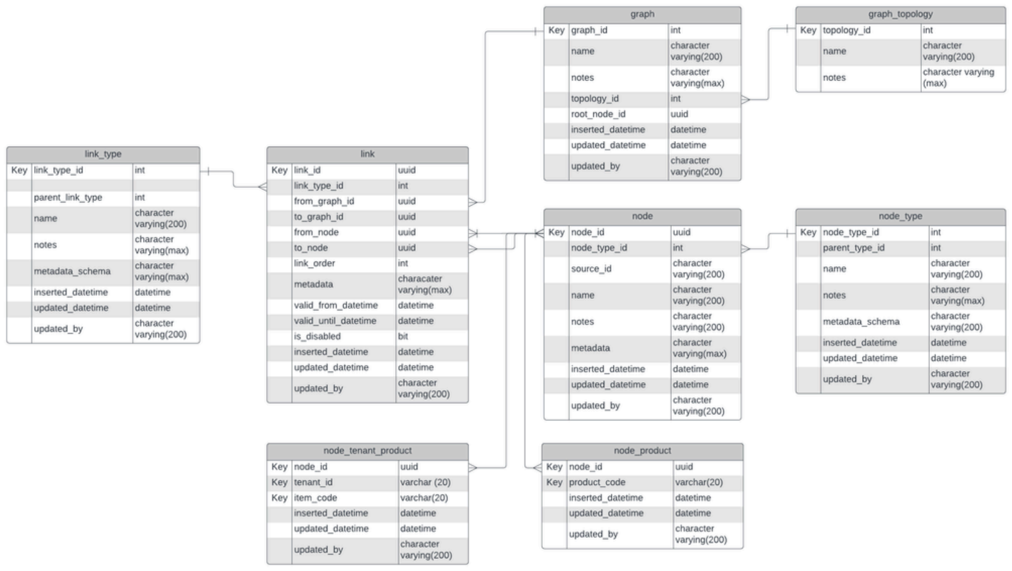
# Taxonomy  API

## Schema


graph
Key graph_id int
name character varying(200)
notes character varying(max)
topology_id int
root_node_id uuid
inserted_datetime datetime
updated_datetime datetime
updated_by character varying(200)
graph_topology
Key topology_id int
name character varying(200)
notes character varying (max)
link_type
Key link_type_id int
parent_link_type int
name character varying(200)
notes character varying(max)
metadata_schema character varying(max)
inserted_datetime datetime
updated_datetime datetime
updated_by character varying(200)
link
Key link_id uuid
link_type_id int
from_graph_id uuid
to_graph_id uuid
from_node uuid
to_node uuid
link_order int
metadata characater varying(max)
valid_from_datetime datetime
valid_until_datetime datetime
is_disabled bit
inserted_datetime datetime
updated_datetime datetime
updated_by character varying(200)
node
Key node_id uuid
node_type_id int
source_id character varying(200)
name character varying(200)
notes character varying(200)
metadata character varying(max)
inserted_datetime datetime
updated_datetime datetime
updated_by character varying(200)
node_type
Key node_type_id int
parent_type_id int
name character varying(200)
notes character varying(max)
metadata_schema character varying(200)
inserted_datetime datetime
updated_datetime datetime
updated_by character varying(200)
node_tenant_product
Key node_id uuid
Key tenant_id varchar (20)
Key item_code varchar(20)
inserted_datetime datetime
updated_datetime datetime
updated_by character varying(200)
node_product
Key node_id uuid
Key product_code varchar(20)
inserted_datetime datetime
updated_datetime datetime
updated_by character varying(200)

## API

Swiftly’s Taxonomy system is implemented as a multi-tenant, multi-purpose database with an API that provides definition, modification, query and management functions.  Some additional concepts and API functions are included which link Taxonomy concepts to other important Swiftly Domains, such as Products.

### graph-create

Create a new graph (header) – also creates root node

#### Request

POST api/taxonomy/graph/create

#### Fields

| Field | Type | Required | Description |
|---|---|---|---|
| name | String | Yes | name of the graph |
| notes | String | No | description of the graph |
| topology_id | Integer | No | defaults to 1: definition of "shape" of graph. |
| updated_by | String | Yes | identification of creating or updating user. |

Example

```
{
    "name": "animals",
    "notes": "a hierarchy of animals",
    "topology_id": 2
    "updated_by": "breanna@swiftly.com"
}
```

Response

```
{
    "graph_id": 29,
    "topology_id": 2,
    "name": "animals",
    "notes": "a hierarchy of animals",
    "root_node_id": "2D8D9BE9-0098-41B7-8CE0-4BA74C3AC133",
    "inserted_datetime": "2024-01-09T14:39:53.757000",
    "updated_datetime": "2024-01-09T14:39:53.757000",
    "updated_by": "breanna@swiftly.com"
}
```

## graph-list

List all graphs

### Request

GET api/taxonomy/graph/list

Params: None

### Response

```
[
    {
        "graph_id": 13,
        "topology_id": 1,
        "name": "dierberg navigation",
        "notes": "Dierberg",
        "root_node_id": "C69B2D7C-534C-45AE-90CC-B379E33C1D27",
        "inserted_datetime": "2023-12-19T20:16:15.777000",
        "updated_datetime": "2023-12-19T20:16:15.777000",
        "updated_by": "breanna@swiftly.com"
    },
    {
        "graph_id": 22,
        "topology_id": 1,
        "name": "swiftly standard navigation",
        "notes": "A standard navigation taxonomy for swiflty clients",
        "root_node_id": "6A337D57-929C-4C21-B898-4EEF059F7D7D",
        "inserted_datetime": "2023-12-19T20:29:35.617000",
        "updated_datetime": "2023-12-19T20:29:35.617000",
        "updated_by": "breanna@swiftly.com"
    }
]
```

## graph-get

Get graph header info by id

### Request

GET /api/taxonomy/graph/get?graph_id={graph_id}

Example: /api/taxonomy/graph/get?graph_id=28

### Parameters

| Parameter | Type | Description |
|---|---|---|
| graph_id | Integer | id of graph |

Response

```
{
    "graph_id": 28,
    "topology_id": 1,
    "name": "swiftly_classification",
    "notes": "Swiftly Product Classification",
    "root_node_id": "50C29661-9B26-4221-BC7D-867706FEAE25",
    "inserted_datetime": "2024-01-08T17:27:23.107000",
    "updated_datetime": "2024-01-08T17:27:23.107000",
    "updated_by": "breanna@swiftly.com"
}
```

## graph-update

Change the name or notes of a graph record

### Request

POST /api/taxonomy/graph/update

Body

```
{
    "graph_id": 26,
    "name": "colors",
    "notes": "example graph",
    "updated_by": "breanna@swiftly.com"
}
```

| Field | Type | Required | Description |
|---|---|---|---|
| graph_id | Integer | Yes | id of the graph |
| name | String | Yes | name of the graph |
| notes | String | No | description of the graph |

### Response

```
{
    "graph_id": 26,
    "topology_id": 1,
    "name": "colors",
    "notes": "example graph",
    "root_node_id": "8FB49972-115B-42C6-8C7C-3C770012CBEF",
    "inserted_datetime": "2024-01-03T14:34:46.620000",
    "updated_datetime": "2024-01-09T15:34:05.760000",
    "updated_by": "breanna@swiftly.com"
}
```

## graph-delete

Delete a graph, all nodes and links (from and to)

This will totally, permanently delete a graph and all it's constituent parts!

Unless! you pass “truncate” which will leave the graph and the root-id in place. This is primarily useful to allow re-import of a graph with the same graph_id.

### Request

DELETE /api/taxonomy/graph/delete

Body:

```
{
    "graph_id": 23
}
```

```
{
    "graph_id": 23,
    "truncate": true
}
```

### Response

Code 200, 404, 500 no json.

## graph-import

Import a whole graph (header, links and nodes In a single pass).  Also supports multi-pass imports where the graph is too large to do in a single call.

While it is possible, and in some cases practical, to import a graph, nodes and links in separate steps with other methods documented here, this call wraps up all three steps into one call which is most convenient for importation of modest-size graphs of up to than 1000 nodes.

