G2Engine Reference

More information:

1. GitHub repository
2. Senzing documentation

Prepare environment

Initialize Senzing configuration

Run senzing-G2ConfigMgr-reference.ipynb to install a Senzing Engine configuration in the database.

Initialize python environment

In [ ]:

```
import os
import sys
import json

# For RenderJSON

import uuid
from IPython.display import display_javascript, display_html, display
```

Helper class for JSON rendering

A class for pretty-printing JSON. Not required by Senzing, but helps visualize JSON.

In [ ]:

```
class RenderJSON(object):
    def __init__(self, json_data):
        if isinstance(json_data, dict):
            self.json_str = json.dumps(json_data)
        elif isinstance(json_data, bytearray):
            self.json_str = json_data.decode()
        else:
            self.json_str = json_data
        self.uuid = str(uuid.uuid4())

    def _ipython_display_(self):
        display_html('<div id="{}" style="height:100%; width:100%; background-color: LightCyan"></div>'.format(self.uuid), raw=True)
        display_javascript("""
        require(["https://rawgit.com/caldwell/renderjson/master/renderjson.js"], function() {
        document.getElementById('%s').appendChild(renderjson(%s))
        });
        """ % (self.uuid, self.json_str), raw=True)
```

System path

Update system path.

In [ ]:

```
python_path = "{0}/python".format(
    os.environ.get("SENZING_G2_DIR", "/opt/senzing/g2"))
sys.path.append(python_path)
```

Initialize variables

Create variables used for G2Engine.

In [ ]:

```
module_name = 'pyG2EngineForAddRecord'

config_path = os.environ.get("SENZING_ETC_DIR", "/etc/opt/senzing")
support_path = os.environ.get("SENZING_DATA_VERSION_DIR", "/opt/senzing/data")

resource_path = "{0}/resources".format(
    os.environ.get("SENZING_G2_DIR", "/opt/senzing/g2"))

sql_connection = os.environ.get(
    "SENZING_SQL_CONNECTION", "sqlite3://na:na@/var/opt/senzing/sqlite/G2C.db")

verbose_logging = False

senzing_config_dictionary = {
    "PIPELINE": {
        "CONFIGPATH": config_path,
        "SUPPORTPATH": support_path,
        "RESOURCEPATH": resource_path
    },
    "SQL": {
        "CONNECTION": sql_connection,
    }
}

senzing_config_json = json.dumps(senzing_config_dictionary)
```

G2Engine

In [ ]:

```
from G2Engine import G2Engine
import G2Exception
```

G2Engine initialization

To start using Senzing G2Engine, create and initialize an instance. This should be done once per process. The `initV2()` method accepts the following parameters:

- **module_name:** A short name given to this instance of the G2Engine object.
- **senzing_config_json:** A JSON string containing configuration parameters.
- **verbose_logging:** A boolean which enables diagnostic logging.
- **config_id:** (optional) The identifier value for the engine configuration can be returned here.

Calling this function will return "0" upon success.

In [ ]:

```
g2_engine = G2Engine()

return_code = g2_engine.initV2(
    module_name,
    senzing_config_json,
    verbose_logging)

print("Return Code: {0}".format(return_code))
```

initWithConfigIDV2

Alternatively `initWithConfigIDV2()` can be used to specify a configuration. For more information, see http://docs.senzing.com/?python.

reinitV2

The `reinitV2()` function may be used to reinitialize the engine using a specified initConfigID. See http://docs.senzing.com/?python.

primeEngine

The `primeEngine()` method may optionally be called to pre-initialize some of the heavier weight internal resources of the G2 engine.

In [ ]:

```
return_code = g2_engine.primeEngine()
print("Return Code: {0}".format(return_code))
```

getActiveConfigID

Call `getActiveConfigID()` to return an identifier for the loaded Senzing engine configuration. The call will assign a long integer to a user-designated variable -- the function itself will return "0" upon success. The `getActiveConfigID()` method accepts one parameter as input:

- **configuration_id:** The identifier value for the engine configuration. The result of function call is returned here

In [ ]:

```
configuration_id = bytearray()
return_code = g2_engine.getActiveConfigID(configuration_id)

print("Return code: {0}\nConfiguration id: {1}".format(
    return_code,
    configuration_id.decode()))
```

exportConfig

Call `exportConfig()` to retrieve your Senzing engine's configuration. The call will assign a JSON document to a user-designated buffer, containing all relevant configuration information -- the function itself will return "0" upon success. The exportConfig function accepts the following parameters as input:

- **response_bytearray:** The memory buffer to retrieve the JSON configuration document
- **config_id_bytearray:** The identifier value for the engine configuration can be returned here.

In [ ]:

```
response_bytearray = bytearray()
config_id_bytearray = bytearray()

return_code = g2_engine.exportConfig(response_bytearray, config_id_bytearray)

print("Return Code: {0}\nConfiguration ID: {1}".format(
    return_code,
    config_id_bytearray.decode()))

RenderJSON(response_bytearray)
```

stats

Call `stats()` to retrieve workload statistics for the current process. These statistics will automatically reset after retrieval.

- **response_bytearray:** A memory buffer for returning the response document. If an error occurred, an error response is stored here.

In [ ]:

```
response_bytearray = bytearray()

return_code = g2_engine.stats(response_bytearray)

print("Return Code: {0}".format(return_code))
RenderJSON(response_bytearray)
RenderJSON(response_bytearray)
```

getRepositoryLastModifiedTime

Call `getRepositoryLastModifiedTime()` to obtain the last modified time of the Senzing repository,measured in the number of seconds between the last modified time and January 1, 1970 12:00am GMT (epoch time). The call will assign a long integer to a user-designated buffer -- the function itself will return "0" upon success. The getRepositoryLastModifiedTime() method accepts one parameter as input:

- **last_modified_unixtime:** The last modified time. The result of function call is returned here

In [ ]:

```
last_modified_timestamp = bytearray()

return_code = g2_engine.getRepositoryLastModifiedTime(last_modified_timestamp)

# Human readable output.

from datetime import datetime
last_modified_unixtime = int(int(last_modified_timestamp.decode()) / 1000)
last_modified_datetime = datetime.fromtimestamp(last_modified_unixtime)

print("Return Code: {0}\nLast modified timestamp: {1}\nLast modified time: {2}"
     .format(
        return_code,
        last_modified_timestamp.decode(),
        last_modified_datetime))
```

Insert

Insert parameters

The following variables are used as parameters to the Senzing API. Documentation for `g2_engine_flags` values is at http://docs.senzing.com/?python

In [ ]:

```
datasource_code_1 = "TEST"
record_id_1 = "1"
datasource_code_2 = "TEST"
record_id_2 = "2"
datasource_code_3 = "TEST"
record_id_3 = "3"
datasource_code_4 = "TEST"
record_id_4 = "4"
datasource_code_5 = "TEST"
record_id_5 = "5"
datasource_code_6 = "TEST"
record_id_6 = "6"
datasource_code_7 = "TEST"
record_id_7 = "7"

load_id = None
g2_engine_flags = G2Engine.G2_EXPORT_DEFAULT_FLAGS
```

Initial data.

