DGS-KORPUS

Using d3js to Visualise Data in iLex

Edited by: Thomas Hanke, 2016-05-31
Thomas Hanke, 2017-02-19, smaller corrections, update to d3js v4 and addition of code-walkthroughs

Introduction

Visualisation is a key to make complex data accessible. Consequently, iLex has provided business charts, graphs, and finally maps functionality since quite some time. Project note AP04-2013-01 documents the possibilities of the three rendering engines available back then. In 2016, we added support for d3js as another rendering engine. d3js offers powerful visualisation components that go far beyond what was possible before.[1] As d3js works with standard web technologies, all its speed and versatility is available to the iLex user without us having to provide new chart styles first before users can work with them. We expect that d3js charts will completely replace charts created with the older rendering engines. While we provide chart styles using d3js that can replace the older ones, going beyond that needs some familiarity of how d3js is integrated into iLex. That is what this report is about.

d3js Charts in iLex

The general approach to creating charts in iLex is as follows:

- The user selects some data to be charted.
- The user selects a chart definition from the menu of all available charts for the current context.
- The main component of a chart definition is an SQL statement providing the data to be visualised. It can be parametrised to include the user selection.
- The result of the query is then fed into the rendering engine. In the case of d3js, the code in iLex just uses the HTML template defined in the chart style and fills in the result data as well as some options.
- The resulting HTML file is then loaded into an in-app web browser window. The d3js Javascript code contained in the HTML file then creates an SVG within the HTML document which is automatically displayed by the browser.

The HTML output can be saved as an HTML file, as an SVG file, or printed to PDF or PNG.[2]

As none of the charts are required for iLex to operate, neither the charts nor the chart styles are part of the iLex standard distribution. However, we make a selection of chart styles and sample charts available in the iLex wiki.

[1] Hanke (2016) shows some examples.
[2] For dynamic charts, we also see the need to save to movie file format. This is, however, not yet implemented.

Standard Data Model for Charts

While not mandatory, we recommend that all chart style definitions expect the chart SQL statement to produce a set of data points in a 3D (or 4D) space plus optional extra elements as such a format is easy to produce in SQL. The idea is to have each data point to be a quadruple

(t,x2,x1,y)

with:

y	The value for the datapoint (x2,x1) at the time t. This is a number value, depending on the type of chart this may need to be a non-negative number. Null values for y a commonly interpreted as 0 although this depends on the chart style implementation.
x1	The "main" x value may be a cardinal or nominal datum, depending on the kind of chart to be produced. In a scatter diagram, it is a cardinal datum whereas in a pie chart x1 is simply the category for a slice the width of which is determined by y.
x2	The "secondary" x value can be understood as the group of data the x1 is belonging to. It is always a nominal value. In a case of a scatter diagram, all points with the same x2 share a colour. For pie charts, the chart style generates separate pies for each x2. For maps, the group is the region code that the (x1,y) value belongs to, i.e. the renderer will draw the (x1,y) value in a predefined way at the position on the map determined by the code x2. In the case of maps, x2 is obligatory, in other cases you can provide null if you do not need a way of grouping the data points.
t	A timestamp in dynamic charts. However, dynamic charts are not yet implemented which means that timestamps are ignored and all data are presented at the same time. As these timestamps can often be used as titles for the chart, this field can be set to a constant string to be show up as the title. If not needed, you can safely select a null for t.

Adhering to this data model has the advantage that the user can switch between different chart styles without having to adapt the SQL query.
Additional columns in the query may provide extra data such as:

label	A text to be displayed on the graphical representation for (x1,y) data point in the context defined by x2 at timestamp t, e.g. the y count on a pie chart slice. Can be null.
tooltip	Extra text for (x1,y) only to be displayed when the user hovers over the graphical representation. This text may be longer than the label. Can be null (i.e. no tooltip is displayed).
target	A URL that is opened if the user clicks on the graphical representation. In most cases, this will be an iLex-internal URL, such as ilex://types.id=13189. Then iLex opens a window to display the data set specified.
color	If provided the graphical representation of a dataset will be colored as specified here. If not provided, the chart style will probably use a standard palette differentiating the data points by their x2 or x1 values.

