Kendraio App

Oct 30, 2020

Overview

Kendraio App is an open source dashboard application for rights owners, music makers, managers and record labels, enabling users to manage and track their digital media assets, collaborations and associated rights.

- Online demo: https://app.kendra.io
- Read the Docs: https://kendraio-app.readthedocs.io
- More information: https://www.kendra.io/kendraio-app

CHAPTER 1

Getting Started

Kendraio App is an open source dashboard application for rights owners, music makers, managers and record labels, enabling users to manage and track their digital media assets, collaborations and associated rights.

- Online demo: https://app.kendra.io
- Read the Docs: https://kendraio-app.readthedocs.io
- More information: https://www.kendra.io/kendraio-app

1.1 Installation

- checkout from git
- `npm install`
- run dev server: `npm run start:dev`

1.2 Contribute

- For help and support, join the conversation on Slack: https://kendraio.slack.com/
- To report bugs, suggest improvements, or request features, use the issue tracker: https://github.com/kendraio/kendraio-app/issues
- To download the source code, visit the repo: https://github.com/kendraio/kendraio-app

1.3 Versioning

Kendraio uses semantic versioning. Please note spec item 4:

Major version zero (0.y.z) is for initial development. Anything MAY change at any time. The public API SHOULD NOT be considered stable.

CHAPTER 2

Documentation

This documentation is built using Sphinx and hosted on Read the Docs. The source code for the documentation is available within the `docs` folder within the main app repository.

You will find the latest version of the documentation online here. The docs are also available in PDF and ePUB formats.

2.1 Building the docs

In order to build the documentation yourself, you will first need to install some dependencies. These instructions are for bash, and should work on Linux and macOS. Assuming you have Python (and therefore, also pip) installed, the following commands will install sphinx, fetch the code, and build the docs:

```
$ pip install sphinx
$ git clone https://github.com/kendraio/kendraio-app.git
$ cd docs
$ make html
```

2.2 Auto-building

Use the sphinx-autobuild command to rebuild the documentation when a change is detected, and auto-reload the browser to display the changes.

Run the following commands (from the root of the repository, or change the paths):

```
$ pip install sphinx-autobuild
$ sphinx-autobuild docs/ docs/_build/html
```

Then go to http://127.0.0.1:8000 in the browser.

CHAPTER 3

Adapters

3.1 What is an Adapter?

Adapters provide configuration for other Kendraio projects (such as App, CLI, etc) to integrate with third party service providers and data formats and schemas.

Within Kendraio Adapter we are assembling a library of API clients that will enable the Kendraio App (and any third party service provider) to plug into the APIs of existing media service providers. We are creating API clients that do not already exist and also assessing/testing those that do.

3.2 Why create an Adapter?

If you represent a third-party service that wishes to integrate with the Kendraio ecosystem, or make use of the shared technology we are developing, then you will need to create an Adapter. The Adapter methodology is in its early stages and under active development. For now, the presence of an Adapter within this repo is a starting point towards a more meaningful integration with the Kendraio ecosystem. It will be a place to store configuration related to how a particular service is accessed, the features provided by a service, and the data formats and schemas supported by the service.

CHAPTER 4

Installing Adapters

Navigate to the `Settings` > `Adapters` screen within the App to install and enable adapters.

CHAPTER 5

Creating an Adapter

An adapter is a set of configuration files that work with the building blocks of functionality within the Kendraio App to provide features and integrations with third-party services. Essentially, an adapter is just a set of JSON configuration files.

CHAPTER 6

Form Builder

The Kendraio Form Builder is a flexible UI generator that will automatically build data entry and editing forms from JSONSchema definitions of data models. Combined with extra customisation options available UI Schema, this offers a complete solution for building front-end user interfaces suitable for administration or content management.

CHAPTER 7

Workflow Blocks

7.1 Overview

Workflows are created by plugging together various reusable blocks of functionality. A workflow is a list of workflow items (or blocks). A task is a running instance of a workflow. Workflow definitions can be nested, as some workflow item blocks contain embedded workflows. Thus, tasks are nested too, and a running task may contain multiple tasks within it.

The available "blocks" of reusable functionality are documented below. The workflow builder within the app lets you create and customise workflows, load existing workflows from adapters, and generate links to share workflows you have created.

7.2 Online Demonstrator

- Master branch, stable: https://app.kendra.io
- Development branch, unstable: https://dev.app.kendra.io

CHAPTER 8

Sharing workflows

Workflows you have created can be shared with others. There are several ways to do this.

