

# D3.js Container Visualisierung



```
{
  "contId": "cont4",
  "subs": [
    {
      "contId": "cont2",
      "subs": [{"contId": "cont4"}]
    },
    {
      "contId": "cont3",
      "subs": [{"contId": "cont5"}]
    }
  ]
}
```

Erstelle mir bitte auf Basis eines JSON Konstrukts wie oben zu sehen ist (beachte dass die subs Verschachtelungen auch länger sein können) einen JavaScript Code mit Draggable Graph (d3.js) welcher die verbindungen dieser Container ids (contid) darstellt. Bitte beachte folgendes:

- > pro 2 Container sollte nur eine Verbindung sein
- > es darf keine ContainerId doppelt vorkommen
- > es darf auch alleinstehende Ids (also ohne Verbindung) geben
- > alle notwendigen Codestücke in einem File!
- > das Json file heißt "graphSpecific.json" und befindet sich im selben Verzeichnis wie das File



Hier ist ein JavaScript-Code, der D3.js verwendet, um ein interaktives, dragbares Graphen-Diagramm zu erstellen, das die Verbindungen zwischen den Container-IDs darstellt. Der Code geht davon aus, dass du D3.js korrekt installiert hast und dass die Datei `graphSpecific.json` im selben Verzeichnis wie das JavaScript-File liegt.

Stelle sicher, dass du D3.js über ein CDN in deinem HTML einbindest, falls du das nicht bereits getan hast.

## HTML + JavaScript-Code

html

```
<!DOCTYPE html>
<html lang="de">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Draggable Graph</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    .node {
      fill: #1f77b4;
```

```

        cursor: pointer;
    }
    .link {
        fill: none;
        stroke: #ccc;
        stroke-width: 1.5px;
    }
    .node text {
        font: 12px sans-serif;
        pointer-events: none;
        fill: #fff;
    }
}
</style>
</head>
<body>
    <svg width="960" height="600"></svg>

    <script>
        // Asynchrones Laden der JSON-Datei
        d3.json('graphSpecific.json').then(function(graphData) {
            // Verbindungen und Knoten vorbereiten
            const nodes = [];
            const links = [];
            const nodeMap = {}; // Um doppelte IDs zu vermeiden

            // Funktion zur Rekursion der Container-Struktur
            function traverse(subs, parentId) {
                subs.forEach(sub => {
                    // Wenn der Container noch nicht im nodes-Array ist
                    if (!nodeMap[sub.contId]) {
                        nodes.push({ id: sub.contId });
                        nodeMap[sub.contId] = true;
                    }
                    // Verbindung zwischen parentId und sub.contId
                    links.push({ source: parentId, target: sub.contId });
                    // Rekursion für tiefer verschachtelte Subs
                    if (sub.subs && sub.subs.length > 0) {
                        traverse(sub.subs, sub.contId);
                    }
                });
            }

            // Startknoten ist der Root-Container
            if (graphData.subs) {
                traverse(graphData.subs, graphData.contId);
            }

            // Alle Knoten, die ohne Verbindung sind, als unabhängige Knoten hinzufügen
            Object.keys(nodeMap).forEach(id => {
                if (!nodes.find(node => node.id === id)) {
                    nodes.push({ id });
                }
            });

            // SVG und D3-Setup
            const svg = d3.select('svg');
            const width = +svg.attr('width');
            const height = +svg.attr('height');

            const simulation = d3.forceSimulation(nodes)
                .force('link', d3.forceLink(links).id(d => d.id).distance(100))
                .force('charge', d3.forceManyBody().strength(-200))
                .force('center', d3.forceCenter(width / 2, height / 2));

            // Links erstellen
            const link = svg.append('g')
                .selectAll('.link')
                .data(links)
                .enter().append('line')
                .attr('class', 'link');

            // Knoten erstellen
            const node = svg.append('g')
                .selectAll('.node')
                .data(nodes)