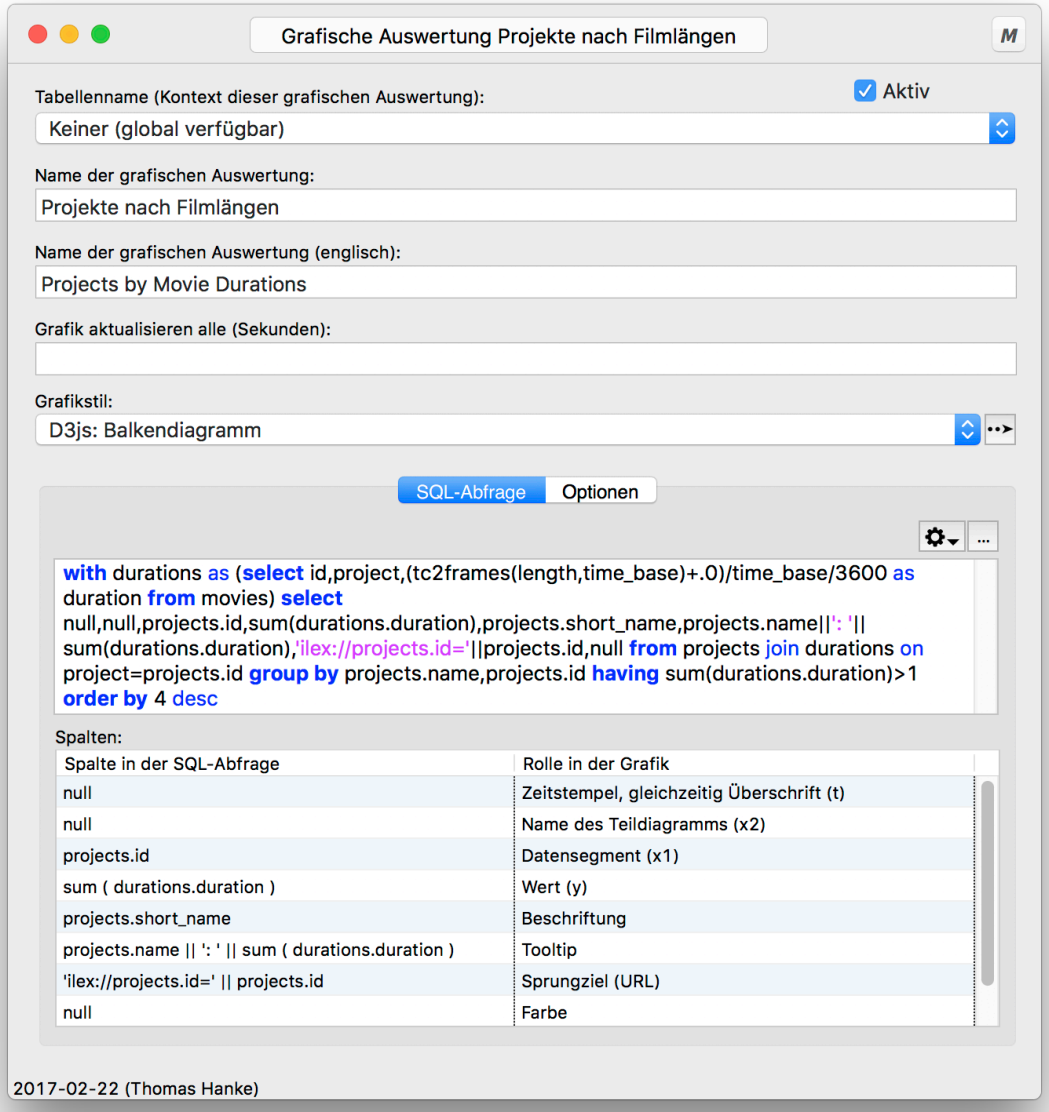
Defining a Chart

A chart is defined by selecting a chart style and providing an SQL query following the style's data model. As the chart style defines the columns to be provided by the query, iLex checks the SQL statement to be compatible with the chart style. In addition, the chart style may define some options that can be checked on or off for the chart to be defined.
If the chart is tied to a specific table (i.e. not globally available from the Charts menu, but via the Charts button above lists displaying content from the selected table), you may refer to the

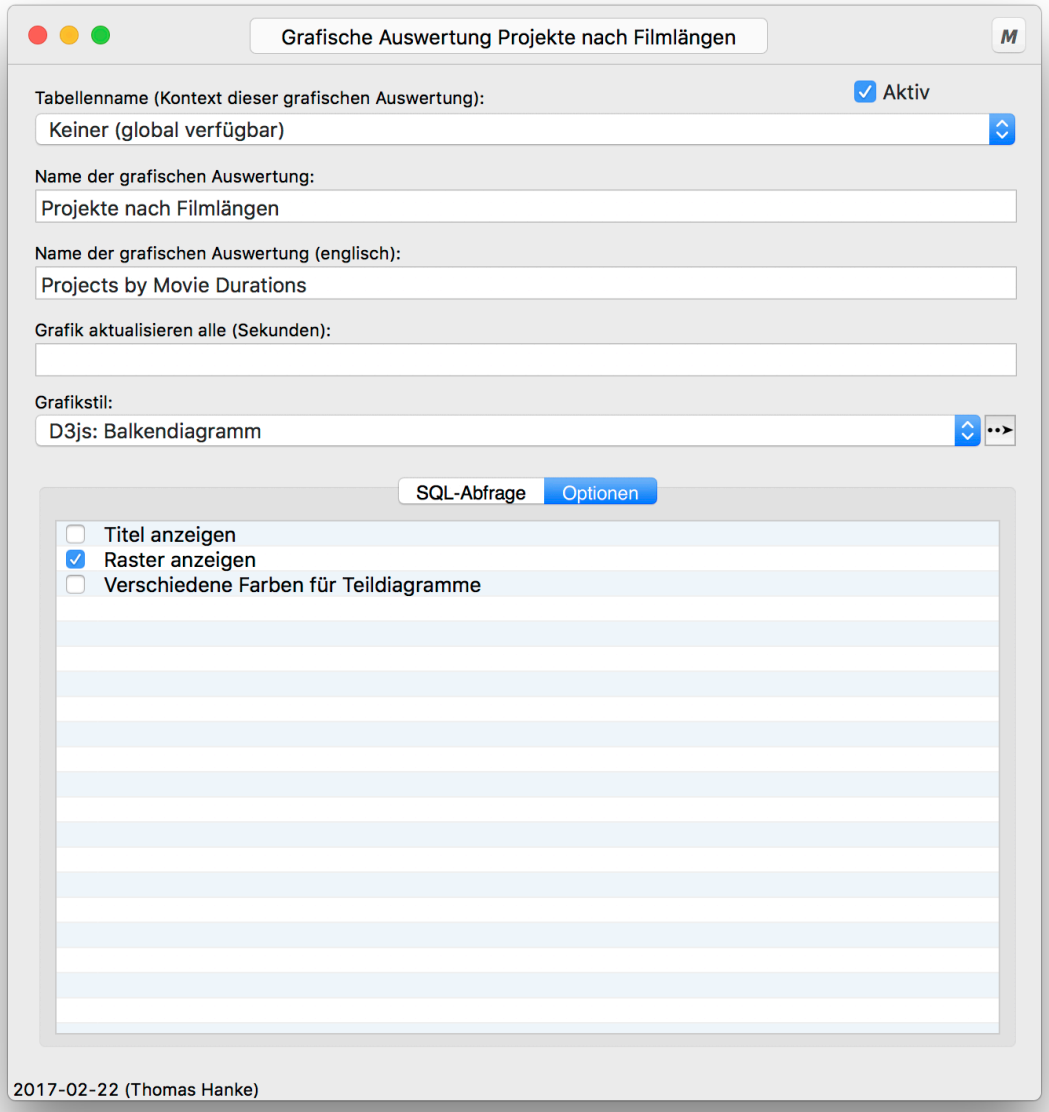
current user selection from within the query: $1 will be replaced by the comma-separated list of ids of the selected items. In that case, the SQL statement typically contains something like

```
where transcripts.id in ($1)
```

Here is an example based upon a chart style for bar charts:



The SQL fills the columns defined by the chart style, although in some cases only with null. As no text is provided for t, it does not make sense to check the Show Title option:



Defining a d3js Chart Style

A d3js graphics in iLex is an HTML document consisting of three parts:

