Visualization API

For a simpler view of using the Grouper visualization features, including screenshots, visit the Grouper Visualization UI page.

- Introduction
- Graphing API
  - Setting the configuration
  - Building the graph
  - Graph nodes and edges
- Styles
  - Typical methods
  - Style inheritance
  - Styles for graph nodes and edges
- Historical discussion notes
- Visualization form
- Representing objects
  - Group
  - Folder
  - Connectors
- Visualizations
  - Groups as members of a group
  - Groups that use a group
  - Groups in a folder
  - Groups for user
  - Groups for user in folder
- Text only
- Notes from recent emails on the visualization topic are below:
  - See also

Introduction

Grouper visualization (available in a Grouper 2.4 patch and above) includes both the API and the visualization UI, allows you to see the relationships between Grouper objects. The relationships may be

- groups within a stem
- group memberships
- loader jobs that populate groups
- PSPNG provisioning targets
- components of a composite group

Grouper visualization utilizes the Grouper graph API -- a set of Java classes to build a directed graph. A graph is initialized with a starting object, such as a group, stem, or subject. Other options can be configured, such as whether to show certain objects types, or how many levels to follow parent/child connections. Building the graph starts a process of following the parents and children of the starting object, recursively until all reachable nodes have been identified. When complete, the results of the analysis, along with some statistics, can be retrieved as needed by the particular visualization module.

The resulting graph consists of all the reachable nodes as a set, and all of the vectors from each parent node->child node. If starting from a stem, it will recursively visit all the parents and children of its child groups. The nodes and edges can be retrieved as sets for iteration. Or, the starting node and each node can be queried for all the parents and children it connects to, so that the nodes can be visited recursively.

Visual styles are configuration objects that define string-based properties for specific style object types. Style object types are a fixed set, including for example: "stem", "start_stem", "loader_group", "subject", etc. The properties that can be defined for a style object are open-ended, and can be extended as needed for a visualization application. Object properties for a style belong to specific modules, such as "graphviz" or "text", or to the default style set. Styles can set an inherit property, so that it can reuse properties from a parent style without needing to repeat them. For example, a start_stem style can inherit the stem styles, and override just the background or other small number of properties.

Graphing API

There are public Java API methods to build the graph, and GSH can also be used.

Setting the configuration

```
import edu.internet2.middleware.grouper.app.graph.RelationGraph
def graph = new RelationGraph().assignStartObject(inObject).assignParentLevels(-1).assignChildLevels(-1).
assignShowMemberCounts(true)
```

A graph is instantiated as a new object, and then various configuration options are chained, as in the example above.

```
graph.assignStartObject(GrouperObject)
graph.assignStartObject(Subject)
```

The one options that must be set is the start object. A starting subject can be a Group or a Stem (as both are subclasses of GrouperObject), or a Subject. The root stem is a valid starting stem.

```
graph.assignParentLevels(int)  // default -1
graph.assignChildLevels(int)   // default -1
assignMaxSiblings(int)         // default 50
```

These are the number of parent and child levels to recursively visit before stopping prematurely. A value of -1 represents all levels. Note that there is a hard limit of 100 levels, as an emergency stop against infinite cycles, which shouldn't occur under normal circumstances.

```
graph.assignShowMemberCounts(boolean)  // default true
graph.assignShowLoaderJobs(boolean)  // default true
graph.assignShowProvisionTargets(boolean)  // default true
graph.assignShowStems(boolean)  // default true
```

Boolean options to include or skip certain data. If there are many groups in the graph, including member counts may be time-consuming, so it may help to disable it. The other options may help to simplify output if there are many objects in the resulting graph.

```
graph.assignSkipFolderNamePatterns(Set<String>)  // default none, use e.g. ["^etc:.*", "^$"] to exclude root
and etc folders
*graph.assignSkipGroupNamePatterns(Set<String>)  // todo
```

A stem or group matching any of the regular expression patterns in the set will be excluded from the result. The initial folder set excludes the etc: subtree and the root stem (which has a blank string as its name).

```
graph.getParentLevels()
graph.getChildLevels()
graph.isShowMemberCounts()
graph.isShowLoaderJobs()
graph.isShowProvisionTargets()
graph.isShowStems()
graph.getSkipFolderNamePatterns()
*graph.getSkipGroupNamePatterns()
```

Retrieve the current settings for the graph configuration

Building the graph

```
graph.build()
```

The build() method starts with the starting node, visiting all its parents and children recursively. If starting with a stem, it will also visiting the parents and children of all the child groups within the stem. Once the method returns, methods can be called to retrieve various aspects of the resulting graph.

