```
define([
    'can',
    'models/account',
    'controls/dashboard/dash
    'controls/misc/titlebar
    'toastr',
    'moment',
    'utils/helpers'
], function(can, Account,
    return can.Control.ext
        defaults: new can
            success: null
            error: null,
            username: nu
            password: nu
        }
    },{
        init: function(
            var self =
            this.eleme
            this.eleme

            App.db.ge
                App.a
                self

            App.
```

TIBCO Live Datamart JavaScript Dashboard Sample

a Best Practices Guide (v1.0)

Contents

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, receive alerts on key events as they occur, and act on opportunities or threats while they matter. The live datamart platform consists of TIBCO LiveView® Server, TIBCO StreamBase LiveView® Desktop, and connectivity with more than 150 integration points.

INTENT

The intent of this guide is to provide materials (including a step by step "recipe" that one could follow) to help users be successful in developing web apps with the JavaScript API for TIBCO Live Datamart (the LiveView JS API). The most basic building block is modular programming. More modular code is easier to maintain. Through a level of abstraction and encapsulation, you will improve the maintainability and extensibility of your application.

This document aims to explain one sample shipped with our product, the level of encapsulation as well as the design patterns used in order that others could use this sample as a starting point for his or her own LiveView JS application development. In this way, this document is intended as a reusable guide for best practices in building LiveView JS applications.

The goal of this document is to describe the shipped sample in depth and use it to outline the steps to follow to produce an application that will meet the customer's needs. By using this sample as your starting point, or by following the outlined steps, your application should be able to be maintained and extended to handle potential future requirements. This sample uses a subset of the total LiveView JS API and only those APIs used in the sample are addressed. There is no intent to have this guide be an API reference or user guide for the API set.

There is some attempt to give an overview of different features of the JavaScript language and patterns used; however, the intent is not to teach any one of the language constructs or patterns. More information on any one of the referenced topics is left to the reader (we provide suggested reading for some of the related topics).

Please note: there is no expectation that this guide will address the performance of your application. There will be some mention of considerations; however, tuning a TIBCO StreamBase® or TIBCO Live Datamart application is not within the scope of this document. Furthermore, it is not our intent to attempt to describe here how to do a similar implementation with a framework such as AngularJS or React. A sample showing the use of those techniques and/or guidelines may be forthcoming. After following this guide, the reader should be able to understand how to use the LiveView JS API in general, how to set up and execute a query and then how to see the results of that query.

OVERVIEW

The idea for this particular shipped sample is to create a dashboard. It is important to note that TIBCO Live Datamart is NOT just a dashboard. Rather, it 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, receive alerts on key events as they occur, and act on opportunities or threats while they matter. TIBCO Live Datamart enables ad hoc queries against tens of millions of live streaming records a day and includes connectivity with more than 150 integration points. TIBCO Live Datamart is built to directly connect to massive streams of data in motion, including sensors, GPS, and other Internet of Things (IoT) data sources. It supports the IoT protocols such as OSI Pi and MQ Telemetry Transport (MQTT). Pre-built connectivity to data sources includes messaging technologies based on JMS, historical data such as JDBC and MySQL, static data such as CSV files, and others.

At the heart of the LiveView server is the continuous query engine that processes high-speed streaming data, creates fully materialized live data tables, manages ad-hoc queries from StreamBase LiveView Desktop or custom user interfaces, and continuously pushes live results as conditions change in real time. This document aims to describe the best practices for how to build a custom user interface to the LiveView server using the LiveView JS API.

PATTERNS

Please keep in mind as you read through this guide for how to best design your custom user interface via this dashboard sample that TIBCO Live Datamart is much more than just a dashboard. To start our custom user interface dashboard building process, we will use the Model, View, Controller (MVC) pattern. If you are unfamiliar with MVC pattern, we recommend this page for an overview:

https://developer.chrome.com/apps/app_frameworks

For a comparison of the MVC/MVP/MVVM patterns, we recommend this reading material:

http://addyosmani.com/resources/essentialjsdesignpatterns/book/#detailmvcmvp

MODEL, VIEW, CONTROLLER OVERVIEW

There are many MVC frameworks, each with its own subtleties. In this sample we will adhere to these general principles:

1. Keep the state in the model.
2. The view is responsible to render current state.
3. The controller reacts to changes in the view, if those changes in the view require updates to the state, then the controller is responsible for that.

We are not using a framework (like Angular), although we will use Promises. A Promise overview is upcoming in the next section.

- **M:** Models – models are primarily concerned with business data.
- **V:** Views – JavaScript views are about building and maintaining a DOM element. A view typically observes a model and is notified when the model changes, allowing the view to update itself accordingly.
- **C:** Controller – an intermediary between models and views that are classically responsible for updating the model when the user manipulates the view. When users click on any elements within the view, it is not the view's responsibility to know what to do next. The view relies on a controller to make this decision for it. Often times this is achieved by adding an event listener to the view that will delegate handling the click behavior back to the controller, passing the model information along with it in case that information is needed.

PROMISES OVERVIEW

The implementation of the LiveView JavaScript API uses promises, as does this sample. Promises are an emerging standard. The promise interface represents a proxy for a value not yet known at the time the promise is created. When making an asynchronous call, a promise object is returned as a delegate or representative of the asynchronously returned value. At any given point in time, the promise will either be pending, fulfilled, or rejected depending upon the status of the asynchronous function for which it was created. It will act like a proxy to the actual data that the asynchronous function will eventually return. You can attach success and failure callbacks to the promise, and you can chain promises together creating a sort of synchronous flow or structure to your asynchronous program logic.