### Request

POST /api/taxonomy/graph/import

### JSON fields

| Field | Type | Required | Description |
|---|---|---|---|
| updated_by | String | No | will be added to all objects imported |

| | | | |
|---|---|---|---|
| default_link_type_id | Integer | No | Defaults to 1 (Hierarchical) when applied to root. Note: link_type_id can be specified for each link |
| overwrite | bool | No | If true, you must provide graph_id. This will truncate the graph down to the graph object and root node and re-import the graph with the same graph_id |
| graph | Graph | Yes | defines at least the name of the graph.<br><br>If graph_id is specified, this function will allow adding additional nodes and links to an existing graph.<br><br>If graph_id is not specified, a new graph will be created and all nodes and links will belong to the new graph. |
| nodes | List: Node | Yes | List of nodes to be added |
| links | List: Link | Yes | Links that connect all nodes to the graph (using source_id as the linkage) |

**Example:**

```
{
    "updated_by": "breanna@swiftly.com",
    "default_link_type_id": 1,
    "graph": {
        "topology_id": 1,
        "name": "colors",
        "notes": "test import"
    },
    "nodes": [
          { "name": "red",
          "source_id": "RED",
          "notes": "a red color"
        },
        { "name": "green",
          "source_id": "GREEN",
          "notes": "a green color"
        },
        { "name": "blue",
          "source_id": "BLUE",
          "notes": "a color on the short end of the visible spectrum"
        },
        { "name": "yellow",
          "source_id": "YELLOW",
          "notes": "equal mix of red and green"
        },
        { "name": "green-blue",
          "source_id": "GREEN_BLUE",
          "notes": "a mix of green and blue"
        },
        { "name": "light-green",
          "source_id": "LIGHT_GREEN",
          "notes": "green with white mixed in"
        },
        { "name": "purple",
          "source_id": "PURPLE",
          "notes": "a mix of red and blue"
        }
    ],
    "links": [
        { "from_source_id": "$ROOT$", "to_source_id": "RED"},
        { "from_source_id": "$ROOT$", "to_source_id": "GREEN"},
        { "from_source_id": "$ROOT$", "to_source_id": "BLUE"},
        { "from_source_id": "RED", "to_source_id": "YELLOW"},
        { "from_source_id": "RED", "to_source_id": "PURPLE"},
        { "from_source_id": "GREEN", "to_source_id": "YELLOW"},
        { "from_source_id": "GREEN", "to_source_id": "GREEN_BLUE"},
        { "from_source_id": "GREEN", "to_source_id": "LIGHT_GREEN"},
        { "from_source_id": "BLUE", "to_source_id": "GREEN_BLUE"},
        { "from_source_id": "BLUE", "to_source_id": "PURPLE"}
    ]
}
```

Response

200, 500

### Graph:

See Graph-create above.

If graph_id is specified explicitly, this can be used to add further nodes and links to an already started graph.  This may be necessary for large graphs (> 1000 nodes/links)

Normally only name and notes and possibly topology_id are specified.

### Nodes:

See Node-create. Requires source-id to be set. Source_id must be unique for the graph.

### Links:

| Field | Type | Required | Description |
|---|---|---|---|
| from_source_id | String | Yes | source_id of originating node |
| to_source_id | String | Yes | source_id of destination node |
| link_type_id | Integer | No | to specify the link_type_id if not the default_link_type_id specified at the root. |
| metadata | String (JSON) | No | If metadata desired for node |
| link_order | integer | No | to specify order relative to originating node. |
| valid_from_datetime | datetime | No | to set starting datetime for link validity |
| valid_until_datetime | datetime | No | to set ending datetime for link validity (exclusive) |
| is _disabled | Integer | No | 1: link is disabled |

Note:  the link fields *from_graph_id* and *to_graph_id* are automatically populated with the graph_id of the newly created graph.

### See Also

graph-clone

graph-export

## graph-clone

Copy a graph, it's nodes and links.

Note:  This will also copy all of the inbound and outbound Related links that connect the original graph to other graphs in the database.

### Request

POST /api/taxonomy/graph/clone

Body

```
    {
        "source_graph_id": 26,
        "clone_link_type_id": 6,
        "clone_related_link_type_ids": [ 3 ],
        "graph": {
            "name": "Colors clone 2",
            "notes": "Cloned colors for batch update tests",
            "topology_id": 1,
            "updated_by": "breanna@swiftly.com"
         }
    }
```

| Field | Type | Required | Description |
|---|---|---|---|
| source_graph_id | Integer | Yes | Source graph to clone |
| clone_link_type_id | Integer | No | If present, then links of this type will be created from the original node to the cloned node.  If not present, then no “clone-links” will be created. |
| clone_related_link_type_ids | Array[Integer] | No | An array of related link type ids which should be cloned.  This allow selection of which related type links from the original graph should be cloned.<br><br>Specify [ 3 ] will clone all Related type links.<br><br>The service logic will expand a link type that has child link-types defined. |
| graph | Graph | Yes | This is the contents of the graph record for the cloned graph. See graph-create above. |
| overwrite | boolean | No | If overwrite field exists and is set to true, then the graph_id of the graph object must be filled in with the id of an exsiting graph |

| | | | |
|---|---|---|---|
| | | | to clone into. The graph will be truncated down to the graph object and the root node and then re-cloned over.<br><br>This is used when maintaining the same graph_id is important as for category graphs for tenants. |

## graph-batch

graph-batch allows a full range of modifying operations to be performed on a graph in a single call. The graph must already exist for graph-batch as it does not support creation of the graph in the first place. See graph-import for creating a graph in a single call.

Graph-batch can execute the output from graph-diff

### Request

POST /api/taxonomy/graph/batch

### Body

```
{
  "default_link_id": 1,
  "updated_by": "breanna@swiftly.com",
  "graph": {
    "graph_id": 46
  },
  "nodes": [
    {
      "action": "NEW",
      "node": {
        "source_id": "BEIGE",
        "name": "beige",
        "notes": "color of a suburban living room",
        "metadata": "{\"primary\":0}"
      }
    },
    {
      "action": "UPDATE",
      "node": {
        "node_id": "5305B7AE-B545-4E90-BF7C-4549386F717A",
        "node_type_id": 0,
        "graph_id": 46,
        "source_id": "TRUE_RED",
        "name": "red",
        "notes": "The real red color",
        "metadata": "{\"primary\":1}"
      }
    },
    {
      "action": "DELETE",
      "node": {
        "source_id": "BLACK"
      }
    },
    {
      "action": "MERGE",
      "merge": {
        "from_node": {
          "source_id": "ROUGE"
        },
        "to_node": {
          "source_id": "TRUE_RED"
        }
      }
    },
    {
      "action": "SPLIT",
      "split": {
        "from_node": {
          "source_id": "BLUE"
        },
        "to_node": {
          "source_id": "AZURE",
          "name": "azure",
          "notes": "sky blue"
        }
      }
    }
  ],
  "links": [
    {
      "action": "NEW",
      "link": {
        "from_source_id": "$ROOT$",
        "to_source_id": "BEIGE"
      }
    },
    {
      "action": "UPDATE",
      "link": {
        "link_id": "479F53E2-2BFB-46F6-89E1-60AC955DE684",
        "from_source_id": "OTHER",
        "to_source_id": "GREY",
        "link_order": -1,
        "is_disabled": true
      }
    },
    {
      "action": "DELETE",
      "link": {
        "link_id": "A68A6C08-5CEF-4659-9964-66429D9E14EF"
      }
    }
  ]
}
```

## Actions

### Nodes

#### New

Add a new node to the graph. This does not create a link so a matching New Link must also be included.

See node-create.

See SPLIT below to create a new node and also copy all of that nodes relationships to the new node.

#### Update

Update a node in the graph. See Node-update. This requires all updateable node attributes to be present in the data structure. Node_id must also be specified to update a node.  The source_id may be updated by this operation.

#### Delete

Delete a node.  Only the node_id is required for this operation.  See Merge below for a way to remove a node and move all relationships to a node to another node.

#### Merge

Merge one node into another. This deletes the "from" node and moves all of the links to the "from" node to the "to" node. This is often a preferrable way to remove a node from a graph in maintenance mode if the intent is to express that the two concepts are in fact one.  if external related links are present to the "from" node, they will now be pointed to the "to" node.

```
{
     "action": "MERGE",
     "merge": {
       "from_node": {
         "source_id": "ROUGE"
       },
       "to_node": {
         "source_id": "TRUE_RED"
       }
     }
   }
```

Unlinke other Node batch operations, the body of the action is not a Node object but a special "merge" json with a "from_node" and "to_node" sub-object.  Either a node_id or a source_id is necessary to refer to the node.  If node_id is present it will be used over the source_id.

See Also: Node-merge

#### Split

Used to split a node (concept) into two finer-grained nodes.  Technically , it creates a new node (to_node) and copies the inbound and outbound links to the new node.

Note: It may be desirable to subsequently delete unwanted links to or from the from and/or to node.

```
{
  "action": "SPLIT",
  "split": {
    "from_node": {
      "source_id": "BLUE"
    },
    "to_node": {
      "source_id": "AZURE",
      "name": "azure",
      "notes": "sky blue"
    }
  }
}
```

Like "MERGE" above, split action has a special data structure "split" to define the action.

The “from_node” only requires a node_id OR a source_id.

The “to_node” requires all fields as for node-create . See node-create for more information.

Note:  “updated_by” and “graph_id”  are filled in from the outermost batch object.

### Links

#### New

Add a new link.

```
{
  "action": "NEW",
  "link": {
    "from_source_id": "$ROOT$",
    "to_source_id": "BEIGE"
  }
}
```

This requires only from_node_id and to_node_id or from_source_id and to_source_id.