```

```

        .enter().append('g')
        .attr('class', 'node')
        .call(d3.drag()
            .on('start', dragStarted)
            .on('drag', dragged)
            .on('end', dragEnded));

// Knotenkreis und Text hinzufügen
node.append('circle')
    .attr('r', 20)
    .attr('class', 'node');

node.append('text')
    .attr('dy', -25)
    .attr('text-anchor', 'middle')
    .text(d => d.id);

// Simulation tickt und aktualisiert Positionen der Knoten und Verbindungen
simulation.on('tick', function() {
    link
        .attr('x1', d => d.source.x)
        .attr('y1', d => d.source.y)
        .attr('x2', d => d.target.x)
        .attr('y2', d => d.target.y);

    node
        .attr('transform', d => `translate(${d.x},${d.y})`);
});

// Drag-Verhalten
function dragStarted(event, d) {
    if (!event.active) simulation.alphaTarget(0.3).restart();
    d.fx = d.x;
    d.fy = d.y;
}

function dragged(event, d) {
    d.fx = event.x;
    d.fy = event.y;
}

function dragEnded(event, d) {
    if (!event.active) simulation.alphaTarget(0);
    d.fx = null;
    d.fy = null;
}
}).catch(function(error) {
    console.log(error);
});
</script>
</body>
</html>

```

## Erklärung:

1. **Laden der JSON-Daten:** Die JSON-Datei `graphSpecific.json` wird mit `d3.json()` geladen.
2. **Rekursion der `subs`:** Die Funktion `traverse` wird verwendet, um die verschachtelten `subs` zu durchlaufen und alle Knoten und Verbindungen zu extrahieren.
3. **Verbindungen und Knoten:** Verbindungen zwischen Container-IDs werden erstellt. Es wird sichergestellt, dass Container-IDs nicht doppelt vorkommen.
4. **D3.js Visualisierung:** Ein SVG wird verwendet, um die Knoten (Container) als Kreise und die Verbindungen als Linien darzustellen. Knoten sind interaktiv und können per Drag & Drop bewegt werden.
5. **Simulation:** D3's `forceSimulation` sorgt dafür, dass die Knoten dynamisch angeordnet und die Verbindungen entsprechend den physikalischen Kräften positioniert werden.

## Wie man den Code verwendet:

1. Speichere den HTML-Code in einer Datei (z.B. `index.html`).
2. Stelle sicher, dass die Datei `graphSpecific.json` im selben Verzeichnis liegt.
3. Öffne die HTML-Datei in einem Browser, um das interaktive Diagramm zu sehen.



```
Error: node not found: cont2
    at Yc (d3.v7.min.js:2:102583)
    at h (d3.v7.min.js:2:232576)
    at Function.initialize (d3.v7.min.js:2:233061)
    at p (d3.v7.min.js:2:236157)
    at Object.force (d3.v7.min.js:2:236803)
    at DraggableGraph.html?_ijt=ia7o5h25rk8v2s1ksgm9fku96g:71:14
```



Der Fehler "node not found: cont2" tritt auf, weil D3 die Knoten in den Links referenziert, bevor sie tatsächlich im `nodes`-Array existieren. Dies passiert, weil in der Rekursionsfunktion zwar die Verbindungen zwischen den Containern erzeugt werden, aber die Knoten für diese Container nicht direkt in das `nodes`-Array eingefügt werden.

Um den Fehler zu beheben, müssen wir sicherstellen, dass alle Container, auf die durch Verbindungen verwiesen wird, auch tatsächlich im `nodes`-Array vorhanden sind. Wir sollten also die Knoten und Verbindungen gleichzeitig in der Rekursion hinzufügen, ohne dass ein Container referenziert wird, bevor er in das `nodes`-Array aufgenommen wurde.

Hier ist der aktualisierte Code, der dieses Problem behebt:

