REM- Reasonable Experiments Manager: Developer Guide

# General Description:

The Graph Generation System was designed to provide an interface for graph generation based on experiment data, using a client-server web interface.

The user can connect to a website, select a specific experiment, select a graph model and parameters, and display a graph.

The graphs are generated using the Bokeh library in python and are interactive- allowing zoom pan and save operations.

The system also allows to save/load/delete presets per experiment file, as well as an option to define your own plugin.

### Definitions:

**Experiment file** – a file containing the experiment data

**Model** – a type of graph. E.g. – line, heat map, step etc

**Input Parameters** – the input fields with the available options. Defined per model. (e.g. dropdown menu with all the machine names)

**Selected parameters** – the parameters selected by the user from the input parameter (e.g. 'time\_stamp')

**Preset** – contains a name, the graph model, and the selected parameters (e.g. myline: line, time\_stamp , performance, vm-1)

**Graph** – a visual representation of the selected model and input parameters

**Plugin** – a python file which can be written by the user. Contains a definition for a graph model and input parameters

# Front End:

## File Explorer:

All file explorer functions are located in **file-explorer.js**

The explorer is created using the jstree library.  
It works by first contacting the server for the initial creation- this provides the root directory and the files and folders that are in it.  
Then upon user selection, if the user double-clicked on a folder an update request is sent to the server and the tree's data is updated with the new directory's information.  
In order to support '1 up' functionality, the directory's path is saved as a global variable.

#### Functions:

//draws the initial tree with the data received from the server - called once  
**function drawTree(data)**

//updates the tree with new data  
**function updateTree(data)**

//makes a request to the server to get the tree information with the given url  
//if the request was successful, the success function is then called  
**function callListdir(url,successFunction)**

//checks that the selected id is indeed a file-type **function isValid(node)**

//sends the selected experiment path to the server to get the experiment parameters  
//upon success, moves to handleParameters which generates the corresponding UI  
**function sendExperimentToServer()**

Global variables:  
var currDir; //holds the current directory

## Parameter Selection:

All the parameter generation and graph display functions are located in **parameter-handling.js**

The experiment parameters that were received from the server are used to generate the input UI elements dynamically.

The response from the server appears in the following format:  
{  
cols:["arguments", "arguments-description", "avg", "cache-auction-round", "cache-auction-time", "cache-bid",…]

models:{test: {y\_axis: {source: "cols", type: "single"}, x\_axis: {source: "cols", type: "single"},…},…}

name:["Host", "vm-1", "vm-2"]  
}

**models:** a dictionary of the available models- key is the model name, and object is the model input parameters. The object itself is a dictionary describing the type of input (single- single selection dropdown menu ; multiple- multiple selection list ; radio – radio buttons etc..)  
and describing the source from which to fill in the options ('cols' for example, would direct us the dictionary object with key 'cols' from which we'll fill our dropdown list options).

**Cols**: Contains the options needed to fill the model input parameter.  
'cols' as a name is not obligatory, just need to match the name that appears in models.  
  
e.g.:

models:{test: {y\_axis: {source: "A", type: "single"}, x\_axis: {source: "B", type: "single"},…},…}

A:["arg1", " arg2", "arg3"…]

B: ["arg1", " arg2", "arg3"…]

**Name**: machine name from exp file meta-data.

#### Functions:

//fills the model list and moves to the parameters tab - called when server returns selected experiment info  
**function handleParameters(data)**

//collects all the graph parameters selected by the user,  
//generates a json and sends to the server  
**function fetchParametersAndGetGraph()**

//gathers all the parameters from the parameters form and returns a json – used in 'fetchParametersAndGetGraph'  
**function fetchParameters()**  
  
//sends the given parameters to the server json – used in 'fetchParametersAndGetGraph'  
**function sendParametersToServer(data)**  
  
//display the graph received from the server  
**function displayGraph(data)**  
  
//fills the model list with available models and images  
**function fillModelListImages(data)**

//requests the preset list from the server - used to fill the 'load' section  
**function getPresetList()**

//fills the preset list with the response from the server  
**function fillPresetList(data)**

//generates the required graph parameters   
//-creates the required input fields based on the selected model function **generateGraphParameters(data, model)**

The file also contains some helper functions:

//returns true if form is filled, and false otherwise  
**function isParametersFormValid()**

//resets the display - hides loaders and error messages  
**function resetDisplay()**

//fixes the height of the graph display tab  
**function fixHeight()**

helper-generation-functions.js contains some additional helper functions solely for generating html fields in javascript.

Such functions include-

**function generateDatalist(id)  
function generateSelect(id,isMultiple)  
function generateButton(id,className,text)**

etc.

## Client-Server communication:

# Back End:

## How to add a new plugin

To add a new graph type plugin you need to create a .py file in server/GraphPlugins the file must contain a class by the same name as the file i.e. steveGraph.py will contain the class steveGraph the class requires 2 functions

def getparameters(self): this function should define a dict (or preferably an ordered dict) that defines what value the web application will request form the user and the source and type of this date here is an example, assuming the class name is steveGraph: params = OrderedDict() params['x\_axis'] = {'type':'single','source':'cols'} params['y\_axis'] = {'type':'single','source':'cols'} params['group\_by'] ={'type':'single','source':'name'} return {'steveGraph':params} the valid parameters to 'type' are currently: single, multiple. where single will ask the use to pick exactly one item and multiple will lets the user pick more than one

the valid parameters to 'source' are cols and name, where cols would be the column names as they where parsed form the experiment file and name is the machine name from the experiment metadata, such as vm-1, vm-2

The return value should be a dict containing a single key:value pair where the key is the class name and the value is the parameters dict as shown above.

the second function will be: def plot(self, filename, sqlpath, x\_axis, y\_axis, group\_by):'filename' is the name of the experiment file we are working in, this is usually used for naming output files, such as CSV of teh data or the html of the graph 'sqlpath' is the path to the relevant sqlite database containing all the parsed experiment information all the other parameters should be the same as the parameters in the 'getparameters' function described above

this function should create a bokeh graph using bokeh.charts or bokeh.plotting (never import from bokeh.charts or bokeh.plotting directly as you may get collisions! instead import charts and/or plotting from bokeh and use them by their long names i.e charts.) then run the generated graph through 'components' function from bokeh.embed which generates a div and js component then it needs to save the js to a file and return a dict containing the div and the path to the js, such as:return {'div':div, 'js':js\_path}

the add an icon to the newly created plugin, drop a png with the class name into server/static/img/pluginImg/

## Configuration file:

The file server/config.py is the configuration file for this server

Currently the used keys are port: the port to start on (default 5000) experiment\_root\_dir: the root folder you want displayed in the file browser (default server/experiments) debug: True/False will enable/disable a lot of extra prints to the console