How to Build a Custom Visualization in TIBCO LiveView™ Web
This document details how to extend TIBCO LiveView™ Web via custom visualizations.

23 June 2016
Version 1.0
CONTENTS

3 | INTRODUCTION
3 | INTENT
3 | CONCEPTUAL OVERVIEW
   3 | Framework
   4 | Abstraction
4 | CONCEPTUAL DETAILS
6 | RECIPE
7 | CONTRACT
   7 | Contract Naming Convention
   7 | Required Contract Implementations
   8 | Additional Contract Implementations
9 | SAMPLE CODE WALKTHROUGH
   11 | com.tibco.sb.ldm.web.plugins.sample.js
   14 | Naming Conventions
   14 | lvSampleVis.js
   16 | onSchemaSet
   16 | onData
   18 | lvSampleVisEditor.js
   19 | Editor Schema Changes (4b)
   20 | Editor Visualization Changes (4c)
21 | SERVER LOAD CONSIDERATIONS / PERFORMANCE RELATED NOTES
   21 | The number of queries running at any given time
   21 | The result set size
   21 | Buffering query returns
21 | APPENDIX
   21 | Contract Functions and Descriptions
24 | CONFIDENTIALITY
24 | CONTENT WARRANTY
INTRODUCTION
TIBCO® Live Datamart is the industry’s first live data mart for Fast Data. It provides a push-based real-time analytics solution that enables business users to analyze, anticipate, and receive alerts on key events as they occur, and act on opportunities or threats while they matter. The TIBCO Live Datamart platform consists of TIBCO LiveView™ Server, any number of TIBCO LiveView clients (TIBCO LiveView™ Desktop, TIBCO LiveView™ Web, or custom clients built against the TIBCO LiveView API set), and connectivity with more than 150 integration points.

INTENT
Intent of this guide is to provide materials (including a step by step recipe that one could follow) to help users successfully customize TIBCO LiveView Web. This document reviews one sample shipped with the product, and examines the level of encapsulation as well as the design patterns used there, in order for others to use this sample as a starting point for extending LiveView Web. In this way, this document is intended as a reusable guide for best practices in building LiveView Web applications. There is no intent to have this guide be an API reference or user guide for the API set.

Furthermore, the intent is to present a conceptual overview along with a summary outline of steps required to implement a custom visualization. We will examine each file shipped in the sample in detail and discuss how it functions to produce the custom visualization. From these, a custom visualization developer can extract and follow the necessary steps to produce their own custom visualization.

There is some attempt to give an overview of different nuances of the AngularJS (https://angularjs.org/) language; however, the intent is not to teach any of the language constructs. In order to extend LiveView Web with custom visualizations, knowledge of both JavaScript and AngularJS is required. Thus, it is assumed that the audience for this guide has those pre-requisites.

Please note: There is no expectation that this guide will address the performance of your application. There will be some mention of performance considerations; however, tuning a TIBCO Live Datamart application is not within the scope of this document. After following this guide, the reader should be able to understand how LiveView Web provides a level of abstraction and a framework. Given that framework and the recipe provided here, the reader should understand how to extend LiveView Web with a custom visualization.

CONCEPTUAL OVERVIEW
FRAMEWORK
TIBCO LiveView Web (LiveView Web) supports extensibility in a variety of ways. One way in which LiveView Web can be extended is through the creation of custom visualizations. LiveView Web provides a framework; including logins, pages, cards and a query builder. End users of LiveView Web create pages and cards on those pages. For each card, the user picks a visualization type, configures a query, then live data is fed to the visualization continuously for an end user to view. Custom visualizations written in AngularJS can be loaded into the LiveView Web framework as a plugin, then those visualizations are made available to the end user in the pull down list of visualization types (just like the types shipped with the product).
ABSTRACTION
LiveView Web provides a level of abstraction for the custom visualization developer. Namely, LiveView Web can be thought of as a service. Once LiveView Web is up and running, it has a connection to the TIBCO LiveView Server and all visualizations use that single server connection. LiveView Web also supplies a query builder / editor. A query is posed to TIBCO Live Datamart and the query results are the source of the continuous live data. As a custom visualization developer, the details involved in building a query and contacting the server to invoke the query are abstracted away. This allows you to concentrate on your custom visualization and how it will display the returned data. Furthermore, changes to the query, as well as changes to the configuration of your custom visualization are also abstracted away. All of these changes are handled by LiveView Web. When a change happens, you are notified. You then decide which of the changes interest you, and how your custom visualization will handle them.

CONCEPTUAL DETAILS
The creation of a custom visualization involves two parts: the visualization and its editor. At a high level, we will talk about those two parts along with how those parts fit into, and interact with the framework that LiveView Web provides. To complete the process, once a custom visualization is created, it must register itself with LiveView Web.

A visualization (custom or built-in) displays data. In a very strict sense, you can think of the visualization as the view (V) in a Model-View-Controller (MVC) paradigm. For the sake of this paper, the following definitions of MVC are used:

M: The model manages fundamental behaviors and data of the application (we will focus on the data)
V: The view provides the user interface element of the application. It renders the data from the model. In the context of this paper, your custom visualization defines a unique view. How you display the data from the TIBCO LiveView Server is fully contained in your implementation.
C: The controller receives user input and performs appropriate actions based on that user input.

We will look at each of these roles in the context of the LiveView Web framework and your custom visualization/custom visualization editor.

In order for the visualization to display data, it must have access to a structure that contains the information that describes both how the visualization should look (did the user configure your visualization to have a horizontal label or a tooltip?) as well as what data types will be returned by the query to be represented by your visualization. Your custom visualization must hand to LiveView Web an object to use for both these types of details. This object MUST be called “model”.

Within the model, there are two top-level attributes:

a visualization
b schema

The visualization attribute holds all configuration details for your custom visualization. The visualization has two attributes:

1 type
2 config
LiveView Web sets the type value. You define the config object based on what your visualization requires. You give it to LiveView Web. LiveView Web does not access the config attribute, rather, LiveView Web simply holds onto the config object and stores it. Configuration properties needed by your visualization should be stored in the config object; there are no restrictions on the content nor structure of this data. When your custom visualization renders the data, it can access these values when determining how to render. In this way, LiveView Web is acting as a broker to share the information between your visualization and your visualization editor.

