D3: Data Driven Documents

Ottmar Gobrecht, Linde AG

D3 ist eine JavaScript-Bibliothek zum Manipulieren von HTML-Dokumenten auf Basis von Daten und setzt dabei auf die Web-Standards HTML, SVG und CSS. Dieser Artikel zeigt die Grundlagen von D3 und weckt das Interesse am Thema „Datenvisualisierung mit Web-Standards". Vorab nur so viel: Auch D3 kennt den Begriff des Joins.

D3 kennt alle üblichen Charts und beim Anblick der Startseite unter „d3js.org" wird einem klar, dass es wohl keine Grenzen bei der Umsetzung zu geben scheint. Deshalb hier der unvollständige Versuch, D3 etwas genauer zu definieren:

- D3 ist eine JavaScript-Entwicklerbibliothek, die es erlaubt, Daten mit grafischen Elementen (SVG) im Browser zu kombinieren und diese so auf vielfältige Art und Weise zu visualisieren
- Durch seinen Data-Join-Ansatz ist D3 im Verhältnis zu anderen Bibliotheken auch bei großen Datenmengen sehr schnell
- Aufgrund der Unterstützung von Webstandards ist D3 zukunftssicher und auch auf mobilen Geräten lauffähig
- D3 basiert auf HTML5 und CSS3, es benötigt also einen modernen Browser (Firefox, Chrome, Safari, IE9 aufwärts)
- D3 ist keine Chart-Engine, bei der man fertige Layouts auswählt und konfiguriert

Das Fundament: SVG

SVG steht für „Scalable Vector Graphics" und basiert auf XML. Die zweidimensionalen SVG-Zeichnungselemente haben gegenüber Raster-Grafiken den Vorteil, verlustfrei skaliert werden zu können. Ihnen fehlt also der berühmte Treppenstufen-Effekt, der bei extremer Vergrößerung in Raster-Formaten auftritt. SVG im Browser wurde im Rahmen von HTML5 standardisiert.

Im Gegensatz zu HTML-Elementen, die im Grunde nur die Rechteckform kennen, gibt es bei SVG-Elementen Pfade und zur einfacheren Handhabung alle grafischen Grundformen wie Kreis, Ellipse, Rechteck, Linie und Polygon. Zusätzlich kann man noch Text und externe Rastergrafiken verwenden. Alle genannten Elemente können durch das Gruppenelement zusammengefasst werden.

Elemente und Gruppen von Elementen können über Style-Attribute in ihrer Erscheinung angepasst und über Transformationen in Position, Orientierung und Form verändert werden. An Transformationen stehen Parallelverschiebung, Rotation, Skalierung und Scherung zur Verfügung, die über CSS3-Transitions auch in einem vorgegebenen Zeitraum verändert werden können. Daneben gibt es noch Animationen sowie grafische Effekte und Filter. Wer sich tiefergehender dafür interessiert, findet im Web eine Menge Informationen und Beispiele.

```
<!DOCTYPE html><html><head><title>SVG Beispiel</title></head><body>
<script src="http://d3js.org/d3.v3.min.js"></script>

<svg style="width:490; height:140;">
   <rect x="10" y="10" height="120" width="160" fill="#ff6600"/>
   <ellipse cx="270" cy="70" rx="120" ry="40" fill="green"/>
   <line x1="40" y1="40" x2="480" y2="120" stroke="blue"/>
</svg>

<script>
   d3.select('svg').append('circle')
       .attr('cx','420')
       .attr('cy','70')
       .attr('r','60')
       .style('stroke','red')
       .style('fill','lightsteelblue')
       .style('fill-opacity',0.5);
</script>

</script></body></html>
```