In [ ]:

```
data = {
    "NAMES": [{
        "NAME_TYPE": "PRIMARY",
        "NAME_LAST": "Smith",
        "NAME_FIRST": "John",
        "NAME_MIDDLE": "M"
    }],
    "PASSPORT_NUMBER": "PP11111",
    "PASSPORT_COUNTRY": "US",
    "PASSPORT_COUNTRY": "US",
    "DRIVERS_LICENSE_NUMBER": "DL11111",
    "SSN_NUMBER": "111-11-1111"
}
data_as_json = json.dumps(data)
```

addRecord

Once the Senzing engine is initialized, use `addRecord()` to load a record into the Senzing repository -- `addRecord()` can be called as many times as desired and from multiple threads at the same time. The `addRecord()` function returns "0" upon success, and accepts four parameters as input:

- **datasource_code:** The name of the data source the record is associated with. This value is configurable to the system
- **record_id:** The record ID, used to identify distinct records
- **data_as_json:** A JSON document with the attribute data for the record
- **load_id:** The observation load ID for the record; value can be null and will default to data_source

In [ ]:

```
return_code = g2_engine.addRecord(
    datasource_code_1,
    record_id_1,
    data_as_json,
    load_id)

print("Return Code: {0}".format(return_code))
```

addRecordWithReturnedRecordID

Alternatively `addRecordWithReturnedRecordID()` can be used to add a record. For more information, see http://docs.senzing.com/?python.

addRecordWithInfo

Use if you would like to know what resolved entities were modified when adding the new record. It behaves identically to addRecord(), but returns a json document containing the IDs of the affected entities. It accepts the following parameters:

- **datasource_code:** The name of the data source the record is associated with. This value is configurable to the system.
- **record_id:** The record ID, used to identify distinct records
- **data_as_json:** A JSON document with the attribute data for the record
- **response_bytearray:** A memory buffer for returning the response document; if an error occurred, an error response is stored here
- **load_id:** The observation load ID for the record; value can be null and will default to data_source
- **g2_engine_flags:** Control flags for specifying what data about the entity to retrieve

In [ ]:

```
response_bytearray = bytearray()

return_code = g2_engine.addRecordWithInfo(
    datasource_code_1,
    record_id_1,
    data_as_json,
    response_bytearray,
    load_id,
    g2_engine_flags)

print("Return Code: {0}".format(return_code))
RenderJSON(response_bytearray)
```

Search

Record search

getRecordV2

Use `getRecordV2()` to retrieve a single record from the data repository; the record is assigned in JSON form to a user-designated buffer, and the function itself returns "0" upon success. Once the Senzing engine is initialized, `getRecordV2()` can be called as many times as desired and from multiple threads at the same time. The `getRecordV2()` function accepts the following parameters as input:

- **datasource_code:** The name of the data source the record is associated with. This value is configurable to the system.
- **record_id:** The record ID, used to identify the record for retrieval
- **g2_engine_flags:** Control flags for specifying what data about the record to retrieve.
- **response_bytearray:** A memory buffer for returning the response document; if an error occurred, an error response is stored here.

In [ ]:

```
response_bytearray = bytearray()

return_code = g2_engine.getRecordV2(
    datasource_code_1,
    record_id_1,
    g2_engine_flags,
    response_bytearray)

print("Return Code: {0}".format(return_code))
RenderJSON(response_bytearray)
```

The function `getRecordV2()` is an improved version of `getRecord()` that also allows you to use control flags. The `getRecord()` function has been deprecated.

Entity Search

getEntityByRecordIDV2

Entity searching is a key component for interactive use of Entity Resolution intelligence. The core Senzing engine provides real-time search capabilities that are easily accessed via the Senzing API. Senzing offers methods for entity searching, all of which can be called as many times as desired and from multiple threads at the same time (and all of which return "0" upon success).

Use `getEntityByRecordIDV2()` to retrieve entity data based on the ID of a resolved identity. This function accepts the following parameters as input:

- **datasource_code:** The name of the data source the record is associated with. This value is configurable to the system.
- **record_id:** The numeric ID of a resolved entity
- **g2_engine_flags:** Control flags for specifying what data about the entity to retrieve.
- **response_bytearray:** A memory buffer for returning the response document; if an error occurred, an error response is stored here.

In [ ]:

```
response_bytearray = bytearray()

return_code = g2_engine.getEntityByRecordIDV2(
    datasource_code_1,
    record_id_1,
    g2_engine_flags,
    response_bytearray)

response_dictionary = json.loads(response_bytearray)
entity_id_1 = response_dictionary["RESOLVED_ENTITY"]["ENTITY_ID"]

print("Return Code: {0}".format(return_code))
RenderJSON(response_bytearray)
```

getEntityByEntityIDV2

Entity searching is a key component for interactive use of Entity Resolution intelligence. The core Senzing engine provides real-time search capabilities that are easily accessed via the Senzing API. Senzing offers methods for entity searching, all of which can be called as many times as desired and from multiple threads at the same time (and all of which return "0" upon success).

called as many times as desired and from multiple threads at the same time (and all of which return "0" upon success).

Use `getEntityByEntityIDV2()` to retrieve entity data based on the ID of a resolved identity. This function accepts the following parameters as input:

- **entity_id:** The numeric ID of a resolved entity
- **g2_engine_flags:** Control flags for specifying what data about the entity to retrieve.
- **response_bytearray:** A memory buffer for returning the response document; if an error occurred, an error response is stored here.