The schema is handled by LiveView Web as the query editor is exercised by the end user. One of the abstractions provided by LiveView Web is that it handles query configuration and execution on behalf of the visualization. Each query will be described by a schema which is composed of some number of fields; each field will have an associated data type. That schema information is held in the schema attribute of the model.

Your custom visualization editor will likely need to watch for changes on each of these model attributes and decide whether to handle those changes. Here’s why: Imagine that a LiveView Web card was displaying a bar chart with the results of this query:

```
“select * from ItemsInventory”
```

The bar chart most likely has one bar for each field of the ItemsInventory table. If the end user changes the query to

```
“select Items, RemainingQuantity from ItemsInventory”
```

then the bar chart should reconfigure itself to render only 2 bars. This type of change is communicated through the schema. The resultant data set from the query will also have changed. Visualizations (custom or system provided) need to listen (via $watch) for LiveView Web to inform that a schema change has happened and then adjust accordingly. Similarly, if an end user were to use the visualization editor and change the horizontal axis label, then the visualization should be re-rendered to reflect that change. Visualization configuration changes like this will affect the values stored in the visualization’s config object. In order to react to these changes, visualizations (custom or system provided) need to listen (via $watch) for LiveView Web to inform that a visualization configuration change has occurred.

Finally, the visualization must know when new or updated data arrives, or when data previously delivered has been deleted. In order to communicate data changes, LiveView Web provides a contract object that contains callback functions that will be invoked when such changes occur. Contract details, including which callback functions must be implemented vs. which can be ignored depending on your visualization’s desired behavior, will be examined in the next section.

Thus, from the conceptual level, custom visualizations must perform these three tasks:

- Provide a custom editor that watches for changes to the visualization and schema attributes of the model (and then handle them appropriately)
- Implement the contract (to be notified of query data changes so the visualization can re-render new or updated data from the TIBCO LiveView Server)
- Register themselves as a plugin with LiveView Web
The remainder of this paper will describe in detail the contract details, how a custom visualization can register as a plugin to LiveView Web, then once registered, how a custom visualization interacts with LiveView Web to display data according to the end user’s query and configuration options.

**RECIPE**

Given the conceptual introduction, the following is a condensed summary of the steps needed to implement a custom visualization within LiveView Web. The following sections will go into detail on the contract functions and then the code walkthrough will explicitly map the shipped sample code to the steps in this recipe.

1. Create plugin.json
   - a. Specify the main JavaScript file
   - b. Specify the custom visualization source file
   - c. Specify the custom visualization editor source file

2. Create the main JavaScript file; within that file, declare the Angular module and call to register your plugin with LiveView Web
   - a. Include com.tibco.sb.ldm as a module dependency to get access to TIBCO LiveView Web
   - b. In the registration:
     - i. Name your visualization type name
     - ii. Specify your visualization source file
     - iii. Specify your visualization editor’s source file
     - iv. Specify the name to display for your visualization
     - v. Provide a default configuration object. It is in this object that your visualization will have a “connection” to the dataset (the query)
     - vi. Provide a custom icon to be used for end users to see and select

3. In your custom visualization source file, add implementations
   - a. Handle onSchemaSet to know what data types are being returned from the query
   - b. Handle onData to receive continuous updates from the TIBCO LiveView Server
   - c. Handle onDataSourceChange to update the visualization relative to the data being displayed in the visualization (most commonly due to a change in the query)
   - d. Handle onConfigChanged to update the appearance of the visualization (example: custom vis editor used to change labels)
   - e. Call subscribeToData to begin to get data from a query
   - f. Implement the remaining parts of the contract (optional)

4. In your custom visualization editor’s source file
   - a. Include a binding to “model” in the editor directive’s scope configuration so you can access and make changes to your visualization configuration
   - b. Watch for changes to the schema and optionally handle them
     (via: `$scope.$watch('model.schemas', handleSchemasChange, true);`)
   - c. Watch for changes to the visualization and optionally handle them
     (via: `$scope.$watch('model.visualization', handleVisChange, true);`
CONTRACT
LiveView Web provides a contract that functions as an interface between your custom visualization and the abstracted connection to the TIBCO LiveView Server. All visualizations must implement the contract provided by LiveView Web. JavaScript does not have the concept of an interface like the Java language does; however, conceptually, the contract provides this functionality. Namely, the contract names the functions that should be implemented by the custom visualization in order to operate in the LiveView Web framework. As mentioned, LiveView Web frees the custom visualization developer from needing to know what type of query is being fired, or whether it has parameters etc. LiveView Web will call the appropriate contract function when things change. For instance, you will be given the data as the continuous query returns new results via the onData contract function. Similarly, you will be given a schema object that represents the schema of the running query via the onSchemaSet callback function. If the query changes to return different items, onSchemaSet will again be invoked to communicate that change.

The contract contains the following functions. The bolded ones must be implemented or called. Unlike a java interface, a custom card can refuse to provide any given function listed in the contract. In addition to the required implementations, it is recommended you also implement the functions denoted with an asterisk. The functions in the contract are:

- **onConfigChanged**
- **subscribeToData**
- **onDataSourceChange**
- **handleVizError**
- **onSchemaSet**
- **handleDataSelected**
- **onData**
- **showDataSensitiveMenu**
- **onError**
- **handleConfigChanged**

We will see each of these contract functions’ implementations in the code walk through portion of this paper.

CONTRACT NAMING CONVENTION
The LiveView Web contract has a naming convention. That naming convention is: function names that start with “on” are invoked when LiveView Web does something and needs to tell the visualization. Functions starting with “handle” are for the visualization to call when it needs to communicate something to TIBCO LiveView Web. Thus in the contract list above, the left column has functions that the visualization needs to implement as they allow the visualization to receive notification when outside things happen. The right column has the functions available to the visualization that it can use to communicate with LiveView web.