Listing 1: Ein einfaches SVG-Beispiel

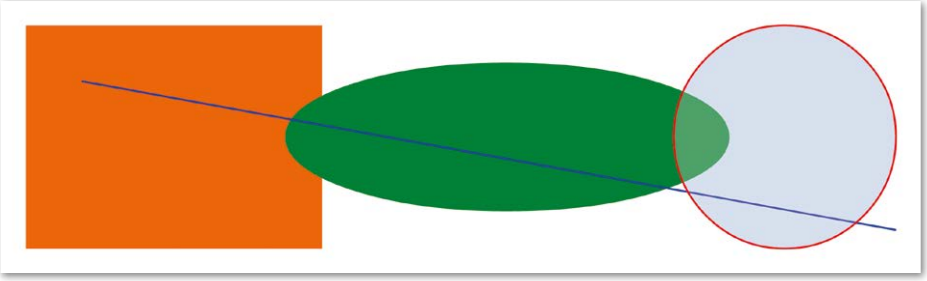

Abbildung 1: Das Ergebnis aus Listing 1

```
// Schleife über alle Elemente: JavaScript
var p = document.getElementsByTagName(‚p‘);
for (var i = 0; i < p.length; i++) {
   var pi = p.item(i);
   pi.style.setProperty(‚color‘,‘red‘, null);
}

// Schleife über alle Elemente: D3
d3.selectAll(‚p‘).style(‚color‘,‘red‘);
```

Listing 2: JavaScript versus D3

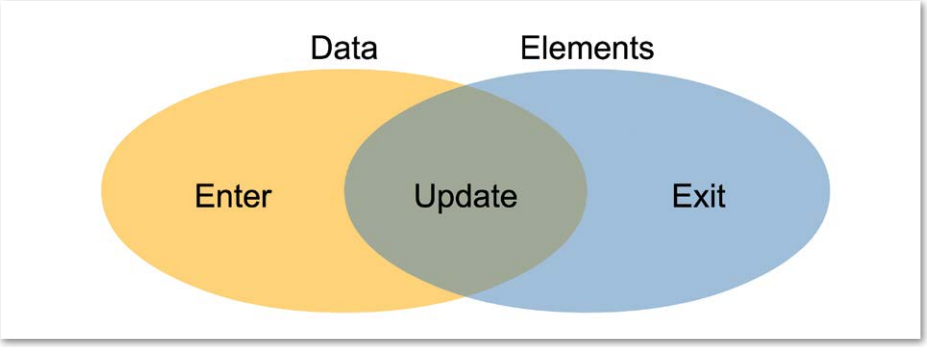

Abbildung 2: Der D3-Data-Join

Man kann SVG-Grafiken auf mehreren Wegen in HTML einbinden: als externe Grafik oder direkt im HTML mit dem Tag „<svg>“. Bei einer externen Grafik ist wie bei XML der Namespace zu definieren. Bei direkter Einbindung in HTML ist das nicht erforderlich – was im Zusammenhang mit D3 den Standardfall darstellt. Bis auf die ungewohnten Tag- und Attribut-Namen wird einem also nicht viel auffallen, wenn man sich eine HTML-Seite mit eingebettetem SVG im DOM-Inspector eines modernen Browsers anschaut. Insofern ist SVG recht pflegeleicht und man kann die gewohnten Werkzeuge des Browsers nutzen.

Wer jQuery kennt, wird sich schnell in D3 zurechtfinden. D3 unterstützt auch die Methodenverkettung; die Basismethoden wie „append“ und „attr“ haben die gleichen Namen oder zumindest ähnliche wie „style“ bei D3 und „css“ bei jQuery. *Listing 1* zeigt ein Beispiel, drei SVG-Elemente direkt in HTML und ein Element prozedural mit D3 *(siehe Abbildung 1)*.

Wie *Listing 1* zeigt, heißt bei SVG das Attribut für die Füllfarbe „fill“ und für die Konturfarbe „stroke“. Der Bezugspunkt für das Positionieren von Elementen ist die linke obere Ecke des SVG-Elements. Das gilt auch beim Rechteck-Element; beim Kreis und bei der Ellipse ist der Bezugspunkt hingegen die Mitte.

Selectors versus Selections

In JavaScript kann man mit den aus CSS bekannten Selektoren Elemente im Document Object Model (DOM) auswählen. Um dann Änderungen an den Elementen vorzunehmen, muss man mit einer Schleife über alle Elemente iterieren. D3 verfolgt einen anderen Ansatz, wie man schon an der Bezeichnung erkennen kann – hier wird von „Selections“ gesprochen. Jede Selection ist ein Array, auch wenn es nur ein oder gar kein Element enthält. Wenn man dieses dann mit einer Methode aufruft, iteriert D3 – wie übrigens jQuery auch – automatisch über alle Elemente. *Listing 2* zeigt den Vergleich.