In [ ]:

```
response_bytearray = bytearray()

return_code = g2_engine.getEntityByEntityIDV2(
    entity_id_1,
    g2_engine_flags,
    response_bytearray)

print("Return Code: {0}".format(return_code))
RenderJSON(response_bytearray)
```

Search By Attributes

searchByAttributes

Entity searching is a key component for interactive use of Entity Resolution intelligence. The core Senzing engine provides real-time search capabilities that are easily accessed via the Senzing API. Senzing offers a method for entity searching by attributes, which can be called as many times as desired and from multiple threads at the same time (and all of which return "0" upon success).

Use `searchByAttributes()` to retrieve entity data based on a user-specified set of entity attributes. This function accepts the following parameters as input:

- **data_as_json:** A JSON document with the attribute data to search for.
- **response_bytearray:** A memory buffer for returning the response document; if an error occurred, an error response is stored here.

In [ ]:

```
response_bytearray = bytearray()
return_code = g2_engine.searchByAttributes(data_as_json, response_bytearray)

print("Return Code: {0}".format(return_code))
RenderJSON(response_bytearray)
```

searchByAttributesV2

This function is similar but preferable to the searchByAttributes() function. This function has improved functionality and a better standardized output structure.

Use `searchByAttributesV2()` to retrieve entity data based on a user-specified set of entity attributes. This function accepts the following parameters as input:

- **data_as_json:** A JSON document with the attribute data to search for.
- **g2_engine_flags:** Operational flags
- **response_bytearray:** A memory buffer for returning the response document; if an error occurred, an error response is stored here.

In [ ]:

```
response_bytearray = bytearray()

return_code = g2_engine.searchByAttributesV2(
    data_as_json,
    g2_engine_flags,
    response_bytearray)

print("Return Code: {0}".format(return_code))
```

```
RenderJSON(response_bytearray)
```

Finding Paths

The `findPathByEntityID()` and `findPathByRecordID()` functions can be used to find single relationship paths between two entities. Paths are found using known relationships with other entities.

Entities can be searched for by either Entity ID or by Record ID, depending on which function is chosen.

These functions have the following parameters:

- **entity_id_2:** The entity ID for the starting entity of the search path
- **entity_id_3:** The entity ID for the ending entity of the search path
- **datasource_code_2:** The data source for the starting entity of the search path
- **datasource_code_3:** The data source for the ending entity of the search path
- **record_id_2:** The record ID for the starting entity of the search path
- **record_id_3:** The record ID for the ending entity of the search path
- **max_degree:** The number of relationship degrees to search

The functions return a JSON document that identifies the path between the entities, and the information on the entities in question. The document contains a section called "ENTITY_PATHS" which gives the path from one entity to the other. Example:

```
{
  "START_ENTITY_ID": 10,
  "END_ENTITY_ID": 13,
  "ENTITIES": [10, 11, 12, 13]
}
```

If no path was found, then the value of ENTITIES will be an empty list.

The response document also contains a separate ENTITIES section, with the full information about the resolved entities along that path.

First you will need to create some records so that you have some that you can compare. Can you see what is the same between this record and the previous one?

In [ ]:

```
data = {
    "NAMES": [{
        "NAME_TYPE": "PRIMARY",
        "NAME_LAST": "Miller",
        "NAME_FIRST": "Max",
        "NAME_MIDDLE": "W"
    }],
    "SSN_NUMBER": "111-11-1111"
}
data_as_json = json.dumps(data)

return_code = g2_engine.replaceRecord(
    datasource_code_2,
    record_id_2,
    data_as_json,
    None)

print("Return Code: {0}".format(return_code))
```

Replace values for Record #3

In [ ]:

```
data = {
    "NAMES": [{
        "NAME_TYPE": "PRIMARY",
        "NAME_LAST": "Miller",
        "NAME_FIRST": "Mildred"
    }],
    "SSN_NUMBER": "111-11-1111"
}
```

```
data_as_json = json.dumps(data)

return_code = g2_engine.replaceRecord(
    datasource_code_3,
    record_id_3,
    data_as_json,
    None)

print("Return Code: {0}".format(return_code))
```

Locate "entity identifier" for Record #1

In [ ]:

```
response_bytearray = bytearray()

return_code = g2_engine.getEntityByRecordID(
    datasource_code_1,
    record_id_1,
    response_bytearray)

response_dictionary = json.loads(response_bytearray)
entity_id_1 = response_dictionary["RESOLVED_ENTITY"]["ENTITY_ID"]

print("Return Code: {0}\nEntity ID: {1}".format(return_code, entity_id_1))
```

Locate "entity identifier" for Record #2

In [ ]:

```
response_bytearray = bytearray()

return_code = g2_engine.getEntityByRecordID(
    datasource_code_2,
    record_id_2,
    response_bytearray)

response_dictionary = json.loads(response_bytearray)
entity_id_2 = response_dictionary["RESOLVED_ENTITY"]["ENTITY_ID"]

print("Return Code: {0}\nEntity ID: {1}".format(return_code, entity_id_2))
RenderJSON(response_bytearray)
```

Locate "entity identifier" for Record #3

In [ ]:

```
response_bytearray = bytearray()

return_code = g2_engine.getEntityByRecordID(
    datasource_code_3,
    record_id_3,
    response_bytearray)

response_dictionary = json.loads(response_bytearray)
entity_id_3 = response_dictionary["RESOLVED_ENTITY"]["ENTITY_ID"]

print("Return Code: {0}\nEntity ID: {1}".format(return_code, entity_id_3))
RenderJSON(response_bytearray)
```

findPathByEntityID

In [ ]:

```
# Define search variables.

max_degree = 3

# Find the path by entity ID.
```

```
response_bytearray = bytearray([])

return_code = g2_engine.findPathByEntityID(
    entity_id_2,
    entity_id_3,
    max_degree,
    response_bytearray)

# Print the results.

print("Return Code: {0}".format(return_code))
RenderJSON(response_bytearray)
```

findPathByEntityIDV2

The function `findPathByEntityIDV2()` is an improved version of `findPathByEntityID()` that also allow you to use control flags.

