**New Axes**

Axes in Chart.js can be individually extended. Axes should always derive from Chart.Scale but this is not a mandatory requirement.

let MyScale = Chart.Scale.extend({

/\* extensions ... \*/

});

// MyScale is now derived from Chart.Scale

Once you have created your scale class, you need to register it with the global chart object so that it can be used. A default config for the scale may be provided when registering the constructor. The first parameter to the register function is a string key that is used later to identify which scale type to use for a chart.

Chart.scaleService.registerScaleType('myScale', MyScale, defaultConfigObject);

To use the new scale, simply pass in the string key to the config when creating a chart.

var lineChart = new Chart(ctx, {

data: data,

type: 'line',

options: {

scales: {

yAxes: [{

type: 'myScale' // this is the same key that was passed to the registerScaleType function

}]

}

}

});

**Scale Properties**

Scale instances are given the following properties during the fitting process.

{

left: number, // left edge of the scale bounding box

right: number, // right edge of the bounding box

top: number,

bottom: number,

width: number, // the same as right - left

height: number, // the same as bottom - top

// Margin on each side. Like css, this is outside the bounding box.

margins: {

left: number,

right: number,

top: number,

bottom: number

},

// Amount of padding on the inside of the bounding box (like CSS)

paddingLeft: number,

paddingRight: number,

paddingTop: number,

paddingBottom: number

}

**Scale Interface**

To work with Chart.js, custom scale types must implement the following interface.

{

// Determines the data limits. Should set this.min and this.max to be the data max/min

determineDataLimits: function() {},

// Generate tick marks. this.chart is the chart instance. The data object can be accessed as this.chart.data

// buildTicks() should create a ticks array on the axis instance, if you intend to use any of the implementations from the base class

buildTicks: function() {},

// Get the value to show for the data at the given index of the the given dataset, ie this.chart.data.datasets[datasetIndex].data[index]

getLabelForIndex: function(index, datasetIndex) {},

// Get the pixel (x coordinate for horizontal axis, y coordinate for vertical axis) for a given value

// @param index: index into the ticks array

getPixelForTick: function(index) {},

// Get the pixel (x coordinate for horizontal axis, y coordinate for vertical axis) for a given value

// @param value : the value to get the pixel for

// @param index : index into the data array of the value

// @param datasetIndex : index of the dataset the value comes from

getPixelForValue: function(value, index, datasetIndex) {},

// Get the value for a given pixel (x coordinate for horizontal axis, y coordinate for vertical axis)

// @param pixel : pixel value

getValueForPixel: function(pixel) {}

}

Optionally, the following methods may also be overwritten, but an implementation is already provided by the Chart.Scale base class.

{

// Transform the ticks array of the scale instance into strings. The default implementation simply calls this.options.ticks.callback(numericalTick, index, ticks);

convertTicksToLabels: function() {},

// Determine how much the labels will rotate by. The default implementation will only rotate labels if the scale is horizontal.

calculateTickRotation: function() {},

// Fits the scale into the canvas.

// this.maxWidth and this.maxHeight will tell you the maximum dimensions the scale instance can be. Scales should endeavour to be as efficient as possible with canvas space.

// this.margins is the amount of space you have on either side of your scale that you may expand in to. This is used already for calculating the best label rotation

// You must set this.minSize to be the size of your scale. It must be an object containing 2 properties: width and height.

// You must set this.width to be the width and this.height to be the height of the scale

fit: function() {},

// Draws the scale onto the canvas. this.(left|right|top|bottom) will have been populated to tell you the area on the canvas to draw in

// @param chartArea : an object containing four properties: left, right, top, bottom. This is the rectangle that lines, bars, etc will be drawn in. It may be used, for example, to draw grid lines.

draw: function(chartArea) {}

}

The Core.Scale base class also has some utility functions that you may find useful.

