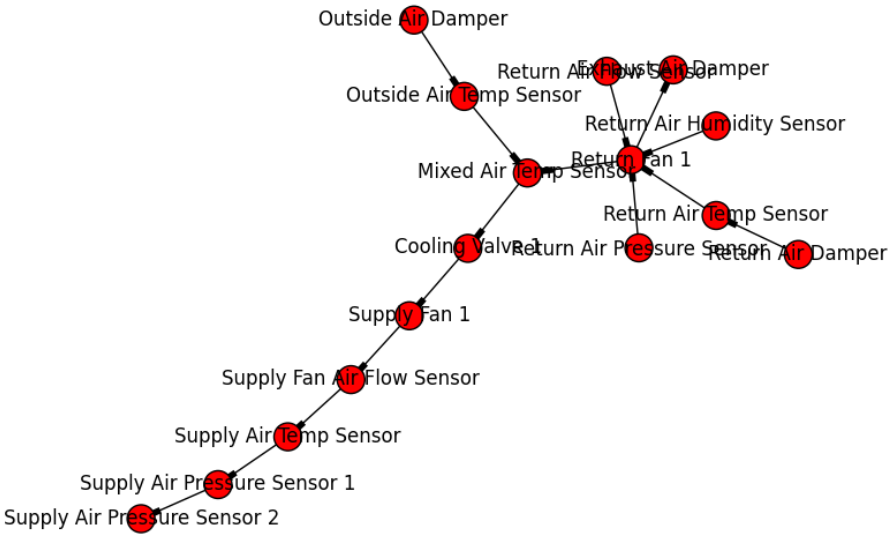
AppStack Query Language

Gabe Fierro

1 Introduction

The purpose of this query language is to facilitate the navigation of and collection of data within the functional graphs created by the user using the AppStack backend. This document has the goal of educating you, dear reader, on the structure of the graphs thus created and how one might use this query language.

1.1 Graph Nodes



Here is a sample functional graph of an Air Handler. A graph consists of a series of **nodes** (representing physical objects in the system) connected by directed edges (representing the flow of the process from one node to another). As we can see in the graph above, the "Outside Air Damper" feeds into the "Outside Air Temp Sensor", which in turn feeds the "Mixed Air Temp Sensor", which is also fed by the "Return Fan", and so on. In this way we are able to capture the functional relationships between pieces in the system.

Each **node** is identified by three different attributes of differing degrees of uniqueness.

- **TYPE**: each object is identified by one of the 2-3 character types listed in `node_types.type_dict` (or listed by calling `node_types.list_types()`). For example: "AH" → "Air Handler", "DMP" → "Damper".

- **NAME**: each object is also initialized with a name, which is a string of arbitrary length, including "[a-zA-Z]-:_". For example: "Air Handler '", "Return Fan"
- **UID**: each object is additionally provisioned with a unique identification number, which is a UUID created upon initialization of the object. For example: "c89bdbdd-e780-4f38-9523-9197d8161f6f".

1.2 Graphs Within Graphs

Although this has not yet been integrated into the current sample code, the AppStack architecture supports (and indeed encourages) the encapsulation of smaller functional pieces (such as the figure above) within higher level objects which are themselves nodes in a larger graph.

For example, the graph above *functions* as an Air Handler, so it would be in another, larger, node that would be labelled as of the type "AH". This node would be part of a larger graph containing other nodes such as a "cooling loop" and "hot water loop" that would feed into the "air handler", represented by a directed edge.

2 Query Language

2.1 What Can I Query?

Any valid query that you provide to the interpreter will return to you a set of nodes in the set of graphs that have been registered with the interpreter (currently it is only the graph created in `test.py`).

You may query over any subset of the nodes that can be indicated by **type**, **name**, **uid**, or **relationship**.

2.2 Prefixes

In order to specify different objects in your queries, you use a small set of prefixes to help the interpreter distinguish.

Prefix	What it Specifies	Examples of a **Prefixed Expression**
#	Object Type/Class	**#AH** (set of all Air Handlers), **#DMP** (set of all Dampers)
$	Object Name	**$Air Handler 1** (set of all air handlers named "Air Handler '")
%	Object UID	**%ab61b939-a133-4d76-b9c4-a5d6fab7abf5** (the object with this UID)
@	Variable	**@x** (returns the set previously stored in this variable)

2.3 Query Format

Queries take the basic form of:

```
TARGET [DELIMITER SET [DELIMITER SET [...]]]
```

`TARGET,SET`	prefixed expression as indicated in the above table such as "#AH", "@varname", "$Name of object"
`DELIMITER`	>: in the expression "X > Y", X is "upstream of" or "feeds into" Y. <: in the expression "X < Y", X is "downstream of" or "is supplied by" Y

The `TARGET` specifies the nature of the object(s) you'd like to receive upon resolving your query. `TARGET` and `SET` both take the form of either a **prefixed expression**, which follows from the table above. `DELIMITER`s specify the relationships between these sets.

2.4 Constructing a Query

We construct a query from **left** to **right** by starting with our target set and appending a series of filters. Queries are perhaps most easily constructed by writing an English sentence describing the point(s) we are trying to find in the graph. For example:

"I want all sensors that measure the air from the return fan in this air handler"

We can immediately see that our **target** are "sensors", so we begin the query by specifying that we want to return objects of `TYPE` "sensor", or "#SEN":

```
#SEN
```

Now because we want sensors that measure air **from** the return fan, this means that we want all sensors that are **downstream** of the fan, so we use the "<" delimiter:

```
#SEN <
```

Lastly, we need to specify what exactly these sensors are downstream **of**. We can specify the Return Fan by name ("$Return Fan 1"):

```
#SEN < $Return Fan 1
```

However, the name "Return Fan 1" isn't guaranteed to be unique (other fans in other air handlers could be named "Return Fan 1"), so we can use the UID of the object instead:

```
#SEN < %ab61b939-a133-4d76-b9c4-a5d6fab7abf5
```

which returns the correct list of sensors:

Mixed Air Temp Sensor d1786922-37e5-4bc9-8f4d-79c897c3d517
Supply Fan Air Flow Sensor 146caeaa-482f-447a-8909-a30521ec56c2
Supply Air Temp Sensor 49501c37-78c6-41fa-9659-f5aea37b9eb6
Supply Air Pressure Sensor 1 7e171958-0eef-4c33-8373-a453ca63da55
Supply Air Pressure Sensor 2 60c9cd79-e0ca-4710-a7a2-d1c428d946f6

Additionally, we can save the result of this query in a variable:

```
@sensors = #SEN < %ab61b939-a133-4d76-b9c4-a5d6fab7abf5
```

and then run further queries on that variable to search this new domain.

2.5 Help

Typing "help" into the interpreter gives you some additional commands you can use to help you learn what exactly you can query.

You run the interpreter by running `python lexerparser.py` in the `appstack` directory.