentication-mode value='Basic' />
    <authentication-options>
      <option name="UsernameAndPassword" default="true" />
    </authentication-options>
    <db-name-prompt value='Database: ' />
    <has-pre-connect-database value="true" />
    <port-prompt value="Port: " default="5432" />
    <show-ssl-checkbox value="true" />
  </connection-config>
</connection-dialog>
```



3 *.tdr

(Optional) Tableau uses the Connector Resolver (.tdr) file to create a connection to your data. The .tdr file calls several javascript files, and includes the `driver-resolver` section.

Tableau database connections have a unique type, the **class** attribute. For example, all Postgres connections have the same **class**. Each connection also has a set of **connection attributes** with unique values. Typically these attributes include the database server, username, and password. If the attributes and their values are identical then the connections are considered the same and can be reused and shared within the Tableau process.

Connection attributes are also used to pass values from the connection dialog or the saved Tableau workbook to the **Connection Resolver**. The Connection Resolver knows how to use these attributes to format an ODBC or JDBC connection string.

```
<tdr class='postgres_odbc'>
  <connection-resolver>
    <connection-builder>
      <script file='connectionBuilder.js' />
    </connection-builder>
    <connection-properties>
      <script file='connectionProperties.js' />
    </connection-properties>
    <connection-matcher>
      <script file='connectionMatcher.js' />
    </connection-matcher>
    <connection-normalizer>
      <script file='connectionRequired.js' />
    </connection-normalizer>
    <driver-resolver>
      <driver-match>
        <driver-name type='regex'>PostgreSQL Unicode*</driver-name>
      </driver-match>
    </driver-resolver>
  </connection-resolver>
</tdr>
```

The driver resolver defines rules for locating the ODBC driver. You don't need a driver resolver for JDBC plugins.

4 connectionBuilder.js

The ODBC connection string and the JDBC connection URL are created by calling the Connection Builder script and passing in a map of attributes that define how the connection is configured. Some values come from the connection dialog and are entered by the user (like username, password, and database name). They are mapped to ODBC connection string values that the PostgreSQL driver understands. Other attributes (like BOOLSASCHAR and LFCONVERSION) have values set to useful defaults.

Example ODBC Connection Builder

```
(function dsbuilder(attr)
{
    var params = {};

    params["SERVER"] = attr["server"];
    params["PORT"] = attr["port"];
    params["DATABASE"] = attr["dbname"];
    params["UID"] = attr["username"];
    params["PWD"] = attr["password"];
    params["BOOLSASCHAR"] = "0";
    params["LFCONVERSION"] = "0";
    params["UseDeclareFetch"] = "1";
    params["Fetch"] = "2048";

    var formattedParams = [];

    formattedParams.push(connectionHelper.formatKeyValuePair(driverLocator.keywordDriver, driverLocator.locateDriver(attr)));

    for (var key in params)
    {
        formattedParams.push(connectionHelper.formatKeyValuePair(key, params[key]));
    }

    return formattedParams;
})
```

These values come from the connection dialog and are entered by the user. Here they are mapped to ODBC connection string values that the Postgres driver can use.

You can set these values to useful defaults.

Here the driver resolver is invoked to find the matching Postgres ODBC driver.

This formats the params map into a set of ODBC key value pairs.



Example JDBC Connection Builder

```
(function dsbuilder(attr) {  
    var urlBuilder = "jdbc:postgresql://" + attr["server"] + ":" +  
attr["port"] + "/" + attr["dbname"] + "?";  
    var params = [];  
    params["user"] = attr["username"];  
    params["password"] = attr["password"];  
    var formattedParams = [];  
    for (var key in params) {  
        formattedParams.push(connectionHelper.formatKeyValuePair(key,  
params[key]));  
    }  
    urlBuilder += formattedParams.join("&")  
    return [urlBuilder];  
})
```



5 connectionProperties.js

This script is needed only if you're using a JDBC driver.

Example connectionProperties.js

```
(function propertiesBuilder(attr)
{
    //This script is only needed if you are using a JDBC driver.
    var params = {};
    //set keys for properties needed for connecting using JDBC
    var KEY_USER = "user";
    var KEY_PASSWORD = "password";
    var KEY_WAREHOUSE = "s3_staging_dir"
    //set connection properties from existing attributes
    params[KEY_USER] = attr[connectionHelper.attributeUsername];
    params[KEY_PASSWORD] = attr[connectionHelper.attributePassword];
    params[KEY_WAREHOUSE] = attr[connectionHelper.attributeWarehouse];
    var formattedParams = [];
    //Format the attributes as 'key=value'. By default some values are
    escaped or wrapped in curly braces to follow the JDBC standard, but you
    can also do it here if needed.
    for (var key in params)
    {
        formattedParams.push(connectionHelper.formatKeyValuePair(key,
params[key]));
    }
    //The result will look like (search for jdbc-connection-properties
in tabprotosrv.log):
    //"jdbc-connection-
properties","v":{"password":"*****","s3_staging_dir":"s3://aws-athena-
s3/","user":"admin"}
    return formattedParams;
})
```

6 connectionMatcher.js

This script defines how connections are matched. In most cases, the default behavior works, so you don't have to include the `connection-matcher` section in your *.tdr file.

7 connectionRequired.js

This script defines what makes up a unique connection. The following values work for most cases.

```
(function requiredAttrs(attr)
{
    return ["class", "server", "port", "dbname", "username", "password"];
})
```

8 Example Connection

The Tableau Connection Resolver file (*.tdr) generates an ODBC ConnectString or a JDBC Connection URL, which you can find in tabprotosvr.txt.

For ODBC, search for `ConnectionString` to find something like this example:

```
ConnectionString: DRIVER={PostgreSQL
Unicode(x64)};SERVER=postgres;PORT=5432;DATABASE=TestV1;UID=test;PWD=*****
***;BOOLSASCHAR=0;LFCONVERSION=0;UseDeclareFetch=1;Fetch=2525
```

For JDBC, search for `Connection URL` to find something like this example:

```
JDBCProtocol Connection URL:
jdbc:postgresql://postgres:5342/TestV1?user=test&password=*****
```



9 *.tdd

After connection, Tableau uses your *.tdd dialect file to determine which SQL to generate when retrieving information from your database. You can define your own dialect in the *.tdd file, or your plugin can inherit a dialect from its parent.

Example dialect.tdd

```
<?xml version="1.0" encoding="utf-8"?>
<dialect name='SimplePostgres'
        class='postgres_odbc'
        base='PostgreSQL90Dialect'
        version='18.1'>
<!-- You can add dialect files here. See the documentation for more
information. -->
</dialect>
```

