EDIFACT-Nachrichten in der Energiewirtschaft: Entwickler-Einführung

Ein praktischer Leitfaden für die deutsche Marktkommunikation

---

□□ Für wen ist diese Einführung?

Du bist Entwickler:in und stehst vor deiner ersten EDIFACT-Nachricht aus der deutschen Energiewirtschaft? Du möchtest verstehen, was all diese kryptischen Zeichen bedeuten und wie du effizient damit arbeiten kannst? Dann bist du hier richtig!

Diese Einführung:

- □ Erklärt EDIFACT von Grund auf – kein Vorwissen nötig
- □ Zeigt praktische Beispiele mit dem `edifact-json-transformer`
- □ Vermittelt regulatorisches Wissen (GPKE, GeLi Gas, WiM, MaBiS)
- □ Gibt Best Practices für die tägliche Arbeit

Powered by: Willi Mako – die intelligente MaKo-Plattform

Open Source Tool: edifact-json-transformer

□□ Inhaltsverzeichnis

□□ Was ist EDIFACT und warum brauchen wir das?

Die Grundidee

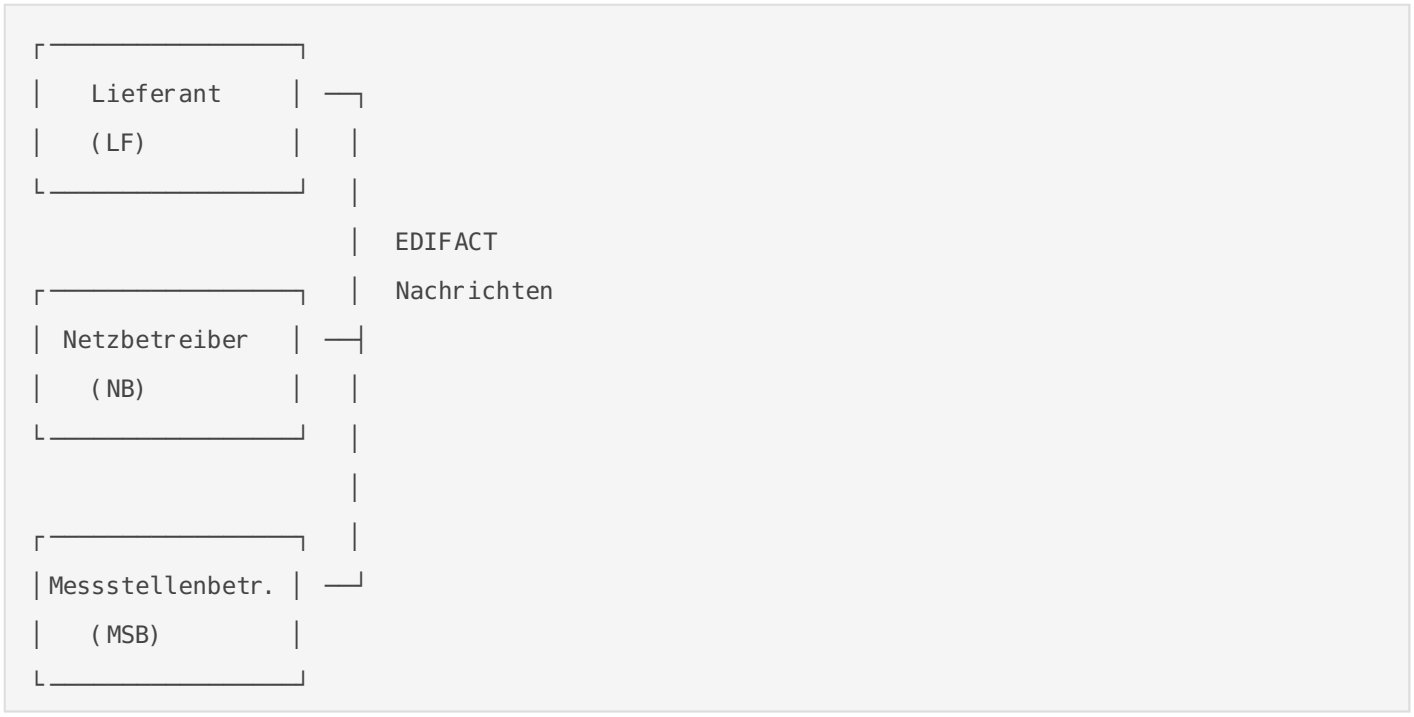
EDIFACT (Electronic Data Interchange For Administration, Commerce and Transport) ist ein UN-Standard für den elektronischen Datenaustausch. In der deutschen Energiewirtschaft ist dieser Standard **gesetzlich vorgeschrieben** und bildet das Rückgrat der Kommunikation zwischen allen Marktteilnehmern.

