TRIFACTA

Developer Guide

Version: 7.1.2
Doc Build Date: 11/25/2020

For third-party license information, please select **About Trifacta** from the Help menu.

1. *Developer . 4*
    1.1 *User-Defined Functions . . 5*
        1.1.1 *Java UDFs . . 8*
    1.2 *Create Custom Data Types Using RegEx . 16*
    1.3 *API Reference 19*
        1.3.1 *API Overview 20*
        1.3.2 *Manage API Access Tokens . 27*
        1.3.3 *API Workflows . 30*
            1.3.3.1 *API Workflow - Develop a Flow . 31*
            1.3.3.2 *API Workflow - Deploy a Flow . 39*
            1.3.3.3 *API Workflow - Run Job . 49*
            1.3.3.4 *API Workflow - Run Job on Dataset with Parameters . 60*
            1.3.3.5 *API Workflow - Run Deployment . 68*
            1.3.3.6 *API Workflow - Publish Results . 74*
            1.3.3.7 *API Workflow - Swap Datasets . 79*
            1.3.3.8 *API Workflow - Manage Outputs . 84*
            1.3.3.9 *API Workflow - Manage AWS Configurations . 97*
            1.3.3.10 *API Workflow - Run Plan 105*

Developer

This section contains topics of interest to data engineers and other developers.

Use of the features documented in this section requires programming skills.

User-Defined Functions

Contents:

- *UDF Service*
- *Supported UDF Language Frameworks*
- *Running a UDF within the Platform*

The Trifacta® platform enables the creation of user-defined functions (UDFs) for use in your Trifacta deployment. A **user-defined function** is a way to specify a custom process or transformation for use in your specific Trifacta solution, using familiar development languages and third-party libraries. Through UDFs, you can apply enterprise- or industry-specific expertise consistently into your data transformations. A user-defined function is a custom function that is created in one of the supported language frameworks. Each user-defined function has a defined set of inputs and generates a single output.

UDF Service

The following diagram provides a high-level overview of the UDF service which provides integration of user-defined functions into recipe execution.

- Diagram 1: The figure illustrates execution of a UDF in interactive mode, where a user interacts with the Transformer grid.
- Diagram 2: This feature illustrates how UDFs interact with the cluster at job execution time.

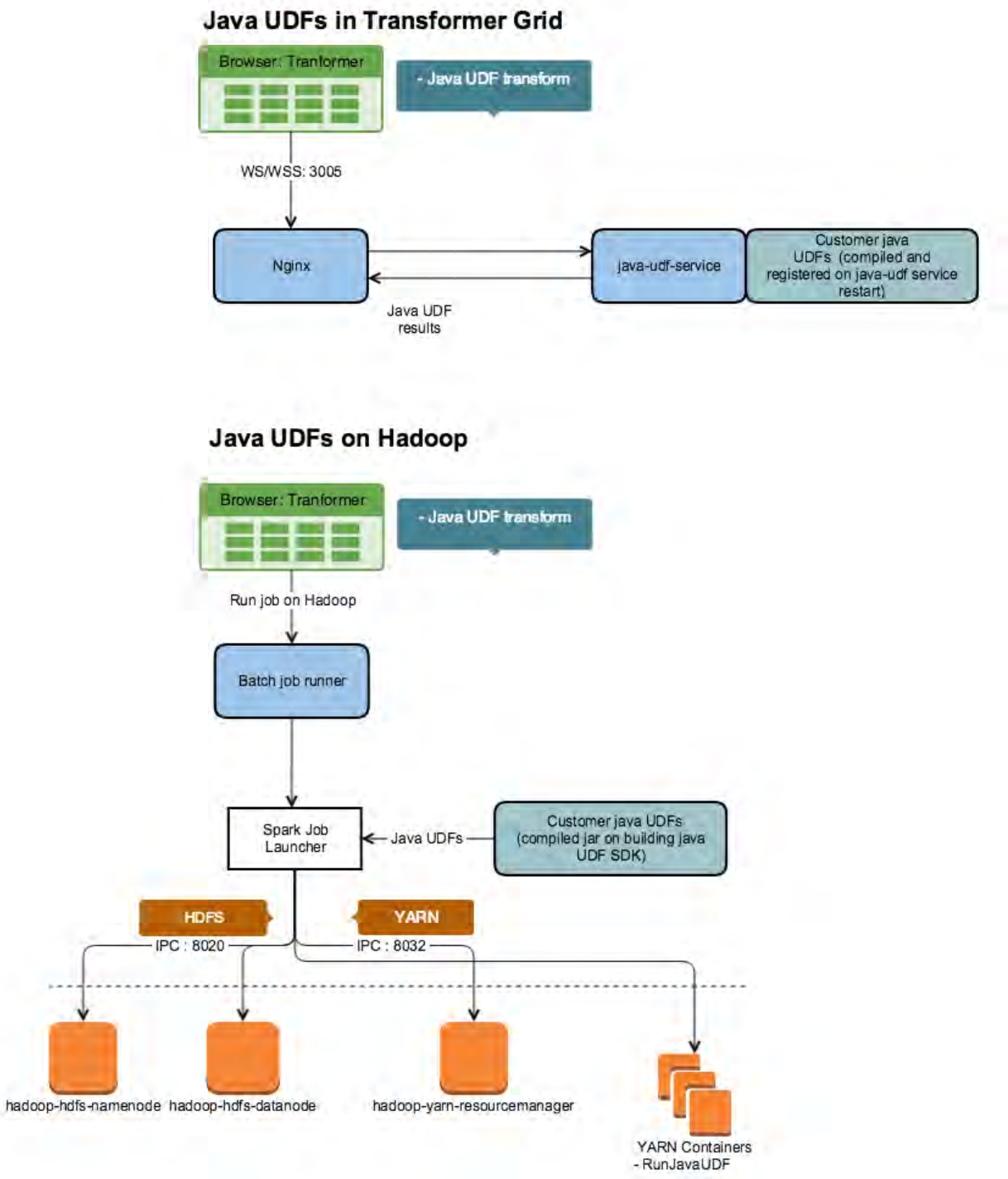

Figure: User-Defined Service

Supported UDF Language Frameworks

Please use the following links to enable the creation of user-defined functions in the listed language.

- *Java UDFs*

Running a UDF within the Platform

After you have created and tested your UDF, you can execute it by entering `udf` in the Search panel and populating the rest of the step in the Transform Builder.

In this example, the `AdderUDF` function is added:

Transformation Name	`Invoke external function`
Parameter: Column	`colA`
Parameter: Arguments	`100`
Parameter: New column name	`myAdderUDFColumn`

Notes:

- After entering `udf`, your UDF should appear in a drop-down list. If not, please verify that it has been properly created, compiled, and registered and that the udf-service has been restarted.
- The Column parameter is a comma-separated list of the source data to be used as inputs to the exec method.
- The Argument parameter is a string of comma-separated values used as inputs to the init method.
- Optionally, The New column name parameter can be used to provide a specific name to the generated column. If it is not used, a column name is generated.

NOTE: When a recipe containing a user-defined function is applied to text data, any non-printing (control) characters cause records to be truncated by the Spark running environment during job execution. In these cases, please execute the job on the Photon running environment.

For more information, see *Invoke External Function*.

NOTE: Running user-defined functions for an external service, such as Hive, is not supported from within a recipe step. As a workaround, you may be able to execute recipes containing such external UDFs on the Photon running environment. Performance issues should be expected on larger datasets.

See *Transformer Page*.

Java UDFs

Contents:

- *Pre-requisites*
- *Overview*
  - *Known Limitations*
- *Enable Service*
- *Deployment*
- *Creating a UDF*
  - *UDF Requirements*
  - *Example - Concatenate strings*
  - *Example - Add by constant*
  - *Error Handling*
- *Testing the UDF*
- *Compiling the UDF*
  - *JDK version mismatches*
- *Registering the UDF*
  - *Enabling UDF service on HDInsight cluster*
- *Running Your UDF*
- *Troubleshooting*
  - *"Websocket Receive()" error in Transformer page UI*
  - *Photon crashes during execution of UDF*

This section describes how to create and deploy Java-based user-defined functions (UDFs) into your Trifacta® deployment.

Creation of UDFs requires development experience and access to an integrated development environment (IDE).

Pre-requisites

1. Access to the Trifacta deployment
2. IDE
3. The Java UDF is stored in the Trifacta deployment in the following location: `libs/custom-udfs-sdk/build/distributions/java-custom-udf-sdk.zip`

NOTE: If you are installing custom UDFs and the Trifacta node does not have an Internet connection, you should download the Java UDF SDK in an Internet-accessible location, build your customer UDF JAR there, and then upload the JAR to the Trifacta node.

Overview

Each UDF can take one or more inputs and produces a single output value (map only).

Inputs and outputs must be one of the following types:

- Bool
- String
- Long
- Double

Known Limitations

- In the Trifacta application, previews are not available for user-defined functions.

- Retaining state information across the exec method is unstable. More information is provided below.

  NOTE: When a recipe containing a user-defined function is applied to text data, any null characters cause records to be truncated by the running environment during Trifacta Photon job execution. In these cases, please execute the job in the Spark running environment.

Enable Service

You must enable the Java UDF service in the Trifacta platform.

Steps:

1. You can apply this change through the *Admin Settings Page* (recommended) or `trifacta-conf.json`. For more information, see *Platform Configuration Methods*.
2. Enable the correct flag:

   ```
   "feature.enableUDFTransform.enabled": true,
   ```

3. Save your changes.

Deployment

Steps:

1. Unzip `java-custom-udf-sdk.zip`.
2. Within the unzipped directory, execute the install command. The following is specific to the Eclipse IDE:

   ```
   gradlew eclipse
   ```

3. Import the project into your IDE.

Creating a UDF

UDF Requirements

All UDFs must implement the `TrifactaUDF` interface. This interface adds the four methods that each UDF must override: init, exec, inputSchema, and finish.

1. **init method:** Used for setting private variables in the UDF. This method may be a no-op function if no variables must be set. See the *Example - Concatenate strings* below.

   Tip: In this method, perform your data validation on the input parameters, including count, data type, and other constraints.
   NOTE: The init method must be specified but can be empty, if there are no input parameters.
2. **exec method:** Contains functionality of the UDF. The output of the exec method must be one of the supported types. It is also must match the generic as described. In the following example, `TrifactaUDF<String>` implements a String. This method is run on each record.

   Tip: In this method, you should check the number of input columns.
   Keep state that varies across calls to the exec method can lead to unexpected behavior. One-time initialization, such as initializing the regex compiler, is safe, but do not allow state information to mutate across calls to exec. This is a known issue.
3. **inputSchema method:** The inputSchema method describes the schema of the list on which the exec method is acting. The classes in the schema must be supported. Essentially, you should support the I/O types described earlier.
4. **finish method:** The finish method is run at the end of UDF. Typically, it is a no-op.

NOTE: If you are executing your UDF on the Spark running environment, the finish method cannot be invoked at this point. Instead, it is invoked as part of the shutdown of the Java VM. This later execution may result in the finish method failing to be invoked in situations like a JVM crash.

Example - Concatenate strings

The following code example concatenates two input strings in the `List<Object>`. This UDF can be easily modified to concatenate more strings by modifying the `inputSchema` function.

Example UDF: ConcatUDF

```
package com.trifacta.trifactaudfs;
import java.io.IOException;
import java.util.List;


/**
 * Example UDF that concatenates two columns
 */
public class ConcatUDF implements TrifactaUDF<String> {
  @Override
  public String exec(List<Object> inputs) throws IOException {
    if (inputs == null) {
      return null;
    }
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < inputSchema().length; i += 1) {
      if (inputs.get(i) == null) {
        return null;
      }
      sb.append(inputs.get(i));
    }
    return sb.toString();
  }
  @SuppressWarnings("rawtypes")
  public Class[] inputSchema() {
    return new Class[]{String.class, String.class};
  }
  @Override
  public void finish() throws IOException {
  }
  @Override
  public void init(List<Object> initArgs) {
  }
}
```

Notes:

- The first line indicates that the function is part of the `com.trifacta.trifactaudfs` package.
- The defined UDF class implements the `TrifactaUDF` class, which is the base interface for UDFs.
  - It is parameterized with the return type of the UDF (a Java `String` in this case).
  - The input into the function is a list with input parameters in the order they are passed to the function within the Trifacta platform. See *Running Your UDF* below.
- The UDF checks the input data for null values, and if any nulls are detected, returns a null.
- The `inputSchema` describes the input list passed into the exec method.
  - An error is thrown if the type of the data that is passed into the UDF does not match the schema.
  - The UDF must handle improper data. See *Error Handling* below.

Example - Add by constant

In this example, the input value is added by a constant, which is defined in the init method.

- The init method consumes a list of objects, each of which can be used to set a variable in the UDF. The input into the init function is a list with parameters in the order they are passed to the function within the Trif acta platform. See *Running Your UDF* below.

Example UDF: AdderUDF

```
package com.trifacta.trifactaudfs;
import java.io.IOException;
import java.util.List;

/**
 * Example UDF. Adds a constant amount to an Integer column.
 */
public class AdderUDF implements TrifactaUDF<Long> {
  private Long _addAmount;
  @Override
  public void init(List<Object> initArgs) {
    if (initArgs.size() != 1) {
      System.out.println("AdderUDF takes in exactly one init argument");
    }
    Long addAmount = (Long) initArgs.get(0);
    _addAmount = addAmount;
  }
  @Override
  public Long exec(List<Object> input) {
    if (input == null) {
      return null;
    }
    if (input.size() != 1) {
      return null;
    }
    return (Long) input.get(0) + _addAmount;
  }
  @SuppressWarnings("rawtypes")
  public Class[] inputSchema() {
    return new Class[]{Long.class};
  }
  @Override
  public void finish() throws IOException {
  }
}
```

Error Handling

The UDF must handle any error that should occur when processing the function. Two ways of dealing with errors:

1. For null data generated in the exec method, a null value can be returned. It appears in the final generated column.
2. Any errors that cause the UDF to stop in the init or exec methods cause an IOException to be thrown. This error signals the platform that an issue occurred with the UDF.

Tip: You can add to the Trifacta logs through Logger. Annotate your exceptions at the appropriate logging level.

Testing the UDF

JUnit can be used to test the UDF. Below are examples of testing the two example UDFs.

Example - JUnit test for Concatenate strings:

ConcatUDF Test

```
@Test
public void concatUDFTest() throws IOException {
```

```
  ConcatUDF concat = new ConcatUDF();
  ArrayList<Object> input = new ArrayList<Object>();
  input.add("hello");
  input.add("world");
  String result = concat.exec(input);
  String expected = "helloworld";
  assertEquals(expected, result);
}
```

Example - JUnit test for Add by constant:

AdderUDF Test

```
@Test
public void adderUDFTest() {
  AdderUDF add = new AdderUDF();
  ArrayList<Object> initArgs = new ArrayList<Object>(1);
  initArgs.add(1L);
  add.init(initArgs);
  ArrayList<Object> inputs1 = new ArrayList<Object>();
  inputs1.add(1L);
  long result = add.exec(inputs1);
  long expected = 2L;
  assertEquals(expected, result);

  ArrayList<Object> inputs2 = new ArrayList<Object>();
  inputs2.add(9000L);
  result = add.exec(inputs2);
  expected = 9001L;
  assertEquals(expected, result);
}
```

Compiling the UDF

After writing the UDF, it must be compiled and included in a JAR before registering it with the platform. To compile and package the function, run the following command from the root directory:

```
gradlew build
```

The UDF code is assembled, and unit tests are executed. If all is well, the following JAR file is created in `build/libs`.

NOTE: Custom UDFs should be compiled to one or more JAR files. Avoid using the example JAR filename, which can be overwritten on upgrade.

JDK version mismatches

To avoid an `Unsupported major.minor version` error during execution, the JDK version used to compile the UDF JAR file should be less than or equal to the JDK version on the Hadoop cluster.

If this is not possible, then set the value of the `Compatibility` properties in the local `build.gradle` file to the JDK version on the Hadoop cluster prior to building the JAR file.

Example:

If the Hadoop cluster is on JDK 1.8, then add the following to the `build.gradle` file:

```
targetCompatibility = '1.8'
sourceCompatibility = '1.8'
```

Registering the UDF

After a function is compiled it must be registered with the platform.:

1. Enable user-defined functions (if not done so already)
2. Path to the JAR file that was generated in the previous steps.
3. The `udfPackages` value should contain the package name where the UDFs can be found.

Example configuration:

To apply this configuration change, login as an administrator to the Trifacta node. Then, edit `trifacta-conf.json`. Some of these settings may not be available through the *Admin Settings Page*. For more information, see *Platform Configuration Methods*.

Example Config

```
...
"feature": {
  "enableUDFTransform": {
    "enabled": true
  }
},
"udf-service": {
  "classpath": "%(topOfTree)s/services/udf-service/build/libs/udf-service.jar:%(topOfTree)s/services/udf-service/build/dependencies/*",
  "additionalJars": [
    "/vagrant/libs/custom-udfs-sdk/build/libs/custom-udfs-example.jar"
  ],
  "udfPackages": [
    "com.trifacta.trifactaudfs"
  ]
},
...
```

Notes:

- Set `enableUDFTransform.enabled` to `true`, which enables UDFs in general.
- Under `udf-service:`
  - specify the full path to the JAR under `additionalJars`
  - append the paths of any extra JAR dependencies that your UDFs require under `classpath`

    NOTE: Do not include any extra JAR dependencies in the `udf-service/build/dependencies` directory, as this directory may be purged at build time.
  - specify the fully qualified package names under `udfPackages`
    - This list contains all fully qualified names of your UDFs.
    - For example. if your UDF is `com.company.ourudfs.MyUDF`, then the package name is the following: `com.company.ourudfs`

Steps:

After modifying the config, the udf-service needs to be restarted.

a. **If you created a new UDF,** restart the Trifacta application:

```
service trifacta restart
```

b. **If you have modified an existing UDF,** restart the UDF service:

NOTE: For an existing UDF, you must rebuild the JAR first. Otherwise, the changes are not recognized during service re-initialization.

```
service java-udf-service restart
```

2. As part of the restart, any newly added Java UDFs are registered with the application.

Enabling UDF service on HDInsight cluster

By default, the UDF service utilizes compression across the websockets when running on the cluster. HDInsight clusters do not support compression on websockets.

To make sure the UDF service works on your HDInsight cluster, please do the following.

Steps:

1. To apply this configuration change, login as an administrator to the Trifacta node. Then, edit `trifacta-conf.json`. Some of these settings may not be available through the *Admin Settings Page*. For more information, see *Platform Configuration Methods*.
2. Locate the `udf-service` configuration.
3. Insert the following extra property in the `udf-service` configuration area:

```
"udf-service": {
   ...
   "jvmOptions": ["-Dorg.apache.tomcat.websocket.DISABLE_BUILTIN_EXTENSIONS=true"],
   ...
}
```

4. Save your changes and restart the platform.

Running Your UDF

For more information on executing your UDF in the Transformer page, see *User-Defined Functions*.