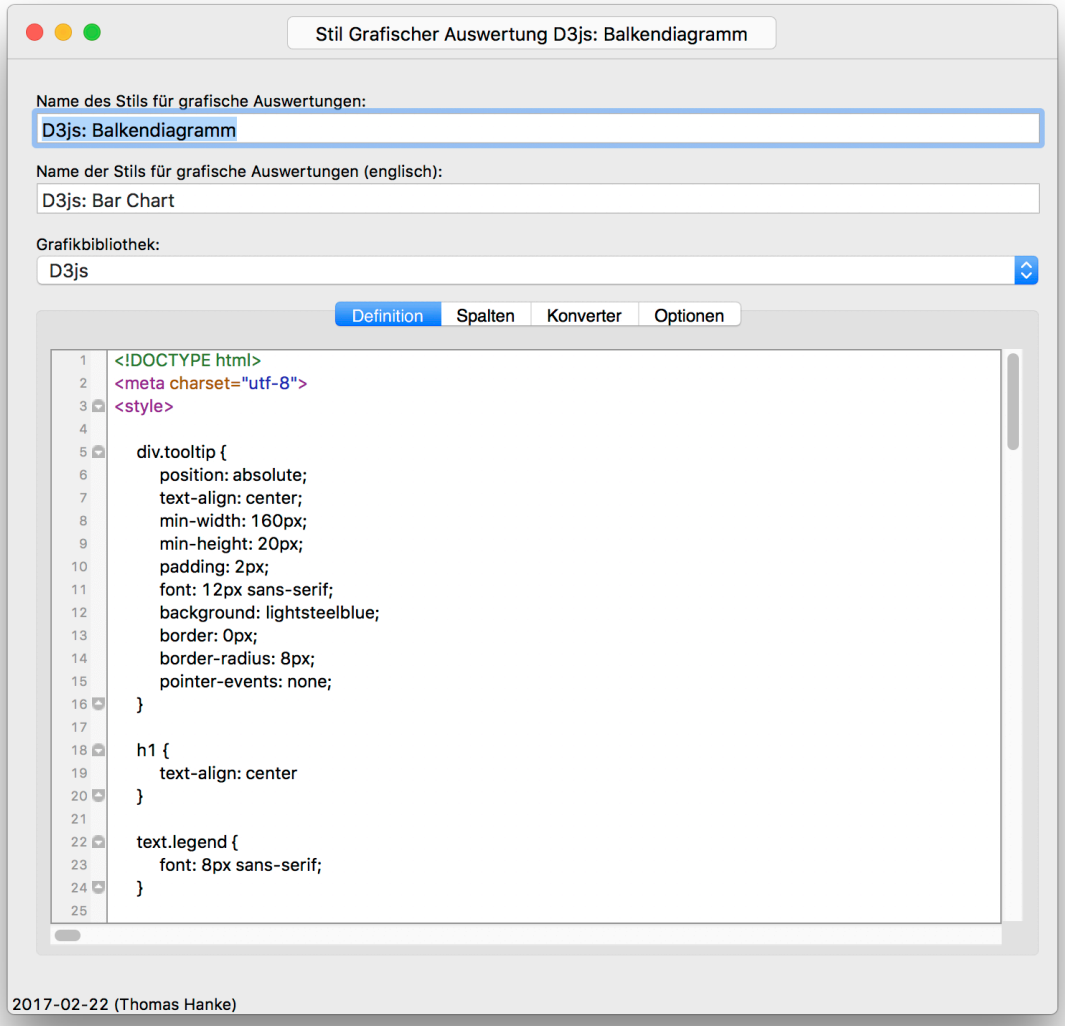
css in the HTML header	Defines the graphical styles used to format graphics and text in the chart.
script in the HTML header	Javascript making use of the d3js library in order to produce one or more svg graphics to the placed in the HMTL document's dom hierarchy. The data are part of the Javascript code.
HTML body, typically empty	Will be filled dynamically by the Javascript code.

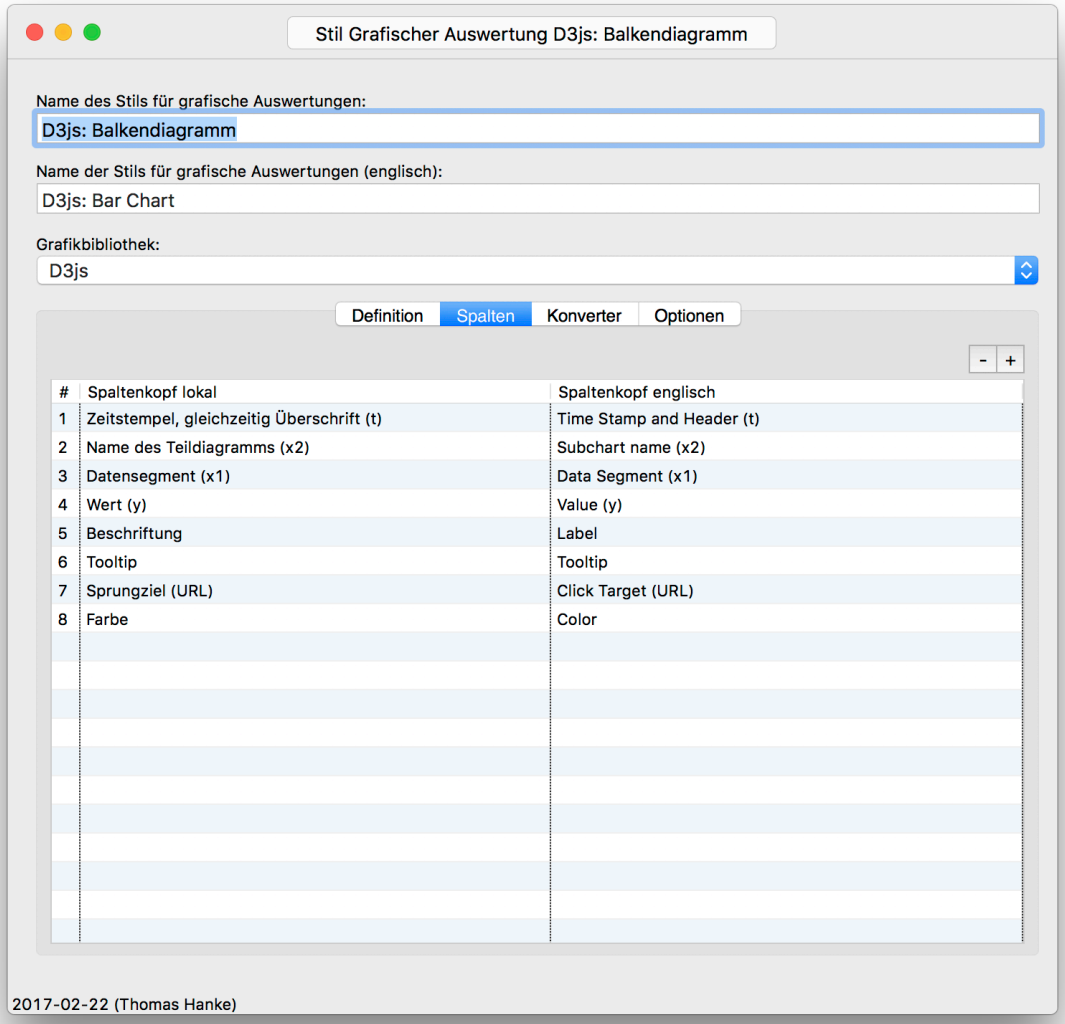
A chart style defines an HTML template using the variable $1 where the data (i.e. the result of the SQL query) is placed and the variables $2…$9 replaced by 0 or 1 in order to hold the option values as determined in the chart definition.
As the easiest way for a Javascript to access structured data is the json format, we recommend that you use the chart style's converter setting to convert the tabular data point format resulting from the SQL query to json:

