DTI Guide: Data Integration on C3 AI Suite

- Introduction for C3 Data Integration
    - Additional Resources
- Specialized Types
    - Canonical Types
        - Additional Resources
    - Transform Types
        - Additional Resources
    - Application Types
- Basic Data Sources
    - CSV Files
    - JSON Files
    - Seed Data
    - Sending Data via POST and RESTful API
        - Helper Scripts and Tools
            - send-file.py: DTI Developed helper script
            - curl
            - Postman
- Monitoring Data Integration
    - Source Systems and Source Collections
    - SourceFile
    - Checking SourceFile Integration And Troubleshooting
- Complex Data Sources
    - Custom External Database
    - C3 AI Supported Database Technologies
    - Other Data Formats

Introduction for C3 Data Integration

Ultimately, we want to connect our C3 cluster to a source of data. This is done through the process of Data Integration. Generally, data flow from a source file or system into a Canonical Type. This Canonical Type is meant to mirror directly the data source. Next, a Canonical Transform is defined which connects the Canonical Type to another C3 Type which is part of your final data model. A general diagram of this flow follows:

blocked URL

Additional Resources

- Developer Documentation
    - https://developer.c3.ai/docs/7.12.25/topic/di-home
    - https://developer.c3.ai/docs/7.12.25/topic/dataIntegrator
    - https://developer.c3.ai/docs/7.12.25/topic/di-sources-and-canonicals
- C3.ai Academy Videos
    - Learning Module: **Data Integration**
        - Data Integration Overview
- DTI Data Integration Video
    - DTI Training Videos - Data Integration Demo.

Specialized Types

First, we'll discuss the specialized Types which are used throughout the Data Integration system, and then discuss what happens once the data enters the C3 AI Suite. Things within the C3 AI Suite are a little cleaner to think about first. Once we've established how things work *inside* the C3 AI Suite, we'll follow with how we can get the data into the first step of the C3 AI Suite's Data Integration System.

Canonical Types

A Canonical Type is the entry point of data into the C3 AI Suite. It is a special Type which mixes in the `Canonical` Type. Mixing in the `Canonical` Type tells the C3 AI Suite to add some capabilities such as a RESTFUL API endpoint to ingest data, the ability to grab data from a seed data directory, and the ability to kick off the Data Integration pipeline when new data arrives. Conventionally, a Canonical Type should start with the word '`Canonical`'. Its fields should match the names of fields in the intended source, and the fields should be primitive Types.

For example, let's look at the Type '`CanonicalSmartBulb`' from the lightbulbAD tutorial package:

```
/*
 * Copyright 2009-2020 C3 (www.c3.ai). All Rights Reserved.
 * This material, including without limitation any software, is the confidential trade secret and proprietary
 * information of C3 and its licensors. Reproduction, use and/or distribution of this material in any form is
 * strictly prohibited except as set forth in a written license agreement with C3 and/or its authorized
distributors.
 * This material may be covered by one or more patents or pending patent applications.
 */

/**
* This type represents the raw data that will represent {@link SmartBulb} information.
*/
type CanonicalSmartBulb mixes Canonical<CanonicalSmartBulb> {
  /**
   * This represents the manufacturer of a {@link LightBulb}
   */
  Manufacturer: string

  /**
   * This represents the bulbType of a {@link LightBulb}
   */
  BulbType:     string

   /**
   * This represents the wattage of a {@link LightBulb}
   */
  Wattage:     decimal

  /**
   * This represents the id of a {@link LightBulb}
   */
  SN:           string

  /**
   * This represents the startDate of a {@link LightBulb}
   */
  StartDate:    datetime

  /**
   * This represents the latitude of a {@link SmartBulb}
   */
  Latitude:     double

  /**
   * This represents the longitude of a {@link SmartBulb}
   */
  Longitude:    double
}
```

Note that far fewer fields are present here than in the `SmartBulb` Type. This is because the Canonical Type is just used as an entry point to the C3 AI Suite. You don't need to define any methods, and the only fields necessary are those needed to hold data from the source. In fact, you'll notice that the '`CanonicalSmartBulb`' Type doesn't use the '`entity`' keyword. This means it isn't persisted either.

Generally speaking, you need to define a new Canonical Type for each type of data source.