Daten an das DOM binden

Um Daten an das DOM binden zu können, müssen diese erst einmal zur Verfügung stehen. D3 stellt für das Lesen von Daten XHR-Methoden („XMLHttpRequest“) zur Verfügung. Damit können Daten per JavaScript nachgeladen werden. Der Vorteil dieses Vorgehens ist, dass sich das Laden der eigentlichen Webseite bei großen Datenmengen nicht verzögert.

Zur einfacheren Handhabung gibt es darauf aufbauend vier Wrapper-Methoden, die für das Laden von CSV-, TSV-, XML- und JSON-Daten genutzt werden können. Häufig kommt es aber auch vor, dass man diese auch „AJAX-Calls“ genannten Aufrufe mit den Methoden des jeweils benutzten Entwicklungs-Frameworks er-

```
<!DOCTYPE html><html><head><title>Data Join</title></head><body>
<script src="http://d3js.org/d3.v3.min.js"></script>

<p style="color:green;">Ein bereits existierendes Element</p>

<script>

   var body = d3.select('body');

   var p = body.selectAll('p').data([1,2,3])
       .style('color', 'red');
       // http://bost.ocks.org/mike/join/
       // http://bost.ocks.org/mike/circles/

   p.enter().append('p')
       .text( function(d){return ‚Neues Element aus Daten , + d;} );

   p.exit().remove();

</script></body></html>
```

Listing 3: Data Join

Ein bereits existierendes Element
Neues Element aus Daten 2
Neues Element aus Daten 3

Abbildung 3: Das Ergebnis aus Listing 3

ledigt, wobei sich das Datenformat JSON durchgesetzt hat.

Bei Apex gibt es zum Beispiel fertige Methoden in den mitgelieferten JavaScript-Bibliotheken und bei kleineren Datenmengen mag es auch ok sein, diese direkt in die Seite zu rendern. Wege gibt es viele. Am Ende braucht man JavaScript-Arrays, die die Daten enthalten und über die D3 iterieren kann. Wie die Zeilen beziehungsweise Objekte des Arrays aussehen, hängt stark vom Anwendungsfall ab.

Nun zur eigentlichen Frage: D3 bindet Daten an das DOM, indem man einer Selection über den Data Operator die Daten in Form eines Arrays übergibt. Hier kommt der eingangs erwähnte Data Join zum Einsatz. Zurückgeliefert werden drei neue Selections, die D3 „Update", „Enter" und „Exit" nennt. *Abbildung 2* zeigt schematisch die Zusammenhänge.

In der Update Selection sind die Elemente des DOM mit den Elementen des Data Arrays in der Reihenfolge ihres jeweiligen Erscheinens verknüpft. Wie man unschwer errät, kann jedoch eine unterschiedliche Anzahl von Elementen im DOM und in den Daten vorhanden sein. Wenn im DOM mehr Elemente vorhanden sind als Daten, dann liegen diese Elemente in der Exit Selection, mit der man die Elemente aus dem DOM entfernen kann. Gibt es mehr Daten als DOM-Elemente, dann befinden sich diese Daten in der Enter Selection, mit der die fehlenden Elemente im DOM erstellt werden können. *Listing 3* zeigt ein Beispiel und *Abbildung 3* das Ergebnis.

Wie man in *Listing 3* sieht, steht die Update Selection sofort in der Methoden-Verkettung zur Verfügung und sie hat keinen eigenen Namen. Es wäre aber auch möglich, die Enter oder Exit Selection zuerst aufzurufen. Wichtig ist zu wissen, dass nach Abarbeitung der ersten Selection die anderen Selections gesondert aufgerufen werden müssen – eine weitere Verkettung ist nicht möglich. Außerdem stehen einem die Elemente der Enter Selection nach Ausführung in der Update Selection zur Verfügung. Man könnte also beispielsweise zuerst die neuen Elemente anlegen, ohne weitere Attribute oder Styles zu definieren, und dann mit einem Aufruf der Update-Selection alle Elemente auf einmal aktualisieren.