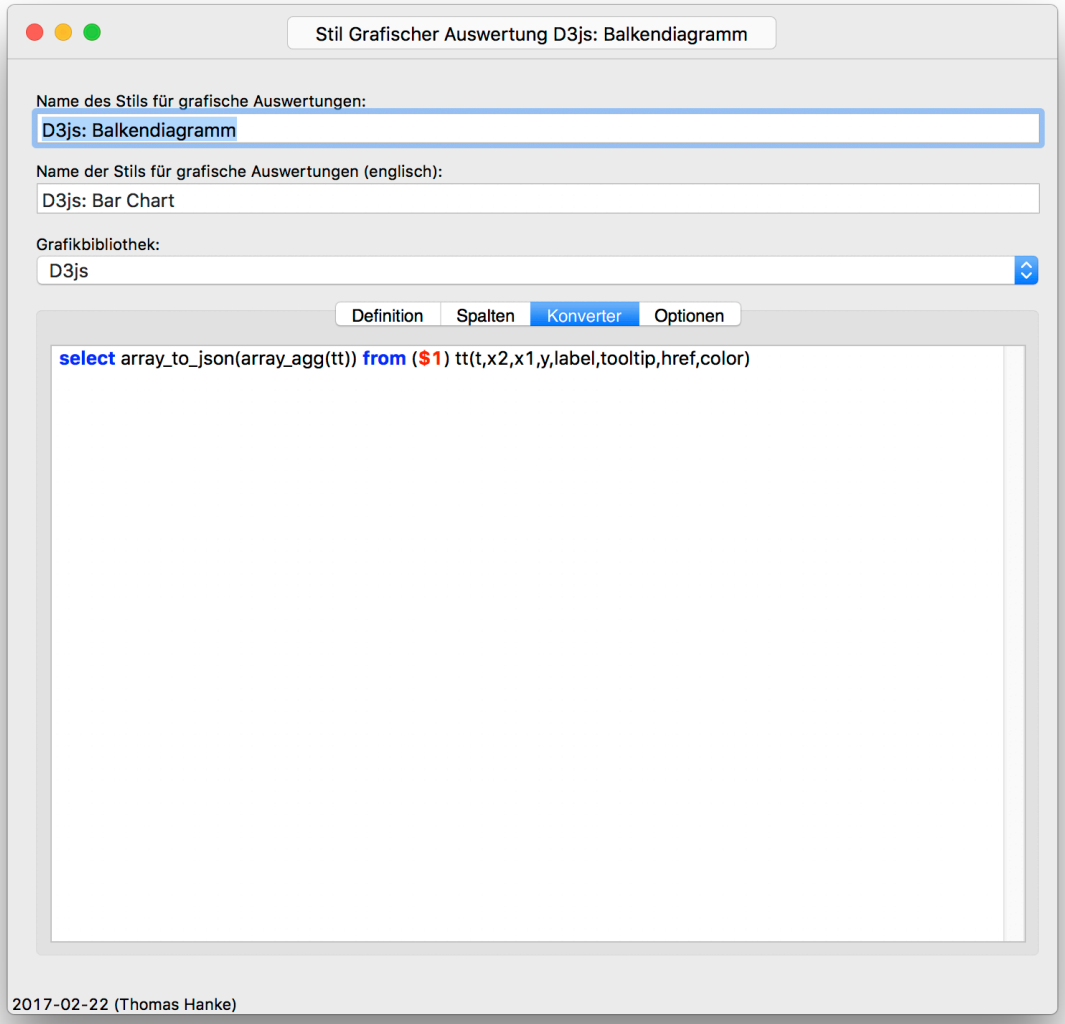
```
select array_to_json(array_agg(tt)) from ($1)
tt(t,x2,x1,y,label,tooltip,href,color)
```