REQUIRED CONTRACT IMPLEMENTATIONS
subscribeToData:
As its name implies, when your visualization calls this function, you begin a subscription to a particular subset of the data in the LiveView server. This means you are registering your visualization to receive data from a specific query posed by an end user in the LiveView Web framework. Once subscribed, all data updates (additions, changes and deletions) will be sent to your visualization.
onData:
This function is called by LiveView Web to provide buffered data from the data source. NOTE: We highly recommend you leave buffering enabled. This function is called two times a second to update the visualization with data from the TIBCO Live Datamart server. You can read more about why we recommend buffering in the performance considerations section.

onSchemaSet:
This function is called by LiveView Web to provide the schema of a data source. When the user either enters a query in the query editor or uses query builder widgets to build one, the fields returned by that query are communicated to your custom visualization when LiveView Web calls this function.

ADDITIONAL CONTRACT IMPLEMENTATIONS

onDataSourceChange:
This function is called by LiveView Web each time the user hits “save” on a query in the query builder. Your visualization should know that the query has changed and you may wish to clear out the previous settings before it receives the new schema definition for the new query (via onSchemaSet).

onConfigChanged:
Called by LiveView Web when the visualization’s configuration has changed (most notably during an editing cycle from your custom visualization editor). The configuration changes from the visualization editor are written to the model. Therefore, if the user changes the horizontal label, that change will be in the model that is bound to the view. However, it is the actual visualization implementation code that knows how to render the visualization based on the values in the model. Thus, when an edit occurs to the configuration, this callback fires to inform the visualization that it should consider rendering the visualization with the new settings.

onError:
This function is called by LiveView Web when there is an error in the framework, or possibly from the underlying LiveView Server. One example is an invalid query. Despite the client side query parsing and indicator of the validity of the query, the end user could hit save and thus pass a query that is invalid. When LiveView Web attempts to then use that query, an error would be thrown by the LiveView Server and your custom visualization would then receive this callback.

handleDataSelected:
This function should be implemented if you ever want or need to communicate to LiveView Web information about selected data from within your visualization (to trigger changes in linked cards for example). If a user clicks in your visualization, what do you expect to happen? If you want or need LiveView Web to take action based on that click, then you must implement this function. This functionality is implemented in multiple visualizations shipped with the product to be able to have chart linking. For more information on chart linking, see the chart linking video tutorial in this collection:
https://vimeopro.com/streambase/liveview-web-tutorials/

handleVizError:
Provided by LiveView Web for visualization error handling in order to provide a uniform experience across all visualizations in case of errors while processing. This allows delegation of error handling to LiveView Web rather than each visualization handling errors on their own and in their own, unique way.
showDataSensitiveMenu:
Delegates the display of a contextual menu to LiveView Web. For example, in the
LiveView Web shipped visualizations, a right click on a data point invokes the menu
to allow for activation of or configuring of actions.

handleConfigChanged:
Visualizations can be interactive. The LiveView Web grid is an example. An
end user can resize grid columns via a direct interaction with the visualization
vs. being in the editor for the visualization. When this situation presents
itself, handleConfigChanged provides a mechanism for the visualization to
communicate those visualization changes. Then the changes can be stored
into the model so that the changes can be persisted. Contrast this to when the
changes happen in the configuration editor itself (onConfigChanged). In the
scenario when changes happen in the editor, the model is tied directly to the view.
Therefore a change of a property in the editor is immediately reflected in the
model. In this situation, we do not have direct access to the model. Therefore we
have this API to call back to LiveView Web to ask it to handle the change to the
configuration (store the updated properties in the model).

SAMPLE CODE WALKTHROUGH
This portion of the paper will walk through the LiveView Web sample shipped
with the product. Once LiveView Web is installed, the sample project exists in
your installation directory by the name of lvweb-samples.zip. We recommend
you use TIBCO StreamBase® Studio to import this project. If you have not done so
already, you will also need to drag and drop the lvweb.war file from the LiveView
Web installation folder into the lv-user-webapps subfolder of the imported
sample’s project folder. Once you have imported the sample project, the sample
plug-in visualization code can be found in the folder:

<project>/lv-web/plugins/com.tibco.sb.ldm.web.plugins.sample
The folder contains the following files:
1. META-INF/plugin.json - describes the plugin to LiveView Web
2. com.tibco.sb.ldm.web.plugins.sample.js - The script that handles AngularJS module definition and registration with LiveView Web
3. lvSampleVis.js - The directive which renders the custom visualization
4. sampleVis.tpl.html - The HTML template which renders the custom visualization directive
5. lvSampleVisEditor.js - The directive which renders the editor for the custom visualization
6. sampleVisEditor.tpl.html - The HTML template which renders the custom visualization editor directive
7. css/com.tibco.sb.ldm.web.plugins.sample.css - The CSS which is used by the custom visualization

Of these files, only plugin.json must be named as shown. In this sample, our custom visualization is named “sample”, thus our files are all named with “Sample” embedded somewhere in the file name. You can name your custom visualization files as you see fit.

When LiveView Web starts up, it looks for META-INF directories, and within them, for plugin.json files. This is the method LiveView Web uses to find and load plugins. We will begin our code walkthrough with the plugin.json file. This is step 1 of the recipe (Create plugin.json).

```json
plugin.json
{
    "name": "com.tibco.sb.ldm.web.plugins.sample",
    "module": {
        "core": ["com.tibco.sb.ldm.web.plugins.sample.js"],
        "addons": [
            "lvSampleVis.js",
            "lvSampleVisEditor.js"
        ]
    },
    "staticContents": [{
        "core": ["css/com.tibco.sb.ldm.web.plugins.sample.css"]
    }],
    "dependencies": []
}
```

**NOTE:**
If the name is not unique across the system, the last loaded plugin by this name is seen. This newly loaded plugin by this name overrides all previously similarly named plugins.

"name" - The plugin name. This name should be unique across the entire system. Using java package name style is encouraged.
**BEST PRACTICE:**
Within the file named by “core”, there will be a module name which is passed to Angular to declare a module. It is best practice for the module name (in the named source file) to match the name given here in "name" of plugin.json. Then, for consistency’s sake, the source file name follows the same naming convention. Thus, our sample is `com.tibco.sb.ldm.web.plugins.sample` and our filename is `com.tibco.sb.ldm.web.plugins.sample.js`.