## HTML + JavaScript-Code (mit Fix)

```
html

<!DOCTYPE html>
<html lang="de">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Draggable Graph</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    .node {
      fill: #1f77b4;
      cursor: pointer;
    }
    .link {
      fill: none;
      stroke: #ccc;
      stroke-width: 1.5px;
    }
    .node text {
      font: 12px sans-serif;
      pointer-events: none;
      fill: #fff;
    }
  </style>
</head>
```

```

<body>
  <svg width="960" height="600"></svg>

  <script>
    // Asynchrones Laden der JSON-Datei
    d3.json('graphSpecific.json').then(function(graphData) {
      // Verbindungen und Knoten vorbereiten
      const nodes = [];
      const links = [];
      const nodeMap = {}; // Um doppelte IDs zu vermeiden

      // Funktion zur Rekursion der Container-Struktur
      function traverse(subs, parentId) {
        subs.forEach(sub => {
          // Wenn der Container noch nicht im nodes-Array ist
          if (!nodeMap[sub.contId]) {
            nodes.push({ id: sub.contId });
            nodeMap[sub.contId] = true;
          }
          // Verbindung zwischen parentId und sub.contId
          links.push({ source: parentId, target: sub.contId });
          // Rekursion für tiefer verschachtelte Subs
          if (sub.subs && sub.subs.length > 0) {
            traverse(sub.subs, sub.contId);
          }
        });
      }

      // Der Root-Container (cont4) wird als Startpunkt genutzt
      nodes.push({ id: graphData.contId });
      nodeMap[graphData.contId] = true;

      // Die Verschachtelung der Subs des Root-Containers verarbeiten
      if (graphData.subs) {
        traverse(graphData.subs, graphData.contId);
      }

      // Alle Knoten, die ohne Verbindung sind, als unabhängige Knoten hinzufügen
      Object.keys(nodeMap).forEach(id => {
        if (!nodes.find(node => node.id === id)) {
          nodes.push({ id });
        }
      });

      // SVG und D3-Setup
      const svg = d3.select('svg');
      const width = +svg.attr('width');
      const height = +svg.attr('height');

      const simulation = d3.forceSimulation(nodes)
        .force('link', d3.forceLink(links).id(d => d.id).distance(100))
        .force('charge', d3.forceManyBody().strength(-200))
        .force('center', d3.forceCenter(width / 2, height / 2));

      // Links erstellen
      const link = svg.append('g')
        .selectAll('.link')
        .data(links)
        .enter().append('line')
        .attr('class', 'link');

      // Knoten erstellen
      const node = svg.append('g')
        .selectAll('.node')
        .data(nodes)
        .enter().append('g')
        .attr('class', 'node')
        .call(d3.drag()
          .on('start', dragStarted)
          .on('drag', dragged)
          .on('end', dragEnded));

      // Knotenkreis und Text hinzufügen
      node.append('circle')
        .attr('r', 20)

```

```

        .attr('class', 'node');

    node.append('text')
        .attr('dy', -25)
        .attr('text-anchor', 'middle')
        .text(d => d.id);

    // Simulation tickt und aktualisiert Positionen der Knoten und Verbindungen
    simulation.on('tick', function() {
        link
            .attr('x1', d => d.source.x)
            .attr('y1', d => d.source.y)
            .attr('x2', d => d.target.x)
            .attr('y2', d => d.target.y);

        node
            .attr('transform', d => `translate(${d.x},${d.y})`);
    });

    // Drag-Verhalten
    function dragStarted(event, d) {
        if (!event.active) simulation.alphaTarget(0.3).restart();
        d.fx = d.x;
        d.fy = d.y;
    }

    function dragged(event, d) {
        d.fx = event.x;
        d.fy = event.y;
    }

    function dragEnded(event, d) {
        if (!event.active) simulation.alphaTarget(0);
        d.fx = null;
        d.fy = null;
    }
}).catch(function(error) {
    console.log(error);
});
</script>
</body>
</html>

```

## Wichtige Änderungen:

1. **Root-Container hinzugefügt:** Der `graphData.contId`` (z.B. `cont4``) wird nun als Knoten explizit zum `nodes``-Array hinzugefügt, bevor wir mit den `subs`` weiterarbeiten.
2. **Rekursion angepasst:** Die Funktion `traverse`` sorgt dafür, dass alle Container, die durch die `subs`` referenziert werden, korrekt im `nodes``-Array erscheinen.
3. **Verbindungen erstellt:** Für jedes `sub`` wird eine Verbindung mit dem übergeordneten Container (`parentId``) hergestellt.

Dieser Code sollte jetzt ohne den Fehler "node not found" laufen und dir das gewünschte dragbare Graph-Diagramm der Container-IDs anzeigen.



Mach bitte die Schriftfarbe und Knotenfarbe schwarz!  
 Ausserdem stelle bitte sicher, dass der gesamte Platz im Browser benutzt werden kann.  
 Außerdem möchte ich, dass es möglich ist, wenn man auf einen Knoten klickt (dass dieser und sein Verbindungen) farblich hervorgehoben werden, wobei die Knoten mit denen der ausgewählte Verbindungen hat in einer anderen Farbe hervorgehoben wird



Hier ist der aktualisierte Code mit den angeforderten Änderungen:

1. **Schrift- und Knotenfarbe auf Schwarz.**
2. **Das Diagramm nutzt den gesamten Platz im Browser.**
3. **Klick auf einen Knoten hebt den Knoten und seine Verbindungen farblich hervor:**
  - Der ausgewählte Knoten wird in einer Farbe (z. B. Blau) hervorgehoben.
  - Die Knoten, die mit ihm verbunden sind, werden in einer anderen Farbe (z. B. Grün) hervorgehoben.

