TOC

Composer 24

Composer 24 Developer Tools

Developer Resources

The following topics provide information about the developer resources available in Composer.

- Upgrade Composer With Custom Applications
- Get Started With The Composer Application Framework
- Composer REST API Overview
- Manage Custom Charts
- Manage UI Themes
- Trusted Access
- White Label The Composer Interface

Upgrade Composer with Custom Applications

Because Composer is installed software, your organization must decide whether and when to install any particular upgrade. The following considerations will help you understand what is involved in upgrading to any version of Composer.

Did You White Label Composer with a Custom CSS?

The CSS in Composer evolves with the client application. Differences in the CSS should be considered before you upgrade. You should examine all existing CSS and modify it accordingly.

Does Your Application Use REST APIs?

Composer's REST API offerings change regularly. Be sure you review the Release Notes regularly for changes in API endpoints.

Does Your Application Use an iFrame-Embedded Dashboard?

Any dashboard already embedded in a custom application using an iFrame will continue to work with newer versions of Composer. iFrame-embedded dashboards have additional capabilities that are invoked using parameters included with the embedding code.

Does Your Application Use the JavaScript Client Library?

The JavaScript client library is used to embed visuals or data directly into a web application without using an iFrame. Before upgrading Composer, consider the following topics.

Visual Variables

Visual variables are written in standard JSON and left unstringified.

JSON Modifications

Some key-value pairs are now wrapped in objects. Review the documentation for updated JSON samples.

Deprecated Objects and Methods

Deprecated objects are methods are mentioned in the Release Notes.

Have You Created a Custom Connector?

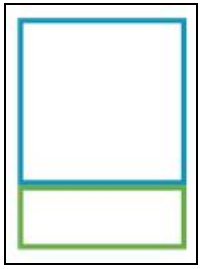
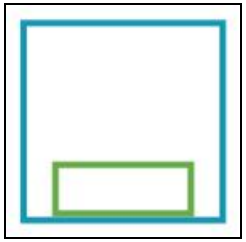
Custom connectors built to work with previous versions of Zoomdata or Composer should continue to function as expected with newer versions.

Get Started with the Composer Application Framework

Composer's application framework provides you with the tools you need to query data or embed visuals from Composer in your own application. If you want to use Composer visuals or data in your own custom application, then read on. You might be looking for information about creating your own custom charts, see Maintain Custom Charts Using The Custom Chart CLI.

Note: insightsoftware recommends using Trusted Access for all embed-related workflows.

Uses of the Application Framework

	**You can use a query to get data from Composer and use it in your own applications.** You might want to go this route if: ■ You just need the data. ■ You already have a visual built into your application and you want to keep it, but need data for it.
	**You can embed a Composer visual in your application and use your query to supply it with data.** You might want to go this route if: ■ You already have a data query set up. ■ You need to use the same query for more than one visual or for visuals and other purposes.
	**You can embed a Composer visual and give it the information it needs to create its own query.** You might want to go this route if you really just need one visual based on its own query.

Dependencies

The application framework depends upon jQuery. Composer recommends that you link jQuery v1.8 or later.

Accessing the Application Framework

Composer's application framework is provided by linking the `zoomdata-client.js` file in your web application. You can find `zoomdata-client.js` on your installation at

```
https://<yourserver>/composer/sdk/zoomdata-client.js
```

replacing `<yourserver>` with the URL for your server.

Best practice: Use the `zoomdata-client.js` file found on the Composer installation that will be supplying your data and visuals. There is not usually a problem using different versions, but following this guidance avoids such issues and may help Composer support your work.

Linking the zoomdata-client.js file into your web application gives you access to the `ZoomdataSDK` object. The main purpose of the ZoomdataSDK object is to create a Composer client for your application. Typically, one Composer client is enough, but if you need to access multiple Composer servers, you'll need one client for each one.

Typical Workflows

There is a lot of flexibility with the Composer application framework, but some steps must precede others. When steps must be kept in a particular order, the Composer application framework uses promises to do so. Below is an example of the most common workflow.

Typical Workflow to Query Data

This workflow creates a query and then queries data from it. This workflow is useful if you already have visuals in your application and you need to supply them with data from Composer. It's also useful if you just want the data.

1. Create a Composer client. Steps to create a client are found in Use A Data Query.
2. Create a Composer query.
3. Run the Composer query.
4. Gather and use the queried data.

To help you get going, you might want to fiddle with an example or download it from GitHub.

Next Steps

It might be that you only need to embed a visual or use data from Composer in your own application. Consider the information at the following links:

- Use A Data Query
- Composer REST API Overview
- Maintain Custom Charts Using The Custom Chart CLI
- Trusted Access

Use a Data Query

The Composer JavaScript client library provides functions to query the Composer server for data. This guide uses a method that does not provide event handling for the queries. To assist you in querying for data without handling events, Composer provides:

- This topic, which includes annotated steps for using a query in your own web app.
- A simple example of a query used to render a table, with annotated code, discussed in this topic, and available for download at the Developer Zone.

For information about the structure of the query configuration object, see Query Configuration Object.

How a Query Works

The Composer object is used to create a Composer client object. This client is then used to create a query object. To run the query, you pass it and a data processing function to `client.runQuery()`. The client.runQuery() function retrieves data and passes it to the data processing function.

Steps for Using a Query in Your Web App

These high-level steps for using a query in a web application will guide your work.

1. Link Dependencies
2. Organize Application And Security Parameters
3. Configure The Query
4. Code The Query
5. Run The Query
6. Find Your Data

Link Dependencies

Using Composer data queries depends on the `zoomdata-client.js` library. You can link to this library on your Composer server at `composer/sdk/zoomdata-client.js`.

The application framework and example code used in this topic also depend on jQuery (`jQuery.js`). Composer recommends that you link jQuery v1.8 or later. Your script must have access to this library to use the example.

Organize Application and Security Parameters

Connecting the webpage to your Composer server requires supplying the Composer server with application and security parameters in the form of JavaScript objects. These two objects are themselves bundled together as a single object to be passed to the `createClient()` method. For example:

```
ZoomdataSDK.createClient({
       credentials: credentialConfig,
       application: applicationConfig
})
```

For more information about the application configuration object, see Application Configuration Object.

For more information about the security configuration object, see Data Discovery Security Configuration Object.

Configure the Query

Before you can create a query, you must create a query configuration object. This object is used to create a query using the Composer client's member function `createQuery()`. For example, the query configuration object below gathers from its data source up to 200 `productGroup` items, sorting them in ascending order and measuring them by their average price, filtering out any groups whose average price is not less than 100.

```
var queryConfig = {
       tz: 'EST',
       time: { timeField: '_ts' },
       player: {},
       filters: [{ path: 'price', operation: 'LT', value: 100 }],
       groups: [{
              name: 'productGroup',
              limit: 200,
              sort: { dir: 'asc', name: 'productGroup' }
       }, {
              name: 'productCategory',
              limit: 200,
              sort: {dir: 'asc', name: 'productCategory'}
       }],
       metrics: [{
              name: 'price',
              func: 'avg' }]
};
```

Code the Query

The `createClient()` and `runQuery()` functions are used to create and run a query. The `runQuery()` function returns a promise, so you can chain `then()` and `done()` functions to it.

To code a query:

1. Instantiate a Composer client if you do not already have one that you want to use.

```
ZoomdataSDK.createClient({
       credentials: credentialConfig,
       application: applicationConfig }).then((client) => {
                    // where client is the newly created instance of the client
              });
```

2. Use the client's `createQuery()` method to create the query. The required parameters are a data source available to the client (based on the server it accesses and its permissions) and a query configuration object.

```
client.createQuery({name: 'Real Time Sales'}, queryConfig).then((queryInstance) => {
                        // resulting queryInstance can be used to query for data from the data source
               });
```

Run the Query

The Composer client library offers different ways to run the query to gather data. This tutorial uses the `client.runQuery()` method, which applies a function to each data object resulting from the query. There is also a `client.run()` method, which returns a thread that contains both the data and other messages that are useful for working with data.

To run a query with client.runQuery():

1. Supply `runQuery()` with these parameters:
   a. a query, which you can create using the steps found in Coding a Query
   b. a function to execute on each object resulting from the query, which should take one parameter

```
client.runQuery( theQuery, function( dataObject ) {
       console.log( 'Data object returned by query: ', dataObject );
});
```

2. Typically, you should chain a .catch() function to handle errors after the event handling function

```
.catch( function( theError ) {
       console.log( 'Error: ', theError );
});
```

The function `runQuery()` extracts a data object, rather than a single piece of data, from your query and passes it to your data processing function. For more information about finding the data you need in the returned data object, see Find Your Data.

Full Example

```
ZoomdataSDK.createClient({
       credentials: credentialConfig,
       application: applicationConfig })
.then(client => {
       client.createQuery({name: 'Real Time Sales'}, queryConfig)
              .then(query => {
                     client.runQuery( query, function( dataObject ) {
                                  // dataObject contains array of data from back end responses.
                                  console.log( 'Data object returned by query: ', dataObject );
                      });
             })
})
});
```

Find Your Data

The `runQuery()` function returns a data object or more commonly an array of data objects. These objects each have the same structure, which can be predicted by the query configuration and also discovered programmatically.

Structure of the Data Object

Each data object consists of the following objects:

- **current** : an object that contains a count value and a metrics object
  - **count** : indicates the number of rows of data represented in the data object
  - **metrics** : contains one object for each metric

```
current:
       count: 131
       metrics:
               revenue:
                       avg: $23.32
               profit:
                       avg: $13.16
```

  - individual metric objects contain a key:value pair. The key is the operation of the metric, such as `'avg'` or `'min'`. The value is the metric value, for example:
  - The current object above indicates that 131 rows of data are in the queried group, and have an average revenue of $23.32 and average profit of $13.16

- **group** : an array listing the groups aggregated in the data object
  - for example

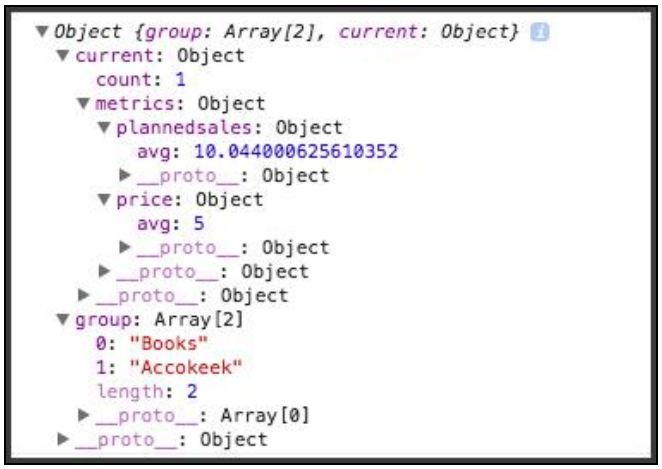
```
group:
       0: 'Gaithersburg'
       1: 'Coffee'
```

The groups object above represents rows of data that satisfy both grouping requirements: `'Gaithersburg'` and `'Coffee'`, which, depending on the data set, may represent sales of coffee in Gaithersburg.

Isolate a Piece of Data Manually

If you know the structure of the query in advance, you can reliably anticipate the structure of the data object. The query returns metrics, groups, and fields in the order they are found in the query object.

To verify the order, use a `console.log()` statement to output to the debugging console a sample data object returned by the query.



Individual elements of the data object can be isolated and used as a JavaScript object. For example, to access the average price in the data object above, use either of the following expressions.

```
theDataObject.current.metrics.price.avg
```

or

```
theDataObject['current']['metrics']['price']['avg']
```

Isolate a Piece of Data Programmatically

You will not always know in advance which metrics, groups, and fields are involved in a data structure. The Query API provides accessor functions to identify programmatically the metrics, groups, and fields involved in a query. These functions are useful in the event that a query is subject to change at runtime. These accessors include:

- `query['metrics'].get()`
- `query['groups'].get()`
- `query['fields'].get()`

Each of the accessors above returns an array of objects with one object for each metric, group, or field, respectively. These objects are structured as they are structured in the query configuration object. For more information about query configuration objects, see Query Configuration Object. In the event that there are no metrics, groups, or fields in the query, that particular accessor will return an empty array.

Using the accessors, you can use the following steps to programmatically iterate through the metrics, groups, or fields used by your data query.

1. Create an array with the names of metrics, groups, or fields
2. Use the array to iterate through the returned data objects

Each of these steps is described below in more detail.

To create an array with the names of metrics, groups, or fields:

These steps use metrics as an example. You can also use groups or fields with the same procedure by replacing metrics with groups or fields.

1. Call query['metrics'].get() and assign the returned array to a variable.

```
var metrics = query['metrics'].get();
```

2. Iterate through the returned array using its forEach() method. Pass to forEach() an anonymous function to extract the name of each metric and add it to an array of the purpose of storing only the names of metrics.

```
var metricNames = [];
metrics.forEach( function(metric) {
       metricNames.push( metric.name );
});
```

These steps combined produce an array containing only the names of the metric objects, in the same order that they appear in both the returned metric object list and in the query configuration object.

To use the array to iterate through the returned data objects:

These steps use metrics as an example. You can also use groups or fields with the same procedure by replacing metrics with groups or fields.

Use the anonymous function to touch each data point and perform on it whatever operation you need.

```
dataObjectArray.forEach( function(dataObject) {
       console.log( metricName, ' is: ', dataObject['current']['metrics'][metricName]['avg'] );
});
```

In the same way, you can gather and automatically process data by field or group.

Composer REST API Overview

The Composer REST API provides methods for retrieving, updating, and deleting Composer metadata pertaining to accounts, connections, and users.

API documentation is provided with your Composer installation at this link: *`https://<composer-URL>/composer/swagger-ui.html`*.

Important: Some API endpoints are marked as `experimental` in the Swagger documentation we provide. These endpoints are in the early stages of design and are subject to change. We make no commitment to their stability and may remove them without notice. These experimental endpoints are not recommended for use in production.

This topic describes the following REST API topics:

Authentication

The API currently uses basic access authentication. Most actions require admin-level access to the particular account manipulated using the API.

Version Compatibility

Each version of the API maintains backward compatibility with previous versions. The HTTP Accepts/Content-type header is used to specify the version for the resources that are produced and used by the API. When sending data to or receiving data from the API, you must use the appropriate HTTP header.

When working with the API, you should explicitly set the version of the API that you want to use. For example, in cURL requests, you would include the `--header` parameter as shown below.

```
# For submitting data
curl --header "Content-type: application/vnd.composer.v3+json" ...

# Consuming data
curl --header "Accept: application/vnd.composer.v3+json,application/vnd.composer+json" ...
```

To ensure you always use the latest version of Composer, use the generic media type: `application/vnd.composer+json`.

Note: There is a risk associated with using this generic media time to ensure you always use the latest version. Because this approach does not tie calls to specific API versions, if the latest version changes, your integration may break. It is safer to tie your API calls to specific versions.

The current API media type version will be sent as a header, `X-Composer-Media-Type`, in the response to every API request.

The rule for specifying media types is simple. Always include the generic media type `application/vnd.composer+json`and, if you specify a particular version, always put it first. For example:

- Correct: `application/vnd.composer.v3+json,application/vnd.composer+json`
- Incorrect: `application/vnd.composer+json, application/vnd.composer.v3+json`

HATEOAS Architecture

The Composer API implements the HATEOAS architectural pattern. Most resources returned by the API have a `links` element. The links elements is an array of objects, each with two properties:

- `rel`: short description of how this link relates to the parent object
- `href`: the actual link value

Every resource will at least have a `rel: "self"` link, which is the canonical reference to that object. You can use the other links to explore the relations of a given object, for example, the users in a tenant or the sources tied to a connection.

API Entry Point

The initial entry point to the API is `http://<host>:<port>/<context>/api`. For example:

```
http://localhost:8080/composer/api
```

This URL points to the root API resources that you can use to browse through the API.

The /api/ Path

Some functions of the Composer client application call APIs using a `/api/` path. Except where noted in Composer's developer documentation, endpoints that use the `/api/` path are for Composer's internal use only.

Note: Applications built with Composer's internal-use APIs, are subject to a high risk of unexpected breakage.

APIs on the `/api/` path that are safe to use include, for example, the `/api/sources/key` endpoint.

Optimistic Locking on PATCH Calls

API data source PATCH calls use optimistic locking to ensure the PATCH is applied to the most recent records. A `version` parameter is used in these calls to track the version counter number. If this counter is not incremented in the payload of the call or if it is in error, the PATCH call will fail with an error similar to this:

```
"details": "Source Epsilon Impala API Update Test has been updated since last read. Refresh required",
"error": "INTERNAL_ERROR"
```

The exact and correct version counter number must be used or the API call will fail. For example, if you just submitted a PATCH call using version 8 and your next PATCH call uses version 10, you will receive an error response.

Consequently, prior to any PATCH call, be sure you do one of the following things:

- Perform a GET call immediately prior to the PATCH call to retrieve the latest version counter number. This is the recommended approach.
- If no changes have been made using the UI, and you can safely remember the version counter number of the previous call after every successful PATCH call, simply increment the version counter number in your newest PATCH call.

Reference Documentation

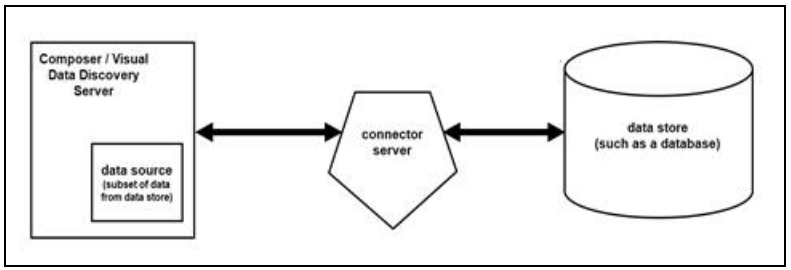
Composer maintains Swagger-based reference materials for the REST APIs. In this testing environment, you can experiment with the APIs features.

API documentation is provided with your Composer installation at this link: *`https://<composer-URL>/composer/swagger-ui.html`*.

Important: Some API endpoints are marked as `experimental` in the Swagger documentation we provide. These endpoints are in the early stages of design and are subject to change. We make no commitment to their stability and may remove them without notice. These experimental endpoints are not recommended for use in production.

Manage Connectors with REST APIs

The administrative REST APIs enable developers to manage connectors, connections, and data sources. Composer's data connectivity with data stores is organized in these segments, illustrated below.



There are two basic parts to the connectivity process. They are listed below along with the REST endpoints used to manage them.

- Composer registers a connector, that is a *connector server* from which it requests data. Registering a connector server involves the following REST endpoints:
  - `/api/connector-server/` - Use this endpoint to manage the Composer object that provides HTTP connectivity information about the connector server
  - `/api/connectors/` - Use this endpoint to list functioning connector servers to validate that Composer has registered the connector server
  - `/api/connection-type/` - Use this endpoint to manage the object that holds the data store parameters and connection parameters that the connector server needs to connect to a data store. This information commonly includes minimum and maximum supported version and connection parameters such as authentication information. A connection-type corresponds to an icon in the administrator's Sources page and makes it possible for end users to provide login credentials and make a data source from a data store.

- The connector server connects to a *data store* such as a database, or file upload to create a subset of data called a *data source*. That data source serves as a source of data for end users to use with visuals. Connecting to a data store involves the following REST endpoints:
  - `/api/connections/` - Use this endpoint to provide required parameters, often login credentials, that a connector server uses to access a data store following parameters set by the connection-type.
  - `/api/sources/{id}` - Use this endpoint to create a data source using a connection to a data store. End users can see the date in this data source using visuals.

Registering Composer with a Connector Server

Before Composer can use a connector server, it must register the connector server. Unless connector microservice discovery is turned off, Composer automatically discovers preinstalled connector servers when Composer is started up.

Composer does not automatically discover custom-built connector servers. You can use the Composer client application to manually register a connector server, or you can do so using REST APIs. For more information about using the Composer client application to register a connector server, see Configuring a New Connector Manually.

Using REST APIs to Register a Composer Connector Server

Before Composer can access a connector server, the connector server must be registered with Composer. If automatic discovery does not work, then you must do this process manually using either the Composer client application or using APIs. Registering a connector server requires providing Composer with the information that it needs to connect with and understand the connector server. Use the following steps to register a connector server using Composer's REST APIs.

To register a connector server with Composer:

1. Provide the Composer server with the information that it needs to connect to the connector server using the POST method with the `/api/connector-servers/` endpoint. Of the body parameters listed in the reference documentation, the following are required:

```
{
        "name": "<myDB Connector Server>",
        "protocol": "<HTTP>"
        "connectionParameters": {
                "HTTP_URL": "<yourdbURLhere>"
        },
}
```

   in which:

i. `name` should be unique and easily remembered.

ii. `protocol` should be set as `HTTP`. It is set by Composer as DISCOVERY or CORE if the connector was automatically discovered or built into Composer. If you do manual API-based registering, use HTTP.

iii. `HTTP_URL` should be set to the URL used to access the connector server.

2. After connector server discovery has detected and registered a connector server, or after you have manually registered the connector server with Composer using the `/api/connector-servers/` endpoint, Composer can validate that the connector server is registered by using the `/api/connectors/` endpoint.

   The API returns a dynamically generated list of all registered connectors.

3. Provide additional connection parameters to Composer using the POST method with the `/api/connection-types/`endpoint.

```
{
        "name": "<mySQL Connector>",
        "storageType": "MYSQL"
}
```

in which:

i. `name` is a string by which the connection type is to be identified by the end user on the Sources page. There may be more than one connection type (configuration) for how each connector is used. End users see each of these as a distinct type of connector, so the best practice is to name it as if it were a distinction connector. For example, you might make two different connection types to access a MySQL database using your MySQL connector server. One connection type might be low-security and use generic credentials for all users. The other connection type might require a user to provide specific, personal credentials that the connector server then passes along to the MySQL database to access specific data available only to that person. As such, the two connection types, showing up as two distinct connectors when a user creates a source, might be called "All-company MySQL Connector" and "Employee-specific MySQL Connector".

ii. `storageType` corresponds to the `storageType` returned by a connector server when it is asked to describe the database to which is connects. The storage types available via a particular connector server can be found by making a GET call to `/api/connector-servers/{connector-server-id}/connection-types`.

4. Optionally, provide a custom icon to denote the new connection type using the `/api/connection-types/{id}/icon/` endpoint. The endpoint expects one formData parameter - the icon image. It should be a 72 x 72 image in the form of either an SVG string or a PNG file in base 64 format.

Connecting to a Data Store

After the connector server is registered with Composer and one or more connection types have been defined, you can use the connector server to connect to a data store and create a data source.

1. Create a connection via the connector server to a data store by using the POST method with the `/api/{accountId}/connections/` endpoint.

   The response object includes the ID of the created connection at `theResponseObject.links[0].href`. This value is a string representation of a URL, the last component of which is the id of the object itself. For instance, in `https://latest.insightsoftware.com/composer/api/connections/58176703e4b06f699c7c70ca` the ID of the object is `58176703e4b06f699c7c70ca`.

   You can also modify an existing connection by using PUT with the `/api/connections/{id}/` endpoint. The parameters required by these endpoint are conditioned by the configuration provided by `/api/connection-type/`.

```
{
        "name": "ES Test Connection",
        "connectorName": "elasticsearch",
        "connectorParameters": [{...}, {...}, ...]
}
```

in which:

i. `name` is the name of the connection.

ii. `connectorName` is the name of the connector used to make the connection.

iii. `connectorParameters` is an object of connector parameters as key-value pairs. For example:

```
{
        "enableSsl": "false",
        "port": "9200",
        "clusterName": "es",
        "host": "10.2.2.129",
        "transportType": "http"
}
```

Connector parameters are different for each connector. Connector servers present these requirements when they are discovered by Composer. The `connectorParameters` can be identified by calling `/api/connection-types` and finding the `parameters` object for the appropriate `storageType`.

Note: Composer periodically introduces or deprecates connector servers. For information about the current status of a particular connector server, see Manage Connectors And Connector Servers.

iv. `sourceParameters` and `impersonationParameters` may also be required for a particular connection. This information can be found using the approach used to discover `connectorParameters`.

2. Create a data source, that is, a subset of data from a data store, by using POST with the endpoint `/api/connections/{connectionId}/sources/`. Required parameters for the source are listed below.

```
{
       "name": "string",
       "sourceParameters": {...},
}
```

in which:

i. `name` is the name of the new data source (subset of data from a data store)

ii. `sourceParameters` is an object containing the parameters required to create a source on its particular kind of connection, connector, and data store.

The key-value pairs required for the `sourceParameters` object are defined by the `sourceParameters` array included in the `connectors` object returned by making a GET call to `/api/connectors`. In the sourceParameters array, each object defines a key-value pair for the `sourceParameters` object required to create a source. For example, the `sourceParameters` definition in this array defines a `sourceParameters` object that requires an `index`, a `mapping`, and a `patternType`:

```
"sourceParameters": [
   {"name": "index",
    "description": "Comma-separated list of index names or patterns",
    "required": true },
   {"name": "mapping",
    "description": "Comma-separated mapping types list",
    "required": false },
   {"name": "patternType",
    "description": "Type of index pattern. Can either be dynamic or time-based",
    "required": false }
]
```

That definition would be satisfied by the `sourceParameters` object below, which could be used to create a source using the `/api/connections/{connectionId}/sources` API.

```
"sourceParameters": {
       "index": "the-requests",
       "mapping": "",
       "patternType": ""
}
```

Materialized Views API (Experimental)

Note: Materialized view functionality is disabled by default. To enable, contact technical support for assistance. The Materialized Views API is deprecated and will be removed in a future release.

Materialized views allow you to use pre-aggregated query results, stored in some external storage, to speed up query processing in certain scenarios, especially when processing a query with heavily aggregated data.

Important: Pre-aggregated data is not managed by Composer, so it should be maintained manually and kept up to date by the owner of the data.

Important: This is an experimental feature.

Note: The Volume metric is required in all materialized view definitions.

API support for materialized views is performed using the REST API endpoint `/api/materialized-views`, as described below.

Endpoint	Method	Description
`/api/materialized-views`	GET	Returns a list of all materialized view mappings. Optionally, include the `sourceId={<data-source-id>}` parameter to obtain a list of materialized views for a specific data source.
`/api/materialized-views`	POST	Creates a new materialized view mapping.
`/api/materialized-views/<source-ID>`	PUT	Updates a specific materialized view mapping.
`/api/materialized-views/<source-ID>`	PATCH	Patches (partially updates) a specific materialized view mapping.
`/api/materialized-views/<source-ID>`	DELETE	Deletes a specific materialized view mapping.
`/api/sources/<source-ID>/fields/meta`	GET	Returns information about all the fields and metrics in a data source.

API documentation is provided with your Composer installation at this link: *`https://<composer-URL>/composer/swagger-ui.html`*.

Here is a sample of the general object structure for materialized views:

```
{
  "name": "string",
  "description": "string",
  "enabled": true,
  "definition": { ... },
  "storageSettings": {
    "targetStorage": { ... },
    "fieldMappings": [ ... ]
  }
}
```

Each object is described in the following table.

Object	Specifies
`name`	The name of the materialized view definition.
`description`	The description of the materialized view definition.
`enabled`	Whether or not the definition is enabled. A value of `true` indicates that it is enabled; `false` indicates that it is not.
`definition`	The query that this materialized view should match.
`storageSettings`	Where the data for the query results are stored.
`targetStorage`	The connection and the specific table or collection of data.
`fieldMappings`	The mapping between the fields used in the definition and the columns or fields in the data identified by the `targetStorage` object. Every field returned by the `definition` query should be mapped in `fieldMappings`. You can skip fields in the target table if they are not used in the query results (for example, redundant metrics).

Configure Data Source Refresh Rates Using the API

You can modify source refresh rates using Composer's REST APIs. Only live data sources have refresh rates. Consequently, you can only modify the refresh rate of a live data source. For more information about live refresh rates or configuring them using the Composer client application, see Configuring the Refresh Rate Settings for Live Data Sources . For more information about global default settings, including refresh rates, see Global Default Settings (v2.4) .

Identify Current Refresh Rates

You can identify whether a data source is live and, if so, what its current refresh rate is using `curl` or another HTTP utility or library to make the following call:

```
   curl --user <name>:<password> -X GET 'https://<yourserver_path>/api/sources/<123456789>
' -H "Content-Type: application/vnd.composer.v3+json;"
```

in which:

- `<name>` and `<password>` are the name and password with authorization to request information about the source
- `<yourserver_path>` is the DNS and path for your Composer server
- `<123456789>` is replaced with the sourceID that contains the source that you wish to inspect

The source configuration object returned by this method includes the follow keys:

```
"live": true
"liveRefreshRate": 1
"name": "SOURCE_NAME",
"sourceParameters": {
   "PARAM": "PARAMVAL"
   },
```

The `live` key indicates whether the data source is live (also known as streaming or real-time).

The `liveRefreshRate` key indicates the refresh frequency of the data source. Note that this frequency has no predefined units. The units are supplied by the granularity of the source's data. For example, if granularity is *hour*, then a refresh rate of 3 indicates a refresh every three hours.

Make note of the `name` and `sourceParameters` keys because they must be included in the PUT method used to modify the refresh rate.

Modify the Source Refresh Rate

You can modify a live data source's refresh rate using cURL or another HTTP utility or library to make the following call:

```
curl --user <name>:<password> -X PUT -d '{"liveRefreshRate": <ratenum>, "name": "<NAME>", "sourceParameters": {"<PARAM>":
"<PARAMVAL>"}}'
'https://<yourserver_path>/api/sources/<123456789>' -H "Content-Type: application/vnd.composer.v3+json;"
```

in which:

- `<name>` and `<password>` are the name and password with authorization to modify the source
- `<ratenum>` is replaced with the refresh rate that you wish to use for the source
- `<yourserver_path>` is the DNS and path for your Composer server
- `<123456789>` is replaced with the sourceID that contains the source that you wish to inspect
- `<name>` is replaced with the name of the data source as discovered in the GET method detailed above
- `<PARAM>` and `<PARAMVAL>` are replaced with the actual values of `<sourceParameters>` as discovered in the GET method detailed above

The method returns the updated source object or an error message if the operation fails.

Admin-Defined Functions

Composer provides a set of functions that you can use in expressions to build derived fields or custom metrics. However, you may want to use your own functions (for which there is no Composer equivalent) to extend your data analytics in Composer. Examples of this might be functions available within your data store either natively or as user-defined functions. In such situations, you can leverage Composer administrator-defined functions to expose this capability throughout Composer.

Administrator-defined functions allow your organization to create functions at the connector level from SQL strings. These functions can be referenced later in data source configurations that use the connector.

Currently, the following limitations exist for administrator-defined functions.

1. They are only available for connectors that are SQL-based and that support row-level expressions in derived fields and custom metrics.
2. They are only available for functions that perform row-level operations, and not aggregation operations (for example, standard deviation or rank).

Support for this feature by connector is shown in the following table.

Key:Y - Supported; **N** - Not Supported; N/A - not applicable

Connector	Supported?
Amazon Redshift	**Y**
Amazon S3	**Y**
Apache Drill	**Y**
Apache Phoenix	**Y**
Apache Phoenix Query Server (QS)	**Y**
Apache Solr	**N**
BigQuery	**Y**
Business Central Jet	**Y**
Cloudera Impala	**Y**
Cloudera Search	**N**
Couchbase	**Y**
Dremio	**N**
Elasticsearch 7.0	**N**
Elasticsearch 8.0	**N**
File Upload	**Y**

Connector	Supported?
HDFS	**Y**
Hive	**Y**
Jira	**N**
MemSQL	**Y**
Microsoft SQL Server	**Y**
MongoDB	**N**
MySQL	**Y**
Oracle	**Y**
PostgreSQL	**Y**
Python	**N**
Real Time Sales	N/A
Salesforce	**N**
SAP Hana	**Y**
SAP S/4HANA	**Y**
SAP IQ	**Y**
Spark SQL	**Y**
Snowflake	**Y**
Teradata	**Y**
TIBCO DV	**Y**
Trino	**Y**
File Upload (Upload API)	**Y**
Vertica	**Y**

If a connector supports derived fields, it also supports row-level expressions.

For more information, see:

- Activate Admin-Defined Functions
- Admin-Defined Function JSON Files

Activate Admin-Defined Functions

Activate an admin-defined function in Composer

1. Create a JSON file containing your admin-defined function definitions. This JSON file should reside on the same machine as the connector server. The recommended location for this JSON file is `/etc/zoomdata` (for example, `/etc/zoomdata/edc-impala-functions.json`). See Admin-Defined Function JSON Files for more information about creating this JSON file.

2. Update the connector server configuration so the connector knows where to find the file containing the admin-defined functions you have created. Complete these steps:

   a. Locate and edit the properties file for the connector in the `/etc/zoomdata` directory. For example, the Impala properties file is called `edc-impala.properties`.

   ```
   vi <property-file>
   ```

   b. Locate and update the `functions.template.json.path` property in the Functions section of the properties file. Supply the path to the JSON file containing your admin-defined functions in the property. In the following example, the path for the `edc-impala-functions.json` file is specified.

   ```
   functions.template.json.path=/etc/zoomdata/edc-impala-functions.json
   ```

   All admin-defined functions for a connector should be defined in the same JSON file, so only one path is needed per data source.

   c. Save the properties file.

3. Restart the connector server.

The connector reads and validates the JSON file only at startup. When you restart the connector, it loads the admin-defined function JSON file and will validate the function definitions in the file. Errors are logged if they are found. If the function definitions are valid, the connector starts successfully and writes a list of the loaded admin-defined functions to the log.

Admin-Defined Function JSON Files

Use a single JSON file to define all the admin-defined functions for a single connector. Store this JSON file in the appropriate location for your Composer environment.

- Linux: `/etc/zoomdata`. As an example, `/etc/zoomdata/edc-impala-functions.json`.
- Windows: `<install-path>/conf-modify`. As an example, `<install-path>/conf-modify/edc-impala-functions.json`.

This topic covers the following information:

JSON File Structure

The basic structure of an admin-defined function in the JSON file is shown below. Each section is described.

```
{
  "<function>": {
    "template": "<SQL template string>",
    "returnType": {
      "type": "<type>",
      "name": "<name>"
       },
       "arguments": {
      "<arg1-key>": {
         "name": "<arg1-name>",
         "returnType": {
           "type": "<type>",
           "name": "<name>"
         },
         "description": "<arg1-description>"
      },
      "<arg2-key>": {
         "name": "<arg2-name>",
         "returnType": {
```

```
            "type": "<type>",
            "name": "<name>"
          },
          "description": "<arg2-description>"
       }
        },
    "description": "<function-description>"
  }
}
```

function

The function name is the identifier used for the row-level admin-defined function. This name is displayed in the Composer UI. The function name must start with a letter or an underscore and can contain letters, underscores, numbers, and periods.

template

The template defines the SQL string that will be used in the SQL query to a data source. You are fully responsible for the validity of the template string. Composer cannot fully validate it.

The template references arguments defined later in the JSON file. The arguments are surrounded by braces (curly braces) and are replaced by their SQL representations at run time. Composer does validate missing arguments when the connector associated with this JSON file starts.

You can use a backslash (`\`) as an escape character. It can be used to escape the following three characters: `\ { }` anywhere in the SQL template string (including inside the argument placeholders).

Here is an example:

```
"template": "({arg_0}) + ({arg_1}) * interval '1 year'"
```

returnType

There are three possible returnTypes: `simple`, `generic`, and `array`. The returnType used for the whole function must be `simple` or `generic`; the `array` returnType is not supported for the whole function. However, all three returnTypes are supported in the arguments defined within an admin-defined function.

Each returnType in the JSON provides a type specification and a name.

simple returnTypes

Five simple returnTypes are supported : NUMBER, INTEGER, STRING, DATE, or BOOLEAN. Here is an example of a simple returnType:

```
"returnType": {
    "type": "simple",
    "name": "DATE"
}
```

generic returnTypes

A generic returnType is an abstract type that is given a name. Here is an example of a generic returnType:

```
"returnType": {
    "type": "generic",
    "name": "T"
}
```

The specified name is the name used for the returnType inference. The only restriction is that the function's generic returnType name must be the same as one of the argument types. This is required to infer a function returnType based on an argument type.

array returnTypes

An array returnType includes a baseType that can be used only for the last argument. The baseType can only be `simple` or `generic`. Here is an example:

```
"returnType": {
    "type": "array",
    "baseType": {
        "type": "simple",
        "name": "STRING"
    }
}
```

arguments

The arguments section defines the argument keys and definitions. You do not have to specify an arguments section if there are no arguments. The argument keys are used in the SQL template string at the beginning of the function definition.