8.1 Add to an adapter

If you are working on an adapter, then workflows can be added to the adapters `configs` folder and they will be accessible by anyone who has your adapter installed and activated.

8.2 Use the share link

Generate a sharable link using the workflow builder share button. This link includes and encoded version of the config, and will load up the app directly and display the included workflow.

8.3 Export the code

You can export the JSON using the "Copy config" button. This can be send over instant messenger, email, or your preferred mode of communication. The receiving user then takes the JSON code and uses the "Paste config" button within the workflow builder.

CHAPTER 9

Workflow Cloud

Workflows can be downloaded from the Workflow Cloud.

These workflows are not specific to any adapter.

This section is under active development and likely to change.

CHAPTER 10

Actions

Use the actions block to add a row of buttons for one or more sub-tasks.

10.1 Default config

```
{
  "type": "actions",
  "buttons": [
    {
      "label": "Action",
      "color": "default",
      "blocks": []
    }
  ]
}
```

10.2 Supported properties

- **buttons** (array) - the list of buttons. Each button in the list has the following properties:
  - **label** (string) - the text for the button label
  - **color** (string) - passed as the "color" attribute to the material button. Use one of the supported Material color values, such as "primary", "warn", "accent". Leave as "default" to use the default button styling for a plain button.
  - **blocks** (array) - the list of workflow items to run when this button is pressed.

10.3 Example

This example shows two buttons. The first one dispatches an asynchronous command using the *Event Dispatch* bloc, the second one does nothing, but includes an `init` block in order to start the inner workflow so that it runs and signals completion to the outer workflow.

```
{
  "type": "actions",
  "buttons": [
    {
      "label": "OK",
      "color": "primary",
      "blocks": [
        {
          "type": "dispatch",
          "action": "resetApp"
        }
      ]
    },
    {
      "label": "Cancel",
      "blocks": [
        {
          "type": "init"
        }
      ]
    }
  ]
}
```

CHAPTER 11

Adapter Info

The Adapter info block is used to read adapter configurations within workflows and get information on which adapters are enabled.

There are two versions of this block `adapter-list` produces a full list of adapter metadata, whereas `adapter-info` is used to query for information on a specific adapter. The block config should include `adapterName` to specify which adapter info is loaded, or `adapterNameGetter` can be set using a JMES Path expression to query the name of the adapter to fetch from either the block input data or workflow context.

11.1 Example

This example is taken from one of the core workflows where the adapter info block is used to create a list of all the enabled adapters:

```
{
  "type": "adapter-list"
}
```

This example is also from a core workflow, and shows how to load information about a specific adapter:

```
{
  "type": "adapter-info",
  "adapterNameGetter": "context.queryParams.adapterName"
}
```

CHAPTER 12

Batch Process

Run the specified workflow for every item in a list. This block expects an array (list) of items to be passed in as data. It will run the embedded workflow for every item in the list, collect the results, and then output the list of results once all the tasks have completed.

12.1 Default config

```
{
  "type": "batch",
  "blocks": []
}
```

12.2 Supported properties

- **blocks** (array) - the workflow (list of blocks) to run for each item in the input data list.
- **flex** (boolean) (default = false) - apply a flex layout to the batch workflow outputs.

CHAPTER 13

Card

Wrap the inner block in card styling. This is a simple block that just provides card styling. It is useful along with the batch block when displaying multiple items.

13.1 Default config

```
{
  "type": "card",
  "blocks": []
}
```

13.2 Supported properties

- **blocks** - the list of workflow blocks that make up the display within the card

CHAPTER 14

Chart

Display values in a configurable chart.

14.1 Default config

```
{
  "type": "chart"
}
```

14.2 Supported properties

- **chartType** (default = “doughnut”): Set this to any of the chart.js chart types to switch the type of chart output.
- **options** (default = {}): Options to pass along to chart.js - the allowed options are different depending on the chart type. More details can be found in the Chart.js documentation: https://www.chartjs.org/docs/latest/charts/

In order to successfully render charts, the input data to this block must match the format of the data expected by the chart type. The mapping block is useful to get data into the correct format for display. Support for all chart types is in progress.

Pie and Doughnut chart types require a list of objects, where each object has a “label” and a “value”.