In [ ]:

```
# Define search variables.

max_degree = 3

# Find the path by entity ID.

response_bytearray = bytearray([])

return_code = g2_engine.findPathByEntityIDV2(
    entity_id_2,
    entity_id_3,
    max_degree,
    g2_engine_flags,
    response_bytearray)

# Print the results.

print("Return Code: {0}".format(return_code))
RenderJSON(response_bytearray)
```

findPathByRecordID

In [ ]:

```
# Define search variables.

max_degree = 3

# Find the path by record ID.

response_bytearray = bytearray([])

return_code = g2_engine.findPathByRecordID(
    datasource_code_2, record_id_2,
    datasource_code_3, record_id_3,
    max_degree,
    response_bytearray)

# Print the results.

print("Return Code: {0}".format(return_code))
RenderJSON(response_bytearray)
```

findPathByRecordIDV2

The function `findPathByRecordIDV2()` is an improved version of `findPathByRecordID()` that also allow you to use control flags.

In [ ]:

```
# Define search variables.
```

```
max_degree = 3

# Find the path by record ID.

response_bytearray = bytearray([])

return_code = g2_engine.findPathByRecordIDV2(
    datasource_code_2, record_id_2,
    datasource_code_3, record_id_3,
    max_degree,
    g2_engine_flags,
    response_bytearray)

# Print the results.

print("Return Code: {0}".format(return_code))
RenderJSON(response_bytearray)
```

Finding Paths with Exclusions

The `findPathExcludingByEntityID()` and `findPathExcludingByRecordID()` functions can be used to find single relationship paths between two entities. Paths are found using known relationships with other entities. In addition, it will find paths that exclude certain entities from being on the path.

Entities can be searched for by either Entity ID or by Record ID, depending on which function is chosen. Additionally, entities to be excluded can also be specified by either Entity ID or by Record ID.

When excluding entities, the user may choose to either (a) strictly exclude the entities, or (b) prefer to exclude the entities, but still include them if no other path is found. By default, entities will be strictly excluded. A "preferred exclude" may be done by specifying the `G2_FIND_PATH_PREFER_EXCLUDE` control flag.

These functions have the following parameters:

- **entity_id_2:** The entity ID for the starting entity of the search path
- **entity_id_3:** The entity ID for the ending entity of the search path
- **datasource_code_2:** The data source for the starting entity of the search path
- **datasource_code_3:** The data source for the ending entity of the search path
- **record_id_2:** The record ID for the starting entity of the search path
- **record_id_3:** The record ID for the ending entity of the search path
- **max_degree:** The number of relationship degrees to search
- **excluded_entities_as_json:** Entities that should be avoided on the path (JSON document)
- **g2_engine_flags:** Operational flags

findPathExcludingByEntityID

In [ ]:

```
# Define search variables.

max_degree = 4
excluded_entities = {
    "ENTITIES": [{
        "ENTITY_ID": entity_id_1
    }]}
excluded_entities_as_json = json.dumps(excluded_entities)

# Find the path by entity ID.

response_bytearray = bytearray([])

return_code = g2_engine.findPathExcludingByEntityID(
    entity_id_2,
    entity_id_3,
    max_degree,
    excluded_entities_as_json,
    g2_engine_flags,
    response_bytearray)

# Print the results.

print("Return Code: {0}".format(return_code))
```

```
RenderJSON(response_bytearray)
```

findPathExcludingByRecordID

In [ ]:

```
# Define search variables.

excluded_records = {
    "RECORDS": [{
        "RECORD_ID": record_id_1,
        "DATA_SOURCE": datasource_code_1
    }]}
excluded_records_as_json = json.dumps(excluded_records)

# Find the path by record ID.

response_bytearray = bytearray([])

return_code = g2_engine.findPathExcludingByRecordID(
    datasource_code_2, record_id_2,
    datasource_code_3, record_id_3,
    max_degree,
    excluded_records_as_json,
    g2_engine_flags,
    response_bytearray)

# Print the results.

print("Return Code: {0}".format(return_code))
RenderJSON(response_bytearray)
```

Finding Paths with Required Sources

The `findPathIncludingSourceByEntityID()` and `findPathIncludingSourceByRecordID()` functions can be used to find single relationship paths between two entities. In addition, one of the enties along the path must include a specified data source.

Entities can be searched for by either Entity ID or by Record ID, depending on which function is chosen. The required data source or sources are specified by a json document list.

Specific entities may also be excluded, using the same methodology as the `findPathExcludingByEntityID()` and `findPathExcludingByRecordID()` functions use.

These functions have the following parameters:

- **entity_id_2:** The entity ID for the starting entity of the search path
- **entity_id_3:** The entity ID for the ending entity of the search path
- **datasource_code_2:** The data source for the starting entity of the search path
- **datasource_code_3:** The data source for the ending entity of the search path
- **record_id_2:** The record ID for the starting entity of the search path
- **record_id_3:** The record ID for the ending entity of the search path
- **max_degree:** The number of relationship degrees to search
- **excluded_entities_as_json:** Entities that should be avoided on the path (JSON document)
- **required_dsrcs_as_json:** Entities that should be avoided on the path (JSON document)
- **g2_engine_flags:** Operational flags

findPathIncludingSourceByEntityID

In [ ]:

```
# Define search variables.

max_degree = 4
excluded_entities = {
    "ENTITIES": [{
        "ENTITY_ID": entity_id_1
    }]}
excluded_entities_as_json = json.dumps(excluded_entities)
```

```
required_dsrcs = {
    "DATA_SOURCES": [
        datasource_code_1
    ]}
required_dsrcs_as_json = json.dumps(excluded_entities)

# Find the path by entity ID.

response_bytearray = bytearray([])

return_code = g2_engine.findPathIncludingSourceByEntityID(
    entity_id_2,
    entity_id_3,
    max_degree,
    excluded_entities_as_json,
    required_dsrcs_as_json,
    g2_engine_flags,
    response_bytearray)

# Print the results.

print("Return Code: {0}".format(return_code))
RenderJSON(response_bytearray)
```

findPathIncludingSourceByRecordID

In [ ]:

```
# Define search variables.

excluded_records = {
    "RECORDS": [{
        "RECORD_ID": record_id_1,
        "DATA_SOURCE": datasource_code_1
    }]}
excluded_records_as_json = json.dumps(excluded_records)

# Find the path by record ID.

response_bytearray = bytearray([])

return_code = g2_engine.findPathIncludingSourceByRecordID(
    datasource_code_2, record_id_2,
    datasource_code_3, record_id_3,
    max_degree,
    excluded_records_as_json,
    required_dsrcs_as_json,
    g2_engine_flags,
    response_bytearray)

# Print the results.

print("Return Code: {0}".format(return_code))
RenderJSON(response_bytearray)
```

Finding Networks

The `findNetworkByEntityID()` and `findNetworkByRecordID()` functions can be used to find all entities surrounding a requested set of entities. This includes the requested entities, paths between them, and relations to other nearby entities.