**“module.core”:**
the main JavaScript file which defines your module. If your plugin is a single file plugin or you are concatenating your source files, then this will be the only source file defined. The path of the source file is as it exists in the jar file (e.g. if the file is stored as “my.module.jar/module_definition.js”, then the value should be “module_definition.js”). Notice the name listed as the core for the module: `com.tibco.sb.ldm.web.plugins.sample.js`. We will look at this file and examine its contents and structure next. This core specification is step 1a of the recipe.

**“module.addons”:**
All the remaining source files for your plugin module. This needs to be specified if your plugin is spread out across multiple files and you are not concatenating them. The path of each of the files is as it exists in the jar file (e.g. if the file is stored as “my.module.jar/module_service.js”, then the value should be “module_service.js”). Items designated as “addons” are loaded in parallel. Thus, if one must be loaded prior to another, then that should be moved to the dependencies. For our sample, the implementation details for the actual visualization reside in the `lvSampleVis` js file and the implementation details for our custom visualization editor reside in `lvSampleVisEditor.js`. Thus, these files are named as addons. These two lines comprise steps 1b and 1c of the recipe.

**“staticContents”:**
All the static content that the plugin depends on. This excludes templates loaded via directives. These would typically be CSS files required by the plugin or CSS files required by the libraries used by the plugin or static data needed by the plugin for proper functioning. If a file is internal to the plugin, then its path is as it exists in the jar file (e.g. if the file is stored as “my.module.jar/css/lvweb.plugins.example.css”, then the value should be “css/lvweb.plugins.example.css”

**“dependencies”:**
The items listed as “dependencies” are loaded first, so your component can run when loaded. The items specified in “core” are loaded in sequence before any “addons”. Core dependencies are loaded in the order specified in the “core” array; therefore, if our plugin requires `cubism.v1.js` which in turn requires `d3.js`, then dependency section should be formatted as follows:
```
“dependencies”: [
    “core”: [
        “lib/d3.js”,
        “lib/cubism.v1.js”
    ]
]
```

`com.tibco.sb.ldm.web.plugins.sample.js`
Step 2 of the recipe is to “Create the main JavaScript file and call to register your plugin with LiveView Web”. The first step of the recipe outlined above was to create the plugin.json. In the plugin.json file, step 1a, is to name the main JavaScript file for your visualization as the “core”. In our sample, this file is `com.tibco.sb.ldm.web.plugins.sample.js`. 
In the main JavaScript file, you declare the Angular module. In this declaration, you need to pass in the name as provided in the “name” of Step 1’s plugin.json file. You must also pass in com.tibco.sb.ldm so as to get access to LiveView Web (step 2a). You must then register your custom visualization. One of the modules that LiveView Web exposes for use is a service called PluginRegistry. In order for any card to be plugged into the LiveView Web environment, the card must register via this service (step 2b).

Let’s examine this file to see the invocation of each of the parts of Step 2. The following is the com.tibco.sb.ldm.web.plugins.sample.js file:

```javascript
;(function(angular){
  'use strict';

  angular
      .module('com.tibco.sb.ldm.web.plugins.sample', ['com.tibco.sb.ldm'])
      .run(onModuleRun);

  onModuleRun.$inject = ['PluginRegistry'];
  Function onModuleRun(PluginRegistry){
    PluginRegistry.register(
        new PluginRegistry.VisualizationPlugin(
            'sample',
            'lv-sample-vis',
            'lv-sample-vis-editor',
            {
              Name: 'Sample Vis', //Limit length to 12 characters
              defaultConfig: {
                dataSourceId: '%datasetId%',
                availableFields: {}
              },
              previewIconUrl: 'assets/img/button/vis-type/grid_60x60.png' // TODO: change this to an icon in your plug-in
            })
      );
  }
})(angular);
```

As mentioned earlier, the module name provided (com.tibco.sb.ldm.web.plugins.sample) should match the name given in the “core” of the module in the plugin.json file. This is not a requirement; however it is best practice.

Next, notice the second parameter passed to angular.module: [com.tibco.sb.ldb]. This designation signifies that TIBCO LiveView Web is a dependency for the plugin module and exposes the LiveView Web modules so they can be available to use in the custom visualization. This is an encapsulation of LiveView Web and is a dependency for the visualization you are building. Your custom visualization needs this. Inclusion of this dependency is step 2a.
The six steps in 2-b are in the call to register your custom visualization with LiveView Web. Registration with the service requires these things:

1. Name of visualization (step 2-b-i)
2. Name of directive to draw the visualization (step 2-b-ii)
3. Directive to represent the editor for the custom visualization (step 2-b-iii)
4. Name to display for the visualization (step 2-b-iv)
5. Default configuration (step 2-b-v)
6. Icon for use (step 2-b-vi)

```javascript
PluginRegistry.register(
    new PluginRegistry.VisualizationPlugin(
        'sample',
        'lv-sample-vis',
        'lv-sample-vis-editor',
        {
            Name: 'Sample Vis', //Limit length to 12 characters
            defaultConfig: {
                dataSourceId: '%datasetId%',
                availableFields: {}
            },
            previewIconUrl: 'assets/img/button/vis-type/grid_60x60.png' // TODO: change this to an icon in your plug-in
        }
    ));
```

1. The type name of this custom visualization is “sample”. This type name must be unique throughout your LiveView Web environment. This type name is what LiveView Web shows in the “Visualizations” drop down menu. No spaces are allowed in the type name.

2. The name of the directive that will render the visualization is “lv-sample-vis”. See the note in the section below about naming conventions.

3. The name of the directive that represents the editor for the custom visualization is “lv-sample-vis-editor”. As with the rendering directive, see the naming convention note below.

4. The name to display for the visualization, “Sample Vis”. Note this string is limited to 12 characters.

5. The default configuration object that represents what we refer to as a visualization’s config object. LiveView Web will take this config object and perform variable substitution. A datasetId will be generated in TIBCO LiveView Web when new instances of the custom visualization are created and that datasetId will be substituted for the datasetId template variable (seen as %datasetId%) in your default configuration object.

