# Data Set Schemata

var color\_palette = // various shades of blue for the charts  
[  
 '#00b0f0', '#1f497d', '#3e6ea7', '#6994c7', '#8db3e2',  
 '#b8cce4', '#31859b', '#4bacc6', '#92cddc', '#b7dde8',  
 '#0082ba', '#004499', '#0062dd', '#ccccee'  
]  
  
var schemata = // describe the data - column names and ordering  
{  
 supply: [ 'datacenter', 'region', 'machine\_type' ],  
 history: [ null, 'datacenter', 'region', 'machine\_type' ],  
 sessions: [ null, 'region', 'datacenter', 'cluster', 'user\_type', 'machine\_type' ],  
 datastores: [ 'region', 'datacenter', 'machine\_type' ],  
 dstrend: [ 'region', 'datacenter', 'machine\_type' ]  
}  
  
var aggregates = // which columns get summed when we render each chart?  
{  
 supply: [ 3, 4 ],  
 history: [ 4, 5 ],  
 sessions: [ 6, 7, 8 ],  
 datastores: [],  
 dstrend: []  
}  
  
var chart\_opts = // chart.js options for each chart  
{  
 supply:  
 {  
 title: 'VDI Supply Available for Use',  
 yAxis: 'User Sessions Available',  
 labels: [ 'Internal', 'Third Party Access' ], // these need to correspond (in order) to the numeric colums in the JSON  
 type: 'bar'  
 },  
 history:  
 {  
 title: 'VDI Supply History',  
 yAxis: 'User Sessions Available',  
 labels: [ 'Internal', 'Third Party Access' ],  
 type: 'line'  
 },  
 sessions:  
 {  
 title: 'Historical Number of VDI Sessions',  
 yAxis: 'User Sessions',  
 labels: [ '# of Free Machines', '# of Users Logged On', 'Total # of Machines Running' ],  
 type: 'line'  
 },  
 datastores:  
 {  
 title: 'VDI Datastores',  
 yAxis: '[Placeholder]',  
 labels: [ 'Category 1', 'Category 2' ],  
 type: 'bar'  
 },  
 dstrend:  
 {  
 title: 'VDI Storage Utiliztion Trends',  
 yAxis: '[Placeholder]',  
 labels: [ 'Category 1', 'Category 2' ],  
 type: 'line'  
 },  
}  
  
var predictions = [] // holds all data, all the time. not mutable  
var selections = [] // holds the subset of data that the user has selected  
var possibles = // all possible unique values for each filter will go here  
{  
 supply:  
 {  
 region: {},  
 datacenter: {},  
 machine\_type: {}  
 },  
 history:  
 {  
 region: {},  
 datacenter: {},  
 machine\_type: {}  
 },  
 sessions:  
 {  
 region: {},  
 datacenter: {},  
 cluster: {},  
 user\_type: {},  
 machine\_type: {}  
 },  
 datastores:  
 {  
 region: {},  
 datacenter: {},  
 machine\_type: {}  
 },  
 dstrend:  
 {  
 region: {},  
 datacenter: {},  
 machine\_type: {}  
 }  
}  
  
var filters = {} // this will be the subset of possible values that the user has selected in the filters  
  
$.extend( true, filters, possibles ) // deep-copy the easy way

# Filter Functions

## Handle Filter

function handle\_filter ( ev )  
{  
 let filter = $( ev.target ).attr( 'chartfilter' )  
 let chart\_key = $( ev.target ).attr( 'chartkey' )  
  
 reset\_filter( chart\_key, filter )  
  
 let filter\_vals = Object.keys( filters[ chart\_key ][ filter ] )  
 let selected\_opts = $( ev.target ).find( 'option:selected' )  
 let selected\_vals = []  
 let delete\_these = []  
  
 for ( i = 0; i < selected\_opts.length; i++ )  
 {  
 selected\_vals.push( $( selected\_opts[i] ).val() )  
 }  
  