Für den Fall, dass der einfache Data Join über die Reihenfolge der Elemente in DOM und Array nicht genügt, gibt es eine sogenannte „Key Function", anhand derer die Elemente im DOM identifiziert werden. Die anonyme Funktion wird bei Bedarf als zweiter Parameter dem Data Operator übergeben. *Listing 4* zeigt ein

```
<!DOCTYPE html><html><head><title>Data Key Function</title></head><body>
<script src="http://d3js.org/d3.v3.min.js"></script><script>

  var data = [
      {"id":1, "x":40,  "y":40},
      {"id":2, "x":120, "y":40},
      {"id":3, "x":200, "y":40}];

  var svg = d3.select('body').append('svg');

  var circle = svg.selectAll('circle')
      .data(data, function(d){return d.id;});
      // Wie all das funktioniert: http://bost.ocks.org/mike/selection/

  circle.exit().remove();

  circle.enter().append('circle')
      .attr('id', function(d) { return d.id; })
      .attr('r', 10);

       circle
      .attr('cx', function(d) { return d.x; })
      .attr('cy', function(d) { return d.y; });

</script></body></html>
```

Listing 4: Data Key Function

Beispiel einer Data-Key-Funktion – hier wird zur Identifikation die ID verwendet. Wer wissen möchte, wie das alles ganz genau funktioniert, der findet unter dem Data Join im Listing einen Link, in dem der Autor von D3 das sehr ausführlich erklärt.

Das Geheimnis der Geschwindigkeit

Die Konsequenz aus dem Vorgehen von D3 ist, dass man immer zuerst eine Selection erstellt, auch wenn man weiß, dass im DOM keine Elemente vorhanden sind. An die Selection werden die Daten gebunden und im Falle einer statischen Visualisierung reicht es dann aus, über die Enter Selection die DOM-Elemente zu erstellen.

Im Falle von dynamischen Visualisierungen spielt dieses Vorgehen sein Stärken aus. Ändern sich nicht alle Daten, muss D3 nur die neuen Elemente im DOM anlegen und alte entfernen; vorhandene können aktualisiert werden. Dies spart Ressourcen im Browser, da nur die nötigste Arbeit zu tun ist, anstatt alle Elemente aus dem DOM zu entfernen, um sie dann wieder neu anzulegen.

Außerdem speichert sich D3 in den Selections die Referenzen auf die DOM-Elemente; dadurch muss für weitere Updates des DOM keine weitere Elementsuche auf dem DOM ausgeführt werden. Nun ist es gelüftet, das Geheimnis hinter der hohen Geschwindigkeit – D3 verhindert ganz einfach unnötige Arbeit für den Browser, womit Rechenkapazität für das Bearbeiten größerer Datenmengen frei wird.

Chart-Code wiederverwenden

Der Schwerpunkt von D3 liegt nicht auf dem Ausliefern von fertigen Chart-Funktionen, in die man einfach seine Daten kippt. D3 konzentriert sich auf grundlegende, immer wieder benötigte Dinge und bietet dafür generisch verwendbare Hilfs-Funktionen an. Man muss zwar Arbeit in seine Charts stecken, hat aber durch den generischen Ansatz der Basis-Funktionen praktisch keine Einschränkungen bei der Umsetzung, wie man an den vielen Beispielen im Internet sehen kann.

Man kommt am Anfang mit diesen Beispielen schnell zu ansehnlichen Ergebnissen. Spätestens wenn man mit dem gleichen Code einen zweiten Chart auf derselben HTML-Seite einbauen möchte, sieht man jedoch die Grenzen: Globale Variablen und Funktionen sind schnell redeklariert und ein zweiter Chart stört die Funktion des ersten. Hier ist es dann an der Zeit, über Kapselung und Wiederverwendung von Chart-Code zu sprechen. Dazu muss man wissen, wie JavaScript mit