For examples, see *Invoke External Function*.

Troubleshooting

"Websocket Receive()" error in Transformer page UI

If you execute a Java UDF, you may see an error similar to the following in the Transformer page:

```
Please reload page (query execution failed).pp::WebSocket::Receive() error: Unspecified failure.
```

When you check the `udf.log` file on the server, the following may be present:

```
UDFWebsocket closed with status: CloseStatus[code=1009, reason=The decoded text message was too big for the
output buffer and the endpoint does not support partial messages]
```

Solution

The above issue is likely to be caused by the Trifacta Photon running environment sending too much data through the buffer of the UDF's Websocket service. By default, this buffer size is set to 1048576 bytes (1 MB).

The Trifacta Photon running environment processes data through the Websocket service in 1024 (1 K) rows at a time for the input and output columns of the UDF. If the data in the input columns to the UDF or output columns from the UDF exceeds 1 KB (1024 characters) in total size for each row, the default size of the buffer is too small, since the Trifacta Photon running environment processed 1K records at a time (1 K characters * 1 K rows > 1048576). The query then fails.

When setting a new buffer size:

- Assume that 1024 rows are processed from the buffer each time.
- Identify the input columns and output columns for the UDF that is failing.
- Identify the dataset that has the widest columns for both inputs and outputs here.

  Tip: You can use the `LEN` function to do string-based computations of column width. See *LEN Function*.
- Perform the following estimate on the widest set of input and output columns that you are processing:
  - Estimate the total expected number of characters for the input columns of the UDF.
  - Add a 20% buffer to the above estimate.
  - Repeat the above estimate for the widest output columns for the UDF.
  - Set your buffer size to the larger of the two estimates (input columns' width or output columns' width).
- Example: A UDF takes two inputs and produces one output:
  - If each input column is 256 characters, then the size of 1K rows of input would be 256 bytes * 2 (input cols) * 1024 rows = 0.5 MB.
  - If the output of the UDF per row is estimated to be 1024 characters, then the output estimate would be 1024 bytes * 1024 rows = 1MB.
  - So, set the buffer size to be 1 MB + 20% buffer over the larger estimate between input and output. In this example, the buffer size should be 1.2 MB or 1258291 Bytes.

Steps:

1. You can apply this change through the *Admin Settings Page* (recommended) or `trifacta-conf.json`. For more information, see *Platform Configuration Methods*.
2. Change the following setting:

   ```
   "udf-service.outputBufferSize": 1048576,
   ```

3. Save your changes and restart the platform.

Photon crashes during execution of UDF

During the execution of a UDF, the Photon client can crash. Possible errors include:

```
Error in changeCurrentEdit Error: Transformation engine has crashed. Please reload your browser (exit code:
null; message ID: 161)
```

Solution:

This crash can be caused by a number of issues. You can try the following:

1. You can apply this change through the *Admin Settings Page* (recommended) or `trifacta-conf.json`. For more information, see *Platform Configuration Methods*.
2. Bump the value for `udf-service.udfCommunicationTimeout` setting. Raise this value a bit at a time to see if that allows the UDF to execute.

   NOTE: Avoid setting this value to high, which can cause the Java heap size to be exceeded and another Photon crash. Maximum value is `2147483646`.
3. Save your changes and restart the platform.

Create Custom Data Types Using RegEx

Contents:

- *Custom Types Location*
- *Examples*
  - *Defining probabilities*
- *Add custom types to manifest*
- *Enable custom types*
- *Register your custom types*
- *Restart platform*

As needed, you can deploy custom data types into the Trifacta® platform, in which type validation is performed against regular expressions that you specify. This method is most useful for validating against patterns, as opposed to specific values.

After a custom type has been added, it cannot be removed or disabled. Please verify your regular expression before saving the type.

Custom Types Location

On the server hosting the Trifacta platform, type definitions are stored in the following directory:

```
/opt/trifacta/node_modules/jsdata/type-packs/trifacta
```

This directory is referenced as `$CUSTOM_TYPE_DIR` in the steps below.

Before you begin creating custom data types, you should backup the `type-packs/trifacta` directory to a location outside of your Trifacta deployment.

NOTE: The `trifacta-extras` directory in the `type-packs` directory contains experimental custom data types. These data types are not officially supported. Please use with caution.

Directory contents:

- The `dictionaries` sub-directory contains user-defined dictionaries.

  NOTE: Please use the user interface to interact with your dictionaries. See *Custom Type Dialog*.

- The `types` sub-directory contains individual custom data type definitions, each in a separate file.
- The `manifest.json` file contains a JSON manifest of all of standard and custom types in the system.

Examples

Each custom data type is created and stored in a separate file. The following example file contains a regular expression method for validating data against the set of days of the week:

```
{
  "name": "DayOfWeek",
  "prettyName": "Day of Week",
  "category" : "Date/Time",
```

```
  "defaultProbability": 1E-15,
  "testCase": {
    "stripWhitespace": true,
    "regexes": [
      "^(monday|tuesday|wednesday|thursday|friday|saturday|sunday)$",
      "^(mon|tue|wed|thu|fri|sat|sun)$"
    ],
    "probability": 0.001
  }
}
```

Parameters:

Parameter Name	Description
`name`	Internal identifier for the custom type. Must be unique across all standard types and custom types. **NOTE:** You should verify that your data type's `name` value does not conflict with other custom data type names.
`prettyName`	Display name for the custom type.
`category`	The category to assign to the type. The current categories are displayed within the data type drop-down for each column.
`defaultProbability`	Assign a default probability for the custom type. See below.
`testCase`	This block contains the regular expression specification to be applied to the column values.
`stripWhitespace`	When set to `true`, whitespace is removed from any value prior for purposes of validation. The original value is untouched.
`regexes`	This array contains a set of regular expressions that are used to validate the column values. For a regex type, the column value must match with at least one value among the set of expressions. **NOTE:** All match types must be double-escaped in the regex expression. For example, to replicate the `\d` pattern, you must enter: `\\d`. Trifacta Wrangler Enterprise implements a version of regular expressions based off of *RE2* and *PCRE* regular expressions.
`probability`	(optional) Assign an incremental change to the probability when a match is found between a value and one of the regular expressions. See *Defining probabilities* below.

Tip: In the `types` sub-directory, you can review the regex-based types that are provided with the Trifacta platform . While you should not edit these files directly, they may provide some guidance and some regex tips on how to configure your own custom data types.

Defining probabilities

For your custom type, the probability values are used to determine the likelihood that matching values indicate that the entire column is of the custom data type.

- The `defaultProbability` value specifies the baseline probability that a match between a value and one of the regular expressions indicates that the column is the specified type. On a logarithmic scale, values are typically 1E-15 to 1E-20.
- When a value is matched to one of the regular expressions, the `probability` value is used to increment the baseline probability that the next matching value is of the specified type. This value should also be expressed on a logarithmic scale (e.g. `0.001`).
- In this manner, a higher number of matching values increases the probability that the type is also a match to the custom type.

Probabilities become important primarily if you are creating a custom type that is a subset of an existing type. For example, the Email Address custom type is a subset of String type. So, matches for the patterns expressed in the Email Address definition should register a higher `probability` value than the same incremental for the String type definition.

Tip: For custom types that are subsets of other, non-String types, you should lower the `defaultProbability` of the baseline type by a factor of 10 (e.g. 1E-15 to 1E-16) and raise the same probability in the custom type by a factor of 10 (e.g. 1E-14). In this manner, you can give higher probability of matching to these subset types.

Add custom types to manifest

To the `$CUSTOM_TYPE_DIR/manifest.json` file, you must add the filenames of any custom types that you have created and stored in the `types` directory:

```
{
  "types": ["bodies-of-water.json", "dayofweek.json"],
  "dictionaries": ["oceans", "seas"]
}
```

Enable custom types

To enable use of your custom data types in the Trifacta platform, locate and edit `enabledSemanticTypes` property.

You can apply this change through the *Admin Settings Page* (recommended) or `trifacta-conf.json`. For more information, see *Platform Configuration Methods*.

NOTE: Add your entries to the items that are already present in `enabledSemanticTypes`. Do not delete and replace entries.
NOTE: Do not use this parameter to attempt to remove specific data types. Removal of the default types is not supported.

```
"webapp.enabledSemanticTypes": [
    "<CustomTypeName1>",
    "<CustomTypeName2>",
    "<CustomTypeNameN>"
]
```

where:

- `<CustomTypeName1>` corresponds to the internal `name` value for your custom data type.

Register your custom types

To add your custom types to the Trifacta platform, run the following command from the js-data directory:

```
node bin/load-types --manifest ${PATH_TO_MANIFEST_FILE}
```

Restart platform

Restart services. See *Start and Stop the Platform*.

Check for the availability of your types in the column drop-down. See *Column Menus*.

API Reference

This section contains reference information on the REST APIs that are made available by Trifacta® Wrangler Enterprise.

Access to API reference docs

NOTE: URLs to API reference documentation are case-sensitive.

To access the reference documentation for each available API endpoint and method, select **Help menu > API Documentation** in the Trifacta application.

Enable Access

Access tokens required

If the API documentation is not available, a workspace administrator must enable the use of API access tokens.

API tokens enable users and processes to access the REST APIs available through the Trifacta platform.

Tip: Individual users do not need personal API access tokens to use the API documentation. The feature must be enabled.

For more information, see *Workspace Settings Page*.

Enable access through the menu

To enable the Help menu option and access to the API documentation, the following parameter must be enabled.

Steps:

1. Login to the application as an administrator.
2. You can apply this change through the *Admin Settings Page* (recommended) or `trifacta-conf.json`. For more information, see *Platform Configuration Methods*.
3. Locate the following parameter and set it to `true`:

   ```
   "webapp.apiDoc.enabled": true,
   ```

4. Save your changes and restart the platform.

API Endpoint Documentation

You can access API reference documentation through the Trifacta application. In the left navigation bar, select **Help menu > API Documentation**.

API Overview

Contents:

- *Design Overview*
  - *URL Format*
  - *Naming Conventions*
  - *Operations and Methods*
  - *Embedding Associations*
  - *Media Type Headers*
  - *Request identifier*
  - *Authentication*
  - *SSL*
  - *Upload*
  - *Versioning and Endpoint Lifecycle*
- *HTTP Status Codes and Errors*
  - *Caching*
- *Use Cases*
  - *REST API Tasks*
- *About This Documentation*
  - *Applicable API versions*
  - *API examples*

To enable programmatic control over its objects, Trifacta® Wrangler Enterprise supports a range of REST API endpoints across the objects in the platform. This section provides an overview of the API design, methods, and supported use cases.

Design Overview

URL Format

```
<http/https>://<my_server>:<port_number>/<version>/<endpoint>/[resource_id]/[association][?args]
```

Elements in square brackets `[brackets]` are optional.

Element	Description	Example
`<http/https>`	HTTP protocol identifier. The protocol should be `https` in a production environment.	`https`
`<my_server>`	Name of the Trifacta node	`myapp.example.com`
`<port_number>`	Port number over which you access the platform. By default, this value is `3005`.	`3005`
`<version>`	API version number. **NOTE:** Unless stated otherwise, the versions for all API endpoints is `v4`.	`v4`
`<endpoint>`	Name of the API endpoint to use.	`/connections`
`[resource_id]`	Internal identifier for the specific resource requested from the endpoint. This value defines the object against which the requested operation is performed.	`/10`

`[association]`	If applicable, the **association** identifiers the API endpoint that is requested using the context determined by the `<endpoint>` and the `[resource_id]`. Associations can also be referenced by query parameter. See Embedding Associations below.	`/jobGroups`
`[?args]`	In some cases, arguments can be passed to the endpoint in the form of query parameters.	`?arg1=value1&arg2=value2`

Naming Conventions

- Resource names are plural and expressed in camelCase.
- Resource names are consistent between main URL and URL parameter.

v4 conventions

The following conventions apply to v4 and later versions of the APIs:

- Parameter lists are consistently enveloped in the following manner:

```
{ "data": [
    {
      ...
    }
  ]
}
```

- Field names are in `camelCase` and are consistent with the resource name in the URL or with the `embed` URL parameter.
- Foreign keys are represented with identifiers like the following:

```
"creator": {
  "id": 1
},

"updater": {
  "id": 2
},
```

Operations and Methods

Support for basic CRUD (Create, Read, Update, and Delete) operations across most platform objects.

NOTE: Some of these specific operations may not be supported in the current release. For a complete list, see *API Reference*.

Operation	HTTP Method	Example URL	Notes
Create	`POST`	`/v4/jobgroups`	
	`POST`	`/v4/importedDatasets`	
Read	`GET`	`/v4/jobgroups/1`	`1` = internal jobgroup Id
	`GET`	`/v4/importedDatasets/10`	`10` = internal imported dataset Id
List	`GET`	`/v4/flows`	
	`GET`	`/v4/wrangledDatasets`	

Update	PATCH	/v4/wrangledDatasets/10	Partial replacement
	PUT	/v4/outputObjects/1	Partial replacement
Delete	DELETE	/v4/importedDatasets/1	
	DELETE	/v4/outputObjects/10	

Embedding Associations

An association can be referenced using the above URL structuring or by applying the `embed` query parameter as part of the reference to the specific resource. In the following example, the sub-jobs of a jobGroup are embedded in the response for jobGroup=1:

```
https:/myapp.example.com/v4/jobGroups/1?embed=jobs
```

Example response:


```
{
    "id": 1,
    "name": null,
    "description": null,
    "ranfrom": "ui",
    "ranfor": "recipe",
    "status": "Complete",
    "profilingEnabled": true,
    "runParameterReferenceDate": "2019-04-23T17:00:56.000Z",
    "createdAt": "2019-04-23T17:00:56.728Z",
    "updatedAt": "2019-04-23T17:01:03.084Z",
    "jobs": {
        "data": [
            {
                "id": 1,
                "executionLanguage": "<running_environment>",
                "cpJobId": null,
                "templateLocation": null,
                "createdAt": "2019-04-23T17:00:56.952Z",
                "updatedAt": "2019-04-23T17:01:00.670Z",
                "status": "Complete",
                "jobType": "wrangle",
                "sampleSize": 100,
                "percentComplete": 100,
                "lastHeartbeatAt": "2019-04-23T17:00:58.589Z",
                "creator": {
                    "id": 1
                },
                "jobGroup": {
                    "id": 1
                },
                "errorMessage": null,
                "wrangleScript": null,
                "emrcluster": null
            },
            {
                "id": 2,
                "createdAt": "2019-04-23T17:00:57.034Z",
                "updatedAt": "2019-04-23T17:01:02.642Z",
                "status": "Complete",
                "jobType": "filewriter",
                "sampleSize": 100,
                "percentComplete": 100,
                "lastHeartbeatAt": "2019-04-23T17:01:00.922Z",
                "creator": {
                    "id": 1
                },
                "jobGroup": {
                    "id": 1
```

```
                },
                "errorMessage": null,
                "scriptResult": {
                    "id": 1
                },
                "writeSetting": {
                    "id": 2
                }
            },
            {
                "id": 3,
                "createdAt": "2019-04-23T17:00:57.037Z",
                "updatedAt": "2019-04-23T17:01:03.077Z",
                "status": "Complete",
                "jobType": "filewriter",
                "sampleSize": 100,
                "percentComplete": 100,
                "lastHeartbeatAt": "2019-04-23T17:01:00.843Z",
                "creator": {
                    "id": 1
                },
                "jobGroup": {
                    "id": 1
                },
                "errorMessage": null,
                "scriptResult": {
                    "id": 2
                },
                "writeSetting": {
                    "id": 1
                }
            }
        ]
    },
    "workspace": {
        "id": 1
    },
    "creator": {
        "id": 1
    },
    "updater": {
        "id": 1
    },
    "snapshot": {
        "id": 8
    },
    "wrangledDataset": {
        "id": 7
    },
    "flowRun": null
}
```


The `id` value of the association is always included in the response.

Media Type Headers

NOTE: Some endpoints may accept and return a custom media type. These endpoints are documented.

Action	Header	Required?
Client request that expects a response body	request header: `Accept: application/json`	should include
Client request that includes a request body	request header: `Content-Type: application/json`	required
Server response that includes a response body	response header: `Content-Type: application/json`	required

Request identifier

Each request contains a request identifier in the following form:

```
x-trifacta-request-id: <myRequestId>
```

This header also appears as a header in the response.

NOTE: If you have an issue with a specific request, please include the `x-trifacta-request-id` value when you contact *Trifacta Support*.

Tip: You can use the request identifier value to scan the logs to identify technical details for an issue with a specific request.

Authentication

Each call to an API endpoint must include authentication credentials for a user with access to the requested objects. See *https://api.trifacta.com/ee/es.t/index.html#section/Authentication*

SSL

If SSL has been enabled for the Trifacta platform, requests to URL endpoints are automatically redirected to the HTTPS equivalent.

Upload

Single-file upload is supported.

Multi-file upload is not supported.

Versioning and Endpoint Lifecycle

NOTE: API versioning is not synchronized to specific releases of Trifacta Wrangler Enterprise. APIs are designed to be backward compatible.
The v3 endpoints are deprecated. You should move to v4 endpoints at this time.

APIs are designed to be backward compatible so that scripts and other tooling built on a previous version of an endpoint remain valid until the previous version has reached end-of-life. Each API is supported across a window of Trifacta Wrangler Enterprise releases, after which you must reference a newer version of the API.

API endpoint routes support a consistent structuring and do not contain business logic.

Version information is available at the following endpoint:

```
<http/https>://<my_server>:<port_number>/<version>/version
```

HTTP Status Codes and Errors

Request Method	Request Endpoint	HTTP Status Code (success)
`POST`	`/v4/<resource>`	201 Created
`GET`	`/v4/<resource>`	200 OK
`GET`	`/v4/<resource>/<id>`	304 Not Modified when client has cached version. See *Caching* below.

PATCH	/v4/<resource>/<id>	200 OK
PUT	/v4/<resource>/<id>	200 OK
DELETE	/v4/<resource>/<id>	204 No Content

The following error codes can apply to any of the above requests:

NOTE: 5xx status codes may be generated by the server.

HTTP Status Code (client errors)	Notes
400 Bad Request	Potential reasons: • Resource doesn't exist. • Request is incorrectly formatted. • Request contains invalid values.
403 Forbidden	Incorrect permissions to access the Resource.
404 Not Found	Resource cannot be found.
410 Gone	Resource has been previously deleted.
415 Unsupported Media Type	Incorrect `Accept` header.

Caching

When a resource has been cached in the client, the client may set an `If-Modified-Since` header in HTTP date format on the request. If so:

General Response	HTTP Status Code
Returns full modified resource	200 OK
Returns an empty response for unmodified resource	304 Not Modified

Use Cases

REST API Tasks

By chaining together sequences of calls to API endpoints, you can create, read, update, and delete objects using identifiers accessible through the returned JSON. For more information, see *API Reference*.

