Angular Data Access API

Each user view interfaces with the back end via an angular service. Angular JS uses the model view controller model of development. Controllers store and process data concerning specific functionally, while directives (javascript) and templates (html) describe how that data and functionality is displayed at the front end. Controllers are hierarchical, and can inherit data from parent controllers. Services provide common functionality that is accessible from all controllers. For all practical purposes the API that front end user use is the data service within the application.

BLIZAAR services

The Blizaar services can be found in the sub folder “blizaar_platform\app\js\services”. To be accessible from a controller a service must be included in the controller declaration. This is referred to as subscribing to a service. Each service is a single instance, containing the same data and functional for all controllers which access it. The primary service is the data service which interface with the node.js middleware. No direct connection to the middleware should be made from any other place in the Front end system.

dataService.js

This is the most important service within the front end angular application. This service is used to retrieve graph data from the node.js server. This data can be graphs, node types or edge types or information such as functionality available in the back end for graph processing. Whatever the information being stored in the back end is, it is delivered via this service. It stores the graph structure that has been retrieved form the back end in a multilayerGraphObject , accessed via the multilayerGraph service.

Many functions dispatch a http POST or GET to the back end. As these calls are asynchronous the front end returns immediately rather than hang and wait for the response. Therefore we use promises, by which we specify functionality, usually using a “.then” function, to handle the call when it does return. When the promise associated with a http call(or any asynchronous function) is returned on, i.e. the “then” function is invoked, the JavaScript global variable “this” may not necessarily point at the original data service object to avoid errors so we set a “self” variable within the parent functions cope to be used with asynchronous calls when they want to refer to this data service's attributes.

Here we provide a brief description of the data access functions offered by the service, but for more information see the service source file. All of the functions listed below make up the functionality that is used to access the back end data as well as process the grpah on the server.

Get Data Methods

getGraphData

This function returns the current graph stored in the multilayerGraphService, without querying the back end.

initGraphData

This function retrieves an initial graph from the back end data base.

getAccessibleFileList

To allow for testing with specific graphs it is possible to store graphml files on the server which can be retrieved by the user. This function returns a list of all graphml files on the server.

getNodeLabelList

This function queries (via http post) the middleware for all node labels (which is a neo4j term and can be thought of as a type descriptor for a node), which are associated with the currently selected graph.

getEdgeTypes

This function queries (via http post) the middleware for all edge types , which are associated with the currently selected graph.

getNumericAttributeNames

This function queries the graphs currently stored in the data service and returns a list of all node attributes which are numeric.

getCentralityList

This function asynchronously queries the back end (via http post) and returns a list of all node centralities which are available at the back end

getServerLayoutList

This function asynchronously queries the back end (via http post) and returns a list of all graph Layouts which are available at the back end

getServerClusteringList

This function asynchronously queries the back end (via http post) and returns a list of all graph

clusterings (i.e. using graph structure) which are available at the back end.

getAvailableNodeNames

This function gets (via http post) the name of all nodes which are stored in the back end, and have the currently specified node labels, and are attached to edges of the currently specified edge type.

queryGraph

This function creates a new front end user graph, by querying (via http post) the back end system. The new graph is constrained to the specified node labels, and relationship types. The function contains a start node and end node parameter. If the user specifes start node and end node, the back end returns all nodes within 1 hop of the start and end nodes as well as all nodes on the shortest paths between them. If only the start node is specified, only that node and its neighbours are returned.

expandNode

This function takes a target node and returns (via http post to the back end) its neigbours from the master graph that are not included in the current graph, constrained by node label and relationship type, and adds them to the users current gaph.

populateGraphIcons

Retrieves a mapping to icons stored on the server for each node type. The mapping between icons and types is speciiefied in server config files.

Graph Processing Operations

getCentrality

This function asynchronously requests the backend (via http post) to calculate the specific centrality for the users front endgraph.

doServerLayout

Send request to the the middleware (asynchronously) to layout the graph with the specifed layout(using the igraph Library from R). A map of positions fro each node is returned form the server and applied to the graph.

doServerClustering

Send request to the the middleware (asynchronously) to cluster the graph with the specified clustering (using the igraph Library from R). The clustering is stored in the cluster_ID field of each node.

updatePositionToServer

Updates the position of all nodes in the graph for the server instance of the graph.

updateNodePositionToServer

Updates the position of all nodes specified in the input array for the server instance.