An argument definition consists of a name, a returnType, and description. The name and description are used to display the argument in the Composer UI. The returnType is used for argument type validation and function returnType inference. The argument returnType can be `simple`, `generic` or `array`.

description

The description is an optional string you can specify to describe a function or an argument in the JSON file and in the Composer UI.

Validating Your JSON File

You can validate your JSON against a JSON schema file `functions-schema.json` that Composer provides in the appropriate directory for your environment.

- Linux: `/opt/zoomdata/docs/<edc-connector>`. For example, the Impala JSON schema is in `/opt/zoomdata/docs/edc-impala/functions-schema.json`.
- Windows: `<install-path>/docs/<edc-connector>`. For example, the Impala JSON schema is in `<install-path>/docs/edc-impala/functions-schema.json`.

Validate your JSON file against the JSON schema

1. Link to the JSON Schema Validator at *https://www.jsonschemavalidator.net/*.
2. Copy the schema found in the JSON schema file for your connector (see above) to the left side of the JSON Schema Validator screen.
3. Copy your JSON file admin-defined function definitions in the **Input JSON** section on the right side of the JSON Schema Validator screen.

Any schema errors will be identified immediately on the screen.

Example

In the following JSON file, two admin-defined functions, `TEST_ADD` and `TEST_ADD_YEAR`, are defined.

Note: Before using this example, be sure to remove the comments. Comments are not allowed in JSON.

```
{

  "TEST_ADD": { //Function name
    "template": "{summand_1} + {summand_2}", //SQL template string
    "returnType": { //Return type of the function
         "type": "simple",
         "name": "NUMBER"
    },
    "arguments": { // List of arguments
         "summand_1": { //Argument key used for lookup in SQL template string
           "name": "Summand 1", //Argument name, will be displayed on Derived Field Editor
```

```
          "returnType": { // Type of argument.
            "type": "simple",
            "name": "NUMBER"
        },
           "description": "An expression evaluated to a numeric value." //Argument description
         },
         "summand_2": {
        "name": "Summand 2",
        "returnType": {
          "type": "simple",
          "name": "NUMBER"
        },
        "description": "An expression evaluated to a numeric value."
      }
    },
       "description": "Addition: add two numbers." //Function description
  },
  "TEST_ADD_YEAR": {
    "template": "({0}) + ({1}) * interval '1 year'",
    "returnType": {
      "type": "simple",
      "name": "DATE"
       },
       "arguments": {
      "0": {
         "name": "Date",
         "returnType": {
           "type": "simple",
           "name": "DATE"
         },
         "description": "An expression 1"
      },
      "1": {
         "name": "Year",
         "returnType": {
           "type": "simple",
           "name": "INTEGER"
         },
         "description": "An expression 2"
      }
       },
    "description": "Add year."
  }
}
```

Manage Custom Charts

Administrators can create custom charts using the custom chart CLI. They can also download, import, and delete custom charts in the Composer user interface (UI). Custom charts can also be made visible on various menus in the Composer user interface (UI). By default, custom charts (visual types) are not visible until enabled for a source. See Available Visual Types.

Note: You must be an administrator to manage custom visual types.

- Maintain Custom Charts Using The Custom Chart CLI
- Manage Custom Charts In The UI
- Visual Type Configuration Properties
- Custom Chart API

A step-by-step tutorial for creating a custom chart is also included in A Custom Chart Tutorial.

Maintain Custom Charts Using the Custom Chart CLI

Composer's Custom Chart Command Line Interface (CLI) allows developers to create, manage, and delete custom charts without being connected to the client application. The CLI tool uses `Node.js` and is installed locally via `npm`. Details about the custom chart CLI can also be found in https://www.npmjs.com/package/composer-chart-cli.

Note: Before installing Composer's custom chart CLI, verify that `Node.js` version 10 or later and `npm` version 5.6 or later are installed on your machine.

After it is installed and configured, the custom chart CLI behaves much like any other command line tool. You can build projects, run tests, and push new visuals using `npm` scripts. The custom chart CLI also supports the configuration of multimetric and group variables.

For a full list of common commands, see Commands For The Custom Chart CLI. A step-by-step tutorial for creating a custom chart can be found in A Custom Chart Tutorial.

This section covers the following topics:

- Supported Custom Chart CLI Versions
- Install And Configure The Custom Chart CLI
- Create A Custom Chart Using The CLI
- Install Dependencies And Build The Custom Chart Using The CLI
- Edit The Custom Chart Using The CLI
- Push The Custom Chart To The Server Using The CLI
- Remove A Custom Chart From The Server Using The CLI
- List All Custom Charts On The Server Using The CLI
- Import A Custom Chart Using The CLI
- Custom Chart Support For Cross-Visual Links And Filters

You can also perform some maintenance tasks for custom charts using the Composer UI. See Manage Custom Charts In The UI.

Supported Custom Chart CLI Versions

Custom charts in Composer can be created and managed using the Composer Custom Chart CLI tool.

- Composer Custom Chart CLI version 1 (`composer-chart-cli`) supports Composer 7.10 and later.
- CLI version 6 (`zoomdata-chart-cli`) supports Composer 7.10 and earlier.
- CLI version 5 (`zoomdata-chart-cli`) supports Composer 7.9 and earlier.

Note: Composer Custom Chart CLI version 1 only supports the use of `vnd.composer.v3+json` as the `Content-Type` for API routes.

A custom chart is bundled before it is pushed to the Composer server. Bundling can be accomplished using a tool such as webpack. This creates a `/dist` directory containing the necessary files required to send the visual to the Composer server.

Existing custom charts from older CLI versions (5 and 6) use the same directory structure as the Composer Custom Chart CLI version 1 directory structure.

Migrate Custom Charts

Custom charts in Composer can be created and managed using the Composer Custom Chart CLI tool.

- Composer Custom Chart CLI version 1 (`composer-chart-cli`) supports Composer 7.10 and later.
- CLI version 6 (`zoomdata-chart-cli`) supports Composer 7.10 and earlier.
- CLI version 5 (`zoomdata-chart-cli`) supports Composer 7.9 and earlier.

Install Composer Custom Chart CLI version 1 globally. Existing custom charts from older CLI versions (5 and 6) use the same directory structure as the Composer Custom Chart CLI version 1 directory structure.

The following table lists the version of the CLI compatible with different versions of Composer.

CLI Version	Supported Versions
1	Composer 7.10 and later. (`composer-chart-cli`)
6	Composer 7.9 and later. (`zoomdata-chart-cli`)
5	Composer 6.9 and through Composer 7.8 and earlier. (`zoomdata-chart-cli`)

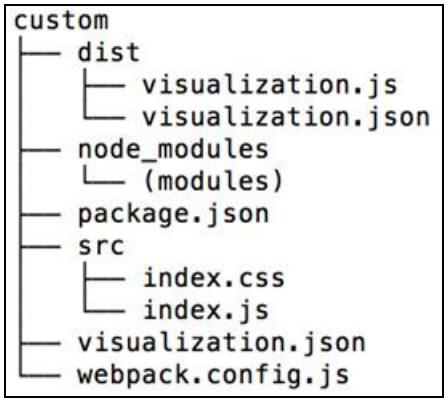
Note: Composer v7.10 and later works with the new version of the Composer CLI tool, version 1 (`composer-chart-cli`). This new version of the CLI tool enables you to manage custom charts in your Composer v7.10 and later environment. Familiar commands and locations have changed: commands that use `zd` now use `cmp`, and that use `zoomdata` now use `composer`.

File Structure

The custom chart file structure for supported versions of custom charts:

- Composer Custom Chart CLI v1 (`composer-chart-cli`)
- CLI v6 (`zoomdata-chart-cli`)
- CLI v5 (`zoomdata-chart-cli`)

Custom Chart CLI File Structure



API Route Differences

Note: Composer Custom Chart CLI version 1 only supports the use of `vnd.composer.v3+json` as the `Content-Type` for API routes.

CLI Command DifferencesCLI Commands

The command differences between CLI versions are described in the following table:

Command v1	Command v5/v6	Description
`cmp-chart import <zip-file-path>`	`zd-chart import <zip-file-path>`	Import a visual in a zip file.
`cmp-chart init`	`zd-chart init`	Create a new visual in a folder you specify.
`cmp-chart edit`	`zd-chart edit`	Update the visual locally, but no longer push the updated visual to the server.
`cmp-chart push`	`zd-chart push`	Push a custom chart to the server.
`cmp-chart watch`	`zd-chart watch`	Watch changes in the `src` directory and pushes them to the server.

If you need to migrate custom charts from an unsupported version of Composer, contact Technical Support for assistance.

Install and Configure the Custom Chart CLI

Note: Before installing Composer's custom chart CLI, verify that `Node.js` version 10 or later and `npm` version 5.6 or later are installed on your machine.

Install and configure the custom chart CLI:

1. Install the custom chart CLI by running the following:

```
$ npm install composer-chart-cli@<x> -g
```

where `<x>` is `6`, the version of the custom chart CLI you are installing.

2. After installation, you can configure the default environment that is used by the custom chart CLI. This makes creating and pushing custom charts easier, as you do not have to provide the server URL and credentials with every command. Run the following to start the configuration process:

```
$ cmp-chart config
```

3. Follow the prompts to store your default server configuration in an encrypted file. After the server configuration has been saved, the CLI checks for the presence of this file when you omit the server URL and credentials when running commands.
4. After installing, you can create new visuals or make updates to current custom charts.

Note: Composer v7.10 and later works with the new version of the Composer CLI tool, version 1 (`composer-chart-cli`). This new version of the CLI tool enables you to manage custom charts in your Composer v7.10 and later environment. Familiar commands and locations have changed: commands that use `zd` now use `cmp`, and that use `zoomdata` now use `composer`.

Create a Custom Chart Using the CLI

A complete example of creating a custom chart using the custom chart CLI can be found in A Custom Chart Tutorial.

Note: Composer Custom Chart CLI version 1 only supports the use of `vnd.composer.v3+json` as the `Content-Type` for API routes.

Create a custom chart locally using the custom chart CLI

1. Open the Composer CLI.
2. Run the following command to create the visual.

```
$ cmp-chart init <path-to-visual>
```

Optionally, you can provide a type parameter. Valid values are `single-group`, `multi-group`, or `raw`. The default is `single-group`. For example:

```
$ cmp-chart init -t multi-group <path-to-visual>
```

The `cmp-chart init` command creates a directory in the specified path containing the files you need to get started. Here is a preview of the directory tree:

The following table describes the function of each file in the tree:

File	Description
`package.json`	Lists the packages your visual depends on. For more information, see https://docs.npmjs.com/creating-a-package-json-file.
`src/index.css`	Your visual's CSS (style sheet) code.

File	Description
`src/index.js`	Your visual's JavaScript code. Additional files can be used and imported into this file.
`visualization.json`	Contains the name, controls, and variables of your visual.
`webpack.config.js`	The webpack configuration. Webpack is used in visuals to bundle your code into a single file.

Install Dependencies and Build the Custom Chart Using the CLI

To install dependencies and build the custom chart using the custom chart CLI:

1. Run the following command at the root level of the newly created custom chart to install dependencies:

```
npm install
```

2. Run the following command at the root level of the custom chart to build the chart:

```
$ npm run build
```

Note: Custom chart CLI version 5 only supports the use of `vnd.composer.v2+json` as the `Content-Type` for API routes. CLI version 4 only supports the use of `vnd.zoomdata.v2+json`.

Edit the Custom Chart Using the CLI

Edit a custom chart using the custom chart CLI

Make sure you are in the visual's folder and run the following edit command:

```
$ cmp-chart edit
```

Alternatively, you can specify the path to the visual in the edit command. For example:

```
$ cmp-chart edit -d <path-to-visual>
```

After running the edit command, follow the prompts to update the visual's controls, name, variables, or visibility.

When all edits are finished, be sure to rebuild the visual before pushing it to the server. See Install Dependencies And Build The Custom Chart Using The CLI.

Note: Composer Custom Chart CLI version 1 only supports the use of `vnd.composer.v3+json` as the `Content-Type` for API routes.

Push the Custom Chart to the Server Using the CLI

Push the custom chart to the server using the Custom Chart CLI

Run the following command:

```
$ cmp-chart push
```

Note: Composer Custom Chart CLI version 1 only supports the use of `vnd.composer.v3+json` as the `Content-Type` for API routes.

List All Custom Charts on the Server Using the CLI

List all custom charts stored on the Composer server using the Custom Chart CLI

Run the following command:

```
$ cmp-chart ls
```

Note: Composer Custom Chart CLI version 1 only supports the use of `vnd.composer.v3+json` as the `Content-Type` for API routes.

Import a Custom Chart Using the CLI

Import a custom chart using the custom chart CLI

Run the `cmp-chart import` command, specifying the path to the library. You can import visual zip files that were downloaded from the Manage Custom Charts page or a custom chart created with an older version of the CLI.

```
$ cmp-chart import [options] <visualname> <path-to-visual>
```

where `<path-to-visual>` is the relative path to the zip file containing the visual you want to import.

The following options are available:

- `-a, -app [URL]` Specify the Composer application server URL (e.g. https://myserver/composer)
- `-u, -user [user:password]` Specify the user name and password to use for server authentication.
- `-h, -help` Output usage information

Note: Composer Custom Chart CLI version 1 only supports the use of `vnd.composer.v3+json` as the `Content-Type` for API routes.

A custom chart is bundled before it is pushed to the Composer server. This creates a `/dist` directory containing the necessary files required to send the visual to the Composer server. This `/dist` custom chart zip file is not compatible with the `cmp-chart import` command and cannot be imported using the Manage Custom Charts page in the UI. To import the `/dist` custom chart zip file, use the `cmp-chart push` command instead.

Remove a Custom Chart from the Server Using the CLI

Remove a custom chart from the server using the Custom Chart CLI

Run the following command:

```
$ cmp-chart rm
```

After running the command you will be prompted to select the visual name and confirm its deletion.

Note: Composer Custom Chart CLI version 1 only supports the use of `vnd.composer.v3+json` as the `Content-Type` for API routes.

Commands for the Custom Chart CLI

Access a list of custom chart CLI commands at any time by running `cmp-chart`. For help on any command, simply enter its name or enter `cmp-chart help`. For example, entering `cmp-chart init` without any parameters will show the help for the `cmp-chart init` command.

Note: Composer Custom Chart CLI version 1 only supports the use of `vnd.composer.v3+json` as the `Content-Type` for API routes.

Common commands

Command	Usage
config	Sets up an encrypted configuration of the servers URL and credentials.
edit	Edits the visual's controls, components, libraries, variables, etc. Updates are not automatically pushed to the serve.
help	Provides help for the custom chart CLI.
import	Imports a visual in a zip file.
init	Creates a new visual in a folder you specify.
ls	Lists the custom charts.
push	Pushes the bundled format of the visual to the server.
rm	Removes a custom chart or library from the server.
watch	Watches the changes in custom chart files and updates them.

Syntax

cmp-chart config

```
$ cmp-chart config <options>
```

Defines an encrypted configuration of Composer's server URL and admin credentials.

Parameters

- server (ex. http://myserver.com:8080/composer)
- user name

- password

Options

- -h, --help output usage information

cmp-chart edit

```
$ cmp-chart edit <visualname>
```

Edits a Composer custom chart.

You can edit the following elements:

- Components
- Controls
- Libraries
- Name
- Visibility

cmp-chart help

Provides help for the custom chart CLI. Alternatively, you get help for a specific command:

```
$ cmp-chart help <command>
```

For example:

```
$ cmp-chart help import
```

cmp-chart import

```
$ cmp-chart import <options> <visualname> <filepath>
```

Imports a custom chart to the Composer server.

Options

- -a, --app [URL] Specify the Composer application server URL (e.g. https://myserver/composer)
- -u, --user [user:password] Specify the user name and password to use for server authentication.
- -h, --help output usage information

cmp-chart init

```
$ cmp-chart init <options> <path-to-visual>
```

Creates a custom chart.

Options

- -t, --type <type> Specify the visual type. Valid values are single-group, multi-group, or raw. The default is single-group.
- -h, --help output usage information

cmp-chart ls

```
$ cmp-chart ls <options>
```

Lists the custom charts you have created.

- -a, --app [URL] Specify the Composer application server URL (e.g. https://myserver/composer)
- -u, --user [user:password] Specify the user name and password to use for server authentication.

- -h, --help output usage information

cmp-chart push

```
$ cmp-chart push [options]
```

Pushes a custom chart in its current state to the Composer server.

Options

- -a, --app [URL] Specify the Composer application server URL (e.g. https://myserver/composer)
- -d, --dir [path/to/source/files] Directory used to look for the visual to push.
- -u, --user [user:password] Specify the user name and password to use for server authentication.
- -h, --help output usage information

cmp-chart rm

```
$ cmp-chart rm [options]
```

Removes a custom chart from the Composer server.

Options

- -a, --app [URL] Specify the Composer application server URL (e.g. https://myserver/composer)
- -u, --user [user:password] Specify the user name and password to use for server authentication.
- -h, --help output usage information

cmp-chart watch

```
$ cmp-chart watch [options]
```

Watches for changes in custom chart files and updates them in the Composer server.

Options

- -a, --app [URL] Specify the Composer application server URL (e.g. https://myserver/composer)
- -d, --dir [path/to/source/files] Directory used to look for the visual to watch.
- -u, --user [user:password] Specify the user name and password to use for server authentication.
- -h, --help output usage information

Manage Custom Charts in the UI

This section describes how you can manage custom charts in the Composer UI.

Note: You must be an administrator to manage custom visual types.

- List Custom Charts
- Delete A Custom Chart
- Download A Custom Chart
- Import A Custom Chart Using The UI

You can also maintain custom charts using the CLI. See Maintain Custom Charts Using The Custom Chart CLI.

Note: Composer v7.10 and later works with the new version of the Composer CLI tool, version 1 (`composer-chart-cli`). This new version of the CLI tool enables you to manage custom charts in your Composer v7.10 and later environment. Familiar commands and locations have changed: commands that use `zd` now use `cmp`, and that use `zoomdata` now use `composer`.

List Custom Charts

Custom charts that have been imported into Composer are listed on the Manage Custom Charts page in the UI.

Access the Manage Custom Charts page

1. Log into the UI as an administrator.
2. Select **Custom Charts** from UI menu. The Manage Custom Charts page appears.

Your custom charts are listed in alphabetical order in the table on the Manage Custom Charts page.

> **Note:** Composer v7.10 and later works with the new version of the Composer CLI tool, version 1 (`composer-chart-cli`). This new version of the CLI tool enables you to manage custom charts in your Composer v7.10 and later environment. Familiar commands and locations have changed: commands that use `zd` now use `cmp`, and that use `zoomdata` now use `composer`.

Download a Custom Chart

Download a custom chart

1. Log into the UI as an administrator.
2. Access the Manage Custom Charts page. See List Custom Charts.
3. Locate the visual in the table of visuals.
4. Select for the visual. The `.zip` file for the visual is downloaded.

Note: When you download the visual from the Manage Custom Charts page, the zip file includes a file called `version` that contains the version of the Composer server. If you later try to import this zip file to the Composer server using a different version an error will occur indicating that the versions do not match. An easy workaround is to modify the version in the `version` file before trying the import.

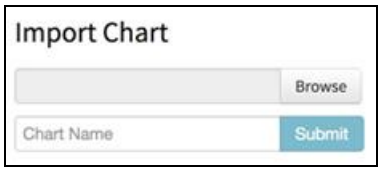
Note: Composer v7.10 and later works with the new version of the Composer CLI tool, version 1 (`composer-chart-cli`). This new version of the CLI tool enables you to manage custom charts in your Composer v7.10 and later environment. Familiar commands and locations have changed: commands that use `zd` now use `cmp`, and that use `zoomdata` now use `composer`.

Import a Custom Chart Using the UI

You can import a custom chart using the Composer UI as well as using the Composer CLI. This topic describes how to import a custom chart using the UI. See Maintain Custom Charts Using The Custom Chart CLI for information about using the CLI to import a custom chart.

Import a custom chart using the Composer UI

1. Log into the UI as an administrator.
2. Access the Manage Custom Charts page. See List Custom Charts.
3. Locate the Import Chart area of the Manage Custom Charts page.



4. Select **Browse** and locate and select the `.zip` file of the custom chart you want to import.
5. Enter a visual name for the imported visual.
6. Select **Submit**. The visual is imported.

> **Note:** Composer v7.10 and later works with the new version of the Composer CLI tool, version 1 (`composer-chart-cli`). This new version of the CLI tool enables you to manage custom charts in your Composer v7.10 and later environment. Familiar commands and locations have changed: commands that use `zd` now use `cmp`, and that use `zoomdata` now use `composer`.

Delete a Custom Chart

Delete a custom chart

1. Log into the UI as an administrator.
2. Access the Manage Custom Charts page. See List Custom Charts.
3. Locate the visual in the table of visuals.
4. Select  for the visual in the Delete column of the table.

Note: Composer v7.10 and later works with the new version of the Composer CLI tool, version 1 (`composer-chart-cli`). This new version of the CLI tool enables you to manage custom charts in your Composer v7.10 and later environment. Familiar commands and locations have changed: commands that use `zd` now use `cmp`, and that use `zoomdata` now use `composer`.

A Custom Chart Tutorial

As an administrator, you can create custom charts (custom visuals) using the custom chart CLI. Download, import, and delete custom charts in the Composer user interface (UI). Make custom charts visible on various menus in the Composer user interface (UI).

This tutorial guides you through the process of developing a new custom chart from scratch. If you have some experience with JavaScript and you want to learn about connecting your data to charts, you are in the right place, we'll walk you through the rest.

Note: You must be an administrator to manage custom visual types.

To complete the custom chart tutorial, complete the following tutorial parts, in order:

- Part 1: Custom Chart Basics
- Part 2: Query Variables, Chart Defaults, Data Preview, And Data Accessors
- Part 3: Third-Party Charting Library Integration
- Part 4: Custom Chart Controls

Part 1: Custom Chart Basics

This part of the custom chart tutorial introduces you to the Custom Chart CLI, how to create a chart with sample code, how to edit the chart's code, and how to preview the chart. The following steps are performed in Part 1:

Step 1. Check Your Development Environment

Verify that you have everything set up to start creating custom charts with Composer.

- A valid version of `node.js` must be installed. `Node.js` is a programming tool for running JavaScript on servers and in your computer's terminal. The Composer custom chart CLI is built using `node.js`.
- A valid version of `npm` must be installed. It is usually installed with `node.js`.
- Composer v22 or later must be installed.

Verify that you have valid versions of node.js and npm installed

1. Open a terminal window on your computer.
2. Enter `node --version` in the terminal window. The version of `node.js` installed on your computer is shown in the window. The minimum supported version of `node.js` by the Composer custom chart CLI is version 10.
3. Enter `npm --version` in the terminal window. The version of npm installed on your computer is shown in the window. The minimum supported version of `npm` supported by the Composer custom chart CLI (`composer-chart-cli`)for Composer v7.10 and later is version 1.

If `node.js` and npm are not installed, go to *https://nodejs.org/* and install the recommended version for your operating system.

Step 2. Install & Configure the Composer Custom Chart CLI

Composer uses the custom chart CLI to build, edit, and remove custom charts.

Install and configure the CLI

1. Enter the following command in the terminal window.

```
npm install composer-chart-cli@1 -g
```

2. After the CLI is installed, enter the following command in the terminal window:

```
cmp-chart config
```

3. The `config` command prompts you to supply the following information:

i. Your Composer server URL.

ii. The user name to use for authentication (typically: `admin`).

> **Note:** You must be an administrator to manage custom visual types.

iii. The password for the user name.

4. After entering this information, a prompt asks whether the configuration content should be stored in an encrypted file residing in your home directory (`~/.config/composer/cmp-chart.pref`). Enter **y** to finish configuring your environment.

5. Verify that everything is set up correctly by listing the custom charts available on the configured server using the following command:

```
cmp-chart ls
```

> **Note:** If your Composer instance does not have any custom charts defined, the output of this command is an empty list. If your instance does have custom charts defined, verify that they are correctly listed in the terminal window.

Step 3. Create a Custom Chart with Sample Code

You can specify one of three chart types to get you started:

- Single Group
- Multigroup
- Raw

This tutorial uses the default Single Group template – a chart skeleton with minimal code designed to make you familiar with the basics of the custom chart API.

Complete the following steps to create a single-group custom chart:

1. Enter the following command in the terminal window:

```
cmp-chart init ./my-first-custom-chart
```

2. The `init` command creates a directory in the specified path containing the files you need to get started. Here is a preview of the directory tree:

The following table describes the function of each of the files.

File Name	Description
`package.json`	Lists the packages on which your chart depends. For more information, see *https://docs.npmjs.com/creating-a-package-json-file*.
`visualization.json`	Contains information about the name, controls, and variables of your chart.
`webpack.config.js`	Contains configuration information for webpack. Webpack is used in charts to bundle your code into a single file. For more information, see *https://webpack.js.org/configuration/*.
`src/index.js`	The main entry point to your chart's Javascript code. Additional files can be used and imported into

File Name	Description
	`index.js`.
`src/index.css`	The main entry point to your chart's CSS (style sheet) code.

3. Install the default `devDependencies` listed in the `package.json` file. This step is required before you can work with the chart's code. In a terminal window, `cd` to your chart's root directory and run the following command:

```
npm install
```

4. To use a similar naming convention as the out-of-the-box charts, update the chart's name in the `visualization.json` file to "My First Custom Chart". Save the file.

Step 4. Edit the Chart Code

Composer's custom charts are comprised of CSS and JS files that make up the chart's code. The chart CLI lets you update the chart's code in the server in one of two ways:

- You can push an updated local copy of the chart back to the server.
- You can use **watch** mode to continuously check for changes to the local component files and automatically update your Composer server's copy.

Throughout this tutorial, we continuously modify the code and preview the changes. It is best to use **watch** mode in this scenario.

Start watch mode

1. Compile a development version of the chart's code using the webpack bundler. In the terminal window from the root directory of the chart, enter the following command:

```
npm start
```

2. In a different terminal window, push the chart for the first time to the configured server:

```
cmp-chart push
```

3. Set up watch mode to continuously update the chart on the Composer server as the code changes. Enter the following command in the second terminal window:

```
cmp-chart watch
```

   The `npm start` command instructs webpack to compile the code into a single `visualization.js` file and continuously watches for changes in the `src` files to update the chart as necessary. The `cmp-chart watch` command running in a separate terminal window starts watching for changes in the `visualization.json` and `visualization.js` files to push the chart changes back to the server.

4. Open the `src/index.js` file in your preferred text editor or integrated development environment. Modify the code to output some text on the chart with the total number of records returned by the default query. Change the `controller.update` function in the `src/index.js` file to contain the code below:

```
controller.update = data => {
  // Called when new data arrives
  controller.element.innerHTML =
    'The total # of records returned by this query is: ' + data.length;
};
```

5. Return to the terminal window where **watch** mode is enabled. A message showing that the My First Custom Chart was updated should appear.

Step 5. Preview the Chart

Before you can preview the newly created chart in a dashboard in the Composer UI, you must enable it for a specific source.

1. Log into the Composer UI as an administrator or a user with the **Administer Sources** privilege.
2. Select **Sources** on the UI menu () or the top-level navigation menu, or select the **Sources** box on the Home page. The Sources page appears.
3. On the Sources page, locate a data source configuration to edit, and select the more menu () button.
4. Select **Available Visual Types**. The Available Visual Types work area for this source opens.
5. Select the chart named "My First Custom Chart" and enable (toggle on) the custom chart in this data source. A message appears, confirming "My First Custom Chart" is enabled.
6. When your changes are complete, close the Available Visual Types work area.

Now let's take a look at the new chart in a Composer dashboard.

1. Log into Composer as an administrator or a user who has been assigned to a group with the **Administer Dashboards** privilege.
2. Select **Library** on the top-level navigation banner or the UI menu, or select the **Dashboards** box on the Home page. The library displays.
3. Select **Add Dashboard**. A blank dashboard appears showing options to add a new visual or place an existing visual.

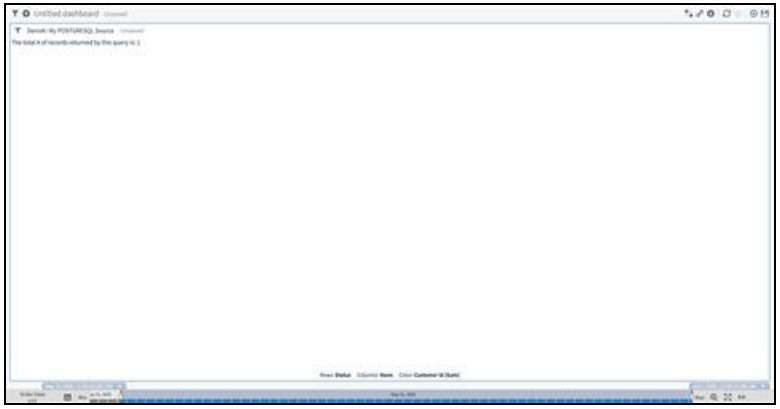
4. Select **Add Visual** to add a new visual to the dashboard. Select **Place Existing Visual** to place an existing visual on the dashboard.

i. If you select **Add Visual**, follow the procedure described in Add Visuals To A Dashboard

ii. If you select **Place Existing Visual**, follow the procedure described in Add Existing Visuals To A Dashboard.

5. On the **Step 1 of 2: Select a Source** dialog, select the data source in which you enabled your custom chart. The Select a **Visual** Type dialog appears.
6. Select your custom visual style ("My First Custom Chart") on the **Step 2 of 2: Select Visual Type** dialog. The visual is created and added to the dashboard.

Your chart should look similar to the following image:



Cool! You have completed Part 1 of the custom chart tutorial. So far, you have learned how to install and configure the Composer custom chart CLI, how to create a new chart with sample code, and how to update its code and preview the results. Continue to Part 2: Query Variables, Chart Defaults, Data Preview, And Data Accessors.

Part 2: Query Variables, Chart Defaults, Data Preview, and Data Accessors

This part of the custom chart tutorial introduces query variables, chart defaults, data preview, and data accessors. The following steps are performed in Part 2:

Step 1. Modify Query Variables

Composer's front-end client communicates with the back-end data repositories using a WebSocket API designed to translate query messages into the appropriate native query language. To generate query messages, a custom chart must define the data variables that specify the query type and required fields.

Follow these steps:

1. Using the chart you created in Part 1: Custom Chart Basics, find the list of query variables created with the default Single-Group chart template.
2. Enter the following command in the terminal window from the chart's root directory window:

```
cmp-chart edit
```

3. Use arrow keys to follow the prompts and press **Enter** to select the following options, in this sequence:
   a. `Variables`
   b. `List variables`

   The output of the terminal window should look like the following image:

```
~$./cmp-chart-cli edit
   Composer Chart CLI
? What would you like to edit?: Variables
? Select an option from the following list: List variables
Name          |Type       |Description
------------------------------------
Multi Group By|multi-group|
Color         |metric     |
? Would you like to make additional edits?: (y/N)
```

Notice the group and metric variable list in the table. A group variable indicates that the chart generates a request for grouped, or aggregated, data in the query. In this case, the chart defines a single group by field. A metric variable indicates that the chart will perform an aggregation function on a numeric field as part of the grouped data query. In this chart, a single metric is specified. The chart's default configuration defines the actual group-by and metric fields at the source level.

4. Enter **Y** for the "Would you like to make additional edits?" prompt.
5. Select the following options, in this sequence:
   a. `Variables`
   b. `Edit a variable`
   c. `Metric |metric|`
6. For the remaining prompts, enter the following information:
   a. What would you like to edit: **Name**
   b. Please enter a new name for the variable: **Size**
7. Enter **N** when prompted to make additional edits.

If you are still running in watch mode in the terminal window, (see Part 1: Custom Chart Basics), the variable change will be automatically pushed to the server. If it is not, run the following command to ensure the change is pushed to the server.

```
cmp-chart push
```

You have now defined a group query variable named **Group By** and a metric query variable named **Size** in the chart.

Step 2. Preview the Data

Preview the data retrieved for your custom chart

1. In the terminal window, change directories to the local directory of “My First Custom Chart” and enter the following command to set the custom chart CLI in watch mode, if it is not in watch mode already.

   ```
   cmp-chart watch
   ```

2. In another terminal window, run the following command to compile a development version of the chart’s code using the webpack bundler:

   ```
   npm start
   ```

   For a refresher on this topic, see Step 4. Edit The Chart Code.

3. Edit the `src/index.js` file in your preferred text editor or integrated development environment.

4. Modify the code of the `controller.update` function as follows to output the data received into the console:

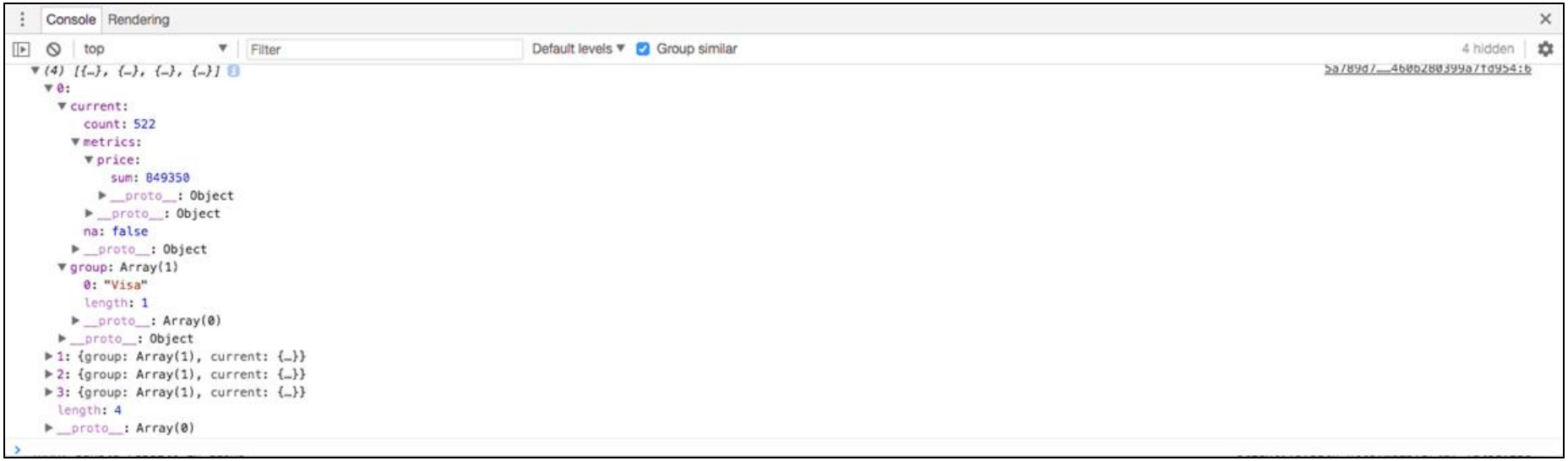
   ```
   // ...
   controller.update = data => {
     // Called when new data arrives
     console.log(data);};
   // ...
   ```

5. Remove the following code that creates the chart container and axis labels and pickers on the custom chart:

```
// Remove the code below
const chartContainer = document.createElement('div');
chartContainer.classList.add(styles.chartContainer);
controller.element.appendChild(chartContainer);

controller.createAxisLabel({
  picks: 'Group By',
  orientation: 'horizontal',
  position: 'bottom',
  popoverTitle: 'Group',
});

controller.createAxisLabel({
  picks: 'Metric',
  orientation: 'horizontal',
  position: 'bottom',
});
```

6. Save the `src/index.js` file. The CLI updates the server's copy with latest changes.

7. Preview the chart in the dashboard again (see Step 5. Preview The Chart) and the chart's data in the browser's console. Since the data is output to the browser's console, open the browser's developer tools to get to console. See instructions for Google Chrome and Mozilla Firefox browsers. The following image shows what the results might look like:



Notice the JSON array with multiple objects in the result. Each object contains a group property with an array of strings (one per group by field) and current object with a nested count property (# of records) and a metrics property (an object with the metrics requested). Each metric objects contains an aggregation function property, like sum and its resulting value.

To make flexible charts that can be used across data sources, without hard-coding, read on.

Step 3. Use Data Accessors

Now that we have captured the results of the chart's query, we need to find a way to reshape the result set in a format that can be consumed by a charting library. We also want to ensure our charts remain flexible and applicable to multiple data sources. We do not want to hard-code any object properties in the code. Composer *data accessors*, exposed via the chart API, can be used to avoid hard-coding property names.