The main benefit afforded by use of promises is better structured, cleaner, easier to read, and maintain code that involves asynchronous calls. Historically, order-dependent asynchronous routines would require the use of nested callback functions. Nested callback code quickly becomes a mess that is hard to understand and harder to maintain.

The majority of the code in this sample is synchronous. As you will see in the code walkthrough section, in the sample we render PageListViews and PageViews. These actions are synchronous. Asynchronous calls occur in the interaction with the LiveView server. One example of an asynchronous call that returns a promise is the LiveView.connect function. When that function completes, the promise that was passed will resolve and any subsequent chained promises will also be resolved. The behavior is: We connect and expect a response. In this scenario, the promise works well because there is an expectation that the asynchronous function will conclude at some point in the future. Contrast that with the notion of a continuous query where you open a connection and just keep listening. In that continuous query scenario there is no notion of finishing. In that situation, rather than a promise, callbacks are more appropriate.

CHARTING PACKAGE USED

This sample uses Highcharts for the JavaScript graphs and charting. Quite a few charting libraries were evaluated when trying to decide on what to use internally. None seemed as complete as Highcharts. One key consideration was to be able to update specific points without redrawing an entire chart. Also, it should be noted that TIBCO's Highcharts license permits our clients to use Highcharts in their web applications so long as they display data coming from a LiveView server. Here are some of the others which were considered:

- jqplot (http://www.jqplot.com/)
- Rickshaw (http://code.shutterstock.com/rickshaw/)
- dc.js (https://dc-js.github.io/dc.js/)
- flotr2 (http://www.humblesoftware.com/flotr2/)
- dimple (http://dimplejs.org/)
- ECharts (https://ecomfe.github.io/echarts/doc/example-en.html)

Note that not all support the data point updating mentioned.

CSS PACKAGE USED

There are several CSS libraries that provide styles and layouts. For our sample, we use Bootstrap.

http://getbootstrap.com/examples/dashboard/

This package gives us easy to use, out of the box, stylish and attractive looking web pages. Use of Bootstraps removes from the programmer the burden of writing the bulk of CSS. You can think of this as "decorating" the pages without doing much work. It is a very clean way to get an attractive UI. If you were to try to implement the dashboard pages in this sample via plain HMTL, it would be tedious and still look fairly bland or bare bones. Bootstrap is open source.

TERMINOLOGY USED

Throughout this guide we will rely on certain terminology. This section is somewhat like a glossary but is presented here to be sure the reader understands this terminology and these concepts. The sample is more easily understood if these concepts are known.

Classes and Objects

JavaScript is a language without the concept of a class. Everything is an object. As we walk through this sample and speak to encapsulation, it may be helpful to think of a more traditional class based language. Each source file in our sample encapsulates the concept of a class. For instance, PageListView.js contains the code for what can be considered the PageListView class. Furthermore, we will have an instance of a PageListView object.

Scope, Variables, and Closure

Within the code and possibly in some of the text describing the code there will be references to "global" scope objects. When there is a line of code like:

```
var PageListView = (function($){
```

we will refer to this as a global PageListView object. The "(function($)" syntax is an immediately invoked function expression (IIFE). This defines a function and invokes it immediately. This is done to make use of closure. Variables are scoped to the function in which they are declared. If you follow this syntax, then the variables are only scoped to this function vs. being added to the window or some more general context. These variables are not needed anywhere else, so keep them local and cause their scope to be closed.

In the line of code above, the PageListView object will be assigned the result of the immediately invoked function.

Shorthand

The sample code also uses the "$" shorthand. In this immediately invoked function, a jQuery object is passed via these lines of code:

```
var PageListView = (function($){ . . .
})(jQuery);
```

Each reference to "$" within this function therefore refers to jQuery. This is another technique. It is a safe way of scoping to know that "$" doesn't refer to something else.

Prototypes

The sample code uses prototypes. JavaScript does not have classical inheritance based on classes (as most object oriented languages do), and therefore all inheritance in JavaScript is made possible through the prototype property. You can use JS without using explicit prototypes. For instance, if you call a string function, like substring, it is defined in the string's prototype. This is not necessarily seen or explicitly called out; but that is how it is implemented and how it works.

Think of an object in JS as being a set of fields with values. Prototypes give the notion of typing, and to be able to define functions to use on the objects. Any time there is an instance of this JS object, programmers can have access to a well defined set of functions that can execute. Prototypes give you this ability. They are understood in the language, whether you choose to explicitly use them or not.

It is important (and possibly somewhat confusing) to note that functions themselves are objects.

Object Prototypes:
Every object in JavaScript has a prototype. An object's prototype is a reference to an object that defines properties and functions inherited by the object. Prototype objects are hierarchically chained such that when JavaScript attempts to find a property or function on an object it first examines the object itself. If the property cannot be found, it proceeds to look for the property on the object's prototype, continuing up the prototype chain until it reaches Object.prototype. If Object.prototype is reached and the property is still not found, it gives up. This works somewhat like single parent inheritance in a language that is class based. The difference: In class-based language, there aren't object instances that are being traversed whereas in JS, prototype objects are instantiated, existing-in-memory objects that store properties and functions. This attempt to find the property by walking the inheritance path is called a prototype chain.

Function Prototypes:
A function's prototype is the object instance that will become the prototype for all objects created using this function as a constructor. You add methods and properties on the prototype when you want instances of an object to inherit those methods and properties. You can think of the prototype as having the shared or class variables. Those shared things are stored in the prototype and then the object instance itself doesn't have a copy. The function prototype is the one place in memory where functions exist for all instances. In this way, use of prototypes can save memory (vs. each instance having its own copy of each function). Possibly important to note depending on your background: Prototype functions (unlike static functions in C++) can actually manipulate properties on object instances.