    NOTE: In later interactions with LiveView Web, this datasetId will be needed; however, as a custom visualization programmer, you need only declare it. LiveView Web will do the variable substitution, then later in API calls for the contract (see the code walkthrough for lvSampleVis.js) you will pass the datasetId to LiveView Web.

6. An icon to use to represent your custom visualization type in the LiveView Web visualization chooser wizard.

Step 2 (declare the module and call LiveView Web to register the plugin) is now complete.
NAMING CONVENTIONS
For those not familiar with AngularJS, it should be noted that name translation occurs automatically when defining directives. Thus, the source file that contains the implementation of the new custom visualization names the directive “lvSampleVisEditor” even though the registry is being called with “lv-sample-vis-editor”. It is common practice to refer to directives by their case-sensitive camelCase normalized name (e.g. lvSampleVisEditor). However, since HTML is case-insensitive, directives in the DOM are given lower-case forms, typically using dash-delimited attributes on DOM elements (e.g. lv-sample-vis-editor).

You will notice in the code walkthrough that there is one function that begins with a capital letter, instead of lower case. The function LvSampleVisCtrl is a constructor. We follow best-practice code styling advice of differentiation constructor functions from other functions by capitalizing the first letter of constructor functions. For more details on best-practice code styling advice, read here: https://github.com/johnpapa/angular-styleguide/blob/master/a1/README.md#style-y123

lvSampleVis.js
Step 3 of the recipe is to “implement the custom visualization”. The implementation code is contained in the file named in the 2nd parameter to PlugInRegistry.register. For our sample that file is lvSampleVis.js. Within this file, steps 3a through 3e must be completed:

3a: handle onSchemaSet
3b: handle onData
3c: handle onDataSourceChange
3d: handle onConfigChanged
3e: call subscribeToData
3f: optionally implement remaining parts of the contract

lvSampleVis.js contains the bulk of the implementation details for the custom visualization. The majority of this file is devoted to the contract’s callback function implementations. To aid with structured encapsulation, the contract’s callback functions are configured within the directive’s controller function (named as the “controller” property in the returned directive object):

```
return {
    restrict: 'A',
    templateUrl: 'plugins/com.tibco.sb.ldm.web.plugins.sample/sampleVis.tpl.html',
    scope: false,
    controller: LvSampleVisCtrl,
    controllerAs: 'vm',  // the controller instance will be referred to as “vm” in the sampleVis.tpl.html file
    bindToController: true  //scope values will be bound to the controller instance
};
```

Recall the MVC discussion earlier in the paper and the definition of the controller:

C: The controller receives user input and performs appropriate actions based on that user input
The “LvSampleVisCtrl” function is the C in the MVC implementation. It is this function that assigns a specific handler function for each of the contract functions and initiates the visualization’s subscription to the specified dataset. As such, it is controlling what happens in your custom visualization.

```javascript
function LvSampleVisCtrl($scope){
    Conceptually this controller function need only to implement some of the contract functions to be able to receive data via LiveView Web. For each of the card-to-visualization contract functions (i.e. those prefixed with “on”), your Controller should map the contract callback function pointer to the appropriate function implementation. In the sample code, the mapping is done in these lines of code in lvSampleVis.js within LvSampleVisCtrl:

    $scope.contract.onConfigChanged = onConfigChanged;
    $scope.contract.onDataSourceChange = onDataSourceChange;
    $scope.contract.onSchemaSet = onSchemaSet;
    $scope.contract.onData = onData;

    Then, within each of those handler functions, you implement what happens within your custom visualization. For instance, if new data arrives, your visualization should update to show that new data. In early releases, these contract callbacks were required to be set up prior to the call to subscribe to receive data. As soon as the subscription happens, the callbacks will be invoked; therefore, they needed to be set up before subscribing. Setting up callbacks before subscribing is no longer enforced by LiveView Web; however it is still considered best practice.

    Query results will come back in some form. LiveView Web tells you what form via onSchemaSet. As soon as you call to subscribe to a query via subscribeToData, LiveView Web will “call you back” with the schema information for that query via onSchemaSet. Implementation of this function is step 3a.

    The next thing that happens is data arrives to the visualization from the query to which the visualization subscribed. That data arrives via onDataChange. Implementation of this function is step 3b. By default, buffering is enabled. Unlike prior versions, regardless of whether buffering is enabled or not, ONLY onDataChange is called by LiveView Web to inform the card of data changes.

    All data changes are communicated through the singular contract of onDataChange. We highly recommend you leave buffering enabled. Buffering basically conflates the data for you. Imagine a data source that is changing very rapidly. If your custom visualization was notified tens or hundreds of times per second of updates on the data, and you attempted to render your visualization with each update, that may well result in a poor visual experience. The human eye has a physical limit on how much data change it is able to see. Thus, attempting to refresh any chart more often than the rate the eye can detect is wasted. Furthermore, re-rendering with each change may require a great number of resources and thus impact the performance of your application. For these reasons, the default behavior in LiveView Web is buffering. LiveView Web will call onDataChange two times per second even when the data changes more rapidly than that. The dataStore parameter contains a coalesced data set for the data source for the buffered time window. The dataEvents parameter contains ALL data events which occurred in that time window. You can choose whether to use the final value or to attempt to render all the changes. If any data point triggered some kind of condition worth acting on, even with buffering, you will have the data delivered.

NOTE:
Prior to v1.1 the API for receiving notifications of data changes was called onDataset. Furthermore, when buffering was disabled, data changes were communicated via the trio of data update functions: onDataChange, onDataRemoved, onDataUpdated. The individual data notification functions are not invoked in v1.1. Thus if you have code in such an implementation, that code will not be executed.
onSchemaSet
As stated above, after you call to subscribe to a query via subscribeToData, LiveView Web will “call you back” with the schema information for that query via onSchemaSet. In the sample, the following code loops through each item returned from onSchemaSet and adds the fields in the schema to the view model. We also call the formatter service so that the default formatting of both double and timestamp data types is readable. You need not call the formatter, or you can call it with different options.