Use data accessors

1. Create some variables to hold the data accessors. We use one per query variable.

```
const groupAccessor = controller.dataAccessors['Group By']; // Group By is the name given to variable of type group.
```

```
const metricAccessor = controller.dataAccessors.Size; // Size is the name we gave to the variable of type metric.
```

   The API is simple: `controller.dataAccessors[<variable-name>]`. Each data accessor holds the information about the field defined for the variable and provides a raw method that can be used to extract data values from a data object inside the returned data array.

2. Assume we want to reshape our data array so that the contained JSON object looks like:

```
{
  name: STRING
  value: NUMBER
}
```

   With this information, let's create a `reshapeData` function that takes the original Composer data array and returns a new array with the desired object structure, as follows:

```
function reshapeData(data) {
  return data.map(d => ({
    name: groupAccessor.raw(d),
```

```
    value: metricAccessor.raw(d),
  }));
}
```

3. Use the new `reshapeData` function to output the reshaped results into the console. The chart's code should look like this:

```
/* global controller */
import styles from './index.css';

const groupAccessor = controller.dataAccessors['Group By'];
const metricAccessor = controller.dataAccessors.Size;

controller.update = function(data) {
  // Called when new data arrives
  console.log(reshapeData(data));
};

function reshapeData(data) {
  return data.map(d => ({
    name: groupAccessor.raw(d),
    value: metricAccessor.raw(d),
  }));
}
```

4. Refresh the dashboard with a copy of the chart and check the contents of the browser console.

Well Done! You have completed Part 2 of the custom chart tutorial, where you learned about query variables and data accessors, and how to reshape data to make the chart flexible and dynamic. Continue to Part 3: Third-Party Charting Library Integration.

Part 3: Third-Party Charting Library Integration

This part of the custom chart tutorial explores integrating third-party charting libraries, such as D3, ECharts, and Highcharts, into your custom charts to build dynamic charts. It also covers how to create a chart container and how to use a library's API to render a standard chart. The following steps are performed in Part 3:

Step 1. About Third-Party Libraries

Composer makes use of third-party libraries in the out-of-the-box charts provided with the product. Typically, these libraries fall into two categories:

- Charting libraries, which assist with the rendering of standard chart types like bar, line, pie, and scatter plot charts.
- Utility libraries, which assist with shaping data sets, formatting values, or providing useful JavaScript methods.

You can use third-party libraries to assist with the rendering and formatting requirements of a custom chart. This tutorial uses ECharts to assist with the rendering of your custom chart.

Step 2. Add Libraries

Begin using ECharts with your custom chart

1. Add ECharts as a dependency of the custom chart. You can do this by installing ECharts using `npm`, which provides an easy-to-use software registry that can leveraged to install packages for your custom charts.

```
npm install echarts --save
```

2. Review the contents of the file `package.json` to ensure that Echarts is in the list of dependencies.

Step 3. Test Libraries

Test the libraries

1. In the terminal window, change directories to the local directory of “My First Custom Chart” and enter the following command to set the custom chart CLI in watch mode, if it is not in watch mode already.

```
cmp-chart watch
```

2. In another terminal window, run the following command to compile a development version of the chart’s code using the webpack bundler:

```
npm start
```

For a refresher on this topic, see Step 4. Edit The Chart Code.

3. Edit the `src/index.js` file in your preferred text editor or integrated development environment.

4. Add a `console.log` line towards the top of the file, as shown:

```
/* global controller */
import echarts from 'echarts';
import styles from './index.css';

console.log(echarts ? 'The library is defined.' : 'The library is undefined');const groupAccessor =
controller.dataAccessors['Group By'];const metricAccessor = controller.dataAccessors.Size;//...
```

5. Check the contents of the browser console after adding the updated chart to a dashboard. The following image shows what the console should look like:

6. Once you confirm that the global variable is defined, you can remove the `console.log` statement.

Step 4. Create a Chart Container

As part of its charting API, Composer provides a `<div>` element that can hold the rendered contents of your custom chart. While it is perfectly fine to use that element as the chart container, we recommend that you create a separate container inside of that element to gain full control over its styles.

Create a chart container

1. Add the following `create chart containerconst` lines of code to the top of the `src/index.js` file:

```
/* global controller */

import echarts from 'echarts';

import styles from './index.css';


// create chart container
const chartContainer = document.createElement('div');
chartContainer.style.width = '100%';
chartContainer.style.height = '100%';
chartContainer.classList.add(styles.chartContainer);
controller.element.appendChild(chartContainer);

const groupAccessor = controller.dataAccessors['Group By'];
const metricAccessor = controller.dataAccessors.Size;
//...
```

Notice the line: `controller.element.appendChild(chartContainer);`. `controller.element` is the `<div>` Composer provides with its API.

2. Modify the background color of the newly added container. Open the src/index.css component file, clear its contents, and add the following lines to the top of the file:

```
.chartContainer {
  background-color: #323232;
}
```

3. Refresh the dashboard with the custom chart, and you should see a dark gray background for your custom chart.

4. Change the background color to background-color: `#ffffff;` to set it back to white.

Step 5. Render the Chart

InPart 2: Query Variables, Chart Defaults, Data Preview, And Data Accessors, we defined a group-by variable and a metric variable to generate a grouped data set with a single metric value. This data structure works well with bar and pie charts. This step uses the ECharts API and its chart configuration option to create a pie chart with our data set.

1. In the `src/index.js` file, create a variable to hold an instance of the pie chart and a second variable to hold the chart configuration option. The chart's instance is created using the `echarts.init` method and passing in a chart container. Add it right below the data accessors you created in Part 2:

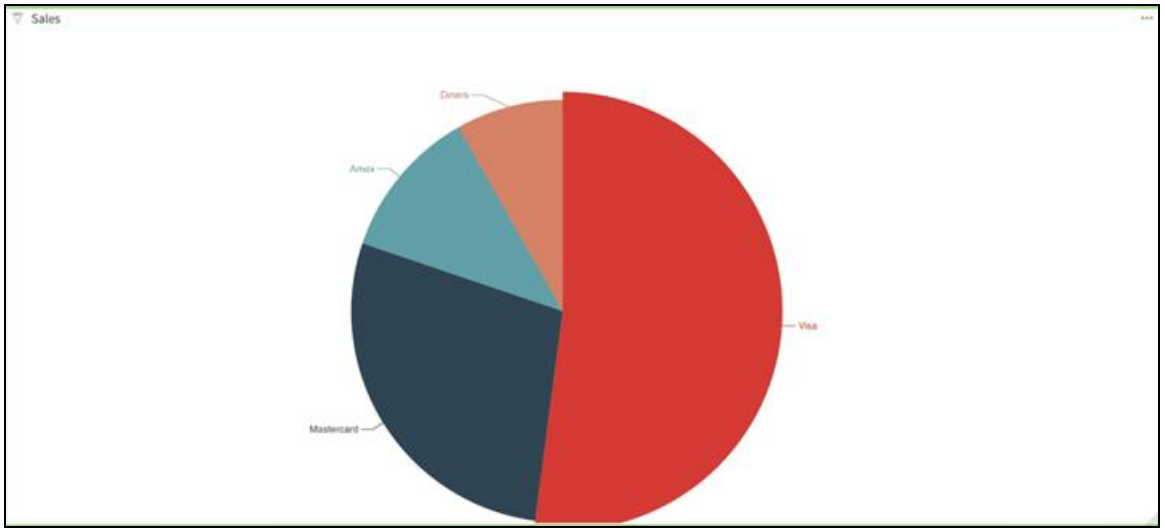
```
//...
const groupAccessor = controller.dataAccessors['Group By'];
const metricAccessor = controller.dataAccessors.Size;

const chart = echarts.init(chartContainer);
const option = {
  series: [
    {
      name: metricAccessor.getLabel(),
      type: 'pie',
    },
  ],
};
//...
```

   Notice how we used the metric accessor to dynamically capture the name of the series.

2. Specify the **pie** series data and call the chart's `setOption` method inside of the `controller.update` function:

```
//...
controller.update = data => {
  // Called when new data arrives
  option.series[0].data = reshapeData(data);
  chart.setOption(option);
};
//...
```

A pie chart renders similar to this:



Unfortunately, the pie chart is slightly cut off at the bottom, so the window needs to be resized. Read on.

Step 6. Handle Resizing

Composer provides an API to notify the charts of a resize operation.

Resize the window

- Add the `controller.resize` function at the end of the `src/index.js` file.

```
//...
controller.resize = (newWidth, newHeight) => {
  // Called when the widget is resized
  chart.resize();
};
```

The `resize` method of the EChart's chart instance tells the chart to fit the new dimensions of the chart container.

Play around and add some filters or change the chart's default configuration. You should see the data re-render as you modify the query with the filter or time bar control.

Looking Good! You now have a chart you can reuse with any data source. Continue to Part 4: Custom Chart Controls.

Part 4: Custom Chart Controls

This part of the custom chart tutorial adds controls found in out-of-the-box Composer charts: axis labels/pickers, tooltips, and a context menu. The following steps are performed in Part 4:

Step 1. Add Axis Labels and Pickers

It is very common for end users to want to explore a data set by changing the group-by and metric fields used by a chart. Composer provides an API for chart developers to quickly add axis labels that can act as controls to replace group-by and metric selections.

Your current custom chart, created in this tutorial, has a default group-by and metric selection defined in the data source.

Add axis labels so that you can change the group-by and metric selections within the chart itself

1. In the terminal window, change directories to the local directory of “My First Custom Chart” and enter the following command to set the custom chart CLI in watch mode, if it is not in watch mode already.

   ```
   cmp-chart watch
   ```

2. In another terminal window, run the following command to compile a development version of the chart’s code using the webpack bundler:

   ```
   npm start
   ```

   For a refresher on this topic, see Step 4. Edit The Chart Code.

3. Edit the `src/index.js` file in your preferred text editor or integrated development environment.

4. Add the following code to the end of the `src/index.js` file:

```
//...
controller.createAxisLabel({
  picks: 'Group By',
                                        position: 'bottom',
                                        orientation: 'horizontal',
                                        });

                                        controller.createAxisLabel({
                                        picks: 'Size',
                                        position: 'bottom',
                                        orientation: 'horizontal',
                                  });
```

The `controller.createAxisLabel` method takes an object with the configuration options of the axis label. The following table describes possible option names and values you can use:

Option	Description
`picks`	The name of the query variable. Use the data accessors `getName` method to avoid hard-coding variable names.
`position`	The location of the axis label in the chart widget. Valid options are `bottom`, `left`, `right`, and `top`.
`orientation`	The orientation of the axis label text. Valid options are `vertical` and `horizontal`.

5. Preview the chart in a dashboard and notice the new axis labels/pickers below the chart. Play around with the controls and notice how the chart reacts to the changes in the query.

Step 2. Add Tooltips

In addition to axis labels, custom charts can define Composer tooltips so the custom chart's look and feel is consistent with out-of-the-box charts.

Define the tooltips using Composer's API

1. Add the original Composer data to the objects in the data set. We simply need to modify the `reshapeData` function in the `src/index.js` file as follows:

```
function reshapeData(data) {
                                        return data.map(d => ({
```

```
                                    name: groupAccessor.raw(d),
                                    value: metricAccessor.raw(d)
                                    datum: d
                                    }));
                                }
```

2. Add `mousemove` and `mouseout` event handlers to the chart:

```
chart.on('mousemove', param => {
                                    controller.tooltip.show({
                                    event: param.event.event,
                                    data: () => param.data.datum,
                                    });
                                });
```

```
chart.on('mouseout', param => {
                                    controller.tooltip.hide();
                                });
```

Notice the use of `controller.tooltip.show` and `controller.tooltip.hide`. The `show` method receives an option with the following properties:

Option	Description
x	The x coordinate in the screen where the tooltip should render
y	The y coordinate in the screen where the tooltip should render
data	Function that returns a Composer datum object

Optional Properties

Option	Description
event	Takes a native browser event to specify the tooltip position. An alternative to x and y properties
content	Function that returns an HTML string to replace the content inside of the tooltip boxes

When you mouse over a pie slice, the `mousemove` handler runs and the `controller.tooltip.show` API is called to display the tooltip. Likewise, the `mouseout` event handler runs when the mouse leaves the pie slice and the `controller.tooltip.hide` API is called to hide the tooltip.

Step 3. Add the Context Menu

The context menu is a control found in many of the out-of-the-box charts. Visual developers select a chart's data point to trigger the display of the context menu. Usually, the context menu is used to do quick filtering operations on the chart itself or other charts in the dashboard. An overview of the actions that can be selected on the context menu can be reviewed in Use The Context Menu.

Composer provides an API that can be used to show the context menu with default actions.

Add the context menu to your custom pie chart

- Add the following code in the `src/index.js` file:

```
chart.on('mousemove', param => {
                                        controller.tooltip.show({
                                        event: param.event.event,
                                        data: () => param.data.datum,
                                        });
                                        });

                                        chart.on('mouseout', param => {
                                        controller.tooltip.hide();
                                        });

                                                chart.on('click', param => {
                                                controller.menu.show({
                                                event: param.event.event,
                                                data: () => param.data.datum,
                                                });
                                    });
```

Notice the use of `controller.menu.show`. The `show` method receives an option with the following properties:

Option	Description
x	The x coordinate in the screen where the menu should render
y	The y coordinate in the screen where the menu should render
data	Function that returns a Composer datum object

Optional Properties

Option	Description
event	Takes a native browser event to specify the menu position. An alternative to `x` and `y` properties.
menu	Used to customize the list of actions that display in the menu. Custom actions along with their callbacks can be defined. Example: `{ items: ['Details', 'Filter', 'Keyset', 'Trend', 'Zoom', 'Remove', 'Custom'], 'Custom': function() { // some custom action } }`

Now, whenever we click an option, the context menu displays and any of its actions can be triggered.

Step 4. Enable Cross-Visual Filtering

If you want your new chart to interact with other charts using cross-visual filtering, some controls must be enabled using the CLI.

1. Enter the following command in the terminal window:

```
cmp-chart edit
```

2. Follow the prompts and select the **Controls** option.
3. Use the arrows and the spacebar to select:

i. **Publish** to allow a chart with the context menu enabled to filter other charts on a dashboard.

ii. **Subscribe** to allow a chart to receive filters from other charts on the dashboard

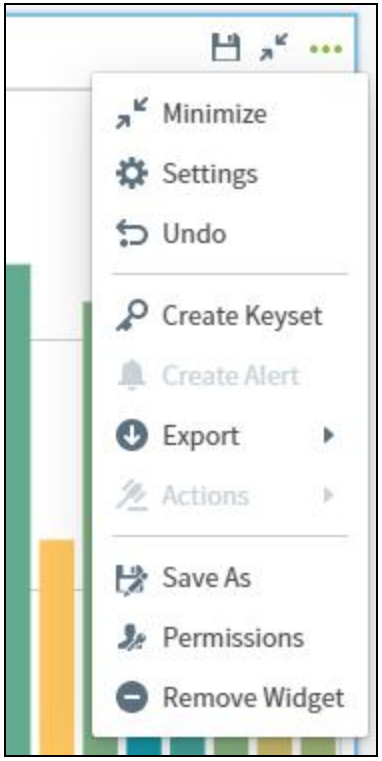
4. Press **Enter** to make your selections.
5. Enter **N** to the prompt asking if you would like to make additional edits.
6. Refresh the custom chart in a dashboard.
7. Open the Dashboard Interactions dialog and select the **Cross-Visual Filtering** tab. See Understand The Cross-Visual Filtering Tab.
8. Select your custom chart from the list on the left of the tab and configure the publish and subscribe settings for the custom chart. See Publish A Link and Subscribe A Visual To A Link.

9. Select a pie slice in the custom pie chart to open the context menu.

   If you published the link for the field on which the pie chart is grouped, the Filter option appears in the context menu. Select this option to create a cross-visual filter using the link field for all the visuals on the dashboard that subscribe to the link.

Step 5. Add Other Controls

In addition to the controls described elsewhere in this tutorial, Composer allows chart developers to show or hide controls on the chart's menu:



The display of these controls is configured using the CLI.

Add the Color control to the chart

1. Enter the following command in the terminal window:

```
cmp-chart edit
```

2. Follow the prompts and select the **Controls** option.
3. Use the arrows and the spacebar to select **Color** from the list.
4. Press **Enter** to make your selections.
5. Enter **N** to the prompt asking if you would like to make additional edits.
6. Refresh the custom chart in a dashboard and notice the new color control on the chart's drop-down menu.

   If you try to use the Color control, however, the panel does not open. The color control only works if you set one of the query variables as the driver color. Since this is a pie chart with a different color per slice, modify the group-by variable to use it as the driver of color in the chart.
7. Enter the following command in the terminal window:

   ```
   cmp-chart edit
   ```
8. Follow the prompts and select the **Variables** option.
9. Select **Edit a variable**.
10. Select **Group By | group |**.
11. Select **Configuration**.
12. Select **Attribute & Time**.
13. Enter **Y** to the prompt asking whether this variable will drive the color in your chart.
14. Enter **N** to the prompt asking if you would like to make additional edits.
15. Switch back to your dashboard and refresh the screen to get the latest changes to the chart. The color control should now work, but the colors displayed do not match the ones in the pie chart.
16. To fix the color mismatch, ECharts must be updated to identify the color to use for each of the pie slices. You can use a data accessor in the API. Edit the `src/index.js` file and make the following minor modification:

```
function reshapeData(data) {
                                        return data.map(d => ({
                                        name: groupAccessor.raw(d),
                                        value: metricAccessor.raw(d),
                                        datum: d,
                                        itemStyle: {
                                        color: groupAccessor.color(d),
                                        },
                                        }));
                                    }
```

Notice how the `groupAccessor` is used to extract the color since it is tied to the Group By variable that is driving the color of the chart. In the Composer dashboard, open the chart's color palette and select the second color palette from the list.

Step 6. Create a Production Bundle

Throughout the tutorial, you have worked with the development version of the chart's code. When you run the `npm start` command, webpack compiles the chart's assets into a single `visualization.js` file that is optimized for production usage. If you inspect the contents of the file, you will notice that the code is not minified and uglified. When the chart's code is in a state that's ready to be consumed by end-users, we recommend that you build a production version of the code that is optimized and has a reduced bundle size.

Create a production bundle

1. Run the following command in a terminal window. This instructs webpack to create an optimized bundle in the `visualization.js` file.

```
npm run build
```

2. Run the following command to push the production bundle to the Composer server:

```
cmp-chart push
```

You have completed the final part of the custom chart tutorial! You are now armed with the necessary tools and knowledge to go and build new and feature-rich charts in Composer. You can also view the Spark Cast produced about custom charts.

Custom Chart API

Getting Started

The custom chart API provides developers with an interface to create interactive charts that integrate with the native Composer chart controls. To get started with the custom chart API, you need to install the Custom Chart CLI. The tool is designed to help you manage all aspects of the custom chart creation process in Composer.

Note: You must be an administrator to configure the chart CLI.

For more detailed step-by-step instructions on getting started, see *A Custom Chart Tutorial*.

Install Composer's Custom Chart CLI

```
npm install composer-chart-cli@latest -g
```

Use the ComposerCustom Chart CLI

1. Configure the CLI. `cmp-chart config`
2. Create a new custom chart. `cmp-chart init <some_path>/<your_chart_name>`
3. From the newly created custom chart directory, run `npm install` – This will install your custom chart's development dependencies.
4. Compile the custom chart code that will be pushed to the server —`npm run build`
5. Push the custom chart to the server – `cmp-chart push`
6. Enable the custom chart for a source. Navigate to the Sources page, locate a data source configuration to edit, and select the more menu (•••) button. Select **Available Visual Types**, then locate and enable your chart in the Custom Visual Types list.
7. Create a new dashboard with your custom chart. You should see a chart widget with a Group and Metric picker.
8. You are now ready to continue building out your custom chart.

Add Custom Chart Packages

Run `cmp-chart import <name> <filepath.zip>` to add the specified custom chart to the Composer server.

For more information about working with custom charts, see these topics:

- Chart Variables
- Controller
- Create Your Own Chart Container
- Receive Chart Data
- Transform Data Using Data Accessors
- Get Values From Constant Variables
- React To Resize Events
- Update Queries With Axis Labels Or Pickers
- Add Composer Tooltips
- Interacting With The Composer Context Menu
- Listening To Other Query Events

Add Composer Tooltips

Charting libraries often include a tooltip implementation that may not look similar to the tooltips included with Composer native charts. As a developer, you can re-use the tooltip design in your custom charts by leveraging the `show` and `hide` methods in `controller.tooltip`.

To show a tooltip, use `controller.tooltip.show`. The `show`method takes an object as its only argument with the following properties:

Option	Description
`x`	The x coordinate on the screen where the tooltip should render.
`y`	The y coordinate on the screen where the tooltip should render.
`data`	Function that returns a Composer data element.

Optional properties:

Option	Description
`event`	Takes a native browser event to specify the tooltip position. Use as an alternative to x and y properties.
`content`	Function that returns an HTML string to replace the content inside of the tooltip boxes. This can be used to render custom tooltips.

Example:

```
myChart.on('mouseover', function(param) {
  controller.tooltip.show({
    x: param.x,
    y: param.y,
    data: function() {
      return param.composerDataElement;
    },
  });
});
```

In the above example, we have a reference to chart instance stored in the `myChart` variable. It hooks into the `mouseover`event of the chart and provides a handler function. The chart provides the handler function with an object argument `param` with the x and y screen coordinates that describe where the tooltip should display.

In this example, the chart has also provided the original Composer data element for the hovered point. `controller.tooltip.show` can be called with an object that specifies the tooltip location and data for the tooltip.

To hide the tooltip, we can call the `controller.tooltip.hide` method.

Example:

```
myChart.on('mouseout', function() {
  controller.tooltip.hide();
})
```

Here you hook into the `mouseout` event and provide a handler function that calls the `controller.toolitp.hide` method.

Chart Variables

Chart variables serve as configuration parameters that can be read from the chart's code and are used to promote reusability of charts across Composer data sources. There are two types of chart variables in Composer:

- Query Variables
- Constant Variables

Query Variables

These variables drive the type of query that executes against the backend database. For example, a chart with a single query variable of type **Group** aggregates the data based on the configured field and uses **count** as the aggregation function. If the developer additionally adds a query variable of type **Multi-Metric**, the data includes the aggregated values of the configured metrics based on the aggregation functions specified.

Constant Variables

These variables are useful for chart settings specified as numeric, text, or list values. For example, a custom chart might use a service that requires an API key. A chart developer can create a **string** constant variable for the API key to allow users to specify a different key per data source.

Supported Variable Types

A list of the various variable types supported by Composer custom charts and their configurations can be found here.

Create Your Own Chart Container

Composer provides a reference to an HTML DIV element that developers can use as a container for their charts. This element is accessed via the `controller.element` property. Sometimes, it is useful to create an inner chart container inside the `controller.element` div to gain full control over its styling via the CSS chart components.

Example:

```
// In visualization.js

var chartContainer = document.createElement('div');
chartContainer.style.width = '100%';
chartContainer.style.height = '100%';
chartContainer.classList.add('chart-container');
controller.element.appendChild(chartContainer);
```

To add a border around the chart container:

```
/* In GENERIC-TABLESTYLE.css */

div.chart-container {
  border: 1px solid black;
}
```

Now you can use the div with the class `chart-container` to render your chart.

Controller

The global object controller exposed the Composer custom chart API. The controller object contains several properties holding information about the chart. These include, but are not limited to:

- HTML Element to be used as a chart container (`controller.element`)
- Object with properties containing the configuration of each query variable (`controller.dataAccessors`)
- Object with the information about the current Composer data source (`controller.source`)
- Object with properties to access the value of each constant variable (`controller.variables`)

Handler functions can override several methods of the `controller` object to react to chart events like data updates, chart resizing, and query errors. Triggering these events execute the specified handler function.

For more information about the `controller` properties and methods, visit the individual guides for tips on how to use custom chart API.

Interacting with the Composer Context Menu

The Composer context menu is a native control that can be added to custom charts to provide users with a set of standard chart interactions. Your chart must be running an aggregated data query to leverage the context menu.

Default context menu actions:

Action	Description
Details	Open a panel with a Raw Data table chart displaying the full set of records matching the selected data point.
Filter	Open a panel to select other charts in the dashboard to filter with a condition matching the selected data point.
Keyset	Open a panel to create a keyset from a data point.
Trend	Switches the current chart to the included **Line Chart: Multiple Metrics** chart and adds the selected data point as a filter.
Zoom	Open a panel to select a new attribute to drill into. The selection automatically changes the group-by field in the query and adds a filter condition matching the selected data point.
Remove	Removes the selected data point by adding a filter to the query to exclude it.
Settings	Opens the visual sidebar menu.

Optional context menu options:

Action	Description
Actions	Invoke an action for the visual. Only visible if an action template is defined and enabled for the data source.
Link	Access a dashboard that has been linked to the visual. Only visible if a dashboard link exists.
Create Alert	Open the create alert work area.

Create a Context Menu with Custom Actions

Add custom actions to your context menu, and bind methods to left and right mouse click actions in place of default actions.

```
menuEventsConfig: {
click: 'filter',
contextmenu: 'openMenu',
customActions: [{
name: 'google help',
action: (data) => window.open (''https://www.google.com/', '_blank').focus(),
```

```
                },
            }]
```

Define your base configuration object. This is optional; if you don't define a base configuration object, Composer deploys the default context menu behavior.

Optionally, include a `customActions` items array. Your `customActions` can be bound as a method for the left or right mouse click in place of one of the default actions.

Defining `customActions`:

Option	Description
`name`	A name for each of the `customActions`. Required.
`action`	The action you define for the `customActions`. Required.
`icon`	An icon to associate with the `customActions`. Optional.

Listening to Other Query Events

Chart developers may need to apply particular logic to their charts whenever a query event is triggered. Handler functions can be defined to override the following list of query methods:

- `controller.onStart`: Called as soon as the query execution begins.
- `controller.onNoDataFound`: Called when no data is available for the given query.
- `controller.onNotDirtyData`: Called as soon as all of the data is received.
- `controller.onStreamError`: Called when the query execution returns an error from the server.

Get Values From Constant Variables

The values of constant variables are accessed by reading the properties of `controller.variables`. This object contains a property for each constant variable defined.

Example:

```
var apiKey = controller.variables['API Key'];

// use the apiKey in your code
```

In the above example, the chart has a constant variable of type “string” defined with the name “API Key”. We access the value by reading the property “API Key” from `controller.variables`.

React to Resize Events

When a user resizes a chart widget, you must account for the new widget dimensions. You can specify your own resize handler function by overriding the `controller.resize` method.

Example:

```
controller.resize = function(newWidth, newHeight) {
  // If needed, use the newWidth and newHeight values to re-size chart
}
```

The specified handler executes every time a user action causes the widget dimensions to change. The following list provides a few examples of actions that trigger the resize event:

- A user changes the dimensions of the browser window.
- A user changes the dimensions of an individual widget.

Hiding or displaying controls like the Time Bar may reduce or increase the space available for widgets in a dashboard.

Update Queries with Axis Labels or Pickers

Axis Labels or Axis Pickers are Composer native controls that can be added to charts to provide a way for users to change query parameters dynamically. Chart developers can these controls by calling the method `controller.createaAxisLabel`. The `createAxislabel` method takes an object as its only argument with the following properties:

Option	Description
`picks`	Name of the query variable. **Note:** Use the data accessor's `getName` method to avoid hardcoding variable names.
`position`	Location of the axis label in the chart widget. Valid options: `bottom`, `left`, `right`, `top`
`orientation`	Orientation of the axis label text. Valid options: `vertical`, `horizontal`

Example:

```
controller.createAxisLabel({
  picks: 'Group By',
  position: 'bottom',
  orientation: 'horizontal',
});

controller.createAxisLabel({
  picks: 'Size',
  position: 'bottom',
  orientation: 'horizontal',
});
```

The above example adds two axis pickers to the chart that provide users with the ability to change the fields used for the `Group By` and `Size` query variables.

Transform Data Using Data Accessors

When working with charting libraries, it is often necessary to structure your data elements in a way the charting libraries understand.

For instance, given the following structure of data element received from the server:

```
{
  "group": ["$0 to $25,000"],
  "current": {
    "count": 400,
    "metrics": {
      "price": {
        "sum": 52150
      }
    }
  }
}
```

You may need to transform it to:

```
{
  "name": "$0 to $25,000",
  "value": "52150"}
```

We are interested in the values specified in the `group` array and the metric value defined in `price.sum`. You may be inclined to create a new data array by mapping each data element to the necessary structure like:

```
var newData = data.map(function(d) {
  return {
    name: d.group[0],
    value: d.current.metrics.price.sum,
  };
});
```

The downside of this approach is that you have now hardcoded the paths of your metric values to a specific field name (“price”) and a specific aggregation function (“sum”).

A Better Approach Using Data Accessors

A `Data Accessor` is an object with methods that can be used to extract information about the current configuration of query variables. A data accessor can also extract data values out of the data elements received from query execution results.

Example:

```
var groupAccessor = controller.dataAccessors['Group By'];
var metricAccessor = controller.dataAccessors.Metric;

var newData = data.map(function(d) {
  return {
    name: groupAccessor.raw(d),
    value: metricAccessor.raw(d),
  };
});
```

To access a data accessor, use the `controller.dataAccessors` object. The query variable's name exists as property in this object. The most commonly used method in data accessors is `raw` which accepts a Composer data element and returns a value corresponding to the query variable. In the above example, we used the `groupAccessor` and `metricAccessor` to retrieve the group and metric values without hard-coding the name of the fields.

Receive Chart Data

To receive the results of queries executed against a Composer data source, chart developers can override the `controller.update` method with a function that will receive the array of data elements.

Example:

```
// In visualization.js

/* This function will get called when new data is received from the server */
controller.update = function(data) {
  // code to update the chart with new data
  // the data argument is an array of objects (data elements)
};
```

Structure of a Data Element in Aggregated Queries

When working with aggregated queries that have `Group` or `Multi-Group` variables defined, you can expect to receive data elements with the following structure:

```
{
  "group": ["Books", "$0 to $25,000"],
  "current": {
    "count": 1267,
    "metrics": {
      "price": {
        "sum": 1077100
      }
    }
  }
}
```

The above JSON represents the results of a chart query generated by defining a `Multi-Group` variable and `Metric` variable. The `Multi-Group` variable contains two levels of grouping (“Product Group” & “User Income” fields), and the `Metric` variable has a configuration using the “Price” field and the “SUM” function. The count property represents the total count of records for the grouping combination. This property is always available in all aggregated queries.

Structure of a Data Element in Non-Aggregated Queries

When working with non-aggregated queries, the data elements in the array received from the server represent a row of data in the data source. Each row is made up of an array of values; one for each of the fields requested:

```
["Male", "Visa", "400"]
```

The above JSON represents the results of a chart query generated by defining an `Ungrouped` variable with three fields requested: “Gender”, “Payment Type”, “Price”.

Visual Type Configuration Properties

This section describes the properties of visualization type configurations, possible values, and the definition of those values for creating or editing visual types (custom charts) for Composer.

Note: You must be an administrator to manage custom visual types.

Properties

`name` - ***string***

`name` is the name of visual type the configuration file represents, and displays in the Composer user interface. This name must be unique in your Composer instance, or it will overwrite or be overwritten by a visual type using the same name.

`type` - ***string***

`type` is one of the properties Composer uses internally to represent and distinguish visual types. Built-in visuals have unique `type` properties. Any visual you add to the system should use the same value, `CUSTOM`.

`controls` - ***array of strings***

`controls` is the list of things you enable this visual type to do. Supported values are included in the chart below:

Control	Description
UberStyle	Allows an instance of this visual type to use the Style Switcher to switch to other visual types.
Color	Allows an instance of this visual type to show the Color panel within the sidebar or pop-up editor.
Download	Allows instances of this visual type to be downloaded as an image, PDF, or configuration files.
Filters	Allows this visual type to be filtered using the filter editor in the sidebar or pop-up editor.
Publish	Enables this visualization for publication. If this control is not present, the “Filter” option is shown in a context menu.
Settings	Enables the settings panel in the sidebar or pop-up editor. Use the settings panel to edit the variables added to the visualization.
Sort	Enables the Sort & Limit panel in the sidebar or pop-up editor. This panel is used to sort and limit the data returned. Only certain variable types support sort and limit.
Subscribe	Allows the visual type to be the target of filters added from external sources and other visuals on the dashboard.
TimePlayer	Enables the Time panel in the sidebar or pop-up editor. This allows changing the field used for the timebar or turning off the timebar for each instance of the visual.
Undo	Allows users to undo changes they make to instances of this visualization.

Note: Additional properties may be present in a visualization's `controls` array but they may deprecated, irrelevant to a custom visual type that doesn't use the properties.

`variables` - ***array of objects***

`variables` are required for the configuration. This array of objects define data and non-data values associated with instances of this visualization. All supported variable configuration are included in the list below.

- attribute
- bool
- box-plot-metric
- color
- float
- group
- histogram-group
- integer
- metric
- multi-group
- multilist
- multi-metric
- singlelist
- string

- text
- ungrouped
- ungroupedList

You can also maintain custom charts using the CLI. See Maintain Custom Charts Using The Custom Chart CLI.

attribute

The `attribute` variable is part of the array of objects you use to define data and non-data values for this visualization.

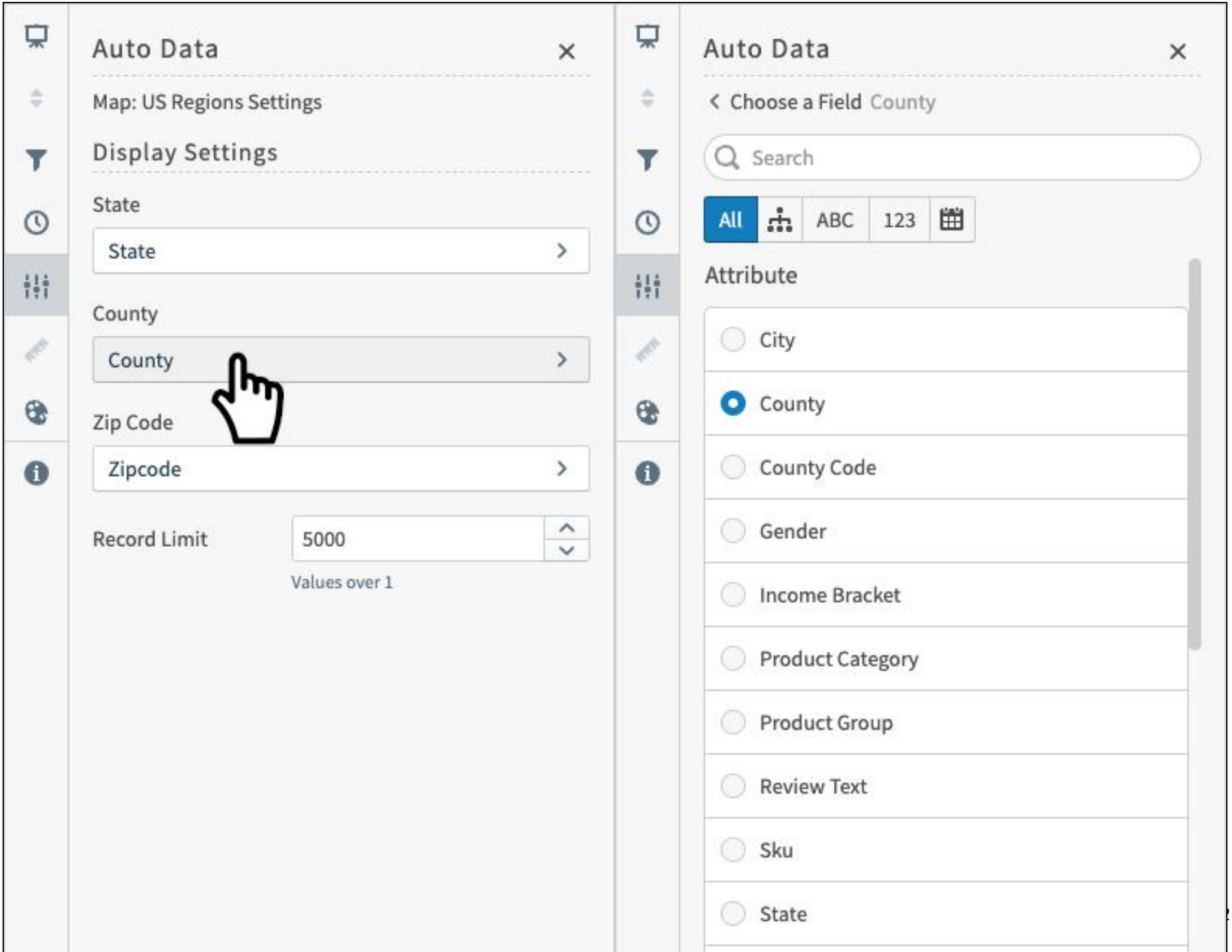
Note: You must be an administrator to manage custom visual types.