Closures in JavaScript

In JavaScript ist alles ein Objekt und Funktionen können Objekte, also auch Funktionen, zurückgeben. Eine Closure ist am Ende nichts anderes als ein Funktionsaufruf, bei dem eine Funktion zurückgegeben wird. Die zurückgegebene Funktion behält ihren vorherigen Kontext, kann also auf die vor der Rückgabe vorhandenen Variablen innerhalb ihrer Ursprungsfunktion zugreifen. Sie kapselt quasi ihre Umgebung, daher der Name. Die Folge eines solchen Vorgehens ist, dass man dadurch private Variablen erhält, die von außen nicht einsehbar sind. Um trotzdem an diese Variablen zu kommen, muss man dann entsprechende Get- und Set-Funktionen definieren. *Listing 5* zeigt die Prinzipien. *Abbildung 4* zeigt das Ergebnis aus *Listing 5* und kann einfach in der JavaScript-Konsole des Browsers nachvollzogen werden.

```
<!DOCTYPE html><html><head><title>Prinzip Closure</title></head><body>
<script>

   function myChart() {

       var conf = { "width": 600, "height": 400 };

       function chart(){/*create chart with conf*/}

       chart.render = function(){
           chart();
           return chart;
       };

       chart.width = function(value) {
           if (!arguments.length) return conf.width;
           conf.width = value;
           return chart;
       };

       return chart;
   }

</script></body></html>
```

Listing 5: Prinzip Closure

```
Console
> my = myChart().render();
  function chart(){/*create chart with conf*/}
> my.width();
  600
> my.width(500);
  function chart(){/*create chart with conf*/}
> my.width();
  500
```

Abbildung 4: Test Closure in Browser-Konsole

Variablen und Funktionen umgeht. Das Stichwort hier sind Closures *(siehe Kasten)*.

Wenn man nun noch wie in *Listing 5* bei diesen Get- und Set-Funktionen wiederum die Funktion selbst zurückliefert, dann hat man die Basis für eine Methoden-Verkettung geschaffen. Wie man sieht, lohnt es sich, ein wenig Arbeit in ein kleines API zu stecken – als Gegenleistung bekommt man universell einsatzbare Charts, die je nach Gegebenheit initialisiert und zur Laufzeit angepasst werden können, um zum Beispiel auf User-Interaktionen zu reagieren.

Die Königsklasse: Physik-Simulationen

D3 bietet für verschiedene Anwendungsfälle sogenannte Layouts an. Das sind im Grunde die angesprochenen wiederverwendbaren Charts, aber auf einem generischen Level. Wir gehen das an dem Beispiel eines Force-Layouts durch, einer beliebten, kräftebasierten Darstellung von Netzwerken.

Die Besonderheit einer kräftebasierten, selbstorganisierenden Netzwerk-Visualisierung ist die dahinterliegende physikalische Simulation. D3 berechnet fortwährend die Positionen der Netzwerk-Knoten. Damit der Eindruck einer Animation entsteht, muss man natürlich die Knoten seines Graphen irgendwann einmal neu positionieren. Diese Arbeit nimmt einem D3 nicht ab. D3 bietet aber ein sogenanntes „Tick Event", das ungefähr alle zwanzig Millisekunden feuert. Wenn man dann bei jedem Tick-Event alle Knoten und Verbindungen des Netzwerks an die jeweils aktuelle Position der physikalischen Simulation verschiebt, bekommt man den Eindruck einer Animation. Die Animation des Netzwerks hört dann auf, wenn sich die Simulation in einem kräftemäßig ausgewogenen Zustand befindet.

Listing 6 zeigt ein minimales, vollständig lauffähiges Skript für ein Force Layout. Zuerst die Daten – sie dürften den meisten bekannt sein. Als Knoten nehmen wir die Mitarbeiter der EMP-Tabelle, als Links die Beziehung zum Vorgesetzten. Die Links beziehen sich auf die Knoten in der Reihenfolge ihres Erscheinens, es wird also keine Key Function eingesetzt, um die

