DSS Visualizer System Maintenance Guide

## What is the DSS Visualizer?

The DSS Visualizer system is an application that can present evaluator information collected using the ECEL tool in a graphical and easy to understand manner to researchers. An evaluator is defined as an individual who is conducting a cyber security penetration test or exercise. The evaluator could be a student or a security professional. These graphical visualizations will allow researchers to gain better insight pertaining to the actions of evaluators during cyber security events. This system receives data from the ECEL system and is responsible for visualizing the data in an intuitive manner. Data generated by ECEL includes evaluator keypresses, mouse clicks, timed/manual screenshots, system function calls, and network traffic data (both throughput and packet information). This tool will allow researchers to better understand cyber security incidents from the evaluator point of view and will eventually be used in order to better train cyber security professionals and students.

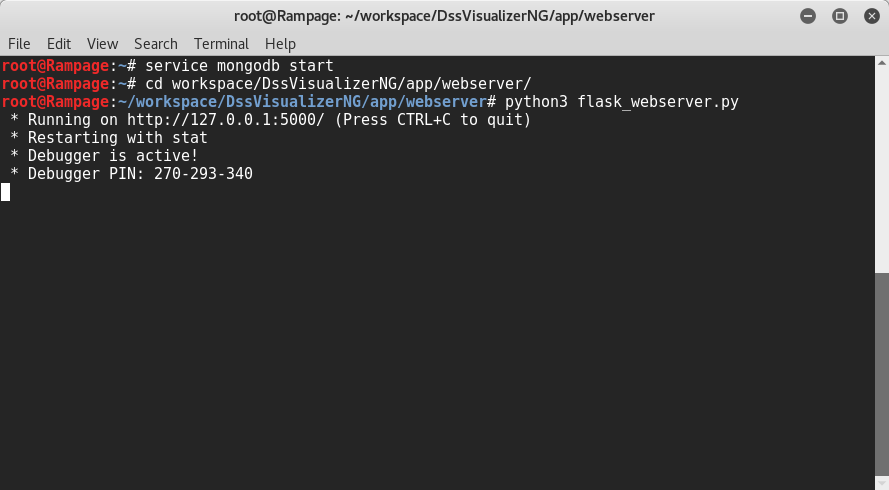
## What is the purpose of this document?

The purpose of this document is to teach system maintainers how the system works. This will give system maintainers knowledge on how the systems, information regarding how to create new plugins for the system, as well as guidance regarding how fix any potential bugs within the system. This artifact also serves as a documentation of the existing architecture of the system and should be maintained as the system is updated.

## Basic Overview of the System.

In order to start the system, both mongodb and the webserver must be loaded. In kali linux, mongodb can be started by typing “service mongodb start” in the CLI. The webserver can be started using the “flask\_webserver.py” script that is included with the project. Please refer to screenshot 1 for more info:

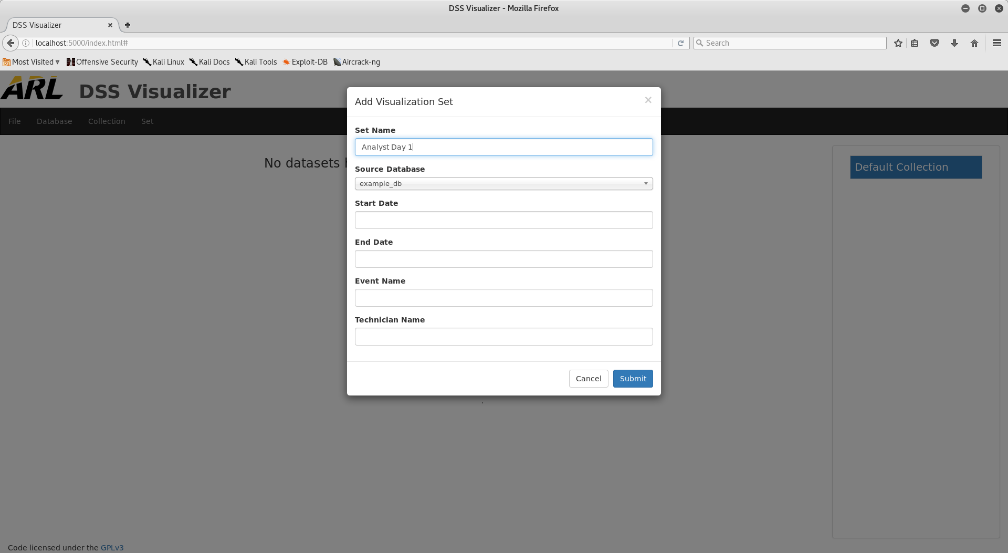
Screenshot 1: Starting database and webserver