There is subtle detail relative to prototypes and constructor properties. For those without a JavaScript background, it may be too detailed for this paper. It is presented here because of the programming style used in the sample. In the sample, each prototype starts with code as follows: (from PageListView.js)

```
PageListView.prototype = {
        constructor: PageListView, . . .
```

A function's prototype defines the constructor property (which points to the constructor of the function). When a programmer explicitly declares a function prototype, it overwrites what had been in place. One side effect or disadvantage of that overwrite is that the constructor property no longer points to the prototype. It is desirable to have the constructor property point to the prototype; therefore we have to set it manually. Thus the first line in the prototype function in our sample is the reset to point the constructor property back at the constructor function. This is not the only way of defining prototype functions and properties. The technique used in this sample requires the reset. (A relatively quick, good read with a pictorial explanation on this topic can be found here: http://javascript.info/tutorial/constructor.)

Function Constructors
Another topic/feature use: function constructors. Function constructors go hand in hand with prototypes. For the sake of this portion of the overview/description of the sample, we will show some code but not go into full description of the code. In our sample will have code like this:

```
var PageListView = (function($){
  'use strict';
  function PageListView(element, model, config){ . . .
```

This part of the above code sets a global variable named PageListView to point to the Immediately Invoked Function Expression's (IIFE) returned value.

```
        var PageListView = (function($){
```

It is the function declaration inside of this IIFE that is the constructor. Thus our function constructor is:

```
        function PageListView(element, model, config){ . . .
```

For this object, the function prototype is this:

```
        PageListView.prototype = {
                constructor: PageListView, . . .
```

We can have variable declarations for these objects like:

```
        var pageListView;
```

Finally, to complete this portion of description of the language features and our sample code, variable assignment via a call to the function constructor looks like this:

```
        pageListView = new PageListView($('#pageList'), appModel.pages, {});
```

The variable assignment line of code assigns to the PageListView variable the return of the call to the function constructor.

PROJECT DESCRIPTION

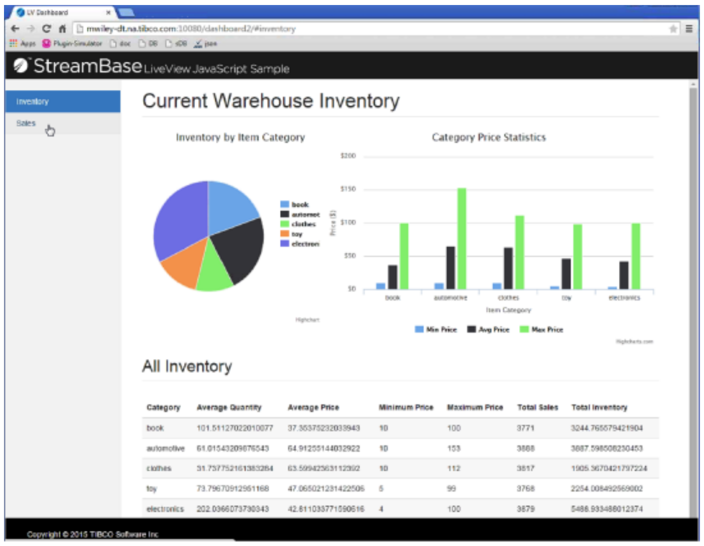
The high level overview of this sample is that we are building a dashboard. The dashboard will have multiple visualization components, each one driven by a query to TIBCO Live Datamart that uses the LiveView JS API. It is possible to issue a snapshot only query; however, we do not issue any of those here. All queries issued in this sample are continuous. To get started, we begin by thinking about the MVC pattern; specifically we start by thinking about the "M," the Model. This will be the data representation of the dashboard. There are several CSS libraries that do styles and layouts. For our sample, as mentioned above, we use bootstrap.

Our dashboard will have static content at the top (a header, "StreamBase LiveView JavaScript Sample") and at the bottom (a footer, "Copyright 2015 TIBCO Software Inc").

On the left, we will have a list of pages that will show up on the dashboard (PageList). For this sample, there are two pages "Inventory" and "Sales."

On the right are the pages or views (PageView). Each page view has a title and then some sections. Within each section, there can a title (or note) and there can be multiple components.

Thus, our dashboard will look something like this:



Category	Average Quantity	Average Price	Minimum Price	Maximum Price	Total Sales	Total Inventory
book	101.51127022010077	37.35375232033943	10	100	3771	3244.765579421904
automotive	61.01543209876543	64.91255144032922	10	153	3868	3887.598508230453
clothes	31.737752161383284	63.59942563112392	10	112	3817	1905.3670421797224
toy	73.79670912951168	47.065021231422506	5	99	3768	2254.008492569002
electronics	202.0366073730343	42.811033771590616	4	100	3879	5488.933488012374

SAMPLE CODE WALKTHROUGH

GENERAL

Before we get into the low level details of the source code, we will discuss a few general recommendations.

First recommendation is use of "use strict." This directive tells the browser, "if you encounter this thing and it's not quite right, generate an error rather than try to interpret it." This directive makes code better and less error prone. Rather than the browser trying to best guess and getting weird behavior later that you may not be able to track down, you will get an error if you "use strict." This is used throughout the sample and it is recommended that you too use it.