A chart block is usually preceded by a mapping block to get the data into the correct format. Useful examples are given here for each chart type that is supported:

14.3 Pie and Doughnut charts

```
{
  "type": "mapping",
  "mapping": "data[].{ label: originalTitle, value: length(associatedISRCs) }"
}
```

(NB: more mapping examples will be added here as support for more chart types is added)

14.4 Examples

The Chart.JS options can be customised. Callback functions in the configuration are not supported. Instead, you can use a JMES Path expression anywhere in the Chart.JS options that expects a callback function. For example, to use a logarithmic scale, customise the `yAxes` property:

```
{
    "type": "chart",
    "chartType": "line",
    "multi": true,
    "options": {
        "scales": {
            "yAxes": [
                {
                    "type": "logarithmic",
                    "ticks": {
                        "callback": "value",
                        "maxTicksLimit": 10
                    }
                }
            ]
        }
    }
}
```

The chart block can be configured to plot multiple sets of data on the same chart. To enabled this, set `multi` to true:

```
{
    "type": "chart",
    "chartType": "line",
    "multi": true
}
```

Then in the mapping that prepares the data for the chart block, create an array of data sets. Either using a `batch` block to query multiple sources or a single mapping that produces an array of data sets from a single source. Each dataset contains a `data` key and `label`. Here is an example of a query against multiple sources:

```
{
    "type": "multi",
    "batches": [
        {
            "blocks": [
                {
                    "type": "http",
                    "method": "get",
                    "endpoint": "https://trends-api.now.sh/api?keyword=angular"
                },
                {
```

(continues on next page)

(continued from previous page)

```
                    "type": "mapping",
                    "mapping": "{ data: data.default.timelineData[].{ value: value[0],
↪ label: formattedAxisTime }, label: 'angular'}"
                }
            ]
        },
        {
            "blocks": [
                {
                    "type": "http",
                    "method": "get",
                    "endpoint": "https://trends-api.now.sh/api?keyword=react"
                },
                {
                    "type": "mapping",
                    "mapping": "{ data: data.default.timelineData[].{ value: value[0],
↪ label: formattedAxisTime }, label: 'react'}"
                }
            ]
        },
        {
            "blocks": [
                {
                    "type": "http",
                    "method": "get",
                    "endpoint": "https://trends-api.now.sh/api?keyword=vue"
                },
                {
                    "type": "mapping",
                    "mapping": "{ data: data.default.timelineData[].{ value: value[0],
↪ label: formattedAxisTime }, label: 'vue'}"
                }
            ]
        }
    ]
}
```

CHAPTER 15

Context

Use the context block to save data into the global context to make it available for all subsequent tasks.

15.1 Default config

```
{
  "type": "context-save",
  "contextKey": "savedValue"
}
```

15.2 Supported properties

- **contextKey** (string) - the key of the value that will be saved into context.

For example, if you use a key of “savedData” then the value will be available in the context as “context.savedData”.

15.3 Usage patterns

The most common usage pattern with this block is to use it with the variable get block, to fetch a value from the adapter storage and save it into the context.

CHAPTER 16

CSV Export

Deprecated, use the separate serialize and file output blocks

Export data to a CSV file. Input data can be one of:

- An array of arrays
- An array of objects
- An object explicitly defining fields and data

16.1 Default config

```
{
  "type": "csv-export",
  "header": true,
  "skipEmptyLines": true
}
```

16.2 Supported properties

- quotes: false (or array of booleans)
- quoteChar: “”
- escapeChar: “”
- delimiter: “,”
- header: true
- newline: “rn”
- skipEmptyLines: false (or ‘greedy’)

- columns: null (or array of strings)

CHAPTER 17

CSV Import

Deprecated, use separate file input and parse data blocks

Load in data from a CSV or Excel file.

17.1 Default config

```
{
  "type": "csv-import",
  "header": true,
  "skipEmptyLines": true
}
```

17.2 Supported properties

- header (default = false): Set this to true and then the first row of parsed data will be interpreted as field names
- skipEmptyLines (default = false): If true, lines that are completely empty (those which evaluate to an empty string) will be skipped. If set to 'greedy', lines that don't have any content (those which have only whitespace after parsing) will also be skipped.

Most of the other config properties can be omitted, as the default values will automatically detect the correct settings to use from the imported CSV file. Other supported properties:

- delimiter (default = ""): Leave blank to auto-detect from a list of most common delimiters, or any values passed in through delimitersToGuess
- newline (default = ""): Leave blank to auto-detect. Must be one of r, n, or rn.
- quoteChar (default = """) The character used to quote fields. The quoting of all fields is not mandatory. Any field which is not quoted will correctly read.
- escapeChar: """,

- dynamicTyping (default = false): If true, numeric and boolean data will be converted to their type instead of remaining strings.
- preview: 0 - if greater than 0 then only this many lines will be read
- delimitersToGuess: [‘,’, ‘t’, ‘|’, ‘;’]

CHAPTER 18

Debug

Output the current data model values and list available contextual values.