Additional Resources

- Developer Documentation
  - https://developer.c3.ai/docs/7.12.25/topic/tutorial-canonical-types
  - https://developer.c3.ai/docs/7.12.25/topic/di-sources-and-canonicals
- C3.ai Academy Videos
  - Learning module: **Data Integration**
    - Canonicals

Transform Types

With a Canonical Type defined to receive new data into the C3 AI Suite, we need to move this data into the data model. Transform Types define this operation. First, a Transform Type mixes the destination Type and then uses the `**transforms**` keyword followed by the source canonical Type to indicate where it should take data from. Secondly, Transform Types support a special syntax for their fields. This syntax defines an expression for each field which takes data from the source Type and produces a result to be stored in the target Type in the given field.

Let's look at an example and discuss some of the syntax:

```
/*
 * Copyright 2009-2020 C3 (www.c3.ai). All Rights Reserved.
 * This material, including without limitation any software, is the confidential trade secret and proprietary
 * information of C3 and its licensors. Reproduction, use and/or distribution of this material in any form is
 * strictly prohibited except as set forth in a written license agreement with C3 and/or its authorized
distributors.
 * This material may be covered by one or more patents or pending patent applications.
 */

/**
 * This type encapsulates the data flow from the {@link CanonicalSmartBulb} to the {@link SmartBulb} type.
 */
type TransformCanonicalSmartBulbToSmartBulb mixes SmartBulb transforms CanonicalSmartBulb {

  id:              ~ expression "SN"
  manufacturer:    ~ expression {id: "Manufacturer"}
  bulbType:        ~ expression "BulbType"
  wattage:         ~ expression "Wattage"
  startDate:       ~ expression "StartDate"
  latitude:        ~ expression "Latitude"
  longitude:       ~ expression "Longitude"
  lumensUOM:       ~ expression "{ \"id\": \"'lumen'\" }"
  powerUOM:        ~ expression "{ \"id\": \"'watt'\" }"
  temperatureUOM:  ~ expression "{ \"id\": \"'degrees_fahrenheit'\" }"
  voltageUOM:      ~ expression "{ \"id\": \"'volt'\" }"

}
```

There are a few different types of fields you might see in a Transform Type, some of which are visible here.

1. Constant

Constant fields are not present in the current example, but they would appear without the '`~ expression`' notation. For example:

```
id: "ID"
value: 1.5
```

In this case, the field is simply set to the constant value listed.

2. Copy Property of Canonical Type

Many transforms are simply copying the value to the right field in the destination Type. This is done with the syntax '`~ expression "<origin_field>"`' , Let's take a look at the first field, '`id`':

```
id: ~ expression "SN"
```

Here, the `~` means the result of the expression is stored at the field. This is true all of the fields here. Then the expression is evaluated in the context of the source Type. So in this case, the property SN is evaluated and copied to the '`id`' field.

3. Fetch field Type via a key

We often want to link our destination Types to other Types which are related. In this example, the SmartBulb Type contains a field '`manufacturer`' which is of Type Manufacturer. How can we grab the correct Manufacturer? This is done with another expression, this time using the '`{}`' notation. Let's have a look:

```
manufacturer: ~ expression {id: "Manufacturer"}
```

This means, we fetch the Manufacturer by the '`id`' key using the value in the expression "Manufacturer" which in this case is the value of the "Manufacturer" field in the source Type for the transform. So the Manufacturer field from the source Type is used to find the Manufacturer Type with id matching, and this is set as the value of the '`manufacturer`' field of the destination Type.

Finally, we can do the same but use a constant as the key. For example:

```
lumensUOM: ~ expression "{ \"id\": \"'lumen'\" }"
```

This can also be written as:

```
lumensUOM: ~ expression { id: "'lumen'" }
```

Here the expression is evaluated with a constant key. This means the Unit Type with id '`lumen`' will always be used for the `lumensUOM` field of the destination Type.

Conclusion

Once you've defined a transform, when data arrives in a Canonical Type the appropriate transform will be called to populate the data model. You can actually define multiple transforms for the same Canonical Type, so you can populate multiple data model Types from the same Canonical Types.