Second recommendation is to check the object about to be acted upon for correctness. In each source file you will see code like this:

```
//make sure this is what we expect it to be
if(this instanceof PageListView === false){
        return new PageListView(element, config);
}
```

This is the safety check. For the sake of clarity, look at these two declarations of a variable of type PageListView:

```
var a = PageListView();
var b = new PageListView();
```

In the first declaration, the "this" in the PageListView controller would refer to Window (not a PageListView instance). In the second declaration, with a call to new, we are assured of getting a PageListView instance. Thus, this check ensures we are in fact using a PageListView instance. Without such a check, it could lead to an attempt to set properties on an object that was not intended. This, in turn leads to problems that are hard to track down. To prevent such problems, we do the test in each constructor.

REQUIREMENTS

The following are required to be able to use the LV JS API:

```
<!-- Required LiveView libraries -->
<script src="/lv-web/api/lib/jquery.min.js"></script>
<script src="/lv-web/api/lib/jquery.atmosphere.min.js"></script>
<script src="/lv-web/api/liveview.min.js"></script>
```

This line is required for our use of the bootstrap layout/style package:

```
<!-- Supplemental libraries for visualizations and dashboard UI -->
<script src="lib/bootstrap.min.js"></script>
```

These lines are required for use of highcharts for our visualizations:

```
<script src="lib/highcharts.min.js"></script>
<script src="lib/highcharts-more.js"></script>
```

LIVEVIEW JS APIS

The following LV JS APIs are used in this sample:

- **LiveView.connect:** Connects to the LiveView server identified by the settings.url parameter
  - *LiveView.Connection* - Object returned upon successful connection to LiveView
- **LiveView.closeAllConnections:** Closes all currently active LiveView server connections
- **LiveView.Query:** Object that defines a LiveView query (LiveView.Connection.subscribe requires this)

Once connected, and a query object created after connection, these APIs are called:

- **LiveView.Connection.subscribe:** Performs a continuous LiveView query that will be updated with any changes in query results.
- **LiveView.Connection.unsubscribeAll:** Unsubscribes from (cancels) all active query subscriptions that were made via the connection. This is useful when switching dashboard pages rather than closing each querySubscription individually.

At the highest, most basic level, this sample connects to a LiveView server, creates and subscribes to some number of queries (associating with each callbacks depending on the query return), and then executes the queries. There are two different “pages” on the dashboard — when the user switches from one page to the other, the queries that had been running on the previous active page are cancelled or stopped (via unsubscribeAll), then the queries on the newly active page are started (via subscribe).

For each query, there are callbacks associated that direct actions based on what type of event the LiveView server is returning for the query (onSnapshotStart, onInsert, onUpdate, onDelete, onError).

SOURCE CODE FILES

- index.html
- app.model.js
- LiveViewQueryService.js
- TupleStore.js
- View/PageListView.js
- View/PageView.js
- View/TableView.js, GaugeView.js, PieChartView.js, ChartView.js (one for each visualization type)

ROLES AND RESPONSIBILITIES

Given our MVC model, the list of LiveView JS APIs, and the source code file list, let’s look at how each is related. What role does each source file fill? We will walk through each source file in detail in later sections.

The model is held within appModel.js. Here you can add components to or remove components from your dashboard without the need to change any other code. The model is defined as some (variable) number of pages, each page has some (variable) number of sections, and each section has some number of components. It is here that the visualization type for the component is defined (a pie chart vs. a table) and the query string for the component is set (for instance: ‘SELECT * FROM ItemsInventory’).

The View: there are many different views in the application. The PageView and PageListView are the views for the Page and PageList. Similarly, the GaugeView is the view for gauges, the ChartView is the view for charts, etc. In this particular application, the views don’t support any user interactions so there is no need for corresponding controllers (PageController, PageListController, ChartController, etc), although you could argue that the page-changing logic in index.html would be better-placed in a PageListController. Each view is responsible for rendering content on the page. Thus, the PageListView renders the list of pages using an HTML unordered list, the PageView renders the different sections and their components, the ChartView renders a Highcharts chart, etc. The content rendered by the view is primarily determined by the data contained in its model. In the case of the PageView and PageListView, the model is static and provided by the appModel object in app.model.js. The visualization views (ChartView, GaugeView, etc) all use their own instance of LiveQueryModel defined in LiveViewQueryService as their model. These view implementations contain the callback functions that are triggered when the query returns new/updated data (where the implementation's model [LiveQueryModel instance] does the triggering).

The Controller: This demo has very little user interaction that it has to handle. Really the only interaction supported is changing the page when a link in the PageList is clicked. Because of this, one could argue that we should have a PageListController and put all the page-changing logic there. It could also be argued that because we don't have any real controllers that this demo fits more with an MV* pattern than a strict MVC pattern. Earlier in this paper we defined a controller's role as "reacts to changes in the view, if those changes in the view require updates to the state, then the controller is responsible for that." Thus, without much user interaction, we don't have much of a controller. Within index. html there is code that handles the switch from the Sales page to the Inventory Page and vice-versa.

Closely related, though not technically a controller, we have what one could think of as a singleton class that defines the LiveQueryModel object and provides functions to manage the LiveView connection. This functionality is implemented in LiveQueryService.js. The LiveQueryService is a way to encapsulate all access to the LiveView server. It establishes and shares the connection to the server, creates and executes the queries. Without the service, a similar implementation would have a good amount of code repetition. For instance, each visualization type (chart view, gauge view, etc) would have similar/same repeated code to establish a connection, create a query, execute it, set up the callbacks. Coordination would be required to try to share a single connection. In this sample, when the user changes which page they wish to see, the queries on that page are terminated. Without the service implementation, a change of pages would require each visualization to terminate its own set of queries.