Note: this cannot be used to add links between two different graphs. See link-add for that.

#### Update

Update a link. This patches individual attributes of a link so only the updated attribute(s) must be specified

```
 {
   "action": "UPDATE",
   "link": {
     "link_id": "479F53E2-2BFB-46F6-89E1-60AC955DE684",
     "from_source_id": "OTHER",
     "to_source_id": "GREY",
     "link_order": -1,
     "is_disabled": true
   }
 }
```

Note: the example above shows both a link_id and from_source_id and to_source_id. In reality only one or the other is necessary. If link_id is present it will be used.

#### Delete

Delete a link.

```
{
  "action": "DELETE",
  "link": {
    "link_id": "A68A6C08-5CEF-4659-9964-66429D9E14EF"
  }
}
```

the link_id is required for this action.

Note: Deleting a link to a node without deleting the node will make the node an “orphan”. It will still belong to the graph but not be accessible by any traversal from the root.  To find orphan nodes see *node-get-orphans*.

## Response

Returns JSON with graph, nodes and links changed.

<Example>

## graph-diff

Graph-diff compares a persisted graph in the database against a json representation of the graph.  It outputs a json specification for every change it found between the passed json and the stored graph and what steps would need to be taken to make the persisted graph the same as the passed json.

Graph-diff is a companion to graph-batch in that the output of graph-diff is the same format is the input to graph-batch.   Graph-batch can process the additional actions of node: “merge” and “split” which cannot be detected automatically by graph-diff however.

Graph-diff can take the output of graph-export as input.

The output of graph-diff can be edited if needed to adjust the changes and then submitted to graph-batch

This is useful for a range of scenarios.

1. Incremental sync of a taxonomy to an external source-of-truth.
2. Batch oriented change management and editing of a graph
3. Snapshot and restore of a graph state. using graph-export.

### Request

POST /api/taxonomy/graph/diff

### Body

```
{
    "default_link_type_id": 1,
    "graph": {
        "graph_id": 46
    },
    "nodes": [

        { "name": "red",
          "source_id": "RED",
          "notes": "a red color",
          "metadata": "{\"primary\": 1}"
        },
        { "name": "green",
          "source_id": "GREEN",
          "notes": "a green color",
          "metadata": "{\"primary\": 1}"
        },
        { "name": "blue",
          "source_id": "BLUE",
          "notes": "a color on the short end of the visible spectrum",
          "metadata": "{\"primary\": 1}"
        },
        { "name": "yellow",
          "source_id": "YELLOW",
          "notes": "Yellow (green + red)",
          "metadata": "{\"primary\": 0}"
        },
        { "name": "cyan",
          "source_id": "CYAN",
          "notes": "green and blue?",
          "metadata": "{\"primary\": 0}"
        },
        { "name": "green-blue",
          "source_id": "GREEN_BLUE",
          "notes": "a mix of green and blue",
          "metadata": "{\"primary\": 0}"
        },
        { "name": "light-green",
          "source_id": "LIGHT_GREEN",
          "notes": "green with white mixed in",
          "metadata": "{\"primary\": 0}"
        },
        { "name": "purple",
          "source_id": "PURPLE",
          "notes": "a mix of red and blue",
          "metadata": "{\"primary\": 0}"
        }
    ],
    "links": [
        { "from_source_id": "$ROOT$", "to_source_id": "RED"},
        { "from_source_id": "$ROOT$", "to_source_id": "GREEN"},
        { "from_source_id": "$ROOT$", "to_source_id": "BLUE"},
        { "from_source_id": "RED", "to_source_id": "YELLOW"},
        { "from_source_id": "RED", "to_source_id": "PURPLE"},
        { "from_source_id": "GREEN", "to_source_id": "YELLOW"},
        { "from_source_id": "GREEN", "to_source_id": "GREEN_BLUE"},
        { "from_source_id": "GREEN", "to_source_id": "LIGHT_GREEN"},
        { "from_source_id": "GREEN", "to_source_id": "CYAN"},
        { "from_source_id": "BLUE", "to_source_id": "GREEN_BLUE"},
        { "from_source_id": "BLUE", "to_source_id": "CYAN"},
        { "from_source_id": "BLUE", "to_source_id": "PURPLE"}
    ]
}
```

### Params

| Field | Type | Required | Description |
|---|---|---|---|
| default_link_type_id | Integer | No | Defaults to 1 (Hierarchical) when applied to root. Is used to specific link type for any new links added. |
| graph | Graph | Yes | The graph against which the json is to be compared. Only graph_id is used. |
| nodes | List: Node | Yes | List of nodes to be compared to the stored graph |
| links | List: Link | Yes | list of links to be compared to the stored graph |

See graph-import for detailed description of fields for nodes and links.

Nodes and links can be minimal structures without ids as is the case for graph-import but can also be fully-hydrated data structures as output by graph-export. It works with both types of scenarios.

### Response

graph-diff outputs a data structure which is the same as the specification for graph-batch except that it lacks the fields : default_link_id and updated_by

The notable addition is that action: “UPDATE” objects include a “fields_changed” object which documents “field_name”, “orig_value” and “new_value” for informational purposes.  If this is submitted to graph-batch, the “fields_changed” object is ignored.

```
{
    "graph": {
        "graph_id": 46,
        "topology_id": 1,
        "name": "test_colors",
        "notes": "test import for batch testing",
        "root_node_id": "77FBCFCB-0E49-4DA8-850E-92A9F9EE013E",
        "inserted_datetime": "2024-02-21T14:40:58.510000",
        "updated_datetime": "2024-02-21T14:40:58.510000",
        "updated_by": "breanna@swiftly.com"
    },
    "nodes": [
        {
            "action": "UPDATE",
            "node": {
                "name": "red",
                "source_id": "RED",
                "notes": "a red color",
                "metadata": "{\"primary\": 1}",
                "node_type_id": 0,
                "node_id": "5305B7AE-B545-4E90-BF7C-4549386F717A",
                "graph_id": 46
            },
            "fields_changed": [
                {
                    "field_name": "notes",
                    "orig_value": "The real red color",
                    "new_value": "a red color"
                },
                {
                    "field_name": "metadata",
                    "orig_value": "{\"primary\":1}",
                    "new_value": "{\"primary\": 1}"
                }
            ]
        },
        {
            "action": "NEW",
            "node": {
                "name": "cyan",
                "source_id": "CYAN",
                "notes": "green and blue?",
                "metadata": "{\"primary\": 0}"
            }
        },
        {
            "action": "DELETE",
            "node": {
                "node_id": "580CFA7A-7E7F-4C36-8A1E-0594CFE29B6A",
                "node_type_id": 0,
                "graph_id": 46,
                "source_id": "MAGENTA",
                "name": "magenta",
                "notes": "vibrant pink color",
                "metadata": null,
                "inserted_datetime": "2024-02-21T14:51:01.783000",
                "updated_datetime": "2024-02-21T14:51:01.783000",
                "updated_by": "breanna@swiftly.com"
            }
        }
    ],
    "links": [
        {
            "action": "NEW",
            "link": {
                "from_source_id": "GREEN",
                "to_source_id": "CYAN",
                "link_type_id": 1
            }
        },
        {
            "action": "NEW",
            "link": {
                "from_source_id": "BLUE",
                "to_source_id": "CYAN",
                "link_type_id": 1
            }
        },
           {
            "action": "DELETE",
            "link": {
                "link_id": "3002AA87-52C8-406E-802B-61CF315A6FE5",
                "link_type_id": 1,
                "from_graph_id": 46,
                "from_node_id": "BA59648E-3C26-4C37-B0F1-A9E677E33D9B",
                "from_source_id": "BLUE",
                "to_graph_id": 46,
                "to_node_id": "580CFA7A-7E7F-4C36-8A1E-0594CFE29B6A",
                "to_source_id": "MAGENTA",
                "link_order": null,
                "metadata": null,
                "valid_from_datetime": null,
                "valid_until_datetime": null,
                "is_disabled": null
            }
        }
    ]
}
```

### See also

Graph-import

Graph-batch

Graph-export

## graph-export

### Request

GET /api/taxonomy/graph/export?graph_id={graph_id}

### Params

graph_id - the identifier of the graph to be exported

### Response

| Field | Type | Required | Description |
|---|---|---|---|
| graph | Graph | Yes | The graph object |
| nodes | List: Node | Yes | List of fully hydrated node objects |
| links | List: Link | Yes | list of fully hydrated link objects.  Note: only includes links from and to the current graph. |