Additional Resources

- Developer Documentation
  - https://developer.c3.ai/docs/7.12.25/topic/tutorial-canonical-transform-types
- C3.ai Academy Videos
  - Learning module: **Data Integration**
    - Transforms

Application Types

Finally, we should mention application Types. There is no special syntax, however, any data you wish to store and later retrieve must end up in a persistable Type. These Types start with the `` `entity` `` keyword. Together your defined Types build a data model against which you can easily make complex queries that previously would've required complex multi-database queries.

For instance, for the SmartBulb Types, we can see that the SmartBulb Type includes several other Types, like 'Manufacturer' mentioned earlier. This allows us to select SmartBulbs based on Manufacturer.

Basic Data Sources

Now that we know what the C3 AI Suite does with data once it enters through a Canonical Type, we can explore how to get data to this entry point.

CSV Files

Probably the easiest method is through a CSV file. Simply define a .csv file with appropriate column names, create a Canonical Type whose fields match those names, and C3 AI Suite can use this data to create Canonical Types. Consider the SmartBulb.csv seed data (we'll get into seed data below) located in the lightbulbAD package at 'seed/CanonicalSmartBulb/SmartBulb.csv':

```
Manufacturer,BulbType,Wattage,SN,StartDate,Latitude,Longitude
Bell,LED,9.5,SMBLB1,2011-01-01T04:00:00-0800,37.48596719,-122.2428196
GE,LED,9.5,SMBLB2,2011-01-01T04:00:00-0800,37.49115273,-122.2319523
GE,LED,9.5,SMBLB3,2011-01-01T04:00:00-0800,37.49099652,-122.2324551
...
```

If this file is placed or sent to the right place the C3 AI Suite will 'ingest' it, which will start the Data Integration system described above.

JSON Files

Data can also be sent in the JSON format. Generally, the JSON format can have two formats:

First, we define an array of a specific Type:

```
{
  "type": "[Unit]",
  "value": [
    { "id": "PCS", "name": "pieces", "symbol": "PCS", "concept": "count" },
    ...
  ]
}
```

Second, we define an array of multiple possibly different Types:

```
[
  {
    "type": "Unit",
    "id": "PCS",
    "name": "pieces",
    "symbol": "PCS",
    "concept": "count"
  },
  {
    "type": "Unit",
    ...
  },
  ...
]
```

Consider the following data which can work with the lightbulbAD package:

```
[
  {
    "status": 1,
    "end": "2012-10-23T12:00:00.000-07:00",
    "temperature": 121,
    "power": 63,
    "lumens": 14,
    "start": "2012-10-23T11:00:00.000-07:00",
    "parent": {
      "id": "SBMS_serialNo_SMBLB74"
    },
    "voltage": 944.6,
    "type": "SmartBulbMeasurement",
    "id": "SBMS_serialNo_SMBLB74#BJ"
  },
  {
    "status": 1,
    "end": "2013-09-12T22:00:00.000-07:00",
    "temperature": 13,
    "power": 58,
    "lumens": 919.1,
    "start": "2013-09-12T21:00:00.000-07:00",
    "parent": {
      "id": "SBMS_serialNo_SMBLB74"
    },
    "voltage": 120,
    "type": "SmartBulbMeasurement",
    "id": "SBMS_serialNo_SMBLB74#V"
  },
```

Seed Data

Now that we've discussed some basic data formats, we can talk about where to put the data. The first place is the 'seed' directory in your package. Place your seed data file in a subdirectory of the 'seed' folder with the same name as the canonical Type the data is destined for. For example:

```
seed/CanonicalSmartBulb/SmartBulb.csv
```

Now, when you provision your Package, C3 AI Suite will take this file and run it through the Data Ingestion pipeline.

One negative to this method is it's primarily for small amounts of data. Any data you send in this method will bloat your package and, if you choose to use the browser provisioner, this data will be loaded into browser memory which can be quite limiting. We'll now discuss another method to send data to your C3 Cluster which can work with practically unlimited amounts of data.

Sending Data via POST and RESTful API

C3 AI Suite creates a REST API endpoint for each Canonical Type. This is accessible through the path: '`https://<vanity_url>/import/1/<tenant>/<tag>/<CanonicalType>/<FileName>`', where:

- **<vanity_url>**: Your Vanity URL.
- **<tenant>**: Your tenant.
- **<tag>**: Your tag.
- **<CanonicalType>**: The target canonical Type.
- **<FileName>**: Destination name of the file.