 // console.log( 'need to search ' + filter + ' for ' + selected\_vals )  
  
 for ( i = 0 ; i < filter\_vals.length; i++ )  
 {  
 let filter\_value = filter\_vals[i]  
  
 if ( selected\_vals.indexOf( filter\_value ) == -1 )  
 {  
 delete\_these.push( filter\_value )  
 }  
 }  
  
 for ( i = 0; i < delete\_these.length; i++ )  
 {  
 delete filters[ chart\_key ][ filter ][ delete\_these[i] ]  
 }  
  
 //console.log( 'we deleted these non-matching items from filters.' + filter )  
 //console.log( delete\_these )  
 //console.log( filters[ filter ] )  
  
 reset\_selections( chart\_key ) // start from zero. get all the data and work from there.  
 apply\_filters( chart\_key ) // scan through the data and only select the data that matches the filters  
 // ...which calls: update\_widgets() // scan through the widgets and enable/disable options based on user choice  
 // ...and then calls: render\_chart() // show the chart with filters applied  
}

## Initialize All

function init\_all ()  
{  
 // create tabs for content  
 $( 'div#vdi-tabs' ).tabs()  
  
 let chart\_keys = Object.keys( possibles )  
  
 for ( i = 0 ; i < chart\_keys.length; i++ )  
 {  
 let key = chart\_keys[i]  
  
 // get json from server for the visuals...  
 $.getJSON( '/wimi/kabiyesi/' + key, function ( json ) { setup\_chart( key, json ) } )  
 }  
  
 // JSON payload will be a list of records, where each record is a list itself  
 // wherein the column order for each record will be as follows:  
 // datacenter region machine\_type INT\_supply TPA\_supply  
  
 $( 'select.filter' ).change( handle\_filter )  
  
 $( 'input.filter-reset' ).click( reset\_all )  
}

## Reset All

function reset\_all ( ev )  
{  
 let chart\_key = $( ev.target ).attr( 'chartkey' )  
  
 init\_filters( chart\_key ) // reset filters back to default state (all-selected)  
 reset\_selections( chart\_key ) // reset selections back to default state (all data)  
 populate\_filter\_widgets( chart\_key ) // put back all the options in the filters  
 update\_widgets( chart\_key ) // make all options selectable now  
 render\_chart( chart\_key ) // show the chart with all options selected  
}

## Initialize Filters

function init\_filters ( chart\_key )  
{  
 let chart\_columns = schemata[ chart\_key ]  
  
 // initialize or reset filter state-tracking variable - by default, all data  
 // is selected (filter state variable should include all possible choices)  
 for ( i = 0; i < predictions[ chart\_key ].length; i++ )  
 {  
 for ( j = 0; j < chart\_columns.length; j++ )  
 {  
 if ( chart\_columns[j] == null ) { continue }  
  
 possibles[ chart\_key ][ chart\_columns[j] ][ predictions[ chart\_key ][i][j] ] = predictions[ chart\_key ][i][j]  
 }  
 }  
  
 select\_all\_filters( chart\_key )  
  
 return filters[ chart\_key ]  
}

## Select All Filters

function select\_all\_filters ( chart\_key )  
{  
 let possible\_categories = Object.keys( possibles[ chart\_key ] )  
  
 for ( i = 0; i < possible\_categories.length; i++ )  
 {  
 let column = possible\_categories[i]  
 let values = Object.keys( possibles[ chart\_key ][ column ] )  
  
 for ( j = 0; j < values.length; j++ )  
 {  
 filters[ chart\_key ][ column ][ values[j] ] = values[j]  
 }  
 }  
  
 return filters  
}

## Reset Filter

function reset\_filter ( chart\_key, filter )  
{  
 //console.log( 'resetting filters.' + filter + ' to ... ' )  
 //console.log( possibles[ filter ] )  
  
 let restore\_vals = Object.keys( possibles[ chart\_key ][ filter ] )  
  
 for ( i = 0; i < restore\_vals.length; i++ )  
 {  
 filters[ chart\_key ][ filter ][ restore\_vals[i] ] = restore\_vals[i]  
 }  
}