The graph contains a set of nodes (of type edu.internet2.middleware.grouper.app.graph.GraphNode), and a set of directed edges (of type edu.internet2.middleware.grouper.app.graph.GraphEdge) having a parent node and child node.

```
graph.getEdges().each { e -> ... }
graph.getNodes().each { n -> ... }
```

These methods retrieve all the edges or nodes in the graph, as an unordered but unique set.

```
graph.getLeafParentNodes()
graph.getLeafChildNodes()
```

These methods retrieve all the nodes which do not have a connection to any parent or child, respectively.

```
graph.getMaxParentDistance()
graph.getMaxChildDistance()
```

These return the maximum number of levels of parents or children away from the starting node, respectively. These can be used, for example, in a text-based visualization to know how far to indent without surpassing the margin.

```
graph.getNode(GrouperObject)
```

Retrieves the node which contains the specified object. For a group or stem, these objects are subclasses of GrouperObject and can be passed directly. For a subject, wrap the subject with a `new GrouperObjectSubjectWrapper(subject)` object. Provisioner objects are a pseudo-object of class GrouperObjectProvisionerWrapper, and only contain a unique id string (the name and display name repeat the id value). To get a provisioner node, pass the wrapped object as `new GrouperObjectProvisionerWrapper(id)`.

```
getNumLoaders()
getNumGroupsFromLoaders()
getNumGroupsToProvisioners()
getNumProvisioners()
getNumSkippedFolders()
*getNumSkippedGroups()
getNumMembers()
```

These return statistics about the build that are updated after it completes.

Graph nodes and edges

The nodes and edges created during the build have data that can be queried.

```
node.isGroup()
node.isStem()
node.isSubject()
node.isLoaderGroup()
node.isProvisionerTarget()
node.isIntersectGroup()
node.isComplementGroup()
node.isStartNode()
node.getGrouperObject()

node.getDistanceFromStartNode()
node.getParentNodes()
node.getChildNodes()
node.getMemberCount()

edge.getFromNode()
edge.getToNode()
```

There is also a method on nodes and edges, `getStyleObjectType()`, which invokes a number of conditional logic to determine which visual StyleObjectType (See below) the object corresponds to.

Styles

Visual styles are sets of properties defined in grouper.properties (or grouper.base.properties). Property values are strings aggregated into specific style object types, and these object types are then aggregated into a style set module. Instantiating a VisualSettings object the first time will initialize all of the visualization modules defined in grouper.properties. Using the VisualSettings object, it can be queried to retrieve property values for specific properties.

While the properties are open ended, the object types are a closed set. The set of objet types includes:

- default, graph
- stem, start_stem, skip_stem
- group, start_group
- loader_group, start_loader_group
- complement_group, intersect_group
- subject, start_subject
- provisioner
- edge, edge_loader, edge_provisioner, edge_stem, edge_membership
- edge_complement_left, edge_complement_right, edge_intersect_left, edge_intersect_right

Typical methods

```
import edu.internet2.middleware.grouper.app.visualization.*
def settings = new VisualSettings()

settings.getGroupUiLinkPrefix()
settings.getStemUiLinkPrefix()

def styleSetNames = settings.getStyleSetNames()  // ["default", "graphviz", "text"]
def defaultStyleSet = settings.getDefaultStyleSet()
def styleSet = settings.getStyleSet("graphviz")

def styleNames = styleSet.getStyleNames() // ["default", "graph", "stem", "start_stem", "skip_stem", "group"
...]
def styleGroup = styleSet.getStyle("group")

def groupFont = styleGroup.getProperty("font")
def stemColor = styleSet.getStyleProperty("stem", "color")
def stemBorderOrDefault = styleSet.getStyleProperty("stem", "border", "0")
```

Style inheritance

Style properties can be inherited from a specified inherit style, or from the default style set which is also configured as its own module. For example, given the following grouper.properties configuration:

```
visualization.style.default.font = Courier,monospace
visualization.style.default.font_color = black
visualization.style.default.font_size = 12.0

visualization.style.stem.font_color = purple
visualization.style.start_stem.inherit = stem

visualization.module.graphviz.start_stem.font_color = red
visualization.module.graphviz.default.font_size = 11.0
```

The graphviz start_stem style will have the following properties:

- font: Courier,monospace
- font_color: red
- font_size: 11.0

If a style inherits from another style, it looks for the property in a few different places, in order.