For more information on endpoint workflows, see *API Workflows*.

About This Documentation

Applicable API versions

Unless otherwise noted, the documentation and examples apply to latest version of the platform APIs.

API examples

Examples may require modification to work in your environment.

NOTE: Some examples may not be reflective of your environment. Examples may contain references to examples or features not supported in your environment.

- Reference examples are included as part of endpoint documentation. See *API Reference*.
- Usage examples are included in the workflows. See *API Workflows*.

Manage API Access Tokens

Contents:

- *Enable*
- *Generate New Token*
  - *Via API*
  - *Via UI*
- *Use Token*
- *List Tokens*
- *Renew Token*
- *Delete Token*

This section provides some workflow information for how to use API access tokens as part of your API projects in Trifacta® Wrangler Enterprise. An **access token** is a hashed string that enables authentication when submitted to any endpoint. Access tokens limit exposure of clear-text authentication values and provide an easy method of managing authentication outside of the browser.

Notes:

- An access token is linked to its creator and can be generated by submitting a username/password combination or another valid token from the same user.
  - If a token is created for userA, userB can be provided the token to impersonate userA.
  - You cannot create access tokens for users without their authentication credentials.
  - Changes to passwords do not affect tokens.
- After a token has been created, it cannot be modified or extended.
  - You can create an unlimited number of tokens.
- Access tokens can be used for authentication with any supported version of the APIs.

Enable

This feature must be enabled in your instance of the platform. For more information, see *Enable API Access Tokens*.

Generate New Token

API access tokens must be created.

NOTE: The first time that you request a new API token, you must submit a separate form of authentication to the endpoint. To generate new access tokens after you have created one, you can use a valid access token if you have one.

Via API

For more information, see *https://api.trifacta.com/ee/es.t/index.html#operation/getApiAccessToken*

Via UI

Tokens can be generated from the web application.

Steps:

1. Login to the Trifacta application.
2. From the left nav bar, select **User menu > Preferences > Access Tokens**.
3. Click **Generate New Token**.
4. Specify the number of days for how long the token should live. Enter `-1` to create a non-expiring token.
5. Add a user-friendly description if desired.
6. Click **Generate**.

NOTE: Copy the value of the token to the clipboard and store it in a secure location for use with your scripts.

Tip: If you wish to manage your token via the APIs, you should copy the Token ID value, too. The Token ID can always be retrieved from the Trifacta application.

For more information, see *Access Tokens Page*.

Use Token

After a token has been acquired, it must be included in each request to the server, for as long as it is valid.

NOTE: After a token has been created, it cannot be extended or modified.
NOTE: API access tokens are not used by users in the Trifacta application.

NOTE: When using the APIs in SSO environments, API access tokens work seamlessly with platform-native versions of SAML and LDAP-AD. They do not work with the reverse proxy SSO methods. For more information, see *https://api.trifacta.com/ee/es.t/index.html#section/Authentication*

After you have acquired the token, you submit it with each API request to the platform.

Example - cURL:

```
curl http://tri.example.com:3005/v4/jobs -X GET -H "Authorization: Bearer (tokenValue)"
```

where:

- `(tokenValue)` is the value returned for the token when it was created.

Example - REST client:

If you are submitting your API calls through a REST client, the Authorization header must be specified as follows:

```
Authorization: Bearer (tokenValue)
```

List Tokens

NOTE: For security reasons, you cannot acquire the actual token through any of these means.
Tip: You can see all of your current and expired tokens through the Trifacta application. See *Access Tokens Page*.

Endpoint	Description
https://api.trifacta.com/ee/es.t/index.html#operation/listApiAccessTokens	List all access tokens for your user account.
https://api.trifacta.com/ee/es.t/index.html#operation/getApiAccessToken	List your access token for the specified token ID.

Renew Token

New tokens can be acquired at any time.

NOTE: It is the responsibility of the user to acquire a new API token before the current one expires. If a token is permitted to expire, a request for a new token must include userId and password information.

- See *https://api.trifacta.com/ee/es.t/index.html#operation/createApiAccessToken*
- See *Access Tokens Page*.

Delete Token

- **Via API:** Acquire the `tokenId` value for the token and use the delete endpoint. See *https://api.trifacta.com/ee/es.t/index.html#operation/deleteApiAccessToken*
- **Via UI:** In the Access Tokens page, select **Delete Token...** from the context menu for the token listing. See *Access Tokens Page*.

API Workflows

In this section, you can review examples of how to execute workflows using one or more of the available REST APIs.

API Workflow - Develop a Flow

Contents:

- *Overview*
  - *Example Datasets*
- *Step - Create Containing Flow*
- *Step - Create Datasets*
- *Step - Wrangle Data*
- *Step - Create Output Objects*
- *Step - Run Job*
- *Step - Monitoring Your Job*
- *Step - Re-run Job*

Overview

This example walks through the process of creating, identifying, and executing a job through automated methods. For this example, these tasks are accomplished using the following methods:

NOTE: This API workflow applies to a Development instance of the Trifacta® platform, which is the default platform instance type. For more information on Development and Production instance, see *Overview of Deployment Manager*.

1. **Locate or create flow.** The datasets that you wrangle must be contained within a flow. You can add them to an existing flow or create a new one through the APIs.
2. **Create dataset.** Through the APIs, you create an imported dataset from an asset that is accessible through one of the established connections. Then, you create the recipe object through the API.
   a. For the recipe, you must retrieve the internal identifier.
   b. Through the application, you modify the recipe for the dataset.
3. **Automate job execution.** Using the APIs, you can automate execution of the wrangling of the dataset.
   a. As needed, this job can be re-executed on a periodic basis or whenever the source files are updated.

Example Datasets

In this example, you are attempting to wrangle monthly point of sale (POS) data from three separate regions into a single dataset for the state. This monthly data must be enhanced with information about the products and stores in the state. So, the example has a combination of transactional and reference data, which must be brought together into a single dataset.

Tip: To facilitate re-execution of this job each month, the transactional data should be stored in a dedicated directory. This directory can be overwritten with next month's data using the same filenames. As long as the new files are structured in an identical manner to the original ones, the new month's data can be processed by re-running the API aspects of this workflow.

Example Files:

The following files are stored on your HDFS deployment:

Path and Filename	Description
`hdfs:///user/pos/POS-r01.txt`	Point of sale transactions for Region 1.
`hdfs:///user/pos/POS-r02.txt`	Point of sale transactions for Region 2.
`hdfs:///user/pos/POS-r03.txt`	Point of sale transactions for Region 3.

`hdfs:///user/ref/REF_PROD.txt`	Reference data on products for the state.
`hdfs:///user/ref/REF_CAL.txt`	Reference data on stores in the state.

NOTE: The reference and transactional data are stored in separate directories. In this case, you can assume that the user has read access through his Trifacta account to these directories, although this access must be enabled and configured for real use cases.

Base URL:

For purposes of this example, the base URL for the Trifacta platform is the following:

```
http://www.example.com:3005
```

Step - Create Containing Flow

To begin, you must locate a flow or create a flow through the APIs to contain the datasets that you are importing.

NOTE: You cannot add datasets to the flow through the `flows` endpoint. Moving pre-existing datasets into a flow is not supported in this release. Create or locate the flow first and then when you create the datasets, associate them with the flow at the time of creation.

- See *https://api.trifacta.com/ee/es.t/index.html#operation/createImportedDataset*
- See *https://api.trifacta.com/ee/es.t/index.html#operation/createWrangledDataset*

Locate:

NOTE: If you know the display name value for the flow and are confident that it is not shared with any other flows, you can use the APIs to retrieve the flowId. See *https://api.trifacta.com/ee/es.t/index.html#operation/listFlows*

1. Login through the application.
2. In the Flows page, select or create the flow to contain the above datasets.
3. In the Flow Details page for that flow, locate the flow identifier in the URL:

Flow Details URL	`http://www.example.com:3005/flows/10`
Flow Id	`10`

4. Retain this identifier for later use.

Create:

1. Through the APIs, you can create a flow using the following call:

Endpoint	`http://www.example.com:3005/v4/flows`
Authentication	Required
Method	POST
Request Body	`{ "name": "Point of Sale - 2013", "description": "Point of Sale data for state" }`

2. The response should be status code `201 - Created` with a response body like the following:

```
{
  "id": 10,
  "updatedAt": "2017-02-17T17:08:57.848Z",
  "createdAt": "2017-02-17T17:08:57.848Z",
  "name": "Point of Sale - 2013",
  "description": "Point of Sale data for state",
  "creator": {
    "id": 1
  },
  "updater": {
    "id": 1
  },
  "workspace": {
    "id": 1
  }
}
```

3. Retain the flow identifier (`10`) for later use.

For more information, see *https://api.trifacta.com/ee/es.t/index.html#operation/createFlow*

Checkpoint: You have identified or created the flow to contain your dataset or datasets.

Step - Create Datasets

To create datasets from the above sources, you must:

1. Create an imported dataset for each file.
2. For each imported dataset, create a recipe, which can be used to transform the imported dataset.

The following steps describe how to complete these actions via API for a single file.

Steps:

1. To create an imported dataset, you must acquire the following information about the source. In the above example, the source is the `POS-r01.txt` file.
    a. uri
    b. name
    c. description
    d. bucket (if a file stored on S3)
2. Construct the following request:

Endpoint	`http://www.example.com:3005/v4/importedDatasets`
Authentication	Required
Method	POST
Request Body	`{ "uri": "hdfs:///user/pos/POS-r01.txt", "name": "POS-r01.txt", "description": "POS-r01.txt" }`

3. You should receive a `201 - Created` response with a response body similar to the following:

```
{
  "id": 8,
  "size": "281032",
  "uri": "hdfs:///user/pos/POS-r01.txt",
  "dynamicPath": null,
```

```
  "bucket": null,
  "isSchematized": true,
  "isDynamic": false,
  "disableTypeInference": false,
  "updatedAt": "2017-02-08T18:38:56.640Z",
  "createdAt": "2017-02-08T18:38:56.560Z",
  "parsingScriptId": {
    "id": 14
  },
   "runParameters": {
        "data": []
    },
    "name": "POS-r01.txt",
    "description": "POS-r01.txt",
    "creator": {
        "id": 1
    },
    "updater": {
        "id": 1
    },
    "connection": null
}
```

4. You must retain the `id` value so you can reference it when you create the recipe.
5. See *https://api.trifacta.com/ee/es.t/index.html#operation/createImportedDataset*
6. Next, you create the recipe. Construct the following request:

Endpoint	`http://www.example.com:3005/v4/wrangledDatasets`
Authentication	Required
Method	POST
Request Body	`{ "name":"POS-r01", "importedDataset": { "id":8 }, "flow": { "id":10 } }`

7. You should receive a `201 - Created` response with a response body similar to the following:

```
{
    "id": 23,
    "wrangled": true,
    "updatedAt": "2018-02-06T19:59:22.735Z",
    "createdAt": "2018-02-06T19:59:22.698Z",
    "name": "POS-r01",
    "active": true,
    "referenceInfo": null,
    "activeSample": {
        "id": 23
    },
    "creator": {
        "id": 1
    },
    "updater": {
        "id": 1
    },
    "recipe": {
        "id": 23
    },
    "flow": {
        "id": 10
```

```
    }
}
```

8. From the recipe, you must retain the value for the `id`. For more information, see *https://api.trifacta.com/ee/es.t/index.html#operation/createWrangledDataset*
9. Repeat the above steps for each of the source files that you are adding to your flow.

Checkpoint: You have created a flow with multiple imported datasets and recipes.

Step - Wrangle Data

After you have created the flow with all of your source datasets, you can wrangle the base dataset to integrate all of the source into it.

Steps for Transactional data:

1. Open the `POS-r01` dataset. It's loaded in the Transformer page.
2. To chain together the other transactional data into this dataset, you use a `union` transform. In the Search panel, enter `union` in the textbox and press `ENTER`.
3. In the Union page:
    a. Click **Add datasets**.
    b. Select the other two transactional datasets: `POS-r02` and `POS-r03`.

       NOTE: When you join or union one dataset into another, changes made in the joined dataset are automatically propagated to the dataset where it has been joined.
    c. Add the datasets and align by name.
    d. Check the dataset names and fields. If all looks well, click **Add to Recipe**.

Steps for reference data:

In the columns `Store_Nbr` and `Item_Nbr` are unique keys into the `REF_CAL` and `REF_PROD` datasets, respectively. Using the Join window, you can pull in the other fields from these reference datasets based on these unique keys.

1. Open the `POS-r01` dataset.
2. In Search panel, enter `join datasets` for the transform. The Join window opens.
3. Select the `RED_PROD` dataset. Click **Accept**. Click **Next**.
4. Review the two keys to verify that they are the proper columns on which to structure the join. Click **Next**.
5. Click the All tab. Select all fields to add. Click **Review**.
6. After reviewing your join, click **Add to Recipe**.
7. For each `Item_Nbr` value that has a matching `ITEM_NBR` value in the reference dataset, all of the other reference fields are pulled into the `POS-r01` dataset.

You can repeat the above general process to integrate the reference data for stores.

Checkpoint: You have created a flow with multiple datasets and have integrated all of the relevant data into a single dataset.

Step - Create Output Objects

Before you run a job, you must define output objects, which specify the following:

- Running environment where the job is executed
- Profiling on or off
- outputObjects have the following objects associated with them:
    - **writeSettings:** These objects define the file-based outputs that are produced for the output object
    - **publications:** These objects define the database target, table, and other settings for publication to a relational datastore.

NOTE: You can continue with this workflow without creating outputObjects yet. In this workflow, overrides are applied during the job definition, so you don't have to create the outputObjects and writeSettings at this time.

For more information on creating outputObjects, writeSettings, and publications, see *API Workflow - Manage Outputs*.

Step - Run Job

Through the APIs, you can specify and run a job. In the above example, you must run the job for the terminal dataset, which is `POS-r01` in this case. This dataset contains references to all of the other datasets. When the job is run, the recipes for the other datasets are also applied to the terminal dataset, which ensures that the output reflects the proper integration of these other datasets into `POS-r01`.

NOTE: In the following example, writeSettings have been specified as overrides in the job definition. These overrides are applied for this job run only. If you need to re-run the job with these settings, you must either 1) re-apply the overrides or 2) create the writeSettings objects.For more information, see *API Workflow - Manage Outputs*.

Steps:

1. Acquire the internal identifier for the recipe for which you wish to execute a job. In the previous example, this identifier was `23`.
2. Construct a request using the following:

Endpoint	`http://www.example.com:3005/v4/jobGroups`
Authentication	Required
Method	POST

Request Body:

```
{
  "wrangledDataset": {
    "id": 23
  },
  "overrides": {
    "execution": "photon",
    "profiler": true,
    "writesettings": [
      {
        "path": "hdfs://hadoop:50070/trifacta/queryResults/admin@example.com/POS-r01.csv",
        "action": "create",
        "format": "csv",
        "compression": "none",
        "header": false,
        "asSingleFile": false
      }
    ]
  },
  "ranfrom": null
}
```

3. In the above example, the specified job has been launched for recipe `23` to execute on the Trifacta Photon running environment with profiling enabled.
    a. Output format is CSV to the designated path. For more information on these properties, see *https://api.trifacta.com/ee/es.t/index.html#operation/runJobGroup*
    b. Output is written as a new file with no overwriting of previous files.

4. A response code of `201 - Created` is returned. The response body should look like the following:

```
{

    "sessionId": "79276c31-c58c-4e79-ae5e-fed1a25ebca1",
    "reason": "JobStarted",
    "jobGraph": {
        "vertices": [
            21,
            22
        ],
        "edges": [
            {
                "source": 21,
                "target": 22
            }
        ]
    },
    "id": 3,
    "jobs": {
        "data": [
            {
                "id": 21
            },
            {
                "id": 22
            }
        ]
    }
}
```

5. Retain the `id` value, which is the job identifier, for monitoring.

Step - Monitoring Your Job

You can monitor the status of your job through the following endpoint:

Endpoint	`http://www.example.com:3005/v4/jobGroups/<id>/status`
Authentication	Required
Method	`GET`
Request Body	None.

When the job has successfully completed, the returned status message is the following:

```
"Complete"
```

For more information, see *https://api.trifacta.com/ee/es.t/index.html#operation/runJobGroup*

Step - Re-run Job

In the future, you can re-run the job exactly as you specified it by executing the following call:

Tip: You can swap imported datasets before re-running the job. For example, if you have uploaded a new file, you can change the primary input dataset for the dataset and then use the following API call to re-run the job as specified. See *https://api.trifacta.com/ee/es.t/index.html#operation/updateInputDataset*

Endpoint	`http://www.example.com:3005/v4/jobGroups`

Authentication	Required
Method	POST
Request Body	{ "wrangledDataset": { "id": 23 }, "overrides": { "execution": "photon", "profiler": true, "writesettings": [{ "path": "hdfs://hadoop:50070/trifacta/queryResults/admin@example.com/POS-r01.csv", "action": "create", "format": "csv", "compression": "none", "header": false, "asSingleFile": false }] }, "ranfrom": null }

The job is re-run as it was previously specified.

API Workflow - Deploy a Flow

Contents:

- *Overview*
  - *Pre-requisites*
  - *Workflow*
- *Step - Get Flow Id*
- *Step - Export a Flow*
- *Step - Create Deployment*
- *Step - Create Connection*
- *Step - Create Import Rules*
- *Step - Import Package to Create Release*
- *Step - Activate Release*
- *Step - Run Deployment*
- *Step - Iterate*
- *Step - Set up Production Schedule*

Overview

In this workflow, you learn how to deploy a flow in development to a production instance of the platform. After you have created and finished a flow in a Development (Dev) instance, you can deploy it to an environment designed primarily for production execution of jobs for finished flows (Prod instance). For more information on managing these deployments, see *Overview of Deployment Manager*.

Pre-requisites

Finished flow: This example assumes that you have finished development of a flow with the following characteristics:

- Single dataset imported from a table through a Redshift connection
- Single JSON output

Separate Dev and Prod instances: Although it is possible to deploy flows to the same instance in which they are developed, this example assumes that you are deploying from a Dev instance to a completely separate Prod instance. The following implications apply:

- Separate user accounts to access Dev (User1) and Prod (Admin2) instances.

  Tip: You should do all of your recipe development and testing in Dev/Test. Avoid making changes in a Prod environment.
  NOTE: Although these are separate user accounts, the assumption is that the same admin-level user is using these accounts through the APIs.
- New connections must be created in the Prod instance to access the production version of the database table.

Workflow

In this example, your environment contains separate Dev and Prod instances, each of which has a different set of users.