The remaining source file, TupleStore.js is used as part of the implementation; it manages a query's resultant dataset. Some visualization types require the full data set vs. just the changed data members. A continuous query returns only the changed data members as a matter of efficiency. For any given visualization that requires the full data set, we have to maintain a copy of the data locally. The TupleStore fulfills this need for a local copy of the data.

SOURCE FILE FOR MODEL

The model code is contained in appmodel.js. We have an appModel which is composed of multiple pages. In this sample there are two pages:

1 inventory
2 sales

Each page has some attributes (for instance, a name and title) and multiple sections. Each section has some number of components. One singular component is a visualization from Highcharts (a pie chart or a bar chart) and each chart is populated/updated with a query from the LiveView server. The TableView is an exception. It is not a Highcharts visualization; it just renders HTML. That said, its model and query mechanics are the same.

This code starts that definition our model.

```
var appModel = {
        pages: {
```

Each page has name, title and multiple different sections. This code starts the definition of the pages with multiple sections:

```
inventory: {
        name: ‘Inventory’,
        title: ‘Current Warehouse Inventory’,
        sections:[
```

Each section has some number of components in them. This code starts the definition of the multiple components as well as defining the first component’s visualization type:

```
components: [
        {
          query: ‘SELECT * FROM ItemsInventory’,
           visualizationType: ‘pie’,
           visualizationConfig: . . .
```

The data that drives each component is a query:

```
query: ‘SELECT * FROMItemsInventory’,
```

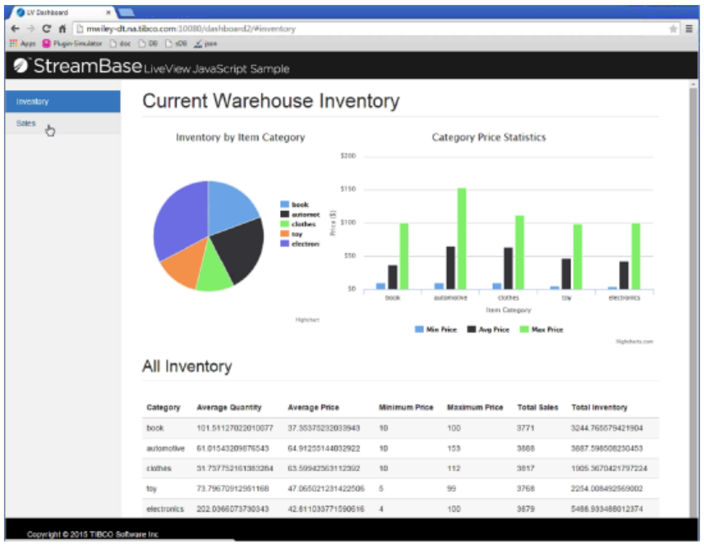
Note the bootstrap implementation uses a grid. Within the definition of a component there is a colSpan directive. This is like a width. It will be used in conjunction with the grid layout of bootstrap. Thus, the last two items in a component definition are “width” (colSpan) and height:

```
colSpan: 4,
height: 400
```

This grid, for simplicity’s sake, breaks a page into 12 columns and depending on how you resize your window, it will scale your components on that page accordingly.

SOURCE FILE FOR VIEW

In order to render the model, we have different views. This dashboard has two primary views: The PageListView and PageView. The PageListView is the left side bar “Inventory” and “Sales”. If a user selects one of those two, then the Page on the right changes to show the relevant information.



Category	Average Quantity	Average Price	Minimum Price	Maximum Price	Total Sales	Total Inventory
book	101.51127022010077	37.35375232033943	10	100	3771	3244.765579421904
automotive	61.0154320987654З	64.91255144032922	10	153	3888	3887.598508230453
clothes	31.737752161383284	63.59942363112392	10	112	3817	1905.3670421797224
toy	73.79670912951168	47.065021231422506	5	99	3768	2254.008492569002
electronics	202.0366073730343	42.811033771590616	4	100	3879	5488.933488012374

Recall from the overview (specifically the description of Promises) that rending of the views is synchronous. The PageList and Page views render synchronously so there’s no need for asynchronous handling (neither callbacks nor promises).

PageListView (View 1 of 2)
Looking at PageListView.js, we see the constructor and prototype functions. The call to the PageListView constructor exists in the source file index.html:

```
pageListView = new PageListView($(‘#pageList’), appModel.pages, {});
```

This call to the constructor passes in the DOM element in which to render our PageListView, along with the model and some configuration parameters (these are NOT currently used and may be removed!)

First thing the constructor does is the safety check to ensure it has a PageListView instance (was this code called via a “new”), per the recommended practice outlined above.

```
//make sure this is what we expect it to be
if(this instanceof PageListView === false){
        return new PageListView(element, config);
}
```

Then the constructor sets the members for the instance from the parameters.

```
_this = this;
 this.element = element;
 this.model = model;
 this.config = config || {};
```

It sets up the DOM element:

```
this.listElement = $(‘<ul class=”nav nav-sidebar”></ul>’);
```

It then populates the DOM element according to what pages are in the model (basically, this block of code looks to the model to decide what to display in the DOM):

```
for(pageId in model){
        if(model.hasOwnProperty(pageId) === false){ continue;}
        l = this.listItemElements.push(
               $(‘<li style=”cursor:pointer;”><a href=”#’ + pageId + ‘”>’
                      + model[pageId].name + ‘</a></li>’)
                      .click(
                        function(event){
                _this.listElement.children(‘li’).removeClass(‘active’);
                $(event.target.parentElement).addClass(‘active’);
                        }
                 )
        );
        this.listElement.append(this.listItemElements[l-1]);
        $(element).append(this.listElement);
}
```

This line of code:

```
$(element).append(this.listElement);
```

adds the DOM element to the DOM so it actually gets rendered.