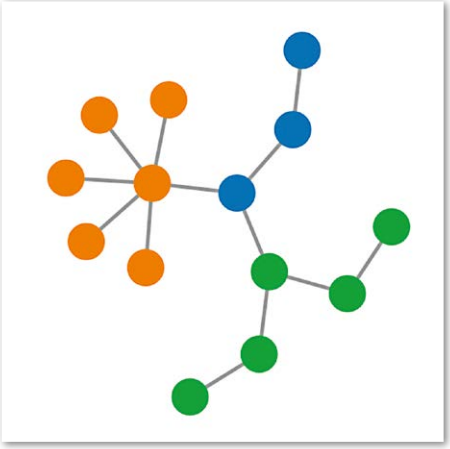

Abbildung 5: Das Ergebnis aus Listing 6

```
<!DOCTYPE html><html><head><title>Force Layout</title></head><body>
<script src="http://d3js.org/d3.v3.min.js"></script><script>

   var nodes = [
       {"name":"King","dept":10},{"name":"Blake","dept":30},
       {"name":"Clark","dept":10},{"name":"Jones","dept":20},
       {"name":"Scott","dept":20},{"name":"Ford","dept":20},
       {"name":"Smith","dept":20},{"name":"Allen","dept":30},
       {"name":"Ward","dept":30},{"name":"Martin","dept":30},
       {"name":"Turner","dept":30},{"name":"Adams","dept":20},
       {"name":"James","dept":30},{"name":"Miller","dept":10}
   ];

   var links = [
       {"source":1,"target":0},{"source":2,"target":0},
       {"source":3,"target":0},{"source":7,"target":1},
       {"source":8,"target":1},{"source":9,"target":1},
       {"source":10,"target":1},{"source":12,"target":1},
       {"source":13,"target":2},{"source":4,"target":3},
       {"source":5,"target":3},{"source":6,"target":5},
       {"source":11,"target":4}
   ];

   var width = 600, height = 400;

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

   var color = d3.scale.category10();

   var force = d3.layout.force().size([width,height]);

   var link = svg.selectAll("line").data(links)
       .enter().append("line").style('stroke','#999');

   var node = svg.selectAll("circle").data(nodes)
       .enter().append("circle")
           .attr("r", 5)
           .style("fill", function(d){ return color(d.dept); })
           .call(force.drag);

   force.on("tick", function(){
       link.attr("x1", function(d){ return d.source.x; })
           .attr("y1", function(d){ return d.source.y; })
           .attr("x2", function(d){ return d.target.x; })
           .attr("y2", function(d){ return d.target.y; });

       node.attr("cx", function(d){ return d.x; })
           .attr("cy", function(d){ return d.y; });
   });

   force.nodes(nodes).links(links).start();

</script></body></html>
```

Listing 6: Force Layout

Knoten zu identifizieren. Dies ist in Ordnung, solange keine Updates auf den Graphen erfolgen sollen, ansonsten wird es einem schwer fallen, die Knoten im DOM zu identifizieren.

Will man zum Beispiel die Knoten in Farbe und Größe passend zu den Daten gestalten, kann man auf die vielen Hilfsfunktionen von D3 zurückgreifen. Als Beispiel sei hier die Farbe der Knoten genannt. In *Listing 6* kann man erkennen, wie die Variable „color" mit der Funktion „d3.scale.category10()" belegt ist. Diese Funktion liefert eine aus zehn verschiedenen Farben für den übergebenen Wert zurück. Wir benutzen dies dafür, um über die Abteilungsnummer eines Mitarbeiters eine Farbe zurückzubekommen – somit haben alle Mitarbeiter einer Abteilung die gleiche Knotenfarbe, ohne dass wir dafür etwas programmieren müssen.

Nach der Variable „color" wird eine Variable „force" mit dem entsprechenden Layout belegt. Dann folgt schon die Variable „link". Hier werden alle Linien im SVG-Element selektiert – in unserem Fall ergibt das ein leeres Array, weil wir ja wissen, dass wir zum ersten Mal den Graphen erstellen.

Dann folgt in derselben Zeile der Data Join mit unseren Links und gleich darauf wird in der Enter Selection für jeden Eintrag in unserem Array „link" eine Linie im SVG-Element gezeichnet. Das Ganze wiederholen wir jetzt für die Knoten. Daraufhin definieren wir noch für das Tick Event die bereits erwähnte Funktion, die dann jeweils unsere Knoten und Links innerhalb des SVG-Elements positionieren. Man kann hier schön erkennen, wie D3 jeweils implizit über das Array „link" und „node" iteriert und für jeden Eintrag in der darauf definierten anonymen Funktion als ersten Parameter die Daten des Eintrags übergibt, die einem dann für die weitere Verarbeitung zur Verfügung stehen. In unserem Fall geben wir einfach nur die jeweilige Position für den Link oder den Knoten zurück.