```javascript
angular.forEach(queryEventData.schema.fields, function(schemaField){
  var vmField = vm.fields[schemaField.name] || //use the exiting field if it exists,
  else make one
  {
    name: schemaField.name,
    type: schemaField.type,
    hide: fieldConfigs[schemaField.name] ? fieldConfigs[schemaField.name].
    enabled !== true : false
  }
  if(schemaField.type === 'double'){
    vmField.formatter = FormatterService.compile('${'+vmField.
    name+'|number:2}', queryEventData.schema);
  }
  else if(schemaField.type === 'timestamp'){
    vmField.formatter = FormatterService.compile('${'+vmField.
    name+'|date:HH:mm:ss:sss}', queryEventData.schema);
  }
  vm.fields[vmField.name] = vmField;
});
```

onData
Based on the setting for the buffering interval, onData is called to deliver updated data from the continuous query to your visualization. As stated above, the dataStore parameter contains a coalesced data set for the data source for the buffered time window. The dataEvents parameter contains ALL data events which occurred in that time window. You can choose whether to use the final value or to attempt to render all the changes. In our sample, we loop through all the events, examine the change type, and call a function to handle the change type (addition of new data, deletion, or update):

```javascript
for(i = 0; i < dataEvents.length; i++){
  if(dataEvents[i].type === VisualizationContract.DataEvent.Types.ADD){
    handleNewData(dataEvents[i].data);
  }
  else if(dataEvents[i].type === VisualizationContract.DataEvent.Types.UPDATE){
    //On updates, we don’t want to waste resources by updating unchanged values.
  }
}
The oldValues
  // property of the EventData’s affectedPositions object is a fieldName->value
map of only those
  // fields whose value changed.
  handleUpdateData(dataEvents[i].data, dataEvents[i].affectedPositions.
oldValues);
}  
else if(dataEvents[i].type === VisualizationContract.DataEvent.Types.DELETE){  
  handleDeletedData(dataEvents[i].data);
}
}

Then, in each of the handle<Action>Data functions in lvSampleVis.js, the fields
from the Live Datamart Server are updated to the local view model. In the sample,
it is that local view model (vm) that is used to drive the rendering of the custom
visualization. We can see that here, in handleUpdateData:

function handleUpdateData(updateData, oldValues){
  //When a tuple is updated, we only replace the values that changed
  angular.forEach(oldValues, function(oldValue, fieldName){
      vm.fields[fieldName].value = vm.fields[fieldName].formatter ?
          vm.fields[fieldName].formatter(updateData): //
          the formatter requires the full tuple data
          updateData[fieldName];
  });
}

As stated above, your visualization may not want to loop through all the events
that occurred, but instead use the coalesced data in the dataStore. The sample
comments speak to this:

    // If we were being efficient, we would instead just set all of the
    // field values in the vm.fields map using the corresponding data
    // in the dataStore.

The dataStore has a schema field that points to the schema of the query, a map
field that is a key-value pair structure where key is the tuple.id and value is
tuple and finally a data field that is the array of the tuples in the tupleStore.
Thus, to access a tuple by id you use
dataStore.map[<some_id>] and to access a tuple by index you use
dataStore.data[<index>]
There is also a method called
indexOf(tuple)
that returns the index of the tuple in the tuple.data array or -1 if not found.
Thus, from a minimalist point of view, the only functions in the contract that
MUST be filled are:
1 onSchemaSet
2 onData
It is most likely the case that you will want to also implement:
3 onDataSourceChange
4 onConfigChanged
The implementation of these four callbacks covers steps 3a through 3d.

Finally, the card’s controller function must call subscribeToData (step 3e) and pass in the dataSourceId for the data source it is interested in receiving updates for (recall, this dataSourceId was declared in the com.tibco.sb.ldm.web.plugins.sample.js file while registering the visualization plugin). When subscribeToData is called, LiveView Web interprets that as a request to execute a query. When the server sends data, LiveView Web will notify the visualization of the data arrival from the TIBCO LiveView Server (via onSchemSet, then onData), callbacks. We see the required subscribe call in lvSampleVis.js here:

```javascript
//subscribe to dataset with default settings, (IMPORTANT - do this _after_ configuring contract callbacks)
dataUnsubscriberFunc = $scope.contract.subscribeToData($scope.visualization.config.dataSourceId);
```

This is the only contract function that is explicitly called by the visualization in the controller function. All other contract functions are implemented in this source file and may be called by LiveView Web when conditions change (examples: as data is modified, the query changes etc.) in the system.

Optionally, other contact functions may be implemented (step 3f). See the Contract section above for a description of the other contract functions including possible scenarios for when the callback may be fired in order to decide whether you want or need to implement them.

Call to subscribe (subscribeToData); handle the schema definition for the query (onSchemaSet), and then handle data arrivals (onData). You most likely also wish to handle both configuration (onConfigChanged) and DataSource changes (onDataSourceChange). Performing this set of actions completes step 3 of the recipe.

**lvSampleVisEditor.js**

Your custom visualization will most likely require a configuration editor. An editor allows the end user to configure your visualization. For instance, you may wish to allow an end user of your visualization to add a label, or change the colors used to display the data. If you want an end user of your custom visualization to have the ability to make such changes, then you need an editor. Even if your visualization has no options for configuration, it could still be altered if the end user changes the query that is feeding data to your visualization. A visualization configuration editor has a contract of sorts with LiveView Web. In v1.1 there is no official contract (as there is with the visualization); however, there are expectations about the scope and behavior of your editor that need to be followed in order to interact with LiveView Web.

Step 4 of the recipe is to “implement the custom visualization editor”. The implementation code is contained in the file named in the 3rd parameter to PluginRegistry.register. For our sample that file is lvSampleVisEditor.js. Steps 4a, 4b and 4c are implemented in this file. Recall, our steps are:

4a: Bind to model so changes to your visualization configuration will be reflected in your custom visualization editor
4b: watch for changes to the schema and possibly handle them
4c: watch for changes to the visualization and possibly handle them

Earlier we described the top level model object as having two attributes: visualization and schema. Depending on what your custom visualization editor does and what functionality it affords the end user, you may need to watch these properties for any changes that may occur.
However, it may be the case that your custom visualization affords the user no opportunity to change the layout or appearance of the visualization. In that case, you would not watch for changes on the visualization attribute of the model.