Item	Dev	Prod
Environment	http://wrangle-dev.example.com:3005 **Tip:** Dev environment work can be done through the UI, which may be easier.	http://wrangle-prod.example.com:3005

User	User1 **NOTE:** User1 has no access to Prod.	Admin2
Source DB	devWrangleDB	prodWrangleDB
Source Table	Dev-Orders	Prod-Orders
Connection Name	Dev Redshift Conn	Prod Redshift Conn

Example Flow:

`User1` is creating a flow, which is used to wrangle weekly batches of orders for the enterprise. The flow contains:

- A single imported dataset that is created from a Redshift database table.
- A single recipe that modifies the imported dataset.
- A single output to a JSON file.
- Production data is hosted in a different Redshift database. So, the Prod connection is different from the Dev connection.

Steps:

1. **Build in Dev instance:** User1 creates the flow and iterates on building the recipe and running jobs until a satisfactory output can be generated in JSON format.
2. **Export:** When User1 is ready to push the flow to production, User1 exports the flow and downloads the export package ZIP file to the local desktop.
3. **Deploy to Prod instance:**
    a. Admin2 creates a new deployment in the Prod instance.
    b. Admin2 creates a new connection (Prod Redshift Conn) in the Prod instance.
    c. Admin2 creates new import rules in the Prod instance to map from the old connection (Dev Redshift Conn) to the new one (Prod Redshift Conn).
    d. Admin2 uploads the export ZIP package.
4. **Test deployment:** Through Flow View in the Prod instance, Admin2 runs a job. The results look fine.
5. **Set schedule:** Using cron, Admin2 sets a schedule to run the active release for this deployment once per week.
    a. Each week, the Prod-Orders table must be refreshed with data.
    b. The dataset is now operational in the Prod environment.

Step - Get Flow Id

The first general step is for the Dev user (User1) to get the flowId and export the flow from the Dev instance.

Steps:

Tip: If it's easier, you can gather the flowId from the user interface in Flow View. In the following example, the flowId is 21:

```
http://www.wrangle-dev.example.com:3005/flows/21
```

1. Through the APIs, you can create a flow using the following call:

Endpoint	`http://www.wrangle-dev.example.com:3005/v4/flows`
Authentication	Required
Method	`GET`
Request Body	None.

2. The response should be status code `200 - OK` with a response body like the following:


```
{    "data": [
        {
            "id": 21,
            "name": "Intern Training",
            "description": "null",
            "createdAt": "2019-01-08T18:14:37.851Z",
            "updatedAt": "2019-01-08T18:57:26.824Z",
            "creator": {
                "id": 2
            },
            "updater": {
                "id": 2
            },
            "folder": {
                "id": 1
            }
            "workspace": {
                "id": 1
            }
        },
        {
            "id": 19,
            "name": "example Flow",
            "description": null,
            "createdAt": "2019-01-08T17:25:21.392Z",
            "updatedAt": "2019-01-08T17:30:30.959Z",
            "creator": {
                "id": 2
            },
            "updater": {
                "id": 2
            },
            "folder": {
                "id": 4
            }
            "workspace": {
                "id": 1
            }
        }
    ]
}
```


3. Retain the flow identifier (`21`) for later use.

Checkpoint: You have identified the flow to export.

For more information, see *https://api.trifacta.com/ee/es.t/index.html#operation/listFlows*

Step - Export a Flow

Export the flow to your local desktop.

Tip: This step may be easier to do through the UI in the Dev instance.

Steps:

1. Export flowId=21:

Endpoint	`http://www.wrangle-dev.example.com:3005/v4/flows/21/package`
Authentication	Required

Method	GET
Request Body	None.

2. The response should be status code `200 - OK`. The response body is the flow itself.
3. Download and save this file to your local desktop. Let's assume that the filename you choose is `flow-WrangleOrders.zip`.

For more information, see *https://api.trifacta.com/ee/es.t/index.html#operation/getFlowPackage*

Step - Create Deployment

In the Prod environment, you can create the deployment from which you can manage the new flow. Note that the following information has changed for this environment:

Item	Prod env value
userId	Admin2
baseURL	http://www.wrangle-prod.example.com:3005

Steps:

1. Through the APIs, you can create a deployment using the following call:

Endpoint	`http://www.wrangle-prod.example.com:3005/v4/deployments`
Authentication	Required **NOTE:** Username and password credentials must be submitted for the `Admin2` account.
Method	POST
Request Body	`{ "name": "Production Orders" }`

2. The response should be status code `201 - Created` with a response body like the following:

3. Retain the deploymentId (3) for later use.

For more information, see *https://api.trifacta.com/ee/es.t/index.html#operation/createDeployment*

Step - Create Connection

When a flow is exported, its connections are not included in the export. Before you import the flow into a new environment:

- Connections must be created or recreated in the Prod environment. In some cases, you may need to point to production versions of the data contained in completely different databases.
- Rules must be created to remap the connection to use in the imported flow.

This section and the following step through these processes.

Steps:

1. From the Dev environment, you collect the connection information for the flow:

Endpoint	`http://www.wrangle-dev.example.com:3005/v4/connections`
Authentication	Required **NOTE:** Username and password credentials must be submitted for the `User1` account.
Method	`GET`
Request Body	None.

2. The response should be status code `200 - Ok` with a response body like the following:


```
{
    "data": [
        {
            "id": 9,
            "host": "dev-redshift.example.com",
            "port": 5439,
            "vendor": "redshift",
            "params": {
                "connectStrOpts": "",
                "defaultDatabase": "devWrangleDB",
                "extraLoadParams": "BLANKSASNULL EMPTYASNULL TRIMBLANKS TRUNCATECOLUMNS"
            },
            "ssl": false,
            "vendorName": "redshift",
            "name": "Dev Redshift Conn",
            "description": "",
            "type": "jdbc",
            "isGlobal": true,
            "credentialType": "custom",
            "credentialsShared": true,
            "uuid": "b8014610-ce56-11e7-9739-27deec2c3249",
            "disableTypeInference": false,
            "createdAt": "2017-11-21T00:55:50.770Z",
            "updatedAt": "2017-11-21T00:55:50.770Z",
            "credentials": [
                {
                    "user": "devDBuser"
                }
            "creator": {
                "id": 2
            },
            "updater": {
                "id": 2
            },
            "workspace": {
                "id": 1
            }
            ]
        }
    ],
    "count": {
        "owned": 1,
        "shared": 0,
        "count": 1
    }
}
```

3. You retain the above information for use in Production.
4. In the Prod environment, you create the new connection using the following call:

Endpoint	`http://www.wrangle-prod.example.com:3005/v4/connections`
Authentication	Required **NOTE:** Username and password credentials must be submitted for the `Admin2` account.
Method	POST
Request Body	<pre>{ "host": "prod-redshift.example.com", "port": 1433, "vendor": "redshift", "params": { "connectStrOpts": "", "defaultDatabase": "prodWrangleDB", "extraLoadParams": "BLANKSASNULL EMPTYASNULL TRIMBLANKS TRUNCATECOLUMNS" }, "vendorName": "redshift", "name": "Redshift Conn Prod", "description": "", "isGlobal": true, "type": "jdbc", "ssl": false, "credentialType": "custom", "credentials": [{ "username": "prodDBUser", "password": "<password>" }] }</pre>

5. The response should be status code `201 - Created` with a response body like the following:


```
{
  "id": 12,
  "host": "prod-redshift.example.com",
  "port": 5439,
  "vendor": "redshift",
     "params": {
       "connectStrOpts": "",
       "defaultDatabase": "prodWrangleDB",
       "extraLoadParams": "BLANKSASNULL EMPTYASNULL TRIMBLANKS TRUNCATECOLUMNS"
     },
  "ssl": false,
  "name": "Redshift Conn Prod",
  "description": "",
  "type": "jdbc",
  "isGlobal": true,
  "credentialType": "custom",
  "credentialsShared": true,
  "uuid": "fa7e06c0-0143-11e8-8faf-27c0392328c5",    "disableTypeInference": false,
  "createdAt": "2018-01-24T20:20:11.181Z",
  "updatedAt": "2018-01-24T20:20:11.181Z",
  "credentials": [
      {
          "username": "prodDBUser"
      }
  ],
  "creator": {
      "id": 2
  },
  "updater": {
      "id": 2
```

```
  }
}
```

6. When you hit the `/v4/connections` endpoint again, you can retrieve the connectionId for this connection. In this case, let's assume that the connectionId value is `12`.

See *https://api.trifacta.com/ee/es.t/index.html#operation/createConnection*

Step - Create Import Rules

Now that you have defined the connection to use to acquire the production data from within the production environment, you must create an import rule to remap from the Dev connection to the Prod connection within the flow definition. This rule is applied during the import process to ensure that the flow is working after it has been imported.

In this case, you must remap the `uuid` value for the Dev connection, which is written into the flow definition, with the connection Id value from the Prod instance.

For more information on import rules, see *Define Import Mapping Rules*.

Steps:

1. From the Dev environment, you collect the connection information for the flow:

Endpoint	`http://www.wrangle-dev.example.com:3005/v4/connections`
Authentication	Required **NOTE:** Username and password credentials must be submitted for the `User1` account.
Method	`GET`
Request Body	None.

2. The response should be status code `200 - Ok` with a response body like the following:

```
{
    "data": [
        {
            "id": 9,
            "host": "dev-redshift.example.com",
            "port": 5439,
            "vendor": "redshift",
            "params": {
                "connectStrOpts": "",
                "defaultDatabase": "devWrangleDB",
                "extraLoadParams": "BLANKSASNULL EMPTYASNULL TRIMBLANKS TRUNCATECOLUMNS"
            },
            "ssl": false,
            "vendorName": "redshift",
            "name": "Dev Redshift Conn",
            "description": "",
            "type": "jdbc",
            "isGlobal": true,
            "credentialType": "custom",
            "credentialsShared": true,
            "uuid": "b8014610-ce56-11e7-9739-27deec2c3249",
            "disableTypeInference": false,
            "createdAt": "2017-11-21T00:55:50.770Z",
            "updatedAt": "2017-11-21T00:55:50.770Z",
            "credentials": [
                {
                    "user": "devDBuser"
                }
            "creator": {
```

```
                "id": 2
            },
            "updater": {
                "id": 2
            },
            "workspace": {
                "id": 1
            }
            ]
        }
    ],
    "count": {
        "owned": 1,
        "shared": 0,
        "count": 1
    }
}
```

3. From the above information, you retain the following, which uniquely identifies the connection object, regardless of the instance to which it belongs:

```
"uuid": "b8014610-ce56-11e7-9739-27deec2c3249",
```

4. Against the Prod environment, you now create an import mapping rule:

Endpoint	`http://www.wrangle-prod.example.com:3005/v4/deployments/3/objectImportRules`
Authentication	Required
Method	`PATCH`

Request Body:

```
[{"tableName":"connections","onCondition":{"uuid": "b8014610-ce56-11e7-9739-27deec2c3249"},"withCondition":{"id":12}}]
```

5. The response should be status code `200 - Ok` with a response body like the following:

```
{
    "deleted": []
}
```

Since the method is a `PATCH`, you are updating the rules set that applies to all imports for this deployment. In this case, there were no pre-existing rules, so the response indicates that nothing was deleted. If another set of import rules is submitted, then the one you just created is deleted.

See *https://api.trifacta.com/ee/es.t/index.html#operation/updateObjectImportRules*

See *https://api.trifacta.com/ee/es.t/index.html#operation/updateValueImportRules*

Step - Import Package to Create Release

You are now ready to import the package to create the release.

Steps:

1. Against the Prod environment, you now import the package:

Endpoint	`http://www.wrangle-prod.example.com:3005/v4/deployments/3/releases`
Authentication	Required
Method	POST
Request Body	The request body must include the following key and value combination submitted as form data: **key**: data **value**: "@path-to-flow-WrangleOrders.zip"

2. The response should be status code `201 - Created` with a response body like the following:

```
{    "importRuleChanges": {
        "object": [{"tableName":"connections","onCondition":{"uuid": "b8014610-ce56-11e7-9739-
27deec2c3249"},"withCondition":{"id":12}}],
        "value": []
    },
    "flowName": "Wrangle Orders"
}
```

See *https://api.trifacta.com/ee/es.t/index.html#operation/importPackageForDeployment*

Step - Activate Release

When a package is imported into a release, the release is automatically set as the active release for the deployment. If at some point in the future, you need to change the active release, you can use the following endpoint to do so.

Steps:

1. Against the Prod environment, use the following endpoint:

Endpoint	`http://www.wrangle-prod.example.com:3005/v4/releases/5`
Authentication	Required
Method	PATCH
Request Body	`{ "active": true }`

2. The response should be status code `200 - OK` with a response body like the following:

```
{
    "id": 3,
    "updater": {
        "id": 3,
    }
    "updatedAt": "2017-11-28T00:06:12.147Z"
}
```

See *https://api.trifacta.com/ee/es.t/index.html#operation/patchRelease*

Step - Run Deployment

You can now execute a test run of the deployment to verify that the job executes properly.

NOTE: When you run a deployment, you run the primary flow in the active release for that deployment. Running the flow generates the output objects for all recipes in the flow.
NOTE: For datasets with parameters, you can apply parameter overrides through the request body through the following API call. For more information, see *https://api.trifacta.com/ee/es.t/index.html#operation/runDeployment*

Steps:

1. Against the Prod environment, use the following endpoint:

Endpoint	`http://www.wrangle-prod.example.com:3005/v4/deployments/3/run`
Authentication	Required
Method	POST
Request Body	None.

2. The response should be status code `201 - Created` with a response body like the following:

See *https://api.trifacta.com/ee/es.t/index.html#operation/runDeployment*

Step - Iterate

If you need to make changes to fix issues related to running the job:

- Recipe changes should be made in the Dev environment and then passed through export and import of the flow into the Prod deployment.
- Connection issues:
  - Check Flow View in the Prod instance to see if there are any red dots on the objects in the package. If so, your import rules need to be fixed.
  - Verify that you can import data through the connection.
- Output problems could be related to permissions on the target location.

Step - Set up Production Schedule

When you are satisfied with how the production version of your flow is working, you can set up periodic schedules using a third-party tool to execute the job on a regular basis.

The tool must hit the Run Deployment endpoint and then verify that the output has been properly generated.

API Workflow - Run Job

Contents:

- *Pre-requisites*
- *Step - Run Job*
- *Step - Monitoring Your Job*
- *Step - Re-run Job*
- *Step - Run Job with Overrides - Files*
  - *Input file overrides*
  - *Output file overrides*
- *Step - Run Job with Overrides - Tables*
- *Step - Run Job with Overrides - Webhooks*
- *Step - Run Job with Parameter Overrides*
- *Step - Spark Job Overrides*

This section describes how to run a job using the APIs available in Trifacta® Wrangler Enterprise.

A note about API URLs:

In the listed examples, URLs are referenced in the following manner:

```
<protocol>://<platform_base_url>/
```

In your product, these map references map to the following:

```
<http or https>://<hostname>:<port_number>/
```

For more information, see *API Reference*.

Pre-requisites

Before you begin, you should verify the following:

1. **Get authentication credentials**. As part of each request, you must pass in authentication credentials to the platform.

   Tip: The recommended method is to use an access token, which can be generated from the Trifacta application. For more information, see *Access Tokens Page*.

   For more information, see *https://api.trifacta.com/ee/es.t/index.html#section/Authentication*
2. **Verify job execution.** Run the desired job through the Trifacta application and verify that the output objects are properly generated.
3. **Acquire recipe (wrangled dataset) identifier.** In Flow View, click the icon for the recipe whose outputs you wish to generate. Acquire the numeric value for the recipe from the URL. In the following, the recipe Id is 28629:

   ```
   http://<platform_base_url>/flows/5479?recipe=28629&tab=recipe
   ```
4. **Create output object.** A recipe must have at least one output object created for it before you can run a job via APIs. For more information, see *Flow View Page*.

If you wish to apply overrides to the inputs or outputs of the recipe, you should acquire those identifiers or paths now. For more information, see "Run Job with Parameter Overrides" below.

Step - Run Job

Through the APIs, you can specify and run a job. To run a job with all default settings, construct a request like the following:

NOTE: A `wrangledDataset` is an internal object name for the recipe that you wish to run. Please see previous section for how to acquire this value.

Endpoint	`<protocol>://<platform_base_url>/v4/jobGroups`
Authentication	Required
Method	`POST`
Request Body	<pre>{ "wrangledDataset": { "id": 28629 } }</pre>
Response Code	`201 - Created`
Response Body	<pre>{ "sessionId": "79276c31-c58c-4e79-ae5e-fed1a25ebca1", "reason": "JobStarted", "jobGraph": { "vertices": [21, 22], "edges": [{ "source": 21, "target": 22 }] }, "id": 961247, "jobs": { "data": [{ "id": 21 }, { "id": 22 }] } }</pre>

If the `201` response code is returned, then the job has been queued for execution.

Tip: Retain the `id` value in the response. In the above, `961247` is the internal identifier for the job group for the job. You will need this value to check on your job status.

For more information, see *https://api.trifacta.com/ee/es.t/index.html#operation/runJobGroup*

Checkpoint: You have queued your job for execution.

Step - Monitoring Your Job

You can monitor the status of your job through the following endpoint:

Endpoint	`<protocol>://<platform_base_url>/v4/jobGroups/<id>/`
Authentication	Required
Method	`GET`
Request Body	None.
Response Code	`200 - Ok`
Response Body	{ "id": 961247, "name": null, "description": null, "ranfrom": "ui", "ranfor": "recipe", "status": "Complete", "profilingEnabled": true, "runParameterReferenceDate": "2019-08-20T17:46:27.000Z", "createdAt": "2019-08-20T17:46:28.000Z", "updatedAt": "2019-08-20T17:53:17.000Z", "workspace": { "id": 22 }, "creator": { "id": 38 }, "updater": { "id": 38 }, "snapshot": { "id": 774476 }, "wrangledDataset": { "id": 28629 }, "flowRun": null }

When the job has successfully completed, the returned status message includes the following:

```
"status": "Complete",
```

For more information, see *https://api.trifacta.com/ee/es.t/index.html#operation/getJobGroup*

Tip: You have executed the job. Results have been delivered to the designated output locations.

Step - Re-run Job

In the future, you can re-run the job using the same, simple request:

Endpoint	`<protocol>://<platform_base_url>/v4/jobGroups`
Authentication	Required

Method	POST
Request Body	<pre>{ "wrangledDataset": { "id": 28629 } }</pre>

The job is re-run as it was previously specified.

For more information, see *https://api.trifacta.com/ee/es.t/index.html#operation/createJobGroup*

Step - Run Job with Overrides - Files

As needed, you can specify runtime overrides for any of the settings related to the job definition or its outputs. For file-based jobs, these overrides include:

- Data sources
- Execution environment
- profiling
- Output file, format, and other settings

Input file overrides

You can override the file-based data sources your job run. In the following example, two parameterized datasets are overridden with new files.

NOTE: Overrides for data sources apply only to file-based sources. File-based sources that are converted during ingestion, such as Microsoft Excel files, cannot be swapped in this manner.

Endpoint	`<protocol>://<platform_base_url>/v4/jobGroups`
Authentication	Required
Method	POST
Request Body	

The job specified for recipe `28629` is re-run using the new data sources.