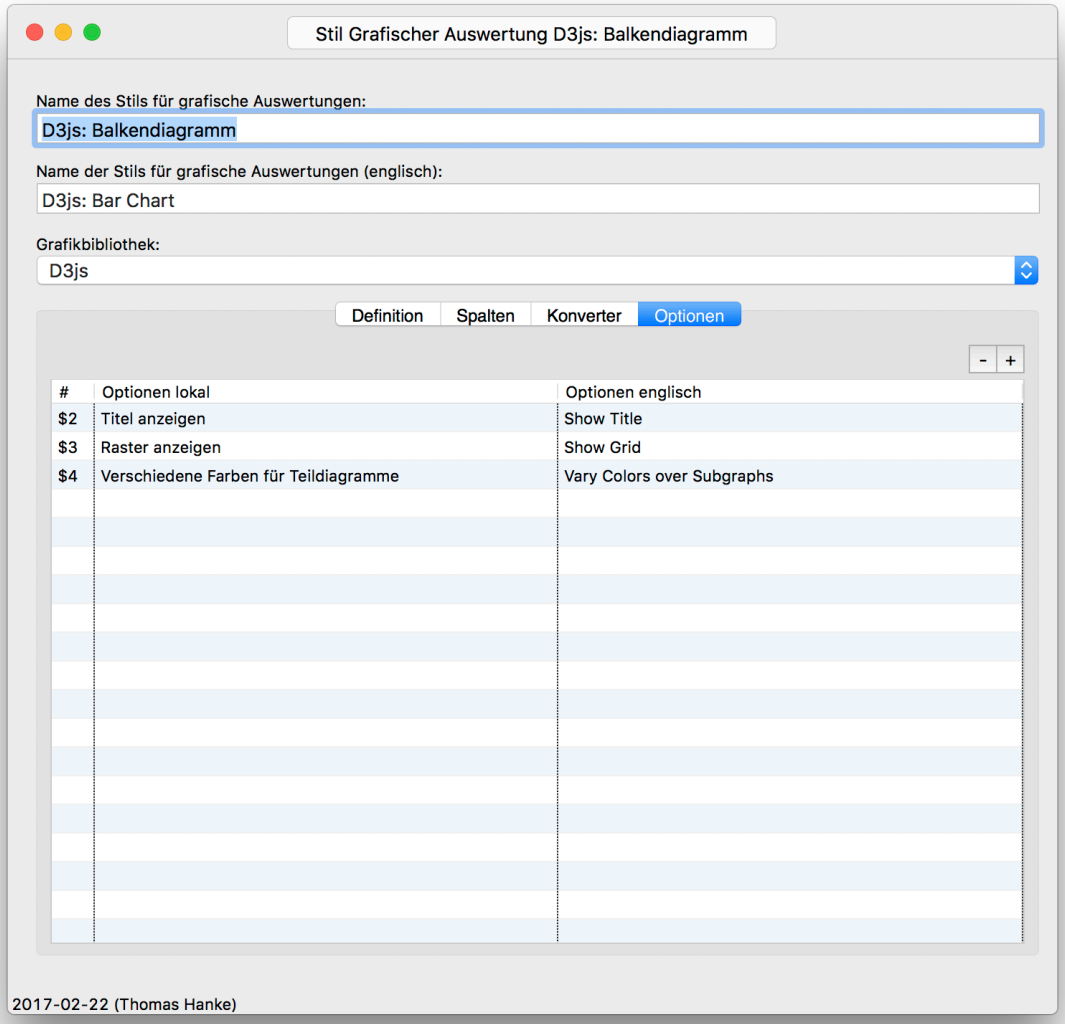
This creates the json data string using the field names as set up in the columns definition and expected by the Javascript source code without further complicating the SQL statement to be defined in the chart.

Here is the definition of the bar chart style referred to in the earlier example:




Stil Grafischer Auswertung D3js: Balkendiagramm
Name des Stils für grafische Auswertungen:
D3js: Balkendiagramm
Name der Stils für grafische Auswertungen (englisch):
D3js: Bar Chart
Grafikbibliothek:
D3js
Definition
Spalten
Konverter
Optionen
-
+
Spaltenkopf lokal Spaltenkopf englisch
1 Zeitstempel, gleichzeitig Überschrift (t) Time Stamp and Header (t)
2 Name des Teildiagramms (x2) Subchart name (x2)
3 Datensegment (x1) Data Segment (x1)
4 Wert (y) Value (y)
5 Beschriftung Label
6 Tooltip Tooltip
7 Sprungziel (URL) Click Target (URL)
8 Farbe Color
2017-02-22 (Thomas Hanke)


Stil Grafischer Auswertung D3js: Balkendiagramm
Name des Stils für grafische Auswertungen:
D3js: Balkendiagramm
Name der Stils für grafische Auswertungen (englisch):
D3js: Bar Chart
Grafikbibliothek:
D3js
Definition
Spalten
Konverter
Optionen
select array_to_json(array_agg(tt)) from ($1) tt(t,x2,x1,y,label,tooltip,href,color)
2017-02-22 (Thomas Hanke)



Here is the complete HTML code for this example with comments interspersed:

```
<!DOCTYPE html>
<meta charset="utf-8">
```

As you can see, you can be rather sloppy with respect to HTML elements as we are dealing with a rather forgiving HTML renderer. The first part is the CSS definitions:

```
<style>

div.tooltip {
      position: absolute;
      text-align: center;
      min-width: 160px;
      min-height: 20px;
      padding: 2px;
      font: 12px sans-serif;
      background: lightsteelblue;
      border: 0px;
      border-radius: 8px;
      pointer-events: none;
}
```

```
h1 {
	text-align: center
}

text.legend {
	font: 8px sans-serif;
}

text.subheader {
	font: 12px sans-serif;
}

.grid line {
	stroke: lightgrey;
	opacity: 0.7;
}
.grid path {
	stroke-width: 0;
}

</style>
<body>
```

After the CSS, we have the Javascript, the meat of the HTML document. The first `script` element includes the d3js library, which iLex copies to the directory where our HTML will reside.

```
<script src="./d3.v4.min.js" charset="utf-8"></script>
<script>

var data=JSON.parse('$1');

var optionShowTitle=$2;
var optionShowGrid=$3;
var optionVaryColorsOverSubgraphs=$4;
```

This is where the chart data are filled in: $1 gets the (converted) result of the SQL query, the other variables reflect the option settings and are replaced by either 0 or 1.

```
if ((optionShowTitle==1) && (data[0].t!=undefined) && (data[0].t!="")) {
	d3.select("body").append("h1").text(data[0].t)
}
```

If the SQL query provides a title for the graph, and the option “Show Title” has been activated, an `h1` element with the tile is added to the document (as its first content).

```
var div = d3.select("body").append("div")
	.attr("class", "tooltip")
	.style("opacity", 0);
```

Now the (invisible) tooltip `div` is created and a reference to it stored.

```
var yMin=d3.min(data,function(d){return d.y});
if (yMin>0) {
	yMin=0
}
var yMax=d3.max(data,function(d){return d.y});
```

This computes the min and max for all y’s contained in the data.

```
var x2s=data.map(function(d){return d.x2});
x2s=x2s.filter(function(item, pos) { return x2s.indexOf(item) == pos; });
if (x2s.length==1) {
     optionVaryColorsOverSubgraphs=0
}
```

This computes the list of all distinct x2 values. If there is only one, there is no need to vary the colours over the subgraphs as there will be only one graph.