## Populate Filter Widgets

function populate\_filter\_widgets ( chart\_key )  
{  
 // first empty any non-header options from all filter widgets  
 $( 'select.filter[chartkey="' + chart\_key + '"] option' ).each  
 (  
 function ()  
 {  
 if ( $( this ).val() != '' ) // header options have no value  
 {  
 $( this ).remove()  
 }  
 }  
 )  
  
 let filter\_names = Object.keys( possibles[ chart\_key ] )  
  
 // now iterate over predictions data and populate filters  
 for ( i = 0; i < filter\_names.length; i++ )  
 {  
 let filter = filter\_names[i]  
 let values = Object.keys( possibles[ chart\_key ][ filter ] ).sort()  
 let widget = $( 'select.filter[chartkey="' + chart\_key + '"][chartfilter="' + filter + '"]' )  
  
 //console.log( 'restoring all possible options to filter: ' + filter + ' for chart key: ' + chart\_key )  
 //console.log( { restoring\_values: values, to\_select\_widget: widget } )  
  
 // then stuff the widget with the options that it should have, based on the data (nod to Marta ;-))  
 for ( j = 0; j < values.length; j++ )  
 {  
 widget.append( '<option value="' + values[j] + '">' + values[j] + '</option>' )  
 }  
 }  
}

## Update Filter Widgets

function update\_widgets ( chart\_key )  
{  
 // based on user selections, update filter widgets to reflect what choices  
 // have been made, and what choices are/are not possible  
 let filter\_names = Object.keys( filters[ chart\_key ] )  
  
 filter\_scan: for ( i = 0; i < filter\_names.length; i++ ) // loop through each filter widget  
 {  
 let filter\_name = filter\_names[i]  
 let col\_index = schemata[ chart\_key ].indexOf( filter\_name )  
 let options = $( 'select.filter[chartkey="' + chart\_key + '"][chartfilter="' + filter\_name + '"] option' ) // get filter widget options  
  
 options.attr( { disabled: true } ) // disable all options in the widget  
  
 opt\_scan: for ( j = 0; j < options.length; j++ ) // scan through the widget options and...  
 {  
 let opt\_val = $( options ).eq( j ).val()  
  
 data\_scan: for ( k = 0; k < selections[ chart\_key ].length; k++ ) // check selections list for matching records  
 {  
 if ( selections[ chart\_key ][k][ col\_index ] == opt\_val ) // enable the option if it matches user choices in "selections" variable  
 {  
 $( options ).eq( j ).attr( { disabled: false } )  
  
 break data\_scan // bail on first match (no need to keep pounding)  
 }  
 }  
 }  
 }  
  
 // delete the disabled options unless it's the header opt with a none-value  
 $( 'select.filter[chartkey="' + chart\_key + '"] option:disabled' ).each  
 (  
 function ()  
 {  
 if ( $( this ).val() != '' )  
 {  
 $( this ).remove()  
 }  
 }  
 )  
}

## Apply Filters

function apply\_filters ( chart\_key )  
{  
 filter\_selections( chart\_key )  
  
 update\_widgets( chart\_key )  
  
 render\_chart( chart\_key )  
}

## Reset Selections

function reset\_selections ( chart\_key ) // reset selections to 'everything' selected  
{  
 selections[ chart\_key ] = predictions[ chart\_key ] // for now, it's really not more complicated than this ;-)  
}

## Filter Selections

function filter\_selections ( chart\_key )  
{  
 let results = []  
 let filter\_names = Object.keys( filters[ chart\_key ] )  
  
 // iterate over records and check if each one matches users filters (choices)  
 scan\_records: for ( i = 0; i < selections[ chart\_key ].length; i++ )  
 {  
 let record = selections[ chart\_key ][i]  
  