Notes:

- You can use this API method to overwrite the bucket name for your source, which is not possible through standard parameterization.
  - The parameterized list of files can be from different folders, too.
- File type and size information is not displayed in the Job Details page for these overridden jobs.
- No validation is performed on the existence of these files prior to execution. If the files do not exist, the job fails.

For more information, see *https://api.trifacta.com/ee/es.t/index.html#operation/createJobGroup*

Output file overrides

NOTE: Override values applied to a job are not validated. Invalid overrides may cause your job to fail.

- See *API Workflow - Manage Outputs*.
- See *https://api.trifacta.com/ee/es.t/index.html#operation/getWriteSetting*

1. Acquire the internal identifier for the recipe for which you wish to execute a job. In the previous example, this identifier was `28629`.
2. Construct a request using the following:

Endpoint	`<protocol>://<platform_base_url>/v4/jobGroups`
Authentication	Required
Method	`POST`

Request Body:

```
{
  "wrangledDataset": {
    "id": 28629
  },
  "overrides": {
    "profiler": true,
    "execution": "spark",
    "writesettings": [
      {
        "path": "<new_path_to_output>",
        "format": "csv",
        "header": true,
        "asSingleFile": true
      }
    ]
  },
  "ranfrom": null
}
```

3. In the above example, the job has been launched with the following overrides:
   a. Job will be executed on the Spark cluster. Other supported values depend on your deployment:

Value for `overrides.execution`	Description
`photon`	Running environment on Trifacta node
`spark`	Spark on integrated cluster, with the following exceptions.
`databricksSpark`	Spark on Azure Databricks
	Spark on AWS EMR

`emrSpark`	

b. Job will be executed with profiling enabled.
c. Output is written to a new file path.
d. Output format is CSV to the designated path.
e. Output has a header and is generated as a single file.

4. A response code of `201 - Created` is returned. The response body should look like the following:

```
{

    "sessionId": "79276c31-c58c-4e79-ae5e-fed1a25ebca1",
    "reason": "JobStarted",
    "jobGraph": {
        "vertices": [
            21,
            22
        ],
        "edges": [
            {
                "source": 21,
                "target": 22
            }
        ]
    },
    "id": 962221,
    "jobs": {
        "data": [
            {
                "id": 21
            },
            {
                "id": 22
            }
        ]
    }
}
```

5. Retain the `id` value, which is the job identifier, for monitoring.

Step - Run Job with Overrides - Tables

You can also pass job definition overrides for table-based outputs. For table outputs, overrides include:

- Path to database to which to write (must have write access)
- Connection to write to the target.

  Tip: This identifier is for the connection used to write to the target system. This connection must already exist. For more information on how to retrieve the identifier for a connection, see

  https://api.trifacta.com/ee/es.t/index.html#operation/listConnections

- Name of output table
- Target table type

  Tip: You can acquire the target type from the `vendor` value in the connection response. For more information, see

  https://api.trifacta.com/ee/es.t/index.html#operation/listConnections

- action:

Key value	Description

`create`	Create a new table with each publication.
`createAndLoad`	Append your data to the table.
`truncateAndLoad`	Truncate the table and load it with your data.
`dropAndLoad`	Drop the table and write the new table in its place.

- Identifier of connection to use to write data.

- See *API Workflow - Manage Outputs*.
- See *https://api.trifacta.com/ee/es.t/index.html#operation/getPublication*

1. Acquire the internal identifier for the recipe for which you wish to execute a job. In the previous example, this identifier was `28629`.
2. Construct a request using the following:

Endpoint	`<protocol>://<platform_base_url>/v4/jobGroups`
Authentication	Required
Method	`POST`

Request Body:

```
```

3. In the above example, the job has been launched with the following overrides:

   NOTE: When overrides are applied to publishing, any publications that are already attached to the recipe are ignored.
   a. Output path is to the `prod_db` database, using table name is `Table_CaseFctn2`.
   b. Output action is "create and load." See above for definitions.
   c. Target table type is a PostgreSQL table.
4. A response code of `201 - Created` is returned. The response body should look like the following:

```
{


    "sessionId": "79276c31-c58c-4e79-ae5e-fed1a25ebca1",
    "reason": "JobStarted",
    "jobGraph": {
        "vertices": [
            21,
            22
        ],
        "edges": [
            {
                "source": 21,
```

```
                "target": 22
            }
        ]
    },
    "id": 962222,
    "jobs": {
        "data": [
            {
                "id": 21
            },
            {
                "id": 22
            }
        ]
    }
}
```

5. Retain the `id` value, which is the job identifier, for monitoring.

Step - Run Job with Overrides - Webhooks

When you execute a job, you can pass in a set of parameters as overrides to generate a webhook message to a third-party application, based on the success or failure of the job.

For more information on webhooks, see *Create Flow Webhook Task*.

1. Acquire the internal identifier for the recipe for which you wish to execute a job. In the previous example, this identifier was `28629`.
2. Construct a request using the following:

Endpoint	`<protocol>://<platform_base_url>/v4/jobGroups`
Authentication	Required
Method	POST

Request Body:

```
{
  "wrangledDataset": {
    "id": 28629
  },
  "overrides": {
    "webhooks": [{
      "name": "webhook override",
      "url": "http://example.com",
      "method": "post",
      "triggerEvent": "onJobFailure",
      "body": {
        "text": "override"
       },
      "headers": {
        "testHeader": "val1"
       },
      "sslVerification": true,
      "secretKey": "123",
  }]
 }
}
```

3. In the above example, the job has been launched with the following overrides:

Override setting	Description
name	Name of the webhook.
url	URL to which to send the webhook message.
method	The HTTP method to use. Supported values: `POST`, `PUT`, `PATCH`, `GET`, or DELETE. Body is ignored for `GET` and `DELETE` methods.
triggerEvent	Supported values: `onJobFailure` - send webhook message if job fails `onJobSuccess` - send webhook message if job completes successfully `onJobDone` - send webhook message when job fails or finishes successfully
body	(optional) The value of the `text` field is the message that is sent. **NOTE:** Some special token values are supported. See *Create Flow Webhook Task*.
header	(optional) Key-value pairs of headers to include in the HTTP request.
sslVerification	(optional) Set to `true` if SSL verification should be completed. If not specified, the value is `true`.
secretKey	(optional) If enabled, this value should be set to the secret key to use.

4. A response code of `201 - Created` is returned. The response body should look like the following:


```
{


    "sessionId": "79276c31-c58c-4e79-ae5e-fed1a25ebca1",
    "reason": "JobStarted",
    "jobGraph": {
        "vertices": [
            21,
            22
        ],
        "edges": [
            {
                "source": 21,
                "target": 22
            }
        ]
    },
    "id": 962222,
    "jobs": {
        "data": [
            {
                "id": 21
            },
            {
                "id": 22
            }
        ]
    }
}
```


5. Retain the `id` value, which is the job identifier, for monitoring.

Step - Run Job with Parameter Overrides

You can pass overrides of the default parameter values as part of the job definition. You can use the following mechanism to pass in parameter overrides of the following types:

- Datasets with parameters (variable type)

- Output object parameters
- Flow parameters

The syntax is the same for each type.

1. Acquire the internal identifier for the recipe for which you wish to execute a job. In the previous example, this identifier was `28629`.
2. Construct a request using the following:

Endpoint	`<protocol>://<platform_base_url>/v4/jobGroups`
Authentication	Required
Method	`POST`

Request Body:

```
```

3. In the above example, the specified job has been launched for recipe `28629`. The run parameter `varRegion` has been set to `02` for this specific job. Depending on how it's defined in the flow, this parameter could influence change either of the following:
   a. The source for the imported dataset.
   b. The path for the generated output.
   c. A flow parameter reference in the recipe
   d. For more information, see *Overview of Parameterization*.
4. A response code of `201 - Created` is returned. The response body should look like the following:

```
{
    "sessionId": "79276c31-c58c-4e79-ae5e-fed1a25ebca1",
    "reason": "JobStarted",
    "jobGraph": {
        "vertices": [
            21,
            22
        ],
        "edges": [
            {
                "source": 21,
                "target": 22
            }
        ]
    },
    "id": 962223,
    "jobs": {
        "data": [
            {
                "id": 21
```

```
            },
            {
                "id": 22
            }
        ]
    }
}
```

5. Retain the `id` value, which is the job identifier, for monitoring.

Step - Spark Job Overrides

When it is enabled, you can submit overrides to a specific set of Spark properties for your job.

This feature and the Spark properties to override must be enabled. For more information on enabling this feature, see *Enable Spark Job Overrides*.

The following example, shows how to run a job for a specified recipe with Spark property overrides applied to it. This example assumes that the job has already been configured to be executed on Spark (`"execution": "spark"`):

Endpoint	`<protocol>://<platform_base_url>/v4/jobGroups`
Authentication	Required
Method	`POST`

Request Body:

API Workflow - Run Job on Dataset with Parameters

Contents:

- *Overview*
  - *Basic Workflow*
  - *Example Datasets*
- *Step - Create Containing Flow*
- *Step - Create Datasets with Parameters*
  - *Example 1 - Dataset with Datetime parameter*
  - *Example 2 - Dataset with Variable*
  - *Example 3 - Dataset with pattern parameter*
- *Step - Wrangle Data*
- *Step - Run Job*
  - *Example 1 - Dataset with Datetime parameter*
  - *Example 2 - Dataset with Variable*
  - *Example 3 - Dataset with pattern parameter*
- *Step - Monitoring Your Job*
- *Step - Re-run Job*

Overview

This example workflow describes how to run jobs on datasets with parameters through Trifacta® Wrangler Enterprise. A **dataset with parameters** is a dataset in which some part of the path to the data objects has been parameterized. Since one or more of the parts of the path can vary, you can build a dataset with parameters to capture data that spans multiple files. For example, datasets with parameters can be used to parameterize serialized data by region or data or other variable.

For more information on datasets with parameters, see *Overview of Parameterization*.

Basic Workflow

The basic method by which you build and run a job for a dataset with parameters is very similar to the non-parameterized dataset method with a few notable exceptions. The steps in this workflow follow the same steps for the standard workflow. Where the steps overlap links have been provided to the non-parameterized workflow. For more information, see *API Workflow - Develop a Flow*.

Example Datasets

This example covers three different datasets, each of which features a different type of dataset with parameters.

Example Number	Parameter Type	Description
1	Datetime parameter	In this example, a directory is used to store daily orders transactions. This dataset must be defined with a Datetime parameter to capture the preceding 7 days of data. Jobs can be configured to process all of this data as it appears in the directory.
2	Variable	This dataset segments data into four timezones across the US. These timezones are defined using the following text values in the path: `pacific`, `mountain`, `central`, and `eastern`. In this case, you can create a parameter called `region`, which can be overridden at runtime to be set to one of these four values during job execution.
3	Pattern parameter	This example is a directory containing point-of-sale transactions captured into individual files for each region. Since each region is defined by a numeric value (`01`, `02`, `03`), the dataset can be defined using a pattern parameter.

Step - Create Containing Flow

You must create the flow to host your dataset with parameters.

In the response, you must capture and retain the flow Identifier. For more information, see *API Workflow - Develop a Flow*.

Step - Create Datasets with Parameters

NOTE: When you import a dataset with parameters, only the first matching dataset is used for the initial file. If you want to see data from other matching files, you must collect a new sample within the Transformer page.

Example 1 - Dataset with Datetime parameter

Suppose your files are stored in the following paths:

```
MyFiles/1/Datetime/2018-04-06-orders.csv
MyFiles/1/Datetime/2018-04-05-orders.csv
MyFiles/1/Datetime/2018-04-04-orders.csv
MyFiles/1/Datetime/2018-04-03-orders.csv
MyFiles/1/Datetime/2018-04-02-orders.csv
MyFiles/1/Datetime/2018-04-01-orders.csv
MyFiles/1/Datetime/2018-03-31-orders.csv
```

When you navigate to the directory through the application, you mouse over one of these files and select **Parameterize**.

In the window, select the date value (e.g. `YYYY-MM-DD`) and then click the Datetime icon.

Datetime Parameter:

- Format: `YYYY-MM-DD`
- Date Range: Date is last 7 days.
- Click **Save**.

The Datetime parameter should match with all files in the directory. Import this dataset and wrangle it.

After you wrangle the dataset, return to its flow view and select the recipe. You should be able to extract the flowId and recipeId values from the URL.

For purposes of this example, here are some key values:

- flowId: 35
- recipeId: 127

Example 2 - Dataset with Variable

Suppose your files are stored in the following paths:

```
MyFiles/1/variable/census-eastern.csv
MyFiles/1/variable/census-central.csv
MyFiles/1/variable/census-mountain.csv
MyFiles/1/variable/census-pacific.csv
```

When you navigate to the directory through the application, you mouse over one of these files and select **Parameterize**.

In the window, select the region value, which could be one of the following depending on the file: `eastern`, `central`, `mountain`, or `pacific`. Click the Variable icon.

Variable Parameter:

- Name: `region`
- Default Value:Set this default to `pacific`.
- Click **Save**.

In this case, the variable only matches one value in the directory. However, when you apply runtime overrides to the `region` variable, you can set it to any value.

Import this dataset and wrangle it.

After you wrangle the dataset, return to its flow view and select the recipe. You should be able to extract the flowId and recipeId values from the URL.

For purposes of this example, here are some key values:

- flowId: 33
- recipeId: 123

Example 3 - Dataset with pattern parameter

Suppose your files are stored in the following paths:

```
MyFiles/1/pattern/POS-r01.csv
MyFiles/1/pattern/POS-r02.csv
MyFiles/1/pattern/POS-r03.csv
```

When you navigate to the directory through the application, you mouse over one of these files and select **Paramet erize**.

In the window, select the two numeric digits (e.g. `02`). Click the Pattern icon.

Pattern Parameter:

- Type: Trifacta pattern
- Matching regular expression: `{digit}{2}`
- Click **Save**.

In this case, the Trifacta pattern should match any sequence of two digits in a row. In the above example, this expression matches: `01`, `02`, and `03`, all of the files in the directory.

Import this dataset and wrangle it.

After you wrangle the dataset, return to its flow view and select the recipe. You should be able to extract the flowId and recipeId values from the URL.

For purposes of this example, here are some key values:

- flowId: 32
- recipeId: 121

Checkpoint: You have created flows for each type of dataset with parameters.

Step - Wrangle Data

After you have created your dataset with parameter, you can wrangle it through the application. For more information, see *Transformer Page*.

Step - Run Job

Below, you can review the API calls to run a job for each type of dataset with parameters, including relevant information about overrides.

Example 1 - Dataset with Datetime parameter
NOTE: You cannot apply overrides to these types of datasets with parameters. The following request contains overrides for write settings but no overrides for parameters.

1.

Endpoint	`http://www.example.com:3005/v4/jobGroups`
Authentication	Required
Method	POST
Request Body	

```
{
  "wrangledDataset": {
    "id": 127
  },
  "overrides": {
    "execution": "photon",
    "profiler": true,
    "writesettings": [
      {
        "path": "MyFiles/queryResults/joe@example.com/2018-04-03-orders.csv",
        "action": "create",
        "format": "csv",
        "compression": "none",
        "header": false,
        "asSingleFile": false
      }
    ]
  },
  "runParameters": {}
}
```

2. In the above example, the job has been launched for recipe `127` to execute on the Trifacta Photon running environment with profiling enabled.
    a. Output format is CSV to the designated path. For more information on these properties, see *https://api.trifacta.com/ee/es.t/index.html#operation/runJobGroup*
    b. Output is written as a new file with no overwriting of previous files.
3. A response code of `201 - Created` is returned. The response body should look like the following:

```
{
    "sessionId": "79276c31-c58c-4e79-ae5e-fed1a25ebca1",
    "reason": "JobStarted",
    "jobGraph": {
        "vertices": [
            21,
            22
        ],
        "edges": [
            {
                "source": 21,
                "target": 22
            }
        ]
    },
    "id": 29,
    "jobs": {
        "data": [
            {
```

```
                "id": 21
            },
            {
                "id": 22
            }
        ]
    }
}
```

4. Retain the `jobgroupId=29` value for monitoring.

Example 2 - Dataset with Variable

In the following example, the `region` variable has been overwritten with the value `central` to execute the job on `orders-central.csv`:

1.

Endpoint	`http://www.example.com:3005/v4/jobGroups`
Authentication	Required
Method	POST
Request Body	

```
{
  "wrangledDataset": {
    "id": 123
  },
  "overrides": {
    "execution": "photon",
    "profiler": true,
    "writesettings": [
      {
        "path": "MyFiles/queryResults/joe@example.com/region-eastern.csv",
        "action": "create",
        "format": "csv",
        "compression": "none",
        "header": false,
        "asSingleFile": false
      }
    ]
  }
  "runParameters": {
    "overrides": {
      "data": [{
        "key": "region",
        "value": "central"
      }
    ]}
  }
}
```

2. In the above example, the job has been launched for recipe `123` to execute on the Trifacta Photon running environment with profiling enabled.
    a. Output format is CSV to the designated path. For more information on these properties, see *https://api.trifacta.com/ee/es.t/index.html#operation/runJobGroup*
    b. Output is written as a new file with no overwriting of previous files.
3. A response code of `201 - Created` is returned. The response body should look like the following:

```
{
    "sessionId": "79276c31-c58c-4e79-ae5e-fed1a25ebca1",
    "reason": "JobStarted",
    "jobGraph": {
        "vertices": [
            21,
            22
```

```
        ],
        "edges": [
            {
                "source": 21,
                "target": 22
            }
        ]
    },
    "id": 27,
    "jobs": {
        "data": [
            {
                "id": 21
            },
            {
                "id": 22
            }
        ]
    }
}
```

4. Retain the `jobgroupId=27` value for monitoring.

Example 3 - Dataset with pattern parameter

NOTE: You cannot apply overrides to these types of datasets with parameters. The following request contains overrides for write settings but no overrides for parameters.

1.

Endpoint	`http://www.example.com:3005/v4/jobGroups`
Authentication	Required
Method	POST
Request Body	<pre>{ "wrangledDataset": { "id": 121 }, "overrides": { "execution": "photon", "profiler": false, "writesettings": [{ "path": "hdfs://hadoop:50070/trifacta/queryResults/admin@example.com/POS-r02.txt", "action": "create", "format": "csv", "compression": "none", "header": false, "asSingleFile": false }] }, "runParameters": {} }</pre>

2. In the above example, the job has been launched for recipe `121` to execute on the Trifacta Photon running environment with profiling enabled.
    a. Output format is CSV to the designated path. For more information on these properties, see *https://api.trifacta.com/ee/es.t/index.html#operation/runJobGroup*
    b. Output is written as a new file with no overwriting of previous files.
3. A response code of `201 - Created` is returned. The response body should look like the following:

```
{
    "sessionId": "79276c31-c58c-4e79-ae5e-fed1a25ebca1",
    "reason": "JobStarted",
    "jobGraph": {
        "vertices": [
            21,
            22
        ],
        "edges": [
            {
                "source": 21,
                "target": 22
            }
        ]
    },
    "id": 28,
    "jobs": {
        "data": [
            {
                "id": 21
            },
            {
                "id": 22
            }
        ]
    }
}
```


4. Retain the `jobgroupId=28` value for monitoring.

Step - Monitoring Your Job

After the job has been created and you have captured the jobGroup Id, you can use it to monitor the status of your job. For more information, see *https://api.trifacta.com/ee/es.t/index.html#operation/getJobGroup*

Step - Re-run Job

If you need to re-run the job as specified, you can use the wrangledDataset identifier to re-run the most recent job.