The colors function is set up to map from x2 values to distinct colors. d3js provides several color schemes. The script uses the 10-colors scheme if not more colors are needed as the colors here are more distinct than in the set of 20.

```
var colors = d3.scaleOrdinal((x2s.length<11) ? d3.schemeCategory10 :
d3.schemeCategory20).domain([0,x2s.length-1]);

var maxX1s=0;
x2s.forEach(function(x2) {
     var dataForX2=data.filter(function(d) { return d.x2==x2 });
     var x1s=dataForX2.map(function(d){return d.x1});
     x1s=x1s.filter(function(item, pos) { return x1s.indexOf(item) == pos; });
     maxX1s=d3.max([maxX1s,x1s.length]);
});
var margin = {top: 10, right: 20, bottom: 80, left: 80},
binWidth = 960 - margin.left - margin.right,
height = 500 - margin.top - margin.bottom;
if (x2s.length>1) {
margin.top=60
}
binWidth=d3.max([d3.min([100,binWidth/maxX1s]),10]);

var yScale = d3.scaleLinear()
.range([height,0])
.domain([yMin,yMax]);
```

As the subgraphs should all have the same bar width, we compute the maximum number of bars in a subgraph. From there, we compute the bar widths to be between 10 and 100 and hopefully to fit into the subgraph size.

For y, a d3js linear scaler is set up. For x1, this will be done later as the calculation needs to be done for each x2, as everything else below:

```
x2s.forEach(function(x2,i) {
     var dataForX2=data.filter(function(d) { return d.x2==x2 });

     var x1s=dataForX2.map(function(d){return d.x1});
     x1s=x1s.filter(function(item, pos) { return x1s.indexOf(item) == pos; });

     var width=binWidth*x1s.length + margin.left + margin.right;

     var x1Scale = d3.scaleLinear()
     .range([0,binWidth*x1s.length])
     .domain([0,x1s.length]);

     dataForX2.forEach(function(d) {
          if (d.color==undefined) {
               if (optionVaryColorsOverSubgraphs) {
                    d.color=colors(i)
               } else {
                    d.color="steelblue"
               }
          }
     });
```

For each bar, we need to determine the fill color if it is not part of the data. If the user wants to have different colors for the subgraphs, we use the color mapper defined earlier. Otherwise, we use one standard color.

Now we add the svg (for each subgraph) to the body of the HTML document. Then we use a trick to draw the grid if needed: We use a y axis with the tick marks as long as the svg is wide. By drawing the grid first, we make sure that the bars are in front of the grid lines.

```
var svg= d3.select("body")
          .append("svg")
          .attr("width", width + margin.left + margin.right)
          .attr("height", height + margin.top + margin.bottom)
          .append("g")
          .attr("transform", "translate(" + margin.left + "," + margin.top + ")");

     if (optionShowGrid) {
          svg.append("g")
          .attr("class","grid")
          .call(d3.axisLeft(yScale)
          .tickSize(-width+margin.left+margin.right,0,0)
          .tickFormat("")
          );
     }
```

Now to the bars themselves: They are simply rectangles. Note the d3js-specific construction with a `selectAll` on non-existing rectangles, and the data binding. This implicitly creates a loop over all elements of the data bound, in this case the x2-subset of the whole set. All the attributes then are then computed with functions accessing a single element of the data subset.

In the end, we add `mouseover` and `mouseout` handlers for the tooltips as well as a click handler to deal with target URLs.

```
     svg.selectAll("rect")
     .data(dataForX2)
     .enter().append("rect")
     .attr("class", "bar")
     .attr("cat", function(d,i) { return i;})
     .attr("transform", function(d,i) {
     return "translate(" + x1Scale(i) + "," + yScale(d.y) + ")"; })
     .attr("width", function(d) { return binWidth-1 ; })
     .attr("height", function(d) { return  (height-yScale(d.y)); })
     .attr("fill", function(d) { return d.color })
     .on("mouseover", tooltipIn)
     .on("mouseout", tooltipOut)
     .on("click",gotoTarget);
```

Finally, we add the axes (the real ones, this time) and, in case there is more than one graph, a subheader for each. The only complication here is that we want the labels to be rotated under the x axis. Setting the `text-anchor` to `end` makes the labels right-justified or better said, taking rotation into account, top-justified. A bit of position calculation makes nice rotated labels:

```
     svg.selectAll("legend")
     .data(dataForX2)
     .enter()
     .append("text")
     .attr("class","legend")
     .attr("dy", ".7em")
     .attr("text-anchor","end")
     .attr("transform", function(d,i) {
```

```
      return "translate(" + ((i+.5)*binWidth-4)+","+(height+3)+") rotate(-90)";})
      .text(function(d) { return ((d.label) ? d.label : d.x1); });

      svg.append("g")
      .attr("transform", "translate(0," + height + ")")
      .call(d3.axisBottom(x1Scale)
      .ticks(0));
      svg.append("g")
      .call(d3.axisLeft(yScale)
      .tickFormat(d3.format('.0f')));


      if ((optionShowTitle==1) && (x2)) {
            svg.append("text")
            .attr("class","subheader")
            .attr("x",20)
            .text(x2)
      }
})
```

What remains, is to define the event handlers already installed. They are pretty much self-explanatory. Note however how easy it is to define animated transitions: You just add a duration to a change.

```
function gotoTarget(bar) {
      var href=bar.href;
      if (href) {
            window.location.assign(href);
      }
}

function tooltipIn (bar) {
      var tt=bar.tooltip;
      if (tt) {
            div
            .transition()
            .duration(200)
            .style("opacity", .9);
            div.html(tt)
            .style("left", (d3.event.pageX+10) + "px")
            .style("top", (d3.event.pageY - 28) + "px");
      }
}

function tooltipOut(s) {
      div.transition()
      .duration(500)
      .style("opacity", 0);
}

</script>
```

Debugging d3js Charts and Chart Styles

When developing new chart styles, it is not unlikely that initially the chart does not produce the expected results. Unfortunately, if there is an error in the Javascript code, you only see an empty chart window, not giving you any clue what went wrong.
In that case, it is easy to move the code to an excellent debugging environment for Javascript, Cascading Stylesheets and the HTML Document Object Model, namely your favourite Browser.
When iLex is asked to render a d3js chart, it creates the HTML document from the HTML template and the SQL code and stores that HTML in a temporary directory before asking the

in-app browser to load that file. So if you open exactly that file with a standard browser, you have all the debugging facilities of your browser at your hands. Here is how you find the document:
The path to the temporary folder/directory for applications depends on your operating system. (On MacOS, it is particularly tricky to find the temp folder used. The easiest way is to open a Terminal window and to enter

```
open $TMPDIR/TemporaryItems
```

The finder will open the Temporary Items folder.)
iLex numbers the HTML files it creates for each charting command it executes. So open the newest file iLex*nn*.html with your browser.
Probably, the window will remain white here as well, but you activate the debugging facilities, e.g. show the runtime errors that occurred, set breakpoints etc. to see what went wrong.