Example

```
{
    "graph": {
        "graph_id": 26,
        "topology_id": 1,
        "name": "colors",
        "notes": "example graph",
        "root_node_id": "8FB49972-115B-42C6-8C7C-3C770012CBEF",
        "inserted_datetime": "2024-01-03T14:34:46.620000",
        "updated_datetime": "2024-01-09T15:34:05.760000",
        "updated_by": "breanna@swiftly.com"
    },
    "nodes": [
       {
            "node_id": "9F4ECA30-AA7C-42B8-A27C-2BA451068BCE",
            "node_type_id": 0,
            "graph_id": 26,
            "source_id": "YELLOW",
            "name": "yellow",
            "notes": "equal mix of red and green",
            "metadata": "{\"primary\": 0}",
            "inserted_datetime": "2024-01-03T14:35:55.813000",
            "updated_datetime": "2024-01-03T14:35:55.813000",
            "updated_by": "breanna@swiftly.com"
        },
        {
            "node_id": "AF626586-B846-4455-8C19-342A1F22870C",
            "node_type_id": 0,
            "graph_id": 26,
            "source_id": "RED",
            "name": "red",
            "notes": "a red color",
            "metadata": "{\"primary\": 1}",
            "inserted_datetime": "2024-01-03T14:35:55.813000",
            "updated_datetime": "2024-01-03T14:35:55.813000",
            "updated_by": "breanna@swiftly.com"
        },
        {
            "node_id": "4F4BCEA7-F2F6-4FE3-8650-49589C7F0C18",
            "node_type_id": 0,
            "graph_id": 26,
            "source_id": "GREEN",
            "name": "green",
            "notes": "a green color",
            "metadata": "{\"primary\": 1}",
            "inserted_datetime": "2024-01-03T14:35:55.813000",
            "updated_datetime": "2024-01-03T14:35:55.813000",
            "updated_by": "breanna@swiftly.com"
        },
        {
            "node_id": "05BD3561-161E-4E19-A9EB-68515CA268B9",
            "node_type_id": 0,
            "graph_id": 26,
            "source_id": "LIGHT_GREEN",
            "name": "light-green",
            "notes": "green with white mixed in",
            "metadata": "{\"primary\": 0}",
            "inserted_datetime": "2024-01-03T14:35:55.813000",
            "updated_datetime": "2024-01-03T14:35:55.813000",
            "updated_by": "breanna@swiftly.com"
        },
        {
            "node_id": "9E400BC9-9257-4066-BABC-B157B0A8695F",
            "node_type_id": 0,
            "graph_id": 26,
            "source_id": "GREEN_BLUE",
            "name": "green-blue",
            "notes": "a mix of green and blue",
            "metadata": "{\"primary\": 0}",
            "inserted_datetime": "2024-01-03T14:35:55.813000",
            "updated_datetime": "2024-01-03T14:35:55.813000",
            "updated_by": "breanna@swiftly.com"
        },
        {
            "node_id": "B35F2EBE-2C72-4E2D-9FE3-B907CD025572",
            "node_type_id": 0,
            "graph_id": 26,
            "source_id": "BLUE",
            "name": "blue",
            "notes": "a color on the short end of the visible spectrum",
            "metadata": "{\"primary\": 1}",
            "inserted_datetime": "2024-01-03T14:35:55.813000",
            "updated_datetime": "2024-01-03T14:35:55.813000",
            "updated_by": null
        }
    ],
    "links": [
        {
            "link_id": "36529865-459A-4366-96D3-17BDBB09E6AF",
            "link_type_id": 1,
            "from_graph_id": 26,
            "from_node_id": "8FB49972-115B-42C6-8C7C-3C770012CBEF",
            "from_source_id": "$ROOT$",
            "to_graph_id": 26,
            "to_node_id": "4F4BCEA7-F2F6-4FE3-8650-49589C7F0C18",
            "to_source_id": "GREEN",
            "link_order": null,
            "metadata": null,
            "valid_from_datetime": null,
            "valid_until_datetime": null,
            "is_disabled": null
        },
        {
            "link_id": "25DFF3D0-311F-4BEF-8820-A314470AA053",
            "link_type_id": 1,
            "from_graph_id": 26,
            "from_node_id": "8FB49972-115B-42C6-8C7C-3C770012CBEF",
            "from_source_id": "$ROOT$",
            "to_graph_id": 26,
            "to_node_id": "B35F2EBE-2C72-4E2D-9FE3-B907CD025572",
            "to_source_id": "BLUE",
            "link_order": null,
            "metadata": null,
            "valid_from_datetime": null,
            "valid_until_datetime": null,
            "is_disabled": null
        },
        {
            "link_id": "58B87D46-61BE-4717-9A77-DF0FEB8D500F",
            "link_type_id": 1,
            "from_graph_id": 26,
            "from_node_id": "8FB49972-115B-42C6-8C7C-3C770012CBEF",
            "from_source_id": "$ROOT$",
            "to_graph_id": 26,
            "to_node_id": "AF626586-B846-4455-8C19-342A1F22870C",
            "to_source_id": "RED",
            "link_order": null,
            "metadata": null,
            "valid_from_datetime": null,
            "valid_until_datetime": null,
            "is_disabled": false
        },
        {
            "link_id": "E36F6C78-667C-4073-BB08-DF80464A764D",
            "link_type_id": 1,
            "from_graph_id": 26,
            "from_node_id": "AF626586-B846-4455-8C19-342A1F22870C",
            "from_source_id": "RED",
            "to_graph_id": 26,
            "to_node_id": "9F4ECA30-AA7C-42B8-A27C-2BA451068BCE",
            "to_source_id": "YELLOW",
            "link_order": null,
            "metadata": null,
            "valid_from_datetime": null,
            "valid_until_datetime": null,
            "is_disabled": null
        },
        {
            "link_id": "8A119141-A460-4976-9C65-94A0517E55BC",
            "link_type_id": 1,
            "from_graph_id": 26,
            "from_node_id": "B35F2EBE-2C72-4E2D-9FE3-B907CD025572",
            "from_source_id": "BLUE",
            "to_graph_id": 26,
            "to_node_id": "9E400BC9-9257-4066-BABC-B157B0A8695F",
            "to_source_id": "GREEN_BLUE",
            "link_order": null,
            "metadata": null,
            "valid_from_datetime": null,
            "valid_until_datetime": null,
            "is_disabled": null
        },
        {
            "link_id": "AE85F497-35F9-47A6-9B71-0CCA58F99AC3",
            "link_type_id": 1,
            "from_graph_id": 26,
            "from_node_id": "4F4BCEA7-F2F6-4FE3-8650-49589C7F0C18",
            "from_source_id": "GREEN",
            "to_graph_id": 26,
            "to_node_id": "05BD3561-161E-4E19-A9EB-68515CA268B9",
            "to_source_id": "LIGHT_GREEN",
            "link_order": null,
            "metadata": null,
            "valid_from_datetime": null,
            "valid_until_datetime": null,
            "is_disabled": null
        },
        {
            "link_id": "3E3686E0-E9FD-44EF-80BA-7CEB8C292D20",
            "link_type_id": 1,
            "from_graph_id": 26,
            "from_node_id": "4F4BCEA7-F2F6-4FE3-8650-49589C7F0C18",
            "from_source_id": "GREEN",
            "to_graph_id": 26,
            "to_node_id": "9E400BC9-9257-4066-BABC-B157B0A8695F",
            "to_source_id": "GREEN_BLUE",
            "link_order": null,
            "metadata": null,
            "valid_from_datetime": null,
            "valid_until_datetime": null,
            "is_disabled": null
        },
        {
            "link_id": "54C305B2-9DDA-40AF-A5BA-97984E51944F",
            "link_type_id": 1,
            "from_graph_id": 26,
            "from_node_id": "4F4BCEA7-F2F6-4FE3-8650-49589C7F0C18",
            "from_source_id": "GREEN",
            "to_graph_id": 26,
            "to_node_id": "9F4ECA30-AA7C-42B8-A27C-2BA451068BCE",
            "to_source_id": "YELLOW",
            "link_order": null,
            "metadata": null,
            "valid_from_datetime": null,
            "valid_until_datetime": null,
            "is_disabled": null
        }
    ]
}
```

## See also

Graph-import

Graph-diff

## graph-query

graph-query permits retrieval of graph data by way of node queries and traversals of links.  It is fairly advanced and can be used to address both simple an complex scenarios:

- Get node parent
- get node ancestors (all)
- get node children
- get node descendents (children and all their children recursive)
- get related nodes from another graph
- get all nodes from another graph directly connected to a node or all ancestor nodes (inherited linkages)
- and many more...

Please see the Graph Query Developer Guide document for a more in-depth coverage.