Tip: When you re-run a job, you can change any variable values as part of the request.

Example request:

Endpoint	`http://www.example.com:3005/v4/jobGroups`
Authentication	Required
Method	POST
Request Body	

For more information, see *API Workflow - Develop a Flow*.

API Workflow - Run Deployment

Contents:

- *Pre-requisites*
- *Step - Run Deployment*
- *Step - Monitoring Your Deployment Job*
- *Step - Run Deployment with Overrides*
  - *Acquire the active outputs for the deployment*
  - *Acquire output object information*
  - *Run deployment with overrides*
- *Step - Run Deployment with Spark Overrides*

This section describes how to run a deployment using the APIs available in Trifacta® Wrangler Enterprise.

- A **deployment** is a packaging mechanism for versioning your production-level flows.
- Deployments are created and managed through a separate interface.
- For more information, see *Overview of Deployment Manager*.

A note about API URLs:

In the listed examples, URLs are referenced in the following manner:

```
<protocol>://<platform_base_url>/
```

In your product, these map references map to the following:

```
<http or https>://<hostname>:<port_number>/
```

For more information, see *API Reference*.

Pre-requisites

Before you begin, you should verify the following:

1. **Get authentication credentials**. As part of each request, you must pass in authentication credentials to the Trifacta platform.

   Tip: The recommended method is to use an access token, which can be generated from the Trifacta application. For more information, see *Access Tokens Page*.

   See *https://api.trifacta.com/ee/es.t/index.html#operation/Authentication*
2. **Develop your flow.** Before you deploy a flow through the Deployment Manager, you should build and test your flow in a development environment. See *API Workflow - Develop a Flow*.
3. **Deploy your flow.** After you have developed a flow, you can deploy it. See *API Workflow - Deploy a Flow*.
4. **Acquire deployment identifier.** In Deployment Manager, acquire the numeric value for the deployment. See *Deployment Manager Page*.

Step - Run Deployment

Through the APIs, you can run a deployment. When a deployment is executed:

- All of the recipes that are included in the active release of the deployment are executed.

- All default values for outputs are applied.

In the following example, the deployment identifier is `2`.

Endpoint	`<protocol>://<platform_base_url>/v4/deployments/2/run`
Authentication	Required
Method	POST
Request Body	None.
Response Code	`201 - Created`
Response Body	

If the `201` response code is returned, then the deployment job has been queued for execution.

Tip: Retain the `id` value in the response. In the above, `961247` is the internal identifier for the job group for the job. You will need this value to check on your job status.

For more information, see *API Workflow - Deploy a Flow*.

Checkpoint: You have queued your deployment job for execution.

Step - Monitoring Your Deployment Job

You can monitor the status of your deploy job through the following endpoint using the `id` value that was returned in the previous step:

```
<protocol>://<platform_base_url>/v4/jobGroups/<id>/status
```

For more information, see *API Workflow - Run Job*.

Step - Run Deployment with Overrides

When you run a deployment, you cannot apply overrides to the request. However, you can complete the following steps to apply overrides when you execute the jobs within the active release. In this workflow, you run jobs on the specific recipes of the active deployment, applying overrides as needed.

Suppose you are running the jobs for deployment id `2`, and you want to apply some overrides.

NOTE: A deployment can trigger several different jobs within a single flow. In the following example, it is assumed that there is only one output.

Acquire the active outputs for the deployment

The first step is to acquire all of the active outputs for the deployment.

Endpoint	`<protocol>://<platform_base_url>/v4/deployments/2/activeoutputs`
Authentication	Required
Method	`GET`
Request Body	None.
Response Code	`200 - OK`
Response Body	

in the above response:

- The `flowNodeId` value corresponds to the recipe (`wrangledDataset`) identifier that you wish to modify.
- The `outputObjectId` value corresponds to the output object that is produced by default from the recipe.

Acquire output object information

The next step is to review the output object to determine what needs to be overridden. Since you are overriding a file-based publication, you can query directly for the `writeSettings` objects associated with the output object:

Endpoint	`<protocol>://<platform_base_url>/v4/outputObjects/6/writeSettings`
Authentication	Required
Method	`GET`
Request Body	None.
Response Code	`200 - OK`
Response Body	


```
{
    "data": [
        {
            "delim": ",",
            "id": 17,
            "path": "hdfs://hadoop:50070/example/joe@example.com/USDA_Farmers_Market_2014.
csv",
            "action": "create",
            "format": "csv",
            "compression": "none",
            "header": true,
            "asSingleFile": true,
            "prefix": null,
            "suffix": "_increment",
            "hasQuotes": true,
            "createdAt": "2019-11-05T18:26:31.972Z",
            "updatedAt": "2019-11-05T18:30:56.756Z",
            "creator": {
```

```
                "id": 2
            },
            "updater": {
                "id": 2
            },
            "outputObject": {
                "id": 6
            }
        },
        {
            "delim": ",",
            "id": 16,
            "path": "hdfs://hadoop:50070/example/joe@example.com/USDA_Farmers_Market_2014.
json",
            "action": "create",
            "format": "json",
            "compression": "none",
            "header": false,
            "asSingleFile": false,
            "prefix": null,
            "suffix": "_increment",
            "hasQuotes": false,
            "createdAt": "2019-11-05T18:26:44.983Z",
            "updatedAt": "2019-11-05T18:30:56.743Z",
            "creator": {
                "id": 2
            },
            "updater": {
                "id": 2
            },
            "outputObject": {
                "id": 6
            }
        }
    ]
}
```


Run deployment with overrides

Now that you have access to the outputs generated by the deployment, you decide to override the following for each file:

- Filename:
  - Remove the year information at the end of the filename
  - Store in a separate folder called `final`
- Wipe the table and reload it each time (`action=overwrite`)
- Disable profiling

From the activeOutputs endpoint, you retrieved the flowNodeId (`27`).

Endpoint	`<protocol>://<platform_base_url>/v4/jobGroups/`
Authentication	Required
Method	POST

```
csv",
        "format": "csv",
        "action": "overwrite"
      },
      {
        "path": "hdfs://hadoop:50070/example/joe@example.com/final/USDA_Farmers_Market.
json",
        "format": "json",
        "action": "overwrite"
      }
    ]
  },
  "ranfrom": null
}
```


Response Code

201 - Created

Response Body


```
{
    "sessionId": "b29467c3-fc59-499e-aed6-d797746a86eb",
    "reason": "JobStarted",
    "jobGraph": {
        "vertices": [
            10,
            11,
            12
        ],
        "edges": [
            {
                "source": 10,
                "target": 11
            },
            {
                "source": 10,
                "target": 12
            }
        ]
    },
    "id": 4,
    "jobs": {
        "data": [
            {
                "id": 10
            },
            {
                "id": 11
            },
            {
                "id": 12
            }
        ]
    }
}
```


Checkpoint: Your job with overrides has been queued for execution.

You can use the job identifier (4) to monitor the job status.

Step - Run Deployment with Spark Overrides

When you run a deployment, you can specify override values to the Spark properties that have been made available for overrides.

NOTE: These overrides only apply if you are running the job on Spark and if the feature has been enabled in your deployment.

- This feature and the properties to override must be enabled. See *Enable Spark Job Overrides*.
- All of the recipes that are included in the active release of the deployment are executed.

All other default values are applied. In the following example, the deployment identifier is `2`.

Endpoint	`<protocol>://<platform_base_url>/v4/deployments/2/run`
Authentication	Required
Method	`POST`
Request Body	
Response Code	`201 - Created`
Response Body	

If the `201` response code is returned, then the deployment job has been queued for execution.

Checkpoint: You have queued your deployment job for execution.

API Workflow - Publish Results

Contents:

- *Overview*
  - *Basic Workflow*
- *Step - Create Connections*
  - *Redshift connection*
  - *Hive connection*
  - *Tableau Server connection*
  - *SQL DW connection*
- *Step - Run Job*
- *Step - Publish Results to Hive*
- *Step - Publish Results to Tableau Server*
- *Step - Publish Results to SQL DW*
- *Step - Publish Results to Redshift*
- *Step - Publish Results with Overrides*

Overview

After you have run a job to generate results, you can publish those results to different targets as needed. This section describes how to automate those publishing steps through the APIs.

NOTE: This workflow applies to re-publishing job results after you have already generated them.
NOTE: After you have generated results and written them to one target, you cannot publish to the same target. You must configure the outputs to specify a different format and location and then run a new job.

In the application, you can publish after generating results. See *Publishing Dialog*.

Basic Workflow

1. Create connections to each target to which you wish to publish. Connections must support write operations.
2. Specify a job whose output meets the requirements for the target.
3. Run the job.
4. When the job completes, publish the results to the target(s).

Step - Create Connections

For each target, you must have access to create a connection to it. After a connection is created, it can be reused, so you may find it easier to create them through the application.

- Some connections can be created via API. For more information, see *https://api.trifacta.com/ee/es.t/index.html#operation/createConnection*
- Other connections must be created through the application. Links to instructions are provided below.

NOTE: Connections created through the application must be created through the Connections page, which is used for creating read/write connections. Do not create these connections through the Import Data page. See *Connections Page*.

Redshift connection

- **Required Output Format:** Avro

- **Example Id:** 2
- **Create via API:** N
- **Doc Link:** *Create Redshift Connections*

Other Requirements:

- Requires S3 set as the base storage layer. See *Set Base Storage Layer*.

Hive connection

- **Required Output Format:** Avro
- **Example Id:** 1
- **Create via API:** Y
- **Doc Link:** *Create Hive Connections*
- **Other Requirements:**
  - Requires integration with a Hadoop cluster.

Tableau Server connection

- **Required Output Format:** HYPER or TDE

  NOTE: TDE will be deprecated in a future release. Please switch to using Hyper format.
- **Example Id:** 3
- **Create via API:** Y
- **Doc Link:** *Create Tableau Server Connections*
- **Other Requirements:**
  - None.

SQL DW connection

- **Required Output Format:** Parquet
- **Example Id:** 4
- **Create via API:** N
- **Doc Link:** *Create SQL DW Connections*
- **Other Requirements:**
  - Available only on Azure deployments. See *Configure for Azure*.

Step - Run Job

Before you publish results to a different datastore, you must generate results and store them in HDFS.

NOTE: To produce some output formats, you must run the job on the Spark running environment.

In the examples below, the following example data is assumed:

Identifier	Value
jobId	2
flowId	3

For more information on running a job, see *https://api.trifacta.com/ee/es.t/index.html#operation/runJobGroup*
For more information on the publishing endpoint, see
https://api.trifacta.com/ee/es.t/index.html#operation/publishJobGroup

Step - Publish Results to Hive

The following uses the Avro results from the specified job (jobId = 2) to publish the results to the `test_table` table in the `default` Hive schema through connectionId=1.

NOTE: To publish to Hive, the targeted database is predefined in the connection object. For the `path` value in the request body, you must specify the schema in this database to use. Schema information is not available through API. To explore the available schemas, click the Hive icon in the Import Data page. The schemas are the first level of listed objects. For more information, see *Import Data Page*.

Request:

Endpoint	`http://www.wrangle-dev.example.com:3005/v4/jobGroups/2/publish`
Authentication	Required
Method	PUT
Request Body	

Response:

Status Code	`200 - OK`
Response Body	{ "jobgroupId":2, "reason":"JobStarted", "sessionId":"24862060-4fcd-11e8-8622-fda0fbf6f550" }

Step - Publish Results to Tableau Server

The following uses the TDE results from the specified job (jobId = 2) to publish the results to the `test_table3` table in the `default` Tableau Server database through connectionId=3.

Request:

Endpoint	`http://www.wrangle-dev.example.com:3005/v4/jobGroups/2/publish`
Authentication	Required
Method	PUT
Request Body	{ "connection": { "id": 3 }, "path": ["default"], "table": "test_table3",

```
  "action": "createAndLoad",
  "inputFormat": "hyper"
}
```

Response:

Status Code	200 - OK
Response Body	{ "jobgroupId":2, "reason":"JobStarted", "sessionId":"24862060-4fcd-11e8-8622-fda0fbf6f552" }

Step - Publish Results to SQL DW

The following uses the Parquet results from the specified job (jobId = 2) to publish the results to the `test_table4` table in the `dbo` SQL DW database through connectionId=4.

Request:

Endpoint	*http://www.wrangle-dev.example.com:3005/v4/jobGroups/2/publish*
Authentication	Required
Method	PUT
Request Body	

Response:

Status Code	200 - OK
Response Body	{ "jobgroupId": 2, "jobIds": 22, "reason": "JobStarted", "sessionId": "855f83a0-dc94-11e8-bd1a-f998d808020d" }

Step - Publish Results to Redshift

The following uses the Avro results from the specified job (jobId = 2) to publish the results to the `test_table2` table in the `public` Redshift schema through connectionId=2.

NOTE: To publish to Redshift, the targeted database is predefined in the connection object. For the `path` value in the request body, you must specify the schema in this database to use. Schema information is not available through API. To explore the available schemas, click the Redshift icon in the Import Data page. The schemas are the first level of listed objects. For more information, see *Import Data Page*.

Request:

Endpoint	`http://www.wrangle-dev.example.com:3005/v4/jobGroups/2/publish`
Authentication	Required
Method	PUT
Request Body	

Response:

Status Code	`200 - OK`
Response Body	`{ "jobgroupId":2, "reason":"JobStarted", "sessionId":"fae64760-4fc4-11e8-8cba-0987061e4e16" }`

Step - Publish Results with Overrides

When you are publishing results to a relational source, you can apply overrides to the job to redirect the output or change the action applied to the target table. For more information, see *API Workflow - Run Job*.

API Workflow - Swap Datasets

Contents:

- *Overview*
  - *Example Datasets*
  - *Assumptions*
- *Step - Import Dataset*
- *Step - Swap Dataset from Recipe*
- *Step - Rerun Job*
- *Step - Monitor Your Job*
- *Step - Schedule Your Job*

Overview

After you have created a flow, imported a dataset, and created a recipe for that dataset, you may need to swap in a different dataset and run the recipe against that one. This workflow steps through that process via the APIs.

NOTE: If you are processing multiple parallel datasources in a single job, you should create a dataset with parameters and then run the job. For more information, see *API Workflow - Run Job on Dataset with Parameters*.

This workflow utilizes the following methods:

1. **Creating an imported dataset.** After the new file has been added to the backend datastore, you can import into Trifacta® Wrangler Enterprise as an imported dataset.
2. **Swap dataset.** Using the ID of the imported dataset you created, you can now assign the dataset to the recipe in your flow.
3. **Run a job.** Run the job against the dataset.
4. **Monitor progress.** Monitor the progress of the job until it is complete.

Example Datasets

In this example, you are wrangling data from orders placed in different regions on a quarterly basis. When a new file drops, you want to be able to swap out the current dataset that is assigned to the recipe and swap in the new one. Then, run the job.

Example Files:

The following files are stored on your HDFS deployment:

Path and Filename	Description
`hdfs:///user/orders/MyCo-orders-west-Q1.txt`	Orders from West region for Q1
`hdfs:///user/orders/MyCo-orders-west-Q2.txt`	Orders from West region for Q2
`hdfs:///user/orders/MyCo-orders-north-Q1.txt`	Orders from North region for Q1
`hdfs:///user/orders/MyCo-orders-north-Q2.txt`	Orders from North region for Q2
`hdfs:///user/orders/MyCo-orders-east-Q1.txt`	Orders from East region for Q1
`hdfs:///user/orders/MyCo-orders-east-Q1.txt`	Orders from East region for Q2

Assumptions

You have already created a flow, which contains the following imported dataset and recipe:

NOTE: When an imported dataset is created via API, it is always imported as an unstructured dataset. Any recipe that references this dataset should contain initial parsing steps required to structure the data.
Tip: Through the UI, you can import one of your datasets as unstructured. Create a recipe for this dataset and then edit it. In the Recipe panel, you should be able to see the structuring steps. Back in Flow View, you can chain your structural recipe off of this one. Dataset swapping should happen on the first recipe.

Object Type	Name	Id
flow	MyCo-Orders-Quarter	2
Imported Dataset	MyCo-orders-west-Q1.txt	8
Recipe (wrangledDataset)	n/a	9
Job	n/a	3

Base URL:

For purposes of this example, the base URL for the platform is the following:

```
http://www.example.com:3005
```

Step - Import Dataset

NOTE: You cannot add datasets to the flow through the `flows` endpoint. Moving pre-existing datasets into a flow is not supported in this release. Create or locate the flow first and then when you create the datasets, associate them with the flow at the time of creation.

- See *https://api.trifacta.com/ee/es.t/index.html#operation/createImportedDataset*
- See *https://api.trifacta.com/ee/es.t/index.html#operation/createWrangledDataset*

NOTE: When an imported dataset is created via API, it is always imported as an unstructured dataset. Any recipe that references this dataset should contain initial parsing steps required to structure the data.

The following steps describe how to create an imported dataset and assign it to the flow that has already been created (flowId=2).

Steps:

1. To create an imported dataset, you must acquire the following information about the source.
    a. path
    b. type
    c. name
    d. description
    e. bucket (if a file stored on S3)
2. In this example, the file you are importing is `MyCo-orders-west-Q2.txt`. Since the files are similar in nature and are stored in the same directory, you can acquire this information by gathering the information from the imported dataset that is already part of the flow. Execute the following:

Endpoint	*`http://www.example.com:3005/v4/importedDatasets`*
Authentication	Required

Method	POST
Request Body	<pre>{ "path": "hdfs:///user/orders/MyCo-orders-west-Q2.txt", "name": "MyCo-orders-west-Q2.txt", "description": "MyCo-orders-west-Q2" }</pre>

3. The response should be a `201 - Created` status code with something like the following:


```
{
    "id": 12,
    "size": "281032",
    "path": "hdfs:///user/orders/MyCo-orders-west-Q2.txt",
    "dynamicPath": null,
    "workspaceId": 1,
    "isSchematized": false,
    "isDynamic": false,
    "disableTypeInference": false,
    "createdAt": "2018-10-29T23:15:01.831Z",
    "updatedAt": "2018-10-29T23:15:01.889Z",
    "parsingRecipe": {
        "id": 11
    },
    "runParameters": [],
    "name": "MyCo-orders-west-Q2.txt.txt",
    "description": "MyCo-orders-west-Q2.txt",
    "creator": {
        "id": 1,
    },
    "updater": {
        "id": 1,
    },
    "connection": null,
}
```


4. You must retain the `id` value so you can reference it when you create the recipe.
5. See *https://api.trifacta.com/ee/es.t/index.html#operation/createImportedDataset*

Checkpoint: You have imported a dataset that is unstructured and is not associated with any flow.