18.1 Default config

```
{
  "type": "debug",
  "open": 1,
  "showContext": false
}
```

18.2 Supported properties

- open (number) (default = 1): how many levels of debug output to show by default.
- showContext (default = false): This defaults to false, in which case only output the input data into the debug block is shown. Set this to true to enable debugging of context values too.

CHAPTER 19

Dialog

Launch a modal dialog in which to run an embedded workflow.

19.1 Default config

```
{
  "type": "dialog",
  "blocks": []
}
```

NB: The modal dialog will close as soon as the inner workflow completes. Therefore you probably want to include at least one workflow item that requires interaction from the user, for example, `actions` buttons or a `form`.

19.2 Example

The following example is used within the blocks of a button to launch a confirmation dialog with two options, OK, and Cancel.

```
{
  "type": "dialog",
  "blocks": [
    {
      "type": "message",
      "title": "Are you sure you want to reset the app?",
      "message": "This action will remove all data and settings."
    },
    {
      "type": "actions",
      "buttons": [
        {
```

(continues on next page)

(continued from previous page)

```
          "label": "OK",
          "color": "primary",
          "blocks": [
            {
              "type": "dispatch",
              "action": "resetApp"
            }
          ]
        },
        {
          "label": "Cancel",
          "blocks": [
            {
              "type": "init"
            }
          ]
        }
      ]
    }
  ]
}
```

CHAPTER 20

Event Dispatch

Dispatch an event to trigger asynchronous processing.

20.1 Default config

```
{
  "type": "dispatch",
  "action": "[TASK_NAME]"
}
```

20.2 Usage

This block is only really useful by core workflows, as adapter workflows are not yet able to register asynchronous workflows to run in response to events.

To track development of this feature, see this issue: https://github.com/kendraio/kendraio-app/issues/53

CHAPTER 21

Faker

Generate fake data for testing purposes.

21.1 Default config

```
{
  "type": "faker",
  "jsonSchema": {}
}
```

21.2 Supported properties

- **jsonSchema** - The schema for the generated fake data. For full details of supported features see: https://github.com/json-schema-faker/json-schema-faker/blob/master/docs/USAGE.md

The following properties from JSON Schema are supported by the fake data generator:

- **$ref** — Resolve internal references only, and/or external if provided.
- **required** — All required properties are guaranteed, if not can be omitted.
- **pattern** — Generate samples based on RegExp values.
- **format** — Core formats v4-draft only: date-time, email, hostname, ipv4, ipv6 and uri - also uri-reference, uri-template, json-pointer and uuid are supported.
- **enum** — Returns any of these enumerated values.
- **minLength**, **maxLength** — Applies length constraints to string values.
- **minimum**, **maximum** — Applies constraints to numeric values.
- **exclusiveMinimum**, **exclusiveMaximum** — Adds exclusivity for numeric values.

- **multipleOf** — Multiple constraints for numeric values.
- **items** — Support for subschema and fixed item values.
- **minItems**, maxItems — Adds length constraints for array items.
- **uniqueItems** — Applies uniqueness constraints for array items.
- **additionalItems** — Partially supported (?)
- **allOf**, **oneOf**, **anyOf** — Subschema combinators.
- **properties** — Object properties to be generated.
- **minProperties**, **maxProperties** — Adds length constraints for object properties.
- **patternProperties** — RegExp-based object properties.
- **additionalProperties** — Partially supported (?)
- **dependencies** — Partially supported (?)
- **not** — Not supported yet (?)

JSON schema faker includes basic support for generating values, but more advanced generators are available by using the "faker" extension. All fake data generators from the faker.js package can be used, see https://github.com/marak/Faker.js/ for full list of available data generators. To specify a faker generator, add the "faker" key to the JSON Schema, for example:

```
{
  "type": "string",
  "faker": "internet.email"
}
```

CHAPTER 22

File Export

Export data to a file.

22.1 Expected input data

This block expects the input data to contain the following keys. These match the output of the serialize block, so the two blocks can be easily combined to output data:

- **format**: the format of the data
- **data**: the data serialized as a text string

If the input data is a string then it will be exported as a plain text file. If the input data is an object, but does not contain the "type" and "data" keys then it will be exported as JSON data.

22.2 Default config

```
{
  "type": "file-export",
  "label": "Export",
  "fileName": "exported"
}
```

22.3 Supported properties

- **label** (default = "Export") the label to use for the export button
- **fileName** (default = "export") the name of the downloaded file, the file extension will be added automatically based on the input data format.

CHAPTER 23

File Input

Import a file from the local system.