Rechtliche Grundlagen

Die Basis bildet das **Energiewirtschaftsgesetz (EnWG)**. Darauf aufbauend hat die **Bundesnetzagentur (BNetzA)** detaillierte Festlegungen erlassen, die jeden Aspekt der Marktkommunikation regeln:

Festlegung	Rechtsgrundlage	Was wird geregelt?
GPKE	Geschäftsprozesse Kundenbelieferung Elektrizität	Strom-Lieferantenwechsel, An-/Abmeldungen, Stammdaten
GeLi Gas	Geschäftsprozesse Lieferantenwechsel Gas	Gas-Lieferantenwechsel, Gasbelieferung
WiM	Wechselprozesse im Messwesen	Zählerwechsel, Messstellenbetrieb, Messdaten
MaBiS	Marktregeln Bilanzierung Strom	Bilanzkreisabrechnung, Ausgleichsenergie

Die Marktteilnehmer



Rollen:

- **Lieferant (LF)**: Versorgt Endkunden mit Energie, verantwortlich für Abrechnung
- **Netzbetreiber (NB)**: Betreibt das Netz, zentrale Kommunikationsstelle
- **Messstellenbetreiber (MSB)**: Betreibt Zähler, liefert Messdaten

---

□□ Die vier Säulen der Marktkommunikation

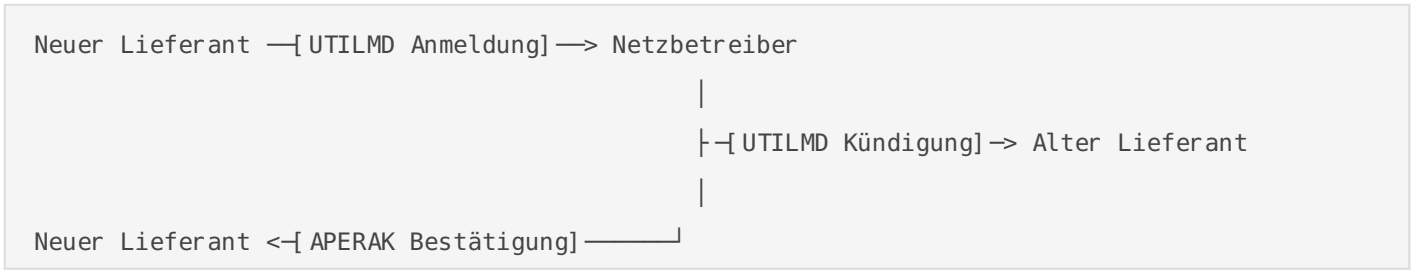
1. GPKE – Strom-Lieferantenwechsel

Kernprozesse:

- Anmeldung zur Netznutzung (Prüf-ID 44001)
- Abmeldung (Prüf-ID 44004)
- Kündigung beim bisherigen Lieferanten (Prüf-ID 44016)
- Stammdatenänderungen (Prüf-ID 44112, 44123)

Typische Nachrichten: UTILMD, MSCONS, INVOIC, APERAK

Beispiel-Workflow:



2. GeLi Gas – Gas-Lieferantenwechsel

Besonderheiten:

- Zusätzliche Brennwert- und Zustandszahl-Informationen
- Spezielle Gasbeschaffenheitsdaten in MSCONS
- ORDERS für Anfragen (z.B. nach Brennwerten)

Typische Nachrichten: UTILMD, MSCONS, ORDERS, ORDRSP, APERAK

3. WiM – Wechselprozesse im Messwesen

Kernprozesse:

- Gerätewechsel (Prüf-ID 17xxx-Serie)
- Messkonzeptwechsel (Prüf-ID 21xxx-Serie)
- Störungsmeldungen (INSRPT)
- Auftragsbestätigungen (IFTSTA)

Typische Nachrichten: ORDERS, ORDRSP, IFTSTA, INSRPT, MSCONS

Beispiel-Workflow:

4. MaBiS – Bilanzkreisabrechnung

Kernprozesse:

- Fahrplanmeldungen
- Bilanzkreiszuordnungen
- Mehr-/Mindermengenabrechnung

- Ausgleichsenergieabrechnung

Typische Nachrichten: MSCONS, PRICAT, PARTIN, COMDIS

□□ Anatomie einer EDIFACT-Nachric

Die Grundstruktur

Eine EDIFACT-Nachricht sieht zunächst kryptisch aus:

```
UNH+1+UTILMD:D:11A:UN:2.6'BGM+E01+12345+9'DTM+137:20241019:102'NAD+MS+9900123456789::293'RFF+Z13:44001'IDE+24+12345678901'UNT+6+1'
```

Zerlegt ergibt sich:

```
UNH+1+UTILMD:D:11A:UN:2.6'     ← Nachrichtenkopf
BGM+E01+12345+9'                ← Dokumentkopf
DTM+137:20241019:102'           ← Datum
NAD+MS+9900123456789::293'      ← Absender
RFF+Z13:44001'                  ← Prüfidentifikator
IDE+24+12345678901'             ← Marktlokation
UNT+6+1'                        ← Nachrichtenende
```

Die Trennzeichen

Zeichen	Bedeutung	Beispiel
`'`	Segment-Ende	`NAD+MS+123'`
`+`	Datenelemente-Trenner	`NAD+MS+123`
`:`	Komponenten-Trenner	`RFF+Z13:44001`
`?`	Escape-Character	`?+` für echtes `+`
`,`	Dezimaltrennzeichen	`1234,56`

Wichtige Segmente im Überblick

Segment	Vollständiger Name	Inhalt	Beispiel
UNH	Message Header	Nachrichtentyp, Version, Referenz	`UNH+1+UTILMD:D:11A:UN:2.6'`
BGM	Beginning of Message	Dokumentart, Nummer, Funktion	`BGM+E01+12345+9'`
DTM	Date/Time/Period	Datum/Zeit mit Qualifier	`DTM+137:20241019:102'`
NAD	Name and Address	Partei-Informationen	`NAD+MS+9900123456789::293'`
RFF	Reference	Referenzen (z.B. Prüf-ID)	`RFF+Z13:44001'`
IDE	Identity	Objekt-Identifikation	`IDE+24+12345678901'`
LOC	Place/Location	Ortsangaben	`LOC+172+DE'`
QTY	Quantity	Mengenangaben	`QTY+220:1234.5:KWH'`
MOA	Monetary Amount	Geldbeträge	`MOA+77:1234.56:EUR'`
CCI	Characteristic	Eigenschaften/Klassen	`CCI+Z19++++Bilanzkreis'`
UNT	Message Trailer	Segmentanzahl, Referenz	`UNT+6+1'`

Kardinalitäten verstehen

In den BDEW-Anwendungshandbüchern (AHB) findest du Kardinalitäten:

Code	Bedeutung	Beschreibung
M	Mandatory	Pflichtfeld, muss immer vorhanden sein
C	Conditional	Bedingt, abhängig vom Kontext
R	Required (BDEW)	BDEW-spezifische Pflichtfelder
D	Dependent	Abhängig von anderen Feldern
N	Not used	Nicht verwendet in diesem Kontext