Step - Swap Dataset from Recipe

The next step is to swap the primary input dataset for the recipe to point at the newly imported dataset. This step automatically adds the imported dataset to the flow and drops the previous imported dataset from the flow.

1. Use the following to swap the primary input dataset for the recipe:

Endpoint	*`http://www.example.com:3005/v4/wrangledDatasets/9/primaryInputDataset`*
Authentication	Required
Method	PUT
Request Body	<pre>{ "importedDataset": { "id": 12 } }</pre>

2. The response should be a `200 - OK` status code with something like the following:

```
{
    "id": 9,
    "wrangled": true,
    "createdAt": "2019-03-03T17:58:53.979Z",
    "updatedAt": "2019-03-03T18:01:11.310Z",
    "recipe": {
        "id": 9,
x        "name": "POS-r01",
x        "description": null,
        "active": true,
        "nextPortId": 1,
        "createdAt": "2019-03-03T17:58:53.965Z",
        "updatedAt": "2019-03-03T18:01:11.308Z",
        "currentEdit": {
            "id": 8
        },
        "redoLeafEdit": {
            "id": 7
        },
        "creator": {
            "id": 1
        },
        "updater": {
            "id": 1
        }
    },
    "referenceInfo": null,
    "activeSample": {
        "id": 7
    },
    "creator": {
        "id": 1
    },
    "updater": {
        "id": 1
    },
    "referencedFlowNode": null,
    "flow": {
        "id": 2
    }
}
```


3. The new imported dataset is now the primary input for the recipe, and the old imported dataset has been removed from the flow.

Step - Rerun Job

To execute a job on this recipe, you can simply re-run any job that was executed on the old imported dataset, since you reference the job by jobId and wrangledDataset (recipe) Id.

Endpoint	*http://www.example.com:3005*/v4/jobGroups
Authentication	Required
Method	POST
Request Body	`{ "wrangledDataset": { "id": 9 } }`

The job is re-run as it was previously specified.

If you need to modify any job parameters, you must create a new job definition.

Step - Monitor Your Job

After the job has been queued, you can track it to completion. See *API Workflow - Develop a Flow*.

Step - Schedule Your Job

When you are satisfied with how your flow is working, you can set up periodic schedules using a third-party tool to execute the job on a regular basis.

The tool must hit the above endpoints to swap in the new dataset and run the job.

API Workflow - Manage Outputs

Contents:

- *Overview*
  - *Basic Workflow*
  - *Variations*
- *Step - Get Recipe ID*
- *Step - Create outputObject*
- *Step - Run a Test Job*
- *Step - Create writeSettings Object*
- *Step - Get Connection ID for Publication*
- *Step - Create a Publication*
- *Step - Apply Overrides*
- *Step - Apply Spark Job Overrides*

Overview

Through the APIs, you can separately manage the outputs associated with an individual recipe. This workflow describes how to create output objects, which are associated with your recipe, and how to publish those outputs to different datastores in varying formats. You can continue to modify the output objects and their related write settings and publications independently of managing the wrangling process. Whenever you need new results, you can reference the wrangled dataset with which your outputs have been associated, and the job is executed and published in the appropriate manner to your targets.

Relevant terms:

Term	Description
`outputObjects`	An **outputObject** is a definition of one or more types of outputs and how they are generated. It must be associated with a recipe. **NOTE:** An outputObject must be created for a recipe before you can run a job on it. One and only one outputObject can be associated with a recipe.
`writeSettings`	A **writeSettings** object defines file-based outputs within an outputObject. Settings include path, format, compression, and delimiters.
`publications`	A **publications** object is used to specify a table-based output and is associated with an outputObject. Settings include the connection to use, path, table type, and write action to apply.

NOTE: If you need to make changes for purposes of a specific job run, you can add overrides to the request for the job. These overrides apply only for the current job. For more information, see *https://api.trifacta.com/ee/es.t/index.html#operation/runJobGroup*

Basic Workflow

Here's the basic workflow described in this section.

1. Get the internal identifier for the recipe for which you are building outputs.
2. Create the outputObject for the recipe.
3. Create a writeSettings object and associate it with the outputObject.
4. Run a test job, if desired.
5. For any publication, get the internal identifier for the connection to use.
6. Create a publication object and associate it with the outputObject.
7. Run your job.

Variations

If you are generating exclusively file-based or relational outputs, you can vary this workflow in the following ways:

For file-based outputs:

1. Get the internal identifier for the recipe for which you are building outputs.
2. Create the outputObject for the recipe.
3. Create a writeSettings object and associate it with the outputObject.
4. Run your job.

For relational outputs:

1. Get the internal identifier for the recipe for which you are building outputs.
2. Create the outputObject for the recipe.
3. For any publication, get the internal identifier for the connection to use.
4. Create a publication object and associate it with the outputObject.
5. Run your job.

Step - Get Recipe ID

To begin, you need the internal identifier for the recipe.

NOTE: In the APIs, a recipe is identified by its internal name, a **wrangled dataset**.

Request:

Endpoint	`http://www.wrangle-dev.example.com:3005/v4/wrangledDatasets`
Authentication	Required
Method	`GET`
Request Body	None.

Response:

Status Code	`200 - OK`
Response Body
```
{
    "data": [
        {
            "id": 11,
            "wrangled": true,
            "createdAt": "2018-11-12T23:06:36.473Z",
            "updatedAt": "2018-11-12T23:06:36.539Z",
            "recipe": {
                "id": 10
            },
            "name": "POS-r01",
            "description": null,
            "referenceInfo": null,
            "activeSample": {
                "id": 11
            },
            "creator": {
                "id": 1
            },
            "updater": {
                "id": 1
```
 |

```
            },
            "flow": {
                "id": 4
            }
        },
        {
            "id": 1,
            "wrangled": true,
            "createdAt": "2018-11-12T23:19:57.650Z",
            "updatedAt": "2018-11-12T23:20:47.297Z",
            "recipe": {
                "id": 19
            },
            "name": "member_info",
            "description": null,
            "referenceInfo": null,
            "activeSample": {
                "id": 20
            },
            "creator": {
                "id": 1
            },
            "updater": {
                "id": 1
            },
            "flow": {
                "id": 6
            }
        }
    ]
}
```


cURL example:

```
curl -X GET \
  http://www.wrangle-dev.example.com:3005/v4/wrangledDatasets \
  -H 'authorization: Basic <auth_token>' \
  -H 'cache-control: no-cache'
```

Relevant terms:

Term	Description
URL	URL and method to execute.
`authorization`	Authorization taken to pass to the platform. Basic authorization works. **NOTE:** This token must be passed with each request to the platform.
`cache-control`	Cache control setting.
`content-type`	HTTP content type to send. These applications use `application/json`.

Checkpoint: In the above, let's assume that the recipe identifier of interest is `wrangledDataset=11`. This means that the flow where it is hosted is `flow.id=4`. Retain this information for later.

For more information, see *https://api.trifacta.com/ee/es.t/index.html#operation/getWrangledDataset*

Step - Create outputObject

Create the outputObject and associate it with the recipe identifier. In the following request, the wrangledDataset identifier that you retrieved in the previous call is applied as the `flowNodeId` value.

The following example includes an embedded `writeSettings` object, which generates a CSV file output. You can remove this embedded object if desired, but you must create a `writeSettings` object before you can generate an output.

Request:

Endpoint	`http://www.wrangle-dev.example.com:3005/v4/outputObjects`
Authentication	Required
Method	POST
Request Body	

```
{
    "execution": "photon",
    "profiler": true,
    "isAdhoc": true,
    "writeSettings": {
        "data": [
            {
                "delim": ",",
                "path": "hdfs://hadoop:50070/trifacta/queryResults/admin@example.com/POS_01.
avro",
                "action": "create",
                "format": "avro",
                "compression": "none",
                "header": false,
                "asSingleFile": false,
                "prefix": null,
                "suffix": "_increment",
                "hasQuotes": false
            }
        ]
    },
    "flowNode": {
        "id": 11
    }
}
```

Response:

Status Code	`201 - Created`
Response Body	

```
{
    "id": 4,
    "execution": "photon",
    "profiler": true,
    "isAdhoc": true,
    "updatedAt": "2018-11-13T00:20:49.258Z",
    "createdAt": "2018-11-13T00:20:49.258Z",
    "creator": {
        "id": 1
    },
    "updater": {
        "id": 1
    },
    "flowNode": {
        "id": 11
    }
}
```

cURL example:

```
curl -X POST \
  http://www.wrangle-dev.example.com/v4/outputObjects \
  -H 'authorization: Basic <auth_token>' \
  -H 'cache-control: no-cache' \
  -H 'content-type: application/json' \
  -d '{
    "execution": "photon",
    "profiler": true,
    "isAdhoc": true,
    "writeSettings": {
        "data": [
            {
                "delim": ",",
                "path": "hdfs://hadoop:50070/trifacta/queryResults/admin@example.com/POS_01.avro",
                "action": "create",
                "format": "avro",
                "compression": "none",
                "header": false,
                "asSingleFile": false,
                "prefix": null,
                "suffix": "_increment",
                "hasQuotes": false
            }
        ]
    },
    "flowNode": {
        "id": 11
    }
}'
```

Relevant terms:

Term	Description
URL	URL and method to execute.
`authorization`	Authorization taken to pass to the platform. Basic authorization works. **NOTE:** This token must be passed with each request to the platform.
`cache-control`	Cache control setting.
`content-type`	HTTP content type to send. These applications use `application/json`.

Checkpoint: You've created an outputObject (`id=4`) and an embedded writeSettings object and have associated them with the appropriate recipe `flowNodeId=11`. You can now run a job for this recipe generating the specified output.

For more information, see *https://api.trifacta.com/ee/es.t/index.html#operation/createOutputObject*

Step - Run a Test Job

Now that outputs have been defined for the recipe, you can just execute a job on the specified recipe `flowNodeId=11`:

Request:

Endpoint	`http://www.wrangle-dev.example.com:3005/v4/jobGroups`
Authentication	Required
Method	POST
Request Body	`{` `  "wrangledDataset": {`

```
    "id": 11
  }
}
```

Response:

Status Code	201 - Created
Response Body	

NOTE: To re-run the job against its currently specified outputs, writeSettings, and publications, you only need the recipe ID. If you need to make changes for purposes of a specific job run, you can add overrides to the request for the job. These overrides apply only for the current job. For more information, see *https://api.trifacta.com/ee/es.t/index.html#operation/runJobGroup*

To track the status of the job:

- You can monitor the progress through the application.
- You can monitor progress through the `status` field by querying the specific job. For more information, see *https://api.trifacta.com/ee/es.t/index.html#operation/getJobGroup*

Checkpoint: You've run a job, generating one output in Avro format.

Step - Create writeSettings Object

Suppose you want to create another file-based output for this outputObject. You can create a second writeSettings object, which publishes the results of the job run on the recipe to the specified location.

The following example creates settings for generating a parquet-based output.

Request:

Endpoint	http://www.wrangle-dev.example.com:3005/v4/writeSettings/

Authentication	Required
Method	POST
Request Body	{ "delim": ",", "path": "hdfs://hadoop:50070/trifacta/queryResults/admin@example.com/POS_r03.pqt", "action": "create", "format": "pqt", "compression": "none", "header": false, "asSingleFile": false, "prefix": null, "suffix": "_increment", "hasQuotes": false, "outputObjectId": 4 }

Response:

Status Code	201 - Created
Response Body	{ "delim": ",", "id": 2, "path": "hdfs://hadoop:50070/trifacta/queryResults/admin@example.com/POS_r03.pqt", "action": "create", "format": "pqt", "compression": "none", "header": false, "asSingleFile": false, "prefix": null, "suffix": "_increment", "hasQuotes": false, "updatedAt": "2018-11-13T01:07:52.386Z", "createdAt": "2018-11-13T01:07:52.386Z", "creator": { "id": 1 }, "updater": { "id": 1 }, "outputObject": { "id": 4 } }

cURL example:

```
curl -X POST \
  http://www.wrangle-dev.example.com/v4/writeSettings \
  -H 'authorization: Basic <auth_token>' \
  -H 'cache-control: no-cache' \
  -H 'content-type: application/json' \
  -d '{    "delim": ",",
    "path": "hdfs://hadoop:50070/trifacta/queryResults/admin@example.com/POS_r03.pqt",
    "action": "create",
    "format": "pqt",
    "compression": "none",
    "header": false,
    "asSingleFile": false,
    "prefix": null,
    "suffix": "_increment",
```

```
    "hasQuotes": false,
    "outputObject": {
      "id": 4
    }
}
```

Relevant terms:

Term	Description
URL	URL and method to execute.
`authorization`	Authorization taken to pass to the platform. Basic authorization works. **NOTE:** This token must be passed with each request to the platform.
`cache-control`	Cache control setting.
`content-type`	HTTP content type to send. These applications use `application/json`.

Checkpoint: You've added a new writeSettings object and associated it with your outputObject (`id=4`). When you run the job again, the Parquet output is also generated.

For more information, see *https://api.trifacta.com/ee/es.t/index.html#operation/createWriteSetting*

Step - Get Connection ID for Publication

To generate a publication, you must identify the connection through which you are publishing the results.

Below, the request returns a single connection to Hive (`id=1`).

Request:

Endpoint	`http://www.wrangle-dev.example.com:3005/v4/connections`
Authentication	Required
Method	`GET`
Request Body	None.

Response:

Status Code	`200 - OK`
Response Body	{ "data": [{ "id": 1, "host": "hadoop", "port": 10000, "vendor": "hive", "params": { "jdbc": "hive2", "connectStringOptions": "", "defaultDatabase": "default" }, "ssl": false, "vendorName": "hive", "name": "hive", "description": null, "type": "jdbc", "isGlobal": true,

```
            "credentialType": "conf",
            "credentialsShared": true,
            "uuid": "28415970-e6c4-11e8-82be-9947a31ecdd5",
            "disableTypeInference": false,
            "createdAt": "2018-11-12T21:44:39.816Z",
            "updatedAt": "2018-11-12T21:44:39.842Z",
            "credentials": [],
            "creator":  {
                "id":  1
            },
            "updater":  {
                "id":  1
            },
            "workspace":  {
                "id":  1
            }
        }
    ],
    "count": 1
}
```


cURL example:

```
curl -X GET \
  http://www.wrangle-dev.example.com/v4/connections \
  -H 'authorization: Basic <auth_token>' \
  -H 'cache-control: no-cache' \
  -H 'content-type: application/json'
```

Relevant terms:

Term	Description
URL	URL and method to execute.
`authorization`	Authorization taken to pass to the platform. Basic authorization works. **NOTE:** This token must be passed with each request to the platform.
`cache-control`	Cache control setting.
`content-type`	HTTP content type to send. These applications use `application/json`.

For more information, see *https://api.trifacta.com/ee/es.t/index.html#operation/listConnections*

Step - Create a Publication

You can create publications that publish table-based outputs through specified connections. In the following, a Hive table is written out to the `default` database through connectionId = 1. This publication is associated with the outputObject id=4.

Request:

Endpoint	`http://www.wrangle-dev.example.com:3005/v4/publications`
Authentication	Required
Method	POST
Request Body	``` { "path": ["default"], "tableName": "myPublishedHiveTable", ```

```
    "targetType": "hive",
    "action": "create",
    "outputObject": {
        "id": 4
    },
    "connection": {
        "id": 1
    }
}
```

Response:

Status Code	201 - Created
Response Body	

```
{
    "path": [
        "default"
    ],
    "id": 3,
    "tableName": "myPublishedHiveTable",
    "targetType": "hive",
    "action": "create",
    "updatedAt": "2018-11-13T01:25:39.698Z",
    "createdAt": "2018-11-13T01:25:39.698Z",
    "creator": {
        "id": 1
    },
    "updater": {
        "id": 1
    },
    "outputObject": {
        "id": 4
    },
    "connection": {
        "id": 1
    }
}
```

cURL example:

```
curl -X POST \
  http://example.com:3005/v4/publications \
  -H 'authorization: Basic <auth_token>' \
  -H 'cache-control: no-cache' \
  -H 'content-type: application/json' \
  -d '{
    "path": [
        "default"
    ],
    "tableName": "myPublishedHiveTable",
    "targetType": "hive",
    "action": "create",
    "outputObject": {
        "id": 4
    },
    "connection": {
        "id": 1
    }
}'
```

Relevant terms:

Term	Description
URL	URL and method to execute.
`authorization`	Authorization taken to pass to the platform. Basic authorization works. **NOTE:** This token must be passed with each request to the platform.
`cache-control`	Cache control setting.
`content-type`	HTTP content type to send. These applications use `application/json`.

For more information, see *https://api.trifacta.com/ee/es.t/index.html#operation/createPublication*

Checkpoint: You're done.

You have done the following:

1. Created an output object:
    a. Embedded a writeSettings object to define an Avro output.
    b. Associated the outputObject with a recipe.
2. Added another writeSettings object to the outputObject.
3. Added a table-based publication object to the outputObject.

You can now generate results for these three different outputs whenever you run a job (create a jobgroup) for the associated recipe.

Step - Apply Overrides

When you are publishing results to a relational source, you can optionally apply overrides to the job to redirect the output or change the action applied to the target table. For more information, see *API Workflow - Run Job*.

Step - Apply Spark Job Overrides

You can optionally submit override values for a predefined set of Spark properties on the output object. These overrides are applied each time that the outputobject is used to generate a set of results.

NOTE: This feature and the Spark properties available for override must be configured by a workspace administrator. For more information, see *Enable Spark Job Overrides*.

Tip: You can apply Spark job overrides to the job itself, instead of applying overrides to the outputobject. For more information, see *API Workflow - Run Job*.

In the following example, an existing outputObject (id=4) is modified to include override values for the default set of Spark overrides. Each Spark property and its value as specified as a key-value pair in the request:

Request:

Endpoint	`http://www.wrangle-dev.example.com:3005/v4/outputObjects/4`
Authentication	Required
Method	PUT
Request Body	`{ "execution": "spark", "outputObjectSparkOptions": [ { "key": "spark.driver.memory", "value": "10G" },`

```
    {
      "key": "spark.executor.memory",
      "value": "10G"
    },
    {
      "key": "spark.executor.cores",
      "value": "5"
    },
    {
      "key": "transformer.dataframe.checkpoint.threshold",
      "value": "450"
    }
  ]
}
```

Response:

Status Code	200 - Ok
Response Body	<pre>{ "id": 4, "updater": { "id": 1 }, "updatedAt": "2020-03-21T00:27:00.937Z", "createdAt": "2020-03-20T23:30:42.991Z" }</pre>

cURL example:

```
curl -X PUT \
  http://www.wrangle-dev.example.com/v4/outputObjects/4 \
  -H 'authorization: Basic <auth_token>' \
  -H 'cache-control: no-cache' \
  -H 'content-type: application/json' \
  -d '{
  "execution": "spark",
  "outputObjectSparkOptions": [
    {
      "key": "spark.driver.memory",
      "value": "10G"
    },
    {
      "key": "spark.executor.memory",
      "value": "10G"
    },
    {
      "key": "spark.executor.cores",
      "value": "5"
    },
    {
      "key": "transformer.dataframe.checkpoint.threshold",
      "value": "450"
    }
  ]
}'
```

Relevant terms:

Term	Description
URL	URL and method to execute.