Type

```
attribute
```

Editor

`attribute` variables are represented in the side panel as a button that users use to select a field. This variable's selected field is not used as part of the data request by default, but the name is provided for other uses or for manual configuration within the visualization's code.


Auto Data
Map: US Regions Settings
Display Settings
State
State
County
County
Zip Code
Zipcode
Record Limit
5000
Values over 1
Auto Data
Choose a Field County
Search
All
ABC
123
Attribute
City
County
County Code
Gender
Income Bracket
Product Category
Product Group
Review Text
Sku
State

Variable Specific Properties

Property	Argument	Description
None.		

Generic Properties

Generic properties apply to many variables.

Property	Argument	Description
`name`	string	The name of the property. It must be unique among all variables on the visualization. This name is used to access the data accessor and/or variable value on the controller provided to the visualization.
`descr`	string	A description of the variable for your reference. This is not displayed in the UI.
`attributeType`	array of strings	A list of field types to include when selecting fields for this variable. Accepted values: ■ `ATTRIBUTE` ■ `TIME` ■ `NUMBER` ■ `INTEGER`
`required`	boolean	Triggers the UI to require a value for certain variable types. Not used for this variable type.

Deprecated Properties

Property
None.

Samples

```
{
        "name": "State",
        "type": "attribute",
        "descr": "A field which represents a US State",
        "attributeType": [
                "ATTRIBUTE"
        ],
        "required": false
}
```

See Visual Type Configuration Properties.

bool

The `bool` variable is part of the array of objects you use to define data and non-data values for this visualization.

Note: You must be an administrator to manage custom visual types.

Type

```
bool
```

Editor

`bool` variables are represented in the side panel as a simple switch.

Variable Specific Properties

Property	Argument	Description
None.		

Generic Properties

Property	Argument	Description
`name`	string	The name of the property. It must be unique among all variables on the visualization. This name is used to access the data accessor and/or variable value on the controller provided to the visualization.
`descr`	string	A description of the variable for your reference. This is not displayed in the UI.
`attributeType`	array of strings	A list of field types to include when selecting fields for this variable. Accepted values:

Property	Argument	Description
		■ ATTRIBUTE ■ TIME ■ NUMBER ■ INTEGER
required	boolean	Triggers the UI to require a value for certain variable types. Not used for this variable type.

Deprecated Properties

Property
None.

Samples

```
{
        "name": "Show Metric Values",
        "type": "bool",
        "defaultValue": true,
        "colorMetric": false,
        "colorNumb": 0,
        "required": false
}
```

See Visual Type Configuration Properties.

box-plot-metric

The `box-plot-metric` variable is part of the array of objects you use to define data and non-data values for this visualization.

Note: You must be an administrator to manage custom visual types.

Type

```
box-plot-metric
```

Editor

`box-plot-metric` variables are not represented on the settings panel. Enable editing a `box-plot-metric` variable by adding an axis label for the variable using the `createAxisLabel` method on the controller.

Variable Specific Properties

Property		Argument	Description
`defaultValue`		object	Allows you to set the default configuration for the specified field.
	`func`	string	Set to `percentiles`. This is the only value supported.
	`args`	array of integers	A list of the percentiles to calculated. Supported values: ■ 0 ■ 25 ■ 50 ■ 75 ■ 100

Generic Properties

Generic properties apply to many variables.

Property	Argument	Description
`name`	string	The name of the property. It must be unique among all variables on the visualization. This name is used to access the data accessor and/or variable value on the controller provided to the visualization.
`descr`	string	A description of the variable for your reference. This is not displayed in the UI.
`attributeType`	array of strings	A list of field types to include when selecting fields for this variable. Accepted values: ■ `ATTRIBUTE` ■ `TIME` ■ `NUMBER` ■ `INTEGER`
`colorNumb`	integer	The number of colors to include in the color palette by default. 0 will choose the smallest palette available.
`required`	boolean	Triggers the UI to require a value for certain variable types. Not used for this variable type.

Deprecated Properties

Property
None.

Samples

```
{
       "name": "Box Plot Metric",
       "type": "box-plot-metric",
```

```
        "attributeType": [
                "INTEGER",
                "NUMBER"
        ],
        "defaultValue": {
                "func": "percentiles",
                "args": [
                        0,
                        25,
                        50,
                        75,
                        100
                ]
        },
        "colorNumb": 0,
        "required": false
}
```

See Visual Type Configuration Properties.

color

The `color`variable is part of the array of objects you use to define data and non-data values for this visualization.

Note: You must be an administrator to manage custom visual types.

Type

```
color
```

Editor

Edit `color` variables using the color panel. The visual type must include the control color in order to modify these variables.

Variable Specific Properties

Property	Argument	Description
`defaultValue`	string	A color in any CSS color format. The value `_inherit` uses the color from the currently selected theme.

Generic Properties

Generic properties apply to many variables.

Property	Argument	Description
`name`	string	The name of the property. It must be unique among all variables on the visualization. This name is used to access the data accessor and/or variable value on the controller provided to the visualization.
`descr`	string	A description of the variable for your reference. This is not displayed in the UI.
`attributeType`	array of strings	A list of field types to include when selecting fields for this variable. Accepted values: ■ `ATTRIBUTE`

Property	Argument	Description
		■ TIME ■ NUMBER ■ INTEGER
required	boolean	Triggers the UI to require a value for certain variable types. Not used for this variable type.

Deprecated Properties

Property
None.

Samples

```
{
        "name": "Label Color",
        "type": "color",
        "descr": "A color for the label",
        "defaultValue": "_inherit",
        "required": false
}
```

See Visual Type Configuration Properties.

float

The `float` variable is part of the array of objects you use to define data and non-data values for this visualization.

Note: You must be an administrator to manage custom visual types.

Type

```
float
```

Editor

Edit `float` variables using a standard browser number input. If minimum, maximum, or both values are specified in the `config`, a helper message shows below the input to notify the user of these ranges.

Variable Specific Properties

Property		Argument	Description
`defaultValue`		float	
`config`		object	All properties are optional.
	`min`	float	The minimum value allowed for this variable. Inclusive.
	`max`	float	The maximum value allowed for this variable. Inclusive.

Generic Properties

Generic properties apply to many variables.

Property	Argument	Description
`name`	string	The name of the property. It must be unique among all variables on the visualization. This name is used to access the data accessor and/or variable value on the controller provided to the visualization.
`descr`	string	A description of the variable for your reference. This is not displayed in the UI.

Property	Argument	Description
`attributeType`	array of strings	A list of field types to include when selecting fields for this variable. Accepted values: ■ `ATTRIBUTE` ■ `TIME` ■ `NUMBER` ■ `INTEGER`
`required`	boolean	Triggers the UI to require a value for certain variable types. Not used for this variable type.

Deprecated Properties

Property
None.

Samples

```
{
        "name": "Float Option",
        "type": "float",
        "descr": "A float value",
        "defaultValue": 1.5,
        "config": {
                "min": 1,
                "max": 22.3
                },
                "required": false
}
```

See Visual Type Configuration Properties.

group

The `group` variable is part of the array of objects you use to define data and non-data values for this visualization.

Note: You must be an administrator to manage custom visual types.

Type

```
group
```

Editor

`group` variables are not represented on the settings panel. Enable editing a `group` variable by adding an axis label for the variable using the `createAxisLabel` method on the controller.

Variable Specific Properties

Property			Argument	Description
`config`			object	All properties are optional.
	`groupColorSet`		string	The name of the color palette to use. The value `_inherit` uses the palette from the currently selected theme.
	`colorGroupIndex`		integer	Define this property to enable color configuration for this variable.
	`autoShowColorLegend`		boolean	Define as `true` to show the legend by default.
`defaultValue`			object	Define defaulting sort and limit properties for this variable. All properties are optional.
	`limit`		integer	Limit to the number of groups to return.
	`sort`		object	
		`dir`	string	Default sort direction. Valid values are `asc` or `desc`.

Generic Properties

Generic properties apply to many variables.

Property	Argument	Description
`name`	string	The name of the property. It must be unique among all variables on the visualization. This name is used to access the data accessor and/or variable value on the controller provided to the visualization.
`descr`	string	A description of the variable for your reference. This is not displayed in the UI.
`attributeType`	array of strings	A list of field types to include when selecting fields for this variable. Accepted values: ■ `ATTRIBUTE` ■ `TIME` ■ `NUMBER` ■ `INTEGER`
`colorNumb`	integer	The number of colors to include in the color palette by default. 0 will choose the smallest palette available.
`required`	boolean	Triggers the UI to require a value for certain variable types. Not used for this variable type.

Deprecated Properties

Property
`colorMetric`
`groupType`

Samples

```
{
        "name": "Group By",
        "type": "group",
        "descr": "The group property",
        "attributeType": [
                "ATTRIBUTE",
                "TIME",
```

```
                "NUMBER",
                "INTEGER"
        ],
        "defaultValue": {
                "limit": 20
        },
        "colorNumb": 0,
                "config": {
                        "groupColorSet": "_inherit",
                        "colorGroupIndex": 0,
                        "autoShowColorLegend": true
                },
                "required": true
}
```

See Visual Type Configuration Properties.

histogram-group

The `histogram-group` variable is part of the array of objects you use to define data and non-data values for this visualization.

Note: You must be an administrator to manage custom visual types.

Type

```
histogram-group
```

Editor

`histogram-group` variables are not represented on the settings panel. Enable editing a `histogram-group` variable by adding an axis label for the variable using the `createAxisLabel` method on the controller. Other options are not presented to users.

Variable Specific Properties

Property			Argument	Description
`config`			object	Sets the default values for this variable.
	`binType`		string	How the system should bin the data.
		`auto`		Define this property to enable color configuration for this variable.
		`count`		Try to conform the number of bins to the given value.
		`width`		Try to conform range number of bins to the given value.
	`binsCount`		integer	Default number of bins to display when `binType` is set to `count`.
	`binsCount`		number	Default number of bins to display when `binType` is set to `width`.
	`values`		string	`absolute` suggests to the visual type that the actual sum of the bin should be y-axis values. `relative` suggests to the visual type that the values should be percentages of the whole sum of all bins.
	`cumulative`		boolean	If `true`, the chart should include a representation of the cumulative value as the bins move from left to right. Typically represented as an overlaid line.

Generic Properties

Generic properties apply to many variables.

Property	Argument	Description
`name`	string	The name of the property. It must be unique among all variables on the visualization. This name is used to access the data accessor and/or variable value on the controller provided to the visualization.
`descr`	string	A description of the variable for your reference. This is not displayed in the UI.
`attributeType`	array of strings	A list of field types to include when selecting fields for this variable. Accepted values: ■ `ATTRIBUTE` ■ `TIME` ■ `NUMBER` ■ `INTEGER`
`colorNumb`	integer	The number of colors to include in the color palette by default. 0 will choose the smallest palette available.
`required`	boolean	Triggers the UI to require a value for certain variable types. Not used for this variable type.

Deprecated Properties

Property
None.

Samples

```
{
        "name": "Group By",
        "type": "histogram-group",
```

```
        "attributeType": [
                "INTEGER",
                "NUMBER"
        ],
        "colorNumb": 0,
        "config": {
                "binsType": "auto",
                "binsCount": 10,
                "binsWidth": 100,
                "values": "absolute",
                "cumulative": false
        },
        "required": false
}
```

See Visual Type Configuration Properties.

integer

The `integer` variable is part of the array of objects you use to define data and non-data values for this visualization.

Note: You must be an administrator to manage custom visual types.

Type

```
integer
```

Editor

Edit `integer` variables using a standard browser number input. If minimum, maximum, or both values are specified in the `config`, a helper message shows below the input to notify the user of these ranges.

Variable Specific Properties

Property		Argument	Description
`defaultValue`		integer	
`config`		object	All properties are optional.
	`min`	integer	The minimum value allowed for this variable. Inclusive.
	`max`	integer	The maximum value allowed for this variable. Inclusive.

Generic Properties

Generic properties apply to many variables.

Property	Argument	Description
`name`	string	The name of the property. It must be unique among all variables on the visualization. This name is used to access the data accessor and/or variable value on the controller provided to the visualization.
`descr`	string	A description of the variable for your reference. This is not displayed in the UI.
`attributeType`	array of strings	A list of field types to include when selecting fields for this variable. Accepted values: ■ `ATTRIBUTE` ■ `TIME` ■ `NUMBER` ■ `INTEGER`
`required`	boolean	Triggers the UI to require a value for certain variable types. Not used for this variable type.

Deprecated Properties

Property
None.

Samples

```
{
        "name": "Integer Option",
        "type": "integer",
        "descr": "An integer value",
        "defaultValue": 1,
        "config": {
                "min": 1,
                "max": 22
        },
        "required": false
}
```

See Visual Type Configuration Properties.

metric

The `metric` variable is part of the array of objects you use to define data and non-data values for this visualization.

Note: You must be an administrator to manage custom visual types.

Type

```
metric
```

Editor

`metric` variables are not represented on the settings panel. Enable editing a `metric` variable by adding an axis label for the variable using the `createAxisLabel` method on the controller.

Variable Specific Properties

Property			Argument	Description
`colorSet`			string	The name of the color palette to use by default. The value `_inherit` uses the palette from the currently selected theme. Other available values include `DefaultSequential` and `DefaultQualitative`.
`metricType`			string	Set to `color` to enable this metric as the color metric. Only set this to one metric on the visualization.
`legendType`			string	Available values include `palette` and `range`.
`config`			object	All properties are optional.
	`displayNoneOption`		boolean	When set to `true`, users can select `None` when choosing a field for this variable.
`defaultValue`			array of object	Enable to allow setting of a default configuration for the chosen field.
	`name`		string	A value of `none` combined with `config.displayNoneOption` set to `true` will default the selected field to `None`. A value of `count` will automatically select the `Volume`metric.
	`colorConfig`		object	
		`autoShowColorLegend`	boolean	If `true`, show a legend for this variable by default.

Property			Argument	Description
		colorNumb	integer	The number of colors to include in the color palette by default. 0 will choose the smallest palette available.
		legendType	string	Available values include palette and range.
		colorScaleType	string	Available values include gradient and distinct.

Generic Properties

Generic properties apply to many variables.

Property	Argument	Description
name	string	The name of the property. It must be unique among all variables on the visualization. This name is used to access the data accessor and/or variable value on the controller provided to the visualization.
descr	string	A description of the variable for your reference. This is not displayed in the UI.
attributeType	array of strings	A list of field types to include when selecting fields for this variable. Accepted values: ■ ATTRIBUTE ■ TIME ■ NUMBER ■ INTEGER
colorNumb	integer	The number of colors to include in the color palette by default. 0 will choose the smallest palette available.
required	boolean	Triggers the UI to require a value for certain variable types. Not used for this variable type.

Deprecated Properties

Property
colorMetric

Property
groupType

Samples

```
{
	"name": "Bar Color",
	"type": "metric",
	"attributeType": [
		"INTEGER",
		"NUMBER"
	],
	"defaultValue": [
		{
			"name": "count",
			"colorConfig": {
				"colorNumb": 3,
				"legendType": "palette",
				"autoShowColorLegend": true
			}
		}
	],
	"config": {
		"displayNoneOption": true
		},
		"colorNumb": 3,
		"colorSet": "_inherit",
		"metricType": "color",
		"legendType": "palette",
		"required": false
}
```

See Visual Type Configuration Properties.

multi-group

The `multi-group` variable is part of the array of objects you use to define data and non-data values for this visualization.

Note: You must be an administrator to manage custom visual types.

Type

```
multi-group
```

Editor

`multi-group` variables are not represented on the settings panel. Enable editing a `multi-group` variable by adding an axis label for the variable using the `createAxisLabel` method on the controller.

Variable Specific Properties

Property	Argument	Description
`config`	object	
`groupLevel`	array of integers	A label to show for each grouping level. The number of entries in this array should match the value set for `groupLevel`.
`groupNames`	array of strings	The number of grouping levels to allow.
`groupTypes`	array of strings	The set of field types to allow for each grouping level. Each entry should be a space-delimited list of all field types to allow. The number of entries in this array should match the value set for `groupLevel`. Accepted values: ■ `ATTRIBUTE` ■ `TIME` ■ `NUMBER` ■ `INTEGER`

Property	Argument	Description
		■ MONEY ■ NONE
groupTypes	string	The name of the color palette to use by default. The value _inherit uses the palette from the currently selected theme.
colorGroupIndex	integer	Define this property to enable color configuration for this variable.
autoShowColorLegend	boolean	If true, show a legend for this variable by default.

Generic Properties

Generic properties apply to many variables.

Property	Argument	Description
name	string	The name of the property. It must be unique among all variables on the visualization. This name is used to access the data accessor and/or variable value on the controller provided to the visualization.
descr	string	A description of the variable for your reference. This is not displayed in the UI.
attributeType	array of strings	A list of field types to include when selecting fields for this variable. Accepted values: ■ ATTRIBUTE ■ TIME ■ NUMBER ■ INTEGER
colorNumb	integer	The number of colors to include in the color palette by default. 0 will choose the smallest palette available.
required	boolean	Triggers the UI to require a value for certain variable types. Not used for this variable type.

Deprecated Properties

Property
colorMetric
groupType

Samples

```
{
	"name": "Multi Group By",
	"type": "multi-group",
	"descr": "Multiple, layered groups",
	"attributeType": [
		"ATTRIBUTE",
		"TIME"
	],
	"colorNumb": 0,
	"config": {
		"groupLevel": 2,
		"groupNames": [
			"Group 1",
			"Group 2"
		],
		"groupLimits": [
			50,
			20
		],
		"groupTypes": [
			"ATTRIBUTE TIME NUMBER INTEGER MONEY",
			"ATTRIBUTE TIME NUMBER INTEGER MONEY NONE"
		],
		"groupColorSet": "_inherit",
		"colorGroupIndex": 0,
		"autoShowColorLegend": true
	},
	"required": true
}
```

See Visual Type Configuration Properties.

multilist

The `multilist` variable is part of the array of objects you use to define data and non-data values for this visualization.

Note: You must be an administrator to manage custom visual types.

Type

```
multilist
```

Editor

`multilist` have two modes:

1. If the `values` property is provided, the variable is represented by a multi-select tag input.

2. If `values` is not provided, the variable is represented by a control that provides an arrangeable list of fields that includes an edit button. Users can select which fields should and should not be included.

Variable Specific Properties

Property	Argument	Description
`defaultValue`	array of strings	Only applicable for the static value list.
`values`	array of strings	The options made available to users for selection.

Generic Properties

Generic properties apply to many variables.

Property	Argument	Description
`name`	string	The name of the property. It must be unique among all variables on the visualization. This name is used to access the data accessor and/or variable value on the controller provided to the visualization.
`descr`	string	A description of the variable for your reference. This is not displayed in the UI.
`attributeType`	array of strings	A list of field types to include when selecting fields for this variable. Accepted values: ■ `ATTRIBUTE` ■ `TIME` ■ `NUMBER` ■ `INTEGER`
`required`	boolean	Triggers the UI to require a value for certain variable types. Not used for this variable type.

Deprecated Properties

Property
None.

Samples

Static Values

```
{
	"name": "Test Static Value Multilist",
	"type": "multilist",
	"descr": "Select multiple values. Order is not selectable",
	"values": [
		"Value 1",
		"Value 2",
		"Value 3",
		"Value 4"
	],
	"defaultValue": [
		"Value 2",
		"Value 3"
	],
	"required": false
}
```

Field Selection

```
{
	"name": "Test Field Multilist",
	"type": "multilist",
	"descr": "Select and arrange multiple fields",
	"attributeType": [
		"ATTRIBUTE",
		"TIME",
		"NUMBER",
		"INTEGER"
	],
	"defaultValue": [],
	"required": false
}
```

See Visual Type Configuration Properties.

multi-metric

The `multi-metric` variable is part of the array of objects you use to define data and non-data values for this visualization.

Note: You must be an administrator to manage custom visual types.

Type

```
multi-metric
```

Editor

`multi-metric` variables are not represented on the settings panel. Enable editing a `multi-metric` variable by adding an axis label for the variable using the `createAxisLabel` method on the controller.

Variable Specific Properties

Property			Argument	Description
`colorSet`			string	The name of the color palette to use by default. The value `_inherit` uses the palette from the currently selected theme. Other available values include `DefaultSequential` and `DefaultQualitative`.
`metricType`			string	Set to `color` to enable this metric as the color metric. Only set this to one metric on the visualization.
`legendType`			string	Available values include `palette` and `range`.
`defaultValue`			array of object	Enable to allow setting of a default configuration for the chosen field.
	`name`		string	A value of `none` combined with `config.displayNoneOption` set to `true` will default the selected field to `None`. A value of `count` will automatically select the `Volume`metric.
	`colorConfig`		object	
		`autoShowColorLegend`	boolean	If `true`, show a legend for this variable by default.
		`colorNumb`	integer	The number of colors to include in the color palette by default. 0 will choose the smallest palette available.
		`legendType`	string	Available values include `palette` and `range`.

Property			Argument	Description
		`colorScaleType`	string	Available values include `gradient` and `distinct`.

Generic Properties

Generic properties apply to many variables.

Property	Argument	Description
`name`	string	The name of the property. It must be unique among all variables on the visualization. This name is used to access the data accessor and/or variable value on the controller provided to the visualization.
`descr`	string	A description of the variable for your reference. This is not displayed in the UI.
`attributeType`	array of strings	A list of field types to include when selecting fields for this variable. Accepted values: ■ `ATTRIBUTE` ■ `TIME` ■ `NUMBER` ■ `INTEGER`
`colorNumb`	integer	The number of colors to include in the color palette by default. 0 will choose the smallest palette available.
`required`	boolean	Triggers the UI to require a value for certain variable types. Not used for this variable type.

Deprecated Properties

Property
`colorMetric`

Samples

```
{
        "name": "Y1 Axis",
        "type": "multi-metric",
        "attributeType": [
                "INTEGER",
                "NUMBER"
        ],
        "defaultValue": [
                {
                        "colorConfig": {
                                "autoShowColorLegend": true
                        }
                }
        ],
        "colorNumb": 4,
        "metricType": "color",
        "colorSet": "_inherit",
        "legendType": "palette",
        "required": false
}
```

See Visual Type Configuration Properties.

singlelist

The `singlelist` variable is part of the array of objects you use to define data and non-data values for this visualization.

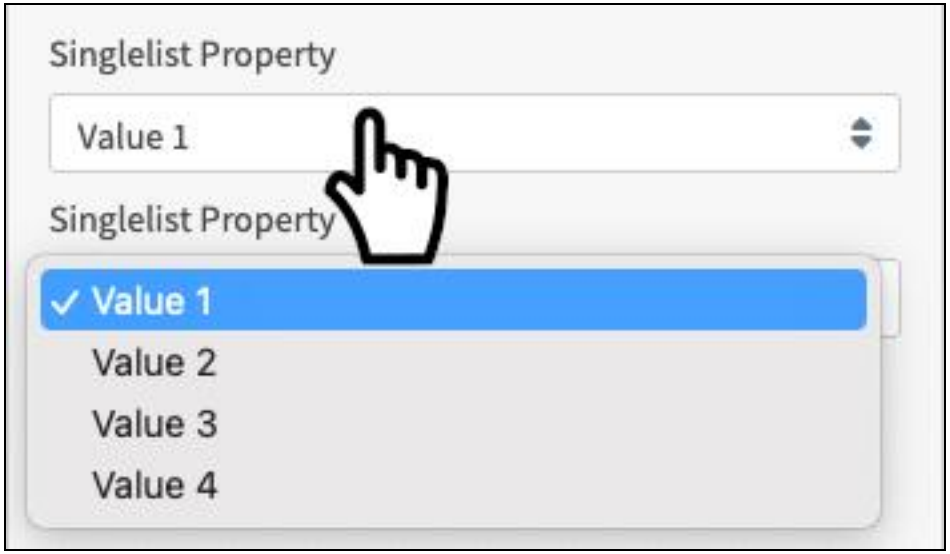
Note: You must be an administrator to manage custom visual types.

Type

```
singlelist
```

Editor

`singlelist` variables are represented in the settings sidebar as a simple drop down.



Variable Specific Properties

Property	Argument	Description
`defaultValue`	string	

Property	Argument	Description
`values`	array of strings	The options you make available for users to select from.

Generic Properties

Generic properties apply to many variables.

Property	Argument	Description
`name`	string	The name of the property. It must be unique among all variables on the visualization. This name is used to access the data accessor and/or variable value on the controller provided to the visualization.
`descr`	string	A description of the variable for your reference. This is not displayed in the UI.
`attributeType`	array of strings	A list of field types to include when selecting fields for this variable. Accepted values: ■ `ATTRIBUTE` ■ `TIME` ■ `NUMBER` ■ `INTEGER`
`required`	boolean	Triggers the UI to require a value for certain variable types. Not used for this variable type.

Deprecated Properties

Property
None.

Samples

```
{
	"name": "Tile Provider",
	"type": "singlelist",
	"descr": "Map tile provider",
	"values": [
		"OpenStreetMap (no API key required)",
		"MapQuest",
		"MapBox",
		"N/A"
	],
	"defaultValue": "OpenStreetMap (no API key required)",
	"required": false
}
```

See Visual Type Configuration Properties.

string

The `string` variable is part of the array of objects you use to define data and non-data values for this visualization.

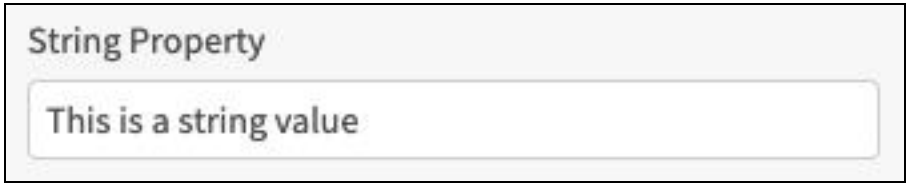
Note: You must be an administrator to manage custom visual types.

Type

```
string
```

Editor

Users can edit `string` variables using a standard browser text input.



Variable Specific Properties

Property	Argument	Description
`defaultValue`	string	

Generic Properties

Generic properties apply to many variables.

Property	Argument	Description
`name`	string	The name of the property. It must be unique among all variables on the visualization. This name is used to access the data accessor and/or variable value on the controller provided to the visualization.

Property	Argument	Description
`descr`	string	A description of the variable for your reference. This is not displayed in the UI.
`attributeType`	array of strings	A list of field types to include when selecting fields for this variable. Accepted values: ■ `ATTRIBUTE` ■ `TIME` ■ `NUMBER` ■ `INTEGER`
`required`	boolean	Triggers the UI to require a value for certain variable types. Not used for this variable type.

Deprecated Properties

Property
None.

Samples

```
{
        "name": "Tile Provider API Key",
        "type": "string",
        "descr": "API key for the tile provider chosen. OpenStreetMap does not require an API key.",
        "required": false
}
```

See Visual Type Configuration Properties.

text

The `text` variable is part of the array of objects you use to define data and non-data values for this visualization.

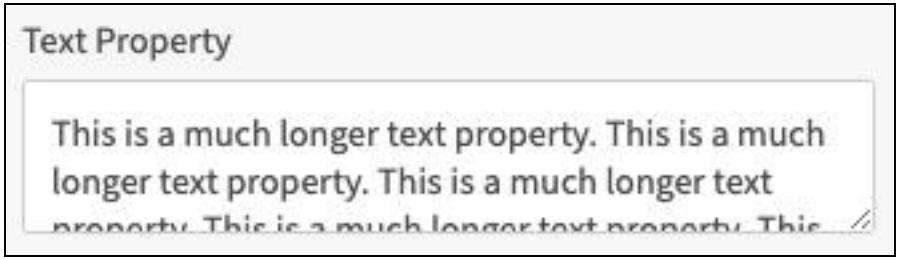
Note: You must be an administrator to manage custom visual types.

Type

```
text
```

Editor

Users can edit `text` variables using a standard browser text input.



Variable Specific Properties

Property	Argument	Description
`defaultValue`	string	

Generic Properties

Generic properties apply to many variables.

Property	Argument	Description
`name`	string	The name of the property. It must be unique among all variables on the visualization. This name is used to access the data accessor and/or variable value on the controller provided to the visualization.
`descr`	string	A description of the variable for your reference. This is not displayed in the UI.
`attributeType`	array of strings	A list of field types to include when selecting fields for this variable. Accepted values: ■ `ATTRIBUTE` ■ `TIME` ■ `NUMBER` ■ `INTEGER`
`required`	boolean	Triggers the UI to require a value for certain variable types. Not used for this variable type.

Deprecated Properties

Property
None.

Samples

```
{
       "name": "Text Property",
       "type": "text",
       "descr": "API key for the tile provider chosen. OpenStreetMap does not require an API key.",
       "required": false
}
```

See Visual Type Configuration Properties.

ungrouped

The `ungrouped` variable is part of the array of objects you use to define data and non-data values for this visualization.

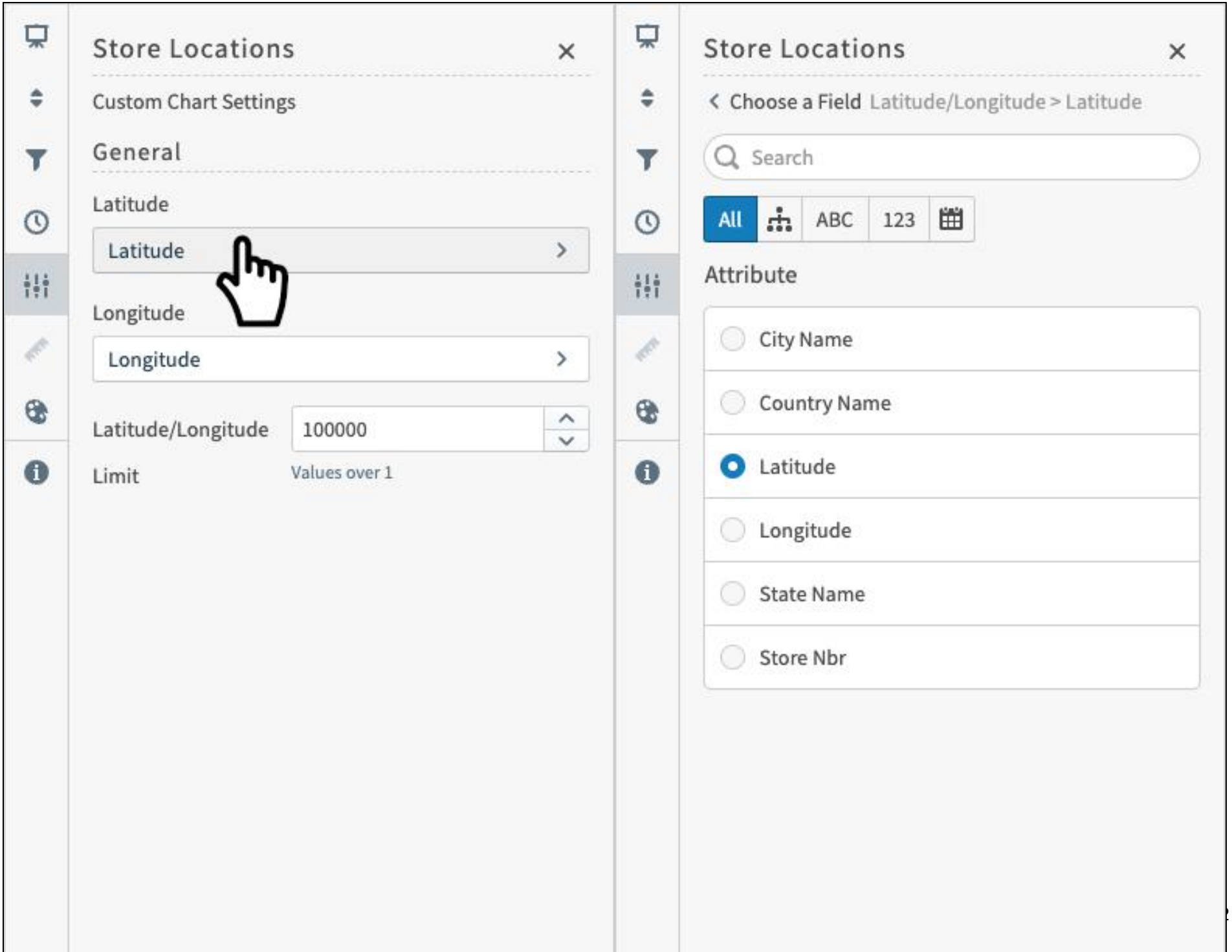
Note: You must be an administrator to manage custom visual types.

Type

```
ungrouped
```

Editor

`ungrouped` variables are represented by a series of buttons to allow users to select a set number of fields matched up with labels. Below the field buttons is a number input to choose a limit to the number of records that should be returned. By default, the labels provided are used to attempt to select the correct fields. If no fields match the labels, the first *n* fields are selected.


Store Locations
Custom Chart Settings
General
Latitude
Latitude
Longitude
Longitude
Latitude/Longitude
100000
Limit
Values over 1
Store Locations
Choose a Field Latitude/Longitude > Latitude
Search
All
ABC
123
Attribute
City Name
Country Name
Latitude
Longitude
State Name
Store Nbr

Variable Specific Properties

Property		Argument	Description
`config`		object	All properties are required.
	`groupLevel`	integer	The number of fields to be selected.
	`groupNames`	array of strings	A list of labels for the fields. The length of this array should match the value set for `groupLevel`.
	`limit`	integer	The number of ungrouped records to return.

Generic Properties

Generic properties apply to many variables.

Property	Argument	Description
`name`	string	The name of the property. It must be unique among all variables on the visualization. This name is used to access the data accessor and/or variable value on the controller provided to the visualization.
`descr`	string	A description of the variable for your reference. This is not displayed in the UI.
`attributeType`	array of strings	A list of field types to include when selecting fields for this variable. Accepted values: ■ `ATTRIBUTE` ■ `TIME` ■ `NUMBER` ■ `INTEGER`
`required`	boolean	Triggers the UI to require a value for certain variable types. Not used for this variable type.

Deprecated Properties

Property
None.

Samples

```
{
        "name": "Latitude/Longitude",
        "type": "ungrouped",
        "descr": "Latitude and Longitude fields",
        "config": {
                "groupLevel": 2,
                "groupNames": [
                        "Latitude",
                        "Longitude"
                        ],
                        "limit": 100000
                        },
                        "required": false
                }
```

See Visual Type Configuration Properties.

ungroupedList

The `ungroupedList`variable is part of the array of objects you use to define data and non-data values for this visualization.

Note: You must be an administrator to manage custom visual types.