As with every card, the query editor is available to allow the user to enter (or build) a query. Step 4b allows the editor to react to a new query entered from the user via the query editor. Finally, step 4c is needed for situations like the Grid when it allows a user to resize the columns outside the editor. The shipped sample does not have such functionality; thus, to demonstrate what the custom visualization editor code should do for step 4c, code from one of the LiveView Web issued visualizations will be used.

Step 4a:

```javascript
'use strict';

angular
  .module('com.tibco.sb.ldm.web.plugins.sample')
  .directive('lvSampleVisEditor', lvSampleVisEditor);

Function lvSampleVisEditor(){
  lvSampleVisEditorCtrl.$inject = ['$scope'];

  return {
    restrict: 'A',
    templateUrl: 'plugins/com.tibco.sb.ldm.web.plugins.sample/sampleVisEditor.tpl.html',
    scope: {
      model: '='
    },
    controller: lvSampleVisEditorCtrl,
    controllerAs: 'vm',
    bindToController: true
  };
}
```

The highlighted line above from lvSampleVisEditor.js,

```
model: '='
```

fulfills step 4a of the Editor requirement. By naming the model object on the directive's scope, your visualization editor becomes tied into LiveView Web and can react to the following events:

- a End user changes the configuration properties of the visualization
- b End user changes the query being posed, which in turn changes the query's schema fields

### EDITOR SCHEMA CHANGES (4B)

The query editor provided by LiveView Web is always available on each card. Thus, the card that holds the shipped sample custom visualization is available to an end user. An end user can click on the query editor and enter or build a new query. The user can then hit 'save' and the changed query is saved to the model's schema. Your custom visualization editor needs to watch for changes to the schema attribute. To watch for schema changes, use a line of code like:

```
$scope.$watch(function(){ return vm.model.schemas; },
  handleSchemasChanged, true);
```
The custom visualization editor needs to then implement a function by the name of “handleSchemasChanged”. In that function, the editor should loop through each field returned values in the schema’s fieldsMap attribute and decide what to do with it:

`angular.forEach(vm.schema.fieldsMap...)`

As the author of your custom visualization, you should know what your visualization wants and needs for each field in a tuple returned from the TIBCO LiveView Server. Keeping with the example given earlier, if there are now five fields to display as bars in a bar chart, and before there were two, then the configuration options you present to the user in your editor need to be updated to reflect this change. Perhaps some of the existing fields must be removed. Perhaps some or all of the current fields returned in the schema need to be added to your configuration. Remember, not only are the changes being handed to you via the editor’s $watch, the visualization’s onSchemaSet contract function will also be called by LiveView Web when the schema changes. In this way, the newly changed configuration options can be handled by your visualization (most likely resulting in a re-rendering of your visualization).

**EDITOR VISUALIZATION CHANGES (4C)**

Quite similar to watching for and then handling changes to the schema, your custom visualization editor should watch for and handle changes to the visualization configuration that might happen outside the editor. When changes happen within the editor, those changes are directly saved to the visualization. However, if changes are made outside the editor, then those changes need to be captured and handled. An example of such an edit to a configuration outside the editor is a resize of column widths on the grid. This can be done by the end user while the grid is running and the editor is not invoked. To watch the visualization configuration changes, use a line of code like:

```
$scope.$watch('model.visualization', handleVisChange, true);
```

The custom visualization editor would then implement a function by the name of “handleVisChange” that takes two arguments (newValue and oldValue). In that function, the editor would want to access the config attribute of the $scope. model.visualization object and update its view accordingly. To access the pre-change config attributes, one can use the oldValue argument passed to the “handleVisChange” function.

The format of the visualization.config object (i.e. what properties it contains and what each is used for) is specific to the visualization. This configuration information is not known to LiveView Web. You, the custom visualization provider are responsible for defining the default config object when registering your visualization with LiveView Web. Remember, not only are the changes being captured via the editor’s $watch, the visualization’s onConfigChanged contract function will also be called by LiveView Web when configuration values change (note the user has to click the save button to apply configuration changes). In this way, the newly changed configuration options can be handled by your visualization (most likely resulting in a re-rendering of your visualization).
SERVER LOAD CONSIDERATIONS / PERFORMANCE RELATED NOTES
When writing a client for the TIBCO LiveView Server, consideration must be made as to the impact on the server by the application. Furthermore, a JavaScript client for a browser might be planned for a large number of clients to use simultaneously, thus care should be taken to review certain points.

THE NUMBER OF QUERIES RUNNING AT ANY GIVEN TIME
Ensure to use strategies to stop queries that are not in use by a user, for example, if using a tab-folder design, consider stopping queries that are not visible and running on other tabs, and restart the query once the user selects the tab again. In LiveView Web, queries running for cards on a particular page are canceled or inactive when that page is not currently being viewed.

THE RESULT SET SIZE
Queries can inadvertently attempt to query a very large number of rows, beyond what would be valuable to the user (for example, loading a grid with one million rows is unlikely to be of value to a human who cannot read nor comprehend that much data at once). The end user creates the query in the query builder and you have no control over the predicates they provide. Your visualization may wish to consider limiting the number of data points rendered. As with buffering, an attempt to render too much data may not be useful to the user and may consume large amounts of processing power. Consider informing the user, and/or encouraging them to alter the predicate to zoom into the data they need.

BUFFERING QUERY RETURNS
By default, buffering is enabled. As noted above, we highly recommend you leave it enabled. When enabled, LiveView Web calls your visualization two times per second with updated data. Although it is possible to replay each data change in the change set, it may require large amounts of processing time to loop through all the changes and then re-render your visualization. Furthermore, the human eye has its limits on how much change it can see in a very short period of time. For these reasons, we recommend you leave buffering enabled.

APPENDIX
CONTRACT FUNCTIONS AND DESCRIPTIONS
The following contract function descriptions are provided as a reference. These have been extracted directly out of the source code base for LiveView Web. NOTE: they are subject to change and as such, the reference in this paper should not be taken as definitive. Rather, please see the most up to date documentation for function definitions, parameters and return values.