23.1 Default config

```
{
  "type": "file-input",
  "label": "Import",
  "accept": ["csv", "json", "xml"]
}
```

23.2 Supported properties

- **accept** (default = ['json']) a list of formats to allow for import. The file selection will limit the selectable files to only these types. Supported types include: ['json', 'csv', 'xml']
- **label** (default = "import") - the label for the button to launch the file picker.

23.3 Emitted data

The data emitted from this block has the following format:

- **name** - name of file that was selected
- **type** - the MIME type of the file
- **size** - the size in bytes of the selected file
- **lastModified** - the last modified timestamp
- **content** - a text string containing the full content of the file

CHAPTER 24

Form

Display a form for data entry or editing.

Forms are displayed based on the UI Schema and JSON Schema provided. These can be provided inline within the block configuration, or loaded from an adapter.

24.1 Default config

```
{
  "type": "form",
  "jsonSchema": {},
  "uiSchema": {}
}
```

24.2 Supported properties

- **jsonSchema**: the data model for which to generate the form
- **uiSchema**: the UI Schema for customisation of the form. This is used to add further options to the form that are beyond what is allowed in JSON schema, such as customising widget types, labels, and widget options.
- **adapter** and formId: if these two properties are provided they are used to fetch the JSON schema and UI Schema from the adapter config repository.
- **label** (default = "Submit): customise the label displayed on the submit button
- **hasSubmit** (default = true): this allows you to remove the submit button from the form. If this is set to true the submit button is not displayed and the form will output all changes.
- **debounceTime** (default = 400): the number of milliseconds to debounce form output when not using a submit button. Multiple changes within this time will be ignored, and only the last change is emitted. This is a useful option to prevent unnecessary execution of multiple tasks within the workflow, for example if the form feeds into

a HTTP block to pull data from an API (such as in an autocomplete) then the debounce will limit the number of requests that are sent while the user is entering input.

- **emitOnInit** (boolean) (default = false): enable this to emit the form values when the block is initialised. This is useful if you need to pass on default values.

CHAPTER 25

Gosub

Embedded a workflow within a workflow.

This allows composition of larger building blocks to create workflows that are easier to manage, and DRY (don't repeat yourself).

Typical use-cases for this are creating custom form widgets that can be referenced from a form's `uiSchema`. Or, creating dashboards that are made up of multiple other workflows combined with a `multi` multiplex workflow block.

```
{
  "type": "gosub",
  "adapterName": "kendraio",
  "workflowId": "myWorkflow"
}
```

CHAPTER 26

Data Grid

This block displays the data passed in as a rich data table using the flexible grid library ag-grid.

26.1 Default config

```
{
  "type": "grid",
  "gridOptions": {}
}
```

26.2 Supported properties

- **sizeColumnsToFit** (boolean) (default = true) - set this to false to disable the auto-size algorithm from running that will resize the columns after display to best fit the available space.
- **gridOptions** (object) - Add supported options for the grid, based on the grid properties listed here: https://www.ag-grid.com/javascript-grid-properties/
- **columnDefs** (array) - A list of columns to display in the grid. The supported options are based on the grid column properties documented here: https://www.ag-grid.com/javascript-grid-column-properties/

26.3 Advanced features

Adding `"cellRenderer": "workflowRenderer"` to a column allows to embed workflow within a cell of the table. The `cellRendererParams` should include `blocks` as an array of workflow tasks to be added to the cell. Examples of use include adding an "Operations" column, using the "actions" task within the cell to add a list of buttons to operate on the data from that row. The cell will be passed in data from that row of the grid only.

26.4 Examples

This example shows some advanced grid features, such as customisation of the pagination, multiple row selection, the addition of a selectable checkbox column, and the use of the "workflow renderer" to embed another workflow within the grid cells. For example, the workflow renderer is useful for adding edit buttons to a content administration table.

```
{
  "type": "grid",
  "gridOptions": {
    "pagination": true,
    "paginationPageSize": 10,
    "rowSelection": "multiple"
  },
  "columnDefs": [
    {
      "width": 50,
      "resizable": false,
      "checkboxSelection": true
    },
    {
      "headerName": "ID",
      "field": "id"
    },
    {
      "headerName": "Name",
      "field": "name"
    },
    {
      "headerName": "Operations",
      "cellRenderer": "workflowRenderer",
      "cellRendererParams": {
        "blocks": [
          {
            "type": "actions",
            "buttons": [
              {
                "label": "View",
                "blocks": [
                  {
                    "type": "dialog",
                    "blocks": [
                      {
                        "type": "debug"
                      },
                      {
                        "type": "actions",
                        "buttons": [
                          {
                            "label": "OK",
                            "blocks": [
                              {
                                "type": "init"
                              }
                            ]
                          }
                        ]
                      }
```