`authorization`	Authorization taken to pass to the platform. Basic authorization works. **NOTE:** This token must be passed with each request to the platform.
`cache-control`	Cache control setting.
`content-type`	HTTP content type to send. These applications use `application/json`.

API Workflow - Manage AWS Configurations

Contents:

- *Overview*
  - *Per-User Authentication Methods*
  - *Basic Workflow*
- *Step - Acquire information*
- *Step - Locate user*
- *Step - Create awsConfig object*
- *Step - Verify Authentication*
- *Step - For Method 2, assign new IAM role to awsConfig object*
- *Step - Switching Persons or Workspaces for an awsRole*
- *Step - Switching Authentication Methods*

Overview

The Trifacta® platform supports multiple methods of authenticating to AWS resources. At the topmost level, authentication can be broken down into two modes: system and user.

- **System mode:** One set of credentials is used for each user of the platform to authenticate to AWS.
- **User mode:** Individual user accounts must be configured with AWS credentials.

  NOTE: This section covers how to manage AWS credentials through the APIs for individual users (user mode). When in system mode, please manage AWS configuration through the application.

Per-User Authentication Methods

To connect to AWS resources and access S3 data, the following information is required for each user, depending on the method of authentication.

Method 1 - AWS Key and Secret

If users are providing key-secret combinations, the following information is required.

Item	Description
key/secret	(credential provider type is `default`) The AWS key and secret for the user to authenticate
default bucket	The default S3 bucket where the user can upload data and store generated results
extra buckets	Any extra S3 buckets to which the user should have access

Method 2 - AWS IAM Role ARNs

Users can access AWS resources by assigning an awsConfig object to the account.

Tip: This method is recommended.

The following information is required:

Item	Description
IAM role	(credential provider type is `temporary`) The IAM role to use to authenticate. **NOTE:** If this information is not immediately available, a placeholder one is created when you create the configuration object. You can assign roles later. More information is provided below.
default bucket	The default S3 bucket where the user can upload data and store generated results

extra buckets	Any extra S3 buckets to which the user should have access

Authentication objects

For each authentication method, the above pieces of information must be provided for each user.

These pieces of information are defined in an `awsConfig` object. An **awsConfig** object is a set of AWS configuration properties that can be created, modified, and assigned to individual users via API.

For Method 2, the awsConfig object maps to an awsRole object. An **awsRole object** references an IAM role and an awsConfig object. When you create an awsConfig object and its credential provider is set to `temporary`, the awsRole object is automatically created for you:

- Each awsRole object maps to a single IAM role.
- Each awsRole object is mapped to an awsConfig object.
- The awsConfig object is then assigned to individual users.
- Through this mechanism, you have more flexibility in assigning the active role to users.
- As needed, the awsConfig object can be mapped at a later time to another awsRole object through the `role` attribute.

This workflow steps through the process for all these methods.

Platform roles

To complete this workflow, your account must have one of the following roles:

- Workspace admin
- Trifacta admin

For more information, see

https://api.trifacta.com/ee/es.t/index.html#section/Authentication

Basic Workflow

1. Choose your method of authentication.
2. Locate the internal identifier for the user to which to assign the configuration object.
3. Create an `awsConfig` object, assigning the object to the user as part of the process.
4. Verify that the assignment is working.

Step - Acquire information

Acquire all of the information listed above for the awsConfig object you wish to create.

Step - Locate user

Now, you need to locate the internal identifier for the user to which you wish to assign this AWS configuration.

Request:

Endpoint	`http://www.wrangle-dev.example.com:3005/v4/people`
Authentication	Required
Method	`GET`
Request Body	None.

Response:

Status Code	200 - Ok
Response Body	


```
{
    "data": [
        {
            "id": 3,
            "email": "4070250@example.com",
            "name": "Test User4070250",
            "ssoPrincipal": null,
            "hadoopPrincipal": null,
            "isAdmin": false,
            "isDisabled": false,
            "forcePasswordChange": false,
            "state": "active",
            "lastStateChange": null,
            "createdAt": "2019-04-16T16:27:51.143Z",
            "updatedAt": "2019-04-16T16:27:56.630Z",
            "outputHomeDir": "/trifacta/queryResults/4070250@example.com",
            "fileUploadPath": "/trifacta/uploads",
            "awsConfigId": 2
        },
        {
            "id": 2,
            "email": "32870@example.com",
            "name": "Test User32870",
            "ssoPrincipal": null,
            "hadoopPrincipal": null,
            "isAdmin": false,
            "isDisabled": false,
            "forcePasswordChange": false,
            "state": "active",
            "lastStateChange": null,
            "createdAt": "2019-04-16T16:27:19.511Z",
            "updatedAt": "2019-04-16T16:27:26.703Z",
            "outputHomeDir": "/trifacta/queryResults/32870@example.com",
            "fileUploadPath": "/trifacta/uploads",
            "awsConfigId": 1
        },
        {
            "id": 1,
            "email": "<admin_email>",
            "name": "Administrator",
            "ssoPrincipal": null,
            "hadoopPrincipal": null,
            "isAdmin": true,
            "isDisabled": false,
            "forcePasswordChange": false,
            "state": "active",
            "lastStateChange": null,
            "createdAt": "2019-04-16T07:44:04.299Z",
            "updatedAt": "2019-04-16T16:28:16.379Z",
            "outputHomeDir": "/trifacta/queryResults/admin@example.com",
            "fileUploadPath": "/trifacta/uploads",
            "awsConfigId": 3
        }
    ]
}
```


Checkpoint: In the above, you noticed that userId=2 is associated with awsConfig object id=1, which is the one you are replacing. This is the user to modify. Retain this value for later.

For more information, see *https://api.trifacta.com/ee/es.t/index.html#operation/getPerson*

Step - Create awsConfig object

Create the AWS configuration object.

NOTE: Optionally, the `personId` value can be inserted into the request to assign the AWS configuration object to a specific user at create time, when it is created by an admin user. If it is created by a non-admin user, the object is assigned to the user who created it.
NOTE: For Method 2, an awsRole object is automatically created for you when you create the awsConfig object. It is mapped to the awsConfig object.

Request:

Endpoint	`http://www.wrangle-dev.example.com:3005/v4/awsConfigs`
Authentication	Required
Method	POST
Request Body	**Method 1:** AWS key-secret combination `{ "credentialProvider": "default", "personId": 2, "key": "<my_key>", "secret": "<my_secret>", "defaultBucket": "main_bucket", "extraBuckets":["extra-bucket1","extra-bucket2"] }` **Method 2:** IAM role `{ "credentialProvider": "temporary", "personId": 2, "role":"<my_iam_role_object>", "defaultBucket":"main_bucket", "extraBuckets":["extra-bucket1","extra-bucket2"] }`

Response for Method 2:

Status Code	`201 - Created`
Response Body	**Method 2 example:** `{ "extraBuckets": [ "extra-bucket1", "extra-bucket2" ], "id": 6, "defaultBucket": "main_bucket", "credentialProvider": "temporary", "externalId": null, "activeRoleId":"4", "updatedAt": "2019-04-16T23:06:32.049Z", "createdAt": "2019-04-16T23:06:32.047Z", "credential": null }`

Checkpoint: In the above, the awsConfig object has an internal identifier (`id=6`).

As part of the request, this object was assigned to user 2 `personId=2`.

The `activeRoleId` attribute indicates that the internal ID of the awsRole object that was automatically created for you.

For more information, see *https://api.trifacta.com/ee/es.t/index.html#operation/createAwsConfig*

Step - Verify Authentication

To verify that the above configuration works:

1. User id=2 should login to the application.
2. User uploads assets through the Import Data page.
3. User creates a short recipe that modifies these assets.
4. User runs a job on that recipe to generate output to the default S3 bucket in CSV or JSON for downloading.
5. User verifies that the results can be downloaded.

Checkpoint: Configuration and verification is complete.

Step - For Method 2, assign new IAM role to awsConfig object

If you need to change the IAM role ARN for a user, you can modify the awsConfig object for that user with the new role information.

NOTE: This section only applies if `credentialProvider` has been set to `temporary` for the object and if you are using multiple IAM role ARNs in the Trifacta platform.

The following request modifies the awsConfig `id=6`.

Request:

Endpoint	`http://www.wrangle-dev.example.com:3005/v4/awsConfigs/6`
Authentication	Required
Method	PUT
Request Body	`{ "role":"<my_iam_role_object_3>" }`

Response:

Status Code	`200 - OK`

```
    "createdAt": "2019-04-16T23:06:32.047Z",
    "credential": null
}
```


Checkpoint: In the above step, you assigned a new IAM role to the awsConfig object. The underlying awsRole object is created for you and automatically assigned. For more information, see *https://api.trifacta.com/ee/es.t/index.html#operation/createAwsRole*

NOTE: After you have completed the above update, the previous awsRole object still exists. If the IAM role associated with it is no longer in use, you should delete the awsRole object. See *https://api.trifacta.com/ee/es.t/index.html#operation/deleteAwsRole*

Step - Switching Persons or Workspaces for an awsRole

When you create or modify an awsRole, you can optionally pass in a person or workspace identifier. When either value is provided, the Trifacta platform attempts to assign the awsRole to the provided identifier based on the related awsConfig object.

Acquire awsRole identifier

Via awsConfig identifier

Use the following endpoint to retrieve the awsConfig object. This one uses awsConfigId=6:

Request:

Endpoint	`http://www.wrangle-dev.example.com:3005/v4/awsConfigs/6`
Authentication	Required
Method	`GET`
Request Body	Empty.

Response:

Status Code	`200 - OK`
Response Body	```{ "extraBuckets": ["extra-bucket1", "extra-bucket2"], "id": 6, "defaultBucket": "main_bucket", "credentialProvider": "temporary", "externalId": null, "activeRoleId":"<awsRoleId>", "updatedAt": "2019-04-16T23:06:32.049Z", "createdAt": "2019-04-16T23:06:32.047Z", "credential": null }```

Acquire the value for `activeRoleId`.

Via awsRoles identifier

Use the following request to retrieve all of the awsRoles:

Request:

Endpoint	`http://www.wrangle-dev.example.com:3005/v4/awsRoles`
Authentication	Required
Method	GET
Request Body	Empty.

In the response, locate the appropriate identifier.

Reassign the awsRole

You can use the PUT method of the following endpoint to re-assign the awsRole to the specified person or workspace. The following example reassigns awsRoleId=3 to personId=6.

Request:

Endpoint	`http://www.wrangle-dev.example.com:3005/v4/awsRoles/3`
Authentication	Required
Method	PUT
Request Body	`{ "personId": 6 }`

If the request is successful, the awsRole is reassigned to the person identifier.

Tip: In the above request, you can replace `"personId"` with `"workspaceId"` to reassign the role to all users in a workspace.

Step - Switching Authentication Methods

Suppose you have created your awsConfig objects to use the AWS Key-Secret method of authenticating. You have now created a set of IAM roles that you would like to assign to your Trifacta users.

The generalized workflow for completing this task is the following:

1. Acquire the identifiers for all of the awsConfigs you wish to modified. For each awsConfig, retain the `personId`, so that you can map your configuration changes to individuals. See *https://api.trifacta.com/ee/es.t/index.html#operation/listAwsConfigs*
    a. For more information on getting your list of users, see *https://api.trifacta.com/ee/es.t/index.html#operation/getPerson*
2. For each user account (`personId`), you must identify the IAM role that you wish to assign it.
3. Use the following modification to the awsConfig object to switch to using the IAM role for the user:

Request:

Endpoint	`http://www.wrangle-dev.example.com:3005/v4/awsConfigs/<awsConfigId>`
Authentication	Required
Method	PUT
Request Body	

```
{
    "credentialProvider": "temporary",
    "role":"<my_iam_role_object>"
}
```

Response for Method 2:

Status Code	200 - Ok
Response Body	**Method 2 example:** `{ "extraBuckets": [ "extra-bucket1", "extra-bucket2" ], "id": <awsConfigId>, "defaultBucket": "main_bucket", "credentialProvider": "temporary", "externalId": null, "activeRoleId":"<awsRoleId>", "updatedAt": "2019-04-16T23:06:32.049Z", "createdAt": "2019-04-16T23:06:32.047Z", "credential": null }`

Notes:

Item	Description
`credentialProvider`	To use IAM roles, this attribute must be updated to be `temporary`.
`role`	The IAM role to assign to the configuration.
`personId`	If needed, you can change the person (user) to which this awsConfig is applied. Note that the former user of the configuration cannot access AWS resources until you create a new configuration object for the user's account.
`activeRoleId`	(response) Internal identifier of the awsRole object that was created for you and assigned to this awsConfig object.

NOTE: The above request must be applied to each awsConfig object that you wish to remap to using an IAM role.

API Workflow - Run Plan

Contents:

- *Pre-requisites*
- *Step - Run Plan*
- *Step - Monitoring Your Plan Run*
- *Step - Add Flow Messages*

This section describes how to run a plan using the APIs available in Trifacta® Wrangler Enterprise.

- A plan is a scheduled sequence of tasks based on a trigger that you define.
    - When a plan is executed via API, the request is the trigger, and the plan is executed immediately.
- Plans can be designed in the Trifacta application. For more information, see *Plans Page*.
- For more information on plans in general, see *Overview of Operationalization*.

A note about API URLs:

In the listed examples, URLs are referenced in the following manner:

```
<protocol>://<platform_base_url>/
```

In your product, these map references map to the following:

```
<http or https>://<hostname>:<port_number>/
```

For more information, see *API Reference*.

Pre-requisites

Before you begin, you should verify the following:

1. **Get authentication credentials**. As part of each request, you must pass in authentication credentials to the platform.

   Tip: The recommended method is to use an access token, which can be generated from the Trifacta application. For more information, see *Access Tokens Page*.

   For more information, see *https://api.trifacta.com/ee/es.t/index.html#section/Authentication*
2. **Verify plan and its flows and outputs:**
    a. You must create a plan first. See *Plan View Page*.
    b. As part of creating that plan, you must verify that all referenced flows and output objects are properly defined and can be executed independently.

       NOTE: In a flow, all recipes that you wish to have executed by the corresponding task must have a defined output object. For each output object, you must create at least one write settings or publication object. During plan runs, these objects are not validated, and tasks fail without them.
    c. Any applicable parameters are applied to the tasks at the time of execution. Parameter overrides are not supported in plans.
    d. See *Flow View Page*.
3. **Verify plan execution.** Run the desired plan through the Trifacta application and verify that the output objects are properly generated. See *Plan View Page*.

4. **Acquire plan identifier.** In Plan View, acquire the numeric value for the plan from the URL. In the following, the plan Id is `1234`:

   ```
   http://<platform_base_url>/plans/1234
   ```

5. **Activate the plan.** Plans are stored as a series of versions of the plan. Only the latest activated version of a plan can be run.

   NOTE: Only the latest active version of a plan can be executed via API. If the Activate button is enabled in Plan View, there are changes to the latest active version in the application. To include those changes in your plan run, click **Activate** again.

Step - Run Plan

Through the APIs, you can run a plan. Construct a request like the following, where:

- `<id>` is the plan identifier that you already extracted from the Plan View URL.

Endpoint	`<protocol>://<platform_base_url>/v4/plans/<id>/run`
Authentication	Required
Method	`POST`
Request Body	None.
Response Code	`201 - Created`
Response Body	<pre>{ "validationStatus": "Valid", "planSnapshotRunId": 2 }</pre>

If the `201` response code is returned, then the plan has been queued for execution.

Tip: Retain the `id` value in the response. In the above, `2` is the internal identifier for the plan run, which is referenced via the generated snapshot of the corresponding flows in the plan's tasks. You will need this value to check on your plan run status.

For more information, see *https://api.trifacta.com/ee/es.t/index.html#operation/runPlan*

Checkpoint: You have queued your plan for execution.

Step - Monitoring Your Plan Run

You can monitor the status of your plan run through the following endpoint, where:

- `<id>` is the plan snapshot identifier for your run that you retained from the previous step.

Endpoint	`<protocol>://<platform_base_url>/v4/planSnapshotRuns/<id>`
Authentication	Required
Method	`GET`
Request Body	None.

Response Code	200 - Ok
Response Body	<pre>{ "id": 2, "status": "InProgress", "scheduleHistoryId": null, "startedAt": "2020-04-23T17:53:33.466Z", "finishedAt": null, "submittedAt": "2020-04-23T17:53:32.993Z", "executionId": null, "createdAt": "2020-04-23T17:53:33.312Z", "updatedAt": "2020-04-23T17:53:33.499Z", "plan": { "id": 1 } }</pre>

When the plan run has successfully completed, the returned status message includes the following:

```
"status": "Complete",
```

For more information, see *https://api.trifacta.com/ee/es.t/index.html#tag/PlanSnapshotRun*You can also review your plan runs through the Trifacta application at the following URL:

```
<protocol>://<platform_base_url>/plans/<planId>/runs/<planSnapshotRunId>
```

Tip: You have executed the plan run. Results have been delivered to the designated output locations.

Step - Add Flow Messages

You can configure flow webhooks and email notifications to deliver to stakeholders through the individual flows that are referenced in your plans.

NOTE: These features may require enablement and configuration in your environment.

For more information on these messaging types, see *Overview of Operationalization*.

A flow **webhook** is a REST API-based message that is triggered on the success or failure of generating an output from a flow. When the output referenced in a plan is generated, any webhook messages for the output are also triggered. Some uses:

- You can configure webhooks to deliver messages for each output referenced in the flow. Based on the schedule for your flow, you can review these messages to determine if the flow executed properly.
- You can configure a final output in the final task that is executed after the upstream recipes in the same flow.
    - All of the upstream recipes in the flow feed into a final recipe, which generates an unused output.
    - When you create a webhook based on this final output, you can send a message that the final task has been executed.
    - For more information on creating webhooks, see *Create Flow Webhook Task*.

An **email notification** is an email that is sent through the configured SMTP server to stakeholders based on the successful or failed execution of an output. You can define email notifications for your individual flows, and these messages get delivered as part of the flow execution that is part of the plan.

Tip: When an email notification is sent as part of task execution, the internal plan identifier is included as part of the message.

For more information on email notification, see *Manage Flow Notifications Dialog*.