Entities can be searched for by either Entity ID or by Record ID, depending on which function is chosen.

These functions have the following parameters:

- **entity_list_as_json:** A list of entities, specified by Entity ID (JSON document)
- **record_list_as_json:** A list of entities, specified by Record ID (JSON document)
- **max_degree:** The maximum number of degrees in paths between search entities
- **buildout_degree:** The number of degrees of relationships to show around each search entity
- **max_entities:** The maximum number of entities to return in the discovered network

They also have various arguments used to return response documents

The functions return a JSON document that identifies the path between the each set of search entities (if the path exists), and the information on the entities in question (search entities, path entities, and build-out entities.

findNetworkByEntityID

In [ ]:

```
# Define search variables.

entity_list = {
    "ENTITIES": [{
        "ENTITY_ID": entity_id_1
    }, {
        "ENTITY_ID": entity_id_2
    }, {
        "ENTITY_ID": entity_id_3
    }]}
entity_list_as_json = json.dumps(entity_list)
max_degree = 2
buildout_degree = 1
max_entities = 12

# Find the network by entity ID.

response_bytearray = bytearray()

return_code = g2_engine.findNetworkByEntityID(
    entity_list_as_json,
    max_degree,
    buildout_degree,
    max_entities,
    response_bytearray)

# Print the results.

print("Return Code: {0}".format(return_code))
RenderJSON(response_bytearray)
```

findNetworkByEntityIDV2

The function `findNetworkByEntityIDV2()` is an improved version of `findNetworkByEntityID()` that also allow you to use control flags.

In [ ]:

```
# Define search variables.

entity_list = {
    "ENTITIES": [{
        "ENTITY_ID": entity_id_1
    }, {
        "ENTITY_ID": entity_id_2
    }, {
        "ENTITY_ID": entity_id_3
    }]}
entity_list_as_json = json.dumps(entity_list)
max_degree = 2
buildout_degree = 1
max_entities = 12

# Find the network by entity ID.

response_bytearray = bytearray()

return_code = g2_engine.findNetworkByEntityIDV2(
    entity_list_as_json,
    max_degree,
    buildout_degree,
    max_entities,
    g2_engine_flags,
    response_bytearray)

# Print the results.

print("Return Code: {0}".format(return_code))
print("Return Code: {0}".format(return_code))
RenderJSON(response_bytearray)
```

findNetworkByRecordID

In [ ]:

```
# Define search variables.

record_list = {
    "RECORDS": [{
        "RECORD_ID": record_id_1,
        "DATA_SOURCE": datasource_code_1
    }, {
        "RECORD_ID": record_id_2,
        "DATA_SOURCE": datasource_code_2
    }, {
        "RECORD_ID": record_id_3,
        "DATA_SOURCE": datasource_code_3
    }]}
record_list_as_json = json.dumps(record_list)


# Find the network by record ID.

response_bytearray = bytearray()

return_code = g2_engine.findNetworkByRecordID(
    record_list_as_json,
    max_degree,
    buildout_degree,
    max_entities,
    response_bytearray)

# Print the results.

print("Return Code: {0}".format(return_code))
RenderJSON(response_bytearray)
```

findNetworkByRecordIDV2

The function `findNetworkByRecordIDV2()` is an improved version of `findNetworkByRecordID()` that also allow you to use control flags.

In [ ]:

```
# Define search variables.

record_list = {
    "RECORDS": [{
        "RECORD_ID": record_id_1,
        "DATA_SOURCE": datasource_code_1
    }, {
        "RECORD_ID": record_id_2,
        "DATA_SOURCE": datasource_code_2
    }, {
        "RECORD_ID": record_id_3,
        "DATA_SOURCE": datasource_code_3
    }]}
record_list_as_json = json.dumps(record_list)

# Find the network by record ID.

response_bytearray = bytearray()

return_code = g2_engine.findNetworkByRecordIDV2(
    record_list_as_json,
    max_degree,
    buildout_degree,
    max_entities,
    g2_engine_flags,
    response_bytearray)
```

```
# Print the results.

print("Return Code: {0}".format(return_code))
RenderJSON(response_bytearray)
```

Connection Details

The `whyEntityByEntityID()` and `whyEntityByRecordID()` functions can be used to determine why records belong to their resolved entities. These functions will compare the record data within an entity against the rest of the entity data, and show why they are connected. This is calculated based on the features that record data represents.

Records can be chosen by either Record ID or by Entity ID, depending on which function is chosen. If a single record ID is used, then comparison results for that single record will be generated, as part of its entity. If an Entity ID is used, then comparison results will be generated for every record within that entity.

These functions have the following parameters:

- **entity_id:** The entity ID for the entity to be analyzed
- **datasource_code:** The data source for the record to be analyzed
- **record_id:** The record ID for the record to be analyzed
- **g2_engine_flags:** Control flags for outputting entities

They also have various arguments used to return response documents.

The functions return a JSON document that gives the results of the record analysis. The document contains a section called "WHY_RESULTS", which shows how specific records relate to the rest of the entity. It has a "WHY_KEY", which is similar to a match key, in defining the relevant connected data. It shows candidate keys for features that initially cause the records to be analyzed for a relationship, plus a series of feature scores that show how similar the feature data was.

The response document also contains a separate ENTITIES section, with the full information about the resolved entity. (Note: When working with this entity data, Senzing recommends using the flags `G2_ENTITY_SHOW_FEATURES_EXPRESSED` and `G2_ENTITY_SHOW_FEATURES_STATS`. This will provide detailed feature data that is not included by default, but is useful for understanding the WHY_RESULTS data.)

The functions `whyEntityByEntityIDV2()` and `whyEntityByRecordV2()` are enhanced versions of `whyEntityByEntityID()` and `whyEntityByRecordID()` that also allow you to use control flags. The `whyEntityByEntityID()` and `whyEntityByRecordID()` functions work in the same way, but use the default flag value `G2_WHY_ENTITY_DEFAULT_FLAGS`.