{

// Returns true if the scale instance is horizontal

isHorizontal: function() {},

// Get the correct value from the value from this.chart.data.datasets[x].data[]

// If dataValue is an object, returns .x or .y depending on the return of isHorizontal()

// If the value is undefined, returns NaN

// Otherwise returns the value.

// Note that in all cases, the returned value is not guaranteed to be a number

getRightValue: function(dataValue) {},

// Returns the scale tick objects ({label, major})

getTicks: function() {}

}

# Chart Prototype Methods

For each chart, there are a set of global prototype methods on the shared chart type which you may find useful. These are available on all charts created with Chart.js, but for the examples, let's use a line chart we've made.

// For example:

var myLineChart = new Chart(ctx, config);

## .destroy()

Use this to destroy any chart instances that are created. This will clean up any references stored to the chart object within Chart.js, along with any associated event listeners attached by Chart.js. This must be called before the canvas is reused for a new chart.

// Destroys a specific chart instance

myLineChart.destroy();

## .update(config)

Triggers an update of the chart. This can be safely called after updating the data object. This will update all scales, legends, and then re-render the chart.

// duration is the time for the animation of the redraw in milliseconds

// lazy is a boolean. if true, the animation can be interrupted by other animations

myLineChart.data.datasets[0].data[2] = 50; // Would update the first dataset's value of 'March' to be 50

myLineChart.update(); // Calling update now animates the position of March from 90 to 50.

**Note:** replacing the data reference (e.g. myLineChart.data = {datasets: [...]} only works starting **version 2.6**. Prior that, replacing the entire data object could be achieved with the following workaround: myLineChart.config.data = {datasets: [...]}.

A config object can be provided with additional configuration for the update process. This is useful when update is manually called inside an event handler and some different animation is desired.

The following properties are supported:

* **duration** (number): Time for the animation of the redraw in milliseconds
* **lazy** (boolean): If true, the animation can be interrupted by other animations
* **easing** (string): The animation easing function. See [Animation Easing](https://www.chartjs.org/docs/2.9.4/configuration/animations.html) for possible values.

Example:

myChart.update({

duration: 800,

easing: 'easeOutBounce'

});

See [Updating Charts](https://www.chartjs.org/docs/2.9.4/developers/updates.html) for more details.

## .reset()

Reset the chart to it's state before the initial animation. A new animation can then be triggered using update.

myLineChart.reset();

## .render(config)

Triggers a redraw of all chart elements. Note, this does not update elements for new data. Use .update() in that case.

See .update(config) for more details on the config object.

// duration is the time for the animation of the redraw in milliseconds

// lazy is a boolean. if true, the animation can be interrupted by other animations

myLineChart.render({

duration: 800,

lazy: false,

easing: 'easeOutBounce'

});

## .stop()

Use this to stop any current animation loop. This will pause the chart during any current animation frame. Call .render() to re-animate.

// Stops the charts animation loop at its current frame

myLineChart.stop();

// => returns 'this' for chainability

## .resize()

Use this to manually resize the canvas element. This is run each time the canvas container is resized, but you can call this method manually if you change the size of the canvas nodes container element.

// Resizes & redraws to fill its container element

myLineChart.resize();

// => returns 'this' for chainability

## .clear()

Will clear the chart canvas. Used extensively internally between animation frames, but you might find it useful.

// Will clear the canvas that myLineChart is drawn on

myLineChart.clear();

// => returns 'this' for chainability

## .toBase64Image()

This returns a base 64 encoded string of the chart in it's current state.

myLineChart.toBase64Image();

// => returns png data url of the image on the canvas

## .generateLegend()

Returns an HTML string of a legend for that chart. The legend is generated from the legendCallback in the options.

myLineChart.generateLegend();

// => returns HTML string of a legend for this chart

## .getElementAtEvent(e)

Calling getElementAtEvent(event) on your Chart instance passing an argument of an event, or jQuery event, will return the single element at the event position. If there are multiple items within range, only the first is returned. The value returned from this method is an array with a single parameter. An array is used to keep a consistent API between the get\*AtEvent methods.

myLineChart.getElementAtEvent(e);

// => returns the first element at the event point.