For PageListView there is only one function:

```
selectPage: function(pageId){
```

Users of this dashboard can navigate to one of the two pages by clicking in the left sidebar on “Inventory” or “Sales,” or they can enter a specific URL. Example URL:

```
<UserMachine>:10080/dashboard2/#inventory
<UserMachine>:10080/dashboard2/#sales
```

When the navigation happens, the correct page must be displayed. The change of which page to display is accomplished through the “selectPage” function of PageListView.

PageView (View 2 of 2)
PageView (PageView.js) has the constructor and prototype functions. Within the prototype there is a call to just one immediately invoked function called setModel:

```
setModel: function(model){
```

When setModel is called the page actually renders itself.
The block of code that begins with this line:

```
Array.prototype.forEach.call(
```

clears out all the existing children (when you switch from page “a” to page “b,” you have to remove all the old info before you can display new page selection info).
Next there is a block of code that loops through each section in the model:

```
model.sections.forEach(function(sectionModel, sectionId){
```

In this loop, there is a call to create the page section div element:

```
sectionElements[sectionId] = $(‘<div class=”page-section”></div>’);
```

The info will be rendered in this div element. Then we add to that element the title (if one exists) via this code:

```
    sectionElements[sectionId].append(‘<h2 class=”sub-header”>’ +
sectionModel.title + ‘</h2>’);
```

Finally, we loop through and add each component in the section with the block of code that starts as follows:

```
sectionModel.components.forEach(function(component){
```

It is here that the visualization type is examined and then the constructor for that particular visualization is called in order to render a specific chart on the dashboard. In the first section of the inventory page, there is a pie chart, column chart, and a table (as defined in the model) and this code loops through to create each of those.

VISUALIZATIONS
For this sample, there are four types of visualizations implemented: table, pie, guage and chart. The table is pure html, the other three are implemented from Highcharts. You can build your own and/or use others from the Highcharts package. For each of the Highcharts visualizations, the options field from the visualizationConfig field of the component model object is used to pass necessary configuration options to Highcharts (this data structure:
`appModel.pageX.sections[sectionIndex].components[componentIndex].visualizationConfig.options`)

The model being used by the sample has a field value named “visualizationType.” The visualization is populated using the data coming from the subscription to the query defined in the field “query.” This type corresponds 1 to 1 with the .js source files in the views folder. Thus, within app.model.js there are visualizationType variables set to “table,” “pie,” “guage,” and “chart,” and within the views folder there are source files TableView.js, PieChartView.js, GaugeView.js and ChartView.js.

For the HighChart visualizations, the configuration of the visualization follows the standard Highchart configuration options. In particular, most if not all of the visualizations take a title, plotType, options, category, and series. It is worth noting that the pie and gauge take a specific set of config options that are distinct and separate from all other HighChart chart types:

- Pie:
  - title – string
  - legend – object that follows Highchart’s legend object schema
  - categoryField – string name of the field in the query schema to use as the category value
  - valueField – string name of the field in the query schema to use as the actual value
- Gauge:
  - title – string title
  - min – number minimum range value for the gauge
  - max – number maximum range value for the gauge
  - units – string identifying what the metric units are (“meters,” “mph,” “dollars,” etc.)
  - valueField – string name of the field in the query schema to use as value

In app.model.js a query and associated visualization are determined. Any/all configuration options for visualization are determined and placed into the visualizationConfig field. These configuration options are then passed through to HighCharts. NOTICE: with the exception of pie and gauge, many other charts from HighCharts are represented as a visualizationType of “chart.” Then, within the visualizationConfig, the plotType field determines the type of HighChart that is displayed. In this sample, there are line charts and column charts (often called a bar chart).

We’ll take a look at each of these pieces of the visualization in depth as we walk through the code in the next sections.

View/TableView (Visualizations)

Each of the four visualization source files operates the same at a conceptual level. They take the element in which they will render themselves. Based on the model that they have, they determine what they need to display. For a table, that means adding new rows as new data comes in, or updating cells, or removing rows upon delete. When a chart is used, it may need to add new points. Or for a pie chart, new slices may need to be added. Those view specific responsibilities are all handled by the view code in the specific view source file.

As stated above, the view gets its query data from the LiveQueryModel (described in detail in the next section). The LiveQueryModel executes all the callback functions registered as listeners. What this view type does with the data varies depending on the type of view.

Within TableView.js there is a block of code that sets up a listener for each type of action:

```
//Subscribe to model updates so we can update the view
if(model instanceof LiveViewQueryService.LiveQueryModel){
   model.addSchemaListener(this.handleSchemaSet, this);
   model.addInsertListener(this.handleDataAdded, this);
   model.addUpdateListener(this.handleDataUpdated, this);
   model.addDeleteListener(this.handleDataRemoved, this);
}
```

Not all visualization types listen for each of these types of events. For instance, a guageView object only cares about inserts and updates; therefore in its source file it does not call addSchemaListener nor addDeleteListener.
TableView.js example

```
        handleSchemaSet: function(schema){
           //set the header rows of the table
           //add a data row to the table
           var cells = [], titleMap = this.config.fieldNameToTitleMap || {};
           this.schema = schema;
           schema.fields.forEach(function(schemaField){
                  cells.push(‘<th>’ + (titleMap[schemaField.name] ||
schemaField.name) + ‘</th>’);
                   });
                   this.tableHead.append(‘<tr>’ + cells.join(‘’) + ‘</tr>’);
           },
```