Converting Maps into TopoJSON

AP04-2013-01 showed the way to convert map data from different formats into R "spatical polygons dataframes" that are used with the R chart renderer which was the only way to produce maps within iLex before d3js became available. If you have such a dataframe holding your map definition, it is easy to convert this to a map json format that is easy to handle in the d3js context. We recommend using TopoJSON and show the way to create TopoJSON data from your R map data:
First make sure that your R installation has the geojson package installed. If not, use

```
install.packages("geojson")
```

and load your data:

```
load("/Users/Shared/geodata/subregions.RData")
subregions1<-spTransform(subregions,CRS("+proj=longlat +datum=WGS84"))
```

Then save your dataframe (here named `subregions1`) to a file:

```
geojson_write(subregions1,file="subregions.geojson")
```

There are several utilities available to convert from geojson to topojson. If you do not want to install software for this purpose locally, you find a number of services on the web, like *The Destillery* at http://shancarter.github.io/distillery/.

Here is the HTML template for a simple map chart style. It allows the user to choose (via the options) between four different maps[3]. The regions of the map have predefined colors and labels. But if you provide a y value for each region, the map is colored from yellow (for minimal y's) to red (for maximum y values).

```
<!DOCTYPE html>
<meta charset="utf-8">
<style>

/* CSS goes here. */
.subunit.sh { fill: #ddc; }
.subunit.hb { fill: #cdd; }
.subunit.hh { fill: #cdc; }
```

[3] While these TopoJSON map data strings are lengthy, you can even include several maps in your Javascript code without effecting speed noticably.

```
.subunit.goe { fill: #dcd; }
.subunit.ber { fill: #ddc; }
.subunit.mvp { fill: #cdd; }
.subunit.mst { fill: #cdc; }
.subunit.koe { fill: #dcd; }
.subunit.fra { fill: #ddc; }
.subunit.stu { fill: #cdd; }
.subunit.nue { fill: #cdc; }
.subunit.mue { fill: #ddc; }
.subunit.lei { fill: #cdd; }

.subunit.s01 { fill: #ddc; }
.subunit.s02 { fill: #cdd; }
.subunit.s03 { fill: #cdc; }
.subunit.s04 { fill: #dcd; }
.subunit.s05 { fill: #ddc; }
.subunit.s06 { fill: #cdd; }
.subunit.s07 { fill: #cdc; }
.subunit.s08 { fill: #dcd; }
.subunit.s09 { fill: #ddc; }
.subunit.s10 { fill: #cdd; }
.subunit.s11 { fill: #cdc; }
.subunit.s12 { fill: #ddc; }
.subunit.s13 { fill: #dcd; }
.subunit.s14 { fill: #cdc; }
.subunit.s15 { fill: #cdd; }
.subunit.s16 { fill: #dcd; }


.subunit-boundary {
fill: none;
stroke: #fff;
}

.subunit-label {
font-family: Sans-Serif;
fill: #555;
fill-opacity: 1;
font-size: 10px;
font-weight: 300;
text-anchor: middle;
}

div.tooltip {
position: absolute;
text-align: center;
min-width: 160px;
min-height: 20px;
padding: 2px;
font: 12px sans-serif;
background: lightsteelblue;
border: 0px;
border-radius: 8px;
pointer-events: none;
}

h1 {
text-align: center
}

</style>
<body>
<script src="http://d3js.org/d3.v4.min.js" charset="utf-8"></script>
<script src="http://d3js.org//topojson.v2.min.js"></script>
```

The TopoJSON library is provided by iLex in the same way as the d3js library.

```
<script>
```

```
var data=JSON.parse('$1');

var optionPoliticalMap=$2;
var optionDetailedMap=$3;
var optionShowLabels=$4;
var optionShowLegend=$5;
var optionIncludeZeroInScale=$6;
var optionShowTitle=$7;

var units;

if (optionPoliticalMap) {
     if (optionDetailedMap) { /* counties */
          units=JSON.parse('{"type":"Topology",…}')
     } else { /* states */
          units=JSON.parse('{"type":"Topology",…}')
     }
} else {
     if (optionDetailedMap) { /* data collection subregions */
          units=JSON.parse('{"type":"Topology",…}')
     } else { /* data collection regions */
          units=JSON.parse('{"type":"Topology",…}')
     }
}
```