□□ Schnellstart mit dem Transforme

Installation

```
npm install edifact-json-transformer
```

Deine erste Transformation

```
const { EdifactTransformer } = require('edifact-json-transformer');

// Transformer mit Validierung erstellen
const transformer = new EdifactTransformer({
  enableAHBValidation: true,      // AHB-Regeln prüfen
  parseTimestamps: true,           // Datumswerte formatieren
  generateGraphRelations: true     // Graph-Beziehungen erstellen
});

// EDIFACT-String (z.B. aus Datei gelesen)
const edifactString = `
UNH+1+UTILMD:D:11A:UN:2.6'
BGM+E01+12345+9'
DTM+137:20241019:102'
NAD+MS+9900123456789::293'
NAD+MR+9900987654321::293'
RFF+Z13:44001'
IDE+24+12345678901'
UNT+7+1'
`;

// Transformation durchführen
const json = transformer.transform(edifactString);

// Ergebnis verwenden
console.log('Nachrichtentyp:', json.metadata.message_type);
console.log('Prüf-ID:', json.metadata.pruefidentifikator);
console.log('Marktlokationen:', json.body.stammdaten.marktlokationen);
```

Ausgabe:

```
{
  metadata: {
    message_type: 'UTILMD',
    message_name: 'Stammdaten',
    category: 'master_data',
    applicable_processes: ['GPKE', 'GeLi Gas', 'WiM', 'MaBiS'],
```

```
    version: '2.6',
    pruefidentifikator: {
      id: '44001',
      description: 'Anmeldung NN'
    }
  },
  body: {
    stammdaten: {
      marktlokationen: [
        { id: '12345678901', valid: true }
      ]
    }
  },
  parties: {
    sender: {
      id: '9900123456789',
      role: 'MS',
      valid_mp_id: true
    },
    receiver: {
      id: '9900987654321',
      role: 'MR',
      valid_mp_id: true
    }
  }
}
```

□□ Die wichtigsten Nachrichtentype im Detail

1. UTILMD – Utility Master Data (Stammdaten)

Zweck: Austausch von Stammdaten zu Netzanschlüssen, Messlokationen und Lieferantenzuordnungen

Zentrale Segmente:

- **UNH, BGM**: Nachrichtenkopf
- **DTM**: Zeitangaben (Lieferbeginn/-ende)
- **RFF+Z13**: Prüfidentifikator (Geschäftsvorfall)
- **NAD**: Beteiligte Marktpartner
- **IDE+24**: Marktlokations-ID (11-stellig)
- **IDE+25**: Messlokations-ID (33-stellig)
- **LOC**: Lokationsangaben
- **CCI**: Eigenschaften (z.B. Bilanzkreis)

Code-Beispiel:

```
const json = transformer.transform(utilmdString);

// Stammdaten auslesen
const malo = json.body.stammdaten.marktlokationen[0];
console.log(`MaLo-ID: ${malo.id}, gültig: ${malo.valid}`);

// Prüfidentifikator prüfen
if (json.metadata.pruefidentifikator.id === '44001') {
  console.log('Anmeldung zur Netznutzung');
}

// Bilanzkreis ermitteln
const bk = json.body.stammdaten.bilanzkreise[0];
console.log(`Bilanzkreis: ${bk.id}`);
```

Typische Anwendungsfälle:

- Lieferantenwechsel (GPKE/GeLi Gas)
- Stammdatenpflege
- Messlokations-Anmeldung (WiM)

---

2. MSCONS – Meter Reading Schedule Consumption (Messwerte)

Zweck: Übermittlung von Verbrauchsdaten zwischen Marktpartnern

Zentrale Segmente:

- **UNH, BGM**: Nachrichtenkopf
- **DTM**: Messperiode (Start/Ende)
- **RFF**: Referenzen (Marktlokation, Geschäftsvorfall)
- **NAD**: Beteiligte Partner
- **LOC**: Messlokation
- **QTY**: Messwerte mit Einheit
- **SEQ**: Zeitreihen-Sequenzen
- **MEA**: Detaillierte Messwerte (oft Viertelstundenwerte)

Code-Beispiel:

```
const json = transformer.transform(msconsString);

// Messwerte auslesen
json.body.messwerte.messwerte.forEach(messwert => {
  console.log(`
    Qualifier: ${messwert.qualifier}
    Wert: ${messwert.value} ${messwert.unit}
    Gültig: ${messwert.valid_unit}
  `);
});

// Zeitreihen-Daten
console.log('Anzahl Zeitscheiben:', json.body.messwerte.zeitreihen.length);

// Messperiode
const periode = json.body.messwerte.messperioden;
console.log(`Von ${periode[0].datetime} bis ${periode[1].datetime}`);
```