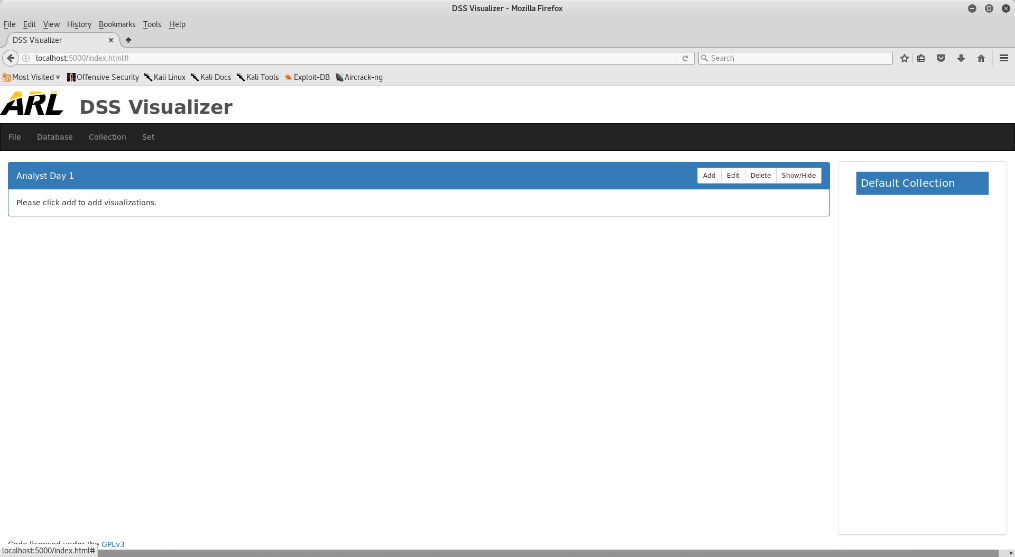
The system requires the user to upload ECEL data in order to generate useful visualizations. The user may create different databases in order to logically separate data or may choose to only use one database. It really depends on how the user wishes to partition the data. A database can be added using Database > Add. A database can be removed using Database > Remove. ECEL data can be uploaded to the database using Database > Upload.

Sets are containers for visualizations. The set defines the specific data to be retrieved from the server. This means that all visualizations within a given set are viewing the same data from the database. A set can have a source database, start date, end date, event name, technician name, and set name. Please refer to screenshots 2 and 3 for more info:

Screenshot 2: Adding new set.