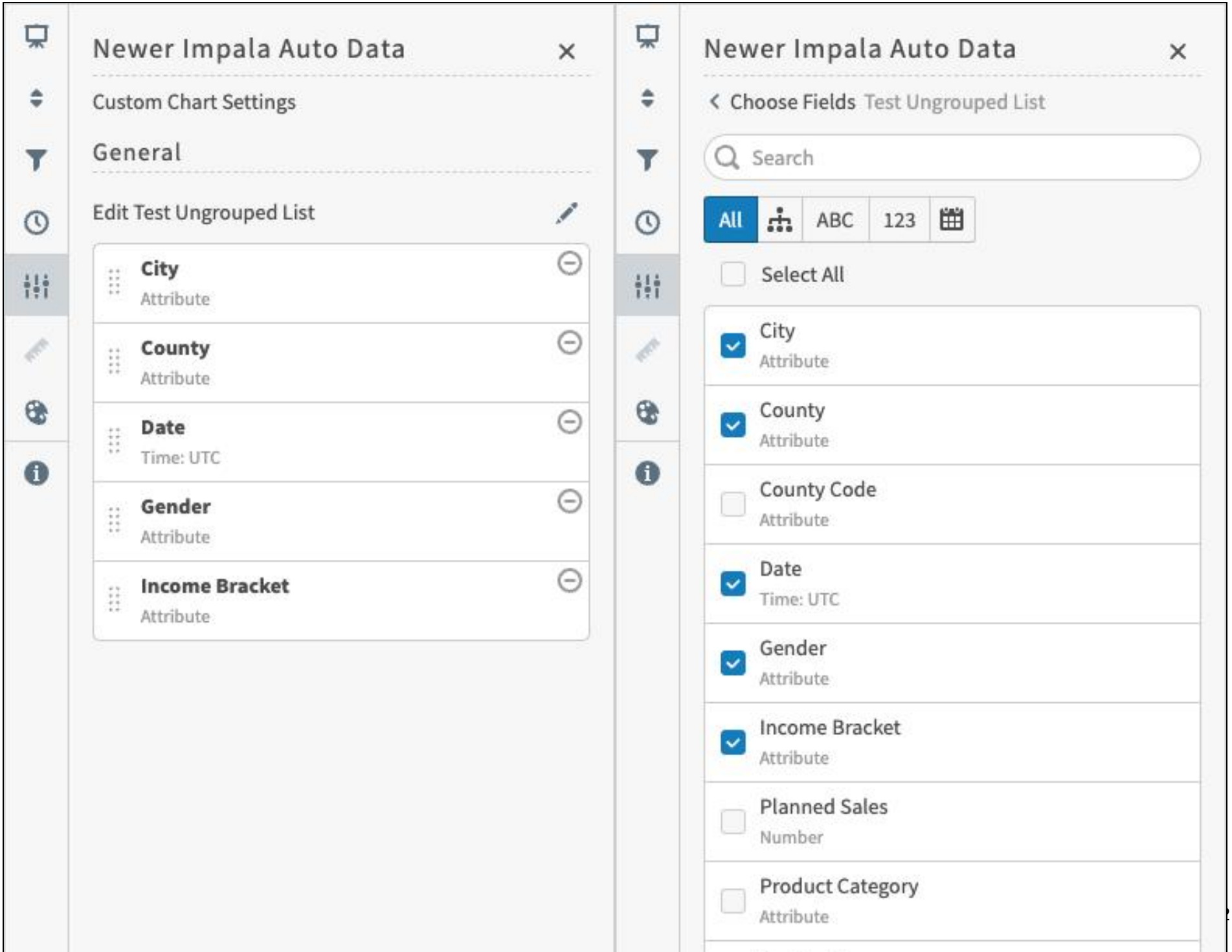
Type

```
ungroupedList
```

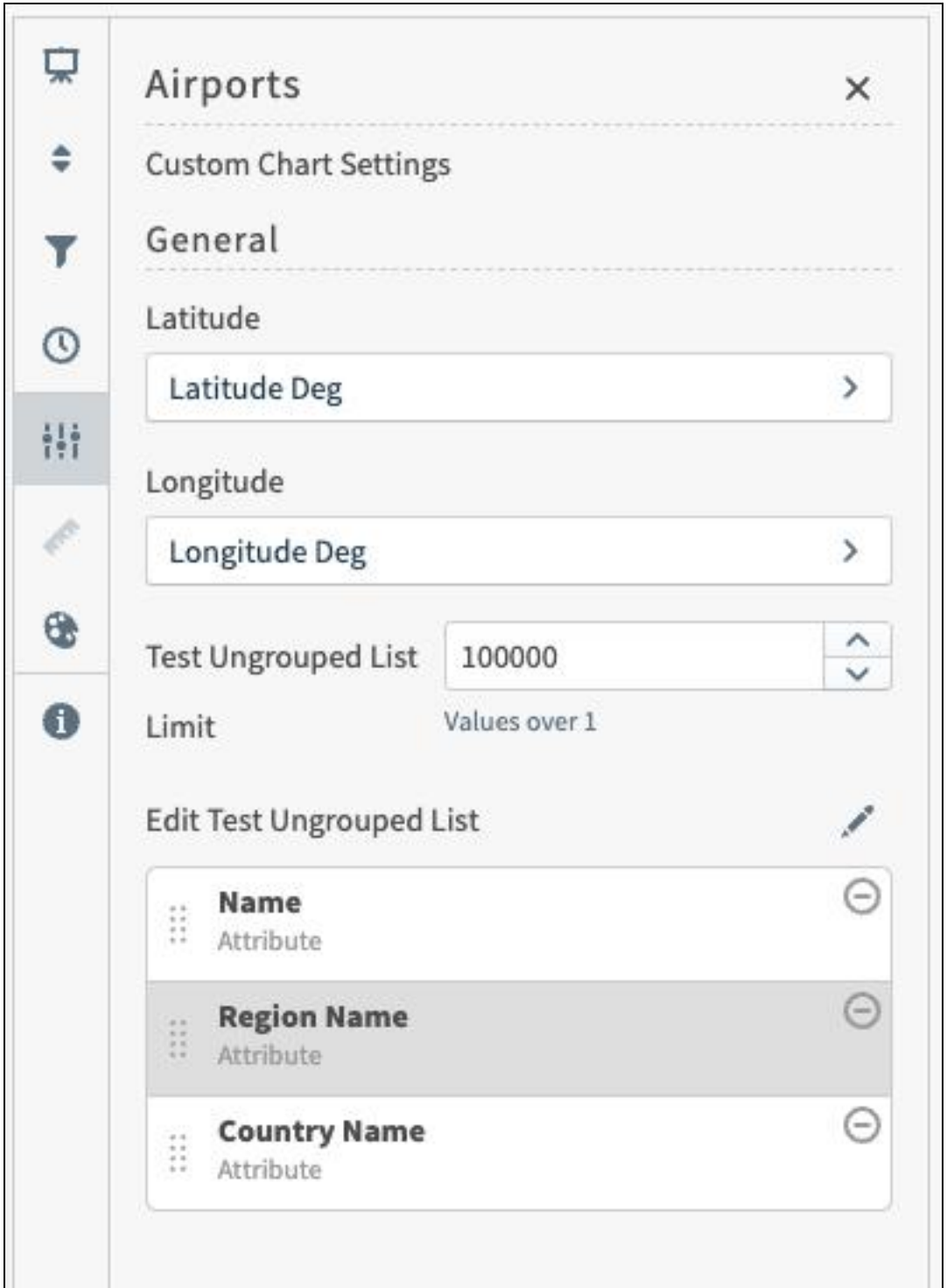
Editor

`ungroupedList` variables are represented by two styles of control depending on the presence or absence of the `groupLevel` and `groupNames` properties in the `config`.

If no `groupLevel` and `groupNames` properties are defined, the control provides only an arrangeable list of fields with an edit button which allows selecting or deselecting which fields should be included.


Newer Impala Auto Data
Custom Chart Settings
General
Edit Test Ungrouped List
City
Attribute
County
Attribute
Date
Time: UTC
Gender
Attribute
Income Bracket
Attribute
Newer Impala Auto Data
Choose Fields Test Ungrouped List
Search
All
ABC
123
Select All
City
Attribute
County
Attribute
County Code
Attribute
Date
Time: UTC
Gender
Attribute
Income Bracket
Attribute
Planned Sales
Number
Product Category
Attribute

If `groupLevel` and `groupNames` are defined in the `config`, the variable is represented by a combination of the UI from the `ungrouped`variable and the UI shown above. This allows you to supply a set number of fields as named properties as well as an undefined number of additional fields for other purposes. When selecting the field for any named properties or for the additional fields list, the fields selected for other uses will not be selectable.


Airports
Custom Chart Settings
General
Latitude
Latitude Deg
Longitude
Longitude Deg
Test Ungrouped List
100000
Limit
Values over 1
Edit Test Ungrouped List
Name
Attribute
Region Name
Attribute
Country Name
Attribute

Variable Specific Properties

Property		Argument	Description
`config`		object	All properties are optional but `groupLevel` and `groupNames` should be provided together if they are provided.
	`groupLevel`	integer	The number of fields to select.
	`groupNames`	array of strings	A list of labels for the fields. The length of this array should match the value provided as `groupLevel`.
	`limit`	integer	The number of ungrouped records that should be returned.
	`additionalFieldsLabel`	string	The label that will be shown above the field list if `groupLevel` and `groupNames` are provided.

Generic Properties

Generic properties apply to many variables.

Property	Argument	Description
`name`	string	The name of the property. It must be unique among all variables on the visualization. This name is used to access the data accessor and/or variable value on the controller provided to the visualization.
`descr`	string	A description of the variable for your reference. This is not displayed in the UI.
`attributeType`	array of strings	A list of field types to include when selecting fields for this variable. Accepted values: ■ `ATTRIBUTE` ■ `TIME` ■ `NUMBER` ■ `INTEGER`
`required`	boolean	Triggers the UI to require a value for certain variable types. Not used for this variable type.

Deprecated Properties

Property
None.

Samples

```
{
        "name": "Test Ungrouped List",
        "type": "ungroupedList",
        "descr": "Select multiple fields for ungrouped data",
        "attributeType": [
                "ATTRIBUTE",
                "INTEGER",
                "NUMBER"
        ],
        "config": {
                "groupLevel": 2,
                "groupNames": [
                        "Latitude",
                        "Longitude"
                ],
                "limit": 100000,
                "additionalFieldsLabel": "Extra Fields"
        }
        "required": false
}
```

See Visual Type Configuration Properties.

Manage UI Themes

Composer supports themes for the UI. Several themes are supplied when you install Composer. See Supplied Themes.

You can define and use your own themes. To manage themes, your Composer user must be assigned to a group with the **Administer Themes** (ROLE_ADMINISTER_THEMES) privilege enabled. See Group Privilege Reference.

Note: At this time, you can only tailor theme colors. Other tailoring properties (such as fonts or font sizes) should not be changed.

Themes are defined, controlled, and managed using the `customization/themes` API endpoint. Any changes you make to the theme are applied for all users in the account. Users in other accounts are not affected.

You can specify a master theme in the theme JSON using the `masterThemeID` property. Master themes are optional, but allow you to identify the Composer-supplied theme from which properties should be inherited by a custom theme, if the properties are not specified in the custom theme JSON. Master themes can only be Composers- supplied themes. See Supplied Themes.

This section covers the following topics:

API documentation is provided with your Composer installation at this link: *`https://<composer-URL>/composer/swagger-ui.html`*.

Manage Alerts

Use alerts to alert your end users when a metric reaches a specified threshold. Use the Composer user interface or API to manage your alert definitions. These definitions describe an alert condition, determine a schedule to evaluate the alert condition, and how notifications are handled when an alert condition is met. SeeManaging Alerts, Create An Alert Definition and Alerts API.

Prerequisites

- Configure the mail service for Composer.
- Configure dashboard alert link redirection.
- Ensure intended recipients have valid email addresses specified in their Composer user definitions.
- Grant the group privilege **Administer Alerts** or **Create Alerts** to users who you want to enable to create or administer alerts. Administrative users are granted these privileges' by default. See Group Privilege Reference.
- Grant users who receive alerts read access to the data sources used in alert conditions. If a user doesn't have read permission for a data source will not receive any alert notifications. The corresponding alert messages also appear in the scheduler report and the job scheduler. See About Source Permissions.

See the following topics:

- Managing Alerts
- Create An Alert Definition
- Alerts API
- Create An Alert Definition - Alerts API
- Configure Dashboard Alert Links
- Alert Definition Object Structure
- Alert Definition Data Query Structures

- Alert Definition Notification Structure
- Alert Definition Examples
- Tracking Scheduled Alert Status

Managing Alerts

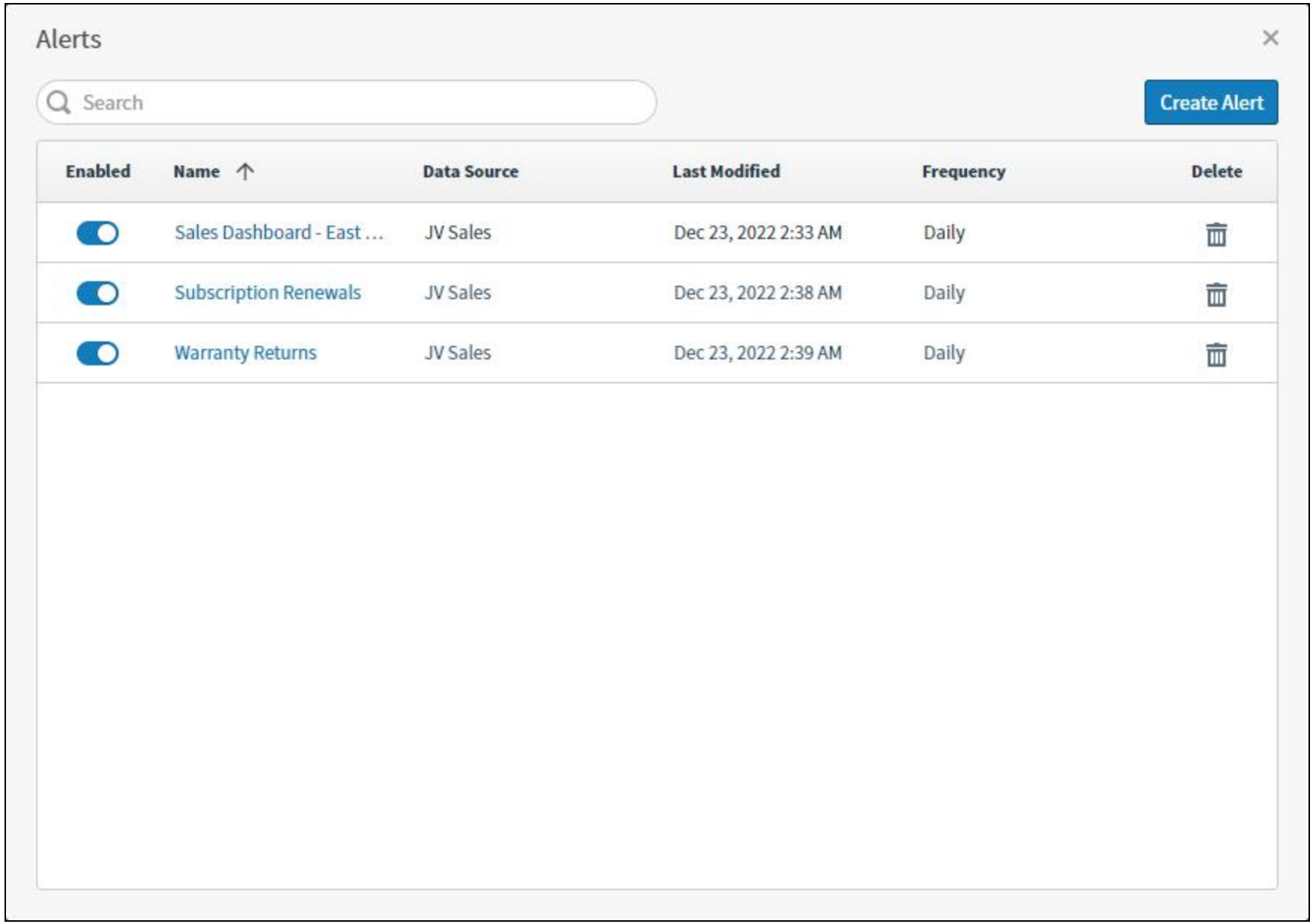
Use the Alerts work area to manage all of the alerts associated with a dashboard. Create, edit, and delete alerts, or temporarily enable and disable alerts as needed.

To manage alerts, you must be logged in as a user belonging to a group with the privilege **Administer Alerts** or **Create Alerts**.

- Users with the privilege **Administer Alerts** and `DATA ACCESS` to the underlying data sources can manage all aspects of alerts associated with a dashboard.
- Users with the privilege **Create Alerts** can only manage the alerts they created for a dashboard.

Access the Alerts Work Area

1. Open or create a dashboard that contains one or more alerts. If you are creating an alert on a new dashboard, save the dashboard first to enable the Manage Alerts option.
2. Select **Manage Alerts** from the dashboard. The Alerts work area opens, listing existing alerts for this dashboard you can manage.



3. Create, edit, and delete alerts, or temporarily enable and disable alerts as needed.

i. Select **Create Alert** to create a new alert. See Create An Alert Definition.

ii. Select the name of an alert to edit the alert. See Edit Alerts.

iii. Select the delete () icon to delete the alert. See Delete Alerts.

iv. Select the toggle for an alert to enable or disable the alert. See Disable And Enable Alerts.

Search Box

Use the search box to filter the alerts shown by Name. For example, if you type a **C** in the search box, only alerts that include the letter *C* in the selected field searched are shown in the working area.

Buttons

The buttons in this work area allow you to create new alerts.

Button	Description
Create Alert	Allows you to create a new alert. See Create An Alert Definition.

The Alerts List

Each column in the table is described below. The Name, Data Source, Last Modified, and Frequency columns are sortable.

Column	Description
Enabled	Indicates if an alert is enabled or disabled. See Disable And Enable Alerts.
Name	The name you assigned to an alert. This name does not need to be unique. Select to edit the alert. See Edit Alerts.
Data Source	The name of the data source used by the visual for an alert.
Last Modified	The time stamp indicating the last date and time the alert was modified. This can indicate the creation date if no modifications have been made since creation, the last time an edit was saved to an alert definition, or the last time an alert was enabled or disabled.
Frequency	How often an alert is run, as indicated in an alert definition's schedule. See Create an Alert Definition .
Delete	Delete an alert. See Delete Alerts.

Create an Alert Definition

Use alerts to alert yourself and other users when a metric reaches a specified threshold. Use the Composer UI to create alert definitions. These definitions describe an alert condition, determine a schedule to evaluate the alert condition, and how notifications are handled when an alert condition is met.

To create an alert definition using the Composer API, see Create An Alert Definition - Alerts API.

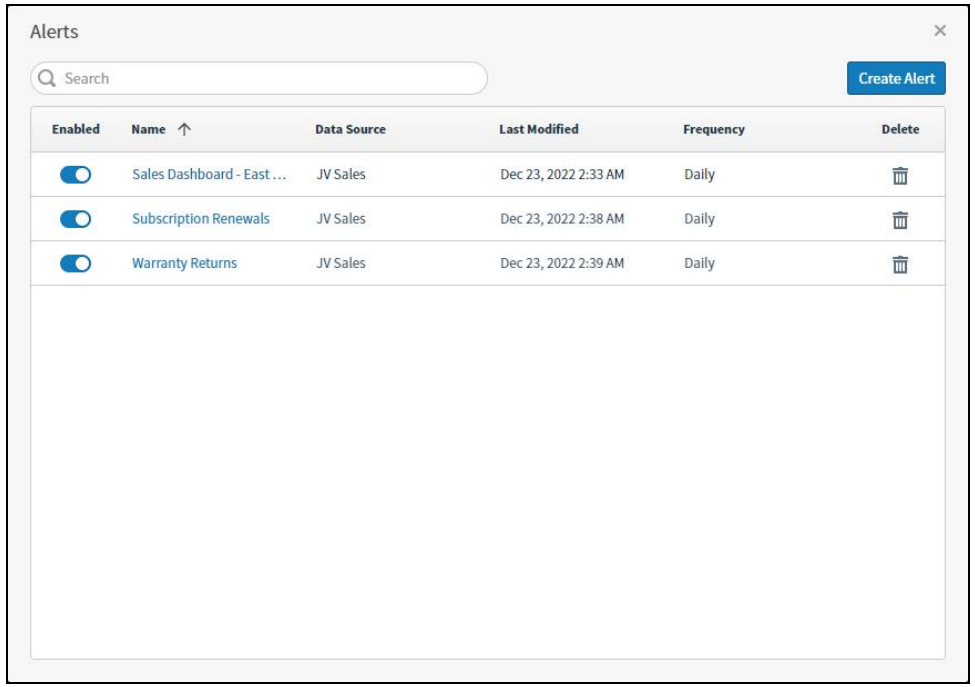
Visuals that support alerts:

- Arc Gauge
- Bars
- Bars: Multiple Metrics
- Donut
- KPI
- Line Trend: Multiple Metrics
- Pie

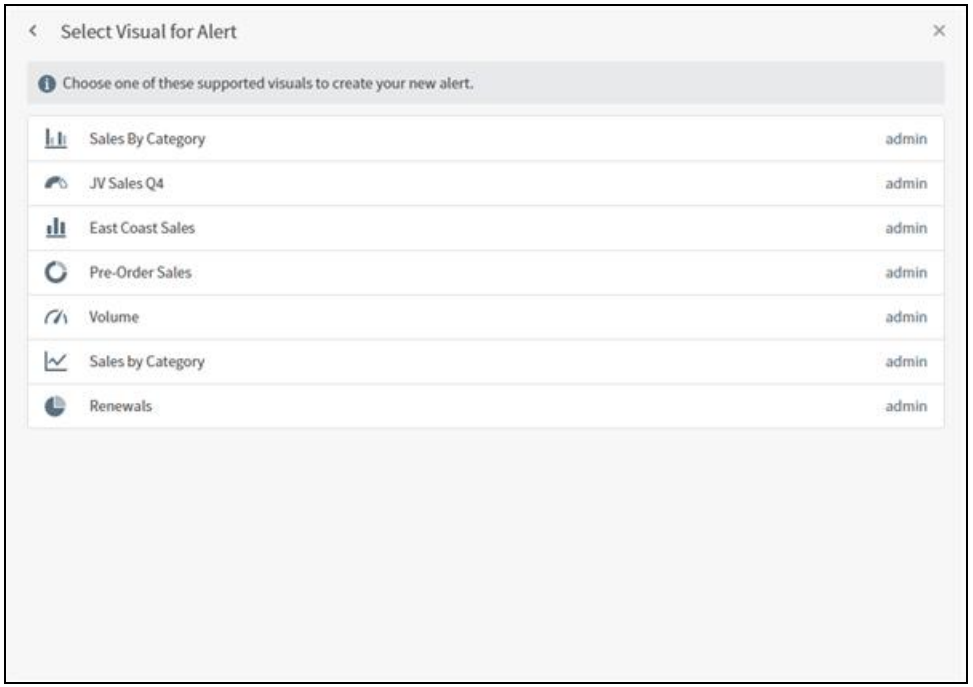
Create an Alert Definition from the Dashboard

To create an alert, you must be logged in as a user belonging to a group with the privilege **Administer Alerts** or **Create Alerts**.

1. Open or create a dashboard that contains one or more visuals that support alerts. If you are creating an alert on a new dashboard, save the dashboard first to enable the Manage Alerts option.
2. Select **Manage Alerts** from the dashboard. The Alerts work area opens, listing existing alerts for this dashboard, if any.



3. Select **Create Alert** to create a new alert. The Select Visual for Alert work area opens.



4. Select a supported visual to create a new alert for that visual. The Create Alert work area opens.
5. Enter a **Name** and optional **Description** for this alert in the Alert Details work area.
6. Accept the default Conditions, or select  to edit and **Apply** to apply your changes.
7. Accept the default Schedule, or select  to edit and **Apply** to apply your changes.
8. Accept the default Notification details, or select  to edit and **Apply** to apply your changes.
9. Select **Save** to save your alert. The Alerts work area opens, including your alert in the list.

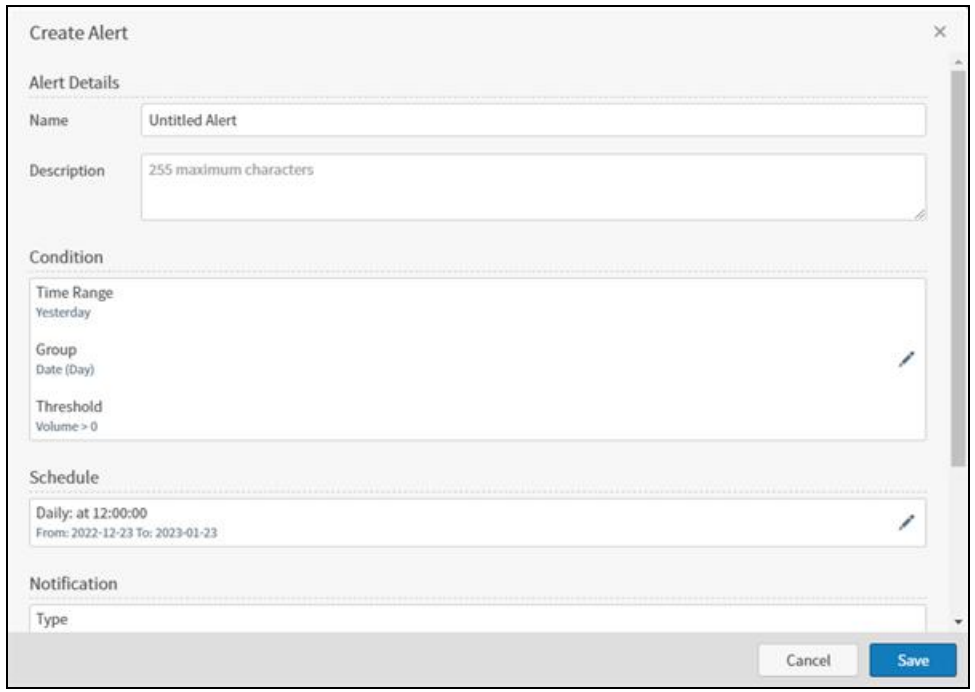
Create an Alert Definition from the Visual Menu

Note: To create an alert, you must be logged in as a user belonging to a group with the privilege **Administer Alerts** or **Create Alerts**.

Note: If you create an alert definition using the Visual Menu, this alert is associated with the dashboard that contains the visual. If you use a visual in another dashboard, the alert does not carry over to the new dashboard.

1. Open or create a dashboard that contains one or more visuals that support alerts. If you are creating an alert on a new dashboard, save the dashboard first to enable the Manage Alerts option.

2. Select **Create Alert** from the visual sidebar menu to create a new alert. The Create Alert work area opens. See Alert Definition Fields And Options for information about adjusting these fields to suit your organization's needs.



3. Enter a **Name** and optional **Description** for this alert in the Alert Details work area.

4. Accept the default Conditions, or select  to edit and **Apply** to apply your changes.
5. Accept the default Schedule, or select  to edit and **Apply** to apply your changes.
6. Accept the default Notification details, or select  to edit and **Apply** to apply your changes.
7. Select **Save** to save your alert.

Alert Definition Fields and Options

Composer populates default field information and defines options related to the alert. This information is summarized in the Create Alert and Edit Alert work areas.

Select  to edit the information for each section.

Alert Details

Field	Description
Name	Enter a name for the alert, shown in the Alerts work area. Must be unique.
Description	Optionally, enter a short description of this alert. 255 character maximum.

Condition

Field	Description
Time Range	Select a time range for this alert from preset time ranges.
Description	Optionally, enter a short description of this alert. 255 character maximum.
Metric	The metric, used in your visual, you are creating this alert condition about.
Group	The attribute, used in your visual, you are creating this alert condition about.
Operator	Select an available operator for this alert. Not all Composer operators may be available.
Value	Define the value of the metric that triggers this alert to be triggered.

Schedule

Field	Description
Frequency	Select a frequency for the scheduled dashboard report using the arrows in the Frequency selection box. Frequencies of **Daily**, **Weekly**, **Monthly**, and **Run Once** are supported. Depending on the frequency you select, additional fields appear.
Run Time	This field only appears if the **Daily**, **Weekly**, or **Monthly** frequencies are selected. Specify the hour and minute of the day at which the alert should be generated and sent. Type the hour of the day on the left side of the colon and the minute of the day on the right side of the colon. Use the arrows in the box to the far right to select AM or PM.
From	This field only appears if the **Daily**, **Weekly**, or **Monthly** frequencies are selected. Select the starting date for the alert. Click in the box to bring up a calendar in which you can select the date.

Field	Description
To	This field only appears if the **Daily**, **Weekly**, or **Monthly** frequencies are selected. Select the ending date for the alert. Click in the box to bring up a calendar in which you can select the date.
Run on	This field only appears if the **Weekly** or **Monthly** frequencies are selected. When **Weekly** is selected, use the arrows in this selection box to select the day of the week on which the alert should run (Sunday, Monday, Tuesday, etc.). When **Monthly** is selected, use the arrows in this selection box to select the date in the month on which the alert should run (valid values range from 1 through 31).
Date	This field only appears if you select the **Run Once** frequency. Select the date for the alert. Click in the box to bring up a calendar in which you can select the date.

Email Notification

Field	Description
To	The To text box contains your user name. Add more recipients here by typing their name or email address in this box. As you type in characters, Composer searches for and returns users that match your entry. Users must be defined in Composer to be added. See Manage Users. You can also set up user attributes and use the Recipient Rules API to specify who your users can see in the recipients list, and select from those users who to send the report to. **Note:** You must have at least one name in this field. Composer displays an error message if this field is blank.
Subject	Specify a subject for the email that will be sent containing the alert. By default, a subject of **Notification from <alert-name>** is used.
Message	Optionally provide a message for the email. By default, a message of **A data threshold was crossed for <alert-name>** is used.

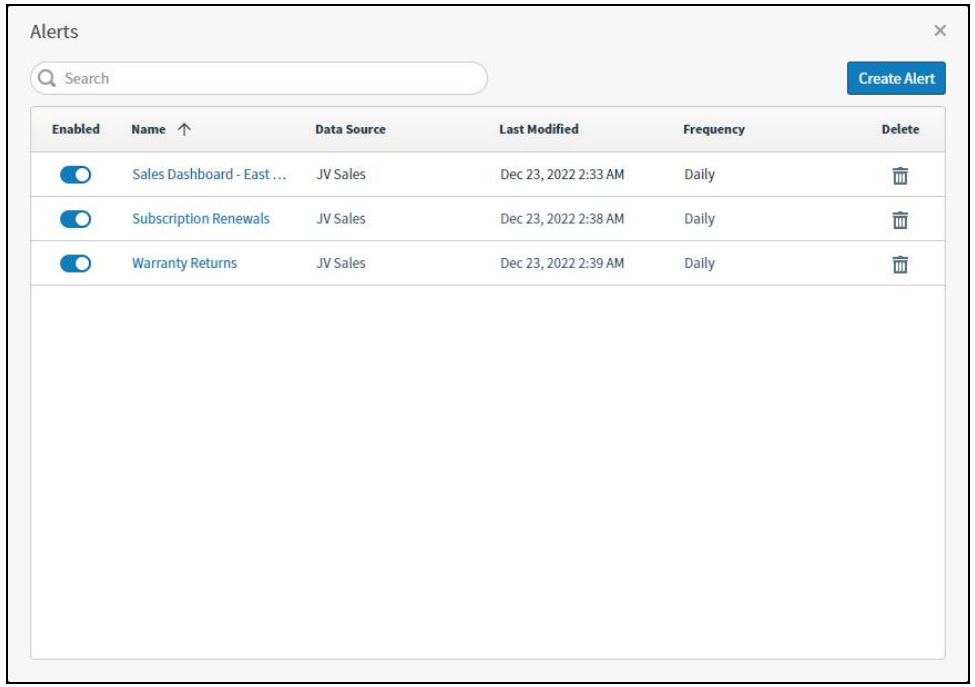
Edit Alerts

Edit alerts directly on the dashboard. To edit alerts, you must be logged in as a user belonging to a group with the privilege **Administer Alerts** or **Create Alerts**.

- Users with the privilege **Administer Alerts** and `DATA ACCESS` to the underlying data sources can manage all aspects of alerts associated with a dashboard.
- Users with the privilege **Create Alerts** can only manage the alerts they created for a dashboard.

Edit an Alert

1. Open a dashboard that contains one or more visuals that support alerts.
2. Select **Manage Alerts** from the dashboard. The Alerts work area opens, listing existing alerts for this dashboard.



3. Select the name of an alert to edit it. The Edit Alert work area opens.

4. You can edit any of the sections of the alert definition as needed. After you select **Apply** to apply your changes to the definition, you must save your edits for the alert.

i. Alert Details : Edit the **Name** and **Description** fields directly here.

ii. Conditions: Select  to open the Edit Condition work area. Edit the conditions for this alert, then select **Apply** to apply your changes.

iii. Schedule: Select  to open the Edit Schedule work area. Edit the schedule for this alert, then select **Apply** to apply your changes.

iv. Notification: Select  to open the Edit Notification work area. Edit the notification information for this alert, then select **Apply** to apply your changes.

Note: See Alert Definition Fields And Options information about adjusting these fields to suit your organization's needs.

5. Select **Save** to save your alert. The Alerts work area opens, including your updated alert in the list.

Disable and Enable Alerts

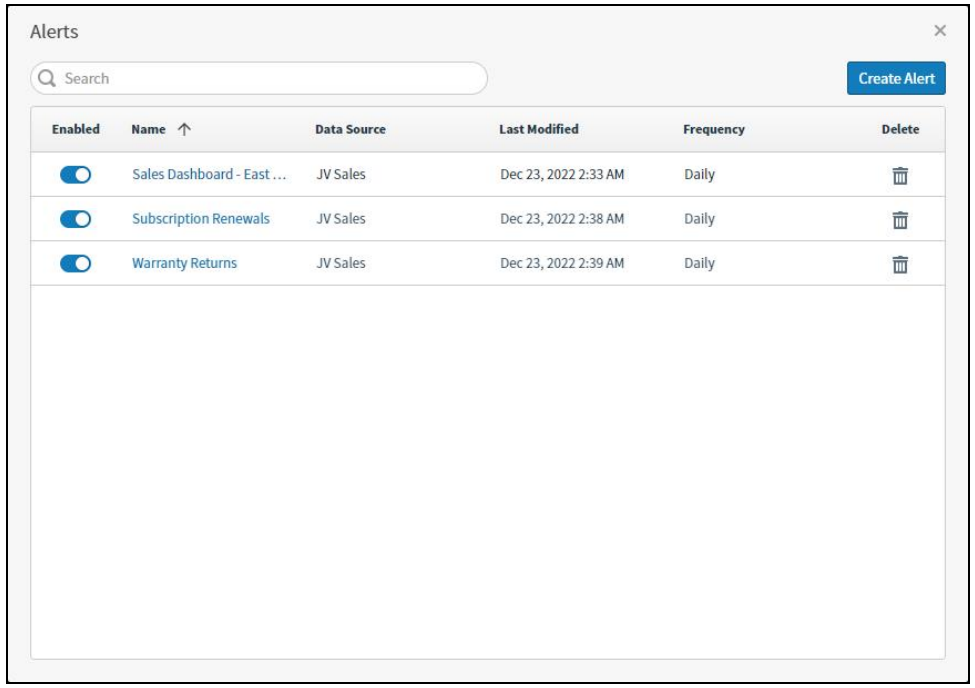
You can easily enable or disable an alert temporarily using the Alerts work area.

To disable alerts, you must be logged in as a user belonging to a group with the privilege **Administer Alerts** or **Create Alerts**.

- Users with the privilege **Administer Alerts** and `DATA ACCESS` to the underlying data sources can manage all aspects of alerts associated with a dashboard.
- Users with the privilege **Create Alerts** can only manage the alerts they created for a dashboard.

Disable an Alert

1. Open a dashboard that contains one or more visuals that support alerts.
2. Select **Manage Alerts** from the dashboard. The Alerts work area opens, listing existing alerts for this dashboard.



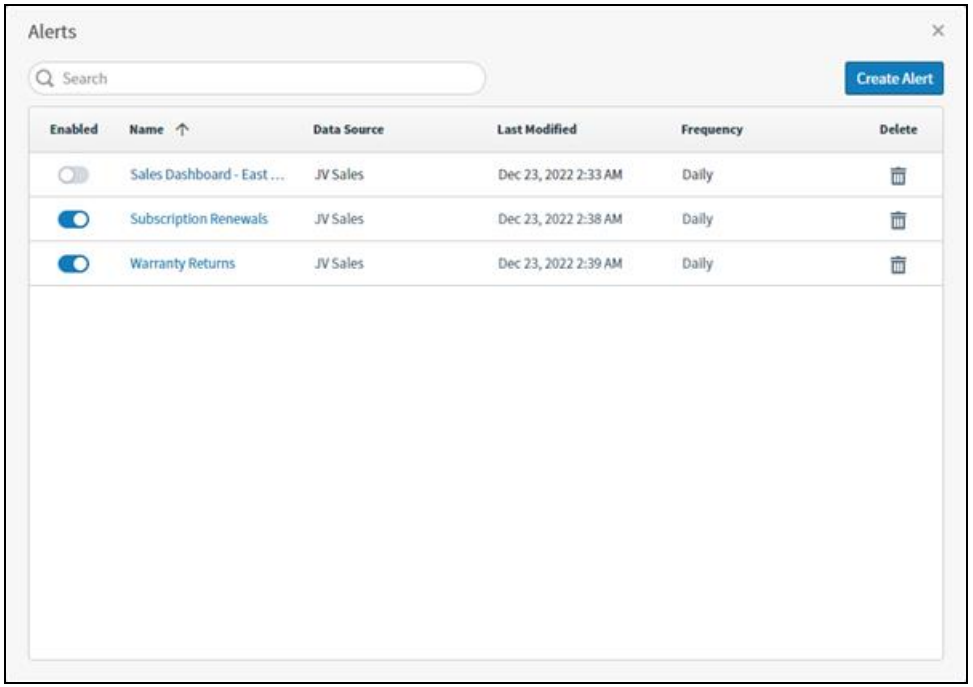
3. Slide the Enabled toggle to the left for an alert to grey the toggle out, disabling the alert.