### Request

POST /api/taxonomy/graph/query

### Body

| Field | Type | Required | Description |
|---|---|---|---|
| graph_id | int | Yes | the graph to query |
| seed | Seed | No | A Node query  - see node-query.  If not specified, the Root node of the graph is the seed node. The seed query may return multiple nodes which all will be used as starting points for subsequent link traversal steps. |
| steps | Array[Step ] | No | One or more step of link traversals.  This is optional. If not specified, the default step is to traverse FROM the seed nodes(s) hierarchical links with no depth limit. (full graph traversal) |
| return | Return | No | Type of data to return. Default: type: “*”, a denormalized link with most from and to node data included. |

### Step

| Field | Type | Required | Description |
|---|---|---|---|
| dir | String | No | FROM, TO (Default FROM).<br><br>FROM - follow linkd From the seed nodes or subsequent nodes,<br><br>TO - follow links from head to tail to Parent nodes. |
| depth | Integer | No | Default: no depth limit. |

| | | | |
|---|---|---|---|
| | | | The number of recursive steps to return results from.<br><br>Examples:<br><br>FROM,  depth:1 would get only direct children.<br><br>depth: 2 would return children and grand-children. |
| link_types | List: Int | No | list of types of links to follow. |
| graph_ids | List: Int | No | List of graphs that can be traversed to.  Default origin graph_id only  Explicitly passing an empty list [] means “all graphs” |
| link_seed | boolean | No | Default: True for step1, False for subsequent steps.<br><br>This is only relevant for multi-step queries.  The first step always starts from seed nodes. This is helpful in multi-step queries. |
| include_results | boolean | No | Default: True for last step, False for any previous steps.   Include the nodes from this step in the output result.<br><br>Normally, only the results of the final step are included in the results.  For multi-step queries, set to True for all steps for which results are to be returned. |

## Return

| Field | Type | Required | Description |
|---|---|---|---|
| type | String | No | *, NODE, LINK<br><br>“*” - return a denormalized link with from and to node data plus fields “depth”, “direction”<br><br>NODE - return a distinct list of Node objects.<br><br>LINK - return conformant Link objects |
| include_seed | Boolean | No | Default: false<br><br>Applies only to type:NODE returns.  Include the seed nodes in the result. |

## Response

Type “*”

| Field | Type | Description |
|---|---|---|
| depth | Integer | link-traversals away from the seed nodes. Starts at 1. |
| direction | String | “FROM” OR “TO”. |

| | | |
|---|---|---|
| | | FROM: means the link was traversed from node to to_node<br><br>TO:  means that the link was traversed in reverse direction starting at the TO node. |
| step | integer | Which Step (in multi-step query) as this data from (starting with 1) |
| link_order | integer | order of the link relative to parent if set |
| link_type_id | integer | type of link |
| link_metadata | String | JSON metadata for link |
| valid_from_datetime | Datetime | link valid start datetime |
| valid_until_datetime | Datetime | link valid until datetime |
| is_disabled | Boolean | link is disabled |
| from_node_id | UUID | From node identifier |
| from_graph_id | Integer | link from graph identifier |
| from_node_type_id | Integer | from node type |
| from_source_id | String | Source_id of from_node |
| from_name | String | Name of from_node |
| from_notes | String | Notes of from_node |
| from_metadata | String | JSON metadata for from_node |
| from_inserted_datetime | Datetime | inserted_datetime for from_node |
| from_updated_datetime | Datetime | updated_datetime for from_node |
| from_updated_by | String | user who made insert or last update to from_node |
| to_* | | Same fields for to_node... |

Example:

```
{
        "depth": 1,
        "direction": "FROM",
        "link_order": null,
        "link_id": "22449C08-0C50-4D40-B385-305AFC53BC30",
        "link_type_id": 1,
        "link_metadata": null,
        "valid_from_datetime": null,
        "valid_until_datetime": null,
        "is_disabled": null,
        "from_node_id": "B35F2EBE-2C72-4E2D-9FE3-B907CD025572",
        "from_graph_id": 26,
        "from_node_type_id": 0,
        "from_source_id": "BLUE",
        "from_name": "blue",
        "from_notes": "a color on the short end of the visible spectrum",
        "from_metadata": "{\"primary\": 1}",
        "from_inserted_datetime": "2024-01-03T14:35:55.813000",
        "from_updated_datetime": "2024-01-03T14:35:55.813000",
        "from_updated_by": null,
        "to_node_id": "2BEBD78A-E06E-46E5-AD49-06DD8CEA6162",
        "to_graph_id": 26,
        "to_node_type_id": 0,
        "to_source_id": "PURPLE",
        "to_name": "purple",
        "to_notes": "a mix of red and blue",
        "to_metadata": "{\"primary\": 0}",
        "to_inserted_datetime": "2024-01-03T14:35:55.813000",
        "to_updated_datetime": "2024-01-03T14:35:55.813000",
        "to_updated_by": "breanna@swiftly.com",
        "step": 1
    },
```

return type “NODE”

Distinct List of Standard Node object, see node-get

Example:

```
    {
        "node_id": "2BEBD78A-E06E-46E5-AD49-06DD8CEA6162",
        "node_type_id": 0,
        "graph_id": 26,
        "source_id": "PURPLE",
        "name": "purple",
        "notes": "a mix of red and blue",
        "metadata": "{\"primary\": 0}",
        "inserted_datetime": "2024-01-03T14:35:55.813000",
        "updated_datetime": "2024-01-03T14:35:55.813000",
        "updated_by": "breanna@swiftly.com"
    }
```

return type “LINK”

List of Link object with source_id denormalized from from and to nodes:

Example:

```
    {
        "link_id": "22449C08-0C50-4D40-B385-305AFC53BC30",
        "link_type_id": 1,
        "from_graph_id": 26,
        "from_node_id": "B35F2EBE-2C72-4E2D-9FE3-B907CD025572",
        "from_source_id": "BLUE",
        "to_graph_id": 26,
        "to_node_id": "2BEBD78A-E06E-46E5-AD49-06DD8CEA6162",
        "to_source_id": "PURPLE",
        "link_order": null,
        "metadata": null,
        "valid_from_datetime": null,
        "valid_until_datetime": null,
        "is_disabled": null
    }
```

### See also

node-query

## node-create

create one or more nodes.

This is the basic method that can be accomplished with graph-import or graph-batch. The underlying implementation is the same.

### Request

POST /api/taxonomy/node/create

### See also

graph-import

graph-batch

## node-get

Get a single node with either a node_id or a graph_id and source_id.

This is the most rudimentary way to get a node object:

### Request

GET/api/taxonomy/node/get

### Params

| Field | Type | Required | Description |
|---|---|---|---|
| node_id | UUID | No | node identifier |
| graph_id | Integer | No | graph id if using source_id to fetch |
| source_id | String | No | Must also provide graph_id |

### Response

```
{
    "node_id": "4F4BCEA7-F2F6-4FE3-8650-49589C7F0C18",
    "node_type_id": 0,
    "graph_id": 26,
    "source_id": "GREEN",
    "name": "green",
    "notes": "a green color",
    "metadata": "{\"primary\": 1}",
    "inserted_datetime": "2024-01-03T14:35:55.813000",
    "updated_datetime": "2024-01-03T14:35:55.813000",
    "updated_by": "breanna@swiftly.com"
}
```

200, 404, 500

### See also

node-query

## node-delete

Delete one or more nodes by node_id. Can also be accomplished with graph-batch.

### Request

DELETE /api/taxonomy/node/delete

### Body

```
{
    "nodes": [
        {
           "node_id": "4FC0CF01-F54A-40C5-8C74-A13CD0858D31"
        }
    ]
}
```

### Response

200, 500, no JSON returned.

### See also

graph-batch

## node-update

Update a single node.

We strongly suggest using graph-batch as it will allow updates of multiple nodes in single call.

Under the covers, graph-batch with a node Update will call the same code.

note:

node-update at this time requires node_id and all of the fields noted below. It does not PATCH individual fields of the node (like update-link does).

This method does not allow change of node_id or graph_id

### Request

POST /api/taxonomy/node/update

### Body

```
{
    "node_id": "4F4BCEA7-F2F6-4FE3-8650-49589C7F0C18",
    "node_type_id": 0,
    "graph_id": 26,
    "source_id": "GREEN",
    "name": "green",
    "notes": "a green-ish color",
    "metadata": "{\"primary\": 1}",
    "updated_by": "breanna@swiftly.com"
}
```

### Response

```
{
    "node_id": "4F4BCEA7-F2F6-4FE3-8650-49589C7F0C18",
    "node_type_id": 0,
    "graph_id": 26,
    "source_id": "GREEN",
    "name": "green",
    "notes": "a green-ish color",
    "metadata": "{\"primary\": 1}",
    "inserted_datetime": "2024-01-03T14:35:55.813000",
    "updated_datetime": "2024-02-28T15:31:58.543000",
    "updated_by": "breanna@swiftly.com"
}
```

### See also

graph-batch

## node-query

Query nodes within or across graphs.

Good for finding nodes by their properties.

This is also the syntax for the “seed” of a graph-query

### Request

POST DELETE /api/taxonomy/node/query

### Body

Example:

```
{
    "graph_id": { "value": 26 },
    "metadata": { "field": "primary", "value": 1 }
}
```

node query can query any Property of a node.