View/ChartView (Another Visualization)
As mentioned earlier, the tableView is pure html. The pie, gauge and chart types are wrappers around a Highcharts visualization type. Here we will examine the generic ChartView.js. The required listeners are add, delete, and update of data. The key function is buildChart:

```
function buildChart(element, config){
          config = config || {};
          config.options = config.options || {};
          return new Highcharts.Chart({
                  chart: $.extend( true,
                          config.options.chart || {},
                          {
                                 type: config.plotType || ‘line’,
                                 renderTo: element,
                                 animation: Highcharts.svg
                          }
                        ),
                        title: {text: config.title},
                        xAxis: config.options.xAxis || {},
                        yAxis: config.options.yAxis || {},
                        tooltip: config.options.tooltip || {},
                        legend: config.options.legend || {},
                        plotOptions: config.options.plotOptions || {},
                        series: config.series.map(function(configSeriesItem){
                               return $.extend({data: []}, configSeriesItem);
                        })
               });
```

One thing to notice: the default chart plotType is line. If you do not specify a plot type, your dashboard will display a line chart. For all possible chart types, see

```
api.highcharts.com/highcharts#plotOptions
```

Highcharts requires a specific model/ data construct for how it displays things. Therefore, within each view source file, there are two different variables used that pertain to models. One is the "model" used locally for LiveQueryModel representation. The other is the "viewmodel" that is used to pass to Highcharts in the format it requires. Notice in the code above that a Highcharts chart is created via a call to new and then each of the options is configured. If you wanted to configure any additional options on a particular Highchart, you would have to modify this buildChart function to include the additional options. For instance, not passed is drilldown. Furthermore somewhere else in your code (most likely in app.model.js) you would have to set the options to be passed. Most charts take the options listed in ChartView.js. Pie and gauge are slightly different, and because of that, they have their own source file.

SOURCE FOR CONTROLLER

Before we delve into the details of the controller source file, a reminder that the current version of JS API uses WebSockets. All events from the LiveView server are communicated through the WebSocket. To try to keep things clean, the server will terminate all queries for a connection when it determines that the WebSocket closes. The previous implementation of the JS API did not use WebSockets. The result of the previous implementation was that a user could run any number of queries and then close the browser thinking it would kill those N queries. However, the queries would continue to run. In order to stop the queries, the JS API programmer had to manually call to cancel each query. In addition, the LiveView server imposes query limit. The net result of the need to manually cancel (which did not happen when users closed their browsers) and the server limit is that users hit the 100 query limit. We mention that here as a point of emphasis both for how the current JS API is implemented and because in this sample, any unused (not currently being viewed) queries are manually cancelled.

As mentioned earlier, the majority of the code in this sample is synchronous. Rendering PageListViews and PageViews are all synchronous. Asynchronous calls occur in the interaction with the LiveView server. With continuous queries, there is no notion of finishing; a connection is opened, per se, and then the connection just keeps listening. In that situation, we use callbacks.

Index.html

Within this file lies the logic that performs the task of an application or dashboard controller, manipulating the page and pagelist according to the URL hash. From index.html the following code controls what happens when a user clicks on a URL in the PageListView (to change from one page to another)

```
                        //listen for URL hash changes and update page and page
list views accordingly
                        $(window).on('hashchange', function(){
                                pageId = location.hash.slice(1);
                                LiveViewQueryService.cancelRunningQueries(); //
clean up queries started by the previous page
                                        pageListView.selectPage(pageId);
                                        pageView.setModel(appModel.pages[pageId]);
                                 });
```

Because the browser or the tab is not being closed, the WebSocket stays open. Therefore for each query fired on the dashboard, the query will continue running until / unless it is cancelled. In the code above the call is made to explicitly cancel the running query(s) when the click happens to cause a page change.

It should be noted that there is a tradeoff to doing this query cancellation. The tradeoff is between server resource management vs. performance. More on this tradeoff in the final section (Additional Considerations).

ADDITIONAL SOURCE FILES (NOT M, V, OR C)

LiveViewQueryService

We saw in the PageView.js an iteration through each component for a section of the dashboard. For each, we create an object called a componentModel:

```
        componentModel = new LiveViewQueryService LiveQueryModel({queryString:
component.query});
```

This componentModel is a LiveQueryModel provided by a LiveViewQueryService. It is within this service that we encapsulate all the LiveView JS API calls. We are delegating all the LiveView related activities to a service. Angular has a specific service type of contract. We are not using a framework, so we wrote our own type of service in the liveviewQueryService code. In general, this service helps maintain separation of sense of responsibilities.

Other source code in the sample returns a constructor function, but for the service, we return an object with a few different functions in it. The definition of a service is providing multiple functions. This LiveQueryService gives access to the LiveQueryModel. The main purpose of query service: maintain one connection to the LiveView server. Any component that wishes to issue a query, uses / shares this one connection. If you do not share a connection, you will consume slightly more memory.

Additionally, in this service you could inspect incoming queries to determine if the incoming query is a repeat. Rather than re-issue identical queries, the components could share results. This would be an optimization. You could have one model per unique query string. Each visualization shares a query, and adds its own listeners to the shared LiveQueryModel for that query. When data is generated for the query, it would be sent to each registered listener. We did NOT do this, but you could. Example: in this sample, the Inventory page has three different visualizations, each one driven by a unique query with its own criteria. An alternative would be to pose one generic query (select * from ItemsInventory) and let that one query feed multiple listeners. By doing this, you relieve pressure on the LiveView server. Rather than the server needing to run three queries, it runs only one.