 // start checking this record, filter by filter, and skip over it at  
 // the first instance where it fails to match  
 check\_record: for ( j = 0; j < filter\_names.length; j++ )  
 {  
 let filter\_name = filter\_names[j] // name of the filter we're dealing with  
 let filter\_column = schemata[ chart\_key ].indexOf( filter\_name ) // db record column index to which this filter pertains  
 let column\_value = record[ filter\_column ] // value of the db record in the column to which this filter pertains  
  
 if ( filters[ chart\_key ][ filter\_name ][ column\_value ] == undefined ) // e.g - filters.supply.region.NAM  
 {  
 continue scan\_records // quit checking for matches on this record at the first sign of failure  
 }  
 }  
  
 results.push( record ) // if we got here, all filters matched, so save result  
 }  
  
 selections[ chart\_key ] = results  
  
 return results  
}

## In Filter

// check if given value exists in given filter using dictionary key lookup  
  
function in\_filter ( value, filter ) { return filter[ value ] != null }

# Chart Functions

## Setup Chart

function setup\_chart ( chart\_key, json )  
{  
 // instantiate data  
 predictions[ chart\_key ] = json // predictions shouldn't be messed with - holds all data  
 selections[ chart\_key ] = predictions[ chart\_key ] // default to all records selected  
  
 // the selections variable should always reflect the predictions data after  
 // all user filters have been applied.  
  
 // initialize filter state-tracking variable - by default, all data is  
 // selected (filter state variable should include all possible choices)  
 init\_filters( chart\_key )  
  
 // populate filter widgets  
 populate\_filter\_widgets( chart\_key )  
  
 render\_chart( chart\_key )  
}

## Render Chart

function render\_chart ( chart\_key )  
{  
 // render chart.js chart when selections change  
 //let match\_count = $( 'select.filter[chartkey="' + chart\_key + '"][chartindicator] option' ).length - 1  
 //let match\_indic = $( 'select.filter[chartkey="' + chart\_key + '"][chartindicator]' ).attr( 'chartindicator' ) // e.g.- "VMs"  
  
 //$( 'span.status[chartkey="' + chart\_key + '"]' ).text( 'showing ' + match\_count + ' matching ' + match\_indic )  
  
 let chart\_data = make\_chartjs\_datasets( chart\_key )  
 let my\_opts = chart\_opts[ chart\_key ]  
  
 $( 'div.chart-div[chartkey="' + chart\_key + '"]' ).html( '<canvas class="chart-canvas" chartkey="' + chart\_key + '"></canvas>' )  
  
 let vdi\_ctx = $( 'canvas.chart-canvas[chartkey="' + chart\_key + '"]' )  
  
 // colors and other per-dataset options  
 for ( i = 0; i < chart\_data.datasets.length; i++ )  
 {  
 chart\_data.datasets[i].backgroundColor = color\_palette[i]  
 chart\_data.datasets[i].borderColor = color\_palette[i]  
 chart\_data.datasets[i].fill = my\_opts.type == 'line' ? false : undefined  
 }  
  
 var vdi\_chart = new Chart  
 (  
 vdi\_ctx,  
 {  
 type: my\_opts.type,  
 data: chart\_data,  
 options:  
 {  
 title: { display: true, text: my\_opts.title },  
 legend: { display: true, position: 'top' },  
 responsive: true,  
 maintainAspectRatio: false,  
 hoverMode: 'index',  
 scales:  
 {  
 yAxes:  
 [  
 {  
 scaleLabel: { display: true, labelString: my\_opts.yAxis },  
 ticks:  
 {  
 beginAtZero: true,  
 min: 0,  
 callback: function ( label, index, labels )  
 {  
 return label.toLocaleString()  
 }  
 }  
 }  
 ]  
 },  
 tooltips:  
 {  
 callbacks:  
 {  
 label: function ( tooltipItem, data )  
 {  
 return data.datasets[ tooltipItem.datasetIndex ].data[ tooltipItem.index ].toLocaleString()  
 }  
 }  
 }  
 }  
 }  
 )  
}

## Make Chart.js Datasets

function make\_chartjs\_datasets ( chart\_key )  
{  
 // make a data structure based on the current contents of the selections  
 // variable. the data structure will be fed to chart.js to create a new  
 // chart, so it needs to be in a certain format that follows certain rules...  
  
 let chart\_struct =  
 {  
 labels: [], // labels for x-axis  
 datasets: [] // values from numeric columns of data, after aggregation  
 }  
  