`['node_id', 'node_type_id', 'graph_id', 'source_id', 'name', 'notes', 'metadata', 'inserted_datetime', 'updated_datetime', 'updated_by']`

Each Property in the query JSON is keyed by a Node property name and has a predicate object

All Predicates are ANDed together, there is no Boolean combiner option at this time.

| Field | Type | Required | Description |
|---|---|---|---|
| op | String | Yes | Operand to use to compare Property to Value, see below. Unary operators do not support Values |
| value | Any | Yes | depends on operator. value should be appropriate to Property being queried. |

|  |  |  |  |
|---|---|---|---|
| field | String | No | Required if Property is “metadata” |

“op”: an operand

which can be one of the operators:

| Operator | Description |
|---|---|
| EQ, = | property is equal to Value |
| NE, != | property is not equal to Value |
| LE, <= | property is <= to Value |
| GE, >= | property is >= the Value |
| LT, < | property is less than the Value |
| GT, > | property is less than the Value |
| LIKE | field is like (Using SQL format) the Value |
| “NOT LIKE” | property is not like Value |

Unary Operators (does not require/support Value)

| Operator | Description |
|---|---|
| ISNULL, “IS NULL” | property is null |
| NOTNULL, “NOT NULL” | property is not null |

## Response

A list of nodes:

Note: The response is not currently paginated.

```
[
    {
        "node_id": "AF626586-B846-4455-8C19-342A1F22870C",
        "node_type_id": 0,
        "graph_id": 26,
        "source_id": "RED",
        "name": "red",
        "notes": "a red color",
        "metadata": "{\"primary\": 1}",
        "inserted_datetime": "2024-01-03T14:35:55.813000",
        "updated_datetime": "2024-01-03T14:35:55.813000",
        "updated_by": null
    },
    {
        "node_id": "4F4BCEA7-F2F6-4FE3-8650-49589C7F0C18",
        "node_type_id": 0,
        "graph_id": 26,
        "source_id": "GREEN",
        "name": "green",
        "notes": "a green-ish color",
        "metadata": "{\"primary\": 1}",
        "inserted_datetime": "2024-01-03T14:35:55.813000",
        "updated_datetime": "2024-02-28T15:31:58.543000",
        "updated_by": "breanna@swiftly.com"
    },
    {
        "node_id": "B35F2EBE-2C72-4E2D-9FE3-B907CD025572",
        "node_type_id": 0,
        "graph_id": 26,
        "source_id": "BLUE",
        "name": "blue",
        "notes": "a color on the short end of the visible spectrum",
        "metadata": "{\"primary\": 1}",
        "inserted_datetime": "2024-01-03T14:35:55.813000",
        "updated_datetime": "2024-01-03T14:35:55.813000",
        "updated_by": null
    }
]
```

### See also

graph-query

## node-get-orphans

If all links are deleted to a Node from the graph to which the node belongs, the node is considered an “orphan”. It can never be found by way of a graph-query.  (It can be found with node-query).

To identify which nodes have been orphaned, use node-get-orphans

### Request

### Body

### Params

| Field | Type | Required | Description |
|---|---|---|---|
| links | List: Link | Yes | list of links to be compared to the stored graph |

### Response

A list of orphan nodes.

## node-merge

Merge one node into another.

This moves all links (from within and across graphs) to the destination node.

### Request

POST /api/taxonomy/node/merge

### Body

```
{
    "graph_id": 45,
    "from_node": { "source_id": "AZURE"},
    "to_node": {"source_id": "BLUE"}
}
```

### See also

graph-batch

## node-split

Split a node into two nodes.

This creates a new node and copies links from the source node to the new node.

See graph-batch which can combine this operation with other appropriate operations such as updating the source node and deletion of unneeded links.

### Request

POST /api/taxonomy/node/split

### Body

```
{
    "graph_id": 45,
    "from_node": { "source_id": "BLUE"},
    "to_node": {
      "source_id": "AZURE",
      "name": "azure",
      "notes": "the color of the sky",
      "updated_by": "breanna@swiftly.com"
    }
}

from_node may be specified with "node_id" or "source_id"

```

### Response

new to-node created.

### See also

graph-batch

## link-move

(function not implemented)

## link-copy

(function not implemented)

## link-type-get

get a link type and all descendant link types.

This operation is used internally by graph-query to determine which link types to query.

### Request

GET /api/taxonomy/link-type/get?link_type_id=3

### Response

```
[
    {
        "link_type_id": 3,
        "parent_link_type": -1,
        "name": "related",
        "notes": "used primarily to link equivalent node across graphs",
        "metadata_schema": null,
        "inserted_datetime": "2024-01-05T22:12:25.440000",
        "updated_datetime": "2024-01-05T22:12:25.440000",
        "updated_by": "breann@swiftly.com"
    },
    {
        "link_type_id": 4,
        "parent_link_type": 3,
        "name": "rel_prod_class_nav",
        "notes": "related type link between product classification and navigation taxonomies",
        "metadata_schema": null,
        "inserted_datetime": "2024-01-05T22:12:25.440000",
        "updated_datetime": "2024-01-05T22:12:25.440000",
        "updated_by": "breann@swiftly.com"
    },
    {
        "link_type_id": 5,
        "parent_link_type": 3,
        "name": "rel_std_class_tenant_class",
        "notes": "related type link between swiftly standard classficication taxonomy and tenant classfication taxonomy",
        "metadata_schema": null,
        "inserted_datetime": "2024-01-05T22:14:12.497000",
        "updated_datetime": "2024-01-05T22:14:12.497000",
        "updated_by": "breann@swiftly.com"
    },
    {
        "link_type_id": 6,
        "parent_link_type": 3,
        "name": "rel_std_nav_tenant_nav",
        "notes": "related type link between swiftly standard navigation taxonomy and tenant-custom navigation",
        "metadata_schema": null,
        "inserted_datetime": "2024-01-05T22:15:11.017000",
        "updated_datetime": "2024-01-05T22:15:11.017000",
        "updated_by": "breann@swiftly.com"
    }
]
```

### See also

graph-query

## node-product-create

Create a join between a Node and a Swiftly Product (keyed by Product_Code (UPC or PLU))

### Request

POST /api/taxonomy/node-product/create

### Body

```
{
    "node_products": [
    {
      "node_id": "69B3AAA2-D11D-4DD7-B667-F7842BF90655",
      "product_code": "4011",
      "updated_by": "breanna@swiftly.com"
    },
    {
      "node_id": "69B3AAA2-D11D-4DD7-B667-F7842BF90655",
      "product_code": "94011",
      "updated_by": "breanna@swiftly.com"
    },
    {
      "node_id": "69B3AAA2-D11D-4DD7-B667-F7842BF90655",
      "product_code": "00074904100005",
      "updated_by": "breanna@swiftly.com"
    }

  ]
}
```

### Response

List of completed node_product records.

```
[
    {
        "node_id": "69B3AAA2-D11D-4DD7-B667-F7842BF90655",
        "product_code": "84011",
        "updated_by": "breanna@swiftly.com",
        "inserted_datetime": "2024-02-28T16:59:10.634226",
        "updated_datetime": "2024-02-28T16:59:10.634226"
    }
]
```

### See also

node-product-get

node-product-delete

## node-product-get

### Request

GET /api/taxonomy/node-product/get?node_id=5DFE9EA6-3716-4E98-9E08-8586BBA97EAE

GET /api/taxonomy/node-product/get?product_code=00732346288605

### Parameters

node_id

product_code

### Response

List of completed node_product records with product_name joined in

```
[
   {
        "node_product_id": 5,
        "node_id": "5DFE9EA6-3716-4E98-9E08-8586BBA97EAE",
        "product_code": "00856098008097",
        "product_name": "HOLIDAY COCKTAIL COLLECTION",
        "inserted_datetime": "2024-01-25T16:53:00.933000",
        "updated_datetime": "2024-01-25T16:53:00.933000",
        "updated_by": "breanna@swiftly.com"
    },
    {
        "node_product_id": 6,
        "node_id": "5DFE9EA6-3716-4E98-9E08-8586BBA97EAE",
        "product_code": "00732346288605",
        "product_name": "Coastal Cocktails (Coastal Cocktails Inc) Cocktail Mix Bottle 4.6foz X3",
        "inserted_datetime": "2024-01-25T16:53:00.933000",
        "updated_datetime": "2024-01-25T16:53:00.933000",
        "updated_by": "breanna@swiftly.com"
    },
    {
        "node_product_id": 7,
        "node_id": "5DFE9EA6-3716-4E98-9E08-8586BBA97EAE",
        "product_code": "00732346302509",
        "product_name": "Coastal Cocktails (Coastal Cocktails Inc) Cocktail Mix Bottle 2.3foz X5",
        "inserted_datetime": "2024-01-25T16:53:00.933000",
        "updated_datetime": "2024-01-25T16:53:00.933000",
        "updated_by": "breanna@swiftly.com"
    },
    {
        "node_product_id": 8,
        "node_id": "5DFE9EA6-3716-4E98-9E08-8586BBA97EAE",
        "product_code": "00732346301915",
        "product_name": "Unbranded Flavoured Mixer  Bottle 4.6foz X3",
        "inserted_datetime": "2024-01-25T16:53:00.933000",
        "updated_datetime": "2024-01-25T16:53:00.933000",
        "updated_by": "breanna@swiftly.com"
    }
]
```

## node-product-delete

Delete node to product relationships

either by the node_id or by a list of node_product_ids (keys to the records).