For more information, see http://docs.senzing.com/?python

whyEntityByRecordID

In [ ]:

```
response_bytearray = bytearray()

return_code = g2_engine.whyEntityByRecordID(
    datasource_code_1,
    record_id_1,
    response_bytearray)

print("Return Code: {0}".format(return_code))
RenderJSON(response_bytearray)
```

whyEntityByRecordIDV2

In [ ]:

```
response_bytearray = bytearray()

return_code = g2_engine.whyEntityByRecordIDV2(
    datasource_code_1,
    record_id_1,
    g2_engine_flags,
    response_bytearray)

print("Return Code: {0}".format(return_code))
```

```
RenderJSON(response_bytearray)
```

whyEntityByEntityID

In [ ]:

```
response_bytearray = bytearray()

return_code = g2_engine.whyEntityByEntityID(
    entity_id_1,
    response_bytearray)

print("Return Code: {0}".format(return_code))
RenderJSON(response_bytearray)
```

whyEntityByEntityIDV2

In [ ]:

```
response_bytearray = bytearray()

return_code = g2_engine.whyEntityByEntityIDV2(
    entity_id_1,
    g2_engine_flags,
    response_bytearray)

print("Return Code: {0}".format(return_code))
RenderJSON(response_bytearray)
```

Replace

replaceRecord

Use the `replaceRecord()` function to update or replace a record in the data repository. If record doesn't exist, a new record is added to the data repository. Like the above functions, `replaceRecord()` returns "0" upon success, and it can be called as many times as desired and from multiple threads at the same time. The `replaceRecord()` function accepts four parameters as input:

- **datasource_code:** The name of the data source the record is associated with. This value is configurable to the system
- **record_id:** The record ID, used to identify distinct records
- **data_as_json:** A JSON document with the attribute data for the record
- **load_id:** The observation load ID for the record; value can be null and will default to datasource_code

In [ ]:

```
data = {
    "NAMES": [{
        "NAME_TYPE": "PRIMARY",
        "NAME_LAST": "Miller",
        "NAME_FIRST": "John",
        "NAME_MIDDLE": "M"
    }],
    "PASSPORT_NUMBER": "PP11111",
    "PASSPORT_COUNTRY": "US",
    "DRIVERS_LICENSE_NUMBER": "DL11111",
    "SSN_NUMBER": "111-11-1111"
}
data_as_json = json.dumps(data)

return_code = g2_engine.replaceRecord(
    datasource_code_1,
    record_id_1,
    data_as_json,
    load_id)

print("Return Code: {0}".format(return_code))
```

replaceRecordWithInfo

`replaceRecordWithInfo()` is available if you would like to know what resolved entities were modified when replacing a record. It behaves identically to `replaceRecord()`, but also returns a json document containing the IDs of the affected entities. It accepts the following parameters:

- **datasource_code:** The name of the data source the record is associated with. This value is configurable to the system.
- **record_id:** The record ID, used to identify distinct records
- **data_as_json:** A JSON document with the attribute data for the record
- **response_bytearray:** A memory buffer for returning the response document; if an error occurred, an error response is stored here.
- **load_id:** The observation load ID for the record; value can be null and will default to datasource_code

In [ ]:

```
data = {
    "NAMES": [{
        "NAME_TYPE": "PRIMARY",
        "NAME_LAST": "Jones",
        "NAME_FIRST": "John",
        "NAME_MIDDLE": "M"
    }],
    "PASSPORT_NUMBER": "PP11111",
    "PASSPORT_COUNTRY": "US",
    "DRIVERS_LICENSE_NUMBER": "DL11111",
    "SSN_NUMBER": "111-11-1111"
}
data_as_json = json.dumps(data)
response_bytearray = bytearray()

return_code = g2_engine.replaceRecordWithInfo(
    datasource_code_1,
    record_id_1,
    data_as_json,
    response_bytearray,
    load_id)

print("Return Code: {0}".format(return_code))
RenderJSON(response_bytearray)
```

Re-evaluate

reevaluateRecord

In [ ]:

```
return_code = g2_engine.reevaluateRecord(
    datasource_code_1,
    record_id_1,
    g2_engine_flags)

print("Return Code: {0}".format(return_code))
```

reevaluateRecordWithInfo

In [ ]:

```
response_bytearray = bytearray()

return_code = g2_engine.reevaluateRecordWithInfo(
    datasource_code_1,
    record_id_1,
    response_bytearray,
    g2_engine_flags)

print("Return Code: {0}".format(return_code))
RenderJSON(response_bytearray)
```

reevaluateEntity

Find an entity.

In [ ]:

```
response_bytearray = bytearray()

return_code = g2_engine.getEntityByRecordIDV2(
    datasource_code_1,
    record_id_1,
    g2_engine_flags,
    response_bytearray)

response_dictionary = json.loads(response_bytearray)
entity_id_1 = response_dictionary["RESOLVED_ENTITY"]["ENTITY_ID"]

print("Return Code: {0}".format(return_code))
RenderJSON(response_bytearray)
```

Re-evaluate the entity.

In [ ]:

```
return_code = g2_engine.reevaluateEntity(entity_id_1, g2_engine_flags)

print("Return Code: {0}".format(return_code))
```

reevaluateEntityWithInfo

In [ ]:

```
response_bytearray = bytearray()

return_code = g2_engine.reevaluateEntityWithInfo(
    entity_id_1,
    response_bytearray,
    g2_engine_flags)

print("Return Code: {0}".format(return_code))
RenderJSON(response_bytearray)
```

Reporting

Exporting entity data from resolved entities is one of the core purposes of Senzing software. In just a few short steps, the Senzing engine allows users to export entity data in either JSON or CSV format.

exportJSONEntityReport

There are three steps to exporting resolved entity data from the G2Engine object in JSON format. First, use the `exportJSONEntityReport()` method to generate a long integer, referred to here as an `export_handle`. The `exportJSONEntityReport()` method accepts one parameter as input:

- **g2_engine_flags**: An integer specifying which entity details should be included in the export. See the "Entity Export Flags" section for further details.