(continues on next page)

(continued from previous page)

```
                    ]
                  }
                ]
              },
              {
                "label": "Edit",
                "blocks": [
                  {
                    "type": "launch",
                    "adapter": "example",
                    "workflowId": "editTheThing",
                    "context": {
                      "id": "data.id"
                    }
                  }
                ]
              }
            ]
          }
        ]
      }
    }
  }
}
```

This example shows filters, checkbox selection, pagination, and valueFormatter options in use:

{ “type”: “grid”, “gridOptions”: {

“pagination”: true, “paginationPageSize”: 20, “rowSelection”: “multiple”

}, “columnDefs”: [

{ “headerCheckboxSelection”: true, “width”: 50, “resizable”: false, “checkboxSelection”: true

}, {

“headerName”: “Description”, “field”: “Description”, “filter”: “agTextColumnFilter”, “filterParams”: {

“applyButton”: true, “resetButton”: true

}

}, {

“headerName”: “Department”, “field”: “Department”, “filter”: “agTextColumnFilter”, “filterParams”: {

“applyButton”: true, “resetButton”: true

}

}, {

“headerName”: “Size / Weight”, “field”: “Size / Weight”, “filter”: “agTextColumnFilter”, “filterParams”: {

“applyButton”: true, “resetButton”: true

```
                }
            }, {
                "headerName": "Price", "field": "Price", "width": 150, "filter": "agNumber-
                ColumnFilter", "valueFormatter": "join('', ['£', value])", "filterParams": {
                    "applyButton": true, "resetButton": true
                }, "cellClass": "text-right"
            }
        ]
    }
```

CHAPTER 27

Google Sheet

Use a Google Spreadsheet as a data source.

First, publishing your Google Sheet: In the File menu pick Publish to the web. Click Start publishing. A URL will appear, IGNORE THIS URL! Now that you've published your sheet, you now need to share it, too.

- Click the Share link in the upper right-hand corner
- Click the very pale Advanced button
- Change... access to "On - Anyone with a link"
- Make sure Access: Anyone says Can view, since you don't want strangers editing your data
- Click Save

Copy the Link to Share. Your URL should look something like https://docs.google.com/spreadsheets/d/1FNFdfnfQqz81igvokIwxdcOoK59NRSZ00KAm0xyIVE8/edit?usp=sharing. It should not have a /d/e in it.

Enter the URL from above as the key option in the block config:

```
{
    "type": "gsheet",
    "simple": true,
    "key": "https://docs.google.com/spreadsheets/d/
↪1pROEg6WMQNcsR4QFMQJyDuGO67JKscCgacnOi8rivCM/edit?usp=sharing"
}
```

Add *simple: true* if the Spreadsheet only includes one sheet/tab.

If you get an error message about JSON parsing issues, or *e is undefined*, or similar, it is probably the case that the link/key to the file is wrong, and instead of getting valid JSON the gsheet block is getting an error back from google.

Check that you have shared a Google Spreadsheet - if it is an XLSX or other format file you need to first go to File menu -> "Save as Google Sheets".

If you are not using simple mode (i.e. you have more than one sheet/tab in the spreadsheet) then you will probably need a mapping to get the data you want, Eg:

```
{
    "type": "mapping",
    "mapping": "data.\"music-recording\".elements"
}
```

Where music-recording is the name of the tab, and elements gives you the data from the rows of the sheet. NB: the ” around “music-recording” are only required because the sheet name includes a special character (-). For simple sheet names, this is not required. Eg:

```
{
    "type": "mapping",
    "mapping": "data.claim.elements"
}
```

CHAPTER 28

HTTP Request

Get, put or post data to an external HTTP endpoint.

28.1 Default config

```
{
  "type": "http",
  "method": "get",
  "endpoint": {
    "protocol": "https:",
    "host": "jsonplaceholder.typicode.com",
    "pathname": "/posts",
    "query": {
      "userId": 1
    }
  }
}
```

28.2 Supported properties

- **method** - REQUIRED - allowed values are get, put, post, and delete.
- **notify** (boolean) (default = true): Show a notification message if the request is successful. This message is not sent when the HTTP method is GET, but can be turned on and off for POST, PUT, and DELETE requests by using this property.