Generally speaking, if a property begins with “on” (e.g. onData or onSchemaSet) it refers to a function that LiveView Web will invoke to communicate events to the visualization. Properties that begin with “handle” refer to functions that a visualization can invoke to communicate information back to LiveView Web.

onConfigChanged — Called by LiveView Web when the visualization’s configuration has changed. This typically occurs when users change settings in the visualization configuration editor, but it may also be triggered by other LiveView Web internal mechanisms (like card decorators). No arguments are passed to this function because the configuration object is already accessible via the visualization property that’s bound to the controller instance (i.e. vm.visualization.config).
onDataSourceChange — Called by LiveView Web when the data configuration has changed. This typically occurs when users change settings in the data configuration tab (e.g. editing a LiveView query), but it may also be triggered by other LiveView Web internal mechanisms (like card decorators).

Arguments:
- addedDataIds — The ids of any added data sources
- updatedDataIds — The ids of any updated data sources
- deletedDataIds — The ids of any deleted data sources

onSchemaSet — Called by LiveView Web to provide the schema of the data source to the visualization.

Arguments:
- dataSourceId — The id of the data source which the schema defines
- queryEventData
  - schema — The schema of the data source
- *other fields in queryEventData should be ignored

NOTE: prior to v1.1 the onData was non-existent. What existed was onDataSet. The function description below was accurate as of v1.0.2; as of v1.1 the functions onDataAdded, onDataUpdated, onDataDeleted functions are removed. ALL data changes are communicated by ONLY ONE function (onData).

onData — Called by LiveView Web to provide data events for the subscribed data source. This callback is invoked whenever an add, update, or delete event is received for the data source if buffering is not enabled. If buffering is enabled (and it is by default), then this function is called when the buffer gets flushed.

Arguments
- dataSourceId — The id of the data source whose data changed
- dataStore — Reference to coalesced data for the data source
- dataEvents — The data events. Each event has the following properties:
  - type — The type of the event. Can be one of:
    - VisualizationContract.DataEvent.Types.ADD
    - VisualizationContract.DataEvent.Types.UPDATE
    - VisualizationContract.DataEvent.Types.DELETE
  - data — The data payload of the event.
  - affectedPositions — An object containing two properties: oldIndex and newIndex. This info can be used to correlate positions between the data that’s in the dataStore and data that’s in your visualization model. On add events, newIndex is the position where the data was inserted and oldIndex will be -1. On update events, oldIndex will be the position of the data before the update was applied and newIndex will be the position of the data after the update was applied. On delete events, newIndex will be -1 and oldIndex will be the position of the data before it was removed. This information is critical for those visualizations that want to ensure consistency with ordered data sources (e.g. those resulting from LiveView queries with an ORDER BY clause).
**onError** - Called by LiveView Web when an error occurs in the data source (e.g. a query error for a LiveView query). LiveView provides a default implementation which shows a dialog. Plugin developers can override that behavior by providing their own implementation by setting the contract.onError to a function of their choosing.

*Arguments*
- **dataSourceId** - The id of the data source that produced the error
- **errorMsg** - The error message

**subscribeToData** - Subscribes a visualization to the data produced by the specified data source. The function should be invoked passing the data source ID that the visualization wants to subscribe to. The function should be invoked when the visualization is ready to receive data-related events (via the contract-defined callback functions). The function returns a function that the visualization can invoke to unsubscribe from the data source. By default, data events are buffered by LiveView Web and passed to visualizations via the contract.onData callback every 500ms. The contract.onData function is passed the data source ID of the data source that updated, a reference to a coalesced view of the data source's data, and an array of the events that occurred since the last call to onDataSet. Buffering can be switched off by setting disableBuffering to true in the optional settings object passed when invoking subscribeToData. For performance reasons, it is highly recommended that you do not disable buffering.

*Arguments*
- **dataSourceId** - The id of the data source to subscribe to
- **settings** - Additional optional settings

**handleVizError** - If an error occurs in a visualization and that visualization wants to notify the user that the error occurred in a LiveView Web-like manner, the visualization should call handleVizError to delegate error handling to LiveView Web.

*Arguments*
- **error** - The error, can be string or Error object
- **show** - Flag indicating whether or not to show the error to the user (true will show the user, false by default)
- **context** - Object which is logged as contextual information

**handleDataSelected** - In order to enable functionality like card links, visualizations need a mechanism to notify LiveView Web when a data point or data marker has been selected within the visualization. The handleDataSelected function provides this mechanism.

*Arguments:*
- **dataSourceId** - The id of the data source driving the selected data point or marker
tuple - The tuple or row associated with the selected data point or marker
categoryField - An optional field which represents the category axis value in the tuple/row
valueField - An optional field which represents the value axis value in the tuple
showDataSensitiveMenu - Delegates the display of a contextual menu to LiveView Web.

Arguments:
x - The x-coordinate of the contextual mouse click
y - The y-coordinate of the contextual mouse click
visualizationsMarkers - Array of VisualizationMarker objects where a VisualizationMarker consists of name, dataSourceId, schema, tuples, and context.

handleConfigChanged - If a visualization is interactive, allowing users to alter visualization configuration by interacting with the visualization (e.g. resizing grid columns or drawing routes on a map), handleConfigChanged provides a mechanism for the visualization to communicate configuration changes based on those user interactions. This function takes no arguments as it’s invocation just acts as a signal to LiveView Web that configuration data should be saved for the visualization.

CONFIDENTIALITY
The information in this document is subject to change without notice. This document contains information that is confidential and proprietary to TIBCO Software Inc. and may not be copied, published, or disclosed to others, or used for any purposes other than review, without written authorization of an officer of TIBCO Software Inc. Submission of this document does not represent a commitment to implement any portion of this specification in the products of the submitters.

CONTENT WARRANTY
The information in this document is subject to change without notice. THIS DOCUMENT IS PROVIDED "AS IS" AND TIBCO MAKES NO WARRANTY, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO ALL WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. TIBCO Software Inc. shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.