Future Format & Types – Technische Dokumentation

1. Einleitung und Kontext

Diese Dokumentation analysiert die Implementierung der Typsicherheit in `types.ts` und beschreibt den strategischen Übergang des Nodges-Projekts zu einem semantisch reichhaltigen "Future Format". Sie dient als Leitfaden für Entwickler, um die aktuellen Datenstrukturen zu verstehen, die Rolle der Validierung (Zod) einzuordnen und zukünftige Erweiterungen wie Multi-Target-Edges und Ports zu planen.

---

2. Analyse der `types.ts`

Die Datei `types.ts` ist in logische Sektionen unterteilt, die den hybriden Zustand der Anwendung widerspiegeln.

2.1 Legacy Typen (`NodeData`, `EdgeData`)

Dies ist das Format, das aktuell von der Rendering-Engine (`NodeObjectsManager`, `EdgeObjectsManager`) erwartet wird.

```
export interface NodeData {
    id: string | number;
    x: number; y: number; z: number; // Explizite Koordinaten
    [key: string]: any;              // Erlaubt beliebige Zusatzdaten
}
```

- **Merkmal:** Stark auf die 3D-Darstellung fokussiert (`x, y, z` sind Pflicht).
- **Flexibilität:** Durch `[key: string]: any` extrem flexibel, aber unsicher. Es gibt keine Garantie, welche Eigenschaften existieren.

2.2 Future Format Types (`DataModel`, `VisualMappings`)

Hier beginnt die Abstraktion. Das Future Format trennt **Daten** strikt von der **Darstellung**.

Data Model Schema (`PropertySchema`, `EntityTypeSchema`)

Anstatt Daten einfach als JSON-Objekt zu laden, definiert das Future Format ein Schema für diese Daten.

```
export interface PropertySchema {
    type: 'continuous' | 'categorical' | 'vector' | 'spatial' | 'temporal';
    range?: [number, number];
    unit?: string;
    // ...
}
```

- **Idee:** Das System "weiß", was ein Wert bedeutet. Ein Wert ist nicht einfach eine Zahl `0.5`, sondern z.B. ein 'continuous' Wert zwischen 0 und 1, der eine Intensität darstellt.
- **Vorteil:** Dies ermöglicht eine automatisierte UI (z.B. Slider für `continuous`, Dropdowns für `categorical`).

Entity & Relationship Data

```
export interface EntityData {
    id: string;
    type: string;
    // Position ist hier optional! Das Framework könnte es berechnen.
    position?: { x: number; y: number; z: number };
    [key: string]: any;
}
```

- **Unterschied zu Legacy:** Ein `Entity` ist ein abstraktes Objekt. Es wird erst durch ein `VisualMapping` zu einem sichtbaren 3D-Objekt.

2.3 Visual Mappings

Dies ist der mächtigste Teil des Future Formats. Er definiert, wie Datenattribute in visuelle Eigenschaften übersetzt werden.

```
export interface VisualMapping {
    source: string;              // Welches Feld? z.B. "traffic_load"
    function: MappingFunction;   // Wie mappen? z.B. "heatmap"
    range?: [number, number];    // Zielbereich?
}
```

- **Anwendung:** Wenn `traffic_load` hoch ist, wird die Farbe Rot (via Heatmap-Funktion) oder die Größe skaliert. Im Code (`FutureDataParser.js`) sind Funktionen wie `sphereComplexity`, `pulse` oder `geographic` bereits angedacht.

---

3. Das Future Format: Nutzung, Vor- & Nachteile

Nutzungskonzept

Das Future Format zielt auf ein **"Connect Interface"** ab. Der Nutzer lädt rohe Daten (Entities/Relationships) und definiert *separat*, wie diese aussehen sollen.

1. **Daten laden:** JSON enthält reine Fakten (z.B. "Server A verbunden mit Server B, Latenz: 50ms").
2. **Mapping definieren:** "Latenz" entscheidet über die Dicke der Kante. "Server-Typ" entscheidet über die Farbe der Node.

Vor- und Nachteile

Aspekt	Legacy Format	Future Format
Simplicity	Hoch. Direkte `x,y,z` Werte, einfach zu parsen.	Niedrig. Erfordert Parsing- und Mapping-Schritte.
Semantik	Keine. Daten sind nur "Props".	Hoch. Datentypen sind bekannt (`spatial`, `temporal`).
Flexibilität	Begrenzt. Neue Visualisierungen erfordern Code-Änderungen.	Extrem hoch. Visualisierung ist Konfiguration, kein Code.
Trennung	Vermischt (Daten & View in einem Objekt).	Sauber getrennt (Datenmodell vs. VisualMappings).

Alternativen

- **Graphology:** Eine Standard-Bibliothek für Graphen in JS. Vorteil: Standardisiert. Nachteil: Weniger Fokus auf 3D-Visualisierungs-Mappings.
- **GEXF (Graph Exchange XML Format):** Ein XML-Standard. Nachteil: XML ist veraltet und mühsam in Web-Apps zu parsen. Das JSON-basierte Future Format ist moderner.