## HTML + JavaScript-Code

html

```
<!DOCTYPE html>
<html lang="de">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Draggable Graph</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    body, html {
      margin: 0;
      padding: 0;
      overflow: hidden;
    }
    svg {
      width: 100vw;
      height: 100vh;
      background-color: #f9f9f9;
    }
    .node {
      fill: black;
      cursor: pointer;
    }
    .link {
      fill: none;
      stroke: #ccc;
      stroke-width: 1.5px;
    }
    text {
      font: 12px sans-serif;
      pointer-events: none;
      fill: black;
    }
    .highlight {
      fill: blue;
    }
    .connected {
      fill: green;
    }
    .highlight-link {
      stroke: orange;
      stroke-width: 2.5px;
    }
  </style>
</head>
<body>
  <svg></svg>

  <script>
    d3.json('graphSpecific.json').then(function(graphData) {
      const nodes = [];
      const links = [];
      const nodeMap = {};
```

```
function traverse(subs, parentId) {
  subs.forEach(sub => {
    if (!nodeMap[sub.contId]) {
      nodes.push({ id: sub.contId });
      nodeMap[sub.contId] = true;
    }
    links.push({ source: parentId, target: sub.contId });
    if (sub.subs && sub.subs.length > 0) {
      traverse(sub.subs, sub.contId);
    }
  });
}

nodes.push({ id: graphData.contId });
nodeMap[graphData.contId] = true;

if (graphData.subs) {
  traverse(graphData.subs, graphData.contId);
}

Object.keys(nodeMap).forEach(id => {
  if (!nodes.find(node => node.id === id)) {
    nodes.push({ id });
  }
});

const svg = d3.select('svg');
const width = window.innerWidth;
const height = window.innerHeight;

const simulation = d3.forceSimulation(nodes)
  .force('link', d3.forceLink(links).id(d => d.id).distance(100))
  .force('charge', d3.forceManyBody().strength(-300))
  .force('center', d3.forceCenter(width / 2, height / 2));

const link = svg.append('g')
  .selectAll('.link')
  .data(links)
  .enter().append('line')
  .attr('class', 'link');

const node = svg.append('g')
  .selectAll('.node')
  .data(nodes)
  .enter().append('g')
  .attr('class', 'node')
  .call(d3.drag()
    .on('start', dragStarted)
    .on('drag', dragged)
    .on('end', dragEnded))
  .on('click', highlightNode);

node.append('circle')
  .attr('r', 20)
  .attr('class', 'node');

node.append('text')
  .attr('dy', -25)
  .attr('text-anchor', 'middle')
  .text(d => d.id);

simulation.on('tick', function() {
  link
    .attr('x1', d => d.source.x)
    .attr('y1', d => d.source.y)
    .attr('x2', d => d.target.x)
    .attr('y2', d => d.target.y);

  node
    .attr('transform', d => `translate(${d.x},${d.y})`);
});

function dragStarted(event, d) {
  if (!event.active) simulation.alphaTarget(0.3).restart();
}
```



```

        d.fx = d.x;
        d.fy = d.y;
    }

    function dragged(event, d) {
        d.fx = event.x;
        d.fy = event.y;
    }

    function dragEnded(event, d) {
        if (!event.active) simulation.alphaTarget(0);
        d.fx = null;
        d.fy = null;
    }

    function highlightNode(event, selectedNode) {
        node.selectAll('circle').classed('highlight', false).classed('connected', false);
        link.classed('highlight-link', false);

        // Knoten hervorheben
        d3.select(event.currentTarget).select('circle').classed('highlight', true);

        // Verbindungen hervorheben
        link.filter(d => d.source.id === selectedNode.id || d.target.id === selectedNode.id)
            .classed('highlight-link', true);

        // Verbundene Knoten hervorheben
        node.filter(d =>
            links.some(link =>
                (link.source.id === selectedNode.id && link.target.id === d.id) ||
                (link.target.id === selectedNode.id && link.source.id === d.id)
            ))
            .select('circle')
            .classed('connected', true);
    }
}).catch(function(error) {
    console.log(error);
});
</script>
</body>
</html>

```

## Änderungen:

### 1. Schrift- und Knotenfarbe auf Schwarz:

- Die Standardfarben für `.node` und `text` wurden auf Schwarz gesetzt.

### 2. SVG auf vollen Browserplatz erweitert:

- Das SVG füllt den gesamten verfügbaren Platz im Browser mit `width: 100vw` und `height: 100vh`.