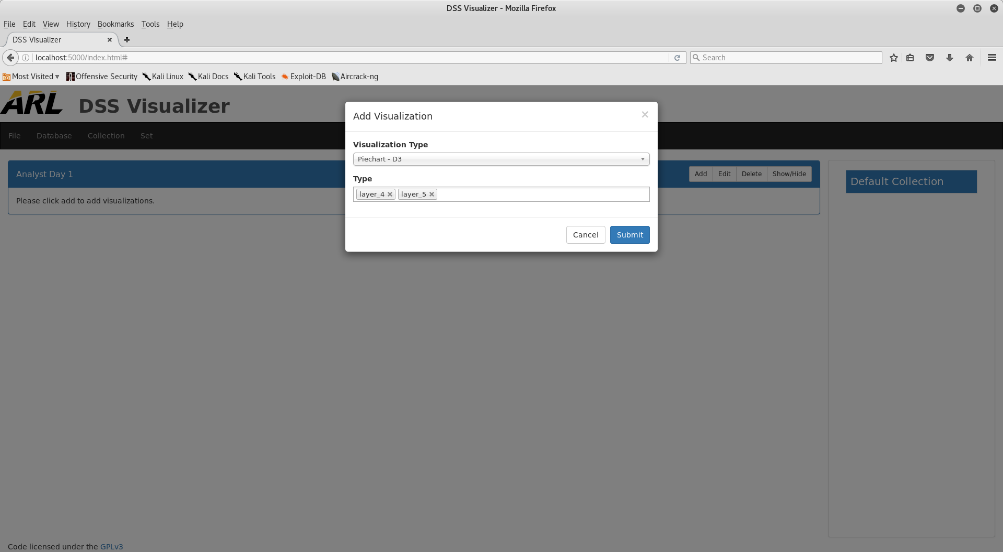


Screenshot 3: Set visible in workspace

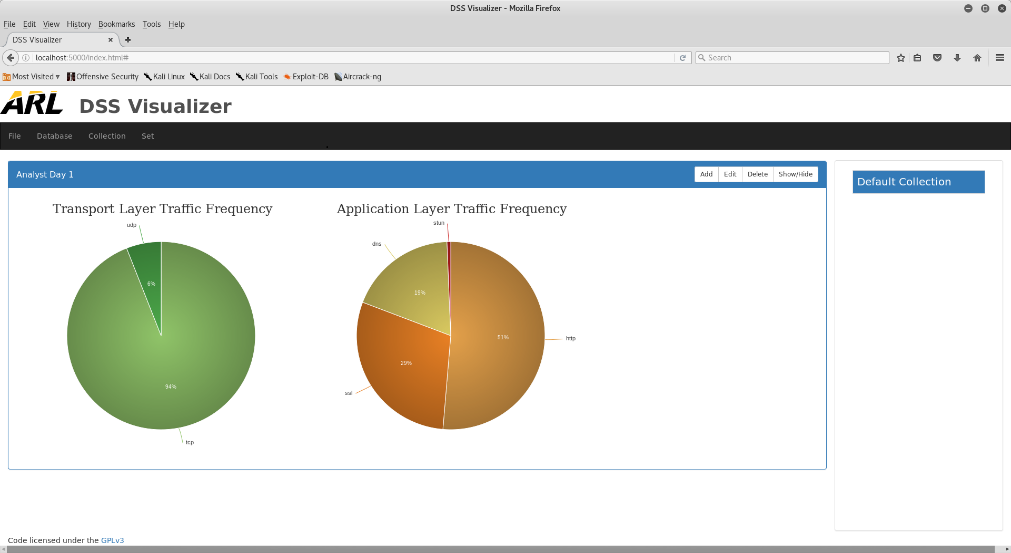


Visualizations give the analyst a way of presenting ECEL data in an easy to understand manner. By clicking the add button on a set, a new visualization can be added. Multiple visualizations can be added to the same set. Supported visualizations are based on which renderer plugins are available. Please see screenshots 4 and 5 for more details:

Screenshot 4: Add visualization



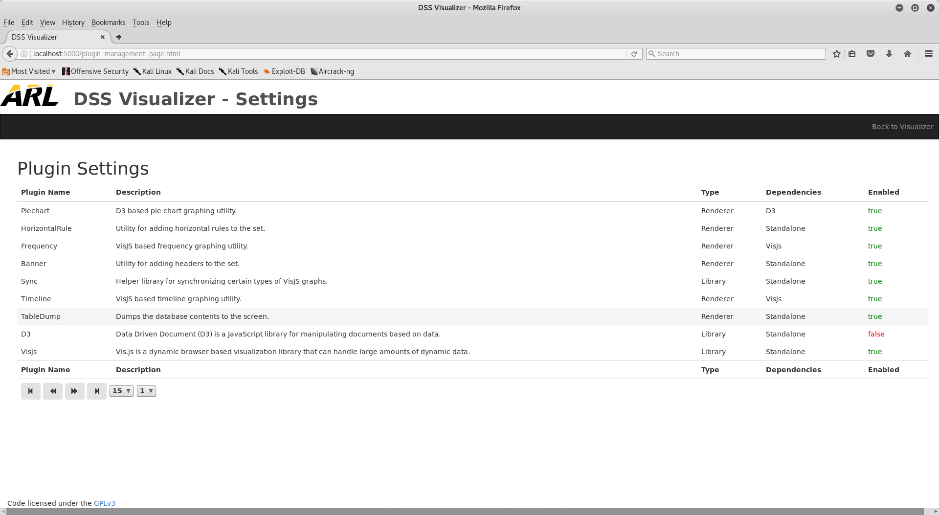
Screenshot 5: New visualization added to set