1. If the style directly defines the property, return it
2. If the module defines the property for the default style, return it
3. If the default module defines the property for the same style type, return it
4. (recursively) Look through the inherited style for a property defined. This may be recursive, if the inherited class is also inherited
5. Return null, or return the default value if using the overloaded method with a default value

Styles for graph nodes and edges

During the Graph build process, all nodes and edges are mapped to a specific style object type. For example, if the start node contains a Subject, its object type will be *start*subject*. If an edge is from a loader group to a normal group, its type will be* edge*loader*. This feature can be used to iterate through nodes and edges to look up their properties, specific to their object type. For example:

```
styleSet = new VisualSettings().getStyleSet("graphviz")

graph.nodes.each { n ->
    labelStyles = styleSet.getStyleProperty(n.styleObjectType, "label_styles", "")
    println "<label ${labelStyles}><node name='${n.grouperObject.name}'/></label>"

    // OR,
    def objStyles = []
    [VisualStyle.Property.SHAPE, VisualStyle.Property.STYLE, VisualStyle.Property.COLOR].each { p ->
        val = styleSet.getStyleProperty(n.styleObjectType, p.name)
        if (val != null) {
            objStyles.add("${p.name}=${val}")
        }
    }
    println "<node ${objStyles.join("; ")}>...</node>"
}
```

For more examples, look in the Grouper project source code, in directory grouper/misc/visualization/.

Historical discussion notes

This is a page for information on Grouper Visualization an item that has been requested by the community and is on the Grouper roadmap.

We will start with groups which are members of a group, and go from there

Visualization form

Before visualizing, the user can pick some options which would change the representation

Representing objects

Group

Show display name

Group types: loader, ref, policy, etc

Number of members (cached in attribute?)

Folder

Show display name

Folder types: ref, basis, etc

Connectors

From	To	Description
Folder	Group	Arrow from folder to group, means group is in folder

Visualizations

Groups as members of a group

This visualization will show groups which are members of a group.

Input form: constrain output to a certain folder?

Groups that use a group

Show groups which have this group as a member

Include privileges?

Groups in a folder

In a folder, show all groups and their relationships

Include groups outside the folder if the groups relate to them?

Include a representation of what folder groups are in?

Groups for user

For a user, show the groups the user has

Include privileges?

Groups for user in folder

Focus on a user in a folder and show the groups and relationships among groups

Include privileges?

Text only

We should have a text only mode... this could be a checkbox at the bottom of the form before displaying that defaults to "visual", but could be "text only" i.e. it would just have all the text for whats on the graph. This would be for several purposes

1. Accessible

2. If the technology of D3 goes stale, there is still a way to get the info
3. Maybe people will take that and copy/paste to excel or do something with it

Notes from recent emails on the visualization topic are below:

There are two parts here (at least), the grouper part which is needed either way (in groovy or java), and the visualization part. You have the visualization part in graphviz which has two downsides:

1. Command line,

2. (is this true?) non-interactive. Nick has used D3 a bit I believe and thinks it could work (right?)

examples:

1. https://github.com/d3/d3/wiki/Gallery
2. This one looks good, auto-lays out, draggable, on mouseover it puts metadata and links on the left
   http://languagenetwork.cotrino.com/
3. Clickable
   http://ramblings.mcpher.com/Home/excelquirks/gassites/d3nodefocus
4. Draggable and metadata on right
   http://visualdataweb.de/webvowl/#
5. D-3 graphviz
   https://github.com/magjac/d3-graphviz
6. D-3 graphviz example
   https://bl.ocks.org/magjac/4acffdb3afbc4f71b448a210b5060bca

See https://github.com/gettes/grouper-graph

from Chad Nov 30, 2018
I just tried out the grouper-graph code last night. Just for kicks I grabbed a Grouper Training Environment image for it to work on some sample data. It works great! I just needed to escape the & in the urls in the svg but otherwise no problems. it's an amazing way to get a quick understanding of the whole system (and may be useful to add to GTE docs as well). Great work! I took a quick look at the code, and to me it looks in great shape already :) Maybe where I can help is plugging this into the UI. If you have design ideas, I can implement whatever you come up with. Also, I could help converting the groovy code to Java. But I think groovy has a compiler to create a Java class from a script, so there is that option too. I don't know yet how the graph-viz js will work -- hopefully it will just build a standalone js file that can be added to the project.

See also

Grouper Visualization UI