In [ ]:

```
export_handle = g2_engine.exportJSONEntityReport(g2_engine_flags)
```

fetchNext

Second, use the `fetchNext()` method to read the exportHandle and export a row of JSON output containing the entity data for a single entity. Note that successive calls of `fetchNext()` will export successive rows of entity data. The `fetchNext()` method accepts the following parameters as input:

- **export_handle:** A long integer from which resolved entity data may be read and exported.
- **response_bytearray:** A memory buffer for returning the response document; if an error occurred, an error response is stored here.

For more information, see http://docs.senzing.com/?python.

In [ ]:

```
while True:
    response_bytearray = bytearray()
    g2_engine.fetchNext(export_handle, response_bytearray)
    if not response_bytearray:
        break
    response_dictionary = json.loads(response_bytearray)
    response = json.dumps(response_dictionary, sort_keys=True, indent=4)
    print(response)
```

closeExport

In [ ]:

```
g2_engine.closeExport(export_handle)
```

exportCSVEntityReport

There are three steps to exporting resolved entity data from the G2Engine object in CSV format. First, use the `exportCSVEntityReportV2()` method to generate a long integer, referred to here as an 'export_handle'.

The `exportCSVEntityReportV2()` method accepts these parameter as input:

- **csv_column_list:** A comma-separated list of column names for the CSV export. (These are listed a little further down.)
- **g2_engine_flags:** An integer specifying which entity details should be included in the export. See the "Entity Export Flags" section for further details.

Second, use the `fetchNext()` method to read the exportHandle and export a row of CSV output containing the entity data for a single entity. Note that the first call of `fetchNext()` will yield a header row, and that successive calls of `fetchNext()` will export successive rows of entity data. The `fetchNext()` method accepts the following parameters as input:

- **export_handle:** A long integer from which resolved entity data may be read and exported
- **response_bytearray:** A memory buffer for returning the response document; if an error occurred, an error response is stored here

For more information, see http://docs.senzing.com/?python.

In [ ]:

```
export_handle = g2_engine.exportCSVEntityReport(g2_engine_flags)

while True:
    response_bytearray = bytearray()
    g2_engine.fetchNext(export_handle, response_bytearray)
    if not response_bytearray:
        break
    print(response_bytearray.decode())

g2_engine.closeExport(export_handle)
```

Redo Processing

Redo records are automatically created by Senzing when certain conditions occur where it believes more processing may be needed. Some examples:

- A value becomes generic and previous decisions may need to be revisited

- Clean up after some record deletes
- Detected related entities were being changed at the same time
- A table inconsistency exists, potentially after a non-graceful shutdown

First we will need to have a total of 6 data sources so let's add 4 more.

Create Record and Entity #6

In [ ]:

```
data = {
    "NAMES": [{
        "NAME_TYPE": "PRIMARY",
        "NAME_LAST": "Owens",
        "NAME_FIRST": "Lily"
    }],
    "SSN_NUMBER": "111-11-1111"
}
data_as_json = json.dumps(data)

return_code = g2_engine.replaceRecord(
    datasource_code_4,
    record_id_4,
    data_as_json,
    None)

print("Return Code: {0}".format(return_code))
```

In [ ]:

```
response_bytearray = bytearray()

return_code = g2_engine.getEntityByRecordID(
    datasource_code_4,
    record_id_4,
    response_bytearray)

response_dictionary = json.loads(response_bytearray)
entity_id_6 = response_dictionary["RESOLVED_ENTITY"]["ENTITY_ID"]

print("Return Code: {0}\nEntity ID: {1}".format(return_code, entity_id_6))
RenderJSON(response_bytearray)
```

Create Record and Entity #7

In [ ]:

```
data = {
    "NAMES": [{
        "NAME_TYPE": "PRIMARY",
        "NAME_LAST": "Bauler",
        "NAME_FIRST": "August",
        "NAME_MIDDLE": "E"
    }],
    "SSN_NUMBER": "111-11-1111"
}
data_as_json = json.dumps(data)

return_code = g2_engine.replaceRecord(
    datasource_code_5,
    record_id_5,
    data_as_json,
    None)

print("Return Code: {0}".format(return_code))
```

In [ ]:

```
response_bytearray = bytearray()

return_code = g2_engine.getEntityByRecordID(
    datasource_code_5,
    record_id_5,
    response_bytearray)

response_dictionary = json.loads(response_bytearray)
entity_id_7 = response_dictionary["RESOLVED_ENTITY"]["ENTITY_ID"]

print("Return Code: {0}\nEntity ID: {1}".format(return_code, entity_id_7))
RenderJSON(response_bytearray)
```

Create Record and Entity #8

In [ ]:

```
data = {
    "NAMES": [{
        "NAME_TYPE": "PRIMARY",
        "NAME_LAST": "Barcy",
        "NAME_FIRST": "Brian",
        "NAME_MIDDLE": "H"
    }],
    "SSN_NUMBER": "111-11-1111"
}
data_as_json = json.dumps(data)

return_code = g2_engine.replaceRecord(
    datasource_code_6,
    record_id_6,
    data_as_json,
    None)

print("Return Code: {0}".format(return_code))
```

In [ ]:

```
response_bytearray = bytearray()

return_code = g2_engine.getEntityByRecordID(
    datasource_code_6,
    record_id_6,
    response_bytearray)

response_dictionary = json.loads(response_bytearray)
entity_id_8 = response_dictionary["RESOLVED_ENTITY"]["ENTITY_ID"]

print("Return Code: {0}\nEntity ID: {1}".format(return_code, entity_id_8))
RenderJSON(response_bytearray)
```

Create Record and Entity #9

In [ ]:

```
data = {
    "NAMES": [{
        "NAME_TYPE": "PRIMARY",
        "NAME_LAST": "Miller",
        "NAME_FIRST": "Jack",
        "NAME_MIDDLE": "H"
    }],
    "SSN_NUMBER": "111-11-1111"
}
data_as_json = json.dumps(data)

return_code = g2_engine.replaceRecord(
    datasource_code_7,
    record_id_7,
    data_as_json,
    None)

print("Return Code: {0}".format(return_code))
```

In [ ]:

```
response_bytearray = bytearray()

return_code = g2_engine.getEntityByRecordID(
    datasource_code_7,
    record_id_7,
    response_bytearray)

response_dictionary = json.loads(response_bytearray)
entity_id_9 = response_dictionary["RESOLVED_ENTITY"]["ENTITY_ID"]

print("Return Code: {0}\nEntity ID: {1}".format(return_code, entity_id_9))
RenderJSON(response_bytearray)
```

countRedoRecords

Once the Senzing engine is initialized, use `countRedoRecords()` to return the remaining internally queued maintenance records in the Senzing repository. `countRedoRecords()` takes no arguments and returns <0 for errors.