28.3 Examples

For simple requests, the `endpoint` can just be a simple string:

```
{
    "type": "http",
    "method": "get",
    "endpoint": "https://covid19.mathdro.id/api"
}
```

For advanced use cases, the payload can be constructed using a JMES Path expression. Custom headers can also be specified using JMES Path expressions:

```
{
    "type": "http",
    "method": "post",
    "endpoint": {
        "protocol": "https:",
        "host": "accounts.spotify.com",
        "pathname": "/api/token"
    },
    "payload": "'grant_type=client_credentials'",
    "headers": {
        "Content-Type": "'application/x-www-form-urlencoded'",
        "Authorization": "join('', ['Basic ', btoa(join('', [data.client_id, ':',
↪data.client_secret]))])"
    }
}
```

It is possible to query a GraphQL endpoint using the HTTP block.

```
{
    "type": "http",
    "method": "post",
    "notify": false,
    "endpoint": {
        "protocol": "https:",
        "host": "valnet.lurker.dev",
        "pathname": "/api/graphql"
    },
    "payload": "{ query: 'query ($token: String) {  viewer(token: $token) {
↪allCommitments {      id      action      plannedStart      committedOn      due
↪  committedQuantity {        numericValue        unit {          name        }
↪}      note      resourceClassifiedAs {        name        category      }
↪involves {        id        resourceClassifiedAs {          name          category
↪      }        trackingIdentifier      }      provider {        id        name
↪}      receiver {        id        name      }      inputOf {        id        name
↪    }      outputOf {        id        name      }      scope {        id
↪name      }      plan {        id        name      }      isPlanDeliverable
↪forPlanDeliverable {        id        action        outputOf {          name
↪}      }      isDeletable    }  }}', variables: { token: context.vfAuth } }"
}
```

CHAPTER 29

Initialisation

Initialise workflow processing on page load.

29.1 Default config

```
{
  "type": "init"
}
```

CHAPTER 30

Map

The map block allows embedding a map for display of geocoded data. Leaflet.JS is used to display the map, and some Leaflet.JS customisation options are supported.

The `height` in pixels of the block (default height 500px), and the initial `zoom` level (default zoom = 8) are provided as configuration options to the block. You can also optionally provide a `latlng` config option for the default map center point. The map defaults to centering at [51.505, -0.09].

30.1 Examples

The map block expects incoming data to be an array of objects that have `lat`, `long`, and `label` properties. Here is an example of a mapping that produces appropriate data:

```
{
    "type": "mapping",
    "mapping": "data[?recovered > `20`].{ lat: lat, long: long, label: join(' ', [to_
↪string(recovered), 'recovered in', combinedKey][? @ != null]) }"
}
```

Example map block configuration:

```
{
    "type": "map",
    "height": 500,
    "zoom": 2
}
```

CHAPTER 31

Mapping

Map data from one format to another, or query the input data using JMESPath or JSONPath expressions. http://jmespath.org/tutorial.html

31.1 Default config

```
{
  "type": "mapping",
  "mapping": "data"
}
```

31.2 Supported properties

- **mapping** - a JMESPath expression

CHAPTER 32

Message

Display a customisable message based on the current data modal values.

32.1 Default config

```
{
  "type": "message",
  "message": ""
}
```

32.2 Supported properties

- **title** - (optional) text to be styled as a title
- **message** - (optional) text to be styled as a normal paragraph

CHAPTER 33

Multiplex

Display and process multiple workflows concurrently.

33.1 Default config

```
{
  "type": "multi",
  "batches": []
}
```

CHAPTER 34

Parse Data

Process incoming data as JSON, CSV, or XML.

This block expects the incoming data to contain at least the following keys. These match the output of the File Input block so the two can be easily chained together:

- **type** - the MIME type of the incoming data, must match one of the supported parse formats.
- **content** - the text content to parse and extract data from.

The correct format for parsing is selected based on the incoming type. If the incoming data does not contain a "type" attribute, AND it is of string type, then it may be parsed with automatic detection of the type. The automatic detection works like this:

- Is the input data a string? No - do nothing and stop parsing
- Does the string start with a "{" character? Yes - attempt to parse as JSON data
- Does the string start with a "<" character? Yes - attempt to parse as XML data
- Otherwise - attempt to parse as CSV.

34.1 Default config

```
{
  "type": "parse-data",
  "csvOptions": {
    "header": true,
    "skipEmptyLines": true
  },
  "xmlOptions": {

  }
}
```

34.2 Supported properties

- **csvOptions** - configuration options to pass along to the CSV parser. This can include:
  - **header** (default = false): Set this to true and then the first row of parsed data will be interpreted as field names
  - **skipEmptyLines** (default = false): If true, lines that are completely empty (those which evaluate to an empty string) will be skipped. If set to 'greedy', lines that don't have any content (those which have only whitespace after parsing) will also be skipped.