Als letzter Schritt wird noch die eigentliche physikalische Berechnung gestartet, die natürlich unsere beiden Knoten und Link Arrays benötigt, um vernünftig arbeiten zu können. Als Ergebnis erhalten wir den Graphen aus *Abbildung 5*, der schon ein wenig an den ersten Graphen unter dem Artikelbild erinnert.

Wie man sieht, muss man oft nur minimal Code erstellen, um Charts an seine eigenen Bedürfnisse anzupassen. Und sollte man doch mal etwas Komplizierteres benötigen, so kann man davon ausgehen, im Internet viele Beispiele für die gleiche oder eine ähnliche Problemstellung zu finden.

Netzwerke: ein Plug-in (nicht nur) für Apex

Wer sich dafür interessiert, was man noch so alles machen muss, um vom eben gezeigten Minimalbeispiel zu der großen Lösung aus dem Artikelbild zu kommen, findet eine vollständige Implementierung eines Netzwerk-Charts als Apex-Plug-in auf GitHub. Allen Nicht-Apex-Entwicklern sei gesagt, dass trotz seines Namens nicht zwingend Apex erforderlich ist.

Wie im Artikel empfohlen, existiert ein vollständiges JavaScript-API. Darüber hinaus gibt es einen interaktiven Customize Wizard und die Chart-Funktion stellt bei nicht vorhandenen Daten selbstständig Beispieldaten zur Verfügung. Mit diesen Voraussetzungen kann man nach dem Einbinden der benötigten Source-Dateien sofort loslegen, Parameter verändern und live die Auswirkungen testen.

Fazit

Mit D3 stehen einem eine Menge Wege offen, eigene Charts zu erstellen. Man muss zwar zuerst ein wenig Arbeit investieren, bekommt aber bei überlegter Umsetzung wiederverwendbaren Code und hat praktisch keine Einschränkungen in der Umsetzung.

Weitere Informationen

1. http://d3js.org
2. https://github.com/ogobrecht/d3-force-apex-plugin
3. https://apex.oracle.com/pls/apex/GERMAN_COMMUNITIES.SHOW_TIPP?P_ID=3481
4. http://www.apex-plugin.com/search-plugin-directory (nach D3 suchen)

Ottmar Gobrecht
ottmar.gobrecht@linde.com

ADF-Buch der deutschen ADF-Community erschienen

Der Wunsch nach Wissensaustausch stand an erster Stelle – herausgekommen ist ein umfangreiches Referenzwerk zum Thema „Oracle ADF". Rund 1.400 Seiten ist das Kompendium stark, das die deutsche ADF-Community zusammengestellt hat. Es steht kostenfrei als PDF-Dokument unter *„http://stream.doag.org/adf/ADF-Buch.pdf"* zum Download bereit.

Für das ADF-Buch haben die Autoren zahlreiche Artikel, Vorträge und Workshop-Unterlagen einer Vielzahl von Referenten zusammengestellt. Die Leser bekommen somit das gesammelte Entwicklungs-Know-how der letzten fünf Jahre und einen breiten, praxisorientierten Querschnitt durch alle ADF-Themengebiete geboten.

Neben den Grundlagen umfasst das Nachschlagewerk zahlreiche Lernbeispiele sowie Erfahrungsberichte zu konkreten Beispielen. Ein Überblick über Architektur, Oracle Mobile Application Framework (MAF) und zahlreiche Spezialthemen wie beispielweise „ADF mit Eclipse" und „Migration Forms zu ADF" vervollständigen das ADF-Buch.

Robert Szilinski, Leiter der DOAG Development Community, resümiert begeistert die Zusammenarbeit aller Beteiligten: „Alle haben sich mit sehr großem Engagement in dieses Buch eingebracht und waren bereit, ihr Wissen und ihre Erkenntnisse im Sinne einer Community untereinander zu teilen. Das ist wirklich fantastisch und es zeigt den einzigartigen Spirit, der sich inzwischen in der Community entwickelt hat."