### 3. Klick-Highlight-Funktion:

- Beim Klick auf einen Knoten wird der ausgewählte Knoten in Blau hervorgehoben.
- Verbundene Knoten werden in Grün hervorgehoben.
- Die Links zwischen diesen Knoten werden orange hervorgehoben.

Dieser Code sollte die gewünschten Funktionen vollständig abdecken. Du kannst ihn speichern und testen!



Perfekt. Brich bitte nun den Code auf seine wichtigsten Bestandteile runter und erkläre diese in

einem Fließtext Format



Hier ist eine Erklärung der wichtigsten Bestandteile des Codes in Fließtextform:

## JSON-Verarbeitung und Datenstruktur

Die JSON-Datei `graphSpecific.json` enthält die Container-IDs und deren verschachtelte Verbindungen. Beim Laden der Datei wird eine Funktion `traverse` verwendet, um die hierarchische Struktur zu durchlaufen. Die Container werden dabei in zwei Arrays aufgeteilt:

1. `nodes`: Enthält alle Container als individuelle Knoten, basierend auf ihren IDs.
2. `links`: Enthält die Verbindungen zwischen Containern, dargestellt als Paare von Quell- und Ziel-Knoten (`source` und `target`).

Ein Hilfsobjekt `nodeMap` stellt sicher, dass keine Container-ID mehrfach in `nodes` aufgenommen wird.

## SVG-Element und D3-Setup

Ein `<svg>`-Element wird dynamisch mit D3 erstellt, um den Graphen darzustellen. Es nutzt den gesamten verfügbaren Platz im Browserfenster (`100vw` x `100vh`).

D3s **Kraft-gestützte Simulation** (`forceSimulation`) berechnet die Positionen der Knoten basierend auf folgenden Kräften:

- `forceLink`: Verbindet Knoten basierend auf den Links.
- `forceManyBody`: Erzeugt eine Abstoßung zwischen Knoten, damit sie nicht zu dicht beieinander liegen.
- `forceCenter`: Zentriert das gesamte Diagramm im SVG-Bereich.

## Knoten und Verbindungen

### 1. Verbindungen (Links):

- Jede Verbindung zwischen zwei Containern wird als Linie dargestellt.
- Die Linien werden aus den Daten in `links` generiert.

### 2. Knoten:

- Jeder Container wird als Kreis dargestellt.
- Jeder Kreis zeigt zusätzlich die Container-ID als Text oberhalb des Kreises an.

Die Knoten sind interaktiv gestaltet, sodass sie durch Ziehen (Drag-and-Drop) bewegt werden können. Dies wird mit D3s Dragging-API umgesetzt.

## Highlighting bei Klick auf einen Knoten

Beim Klick auf einen Knoten werden der Knoten selbst sowie seine Verbindungen hervorgehoben:

- Der angeklickte Knoten erhält eine **blaue Farbe**.
- Die mit ihm verbundenen Knoten erhalten eine **grüne Farbe**.

- Die Verbindungen zwischen diesen Knoten werden **orange markiert**.

Dies wird durch Filterfunktionen umgesetzt, die prüfen, ob eine Verbindung von oder zu dem ausgewählten Knoten führt. Die Klassen ``highlight``, ``connected``, und ``highlight-link`` steuern die Darstellung der markierten Elemente.

---

## Drag-and-Drop-Funktion

Die Knoten können per Drag-and-Drop verschoben werden.

- Beim Start des Ziehens wird die Simulation neu gestartet, um die Positionen dynamisch anzupassen.
  - Während des Ziehens wird die aktuelle Position des Knotens festgelegt.
  - Nach Beendigung des Ziehens wird die Simulation gestoppt, und der Knoten kann frei in seiner neuen Position verbleiben.
- 

## Zusammenfassung

Der Code kombiniert D3.js-Funktionalitäten, um ein interaktives, visuell ansprechendes Graph-Diagramm zu erstellen:

- JSON-Daten werden in Knoten und Verbindungen umgewandelt.
- Eine Simulation bestimmt die Positionen der Elemente.
- Nutzer können Knoten per Drag-and-Drop bewegen und Knoten sowie Verbindungen durch Klicken hervorheben.

Das Ergebnis ist ein flexibles, interaktives Diagramm, das die Beziehungen zwischen Containern auf einfache und effektive Weise visualisiert.

Ist dieses Gespräch bislang hilfreich?