03 Datenmanagement und Validierung

Ein robustes Datenmanagement ist essentiell für Nodges, da die Anwendung unterschiedlichste Graph-Strukturen verarbeiten muss. Dieses Kapitel erläutert die Strategie zur Datenhaltung, Validierung und Transformation.

03.1 Das Datenmodell: Legacy vs. Future

Nodges unterstützt historisch bedingt zwei verschiedene JSON-Datenformate. Die Architektur ist so ausgelegt, dass sie beide Formate nahtlos verarbeiten kann, wobei intern alles auf das moderne Format normalisiert wird.

Das Legacy-Format

Dies ist das ursprüngliche Format, das auf Einfachheit ausgelegt war. Es eignet sich hervorragend für schnelle Prototypen oder einfache Graphen.

Struktur:

- **nodes**: Ein flaches Array von Objekten mit `id`, `x`, `y`, `z` Koordinaten und optionalen Attributen.
- **edges**: Ein flaches Array von Verbindungen, definiert durch `start` und `end` (die auf Node-IDs verweisen).

```
{
  "nodes": [
    { "id": 1, "x": 0, "y": 0, "z": 0, "label": "Start" },
    { "id": 2, "x": 10, "y": 5, "z": -2, "label": "End" }
  ],
  "edges": [
    { "start": 1, "end": 2, "type": "connects_to" }
  ]
}
```

Das Future-Format

Das neue Format ist semantisch reicher und flexibler. Es trennt Strukturdaten von Metadaten und visuellen Definitionen. Es orientiert sich an modernen Graph-Datenbank-Exporten und Standards wie GEXF oder GraphML.

Struktur:

1. **data**: Enthält `entities` (Knoten) und `relationships` (Kanten).
2. **metadata**: Autor, Version, Beschreibung, Erstellungsdatum.
3. **dataModel**: Beschreibt das Schema der Eigenschaften (z.B. "weight ist eine Zahl zwischen 0 und 1").
4. **visualMappings**: Definiert Regeln, wie Daten auf visuelle Eigenschaften abgebildet werden (z.B. "Map 'weight' auf 'edge-thickness'").

```
{
  "system": "Network V2",
  "metadata": { "version": "2.0", "author": "User" },
  "data": {
    "entities": [
      { "id": "n1", "type": "server", "position": { "x": 0, "y": 0, "z": 0 } }
    ],
    "relationships": [
      { "source": "n1", "target": "n2", "type": "http_request" }
    ]
  }
}
```

03.2 Schema-Validierung mit Zod

Um die Integrität der importierten Daten zu gewährleisten, setzt Nodges auf **Zod**. Zod ist eine TypeScript-First Schema-Deklarations- und Validierungsbibliothek.

Warum Zod?

In JavaScript/TypeScript sind Daten, die von "außen" kommen (wie ein JSON-Import aus einer Datei), zur Laufzeit typ-unsicher (`any`). Ein einfacher Cast (`as GraphData`) täuscht Sicherheit vor, die nicht existiert. Zod löst dieses Problem durch strikte Laufzeit-Überprüfungen.

Schema-Definitionen (`types.ts`)

In `types.ts` werden Zod-Schemas definiert, die exakt beschreiben, wie valide Daten auszusehen haben.

Beispiel (vereinfacht):

```
const NodeSchema = z.object({
  id: z.union([z.string(), z.number()]),
  x: z.number(),
  y: z.number(),
  z: z.number(),
  // Optionale Felder
  label: z.string().optional()
});

const GraphSchema = z.object({
  nodes: z.array(NodeSchema),
  edges: z.array(EdgeSchema)
});
```

Automatische Typ-Generierung

Ein großer Vorteil von Zod ist, dass TypeScript-Interfaces direkt aus den Schemas abgeleitet werden können:

```
export type NodeData = z.infer<typeof NodeSchema>;
```

Dies garantiert, dass die Validierungslogik und die verwendeten TypeScript-Typen niemals auseinanderlaufen (**Single Source of Truth**).

Error-Handling

Wenn eine importierte Datei nicht dem Schema entspricht (z.B. fehlende Koordinaten, falsche Datentypen), wirft Zod detaillierte Fehler. Nodges fängt diese ab und zeigt dem Benutzer präzise Fehlermeldungen an (z.B. *"Fehler in Datei: In 'nodes[5]' fehlt das Feld 'z'"*), anstatt dass die Anwendung abstürzt oder undefiniertes Verhalten zeigt.

03.3 Der `DataParser`

Der `DataParser` (`src/core/DataParser.ts`) ist die zentrale Komponente, die Rohdaten in das interne Format der Applikation überführt.

Format-Erkennung

Beim Laden einer Datei analysiert der Parser zunächst die Struktur, um das Format zu bestimmen:

- Hat das Objekt `img` und `nodes` Properties? -> Legacy Format.
- Hat es `data.entities` Properties? -> Future Format.

Normalisierungs-Pipeline

Unabhängig vom Eingabeformat ist das Ziel der `App`, immer mit einheitlichen Datenstrukturen zu arbeiten.

1. **Legacy-Input**: Wird in temporäre Future-Strukturen konvertiert.
    - `nodes` -> `entities` (Typ wird auf 'default' gesetzt).
    - `edges` -> `relationships`.
2. **Future-Input**: Wird validiert und ggf. mit Default-Werten angereichert.
3. **ID-Management**: Da IDs Strings oder Numbers sein können, werden sie intern konsistent zu Strings normalisiert, um Lookup-Maps (`Map<string, Node>`) effizient nutzen zu können.

Konvertierung für visuelle Komponenten

Nach der Daten-Normalisierung bereitet der Parser die Daten für die `ObjectManager` auf. Er extrahiert Positionsdaten und Attribute, die für das Rendering relevant sind, und stellt sicher, dass keine "Dangling Edges" existieren (Kanten, die auf nicht existierende Knoten verweisen).

Ende Kapitel 03