---

4. Exkurs: Zod - Sinn und Zweck

Im Projekt ist **Zod** als Dependency vorhanden, wird jedoch im aktuellen Kern-Code noch nicht voll ausgeschöpft.

Was ist Zod?

Zod ist eine TypeScript-First Schema-Deklarations- und Validierungsbibliothek.

Warum Zod im Future Format essentiell ist

Aktuell (siehe `core/DataParser.ts`) wird "Duck Typing" verwendet:

```
// Manuelle Prüfung - Fehleranfällig und verbos
if (data.nodes !== undefined || data.edges !== undefined) { ... }
```

Mit Zod definiert man das Schema einmal und erhält Validierung **und** Typen automatisch:

```
import { z } from "zod";

// Definition des Schemas
const EntitySchema = z.object({
  id: z.string(),
  type: z.string(),
  position: z.object({
    x: z.number(),
    y: z.number(),
```

```
    z: z.number()
  }).optional(),
}).passthrough(); // WICHTIG: Erlaubt unbekannte Eigenschaften!

// Automatische Validierung zur Laufzeit
const result = EntitySchema.safeParse(loadedJson);
if (!result.success) {
  console.error("Invalid format:", result.error);
}
```

Vorteile von Zod

1. **Runtime Safety:** TypeScript Typen verschwinden beim Kompilieren. Zod prüft das *tatsächliche* JSON zur Laufzeit.
2. **Single Source of Truth:** Man leitet den TypeScript-Typ aus dem Zod-Schema ab (`type Entity = z.infer<typeof EntitySchema>`).
3. **Parsen von unbekannten Strukturen:** Zod kann unbekannte Keys strippen oder behalten (`passthrough`), was für das dynamische Datenmodell von Nodges entscheidend ist.

---

5. Einbettung des JSON-Ladens im Code

Der aktuelle Datenfluss im Projekt ist wie folgt implementiert:

1. **Initialisierung (`App.ts`):** `loadDefaultData` oder `loadData(url)` ruft `fetch` auf und übergibt das Ergebnis an den Parser.
2. **Parsing (`core/DataParser.ts`):** Der Parser normalisiert Legacy- und Future-Daten in eine interne `GraphData` Struktur.
3. **Konvertierung für Rendering (`App.ts`):** Die sauberen `EntityData` Objekte werden temporär zurück in flache Legacy-Objekte (`x, y, z` auf Top-Level) konvertiert, damit `NodeObjectsManager` sie verarbeiten kann.
4. **Verarbeitung (`FutureDataParser.js`):** Enthält Logik für semantisches Mapping (`continuous` -> `heatmap`), die sukzessive in den Kern integriert werden soll.

---

6. Machbarkeitsanalyse: Unbekannte Eigenschaften

Inwiefern können eine unbekannte Anzahl Eigenschaften von Edges und Nodes gespeichert werden?

Das Problem

TypeScript mag statische Typen, JSON-Daten von Nutzern sind dynamisch (`customField_1`, `server_status`, etc.).

Die Lösung

1. **TypeScript:** Nutzung von Index Signatures in Interfaces: `[key: string]: any;`.
2. **Zod (Laufzeit):** Nutzung von `.passthrough()`. Das Schema validiert die *bekannten* Felder (id, type) streng, lässt aber alle zusätzlichen Felder passieren, ohne einen Fehler zu werfen.

Fazit: Es ist technisch absolut machbar. Der Konflikt wird durch die bewusste Konfiguration des Parsers (`passthrough`) gelöst.

---

7. Zukunftsszenarien: Erweiterung des Modells

7.1 One-to-Many Edges (Hyperedges)

Eine Edge könnte eine Liste von Targets haben (Broadcast/Multicast).

Erforderliche Änderungen in `types.ts`:

```
interface HyperEdgeData {
    id: string;
    source: string;
    targets: string[]; // Array statt einzelner String
    type: 'broadcast' | 'multicast';
}
```

Eine solche Änderung erfordert ein Update im `EdgeObjectsManager`, um solche Edges entweder als mehrere Linien oder als verzweigte Geometrie darzustellen.

7.2 Ports (Node -> Port -> Edge -> Port -> Node)

Nodes sind Komponenten mit Anschlüssen. Eine Kante geht von "Port A an Node 1" zu "Port B an Node 2".

Erforderliche Änderungen in `types.ts`:

```
interface Port {
    id: string;
    nodeId: string;
    localPosition: { x: number; y: number; z: number }; // Relativ zur Node
}

interface PortConnection extends RelationshipData {
    sourcePort: string; // Referenz auf Port ID statt nur Node ID
    targetPort: string;
}
```

Dies erfordert Anpassungen im Datenmodell (Ports als Sub-Entities) und im Rendering (Berechnung der Weltkoordinaten der Ports vor dem Zeichnen der Kanten).