Graph Storage Methods

saveGraphOnServer

Requests the server to save the graph so it can be retrieved at a later date. The users must specify a name for the grpah to be saved under.

getSavedUserGraphsFromServer

Fetches a list of previously saved graphs, for this user, from the server.

loadGraphFromServer

Fetches the graph specified by name from the server and loads it.

deleteGraphFromServer

deletes specified graph from the server

MultiayerGraphService.js

The Multilayer service wraps an instance of the MultilayerGraph javascript class. The purpose of this server is simply to provide a unique instance of the multilayer graph object that is accessible across the application. While this service cna be accesed acorss the application, for most purposes it will be for most puposes it is accessed via th dataService.js service:

MultilayerGraph Object

Strictly speaking this is is not a an API function, however it is used extensively by the application so its structure is described here. The constructor function takes in an array of nodes and links (also referred to as vertices and edges, respectively), in the format commonly used by front end visualization applications such as d3.js. The original node and link objects are referenced and not copied. All parameters of nodes and links are are attached to the objects themselves, and not stored in a look up table If the the input array links do not contain references to the endpoint nodes, but only the ids of the nodes, the source and taargets ids are replaced with references to the endpoint nodes.

To simplify finding nodes the object definition also contains a look up tables to fetch node and link objects by ID. These look up tables are “node” and link“ and they provide access to node and link objects without an array search. There is also and adjacency list lookup (adjacencyList). When given a node this map returns a list of all neighboring nodes in the graph (regardless of edge direction). Externally this function is accessed by the objects getNeighbours function.

The multilayerGraph object recursively defines layers as multilayerGraph objects, stored in an array named “Layers”. It is implicit that the current instance is a master data set, and that all graphs in the layers object are the layers derived from this master data set.

Important properties

nodes

An array storing all of the nodes (vertices) in the graph. Node properties are store to each of the node objects.

Links

An array storing all of the links (edges) in the graph. Link properties are stored in each of the link objects.

node

A javascript object that acts as a map between node id and node objects.

node

A javascript object that acts as a map between link id and link objects.

adjacency List

A javascript object that acts as a map between node ID and an array of references to all nodes

adjacent to the key node (regardless of edge direction). This object is not populated by default when the graph is created , so it needs to be populated by an explicit call to the function *buildAdjacencyList*.

name

Stores the name of this graph / layer.

layers

An array storing the layers associated with this graph. Each object in the array should be an instance of multiLayerGraph.

visualAttributes

A javascript attribute containing various visual parameters , such as minimum node size , maximum node size, and the attribute on which node size is based.

Important methods

getNeighbours

Uses the adjacency list object to return an array containing references to all neighbours of the input node. If the the adjacencyList object has not been created at this stage, the function *buildAdjacencyList* will be called first.

buildAdjacencyList

Populates the adjacency list object described above.

copyStructure

Creates a new graph with the same node and link structure as the current one. No node or edge attributes other than id are copied.

copyStructureAndPosition

Creates a new graph with the same node and link structure as the current one. No node or edge attributes other than id and x and y position are copied.

copyGraphEntity

Copies a node in the graph, returning a new node with the same properties

updatePosition

Takes a map of Node ids to 2D positions (x and y) and updates the graph with it, and returns the number of nodes that were updated.

mapLinkEndsToNodeReferences

If the graph only maps link endpoints to vertices by id , this method replaces the source and target Ids with referenecs to the source and target objects.

removeDuplicateEdges

A utility function which edits a graph so that each each vertices pair has at most 1 edge between them. This methodd should not be used when working with Multi edge Graphs.

prepareGraph

When a new graph is returned form the server this method ensures it has all necessary attributes to function at the front end.

setColorCategories

Queries nodes for the specified field to build a map of colors to categories, to be used when coloring the node

updateNodeAttributes

Use the input nodemap to update the attirbutes of the nodes specified in the map,.

updateAllNodeAttributes

Use an input nodemap to update the attributes of all nodes in the graph.(using the array is slightly faster than the preceeding method for when the whole graph is being updated).

getNumericAttributeNames

Queries the graph to find the which numeric attribute exists for the nodes. Assumes all attributes are present on the first node of the graph

setNodeSize

Iterate over the graphs setting the size of each node based on an input range and an attribute value. Dependencey on d3.scale

pushLayer

Creates a new multlayerGraph object based on the input node and link arrays and adds it to the layer array of the current graph.

setLayerAtIndex

Similar to above function except the new graph is created at the specified index.