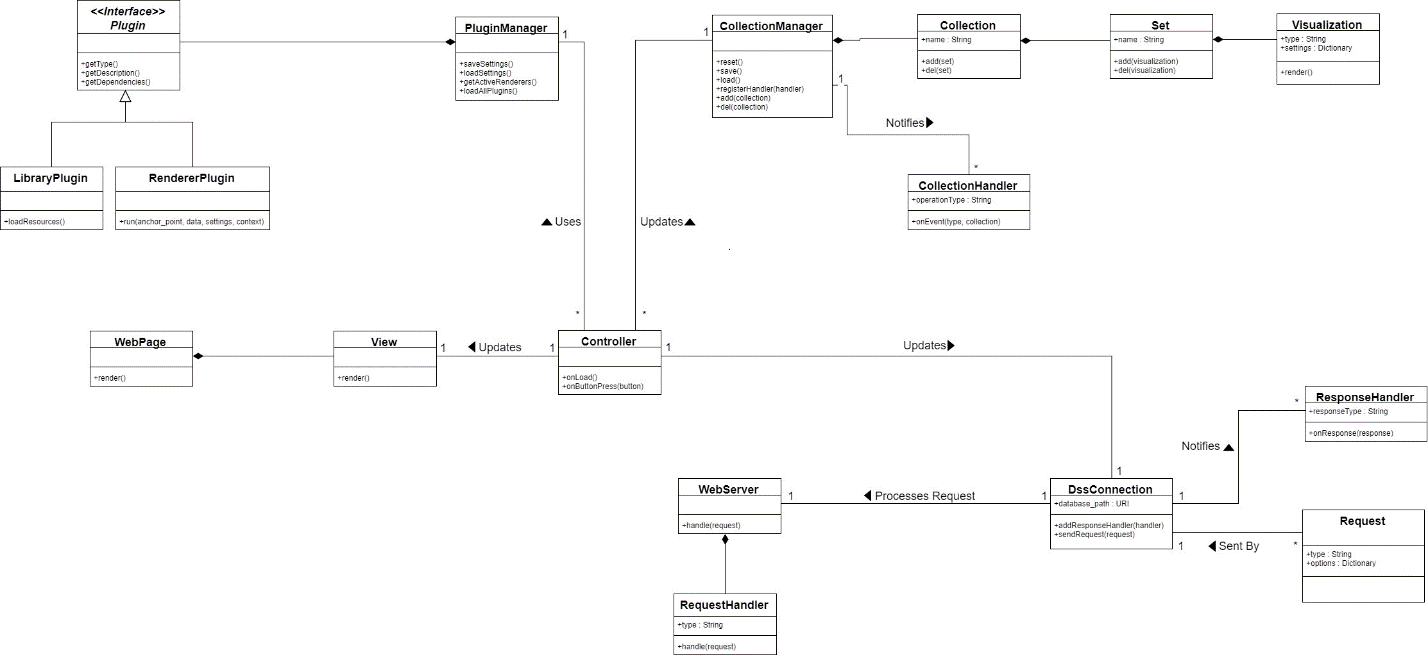
Collections allow for a logical separation between visualizations. By clicking Collection > New, a new collection will be generated. The collection selector on the right hand side will update and show that the new collection is available. A selected collection will be displayed in blue.

Some plugins are more resource intensive than others. Plugins can be enabled or disabled by clicking File > Settings and manually toggling each setting based on user preference. Please see screenshot 6 for more details:

Screenshot 6: Plugin Management



## Overview of the System Architecture.



*Diagram 1: UML Diagram of the System*

The system uses a web server and therefore uses a client server architecture. The client and the server have very distinct responsibilities. The client is responsible with data presentation while the server is responsible for storing and retrieving ECEL data.

The client consists of all the code used by the web browser. This includes static markup such as HTML and CSS. WebPages and Views are composed of this static content. JavaScript files (js) define the dynamic behavior of the web site and are referenced by each WebPage and View. The PluginManager, CollectionManager, DssConnection, and Controllers define the dynamic behavior of the web client. It is a responsibility of the WebServer to provide these scripts to the web browser.

The web server is responsible for storing and retrieving ECEL data. The web server is composed of RequestHandlers. These handlers respond to specific types of requests and either retrieve or store data based on the type of request received. In turn, the DssConnection object uses ResponseHandlers in order to update the presentation based on data retrieved by the web server. Please see table 1 for a description of each module defined in the UML view. Please refer to table 2 for a list of all requests supported by the web server.