### Request

DELETE /api/taxonomy/node-product/delete

### Body

by node_id:

```
{
    "node_id": "69B3AAA2-D11D-4DD7-B667-F7842BF90655",
}
```

by node_product_id list:

```
{
     "node_product_ids": [8, 9, 10]
}
```

### Response

code 200, 404, 500

### See Also

node-product-get

## node-tenant-product-create

create a record that joins a node to a tenant product. This is different than a Swiftly Product in that a tenant_id is required and it registers a proprietary item_code, not a product_code (UPC or PLU).

the node can be specified either by node_id or source_id (if the graph_id is explicitly specified also)

### Request

POST api/taxonomy/node-tenant-product/create

### Body

by node_id

```
{
    "tenant_id": "hlnd",
    "node_products": [
    {
      "node_id": "14C270DF-9548-4D9D-A3B7-7553F56BE465",
      "item_code": "88133401260",
      "updated_by": "breanna@swiftly.com"
    },
    {
      "node_id": "14C270DF-9548-4D9D-A3B7-7553F56BE465",
      "item_code": "7003864188",
      "updated_by": "breanna@swiftly.com"
    }
  ]
}
```

by source_id

```
{
    "tenant_id": "hlnd",
    "graph_id": 49,
    "by_source_id": true,
    "node_products": [
      {
        "source_id": "Product/canned-fruit-for-cooking-and-baking",
        "item_code": "7205860692",
        "updated_by": "breanna@swiftly.com"
      }
    ]
}
```

## Params

| Field | Type | Required | Description |
|---|---|---|---|
| tenant_id | String | Yes | id of tenant to register items for |
| graph_id | Integer | No | Only required if “by_source_id” = true |
| by_source_id | Boolean | No | If True, graph_id is required and NodeProduct.source_id is required. |
| node_products | List[TenantNodeProduct] | Yes | list of TenantNodeProduct objects |

### TenantNodeProduct

| Field | Type | Required | Description |
|---|---|---|---|
| node_id | Uniqueidentifier | No | Required if not by_source_id |
| source_id | String | No | Required if by_source_id = true |
| item_code | String | Yes | tenant item_code identifier |
| updated_by | String | Yes | user that created or updated the record |

## Response

```
[
    {
        "tenant_id": "hlnd",
        "node_id": "14C270DF-9548-4D9D-A3B7-7553F56BE465",
        "item_code": "88133401260",
        "updated_by": "breanna@swiftly.com",
        "inserted_datetime": "2024-02-29T11:43:44.699578",
        "updated_datetime": "2024-02-29T11:43:44.699578"
    },
    {
        "tenant_id": "hlnd",
        "node_id": "14C270DF-9548-4D9D-A3B7-7553F56BE465",
        "item_code": "7003864188",
        "updated_by": "breanna@swiftly.com",
        "inserted_datetime": "2024-02-29T11:43:44.699578",
        "updated_datetime": "2024-02-29T11:43:44.699578"
    }
]
```

### See Also

## node-tenant-product-get

### Request

GET /api/taxonomy/node-tenant-product/get?tenant_id=hlnd&item_code=<item_code>&node_id=<node_id>

### Params

Either item_code or node_id but not both.

item_code: get all records for tenant_id where item_code

node_id: get all records for tenant_id where item_code

### Response

list of TenantNodeProduct objects

```
[
    {
        "node_tenant_product_id": 8,
        "tenant_id": "hlnd",
        "node_id": "14C270DF-9548-4D9D-A3B7-7553F56BE465",
        "item_code": "88133401260",
        "inserted_datetime": "2024-02-29T11:43:44.700000",
        "updated_datetime": "2024-02-29T11:43:44.700000",
        "updated_by": "breanna@swiftly.com"
    },
    {
        "node_tenant_product_id": 9,
        "tenant_id": "hlnd",
        "node_id": "14C270DF-9548-4D9D-A3B7-7553F56BE465",
        "item_code": "7003864188",
        "inserted_datetime": "2024-02-29T11:43:44.700000",
        "updated_datetime": "2024-02-29T11:43:44.700000",
        "updated_by": "breanna@swiftly.com"
    }
]
```

### See Also

## node-tenant-product-delete

delete all node_tenant_product records.

Can be done either by a list of the node_tenant_product_ids or by the node_id to which the items are joined.

If node_is is specified then node_tenant_product_ids cannot be present.

### Request

DELETE /api/taxonomy/node-tenant-product/delete

### Body

by id

```
{
   "tenant_id": "hlnd",
   "node_tenant_product_ids": [ 3, 4]
}
```

by node_id

```
{
   "tenant_id": "hlnd",
   "node_id": "14C270DF-9548-4D9D-A3B7-7553F56BE465"
}
```

### Response

200, 500, No JSON is returned.

### See Also

## node-tenant-product-autoclassify

This endpoint automatically adds node-tenant-product records for the specified tenant by inspecting the ingested tenant_product records and correlating it against the specified graph which contains the Product Classification nodes.

This is the first step of onboarding tenant products. After successful Classification, if the tenant Navigation taxonomy is ready, Categorization can be executed which follows links between Classification and the tenant's Navigation taxonomies to add Category node-tenant-product records.

Normal operation is not incremental. All records for the tenant and the graph will be deleted and replaced.

If records have been added by hand they will be deleted. So If the mapping has been hand-edited, incremental mode will preserve manually entered records.

### Request

POST /api/taxonomy/node-tenant-product/autoclassify

### Body

```
{
  "tenant_id": "hlnd",
  "graph_id": 110,
   "updated_by": "breanna@swiftly.com",
   "incremental": false
 }
```

| Field | Type | Required | Description |
|---|---|---|---|
| tenant_id | String | Yes | id of tenant to register items for |
| graph_id | Integer | Yes | must specify the graph_id which will be used to autoclassify (the Products Classification graph) |

| | | | |
|---|---|---|---|
| updated_by | String | Yes | updater |
| incremental | Boolean | No | If False (default), all existing recods will be deleted and rewritten for tenant_id/graph_id. Otherwise will  preserve records that were hand-entered. |

### Response

200, 500, No JSON is returned.

### See Also

node-tenant-product-autocat

## node-tenant-product-autocat

This endpoint automatically adds node-tenant-product records for the specified tenant by following graph links starting from the nodes saved as a result of the node-tenant-product-autoclassify process.  Nodes in the Swiftly classification graph are linked to each tenant's Product Classification (aka Navigation) graph. These cross-graph links are pre-established between the Swiftly Classification Graph and the template Swiftly  Standard Categories Graph which is cloned to create the starting point for each new Tenant's graph.  The cross-graph links are also cloned in the process.

the general flow us Up the classification Graph to all parent nodes, across to the Category graph and down the category graph to the leaf nodes.  The number of steps is tracked for each possible path between the origin Classlification node and the final leaf Category node.  There may be many paths to many destination nodes but the shortest path is judged as the best quality connection and is selected in the end

As a result new node-tenant-product records are written for the Category graph connected to the same tenant products, this one will be exported to the Product catalog ingestion system to assign Swiftly_category_ids to each product.

### Request

POST /api/taxonomy/node-tenant-product/autocat

### Body

```
{
  "tenant_id": "hlnd",
  "classification_graph_id": 110,
  "category_graph_id": 114,
  "classification_link_type_id": 10,
  "cross_link_type_id": 4,
  "category_link_type_id": 11,
  "updated_by": "breanna@swiftly.com",
  "incremental": false
 }

```

| Field | Type | Required | Description |
|---|---|---|---|
| tenant_id | String | Yes | id of tenant to register items for |
| classification_graph_id | Integer | Yes | must specify the graph_id which was used to autoclassify (the Products Classification graph) |

| | | | |
|---|---|---|---|
| category_graph_id | integer | Yes | the tenant's product category (navigation) graph. |
| classification_link_type_id | Integer | Yes | The value should be (10) |
| cross_link_type_id | integer | Yes | The value should be (4) |
| category_link_type_id | integer | Yes | The value should be (11) |
| updated_by | String | Yes | updater |
| incremental | Boolean | No | If False (default), all existing records will be deleted and rewritten for tenant_id/graph_id. Otherwise it will preserve records that were hand-entered. |

### Response

200, 500, No JSON is returned.

## node-tenant-product-export

Export a complete map of all tenant products associated with a node.