You can send data to this endpoint with an HTTP PUT.

When sending the HTTP PUT ensure the following headers are defined:

- **Authorization**: An auth token generated by '`Authenticator.generateC3AuthToken()`'.
- **Content-Type**: Use 'text/csv' for a CSV file or 'application/json' for a JSON file.

This import API endpoint acts to essentially copy a file from your local file system into the C3 Cluster. C3 AI Suite will associate this file to your tenant/tag and Canonical Type, and it will remember the <FileName> in case you try to send it again.

Helper Scripts and Tools

There are a couple of tools you can use to execute these HTTP PUT commands.

send-file.py: DTI Developed helper script

If you download the git repository available **here**:

We can use the script `**send-file.py**`. This script uses the python requests module to form the HTTP PUT command. Helpfully, it also reports percent completion which is great if the file you're uploading is large or your internet connection is slow. To use this command, first clone the github repo, then execute the script inside with python (assuming you've installed the requests module).

You want to execute it as follows:

```
python send-file.py --vanity-url <vanity_url> --tenant <tenant> --tag <tag> --api-endpoint <endpoint> --file
<path_to_file> --auth-token <auth_token>
```

Where:

- **<vanity_url>**: Your Vanity URL.
- **<tenant>**: Your tenant.
- **<tag>**: Your tag.
- **<endpoint>**: The 'location' you want to copy the file to, after 'https://<vanity_url>/import/1/<tenant>/<tag>/'. This is usually '<CanonicalType>/<FileName>'.
- **<path_to_file>**: The path to the file you want to upload on your local file system.
- **<auth_token>**: An authorization token generated by '`Authenticator.generateC3AuthToken()`'.

curl

You can form an appropriate PUT command with curl. Please follow the detailed instructions in the C3 AI **official documentation**.

Postman

Postman is a GUI tool dedicated to this type of custom POST/PUT command creation. Please see the C3 AI documentation on how to use Postman **here**.

Monitoring Data Integration

Source Systems and Source Collections

See the official C3 AI documentation **here**.

SourceFile

See the official C3 AI documentation **here**.

Checking SourceFile Integration And Troubleshooting

When a .csv file is sent to C3, it is broken into chunks, and processing of the chunks proceeds in parallel. These chunks are stored in the `SourceChunk` Type and their status can be checked with the `SourceChunkStatus` Type. When the source file is being processed, we can examine the `Invalidation Queue` to see that the `SourceQueue` is processing jobs.

```
c3Grid(InvalidationQueue.countAll())
```

Some times a queue can accumulate errors. The errors for a specific queue can be inspected with `<QueueName>.errors()`.

If processing has finished, or we want a closer look, we can inspect the chunk status like so:

```
c3Grid(SourceChunkStatus.fetch())
```

The `SourceChunkStatus` type has several useful fields for tracking processing progress.

```
state, count, successful, skipped, failed, totalErrorCount
```

Most useful here is the '`state`' field. This field will be '`completed`' if processing of the chunk is completed. Sometimes processing of chunks does not start properly. In this situation, the state will be stuck on '`initial`'.

We can trigger a chunk to be reprocessed, or force a chunk to start processing as follows:

```
SourceChunk.forId(<chunk_id>).process(true)
```

Try to force your failed or frozen chunks in this way.

Complex Data Sources

Custom External Database

It is also possible to 'add' an external database into C3 AI Suite as well. Through the `SqlSourceSystem`, `SqlSourceCollection`, and External Types, you can define a 'persistable' Type whose storage exists on the external database. When you execute fetch commands on this Type, instead of querying an internal database as the C3 AI Suite normally does, it will call out to this external database to perform the appropriate query. Please see the detailed C3 AI developer documentation describing how this works **here**.

C3 AI Supported Database Technologies

C3 AI supports numerous connectors to existing databases. Please see the detailed C3 AI developer documentation on what's available and how to use them **here**.

Other Data Formats

Support for other data formats will need to be visited on a case-by-case basis. Contact the DTI at help@c3dti.ai for help determining the right way to ingest your custom data format.