In [ ]:

```
return_code = g2_engine.countRedoRecords()

print("Return Code: {0}".format(return_code))
```

getRedoRecord

Once the Senzing engine is initialized, use `getRedoRecord()` to retrieve the next internally queued maintenance record into the Senzing repository -- `getRedoRecord()` can be called as many times as desired and from multiple threads at the same time but all threads are required to be in the same process. `getRedoRecord()` should not be called from multiple processes. Unlike `processRedoRecord()`, `getRedoRecord()` does not actually process the record. To process the record, you would use the G2Engine `process()` function. The `getRedoRecord()` function returns "0" upon success and an empty response if there is nothing to do.

- **response_bytearray:** A memory buffer for returning the maintenance document (may be XML or JSON). The format is internal to Senzing. If empty it means there are no maintenance records to return.

In [ ]:

```
response_bytearray = bytearray()

return_code = g2_engine.getRedoRecord(response_bytearray)

print("Return Code: {0}".format(return_code))
```

processWithInfo

In [ ]:

```
if (return_code == 0 and response_bytearray):
    process_response_bytearray = bytearray()

    process_return_code = g2_engine.processWithInfo(
        response_bytearray.decode(),
        process_response_bytearray)

    print("Return Code: {0}".format(process_return_code))
    RenderJSON(process_response_bytearray)
```

process

In [ ]:

```
if (return_code == 0 and response_bytearray):
    g2_engine.process(response_bytearray.decode())
```

processRedoRecord

This processes the next redo record and returns it (If `processRedoRecord()` "response" returns 0 and "response_bytearray" is blank then there are no more redo records to process and if you do `count.RedoRecords()` again it will return 0) Has potential to create more redo records in certian situations.

- **response_bytearray:** A buffer that returns a JSON object that summaries the changes cased by adding the record. Also contains the recordID.

In [ ]:

```
response_bytearray = bytearray()
return_code = g2_engine.processRedoRecord(response_bytearray)
print("Return Code: {0}".format(return_code))

# Pretty-print XML.

xml_string = response_bytearray.decode()
if len(xml_string) > 0:
    import xml.dom.minidom
    xml = xml.dom.minidom.parseString(xml_string)
    xml_pretty_string = xml.toprettyxml()
    print(xml_pretty_string)
```

processRedoRecordWithInfo

`processRedoRecordWithInfo()` is available if you would like to know what resolved entities were modified when processing a redo record. It behaves identically to `processRedoRecord()`, but also returns a json document containing the IDs of the affected entities. It accepts the following parameters:

- **response_bytearray:** A buffer that returns a JSON object that summaries the changes cased by adding the record. Also contains the recordID.
- **response_bytearray:** A buffer that returns a JSON object that summaries the changes cased by adding the record. Also contains the recordID.

In [ ]:

```
response_bytearray = bytearray()
info_bytearray = bytearray()

return_code = g2_engine.processRedoRecordWithInfo(
    response_bytearray,
    info_bytearray)

print("Return Code: {0}".format(return_code))

# Pretty-print XML.

xml_string = response_bytearray.decode()
if len(xml_string) > 0:
    import xml.dom.minidom
    xml = xml.dom.minidom.parseString(xml_string)
    xml_pretty_string = xml.toprettyxml()
    print(xml_pretty_string)

# Pretty-print JSON

RenderJSON(info_bytearray)
```

Delete

deleteRecord

Use `deleteRecord()` to remove a record from the data repository (returns "0" upon success); `deleteRecord()` can be called as many times as desired and from multiple threads at the same time. The `deleteRecord()` function accepts three parameters as input:

- **datasource_code:** The name of the data source the record is associated with. This value is configurable to the system.

- **datasource_code:** The name of the data source the record is associated with. This value is configurable to the system.
- **record_id:** The record ID, used to identify distinct records
- **load_id:** The observation load ID for the record; value can be null and will default to dataSourceCode

In [ ]:

```
return_code = g2_engine.deleteRecord(datasource_code_1, record_id_1, load_id)

print("Return Code: {0}".format(return_code))
```

deleteRecordWithInfo

`deleteRecordWithInfo()` behaves the same as `deleteRecord()` but also returns a json document containing the IDs of the affected entities. It accepts the following parameters:

- **datasource_code:** The name of the data source the record is associated with. This value is configurable to the system.
- **record_id:** The record ID, used to identify distinct records.
- **response_bytearray:** A buffer that returns a JSON object that summaries the changes cased by adding the record. Also contains the recordID.
- **load_id:** The observation load ID for the record; value can be null and will default to dataSourceCode

In [ ]:

```
response_bytearray = bytearray()

return_code = g2_engine.deleteRecordWithInfo(
    datasource_code_2,
    record_id_2,
    response_bytearray,
    load_id,
    g2_engine_flags)

print("Return Code: {0}".format(return_code))
RenderJSON(response_bytearray)
```

Attempt to get the record again. It should error and give an output similar to "Unknown record".

In [ ]:

```
try:
    response_bytearray = bytearray()

    return_code = g2_engine.getRecord(
        datasource_code_1,
        record_id_1,
        response_bytearray)

    response_dictionary = json.loads(response_bytearray)
    response = json.dumps(response_dictionary, sort_keys=True, indent=4)
    print("Return Code: {0}\n{1}".format(return_code, response))
except G2Exception.G2ModuleGenericException as err:
    print("Exception: {0}".format(err))
```

Cleanup

To purge the G2 repository, use the aptly named `purgeRepository()` method. This will remove every record in your current repository.

purgeRepository

In [ ]:

```
g2_engine.purgeRepository()
```

destroy

Once all searching is done in a given process, call `destroy()` to uninitialize Senzing and clean up resources. You should always do this once at the end of each process. See http://docs.senzing.com/?python.

In [ ]:

```
return_code = g2_engine.destroy()

print("Return Code: {0}".format(return_code))
```