The output format is customized to the needs of the product ingestion process and returns source_id for the joined node and it's parent node if that node is not the Root of the graph.

The number of results should be nearly the number of products in the tenant catalog so pagination is required.  Suggested page size of 10000 seems to perform well.

If page * page_size > result set , the call will return an empty array [] indicating end of data.

### Request

GET /api/taxonomy/node-tenant-product/export?tenant_id=hlnd&graph_id=114&page_size=10000&page=10

### Params

tenant_id = the tenant for which to export the map

graph_id = the tenant category/nav graph identifier

page_size = the number of rows to return (suggest 10000)

page = page to return (starts at 0)

### Response

```
{
    "tenant_id": "hlnd",
    "graph_id": "114",
    "page_size": 5,
    "page": 100,
    "item_code_category_map": [
        {
            "item_code": "1820025872",
            "categories": [
                "Product/wine",
                "Product/alcoholic-beverages"
            ]
        },
        {
            "item_code": "1820025872",
            "categories": [
                "Product/distilled-spirits",
                "Product/alcoholic-beverages"
            ]
        },
        {
            "item_code": "1820025883",
            "categories": [
                "Product/beer",
                "Product/alcoholic-beverages"
            ]
        },
        {
            "item_code": "1820025883",
            "categories": [
                "Product/wine",
                "Product/alcoholic-beverages"
            ]
        },
        {
            "item_code": "1820025883",
            "categories": [
                "Product/distilled-spirits",
                "Product/alcoholic-beverages"
            ]
        }
    ]
}
```

### See Also

## banner-graph-create

Create a record that registers a graph to a banner (tenant) for a specific purpose.

Purposes are defined by static reference table.

Only one graph can be registered to a given Banner for a specific purpose

Banner-graph records may optionally specify a node_id other than the Root node which will, for this purpose act as the proxy node for the sub-graph.

The record also stores a Status id indicating the status of the graph (in development, ready for test (PPE) or ready for publish to Production. This can be used to control publication functions.

### Request

POST /api/taxonomy/banner-graph/create

### Body

| Field | Type | Required | Description |
|---|---|---|---|
| banner_id | Uniqueidentifier | Yes | Banner (tenant) See shared.Banner.BannerId |
| graph_id | int | Yes | Graph that is assigned to the Banner |
| node_id | Uniqueidentifier | No | optional specification of a node within a graph that is the proxy root of the (sub) graph. |
| graph_purpose_id | int | Yes | id specifying the purpose of the graph for the banner. See DEF_graph_purpose reference. |
| graph_status_id | int | Yes | specifies the status of the graph for the purpose specified. |
| updated_by | string | Yes | user creating or updating the record. |

```

{
    "banner_id": "a4c3da6a-9072-44a7-b83c-9d1227d08037",
    "graph_id": 29,
    "graph_purpose_id": 1,
    "graph_status_id": 1,
    "updated_by": "breanna@swiftly.com"
}
```

graph_purpose_id definitions stored in DEF_graph_purpose

| graph_purpose_id | graph_purpose_name | notes |
|---|---|---|
| 1 | PRODUCT_CATEGORIES | Tenant Product Categories (Nav) |
| 2 | COUPONS_CATEGORIES | Tenant Coupon Categories |
| 3 | PRODUCT_COUPON_CATEGORIES | Combined Product and Coupon Categories |

graph_status_id defintions stored in DEF_graph_status

| graph_status_id | graph_status_name | notes |
|---|---|---|
| 0 | PURGED | The graph has been truncated (in anticipation of re-clone) |
| 1 | IN_DEVELOPMENT | Under devevelopment |
| 2 | PUBLISHED_DEVELOPMENT | Is published to Development (see published_datetime) |
| 3 | READY_PPE | Ready for Pre-Production Environment |

|  |  |  |
|---|---|---|
| 4 | PUBLISHED_PPE | Published to PPE (see published_datetime) |
| 5 | READY_PRODUCTION | Ready to publish to Production |
| 6 | PUBLISHED_PRODUCTION | Published to Production (see published_datetime) |

### Response

```
{
    "banner_graph_id": 6,
    "banner_id": "A4C3DA6A-9072-44A7-B83C-9D1227D08037",
    "graph_id": 29,
    "node_id": null,
    "graph_purpose_id": 3,
    "graph_status_id": 1,
    "published_datetime": null,
    "inserted_datetime": "2024-03-04T11:43:51.510000",
    "updated_datetime": "2024-03-04T11:43:51.510000",
    "updated_by": "breanna@swiftly.com"
}
```

## banner-graph-get

Get a banner-graph record by either the the bander_graph_id, banner_id or the graph_id along with an optional graph_purpose_id.

In addition, if no arguments are passed, all banner_graphs will be returned in a list.

Common use case, I know what the banner_id is; I need to know what graph is the Product/Coupon Category graph (graph_purpose_id=3), pass ?banner_id=<my-banner-id>&graph_purpose_id=3

### Request

GET /api/taxonomy/banner-graph/get?banner_graph_id={}&banner_id={}&graph_id={}

### Parameters

| Field | Type | Required | Description |
|---|---|---|---|
| banner_graph_id | Integer | No | id of the banner_graph record (see get) |
| banner_id | Uniqueidentifier | No | the id of the banner |
| graph_id | Integer | No | the id of the graph |
| graph_purpose_id | Integer | No | get only banner_graphs with this purpose_id. Best used with banner_id to get the specific graph you are looking for. |

### Response

```
[
     {
        "banner_graph_id": 4,
        "banner_id": "A4C3DA6A-9072-44A7-B83C-9D1227D08037",
        "banner_name": "Homeland",
        "graph_id": 29,
        "graph_name": "animals",
        "node_id": null,
        "graph_purpose_id": 1,
        "graph_purpose_name": "PRODUCT_CATEGORIES",
        "graph_status_id": 2,
        "graph_status_name": "PUBLISHED_DEVELOPMENT",
        "published_datetime": "2024-03-05T11:11:50",
        "inserted_datetime": "2024-03-04T11:11:50.400000",
        "updated_by": "breanna@swiftly.com"
    }
]
```

## banner-graph-update

Banner-graph-update allows limited PATCH type update to banner-graph records.

The banner-graph-id must be available.

The fields that can be updated are: graph_status_id and node_id.

for all other fields (purpose, graph...) delete the record and create a new one.

### Request

POST /api/taxonomy/banner-graph/update

### Body

| Field | Type | Required | Description |
|---|---|---|---|
| banner_graph_id | Int | Yes | id if the banner_graph recodrd |
| node_id | Uniqueidentifier | No | optional specification of a node within a graph that is the proxy root of the (sub) graph. |
| graph_status_id | int | No | specifies the status of the graph for the purpose specified. |
| published_datetime | string | No | If graph_status_id is set to 2, 4, 6, set the datetime of publication. |
| updated_by | string | Yes | user creating or updating the record. |

```
    {
        "banner_graph_id": 4,
        "graph_status_id": 2,
        "published_datetime": "2024-03-05T14:10:12",
        "updated_by": "breanna@swiftly.com"
    }
```

### Response

```
{
    "banner_graph_id": 4,
    "banner_id": "A4C3DA6A-9072-44A7-B83C-9D1227D08037",
    "graph_id": 29,
    "node_id": null,
    "graph_purpose_id": 1,
    "graph_status_id": 2,
    "published_datetime": "2024-03-05T14:10:12",
    "inserted_datetime": "2024-03-04T11:11:50.400000",
    "updated_datetime": "2024-03-04T12:10:39.403000",
    "updated_by": "breanna@swiftly.com"
}
```

## banner-graph-delete

Delete a banner_graph record by either it's record id or by the banner_id and the purpose. Either call will delete at most 1 record. The function does not return anything but a status code.

banner-graph-update only supports updating of limited fields (intentionally).  Use banner-graph-delete to remove a record if adding a different graph for a banner or changing it's function. These are considered identity changes so are not allowed by update.

### Request

DELETE /api/taxonomy/banner-graph/update

### Body

by id:

```
{ "banner_graph_id": 1 }
```

by banner_id and purpose:

```
{
   "banner_id": "A4C3DA6A-9072-44A7-B83C-9D1227D08037",
   "graph_purpose_id": 1
}
```

### Response

200, 500

### See Also

banner-graph-update

## banner-graph-publish

Publish the graph associated with the banner_graph to the specified PGSQL environment.

The graph_status_id of the banner_graph must be set to the “ready” level for the respective environment in order for the publish to work.

Publication, whether successful or failure will be logged. See below

### Request

POST /api/taxonomy/banner-graph/publish

## Body

```
{
    "banner_graph_id": 7,
    "environment": "PROD",
    "updated_by": "breanna@swiftly.com"
}
```

## Response

200, 404, 500

Successful response:

```
{
    "status": "success",
    "nodes_published": 175,
    "links_published": 174
}
```

## See Also

banner-graph-publish-log-list