One of the four TopoJSON string representations is loaded into units.

```
var values = data.map(function(d) { return d.y });
values=values.filter(function(d) { return typeof d === "number" });
var min=d3.min(values)
var max=d3.max(values)
if (optionIncludeZeroInScale) {
     if (min>0) min=0;
     if (max<0) max=0;
}
var colors = d3.scaleLinear().range(["#dd6", "#d00"]).domain([min,max]);

if ((optionShowTitle==1) && (data[0].t!=undefined) && (data[0].t!="")) {
     d3.select("body").append("h1").text(data[0].t)
}

function subunitid(unit) {
     var myValue=data.filter(function(d) {return d.x==unit.id});
     if (myValue.length==0) {
          return "subunit-boundary";
     } else {
          myValue=myValue[0].y;
     }
     if (myValue==false) {
          return "subunit-boundary";
     } else if (optionPoliticalMap) {
          if (optionDetailedMap) {
               return "subunit s"+unit.id.replace(/\d\d\d$/,"");
          } else {
               return "subunit s"+unit.id;
          }
     } else {
          if (optionDetailedMap) {
               return "subunit "+unit.id.replace(/\d+/,"");
          } else {
               return "subunit "+unit.id;
          }
     }
}

function subunitcolor(unit) {
     var myValue=data.filter(function(d) {return d.x==unit.id});
     if (myValue.length==0) {
          return "fill:cornsilk";
```

```
	} else {
		myValue=myValue[0].y;
	}
	if (typeof myValue === "number") {
		return "fill: "+colors(myValue);
	} else if ((myValue==false) || (myValue==true)) {
		return "";
	} else {
		return "fill: "+myValue;
	}
}

function textlabel(unit) {
	var dd;
	var myValue=data.filter(function(d) {return d.x==unit.id});
	if (myValue.length==0) {
		return "";
	} else {
		dd=myValue[0].label;
		myValue=myValue[0].y;
	}
	if (myValue==false) {
		return "";
	} else if (dd) {
		return dd;
	} else if (optionShowLabels) {
		return unit.properties.Name;
	} else {
		return "";
	}
}

function tooltip(unit) {
	var myValue=data.filter(function(d) {return d.x==unit.id});
	if (myValue.length==0) {
		myValue="";
	} else {
		myValue=myValue[0].tooltip;
	}
	if (myValue) {
		return myValue;
	} else {
		return textlabel(unit)
	}
}

function gotoTarget(unit) {
	var myValue=data.filter(function(d) {return d.x==unit.id});
	if (myValue.length==0) {
		myValue="";
	} else {
		myValue=myValue[0].target;
	}
	if (myValue) {
		window.location.assign(myValue);
	}
}

function relocator(unit) {
	var c=path.centroid(unit);
	var c1=[0,0];
	if (unit.id=="goe") {
		return "translate(300,360)";
	} else if (unit.id=="12") {
		return "translate(520,260)";
	} else if (unit.id=="ber1") {
		c1=[-1,-5];
	} else if (unit.id=="ber2") {
		c1=[1,5];
	} else if (unit.id=="fra3") {
		c1=[0,20];
```

```
     } else if (unit.id=="mue2") {
          c1=[0, 10];
     } else if (unit.id=="koe3") {
          c1=[0, 10];
     } else if (unit.id=="hb2") {
          c1=[6,-20];
     } else if (unit.id=="mue4") {
          c1=[1,-20];
     } else if (unit.id=="stu2") {
          c1=[-5,10];
     }
     var dd="translate(" + (c[0]+c1[0]) +","+ (c[1]+c1[1]) + ")";
     return dd;
}
```

Since we have not implemented svg collision detection, we use the relocator function to move labels a bit off the region's centre in order to avoid overlaps.

```
function tooltipIn (unit) {
     var t=tooltip(unit);
     if (t) {
          div.transition()
          .duration(200)
          .style("opacity", .9);
          div  .html(tooltip(unit))
          .style("left", (d3.event.pageX) + "px")
          .style("top", (d3.event.pageY - 28) + "px");
     }
}

function tooltipOut(d) {
     div.transition()
     .duration(500)
     .style("opacity", 0);
}

var insertLinebreaks = function (d) {
     var el = d3.select(this);
     var words = d3.select(this).text().split("||");
     el.text('');

     for (var i = 0; i < words.length; i++) {
          var tspan = el.append('tspan').text(words[i]);
          if (i > 0) {
               tspan.attr('x', 0).attr('dy', '10');
          }
     }
};
```

In order to have a longer text distributed over several lines, you need to split the svg text element into tspan elements.

```
var width = 660,
height = 875;

var projection=d3.geoMercator()
.center([10.45,51.4])
.scale(4000)
.translate([width / 2, height / 2])
;

var subunits = topojson.feature(units, units.objects.regions);
var path = d3.geoPath()
.projection(projection)
;
```

```
// Define the div for the tooltip
var div = d3.select("body").append("div")
.attr("class", "tooltip")
.style("opacity", 0);

var svg = d3.select("body").append("svg")
.attr("width", width)
.attr("height", height)
;

svg.selectAll(".subunit")
.data(subunits.features)
.enter()
.append("path")
.attr("class", subunitid)
.attr("d",path)
.attr("style",subunitcolor)
.on("mouseover", tooltipIn)
.on("mouseout", tooltipOut)
.on("click",gotoTarget)
;

svg.append("path")
.datum(topojson.mesh(units,units.objects.regions,function(a,b){ return a !== b; }))
.attr("d", path)
.attr("class", "subunit-boundary")
;

svg.selectAll(".subunit-label")
.data(subunits.features)
.enter().append("text")
.attr("class", function(d) { return "subunit-label s" + d.id; })
.attr("transform", relocator)
.attr("dy", ".35em")
.text(textlabel)
.on("mouseover", tooltipIn)
.on("mouseout", tooltipOut)
.on("click",gotoTarget)
;

svg.selectAll("text").each(insertLinebreaks);

if (optionShowLegend && (min<max)) {
     var defs = svg.append("defs");
     var linearGradient = defs.append("linearGradient")
     .attr("id", "linear-gradient")
     .attr("x1",0)
     .attr("x2",0)
     .attr("y1",1)
     .attr("y2",0);

     linearGradient.selectAll("stop")
     .data(colors.range())
     .enter().append("stop")
     .attr("offset", function(d,i) { return i/(colors.range().length-1); })
     .attr("stop-color", function(d) { return d; });

     var legendWidth = 300;

     //Color Legend container
     var legendsvg = svg.append("g")
     .attr("class", "legendWrapper")
     .attr("transform", "translate(580,700)");

     //Draw the Rectangle
     legendsvg.append("rect")
     .attr("class", "legendRect")
     .attr("y", -legendWidth/2)
     .attr("x", 0)
     .attr("ry", 8/2)
     .attr("height", legendWidth)
```

```
      .attr("width", 8)
      .style("fill", "url(#linear-gradient)");

      //Set scale for axis
      var xScale = d3.scaleLinear()
      .range([legendWidth/2, -legendWidth/2])
      .domain([min, max]);

      //Define axis
      var xAxis = d3.axisRight()
      .ticks(5)
      .tickFormat(function(d) { return d ; })
      .scale(xScale);

      //Set up axis
      legendsvg.append("g")
      .attr("class", "axis")
      .attr("transform", "translate(0,0)")
      .call(xAxis);
}

</script>
```

The last code block is used to show a nice legend if a color range is used so that the reader can interpret the different colors used. Except for the gradient, this is rather close to the axis code given in the earlier example.

References

- AP04-2013-01: Data Visualisation in iLex.
- Hanke, Thomas (2016): “Towards a Visual Sign Language Corpus Linguistics”. In: Efthimiou, Eleni et. al. (eds.): Workshop Proceedings. 7th Workshop on the Representation and Processing of Sign Languages: Corpus Mining. Language Resources and Evaluation Conference (LREC), Portorož, Slovenia, 28 May 2016. ELRA. pp. 89-92. (Available online: http://www.sign-lang.uni-hamburg.de/dgs-korpus/files/inhalt_pdf/LREC2016-Hanke.pdf; last access: 2016-05-31.)