We recommend you put here in the QueryService any type of filter or buffering desired. For instance, if the updates to your data are coming too frequently, then the QueryService is the place where the data can and should be conflated. For readers familiar with the LiveView Java APIs, there is no batchQueryListener implemented in theLiveView JS APIs. There is more on this topic in the upcoming "Additional Considerations" section.

Within our LiveQueryService we define a LiveQueryModel. It is the LiveQueryModel instances that drive the data for each of the visualizations. When an instance of a LiveQueryModel gets an event (onDelete, onInsert etc) from LiveView, it will execute the callback function(s) that was passed in. The instances could be any of the visualization types (a pie, a guage, etc).

Very important point of emphasis: This is how a view gets its query data. What a view then does with the query data is handled in the view specific .js file.

To accomplish this control center of how a view gets its query data, in LiveQueryService.js we have the prototype for LiveQueryModel with functions like:

```
addSchemaListener: function(callback, context){
```

Then within each visualization type source code (TableView.js), we see that visualization call to add different types of listeners. The implementation of the visualization specific listeners determines what the user sees on the screen. The developer must decide which listeners to add. Then, for each listener, the developer must decide what behavior to implement.

TupleStore

The TupleStore is the last piece of the sample source code. It maintains the result set of the query. A continuous query returns a bunch of tuples, and as time goes by, the query may remove or update values. This tupleStore maintains the current result set for the query. Having a current result set is useful (necessary) on updates and deletes because updates come as deltas. Given four fields, A through D, an update may come that notifies you that field A and field D have changed; the update does not tell you about fields B and C. If you need to know those other fields in order to supply that to the visualization, then you need something like the TupleStore that maintains a complete picture of the query result data. You may not need a TupleStore for your visualizations, but we have found it to be necessary with some of ours.

TAKEAWAYS: FORMULA FOR SUCCESS

Samples create practice, both as reference material and as a starting point for many projects. As such, we have created this particular sample as "ideal." Time was taken to follow standard models/premises behind those standard models, so that if this sample were to be reused, the new project could be on a path to success via good programming practices. This LiveView JS sample we hold as "best practice." The guidelines that we recommend you use are the ones we used here:

- Make a commitment to use a model (we highly recommend MVC) and stick to that commitment.
- Define your model, view and controller as well as how they will interact with one another.
- Use a service to encapsulate the access to the LiveView server connection and queries via the APIs.
    - Share the connection to the LiveView server.
    - Optimize/share queries and results. (Our sample does not do this, but if you wanted to optimize or share queries, this is where that functionality would be best suited.)
    - Throttle results. (Our sample does not do this, but if necessary, this is where that functionality would be best suited.)
- Use a TupleStore (or similar type of construct) to hold result data sets.
- Define interactions and encapsulate them via callbacks.
- Minimize the static content in your index.html. Make your app capable of being dynamic in your model. In this example, that dynamic flexibility comes from the app.model; it can accommodate any number of pages, within each page, any number of sections and within the sections, any number of components.
- Define visualizations that are reusable; provide configuration options and have a result be different visually, based on those options, but still using the same code base. In this example ChartView.js is generic and can be used with a large number of the Highchart chart types. One need only call to buildChart with a plotType of "bar" or "spline" to get a different type of visualization on the screen.

ADDITIONAL CONSIDERATIONS

There has been no attempt to address any performance related issues. Tuning a LiveView JS application is outside the scope of this paper. However, two specific items with a high potential to affect the performance of the application will be mentioned here for your consideration.

1 Load placed on the LV server
2 Data rate of result set(s)

LIVEVIEW SERVER LOAD

The number and type of queries posed as well as the start rate of queries has a tremendous potential to affect the performance of any LiveView application. Furthermore, the volume of result set for each query also has a performance impact. In this sample, when we change from one PageView to another, we cancel the queries that were running on the previously active page. Then we execute the queries on the newly selected page. There is a cost to parsing and planning all queries. For a continuous query, the startup time can be large as the snapshot portion must be computed. Therefore, it may be more beneficial to leave the queries running on the other page. Conversely, if the queries on the page that is not being viewed are sending great numbers of updates (on the order of 10K per second) then that is "expensive" and it may be best to cancel those queries when the page is not being shown to the users. Another option for the application programmer is to look at the incoming query and compare it to existing queries. Rather than posing a duplicate query, multiple visualizations could share the results of one query (you can think of it as a shared query). This sample does not do that, but using the service, the TupleStore, and callbacks, it could be implemented if deemed worthy.

If there is a concern around the number of queries being run/left running one option is to use the centralized control mechanism (the service) to keep track of how many queries are being run. You could keep a LRU list or some other policy and use it to keep track of when to cancel a given query or set of queries. All of these possible scenarios are presented here as food for thought. All implementation details are left to the developer.

DATA RATE AND RENDERING

The human eye is physically limited to how much data change it is able to see. Attempting to refresh any chart more often than the rate the eye can detect is wasted. Furthermore, if your data change rate is of a high volume, then an attempt to refresh or render each change may require a great number of resources and thus impact the performance of your application. The application developer should keep this in mind when planning to render query results. Perhaps a conflation of data is warranted. One option is to conflate in the LiveView server via periodic publish; another option is to conflate on the client side. The decision and implementation of either technique is left as an implementation detail for the LiveView JS API programmer.

TIBCO®

Global Headquarters
3307 Hillview Avenue
Palo Alto, CA 94304
+1 650-846-1000 TEL
+1 800-420-8450
+1 650-846-1005 FAX
www.tibco.com