 // each chart.js dataset needs a label. we define these in the chart\_opts  
 // global var at the top of this file, and reference it here, by chart key  
  
 for ( i = 0; i < chart\_opts[ chart\_key ].labels.length; i++ ) // line/bar labels  
 {  
 chart\_struct.datasets[i] = { label: chart\_opts[ chart\_key ].labels[i], data: [] }  
 }  
  
 // part of this process will be aggregation of the data. we will group  
 // the data by the unique values from the first column of data in all the  
 // db records. Then we will sum up the numeric data in each column that  
 // is marked for aggregation in the global "aggreagetes" variable, by chart key.  
  
 let group\_by = {} // keys will be the unique values from the first column of db records for the chart  
  
 // first we need to determine what our groupings are. We let the data tell  
 // us its own story. Specifically, we scan through the data to identify the  
 // unique instances of whatever is getting grouped. Since we always group by  
 // the data in the first column, we can take that as a given and hard-code  
 // the zeroth item in each record we scan. since dictionary keys can't be  
 // duplicated, we leverage that feature in order to create our distinct groups  
  
 for ( i = 0; i < selections[ chart\_key ].length; i++ )  
 {  
 let group\_by\_value = selections[ chart\_key ][i][0] // get the group-by column value  
  
 // the column value is used as the dictionary key, and the value is just  
 // an empty list that will later hold the summed values for that column  
  
 group\_by[ group\_by\_value ] = []  
 }  
  
 // by virtue of the loop above, we now have a distinct list of values from  
 // the first column of data for this chart. It is contained in the keys  
 // of the group\_by dictionary. In addition to needing these values to label  
 // the x-axis of the chart (which we do by saving them in the  
 // chart\_struct.labels list variable), but we're going to need to refer to  
 // this list of values several more times before we're done with the  
 // aggregation we need to do  
  
 chart\_struct.labels = Object.keys( group\_by ).sort()  
  
 // now we need to start summing things up. Our aggregates global variable  
 // contains the column index offset of the column(s) in each DB record that  
 // need to be SUM()'med, so we use that below in order to avoid asking JS  
 // to add non-numeric columns of data  
  
 for ( i = 0; i < selections[ chart\_key ].length; i++ ) // loop through all records  
 {  
 let group\_by\_value = selections[ chart\_key ][i][0]  
  
 // for every record, loop through columns getting SUM()'med  
  
 for ( j = 0 ; j < aggregates[ chart\_key ].length; j++ )  
 {  
 // line below is so we don't get NaN's in the output  
 group\_by[ group\_by\_value ][j] = group\_by[ group\_by\_value ][j] ? group\_by[ group\_by\_value ][j] : 0  
  
 // this actually SUM()'s the column, grouped by whatever the value is  
 // in the first column of the DB record we are looking at in this  
 // iteration of the loop...  
  
 group\_by[ group\_by\_value ][j] += parseFloat( selections[ chart\_key ][i][ aggregates[ chart\_key ][j] ] )  
 }  
 }  
  
 // and now we need to populate the chart.js data structure with the data  
 // we just aggregated (grouped and summed) in the loop above  
  
 for ( i = 0 ; i < aggregates[ chart\_key ].length; i++ ) // loop through columns that got SUM()'med  
 {  
 for ( j = 0; j < chart\_struct.labels.length; j++ ) // loop through each of the distinct values in each grouping  
 {  
 let group\_by\_value = chart\_struct.labels[j] // this is the group-by data category  
  
 // for each of the groupings of data we have, shove the data for that  
 // rouping into the chart.js dataset pertaining to whatever column  
 // we're currently iterating over in the outer loop that encloses  
 // this nested loop...  
  
 chart\_struct.datasets[i].data.push( group\_by[ group\_by\_value ][i] )  
 }  
 }  
  
 // console.log( { group\_by: group\_by, chart\_struct: chart\_struct, chart\_key: chart\_key } )  
  
 return chart\_struct  
}

## Init\_all

$( init\_all )