4. Composer automatically updates the alert definition, and updates the Last Modified field with the current date and time.

   No alerts are run or sent while the alert is disabled.

Enable an Alert

1. Open a dashboard that contains one or more visuals that support alerts.
2. Select **Manage Alerts** from the dashboard. The Alerts work area opens, listing existing alerts for this dashboard.



3. Slide the Enabled toggle to the right for an alert to enable the alert.
4. Composer automatically updates the alert definition, and updates the Last Modified field with the current date and time.

   Alerts will now run and be sent when threshold conditions are met.

Delete Alerts

If you no longer need an alert for a dashboard, use the Alerts work area to delete the alert permanently. Alternatively, disable an alert temporarily if you're not sure if you will need an alert again later.

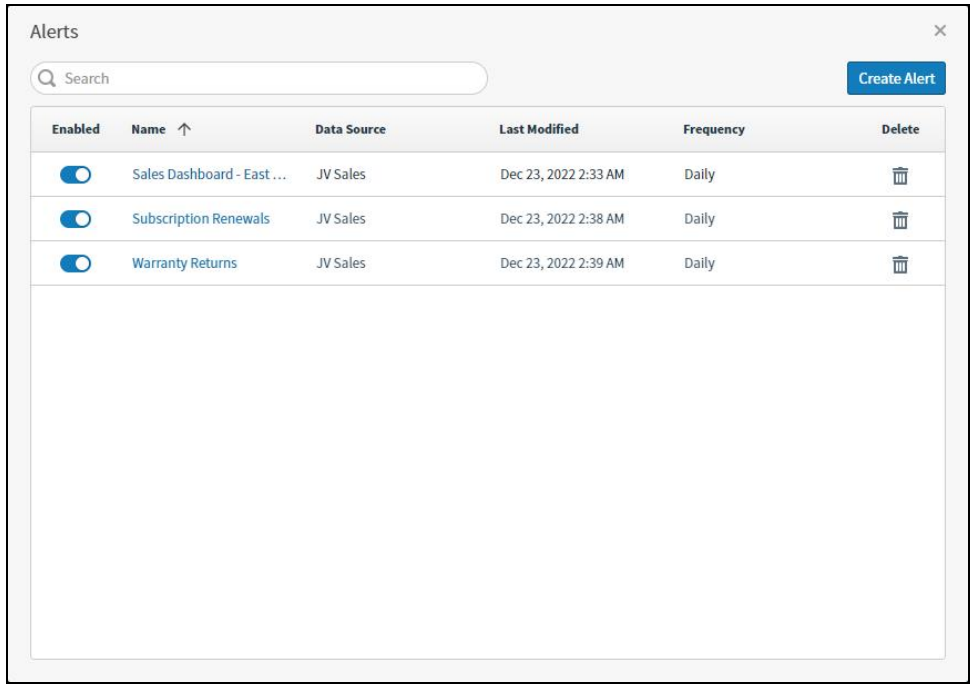
Note: When you delete a dashboard permanently, all alerts associated with the dashboard are deleted as well.

To delete alerts, you must be logged in as a user belonging to a group with the privilege **Administer Alerts** or **Create Alerts**.

- Users with the privilege **Administer Alerts** and `DATA ACCESS` to the underlying data sources can manage all aspects of alerts associated with a dashboard.
- Users with the privilege **Create Alerts** can only manage the alerts they created for a dashboard.

Delete an Alert

1. Open a dashboard that contains one or more visuals that support alerts.
2. Select **Manage Alerts** from the dashboard. The Alerts work area opens, listing existing alerts for this dashboard.



3. Select the delete icon ( ) of an alert to delete it. A confirmation message opens.
4. Select **Delete** to confirm deletion of this alert.

Alerts API

Alerts allow you to alert your end users when a metric reaches a specified threshold. Composer API endpoints can be used to create, update, and delete alert definitions, each describing an alert condition, the schedule by which it is evaluated, and how notification is handled when the alert condition is met.

API support for alerting is performed using the REST API endpoint `/api/alerts`, as described below.

Endpoint	Method	Description
`/api/alerts/{id}`	GET	Returns a specific alert definition, identified by its ID. In a multi tenancy environment, respects Recipients rules.
`/api/alerts/{id}`	PUT	Updates a specific alert definition, identified by its ID. Completely replaces the previous version of the alert definition.
`/api/alerts/{id}`	DELETE	Deletes a specific alert definition, identified by its ID.
`/api/alerts/{id}`	PATCH	Patches a specific alert definition, identified by its ID.
`/api/alerts`	GET	Returns all alert definitions.
`/api/alerts`	POST	Creates a new alert. This endpoint requires the ROLE_CREATE_ALERT privilege.
`/api/alerts/{id}/evaluate-condition`	POST	Evaluates an alert condition in a specific alert definition.

API documentation is provided with your Composer installation at this link: *`https://<composer-URL>/composer/swagger-ui.html`*.

Alert Definition Object Structure

Alert definitions are created by submitting a `POST api/alerts` endpoint with a request body that uses the object structure defined here.

Here is a sample of the general object structure required to create an alert definition:

```
{
  "name": "React on high sales prices",
  "description": "We have a price > $1000",
  "enabled": true,
  "schedule": { ... },
  "condition": { ... },
  "notification": { ... }
}
```

Each object in this structure is described below.

Object	Specifies
`name`	The name of the alert definition.
`description`	A description of the alert definition.
`enabled`	Whether or not the definition is enabled. A value of `true` indicates that it is enabled; `false` indicates that it is not. If an alert is not enabled, it is not scheduled at all.
`schedule`	The frequency by which the alert condition in the definition should be evaluated. Valid values are `ONCE`, `MONTHLY`, `WEEKLY`, and `DAILY`. Each of these values requires additional fields to more specifically identify when the alert condition should be evaluated: ■ `frequency`: Specify `ONCE`, `MONTHLY`, `WEEKLY`, or `DAILY`. Each of these values requires additional fields to more specifically identify when the alert condition should be evaluated: ◦ `ONCE` requires that fields `startDate` and `timeOfDay` be specified. ◦ `MONTHLY` requires that fields `startDate`, `endDate`, `dayOfMonth`, and `timeOfDay` be specified. ◦ `WEEKLY` requires that fields `startDate`, `endDate`, `dayOfWeek`, and `timeOfDay` be specified. ◦ `DAILY` requires that fields `startDate`, `endDate`, and `timeOfDay` be specified. ■ `startDate`: specify the starting date, in `yyyy-mm-dd` format, at which the alert condition should be evaluated. ■ `timeOfDay`: specify the time of day, in `hh:mm:ss` format, at which the alert condition should be evaluated.

Object	Specifies
	■ `endDate`: specify the ending date, in yyyy-mm-dd format, at which the alert condition should be evaluated. ■ `dayOfMonth`: specify the day of the month, using values from 1 through 31, at which the alert should be evaluated. ■ `dayOfWeek`: specify the day of the month, using values Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, or Sunday, at which the alert should be evaluated. Example: `{ ... "schedule": { "frequency": "ONCE", "timeOfDay": "12:30:00", "startDate": "2021-05-14", "endDate": "2021-05-14" } ... }`
`condition`	The alert condition that should be evaluated. The alert condition requires the following fields: ■ `sourceId`: identifies the data source used for the condition query ■ `dataQuery`: a VisQuery object for a raw, KPI, single group by, or multigroup by data query using simple and aggregate filters to define the condition. Data query structures and filters are described in detail in Alert Definition Data Query Structures. ■ `activateAlertWhenData`: Indicates when the alert should be activated. Valid values are `EXISTS` or `NOT_EXISTS`. When `EXISTS` is specified, the data query must determine that data exists to trigger the alert. When `NOT_EXISTS` is specified, the data query must determine that data does not exists to trigger the alert.
`notification`	The notification information for the alert. The notification structure is fully described in detail in Alert Definition Notification Structure. It includes a notification type (only `EMAIL` is currently supported), a subject for the email message, text for the body of the email message, and recipient information.

See Alert Definition Examples for complete examples of some alert definitions.

Alert Definition Data Query Structures

The data query used in an alert definition always specifies a filter condition to be evaluated. Different kinds of data query conditions can be specified: raw, KPI, single group by, or multigroup by queries. Simple filters and aggregate filters are used to establish these query conditions.

This section describes and provides examples for the filter structure and the structures for all query types.

Simple Filters

Note: Because alert conditions are checked periodically, filter conditions must be set properly, especially if the queried data is limited to a dynamic timeframe.

Filters are specified using the following structure. The example specifies a dynamic filter that looks for sales records with sale dates that occurred in the last month.

```
"filters": [
   {
      "operation": "BETWEEN",
      "path": {
         "name": "sale_date"
      },
      "value": [
         "$start_of_month_-1_month",
         "$end_of_month_-1_month"
      ]
```

```
   }
],
```

The following parameters are included in the filter structure.

Parameter	Specifies
`operation`	The filter operation.
`path`	The field in the data source that should be evaluated. Use the `name` field to specify the field name.
`value`	The values or range of values for the filter.

Aggregate Filters

Aggregate filters define thresholds using the `aggregateFilters` object in the following structure. Usually a single threshold is defined, but more complex aggregate filters can be specified. The following aggregate filter example searches for groups (for example, product categories) with 1 to 1000 sales records in which the sum of the sales price fell between $2 and $700,000.

```
"aggregateFilters": [
   {
      "metric": {
         "type": "COUNT"
       },
      "operation": "BETWEEN",
      "value": [
         1,
         1000
      ]
   },
   {
      "metric": {
         "type": "FIELD",
         "field": {
            "name": "price"
         },
         "function": "SUM"
      },
      "operation": "BETWEEN",
      "value": [
         2,
         700000
```

```
      ]
   }
],
```

The following parameters are included in the `aggregateFilters` query structure.

Parameter	Specifies
`metric`	The type of metric.
`operation`	The filter operation.
`value`	The values or range of values for the filter.

Raw Data Query Conditions

A raw data query condition supports only simple filters. The following sample raw query condition searches for sales records with sale dates that occurred in the last month and with prices that equal or exceed $1,000,000.00. When records are found meeting these conditions, and alert notification is sent.

```
{
   ...
   "condition": {
      "sourceId": "<rts-source-id>",
      "dataQuery": {
         "queryType": "RAW",
         "filters": [
            {
               "operation": "BETWEEN",
               "path": {
                  "name": "sale_date"
               },
               "value": [
                  "$start_of_month_-1_month",
                  "$end_of_month_-1_month"
               ]
            },
            {
               "path": {
                  "name": "price"
               },
               "operation": "GE",
               "value": 1000000
```

```
            }
         ]
      },
      "activateAlertWhenData": "EXISTS"
   }
}
```

The following parameters are included in the raw data query structure.

Parameter	Specifies
`queryType`	The type of query. For raw data query conditions, this is always `RAW`. The other possible value for this parameter (but not for raw query conditions) is `AGGREGATE`.
`filters`	The filter conditions for the raw data query.

KPI Data Query Conditions

A KPI query condition is a single-dimension query, without aggregations. Filters can be used to reduce the data to be evaluated. The threshold is defined by aggregate filters.

The following sample KPI query condition searches for sales records from the state of Alabama with sale dates that occurred in the last month and with total planned sales between $1 and $3,035.00. If no records can be found that meet these conditions, an alert notification is sent.

```
{
   ...
   "condition": {
      "sourceId": "<rts-source-id>",
      "dataQuery": {
         "queryType": "AGGREGATE",
         "filters": [
            {
               "operation": "BETWEEN",
               "path": {
                  "name": "sale_date"
               },
               "value": [
                  "$start_of_month_-1_month",
                  "$end_of_month_-1_month"
               ]
            },
```

```
            {
               "path": {
                  "name": "state"
               },
               "operation": "EQ",
               "value": "Alabama"
            }
         ],
         "dimensions": [
            {
               "aggregations": []
            }
         ],
         "aggregateFilters": [
            {
               "metric": {
                  "type": "FIELD",
                  "field": {
                     "name": "plannedsales"
                  },
                  "function": "SUM"
               },
               "operation": "BETWEEN",
               "value": [
                  1.00,
                  3035.00
               ]
            }
         ]
      },
      "activateAlertWhenData": "NOT_EXISTS"
   }
}
```

The following parameters are included in the KPI data query structure.

Parameter	Specifies
`queryType`	The type of query. For KPI data query conditions, this is always `AGGREGATE`. The other possible value for this parameter (but not for KPI query conditions) is `RAW`.
`filters`	The filter conditions for the KPI data query.
`dimensions`	The aggregation type (`TERM` or `TIME`). No aggregation type is supported for KPI data query conditions.
`aggregateFilters`	The threshold conditions for the KPI data query.

Single Group By Queries

Single group by queries are similar to KPI data queries, but add one aggregation. Filters can be used to reduce the data to be evaluated. The threshold is defined by aggregate filters. The window attribute may be defined for the dimension section. But it does not affect the query execution logic.

The following sample single group by query condition searches for sales records from the state of Alabama, aggregated by product group, with sale dates that occurred in the last month and with total planned sales between $1 and $3,035.00. If no records can be found that meet these conditions, an alert notification is sent.

```
{
   ...
   "condition": {
      "sourceId": "<rts-source-id>",
      "dataQuery": {
         "queryType": "AGGREGATE",
         "filters": [
            {
               "operation": "BETWEEN",
               "path": {
                  "name": "sale_date"
               },
               "value": [
                  "$start_of_month_-1_month",
                  "$end_of_month_-1_month"
               ]
            },
            {
               "path": {
                  "name": "state"
               },
               "operation": "EQ",
               "value": "Alabama"
            }
         ],
         "dimensions": [
            {
               "aggregations": [
                  {
                  "type": "TERMS",
                  "field": {
                     "name": "product_group"
                  }
               }
            ]
```

```
            }
         ],
         "aggregateFilters": [
            {
               "metric": {
                  "type": "FIELD",
                  "field": {
                     "name": "planned_sales"
                  },
                  "function": "SUM"
               },
               "operation": "BETWEEN",
               "value": [
                  1.00,
                  3035.00
               ]
            }
         ]
      },
      "activateAlertWhenData": "NOT_EXISTS"
      }
}
```

The following parameters are included in the single group data query structure.

Parameter	Specifies
`queryType`	The type of query. For single group by data query conditions, this is always `AGGREGATE`. The other possible value for this parameter (but not for single group by query conditions) is `RAW`.
`filters`	The filter conditions for the single group by data query.
`dimensions`	The aggregation type (`TERM` or `TIME`). Both `TERM` and `TIME` aggregations are supported. However, time aggregations cannot request the **Include Blanks** function.
`aggregateFilters`	The threshold conditions for the single group by data query.

Multigroup By Queries

Multigroup by queries are the same as Single Group By queries, except that they allow for more than one aggregation.

Note: Top of the Top sorting is not supported; only simple sorting is supported. We recommend that no sorting be specified at all.

The following sample multigroup by query condition searches for sales records from the state of Alabama, aggregated by sales day and city, with sale dates that occurred in the last month and with total planned sales between $1 and $1.050.00. If records can be found that meet these conditions, an alert notification is sent.

```
{
   ...
   "condition": {
   "sourceId": "<rts-source-id>",
   "dataQuery": {
      "queryType": "AGGREGATE",
      "filters": [
         {
            "operation": "BETWEEN",
            "path": {
               "name": "sale_date"
            },
            "value": [
               "$start_of_month_-1_month",
               "$end_of_month_-1_month"
            ]
         },
         {
            "path": {
               "name": "state"
            },
            "operation": "EQ",
            "value": "Alabama"
         }
      ],
      "dimensions": [
         {
            "aggregations": [
               {
                  "type": "TERMS",
                  "field": {
                     "name": "user_city"
                  }
               },
               {
                  "type": "TIME",
                  "field": {
                     "name": "saledate"
                  },
                  "granularity": "DAY"
               }
            ]
```

```
            }
         ],
         "aggregateFilters": [
            {
               "metric": {
                  "type": "FIELD",
                  "field": {
                     "name": "planned_sales"
                  },
                  "function": "SUM"
               },
               "operation": "BETWEEN",
               "value": [
                  1000.00,
                  1050.00
               ]
            }
         ]
      },
      "activateAlertWhenData": "EXISTS"
      }
}
```

The following parameters are included in the multigroup by data query structure.

Parameter	Specifies
`queryType`	The type of query. For multigroup by data query conditions, this is always `AGGREGATE`. The other possible value for this parameter (but not for multigroup by query conditions) is `RAW`.
`filters`	The filter conditions for multigroup by data query.
`dimensions`	The aggregation type (`TERM` or `TIME`). Both `TERM` and `TIME` aggregations are supported. However, time aggregations cannot request the **Include Blanks** function.
`aggregateFilters`	The threshold conditions for the multigroup by data query.

Alert Definition Notification Structure

Here is a sample of the general object structure for the notification object of an alert definition:

```
{
   ...
   "notification": {
      "notificationType": "EMAIL",
      "subject": "RTS: High price",
      "body": "We have a price > $1000",
      "recipients": {
         "users": [
            {
               "id": "user-1",
               "name": "User 1"
            },
            {
               "id": "user-2",
               "name": "User 2"
            }
         ],
         "sendToMe": false
      }
   }
}
```

Each parameter in this structure is described below.

Parameter	Specifies
`notificationType`	The notification type. Currently only `EMAIL` is supported.
`subject`	The subject of the notification email.
`body`	The body text of the notification email.
`recipients`	The users who should be notified. Users must be defined to Composer and all user definitions must have email addresses provided in their user definition. See Add Users. Use the following field pairs to define the recipients for the alert notification: ■ `id`: The Composer user ID (the user ID). This is the only required field (unless the `sendToMe` field is set to `true`). Composer automatically returns the user full name.

Parameter	Specifies
	▪ `name`: The user's login name. This is not required. Repeat user field pairs, as needed, to alert more than one person.
`sendtoMe`	Whether or not the alert notifications should be sent to the author of the alert definition. Specify `true` (send notifications to the author) or `false` (do not send notifications to the author). **Note:** You can elect to send alert notifications only to the author of the alert definition. To do this, eliminate the field pairs in the recipients list and replace it with `sendToMe` set to `true`.

See Alert Definition Examples for complete examples of some alert definitions.

Alert Definition Examples

Two complete examples of an alert definition in JSON format are provided in this section. One uses a raw data query and the other uses a single-group query with dynamic time.

Raw Data Query Example

This example produces an alert when the sales price exceeds $1000. The alert notification is sent to the author of the alert definition.

```
{
   "name": "React on high sales prices",
   "description": "We have a price > $1000",
   "enabled": true,
   "schedule": {
      "frequency": "ONCE",
      "timeOfDay": "12:30:00",
      "startDate": "2021-05-14",
      "endDate": "2021-05-14"
   },
   "condition": {
      "sourceId": "<rts-source-id>",
      "dataQuery": {
         "queryType": "RAW",
         "filters": [
            {
               "path": {
                  "name": "price"
               },
               "operation": "GE",
               "value": 1000
            }
         ]
      },
      "activateAlertWhenData": "EXISTS"
   },
   "notification": {
      "notificationType": "EMAIL",
      "subject": "RTS: High price",
      "body": "We have a price > $1000",
      "recipients": {
         "sendToMe": true
      }
```

```
   }
}
```

Single-Group Query Example

This example produces an alert when sales numbers exceed $100,000 in selected states during the last hour of collected data. The alert notification is sent to author of the alert definition.

```
{
   "name": "Some State has > $100,000 sales (during last hour)",
   "description": "Celebrate good sales",
   "enabled": true,
   "schedule": {
      "frequency": "ONCE",
      "timeOfDay": "12:30:00",
      "startDate": "2021-05-14",
      "endDate": "2021-05-14"
   },
   "condition": {
      "sourceId": "<rts-source-id>",
      "dataQuery": {
         "queryType": "AGGREGATE",
         "dimensions": [
            {
               "aggregations": [
                  {
                     "type": "TERMS",
                     "field": {
                        "name": "userstate"
                     }
                  }
               ]
            }
         ],
         "filters": [
            {
               "operation": "BETWEEN",
               "path": {
                  "name": "ts"
               },
               "value": [
                  "$end_of_data_-1_hour",
```

```
                  "$end_of_data"
               ]
            }
         ],
         "aggregateFilters": [
            {
               "metric": {
                  "type": "FIELD",
                  "field": {
                     "name": "price"
                  },
               "function": "SUM"
            },
            "operation": "GT",
            "value": 100000
         }
      ]
   },
   "activateAlertWhenData": "EXISTS"
 },
 "notification": {
   "notificationType": "EMAIL",
   "subject": "Some State has > $100,000 sales (during last hour)",
   "body": "Wow!",|
   "recipients": {
      "sendToMe": true
    }
  }
}
```

Configure Dashboard Alert Links

When you create an alert definition and set up email notification for alerts, Composer sends recipients your customized alerts message, which includes a link to the dashboard associated with the alert.

Note: In all environments, `{Dashboard.id}` is resolved at run time for an alert's email notification.

Stand Alone Environment

Include the static fully qualified domain name for your Composer environment:

```
alert.dashboard.link.template=${composer.schema}${zoomdata.base-url}/visualization/#{'$'}{Dashboard.id}
```

Where `{composer.schema}=#{${server.ssl.enabled}?'https':'http'}://` and `zoomdata.base-url=${server.address:localhost}:${server.port:8080}${server.servlet.context-path}`.

Tracking Scheduled Alert Status

Use the Console of Refreshing Jobs to view scheduled alert jobs. Optionally, use the `/api/jobs` endpoint to list all jobs, including alert related jobs. For example:

```
http://<host:port>/composer/api/jobs?limit=1000&sort=-lastExecutionEndTime
```

API documentation is provided with your Composer installation at this link: *`https://<composer-URL>/composer/swagger-ui.html`*.

Create an Alert Definition - Alerts API

Use alerts to alert yourself and other users when a metric reaches a specified threshold. Use the Composer alerts endpoints to create alert definitions. These definitions describe an alert condition, determine a schedule to evaluate the alert condition, and how notifications are handled when an alert condition is met. See Create An Alert Definition to create alerts using the Composer UI.

Use the endpoint `/api/alerts` to manage (list, create, update, and delete) alert definitions.

Endpoint	Method	Description
`/api/alerts/{id}`	GET	Returns a specific alert definition, identified by its ID. In a multi tenancy environment, respects Recipients rules.
`/api/alerts/{id}`	PUT	Updates a specific alert definition, identified by its ID. Completely replaces the previous version of the alert definition.
`/api/alerts/{id}`	DELETE	Deletes a specific alert definition, identified by its ID.
`/api/alerts/{id}`	PATCH	Patches a specific alert definition, identified by its ID.
`/api/alerts`	GET	Returns all alert definitions.
`/api/alerts`	POST	Creates a new alert. This endpoint requires the ROLE_CREATE_ALERT privilege.
`/api/alerts/{id}/evaluate-condition`	POST	Evaluates an alert condition in a specific alert definition.

API documentation is provided with your Composer installation at this link: *`https://<composer-URL>/composer/swagger-ui.html`*.

Themes API Endpoint

The API endpoint used to manage themes is `customization/themes`.

Endpoint	Method	Description
`customization/themes`	GET	List all themes.
`customization/themes`	POST	Create a theme.
`customization/themes/<id>`	GET	Review a specific theme by theme ID.
`customization/themes/<id>`	PUT	Update a specific theme.
`customization/themes/<id>`	DELETE	Delete a specific theme.
`customization/themes/<id>`	PATCH	Patch a specific theme.
`customization/themes/activate`	POST	Set the active theme.
`customization/themes/active`	GET	List the active theme.
`customization/themes/name/<name>`	GET	Review a specific theme by name.

API documentation is provided with your Composer installation at this link: *`https://<composer-URL>/composer/swagger-ui.html`*.

Important: Some API endpoints are marked as `experimental` in the Swagger documentation we provide. These endpoints are in the early stages of design and are subject to change. We make no commitment to their stability and may remove them without notice. These experimental endpoints are not recommended for use in production.

List Themes

You can list the themes defined for your Composer environment.

List themes

- Use the `/api/customization/themes`API endpoint in a GET request. For example:

```
curl -X GET "http://<ip-address>:<port>/composer/api/customization/themes" -H "accept: application/vnd.composer.v3+json"
```

  where `<ip-address>` is the IP address or host name of your Composer instance and `<port>` is its port.

  A list of the themes defined to your environment is provided in the body of the response output.

The response output might look like this, with three themes defined: **modern**, **dark**, and **mytheme**. The ID for a theme is generated by Composers when the theme is created in the system.


```
[
  {
    "createdDate": "2020-04-29 10:26:17.135",
    "lastModifiedDate": "2020-04-29 10:26:18.078",
    "id": "dark",
    "name": "dark",
    "system": true
  },
  {
    "createdDate": "2020-04-29 10:26:17.135",
    "lastModifiedDate": "2020-04-29 10:26:18.078",
    "id": "modern",
    "name": "modern",
    "system": true
  },
  {
    "createdByUserID": "<user>",
    "lastModifiedByUserID": "<user>",
    "createdDate": "2020-04-29 13:09:54.855",
    "lastModifiedDate": "2020-04-29 13:24:56.902",
    "id": "5ea97ca22aa608336f499a5b",
    "name": "mytheme",
    "system": false
  }
]
```

Supplied Themes

When Composer is installed, three themes are provided: **composer**, **modern** (light) and **dark**. The **composer** theme is used by default. You can switch to (activate) a different theme when needed.

See Activate A Theme.

Activate a Theme

To set the active theme in your Composer environment, you need to activate it.

Activate a theme

1. Obtain the theme ID. You can do this by listing the themes in your environment. See List Themes.
2. Use the `/api/customization/themes/activate` API endpoint in a POST request to activate the theme. The following request activates the supplied **dark** theme.

```
curl -X POST "http://<ip-address>:<port>/composer/api/customization/themes/activate" -H "accept:
application/vnd.composer.v3+json" -H "Content-Type: application/json" -d "{ \"id\": \"dark\"}"
```

   where `<ip-address>` is the IP address or host name of your Composer instance and `<port>` is its port.

   The theme is set as the active theme for the UI. You must refresh your UI screen to see it.

Review and Download the Theme JSON Code

You can review and download the JSON definition of a theme for the Composer UI. You must know the theme ID or the theme name before you can obtain and review the theme JSON. When you install Composer, three themes are provided with the following IDs and names: **composer**, **modern** and **dark**. The **composer** theme is the same as the **modern** theme.

Review a theme's JSON definition using the theme ID

1. Obtain the theme ID. You can do this by listing the themes in your environment. See List Themes.

2. Use the `/api/customization/themes/<id>` API endpoint in a GET request to obtain the JSON for the theme. The following request obtains the JSON for the supplied **modern** theme. The supplied **modern** theme's ID is modern (the same as its name).

   ```
   curl -X GET "http://<ip-address>:<port>/composer/api/customization/themes/modern" -H "accept:
   application/vnd.composer.v3+json"
   ```

   where `<ip-address>` is the IP address or host name of your Composer instance and <port> is its port.

   The JSON is provided in the response from the request. You can download the JSON and review it.

Review a theme's JSON definition using the theme name

1. Obtain the theme name. You can do this by listing the themes in your environment. See List Themes.

2. Use the `/api/customization/themes/name/<name>` API endpoint in a GET request to obtain the JSON for the theme. The following request obtains the JSON for the theme named **mytheme**.

   ```
   curl -X GET "http://<ip-address>:<port>/composer/api/customization/themes/name/mytheme" -H "accept:
   application/vnd.composer.v3+json"
   ```

   where `<ip-address>` is the IP address or host name of your Composer instance and <port> is its port.

   The JSON is provided in the response from the request. You can download the JSON and review it.

Create a Theme

You can create your own themes to use in the Composer UI.

Create a theme

1. Set up the JSON file that describes your theme. Download and use the JSON files provided with the supplied **modern** and **dark** themes to get started or create your own.

i. To use the JSON files for either the **modern** or **dark** theme as a template for your own, retrieve the theme JSON file using the `/api/customization/themes/name/<name>` API endpoint in a GET request. The following request obtains the JSON for the **modern** theme.

```
curl -X GET "http://<ip-address>:<port>/composer/api/customization/themes/name/modern" -H "accept:
application/vnd.composer.v3+json"
```

where `<ip-address>` is the IP address or host name of your Composer instance and <port> is its port.

The JSON is provided in the response from the request. You can download the JSON and review and alter it as needed.

ii. You can create a JSON file that describes your theme. A sample is provided in Sample Themes JSON File.

2. Use the `/api/customization/themes/` API endpoint in a POST request to create the theme. Use the text of your JSON file as the body (`"content": "<string>"`) of the request. Be sure to remove the `"system": true` and `"id": "<string>"` properties from the JSON file before you use it to create a theme. Composer automatically generates an ID for the theme and sets the value of the `"system"` property when the API request to create the theme is processed.

   If you want to specify a master theme, provide the theme name in the `masterThemeID` property. Master themes are optional, but allow you to identify the Composer-supplied theme from which properties should be inherited by a custom theme, if the properties are not specified in the custom theme JSON. Master themes can only be Composer- supplied themes. See Supplied Themes.

   The following shows a sample request to create a theme based on the supplied **modern** theme, but using the supplied **composer** theme as its master theme.

   Note: At this time, you can only tailor theme colors. Other tailoring properties (such as fonts or font sizes) should not be changed.

```
curl -X POST "<ip-address>:<port>/composer/api/customization/themes" -H "accept: application/vnd.composer.v3+json" -H
"Content-Type: application/json" -d "{ \"masterThemeId\": \"composer\", \"name\": \"<theme-name>\", \"content\": {
<JSONcontent> } }
```

where `<ip-address>` is the IP address or host name of your Composer instance, `<port>` is its port, `<theme-name>` is the name of your new theme, and `<JSONcontent>` is the JSON content for your theme.

After the theme is successfully created, it shows up in the list of available themes when one is generated (see List Themes) and it can be updated and patched (see Update A Theme and Patch A Theme). It can also be deleted (see Delete A Theme). However, it will not be used by the Composer UI until it has been activated (see Activate A Theme).

Update a Theme

When you update a theme, you replace the entire JSON file for an existing theme.

Note: At this time, you can only tailor theme colors. Other tailoring properties (such as fonts or font sizes) should not be changed.

Update a theme

1. Update the JSON file that describes your theme. You can download a copy of it to work with if needed. See Review And Download The Theme JSON Code.
2. Note the ID for your theme. This must be specified separately from the rest of the JSON file in an update request.
3. Remove the ID for your theme from the JSON file.
4. Use the `/api/customization/themes/<id>` API endpoint in a PUT request to update the theme. Use the text of your updated JSON file as the body (`"content" : "<string>"`) of the request, but specify the ID of the theme as a parameter for the endpoint PUT request.

   The following request patches the theme called **mytheme**.

```
curl -X PUT "<ip-address>:<port>/composer/api/customization/themes/<id>" -H "accept: application/vnd.composer.v3+json" -
H "Content-Type: application/json" -d "{ \"createdDate\": \"2020-04-29 10:26:17.135\", \"lastModifiedDate\": \"2020-04-
29 10:26:18.078\", \"name\": \"mytheme\", \"content\": { \"customProperties\": { \"about\": { \"color\":
\"$colors.text\", \"copyright\": \"$colors.muted\" }, \"buttons\": { \"base\": { \"active\": { \"bg\":
\"$colors.intentBaseActive\" }, \"bg\": \"$colors.intentBase\", \"color\": \"$colors.text\", \"hover\": { \"bg\":
\"$colors.intentBaseHover\" } }, \"danger\": { \"active\": { \"bg\": \"$colors.intentDangerActive\" }, \"bg\":
\"$colors.intentDanger\", \"color\": \"$colors.onPrimary\", \"disabled\": { \"bg\": \"$colors.intentDangerDisabled\" },
\"hover\": { \"bg\": \"$colors.intentDangerHover\" } }, \"disabled\": { \"bg\": \"$colors.intentBaseDisabled\" },
\"minimal\": { \"active\": { \"bg\": \"$colors.intentMinimalActive\" }, \"bg\": \"initial\", \"color\":
\"$colors.onSurface\", \"disabled\": { \"color\": \"$colors.intentMinimalDisabled\" }, \"hover\": { \"bg\":
\"$colors.intentMinimalHover\" } }, \"primary\": { \"active\": { \"bg\": \"$colors.intentPrimaryActive\" }, \"bg\":
\"$colors.intentPrimary\", \"color\": \"$colors.onPrimary\", \"disabled\": { \"bg\": \"$colors.intentPrimaryDisabled\"
}, \"hover\": { \"bg\": \"$colors.intentPrimaryHover\" } }, \"success\": { \"active\": { \"bg\":
\"$colors.intentSuccessActive\" }, \"bg\": \"$colors.intentSuccess\", \"color\": \"$colors.onPrimary\", \"disabled\": {
\"bg\": \"$colors.intentSuccessDisabled\" }, \"hover\": { \"bg\": \"$colors.intentSuccessHover\" } }, \"warning\": {
\"active\": { \"bg\": \"$colors.intentWarningActive\" }, \"bg\": \"$colors.intentWarning\", \"color\":
\"$colors.onPrimary\", \"disabled\": { \"bg\": \"$colors.intentWarningDisabled\" }, \"hover\": { \"bg\":
\"$colors.intentWarningHover\" } } }, \"chartLegend\": { \"background\": \"$colors.background\", \"color\":
\"$colors.text\" }, \"chartTooltip\": { \"background\": \"$colors.background\", \"color\": \"$colors.text\" },
\"charts\": { \"base\": { \"axisLabelColor\": \"$colors.text\", \"axisLabelShadow\": \"rgba(255, 255, 255, 0.3)\",
\"axisLineColor\": \"$colors.text\", \"color\": \"$colors.text\", \"pointerColor\": \"$colors.text\",
\"splitLineColor\": \"$colors.text\" } }, \"checkbox\": { \"background\": \"$colors.background\", \"border\":
\"$colors.border\", \"checked\": { \"background\": \"$colors.intentPrimary\", \"border\": \"$colors.intentPrimary\",
```