*Table 1: Module Descriptions*

|  |  |
| --- | --- |
| Name | Description |
| Plugin | A module used by the system for defining new types of system behavior. There are currently two types of plugins: Library and Renderer. |
| RendererPlugin | A module which defines how to draw a new type of graph or visualization aid to the screen. A renderer may require a specific library in order to function. |
| LibraryPlugin | These types of plugins are responsible for loading dependencies that may be required for other plugins to function properly. |
| PluginManager | Loads plugins from memory as well as manages the state (enabled/disabled) of each individual plugin. |
| WebPage | A page on the web site that renders content to the user. A web page is composed of different views, which allow the user to interact with the system in different ways. There are currently two web pages, the index and the plugin management page. |
| View | A view is a standalone module that allows the user to interact with the system. Each view has a unique controller that manages the behavior of that particular view. Views included in the system include: AddDatabaseView, AddVisualizationView, AddVisualizationSetView, CollectionManagementView, CollectionSelectorView, DeleteDatabaseView, EditVisualizationView, FileUploadView, MenuBarView, and SetView. |
| Controller | Defines the dynamic behavior of a particular view. Controllers are permitted to use the CollectionManager, PluginManager, and DssConnection in order to fulfill the responsibility of the controller. There is a 1 to 1 relationship between controllers and views. |
| CollectionManager | Stores and retrieves collections that are stored in memory. |
| Collection | A collection is a cohesive group of sets. A collection allows for storing and retrieving sets related to the collection. |
| Set | A cohesive group of visualizations. A set allows for storing and retrieving visualizations related to the set. |
| Visualization | Renders graphs and other visualization aides. Each visualization is based on a specific type of RendererPlugin. |
| CollectionHandler | Event handler that is used in order to update the client presentation whenever a collection is modified. |
| DssConnection | Facilitates communication between the browser and client using request objects and response handlers. |
| Request | An encapsulated HTTP POST request used by the DssConnection class in order to facilitate communication between the server and client. |
| ResponseHandler | Used by the browser in order to update the presentation upon receiving a response from the web server. |
| WebServer | Responsible for storing and retrieving ECEL data from the database. Uses RequestHandler objects in order to process incoming HTTP POST requests based on user input. |
| RequestHandler | Used by the WebServer in order to process specific types of HTTP POST requests. |

*Table 2: Supported Requests*

|  |  |  |
| --- | --- | --- |
| Request | Type | Description |
| AddAnnotationRequest | DSS\_ADD\_ANNOTATION | Adds an annotation datapoint to the specified table. |
| AddDatabaseRequest | DSS\_ADD\_DB | Adds a new database instance to the current database management system. |
| DeleteDatabaseRequest | DSS\_RM\_DB | Removes a database instance from the current database management system |
| GetAllPluginsRequest | DSS\_GET\_PLUGINS | Retrieves all plugins that are stored on the web server. |
| GetDatabasesRequest | DSS\_LS\_DB | Retrieves the names of all usable databases stored in the database management system. |
| GetDataRequest | DSS\_GET\_DATA | Retrieves data from all ECEL tables based on the specified query. |
| GetEventsRequest | DSS\_GET\_EVENTS | Retrieves user defined events that are stored in the database. This contract is implemented but is not currently used by the system. |
| GetTechniciansRequest | DSS\_GET\_TECHS | Retrieves user defined technicians that are stored in the database. This contract is implemented but is not currently used by the system. |
| RemoveElementRequest | DSS\_REMOVE\_ELEMENT | Removes a data point from a specified ECEL table based on object ID. |
| UploadFileRequest | DSS\_UPLOAD\_FILE | Receives, processes, and stores ECEL data in a specified database. |

## Plugin Overview.

Plugins are used in order to define new system behavior. Currently, two types of plugins are supported: Library and Renderer. The library plugins are used in order to load dependency data. This typically includes visualization library scripts as well as stylesheets and other required resources. Renderer plugins provide a way to visualize ECEL data stored on the web server. These can include visualizations such as Frequency charts, Timeline graphs, and Pie charts. The difference between a visualization and a renderer is that a renderer is a type of plugin that creates visualizations. A visualization, by contrast, is a graphic that was produced by a renderer plugin. All currently available plugins can be found by going to File > Settings.