  Most of the other config properties can be omitted, as the default values will automatically detect the correct settings to use from the imported CSV file. Other supported properties:
  - **delimiter** (default = ""): Leave blank to auto-detect from a list of most common delimiters, or any values passed in through delimitersToGuess
  - **newline** (default = ""): Leave blank to auto-detect. Must be one of r, n, or rn.
  - **quoteChar** (default = """) The character used to quote fields. The quoting of all fields is not mandatory. Any field which is not quoted will correctly read.
  - **escapeChar**: "",
  - **dynamicTyping** (default = false): If true, numeric and boolean data will be converted to their type instead of remaining strings.
  - **preview**: 0 - if greater than 0 then only this many lines will be read
  - **delimitersToGuess**: [',', 't', 'l', ';']
  - **xmlOptions** - options to pass along to the XML parser

CHAPTER 35

Query

Run a configurable query against a data source

NB: Use of this block for HTTP calls is deprecated, and you should use the dedicated HTTP block. This block will be focused on making calls to the local DB

35.1 Default config

```
{
  "type": "query"
}
```

CHAPTER 36

Reference

This block presents a select widget that takes it's options from the data input. It expects to receive an array of options, and shows a drop-down select list where one of the options can be selected. The selected option is output as the data from this block.

36.1 Examples

Here is an example showing most of the possible configuration values:

```
{
    "type": "reference",
    "fieldLabel": "Select artist",
    "labelGetter": "name",
    "valueField": "id",
    "outputGetter": "@"
}
```

Instead of `labelGetter` the label can be taken from a field of the input data using `labelField`. The label field defaults to `label` so if the input data items have a field called `label` you can omit these options.

You can also set the `required` status of the input select list. This defaults to `false`.

CHAPTER 37

Serialize Data

This block will convert the incoming data into an encoded/serialized string of text data. The encoding of the data depends on the configuration and can be selectable by the user.

The output of this block is an object with two keys:

- **format** (default = 'json') - a string representing the type of the serialized data
- **data** - the text string with the encoded data

37.1 Default config

```
{
  "type": "serialize",
  "hasFormatSelection": true,
  "formats": [
    "json",
    "csv",
    "xml"
  ],
  "csvOptions": {
    "header": true,
    "skipEmptyLines": true
  },
  "xmlOptions": {

  }
}
```

37.2 Supported properties

- **hasFormatSelection** (default = true) - show a widget to select the output format.

- **formats** (default = [‘json’, ‘csv’, ‘xml’]) - which formats to allow for output
- **showRecordCount** (default = true) set this to false to disable the display of the record counter
- **csvOptions** - configuration options to pass along to CSV serializer
- **xmlOptions** - configuration options to pass along to the XML serializer

For CSV data the input data can be one of:

- An array of arrays
- An array of objects
- An object explicitly defining fields and data

Supported `csvOptions` with default values:

- **quotes**: false (or array of booleans
- **quoteChar**: “”
- **escapeChar**: “”
- **delimiter**: “,”
- **header**: true
- **newline**: “rn”
- **skipEmptyLines**: false (or ‘greedy’)
- **columns**: null (or array of strings)

CHAPTER 38

Switch

The switch block is used to conditionally run workflows based on evaluation of input value.

38.1 Default config

```
{
  "type": "switch",
  "valueGetter": "data",
  "cases": []
}
```

38.2 Supported properties

- **valueGetter** - JMESPath to evaluate against incoming data to get value for comparison
- **cases** (default = []) - an array of cases, each case contains:
  - **value** - the value that must match for this case to run
  - **blocks** - the workflow items that are run when this case matches
  - **valueGetter** (optional) - a JMESPath expression to evaluate which is used to compare and select the appropriate case. Only the specified case that matches the evaluated expression will be run. If a valueGetter expression is not provided then the input data is used as the value to match against.
  - **default** - object contains one key:
    - **blocks** - the array of workflow items to run if no cases match

CHAPTER 39

Template

Display templated output based on current modal values.

39.1 Default config

```
{
  "type": "template",
  "template": ""
}
```

See the Handlebars documentation for full list of supported templating features.

https://handlebarsjs.com/

CHAPTER 40

Get Variable

Fetch a value from local storage.

40.1 Default config

```
{
  "type": "variable-get",
  "name": "[VARIABLE_NAME]"
}
```

CHAPTER 41

Set Variable

Save a value to local storage.

41.1 Default config

```
{
  "type": "variable-set",
  "name": "[VARIABLE_NAME]"
}
```