\"innerBackground\": \"$colors.#fff\" } }, \"colorPalette\": { \"activeIconColor\": \"$colors.text\", \"background\": \"$colors.background\", \"borderColor\": \"$colors.border\", \"draggingBackgroundColor\": \"$colors.background\", \"hover\": { \"background\": \"$colors.intentBaseHover\" }, \"iconColor\": \"$colors.muted\", \"selected\": { \"background\": \"$colors.intentPrimary\" } }, \"dashboard\": { \"background\": \"$colors.background\", \"header\": { \"background\": \"$colors.background\", \"controls\": { \"filterActiveIcon\": \"#68AD45\", \"filterActiveIconHover\": \"#59933b\", \"filterIcon\": \"#939393\", \"filterIconHover\": \"#7a7a7a\" }, \"text\": \"$colors.onSurface\", \"unsavedText\": \"$colors.muted\" } }, \"dashboardList\": { \"background\": \"$colors.background\", \"preview\": { \"background\": \"$colors.surface\", \"border\": \"#E6E6E6\", \"color\": \"#595959\", \"description\": { \"color\": \"#939393\" }, \"details\": { \"propertyColor\": \"#b2b2b2\", \"valueColor\": \"#323232\" }, \"shadow\": \"rgba(0, 0, 0, 0.2) 2px 2px 10px 0px\", \"title\": { \"background\": \"#D2D2D2\", \"color\": \"#595959\" } }, \"sidePane\": { \"background\": \"rgb(89, 89, 89)\", \"color\": \"#d2d2d2\" }, \"topPanel\": { \"background\": \"rgba(230, 230, 230, 0.8)\", \"color\": \"$colors.onBackground\" } }, \"datePicker\": { \"background\": \"$colors.surface\", \"color\": \"$colors.onSurface\", \"divider\": { \"color\": \"$colors.border\" }, \"hover\": { \"active\": { \"background\": \"$colors.intentPrimaryHover\", \"color\": \"$colors.onPrimary\" }, \"background\": \"$colors.intentMinimalHover\", \"color\": \"$colors.onSurface\" } }, \"dialog\": { \"background\": \"$colors.background\", \"color\": \"$colors.text\", \"footerBackground\": \"$colors.backgroundVariant\" }, \"homePage\": { \"color\": \"$colors.text\", \"menuCard\": { \"background\": \"$colors.surface\", \"color\": \"$colors.onSurface\" }, \"title\": \"$colors.muted\" }, \"icons\": { \"base\": { \"color\": \"$colors.intentMinimal\" }, \"danger\": { \"color\": \"$colors.intentDanger\" }, \"primary\": { \"color\": \"$colors.intentPrimary\" }, \"success\": { \"color\": \"$colors.intentSuccess\" }, \"warning\": { \"color\": \"$colors.intentWarning\" } }, \"input\": { \"background\": \"$colors.surface\", \"border\": \"$colors.border\", \"disabled\": { \"background\": \"$colors.intentMinimalDisabled\" }, \"label\": \"$colors.text\", \"placeholder\": \"$colors.muted\", \"text\": \"$colors.text\" }, \"list\": { \"border\": { \"color\": \"$colors.intentBaseActive\" } }, \"loader\": { \"background\": \"rgba(247, 247, 247, 0.75)\" }, \"menu\": { \"active\": { \"bg\": \"$colors.intentBaseActive\", \"color\": \"$colors.text\" }, \"bg\": \"$colors.surface\", \"color\": \"$colors.text\", \"disabled\": \"$colors.intentMinimalDisabled\", \"divider\": { \"color\": \"$colors.border\" }, \"hover\": { \"bg\": \"$colors.intentBaseHover\", \"color\": \"$colors.onSurface\" }, \"icon\": { \"active\": { \"color\": \"$colors.intentSuccess\", \"hover\": { \"color\": \"#59933b\" } }, \"color\": \"$colors.intentMinimal\", \"delete\": { \"hover\": { \"color\": \"$colors.intentWarning\" } }, \"hover\": { \"color\": \"$colors.onSurface\" } } }, \"metaDataPicker\": { \"background\": \"$colors.surface\", \"color\": \"$colors.text\", \"item\": { \"aggrHover\": \"#dedede\", \"border\": \"#dedede\", \"hover\": { \"bg\": \"$colors.intentBaseHover\" } }, \"secondary\": \"$colors.muted\" }, \"modal\": { \"background\": \"$colors.surface\", \"color\": \"$colors.onSurface\" }, \"multiSelect\": { \"background\": \"$colors.surface\", \"color\": \"$colors.text\", \"tagBackground\": \"$colors.backgroundVariant\", \"tagColor\": \"$colors.onBackgroundVariant\" }, \"navbar\": { \"background\": \"$colors.primary\", \"colorInactive\": \"$colors.muted\", \"menu\": { \"background\": \"$colors.primaryVariant\", \"color\": \"$colors.onPrimary\", \"divider\": \"rgba(255, 255, 255, 0.15)\", \"itemActive\": \"$colors.intentPrimary\", \"itemHover\": \"$colors.intentMinimalHover\" }, \"tabActive\": \"rgba(138, 155, 168, 0.3)\", \"tabBorderActive\": \"$colors.secondary\", \"tabFontActive\": \"$colors.onPrimary\", \"tabHover\": \"rgba(138, 155, 168, 0.15)\" }, \"popup\": { \"background\": \"$colors.surface\", \"border\": \"$colors.border\", \"closeBtn\": \"#999999\", \"closeBtnHover\": \"#7a7a7a\", \"color\": \"$colors.text\" }, \"radialMenu\": { \"bg\": \"rgba(0, 0, 0, 0.7)\", \"color\": \"$colors.onPrimary\", \"hover\": { \"bg\": \"#000\" }, \"removeBtn\": { \"bg\": \"#939393\", \"color\": \"$colors.onPrimary\", \"hover\": { \"bg\": \"#E5683A\" } } }, \"radio\": { \"background\": \"$colors.background\", \"border\": \"$colors.border\", \"selected\": { \"background\": \"$colors.intentPrimary\", \"border\": \"$colors.intentPrimary\", \"innerBackground\": \"#fff\" } }, \"resourcesTable\": { \"background\": \"$colors.surface\",

```
\"border\": \"$colors.border\", \"color\": \"$colors.onSurface\", \"header\": { \"background\": \"#e7e7e7 linear-
gradient(180deg,#fafafa,#f0f0f0)\", \"color\": \"$colors.text\" }, \"hover\": { \"background\": \"$colors.background\",
\"color\": \"$colors.text\" } }, \"schedule\": { \"list\": { \"bg\": \"$colors.surface\", \"border\":
\"$colors.border\", \"item\": { \"active\": { \"bg\": \"$colors.intentPrimary\", \"color\": \"$colors.onPrimary\" },
\"bg\": \"$colors.surface\", \"color\": \"$colors.text\", \"hover\": { \"bg\": \"$colors.intentBaseHover\" } } } },
\"select\": { \"background\": \"$colors.surface\", \"border\": \"$colors.border\", \"color\": \"$colors.text\",
\"disabled\": { \"background\": \"$colors.intentMinimalDisabled\" } }, \"tables\": { \"base\": { \"background\":
\"$colors.surface\", \"backgroundOdd\": \"#fcfdfe\", \"border\": \"#BDC3C7\", \"borderSecondary\": \"#d9dcde\",
\"color\": \"$colors.onSurface\", \"heading\": { \"background\": \"$colors.background\", \"color\": \"$colors.text\",
\"colorSecondary\": \"$colors.muted\" }, \"hover\": { \"background\": \"#ecf0f1\" } } }, \"thumbnail\": { \"delete\": {
\"background\": \"$colors.backgroundVariant\", \"border\": \"$colors.border\", \"color\":
\"$colors.onBackgroundVariant\" }, \"fav\": { \"color\": \"#68ad45\" }, \"subTitle\": \"$colors.muted\", \"title\":
\"$colors.text\" }, \"timebar\": { \"backgroundColor\": \"#DEDEDE\", \"backgroundColorHover\": \"rgba
(167,182,194,0.3)\", \"border\": \"$colors.border\", \"scrubber\": { \"animatedTickColor\": \"#b1bfd2\",
\"backgroundColor\": \"#b1bfd2\", \"backgroundColorHover\": \"rgba(167,182,194,0.3)\",
\"datePickerBackgroundColorMaximized\": \"$colors.intentPrimary\", \"datePickerBackgroundColorMinimized\": \"#b1bfd2\",
\"datePickerColorMaximized\": \"$colors.intentPrimary\", \"datePickerColorMinimized\": \"#ffffff\", \"splitterColor\":
\"$colors.border\", \"splitterColorHover\": \"$colors.border\", \"textColor\": \"#333333\", \"textColorHover\":
\"#b1bfd2\", \"tickColor\": \"$colors.intentPrimary\", \"tickColorHover\": \"#b1bfd2\",
\"tooltipDatePickerBackgroundColor\": \"$colors.intentPrimary\", \"tooltipDatePickerColor\": \"#555555\" },
\"textColor\": \"$colors.text\", \"textColorHover\": \"$colors.text\" }, \"toast\": { \"danger\": { \"bg\":
\"$colors.intentDanger\", \"color\": \"$colors.text\" }, \"primary\": { \"bg\": \"$colors.intentPrimary\", \"color\":
\"$colors.text\" }, \"success\": { \"bg\": \"$colors.intentSuccess\", \"color\": \"$colors.text\" }, \"warning\": {
\"bg\": \"$colors.intentWarning\", \"color\": \"$colors.text\" } }, \"tooltip\": { \"background\":
\"$colors.primaryVariant\", \"color\": \"$colors.onPrimary\" }, \"visualEditor\": { \"background\":
\"$colors.background\", \"borderColor\": \"$colors.border\", \"color\": \"$colors.text\", \"list\": { \"background\":
\"$colors.intentBase\", \"borderColor\": \"$colors.border\", \"secondaryBackground\": \"$colors.intentBaseActive\" },
\"secondaryColor\": \"$colors.muted\" }, \"widget\": { \"background\": \"$colors.surface\", \"borderColor\":
\"$colors.surface\", \"iconColor\": \"$colors.muted\", \"label\": { \"background\": \"$colors.background\", \"color\":
\"#323232\", \"hover\": { \"background\": \"#d8d8d8\" } }, \"selected\": { \"borderColor\": \"$colors.accentColor\" },
\"titleColor\": \"$colors.text\" } }, \"variables\": { \"breakpoints\": [ \"320px\", \"568px\", \"768px\", \"1024px\",
\"1200px\" ], \"colors\": { \"accentColor\": \"rgba(19, 124, 189, 0.5)\", \"background\": \"#f0f0f0\",
\"backgroundVariant\": \"#dedede\", \"border\": \"#cccccc\", \"intentBase\": \"#f7f7f7\", \"intentBaseActive\":
\"#dedede\", \"intentBaseDisabled\": \"#e8e8e8\", \"intentBaseHover\": \"#f0f0f0\", \"intentDanger\": \"#db3737\",
\"intentDangerActive\": \"#c23030\", \"intentDangerDisabled\": \"#a82a2a\", \"intentDangerHover\": \"#f55656\",
\"intentMinimal\": \"#5c7080\", \"intentMinimalActive\": \"rgba(115, 134, 148, 0.3)\", \"intentMinimalDisabled\": \"rgba
(167, 182, 194, 0.6)\", \"intentMinimalHover\": \"rgba(167, 182, 194, 0.3)\", \"intentPrimary\": \"#137cbd\",
\"intentPrimaryActive\": \"#0e5a8a\", \"intentPrimaryDisabled\": \"rgba(19, 124, 189, 0.5)\", \"intentPrimaryHover\":
\"#106ba3\", \"intentSuccess\": \"#94b845\", \"intentSuccessActive\": \"#15b371\", \"intentSuccessDisabled\":
\"#0a6640\", \"intentSuccessHover\": \"#3dcc91\", \"intentWarning\": \"#d9822b\", \"intentWarningActive\": \"#bf7326\",
\"intentWarningDisabled\": \"#a66321\", \"intentWarningHover\": \"#f29d49\", \"muted\": \"#999999\", \"onBackground\":
\"#182026\", \"onBackgroundVariant\": \"#4a4a4a\", \"onPrimary\": \"#fff\", \"onSurface\": \"#182026\", \"primary\":
\"#182026\", \"primaryVariant\": \"#30404d\", \"secondary\": \"#94b845\", \"surface\": \"#fff\", \"text\": \"#4a4a4a\"
```

```
}, \"fontSizes\": [ 12, 14, 16, 18, 24, 32, 48, 64, 72 ], \"fontWeights\": { \"bold\": 700, \"heading\": 500,
\"lightest\": 100, \"normal\": 400 }, \"fonts\": { \"body\": \"\\\"Source Sans Pro\\\", system-ui, sans-serif\",
\"heading\": \"\\\"Source Sans Pro\\\", system-ui, sans-serif\", \"monospace\": \"Monaco, Menlo, \\\"Ubuntu Mono\\\",
Consolas, source-code-pro, monospace\" }, \"lineHeights\": { \"body\": 1.25, \"heading\": 1.125 }, \"links\": {
\"primary\": { \"color\": \"intentPrimary\" } }, \"radii\": { \"default\": 3, \"lg\": 3.6, \"none\": 0, \"pill\": 600,
\"sm\": 2.4 }, \"shadows\": {}, \"sizes\": { \"lg\": 736, \"md\": 532, \"sm\": 288, \"xl\": 960, \"xxl\": 1136 },
\"space\": [ 0, 4, 8, 16, 32, 64, 128, 256, 512 ] } }}"
```

where <ip-address> is the IP address or host name of your Composer instance, <port> is its port, and <id> is the theme ID.

If the theme is the active theme, refresh the Composer UI to see the theme changes. If the theme is not the active theme, activate it to see the theme changes in the UI.

Patch a Theme

When you patch a theme, you replace small sections of an existing theme.

Note: At this time, you can only tailor theme colors. Other tailoring properties (such as fonts or font sizes) should not be changed.

Patch a theme

1. Identify the section of the JSON file that you want to patch. You can download a copy of it to work with if needed. See Review And Download The Theme JSON Code. The section that you select to patch does not need to include an entire section of the file, but it must be clear what you are changing and it must be well-formed. For example, suppose the following color variables are used in your JSON file but you only want to patch the setting for the `background` color variable.

```
{
"content":{
  "variables":{
     "colors":{
        "text":"#4a4a4a",
        "muted":"#999999",
        "border":"#cccccc",
        "background":"#f7f7f7",
        "onBackground":"#182026",
        "backgroundVariant":"#dedede",
        "onBackgroundVariant":"#4a4a4a",
        "surface":"#fff",
        "onSurface":"#182026",
        "primary":"#182026",
        "onPrimary":"#fff",
        "primaryVariant":"#30404d",
        "secondary":"#94b845",
        "accentColor":"rgba(19, 124, 189, 0.5)",
        "intentBase":"#f7f7f7",
        "intentBaseHover":"#f0f0f0",
        "intentBaseActive":"#dedede",
        "intentPrimary":"#137cbd",
        "intentPrimaryHover":"#106ba3",
        "intentPrimaryActive":"#0e5a8a",
        "intentPrimaryDisabled":"rgba(19, 124, 189, 0.5)",
        "intentSuccess":"#94b845",
        "intentWarning":"#d9822b",
        "intentDanger":"#db3737",
        "intentMinimal":"#5c7080",
        "intentMinimalHover":"rgba(167, 182, 194, 0.3)",
        "intentMinimalActive":"rgba(115, 134, 148, 0.3)",
```

```
        "intentMinimalDisabled":"rgba(167, 182, 194, 0.6)"
     },
...
...
},
"createdByUserID": "string",
"createdDate": "2020-04-28 19:40:22.369",
"id": "string",
"lastModifiedByUserID": "string",
"lastModifiedDate": "2020-04-28 19:40:22.543",
"name": "mytheme"
}
```

To patch the setting for the `background` variable, you would use this snippet of code when you submit the API request later in these steps.

```
{
"content":{
  "variables":{
     "colors":{
        "background":"#f00"
     }
  }
},
"createdByUserID": "string",
"createdDate": "2020-04-28 19:40:22.369",
"id": "string",
"lastModifiedByUserID": "string",
"lastModifiedDate": "2020-04-28 19:40:22.543",
"name": "mytheme"
}
```

2. Note the ID for your theme. This must be specified separately from the rest of the JSON file in a patch request.

3. Use the `/api/customization/themes/<id>` API endpoint in a PATCH request to update the theme. Use snippet of JSON code as the body (`"content" : "<string>"`) of the request, but specify the ID of the theme as a parameter for the endpoint PATCH request.

   The following request patches the theme called **mytheme**.

```
curl -X PATCH "http://<ip-address>:<port>/composer/api/customization/themes/<id>" -H "Content-Type: application/json" -d
"{ \"content\": {\"variables\": { \"colors\": { \"background\": \"#f00\"}}, \"createdByUserID\": \"string\",
\"createdDate\": \"2020-04-29T12:21:07.189Z\", \"id\": \"string\", \"lastModifiedByUserID\": \"string\",
\"lastModifiedDate\": \"2020-04-29T12:21:07.189Z\", \"name\": \"mytheme\"
```

where `<ip-address>` is the IP address or host name of your Composer instance, `<port>` is its port, and `<id>` is the theme ID.

If the theme is the active theme, refresh the Composer UI to see the theme changes. If the theme is not the active theme, activate it to see the theme changes in the UI.

Delete a Theme

You can delete the JSON definition of a theme for the Composer UI. You must know the theme ID before you can delete it.

Delete a theme

1. Obtain the theme ID. You can do this by listing the themes in your environment. See List Themes.
2. Use the `/api/customization/themes/<id>` API endpoint in a DELETE request to delete the theme. The following request deletes the theme named **mytheme**.

```
curl -X DELETE "http://<ip-address>:<port>/composer/api/customization/themes/mytheme" -H "accept:
application/vnd.composer.v3+json"
```

   where `<ip-address>` is the IP address or host name of your Composer instance and `<port>` is its port.

   The theme is deleted.

Sample Themes JSON File

Here is a sample JSON file for themes. Use it as a template to start creating your own themes. See Create A Theme.

Note: At this time, you can only tailor theme colors. Other tailoring properties (such as fonts or font sizes) should not be changed.

You can also download the JSON files provided with the supplied **modern** and **dark** themes and use those as templates for your own themes. See Review And Download The Theme JSON Code.

Be sure to remove the `"system": true` and `"id": "<string>"` properties from the JSON file before you use it to create a theme. Composer automatically generates an ID for the theme and sets the value of the `"system"` property when the API request to create the theme is processed.

You can specify a master theme in the theme JSON using the `masterThemeID` property. Master themes are optional, but allow you to identify the Composer-supplied theme from which properties should be inherited by a custom theme, if the properties are not specified in the custom theme JSON. Master themes can only be Composer- supplied themes. See Supplied Themes.

Finally, if you are changing the colors used for a specific visual style or a table, use the following substitutions (shown in blue in the sample):

Substitution	Description
<visual-type>	Specify one of the following: `UBER_BARS`, `SCATTERPLOT`, `PIE`, `MULTI_METRIC_BARS`, `LINE_AND_BARS`, `HISTOGRAM`, `HEAT_MAP`, `FLOATING_BUBBLES`, `DONUT`, `BOX_PLOT`, or `LINE_CHART`.
<table-type>	Specify one of the following: `RAW_DATA_TABLE` or `PIVOT_TABLE`.
<charts-color-parameter>	Specify one of the base color parameters from the `"charts"` section of the JSON file to change for the selected **<visual-type>**. For example, `"axisLabelColor"`.
<tables-color-parameter>	Specify one of the base color parameters from the `"tables"` section of the JSON file to change for the selected **<table-type>**. For example, `"background"`.
<newsetting>	Specify the new color setting for the **<charts-color-parameter>** or the **<tables-color-parameter>**.
<your-palette>	Specify a name for your color palette, if you choose to add your own, personalized, palette to the environment. Palette names can be referenced in the `palette` property in the `customProperties` section of the JSON under `charts`. They can also be referenced in the `palette` property for individual visual styles (**<visual-type>** sections) under `charts`.
<hexcolor*n*>	Specify the hexadecimal representation of up to 9 colors (where `n` ranges from 1 to 9) for your personalized color palette.
<color-palette-name>	Specify the name of a supplied color palette or a color palette defined in this JSON file.

You can specify color settings for more than one visual style or table type. You can also specify more than one color setting for each style or type.

If you are not changing the colors for a specific visual style or table, you can safely remove the `<visual-type>` and `<table-type>` sections in this sample JSON, as appropriate.

Here is a sample themes JSON file:

```
{
"id": "<string>",
"masterThemeId": "<supplied-theme-name>",
"name": "mytheme",
"system": true,
"content": {
  "variables":{
     "colors":{
        "accentColor": "rgba(19, 124, 189, 0.5)",
        "background": "#f7f7f7",
        "backgroundVariant": "#dedede",
        "border": "#cccccc",
        "brandColor": "#182C44",
        "intentBase": "#f7f7f7",
        "intentBaseActive": "#dedede",
        "intentBaseDisabled": "#e8e8e8",
        "intentBaseHover": "#f0f0f0",
        "intentDanger": "#db3737",
        "intentDangerActive": "#c23030",
        "intentDangerDisabled": "#a82a2a",
        "intentDangerHover": "#f55656",
        "intentMinimal": "#5c7080",
        "intentMinimalActive": "rgba(115, 134, 148, 0.3)",
        "intentMinimalDisabled": "rgba(167, 182, 194, 0.6)",
        "intentMinimalHover": "rgba(167, 182, 194, 0.3)",
        "intentPrimary": "#137cbd",
        "intentPrimaryActive": "#0e5a8a",
        "intentPrimaryDisabled": "rgba(19, 124, 189, 0.5)",
        "intentPrimaryHover": "#106ba3",
        "intentSuccess": "#89bb40",
        "intentSuccessActive": "#15b371",
        "intentSuccessDisabled": "#0a6640",
        "intentSuccessHover": "#3dcc91",
        "intentWarning": "#d9822b",
        "intentWarningActive": "#bf7326",
        "intentWarningDisabled": "#a66321",
        "intentWarningHover": "#f29d49",
        "linkColor": "#106ba3",
        "muted": "#999999",
        "onBackground": "#1d384b",
        "onBackgroundVariant": "#4a4a4a",
        "onPrimary": "#fff",
        "onSurface": "#1d384b",
        "primary": "#1d384b",
```

```
      "primaryVariant": "#30404d",
      "secondary": "#89bb40",
      "surface": "#fff",
      "text": "#4a4a4a"
   },
   "breakpoints":[
      "320px",
      "568px",
      "768px",
      "1024px",
      "1200px"
   ],
   "space":[
      0,
      4,
      8,
      16,
      32,
      64,
      128,
      256,
      512
   ],
   "fonts":{
      "body":"\"Source Sans Pro\", system-ui, sans-serif",
      "heading":"\"Source Sans Pro\", system-ui, sans-serif",
      "monospace":"Monaco, Menlo, \"Ubuntu Mono\", Consolas, source-code-pro, monospace"
   },
   "fontSizes":[
      12,
      14,
      16,
      18,
      24,
      32,
      48,
      64,
      72
   ],
   "fontWeights":{
      "lightest":100,
      "normal":400,
      "heading":500,
      "bold":700
   },
```

```
"lineHeights":{
   "body":1.25,
   "heading":1.125
},
"sizes":{
   "sm":288,
   "md":532,
   "lg":736,
   "xl":960,
   "xxl":1136
},
"shadows":{
},
"radii":{
   "none":0,
   "default":3,
   "sm":2.4,
   "lg":3.6,
   "pill":600
},
"links":{
   "primary":{
      "color":"intentPrimary"
   }
}
"palettes": {
   "DefaultSequential": {
      "2": ["#ffc65f", "#0096b6"],
      "3": ["#ffc65f", "#9eb778", "#0096b6"],
      "4": ["#ffd27c", "#ccbd67", "#7eb184", "#008db6"],
      "5": ["#ffd27c", "#ccbd67", "#7eb184", "#0096b6", "#0082b5"],
      "6": ["#ffd27c", "#efc15e", "#9eb778", "#7eb184", "#0096b6", "#0082b5"],
      "7": ["#ffd27c", "#efc15e", "#9eb778", "#7eb184", "#43a79b", "#008db6", "#097bb1"],
      "8": ["#ffdc9c", "#ffc65f", "#efc15e", "#9eb778", "#7eb184", "#43a79b", "#008db6", "#097bb1"],
      "9": ["#ffdc9c", "#ffc65f", "#efc15e", "#9eb778", "#7eb184", "#43a79b", "#008db6", "#0082b5", "#1473ac"]
   },
   "<your-palette>": {
      "2": ["<hexcolor1>", "<hexcolor2>"],
      "3": ["<hexcolor1>", "<hexcolor2>", "<hexcolor3>"],
      "4": ["<hexcolor1>", "<hexcolor2>", "<hexcolor3>", "<hexcolor4>"],
      "5": ["<hexcolor1>", "<hexcolor2>", "<hexcolor3>", "<hexcolor4>", "<hexcolor5>"],
      "6": ["<hexcolor1>", "<hexcolor2>", "<hexcolor3>", "<hexcolor4>", "<hexcolor5>", "<hexcolor6>"],
      "7": ["<hexcolor1>", "<hexcolor2>", "<hexcolor3>", "<hexcolor4>", "<hexcolor5>", "<hexcolor6>", "<hexcolor7>"],
      "8": ["<hexcolor1>", "<hexcolor2>", "<hexcolor3>", "<hexcolor4>", "<hexcolor5>", "<hexcolor6>", "<hexcolor7>",
```

```
"<hexcolor8>"],
          "9": ["<hexcolor1>", "<hexcolor2>", "<hexcolor3>", "<hexcolor4>", "<hexcolor5>", "<hexcolor6>", "<hexcolor7>",
"<hexcolor8>", "<hexcolor9>"]
        }
     }
  },
  "customProperties":{
     "navbar":{
        "background":"$colors.primary",
        "colorInactive":"$colors.muted",
        "tabFontActive":"$colors.onPrimary",
        "tabHover":"rgba(138, 155, 168, 0.15)",
        "tabActive":"rgba(138, 155, 168, 0.3)",
        "tabBorderActive":"$colors.secondary",
        "menu":{
           "background":"$colors.primaryVariant",
           "color":"$colors.onPrimary",
           "itemActive":"$colors.intentPrimary",
           "itemHover":"$colors.intentMinimalHover",
           "divider":"rgba(255, 255, 255, 0.15)"
        }
     },
     "homePage":{
        "title":"$colors.muted",
        "color":"$colors.text",
        "menuCard":{
           "background":"$colors.surface",
           "color":"$colors.onSurface"
        }
     },
     "dashboard":{
        "background":"$colors.background",
        "header":{
           "background":"$colors.background",
           "text":"$colors.onSurface",
           "unsavedText":"$colors.muted",
           "controls":{
              "filterIcon":"#939393",
              "filterIconHover":"#7a7a7a",
              "filterActiveIcon":"#68AD45",
              "filterActiveIconHover":"#59933b"
           }
        }
     },
```

```
"dashboardList": {
   "background": "$colors.background",
   "linkColor": "$colors.linkColor",
   "preview": {
       "background": "$colors.surface",
       "border": "#E6E6E6",
       "color": "#595959",
       "description": {
           "color": "#939393"
       },
       "details": {
           "propertyColor": "#b2b2b2",
           "valueColor": "#323232"
       },
       "shadow": "rgba(0, 0, 0, 0.2) 2px 2px 10px 0px",
       "title": {
           "background": "#D2D2D2",
           "color": "#595959"
       }
}
"input":{
   "background":"$colors.surface",
   "text":"$colors.text",
   "label":"$colors.text",
   "border":"$colors.border",
   "placeholder":"$colors.muted",
   "disabled":{
      "background":"$colors.intentMinimalDisabled"
   }
},
"multiSelect":{
   "background":"$colors.background",
   "color":"$colors.text",
   "tagBackground":"$colors.backgroundVariant",
   "tagColor":"$colors.onBackgroundVariant"
},
"datePicker":{
   "background":"$colors.surface",
   "color":"$colors.onSurface",
   "hover":{
      "color":"$colors.onSurface",
      "background":"$colors.intentMinimalHover",
      "active":{
         "background":"$colors.intentPrimaryHover"
      }
```

```
      },
      "divider":{
         "color":"$colors.border"
      }
   },
   "tooltip":{
      "background":"$colors.primaryVariant",
      "color":"$colors.onPrimary"
   },
   "radio":{
      "background":"$colors.surface",
      "border":"$colors.border",
      "selected":{
         "background":"$colors.intentPrimary",
         "border":"$colors.intentPrimary",
         "innerBackground":"#fff"
      }
   },
   "buttons":{
      "base":{
         "bg":"$colors.intentBase",
         "color":"$colors.text",
         "hover":{
            "bg":"$colors.intentBaseHover"
         },
         "active":{
            "bg":"$colors.intentBaseActive"
         }
      },
      "primary":{
         "bg":"$colors.intentPrimary",
         "color":"$colors.onPrimary",
         "hover":{
            "bg":"$colors.intentPrimaryHover"
         },
         "active":{
            "bg":"$colors.intentPrimaryActive"
         },
         "disabled":{
            "bg":"$colors.intentPrimaryDisabled"
         }
      },
      "minimal":{
         "bg":"initial",
         "color":"$colors.onSurface",
```

```
      "hover":{
         "bg":"$colors.intentMinimalHover"
      },
      "active":{
         "bg":"$colors.intentMinimalActive"
      },
      "disabled":{
         "color":"$colors.intentMinimalDisabled"
      }
   }
},
"popup":{
   "color":"$colors.text",
   "background":"$colors.surface",
   "closeBtn":"#999999",
   "closeBtnHover":"#7a7a7a"
},
"resourcesTable": {
    "active": {
          "background": "#2481bc",
          "color": "$colors.onPrimary"
    },
    "background": "$colors.surface",
    "border": "$colors.border",
    "color": "$colors.onSurface",
    "header": {
          "background": "#e7e7e7 linear-gradient(180deg,#fafafa,#f0f0f0)",
          "backgroundImage": "linear-gradient(rgb(250, 250, 250), rgb(240, 240, 240))",
          "color": "$colors.text",
          "secondaryColor": "$colors.muted"
    },
    "hover": {
          "background": "$colors.background",
          "color": "$colors.text"
    },
    "linkColor": "$colors.linkColor",
    "visualIconColor": "#5c7080"
},
"schedule":{
   "list":{
      "bg":"$colors.surface",
      "border":"$colors.border",
      "item":{
         "bg":"$colors.surface",
         "color":"$colors.text",
```

```
            "hover":{
               "bg":"$colors.intentBaseHover"
            },
            "active":{
               "bg":"$colors.intentPrimary",
               "color":"$colors.onPrimary"
            }
         }
      }
   },
   "menu":{
      "color":"$colors.text",
      "bg":"$colors.surface",
      "hover":{
         "color":"$colors.onSurface",
         "bg":"$colors.intentBaseHover"
      },
      "active":{
         "color":"$colors.onPrimary",
         "bg":"$colors.intentBaseActive"
      },
      "disabled":"$colors.intentMinimalDisabled",
      "icon":{
         "color":"$colors.intentMinimal",
         "active":{
            "color":"$colors.intentSuccess",
            "hover":{
               "color":"#59933b"
            }
         },
         "delete":{
            "hover":{
               "color":"$colors.intentWarning"
            }
         },
         "hover":{
            "color":"$colors.onSurface"
         }
      },
      "divider":{
         "color":"$colors.border"
      }
   },
   "list":{
      "border":{
```

```
      "color":"$colors.intentBaseActive"
   }
},
"metaDataPicker":{
   "color":"$colors.text",
   "secondary":"$colors.muted",
   "background":"$colors.surface",
   "item":{
      "border":"#dedede",
      "aggrHover":"#dedede",
      "hover":{
         "bg":"$colors.intentBaseHover"
      }
   }
},
"radialMenu":{
   "bg":"rgba(0, 0, 0, 0.7)",
   "color":"$colors.onPrimary",
   "hover":{
       "bg":"#000"
   },
   "removeBtn":{
      "color":"$colors.onPrimary",
      "bg":"#939393",
      "hover":{
         "bg":"#E5683A"
      }
   }
},
"chartLegend":{
   "background":"$colors.background",
   "color":"$colors.text"
},
"chartTooltip":{
   "background":"$colors.background",
   "color":"$colors.text"
},
"widget":{
   "background":"$colors.surface",
   "borderColor":"$colors.surface",
   "selected":{
      "borderColor":"$colors.accentColor"
   },
   "titleColor":"$colors.text",
   "label":{
```

```
      "background":"$colors.background",
      "color":"#323232",
      "hover":{
         "background":"#d8d8d8"
      }
   },
   "iconColor":"$colors.muted"
},
"visualEditor":{
   "background":"$colors.background",
   "color":"$colors.text",
   "secondaryColor":"$colors.muted",
   "borderColor":"$colors.border"
},
"select":{
   "color":"$colors.text",
   "background":"$colors.surface",
   "border":"$colors.border",
   "disabled":{
      "background":"$colors.intentMinimalDisabled"
   }
},
"charts":{
   "base":{
      "axisLabelColor":"$colors.text",
      "axisLabelShadow":"rgba(255, 255, 255, 0.3)",
      "axisLineColor":"$colors.text",
      "color":"$colors.text",
      "palette":"<color-palette-name>",
      "pointerColor":"$colors.text",
      "splitLineColor":"$colors.text"
   },
   "<visual-type>":{
      "<charts-color-parameter>":"<newsetting>",
       ...
    },
},
"tables":{
   "base":{
      "color":"$colors.onSurface",
      "background":"$colors.surface",
      "backgroundOdd":"#fcfdfe",
      "border":"#BDC3C7",
      "borderSecondary":"#d9dcde",
      "hover":{
```

```
            "background":"#ecf0f1"
         },
         "heading":{
            "color":"$colors.text",
            "colorSecondary":"$colors.muted",
            "background":"$colors.background"
         }
      },
      "<table-type>":{
         "<tables-color-parameter>":"<newsetting>",
          ...
       },
   },
   "timebar":{
      "backgroundColor":"$colors.backgroundVariant",
      "backgroundColorHover":"rgba(167,182,194,0.3)",
      "border":"$colors.border",
      "textColorHover":"$colors.text",
      "textColor":"$colors.onBackgroundVariant",
      "scrubber":{
         "backgroundColor":"#b1bfd2",
         "backgroundColorHover":"rgba(167,182,194,0.3)",
         "splitterColor":"$colors.border",
         "splitterColorHover":"$colors.border",
         "tickColor":"$colors.intentPrimary",
         "tickColorHover":"#b1bfd2",
         "textColor":"#333333",
         "textColorHover":"#b1bfd2",
         "animatedTickColor":"#b1bfd2",
         "tooltipDatePickerColor":"#555555",
         "tooltipDatePickerBackgroundColor":"$colors.intentPrimary",
         "datePickerColorMaximized":"$colors.intentPrimary",
         "datePickerColorMinimized":"#ffffff",
         "datePickerBackgroundColorMaximized":"$colors.intentPrimary",
         "datePickerBackgroundColorMinimized":"#b1bfd2"
      }
   },
   "dialog":{
      "color":"$colors.onBackgroundVariant",
      "background":"$colors.background",
      "footerBackground":"$colors.backgroundVariant"
   },
   "colorPalette":{
      "background":"$colors.background",
      "borderColor":"$colors.border",
```

```
        "selected":{
           "background":"$colors.intentPrimary"
        },
        "hover":{
           "background":"$colors.intentBaseHover"
        },
        "iconColor":"$colors.muted",
        "activeIconColor":"$colors.text",
        "draggingBackgroundColor":"$colors.background"
     }
  }
}
}
```

Enable Trusted Access

Trusted Access is enabled by default in Composer. This enables you to create clients, and Composer to provide access tokens to authorized clients using Trusted Access. It can be disabled by a system administrator or member of the supervisors group, and another authentication method used.

Note: insightsoftware recommends using Trusted Access for all embed-related workflows.

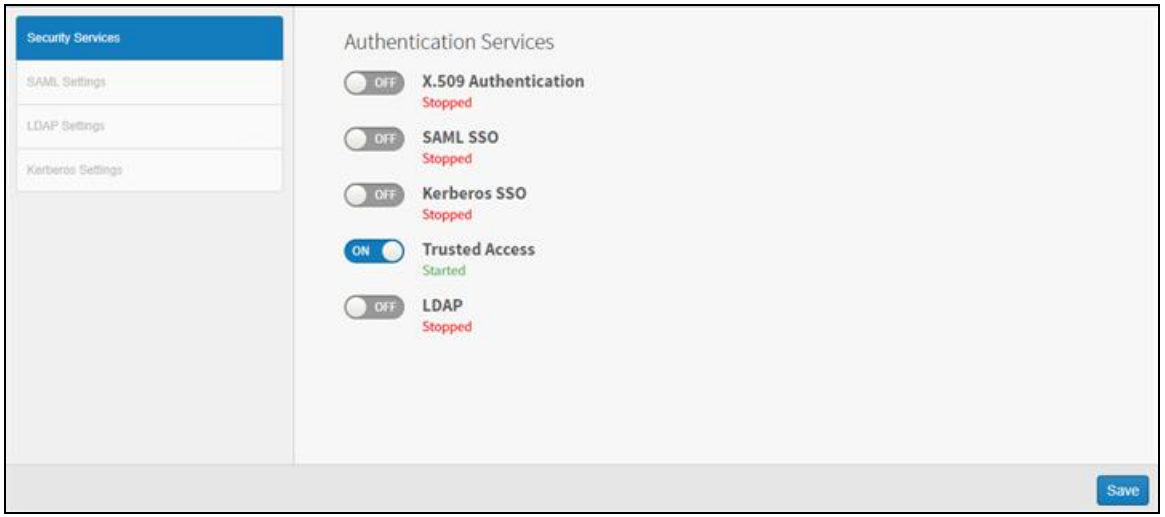
If Trusted Access is not enabled, all Trusted Access client and token-related API endpoints return a 404 response code or a 401 unauthorized response code (when an existing token is used). In addition, the Trusted Access API endpoints are not visible in the Swagger documentation. For information about Trusted Access endpoints, see Trusted Access API Endpoints.

Enable or disable Trusted Access:

1. Log in as a system admin or a member of the Supervisors group.

   Note: The default **supervisor** user is no longer installed; add users to the **Supervisors** group instead.

2. Select **Security** from the menu.



3. Switch the **Trusted Access** setting to **OFF** to disable, or **ON** to enable.

4. Select **Save**.

5. Restart Composer. See Restart ComposerSymphony Microservices.

When Trusted Access is ON, you can register a client or generate an access token.