To get an item that was clicked on, getElementAtEvent can be used.

function clickHandler(evt) {

var firstPoint = myChart.getElementAtEvent(evt)[0];

if (firstPoint) {

var label = myChart.data.labels[firstPoint.\_index];

var value = myChart.data.datasets[firstPoint.\_datasetIndex].data[firstPoint.\_index];

}

}

## .getElementsAtEvent(e)

Looks for the element under the event point, then returns all elements at the same data index. This is used internally for 'label' mode highlighting.

Calling getElementsAtEvent(event) on your Chart instance passing an argument of an event, or jQuery event, will return the point elements that are at that the same position of that event.

canvas.onclick = function(evt) {

var activePoints = myLineChart.getElementsAtEvent(evt);

// => activePoints is an array of points on the canvas that are at the same position as the click event.

};

This functionality may be useful for implementing DOM based tooltips, or triggering custom behaviour in your application.

## .getDatasetAtEvent(e)

Looks for the element under the event point, then returns all elements from that dataset. This is used internally for 'dataset' mode highlighting.

myLineChart.getDatasetAtEvent(e);

// => returns an array of elements

## .getDatasetMeta(index)

Looks for the dataset that matches the current index and returns that metadata. This returned data has all of the metadata that is used to construct the chart.

The data property of the metadata will contain information about each point, rectangle, etc. depending on the chart type.

Extensive examples of usage are available in the [Chart.js tests](https://github.com/chartjs/Chart.js/tree/master/test).

var meta = myChart.getDatasetMeta(0);

var x = meta.data[0].\_model.x;

**Updating Charts**

It's pretty common to want to update charts after they've been created. When the chart data or options are changed, Chart.js will animate to the new data values and options.

**Adding or Removing Data**

Adding and removing data is supported by changing the data array. To add data, just add data into the data array as seen in this example.

function addData(chart, label, data) {

chart.data.labels.push(label);

chart.data.datasets.forEach((dataset) => {

dataset.data.push(data);

});

chart.update();

}

function removeData(chart) {

chart.data.labels.pop();

chart.data.datasets.forEach((dataset) => {

dataset.data.pop();

});

chart.update();

}

**Updating Options**

To update the options, mutating the options property in place or passing in a new options object are supported.

* If the options are mutated in place, other option properties would be preserved, including those calculated by Chart.js.
* If created as a new object, it would be like creating a new chart with the options - old options would be discarded.

function updateConfigByMutating(chart) {

chart.options.title.text = 'new title';

chart.update();

}

function updateConfigAsNewObject(chart) {

chart.options = {

responsive: true,

title: {

display: true,

text: 'Chart.js'

},

scales: {

xAxes: [{

display: true

}],

yAxes: [{

display: true

}]

}

};

chart.update();

}

Scales can be updated separately without changing other options. To update the scales, pass in an object containing all the customization including those unchanged ones.

Variables referencing any one from chart.scales would be lost after updating scales with a new id or the changed type.

function updateScales(chart) {

var xScale = chart.scales['x-axis-0'];

var yScale = chart.scales['y-axis-0'];

chart.options.scales = {

xAxes: [{

id: 'newId',

display: true

}],

yAxes: [{

display: true,

type: 'logarithmic'

}]

};

chart.update();

// need to update the reference

xScale = chart.scales['newId'];

yScale = chart.scales['y-axis-0'];

}

You can also update a specific scale either by specifying its index or id.

function updateScale(chart) {

chart.options.scales.yAxes[0] = {

type: 'logarithmic'

};

chart.update();

}

Code sample for updating options can be found in [toggle-scale-type.html](https://www.chartjs.org/docs/samples/scales/toggle-scale-type.html).

**Preventing Animations**

Sometimes when a chart updates, you may not want an animation. To achieve this you can call update with a duration of 0. This will render the chart synchronously and without an animation.