## How to install new plugins.

In order for the web server to load a plugin, the plugin must be placed in the plugins folder of the webclient. In addition, the plugin name must match the following regular expression: plugin\_[a-zA-Z0-9\_]+.js. Furthermore, the name of the plugin class must match the name of the file. Understores do not count towards this and the first letter of each word must be capitalized. For instance, if the plugin’s name is plugin\_line\_chart.js, then the name of the class should be LineChart. Each plugin class must be added to the plugin namespace. A reference to the plugin namespace and be retrieved using the getPluginNamespace function. Once the namespace is retrieved, the plugin must be added to the table. See code sample 1 below:

Code Sample 1: How to add plugin to namespace (plugin\_example.js)

|  |
| --- |
| var namespace = getPluginNamespace();  if (namespace["PluginExample"] == null) {  namespace.PluginExample = function PluginExample(){  ... //Do stuff here  }  } |

Each plugin requires the following functions:

* loadDependencies() – Used by Library plugins in order to load needed dependencies.
* getType() – Function must return “Library” or “Renderer”. Indicates the plugin type.
* getDependencies() – Used by all types of plugins in order to determine what other plugins are needed in order for the current plugin to function properly.
* getDescription() – An English description used for describing what the plugin does.
* createInstance(anchor\_point, data, settings, context) – Used by renderer plugins in order to create a new visualization. The anchor\_point defines where in the DOM the renderer should add the visualization. The data parameter is the ECEL data that was retrieved from the server. The settings parameter contains user input, and the context parameter contains state information provided by the browser.
* getSettings() – Returns a dictionary that defines the settings that are available to the plugin. This is currently only used by Renderer plugins. The key indicates what the name of the variable will be while the value defines the setting type. Currently, the following types are supported:
  + Integer – A signed, whole number,
  + String – An array of characters,
  + Options(a,b,…,c) – A dropdown with each option annotated using CSV’s.
  + MultiOptions(a,b,…,c) – Same as Options but allows for multiple selections.

Below are two examples, one for a renderer plugin and one for a library plugin. Please see code samples 2 and 3 in the appendix for more details.

## How to add support for new data types.

As ECEL expands, new data types will be defined. There is currently no easy way to update the system in order to add new types. Please see the “Future Changes” category for more details. In order to add new data types to the system, the following modules need to be updated:

* UploadFileRequest.py
* FileUploadController.js

## Future changes to the system.

The following are proposed long term changes for the system:

* Improve performance with large datasets (prevent needless redrawing and reduce number of drawn nodes when density is high [frequency and timeline plugins]).
* Change the collection list view to a tree view (this will permit visualization delete, which is not currently supported).
* Change the plugin enable feature to be more user friendly (auto disable plugins when dependency is disabled).
* Export feature.
* Add new renderers as needed.
* Support different types of DMS such as elastic search (generalize database connection).

## Appendix

Code Sample 2: Simple Renderer Plugin (plugin\_horizontal\_rule.js)

|  |
| --- |
| var namespace = getPluginNamespace();  if (namespace["HorizontalRule"] == null) {  namespace.HorizontalRule = function HorizontalRule(){    this.loadDependencies = function(){  //Nothing to do here  };    this.getSettings = function() {  return {};  };    this.getType = function(){  return "Renderer";  };    this.getDependencies = function(){  return "Standalone";  };    this.getDescription = function(){  return "Utility for adding horizontal rules to the set.";  };    this.createInstance = function(anchor\_point, data, settings, context) {  anchor\_point.append($("<hr>"));  };  };  } |

Code Sample 3: Simple Library Plugin (plugin\_visjs.js)

|  |
| --- |
| var namespace = getPluginNamespace();  if (namespace["Visjs"] == null) {  namespace.Visjs = function Visjs(){    this.loadDependencies = function(){    $.holdReady(true); //Prevents document.ready from loading    //Linking in stylesheet  $('<link>')  .appendTo('head')  .attr({  type: 'text/css',  rel: 'stylesheet',  href: 'js/plugins/visjs/vis.css'  });    $.getScript('js/plugins/visjs/vis.js').done(function(){  $.holdReady(false); //Enables document.ready  });  };    this.getType = function(){  return "Library";  };    this.getDependencies = function(){  return "Standalone";  };    this.getDescription = function(){  return "Vis.js is a dynamic browser based visualization library that can handle large amounts of dynamic data.";  };    this.getSettings = function() {  return {};  };    this.createInstance = function(anchor\_point, data, settings, context) {  alert("Do not create an instance of VisJS. This function is only for renderers");  };  };  } |