White Label the Composer Interface

While Composer provides ways to change the default UI settings for the interface, there may be cases where further customization is desired. To further customize the Composer interface, you can upload a custom CSS file that includes the changes you want to apply to the interface. If you want to change the default logo, copyright information, or help content see Customize The Composer User Interface. The white labeling capabilities described within the overview aim to extend control of customization that the supervisor and developers have over the look-and-feel of the interface that users see.

Note: The Composer user interface is often enhanced between releases and those enhancements may affect the application's CSS. You should plan time to update your custom CSS between version upgrades.

What Can I Do With White Labeling?

- Make the Composer Login page look like my company
- Re-theme banners, backgrounds, and tiles for the Home Page
- Change the colors within the Composer client
- Translate the text into other languages, although you must work with your Composer technical support representative to do this.

What Do I Need?

- System admin access to the account(s) you want to customize
- Familiarity with CSS

Where Do I Get Started?

- Change The Login Page
- Change The Library
- Translate The ComposerSymphony Interface

Customize the Composer User Interface

When you install Composer, default UI settings are applied. These settings include: logo, favicon, header and footer settings, banner links, and login screen settings, and more. You can manage links to help content, customize copyright info, and change or remove the link to terms of use. To match the style of your company, you can also upload a custom `.css` to modify the default Composer skin or a custom JavaScript (`.js`) file to include on every page. This topic guides you in customizing the UI of the installed Composer instance as required.

The customization settings are organized into five categories. Read the following sections for information.

Read...	For information about ...
Customize The Application	Customizing the application title or favicon or about uploading a custom CSS or JavaScript file.
Customize The Header	Customizing the header and its height.
Customize The Footer	Customizing the footer and its height.
Customize The Banner	Customizing the banner on each page.
Log Into the ComposerSymphony UI About Dialog	Customizing the Login and About screens.

Customize the Application

You can customize the Composer application title and favicon. You can also upload a custom `.css` to modify the default Composer skin or a custom JavaScript (`.js`) file to include on every page.

Customize the Application Title

You can customize the Composer application title. The application title is displayed on the tab of your browser window.

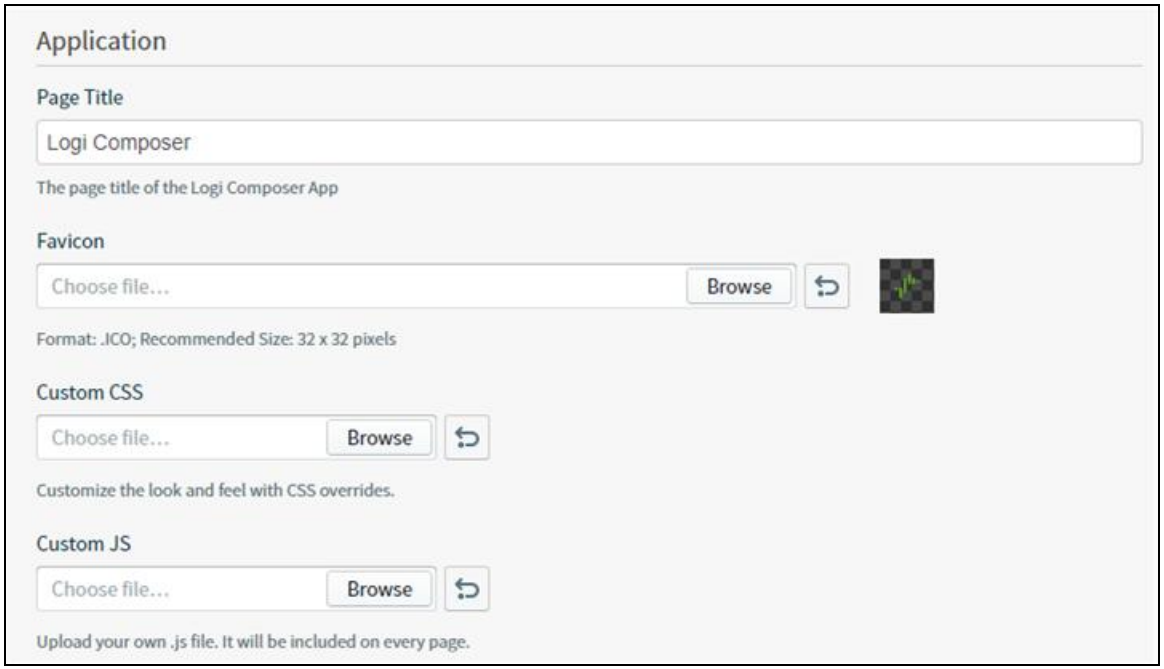
Customize the Composer application title

1. Log in as a system admin or a member of the Supervisors group.

   > **Note:** The default **supervisor** user is no longer installed; add users to the **Supervisors** group instead.

2. Select **Customize UI** on the supervisor menu (). The Customize UI page appears.

3. Under Application, locate the **Page Title** box. Specify a new title in the box. By default, the title is set to **Composer**.



4. Select **Save** to save and apply your changes.

Customize the Application Favicon

You can customize the favicon used by the Composer application. The favicon is displayed on the tab of your browser window:

To customize the application favicon:

1. Log in as a system admin or a member of the Supervisors group.

   > **Note:** The default **supervisor** user is no longer installed; add users to the **Supervisors** group instead.

2. Select **Customize UI** from the menu ( ). The Customize UI page appears.

3. Under Application, locate the **Favicon** box. Select **Browse** to browse for and select a new icon. The icon must be in `.ico` format with a recommended size of 32 x 32 pixels.

The selected icon is rendered on the Customize UI page after it is selected.

4. Select **Save** to save and apply your changes.

Upload a Custom CSS File

A custom CSS file can be uploaded to modify the default Composer skin.

> **Note:** insightsoftware recommends you to check the HTML structure of the page to provide the rules for the corresponding selectors in your CSS file. To override existing CSS rules, use `!important`.

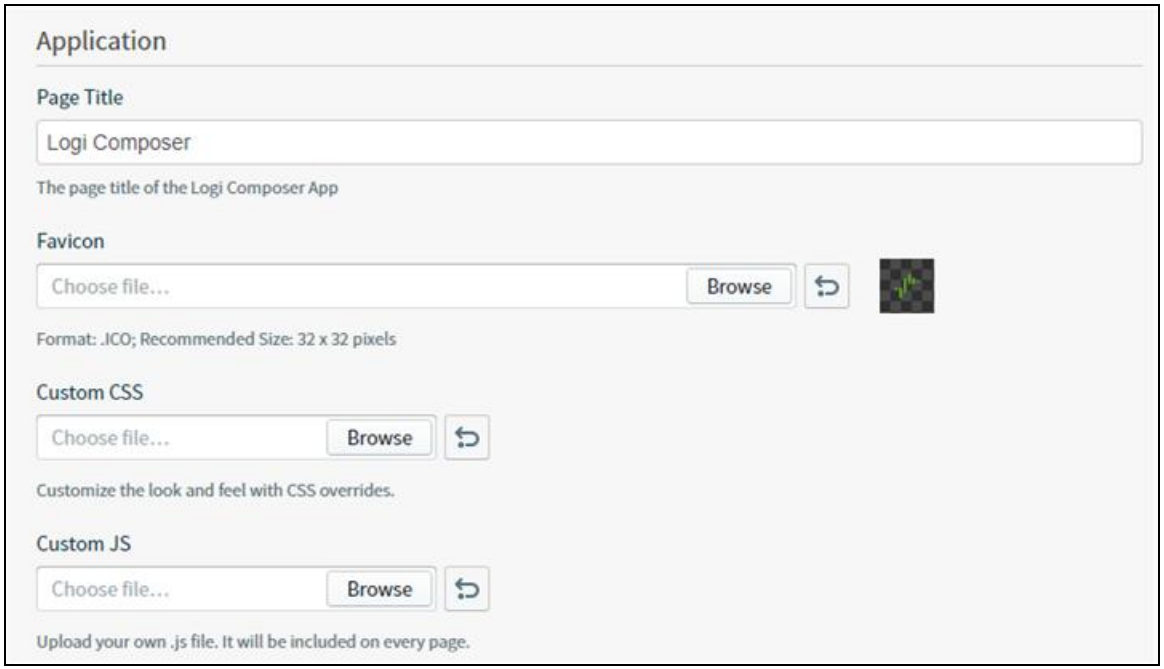
To upload a custom CSS file:

1. Log in as a system admin or a member of the Supervisors group.

   > **Note:** The default **supervisor** user is no longer installed; add users to the **Supervisors** group instead.

2. Select **Customize UI** from the menu ( ). The Customize UI page appears.

3. Under Application, locate the **Custom CSS** box. Select **Browse** to browse for and select a CSS file.



4. Select **Save** to save and apply your changes.

Upload a Custom JS File

A custom JavaScript (`.js`) file can be uploaded to include on every page of the UI.

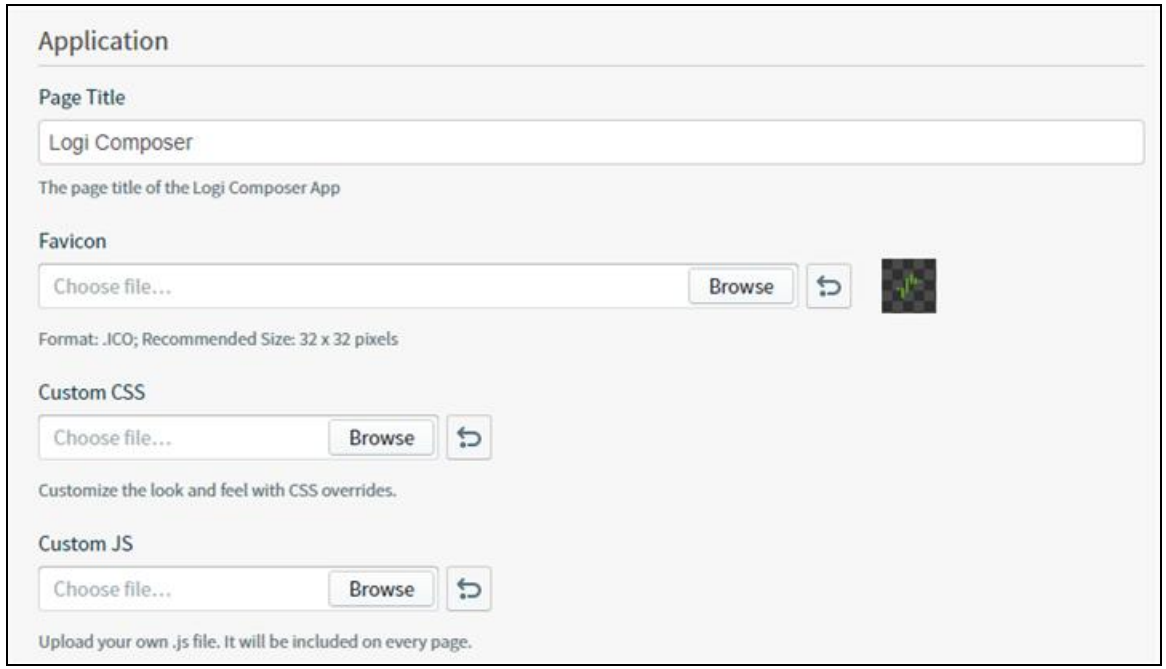
Upload a custom JavaScript file

1. Log in as a system admin or a member of the Supervisors group.

   > **Note:** The default **supervisor** user is no longer installed; add users to the **Supervisors** group instead.

2. Select **Customize UI** on the supervisor menu ( ). The Customize UI page appears.

3. Under Application, locate the **Custom JS** box. Select **Browse** to browse for and select a JavaScript file.



4. Select **Save** to save and apply your changes.

Customize the Login Screen, Home Page Background, and About Box

You can customize the copyright information, terms of service link, Login screen logo, the Login screen background, and the Home screen background. The logo, copyright, and terms of service appear in the About box of the UI. The copyright information, Login screen logo, and Login screen background appear with the Login screen. The Home screen background appears on the Home page.

Customize the Copyright Information

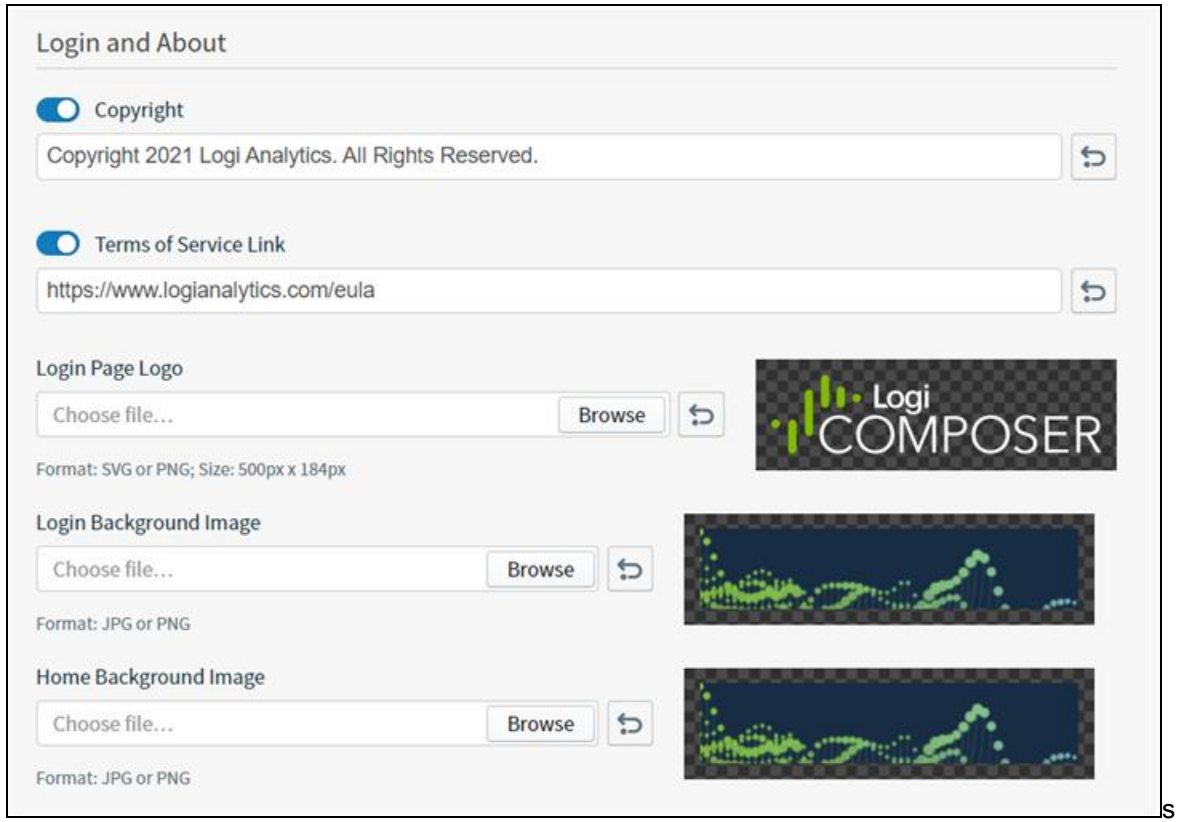
The copyright information appears on both the About box and the Login screen. By default, standard Composer copyright information is displayed.

1. Log in as a system admin or a member of the Supervisors group.

   Note: The default **supervisor** user is no longer installed; add users to the **Supervisors** group instead.

2. Select **Customize UI** from the menu (). The Customize UI page appears.

3. Under Login and About , locate the **Copyright** slider and box. To enable your custom copyright, slide the **Copyright** slider to the right (on). To disable your custom copyright, slide the **Copyright** slider to the left (off). By default the **Copyright** slider is on.



4. In the box below the **Copyright** slider, specify your custom copyright information.
5. Select **Save** to save and apply your changes.

Customize the Terms of Service Link

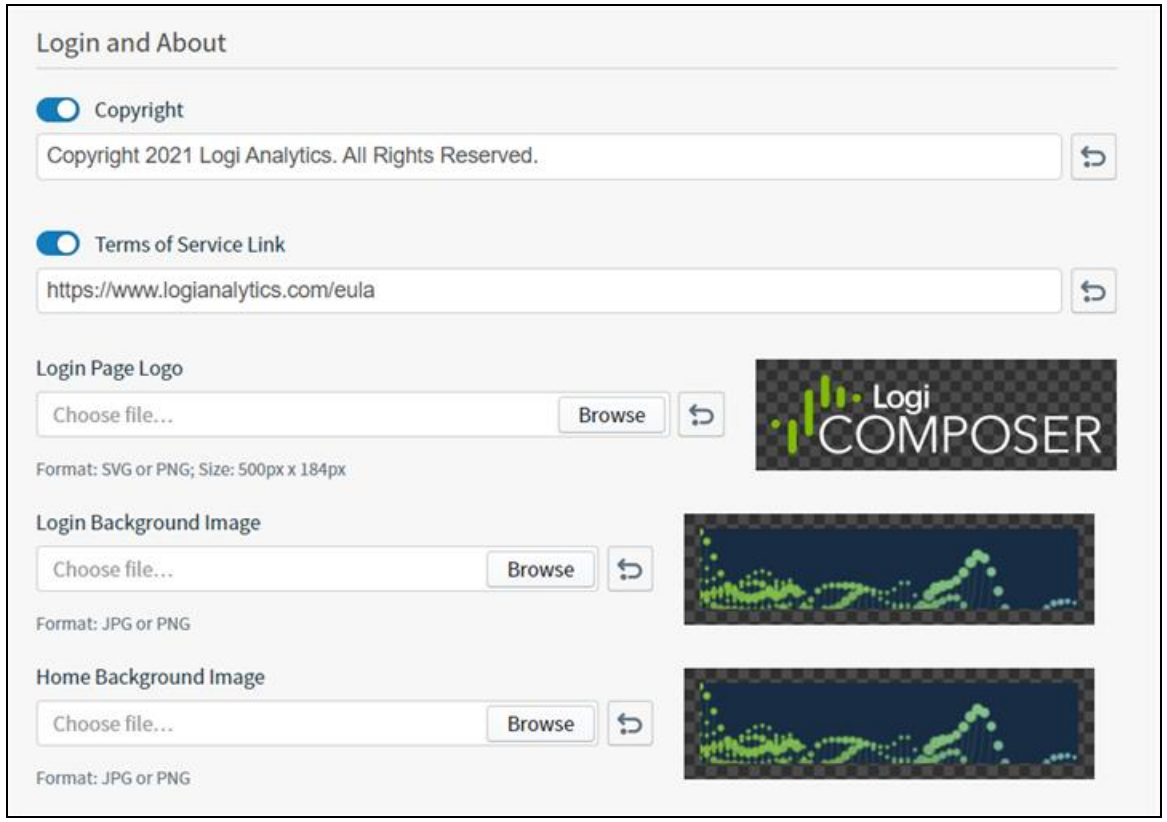
The terms of service link appears on the About box.

1. Log in as a system admin or a member of the Supervisors group.

> **Note:** The default **supervisor** user is no longer installed; add users to the **Supervisors** group instead.

2. Select **Customize UI** on the supervisor menu ( ). The Customize UI page appears.

3. Under Login and About , locate the **Terms of Service Link** slider and box. To enable your custom link, slide the **Terms of Service Link** slider to the right (on). To disable your custom link, slide the **Terms of Service Link** slider to the left (off). By default the **Terms of Service Link** slider is on.



4. In the box below the **Terms of Service Link** slider, specify your custom terms of service link. By default, the link is `https://www.logianalytics.com/eula`.

5. Select **Save** to save and apply your changes.

Customize the About and Login Screen Logo

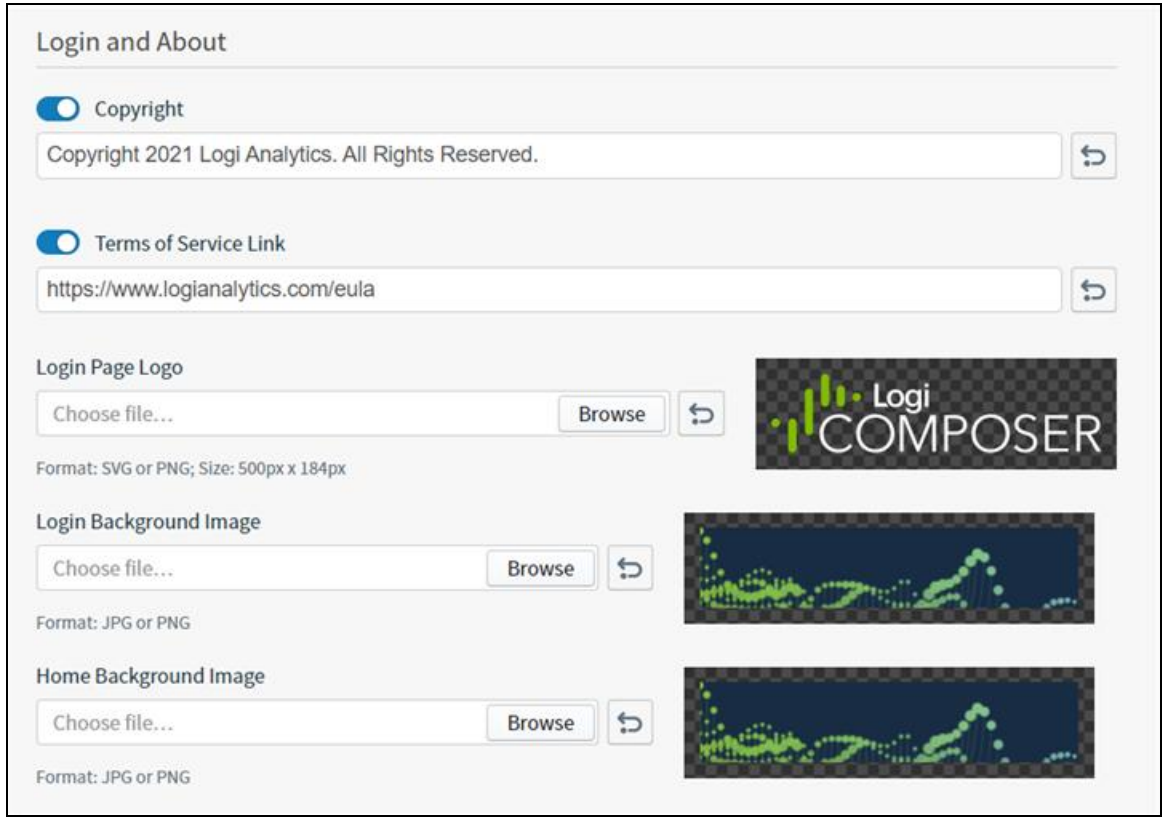
The logo appears on both the About box and the Login screen.

1. Log in as a system admin or a member of the Supervisors group.

   Note: The default **supervisor** user is no longer installed; add users to the **Supervisors** group instead.

2. Select **Customize UI** on the supervisor menu ( ). The Customize UI page appears.

3. Under Login and About , locate the **Login Page Logo** box. Select **Browse** to browse for and select a new logo. The logo must be in `.svg` or `.png` format with a transparent background. In addition, it must be 500 pixels wide and 184 pixels high.



   The selected logo is rendered on the Customize UI page after it is selected.

4. Select **Save** to save and apply your changes.

Customize the Login Screen Background Image

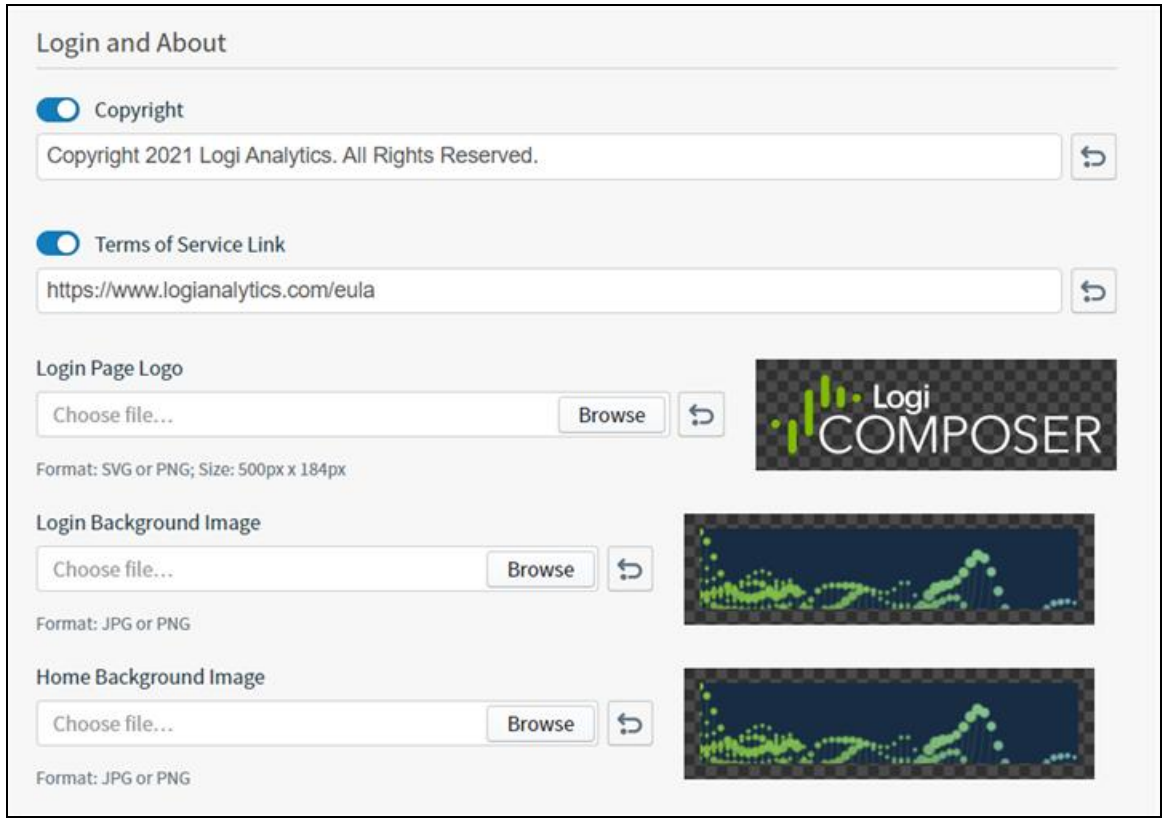
The background image appears on the Login screen.

1. Log in as a system admin or a member of the Supervisors group.

   Note: The default **supervisor** user is no longer installed; add users to the **Supervisors** group instead.

2. Select **Customize UI** from the main menu ( ). The Customize UI page appears.

3. Under Login and About , locate the **Login Background Image** box. Select **Browse** to browse for and select a new background image. The image must be in `.jpg` or `.png` format with a transparent background. In addition, it must be 2560 pixels wide and 1600 pixels high.



The selected image is rendered on the Customize UI page after it is selected.

4. Select **Save** to save and apply your changes.

Customize the Home Page Background Image

The background image appears on the Home page.

1. Log in as a system admin or a member of the Supervisors group.

   Note: The default **supervisor** user is no longer installed; add users to the **Supervisors** group instead.

2. Select **Customize UI** on the supervisor menu ( ). The Customize UI page appears.

3. Under Login and About , locate the **Home Background Image** box. Select **Browse** to browse for and select a new background image. The image must be in `.jpg` or `.png` format with a transparent background.

   The selected image is rendered on the Customize UI page after it is selected.

4. Select **Save** to save and apply your changes.

Customize the Header

You can add custom header for each page of the Composer UI. You can also specify the header height.

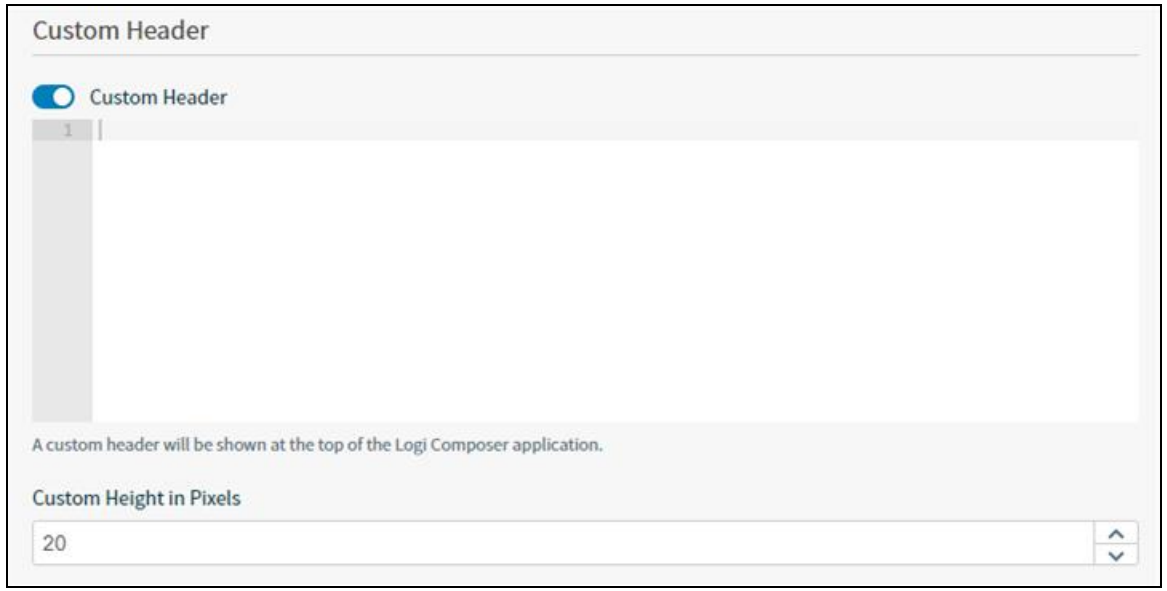
The header appears above the Composer banner in the UI.

Customize the header:

1. Log in as a system admin or a member of the Supervisors group.

   Note: The default **supervisor** user is no longer installed; add users to the **Supervisors** group instead.

2. Select **Customize UI** from the menu( ). The Customize UI page appears. The following screen shows all the custom header settings on the page.



3. Under Custom Header, enable the custom header by sliding the **Custom Header** slider to the right (on). You can always disable the custom header by sliding the **Custom Header** slider to the left (off). By default, the custom header is off.

4. In the box under the **Custom Header** slider, specify HTML structures for your custom header. You can either add inline styles or include the classes corresponding to your custom CSS file.

5. In the **Custom Height in Pixels** box, specify the height (in pixels) of the header. You can use the up and down arrows in the box to increment and decrement this value or you can type a value in the box.

6. Select **Save** to save and apply your changes.

   The header appears above the Composer banner in the UI.

Customize the Footer

You can add custom footer for each page of the Composer UI. You can also specify the footer height.

The footer appears below the normal Composer UI page.

To customize the footer:

1. Log in as a system admin or a member of the Supervisors group.

   > **Note:** The default **supervisor** user is no longer installed; add users to the **Supervisors** group instead.

2. Select **Customize UI** from the menu ( ). The Customize UI page appears. The following screen shows all the custom footer settings on the page.

3. Under Custom Footer, enable the custom footer by sliding the **Custom Footer** slider to the right (on). You can always disable the custom footer by sliding the **Custom Footer** slider to the left (off). By default, the custom footer is off.

4. In the box under the Custom Footer slider, specify HTML structures for your custom footer. You can either add inline styles or include the classes corresponding to your custom CSS file.

5. In the **Custom Height in Pixels** box, specify the height (in pixels) of the footer. You can use the up and down arrows in the box to increment and decrement this value or you can type a value in the box.

6. Select **Save** to save and apply your changes.

   The footer appears below the normal Composer UI page.

Customize the Banner

You can customize the UI banner. This includes customizing the default support link, documentation link, and header logo.

Customize the Support Link

1. Log in as a system admin or a member of the Supervisors group.

   > **Note:** The default **supervisor** user is no longer installed; add users to the **Supervisors** group instead.

2. Select **Customize UI** from the menu ( ). The Customize UI page appears.

3. Under Banner, locate the **Support Link** slider. To enable your custom support link, slide the **Support Link** slider to the right (on). To disable your custom support link, slide the **Support Link** slider to the left (off). By default the **Support Link** slider is on.

4. Specify the support link in the box below the **Support Link** slider. By default, the link is set to `https://www.logianalytics.com/support-portal-info`.

5. Select **Save** to save and apply your changes.

Customize the Documentation Link

1. Log in as a system admin or a member of the Supervisors group.

   > **Note:** The default **supervisor** user is no longer installed; add users to the **Supervisors** group instead.

2. Select **Customize UI** from the menu ( ). The Customize UI page appears.

3. Under Banner, locate the **Documentation Link** slider. To enable your custom documentation link, slide the **Documentation Link** slider to the right (on). To disable your custom documentation link, slide the **Documentation Link** slider to the left (off). By default the **Documentation Link** slider is on.

4. Specify the documentation link in the box below the **Documentation Link** slider. By default, the link is set to `https://documentation.logianalytics.com/composerltsactive/default.htm`.

5. Select **Save** to save and apply your changes.

Customize the Header Logo

You can change the header logo. It appears on the banner of the Composer work area.

1. Log in as a system admin or a member of the Supervisors group.

   Note: The default **supervisor** user is no longer installed; add users to the **Supervisors** group instead.

2. Select **Customize UI** from the menu ( ). The Customize UI page appears.

3. Under Banner, locate the **Header Logo** box. Select **Browse** to browse for and select a new logo. The logo must be in `.svg` or `.png` format with a transparent background. It can have a maximum height of 72 pixels.

   The selected logo is rendered on the Customize UI page after it is selected.

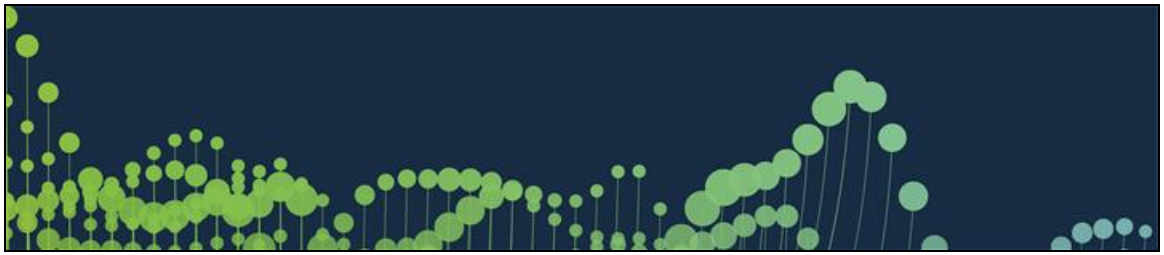
4. Select **Save** to save and apply your changes.

Change the Login Page

You can change the Login page of your Composer environment from the default values by uploading a custom CSS file. This enables you to use your organization's color scheme and branding not only on the background image, but on the background animation, login box, and button. This topic describes:

- Change The Login Page Background Gradient
- Change The Login Box
- Change The Login Button Color

Change the Login Page Background Gradient



The Customize UI tab in the Composer environment enables you to change the background. You can also change the gradient animation color of the background to match the changes applied to the Login Page background. Use the selector illustrated below:

```
#init-page-gradient {
       background: orange !important;
}
```

Change the Login Box

You can change the color of the Login Box for your Composer environment by using selector shown below:

```
#login-box.samlEnabled.collapse_login_box {
       background-color: orange !important;
}
```

You can also make the Login Box transparent by specifying a back-color value of the following:

```
background-color: rgba (0, 50, 50, 0) !important;
```

Change the Login Button Color

You can change the color of the Login button on your Composer environment. Follow the selector below:

```
input.btn.btn-success.btn-large {
       background-color: red !important;
}
```

To change the hover color of the Login button, follow the example selector below:

```
input.btn.btn-success.btn-large:hover {
       background-color: blue !important;
}
```

Change the Library

By customizing how the Composer library looks, you style the look-and-feel of your accounts. By changing the outline colors, background colors, icons, and fonts, the library can take on a whole new look. This topic provides the following tools for you to use:

- Background color of the library
- Library side pane
- Background for My Favorites section of the library

Change the Library Background Color

You can change the background color of the library within your tenant account(s) to match your particular design needs. Use the selector illustrated below:

```
.zdView-Home. zd-home {
       background-color: lightBlue;
}
```

Change the Library Side Pane

You can change the color of the main menu pane. This menu is displayed both in the dashboard library, as well as within visuals and dashboards. Use the selector illustrated below:

```
.zd-side-pane {
       background-color: red !important;
}
```

Change the My Favorites Background

You can change the background color of the My Favorites section within the library. Use the selector illustrated below:

```
.zd-home .zd-carousel-home {
       Background-color: lightBlue;
}
```

For more information, see:

- Customize The Composer User Interface
- White Label The Composer Interface