Wichtige Qualifier:

- `220`: Gelieferte Energie
- `66`: Zählerstand
- `74`: Bezogene Energie

Einheiten:

- `KWH`: Kilowattstunde
- `MWH`: Megawattstunde
- `W`: Watt (Leistung)

Typische Anwendungsfälle:

- Bilanzierungsrelevante Messwerte (MaBiS)
- Abrechnungsdaten für Lieferanten
- Austausch zwischen MSB und NB/LF

---

3. ORDERS – Purchase Order (Bestellung)

Zweck: Elektronische Übermittlung von Bestellungen und Anfragen

Zentrale Segmente:

- **UNH, BGM**: Nachrichtenkopf
- **DTM**: Bestelldatum, Liefertermin
- **RFF**: Bestellnummer, Referenzen
- **NAD**: Besteller und Empfänger
- **LIN**: Bestellpositionen
- **PIA**: Produktidentifikation
- **IMD**: Artikelbeschreibung
- **QTY**: Bestellmengen

Code-Beispiel:

```
const json = transformer.transform(ordersString);


// Bestellpositionen auslesen
json.body.bestellung.order_lines.forEach(line => {
  console.log(`
    Position: ${line.line_number}
    Artikel-ID: ${line.item_id}
    Typ: ${line.item_type}
  `);
});
```

Typische Anwendungsfälle:

- Beauftragung Messstellenbetrieb (WiM)
- Anfrage Gasbeschaffenheit (GeLi Gas)
- Materialbestellungen

---

4. ORDRSP – Purchase Order Response (Bestellantwort)

Zweck: Antwort auf ORDERS-Nachricht

Zentrale Segmente:

- **UNH, BGM**: Nachrichtenkopf
- **DTM**: Antwortdatum
- **RFF**: Referenz auf ORDERS
- **NAD**: Absender/Empfänger
- **STS**: Bestellstatus
- **LIN, PIA, IMD**: Positionsdaten

Statuswerte:

- `7`: Bestätigt
- `27`: Abgelehnt
- `4`: Teilweise bestätigt

---

5. INVOIC – Invoice (Rechnung)

Zweck: Übermittlung von Rechnungen und Gutschriften

Zentrale Segmente:

- **UNH, BGM**: Nachrichtenkopf
- **DTM**: Rechnungs-/Leistungsdatum
- **RFF**: Rechnungs-/Bestellnummer
- **NAD**: Rechnungssteller/-empfänger
- **LIN, PIA, IMD**: Rechnungspositionen
- **MOA**: Geldbeträge
- **TAX**: Steuerdetails

Code-Beispiel:

```
const json = transformer.transform(invoicString);


// Rechnungssumme
const summe = json.body.rechnung.totals['77'];
console.log(`Gesamt: ${summe.amount} ${summe.currency}`);
```

```
// Steuern
json.body.rechnung.tax_amounts.forEach(tax => {
  console.log(`Steuer: ${tax.type}, Rate: ${tax.rate}%`);
});
```

MOA-Qualifier:

- `77`: Rechnungssumme
- `79`: Gesamtbetrag inkl. MwSt
- `125`: Steuerbetrag

---

6. APERAK – Application Error and Acknowledgement

Zweck: Bestätigung oder Fehlermeldung für empfangene Nachrichten

Zentrale Segmente:

- **UNH, BGM**: Nachrichtenkopf
- **DTM**: Erstellungsdatum
- **RFF**: Referenz auf Original-Nachricht
- **ERC**: Fehlerinformationen
- **FTX**: Freitext-Erklärungen

Code-Beispiel:

```
const json = transformer.transform(aperakString);

// Status prüfen
if (json.body.quittierung.status === 'positive') {
  console.log('Nachricht erfolgreich verarbeitet');
} else {
  // Fehler ausgeben
  json.body.quittierung.errors.forEach(error => {
    console.error(`Fehler ${error.code}: ${error.description}`);
  });
}
```

Häufige Fehlercodes:

- `1`: Syntax-Fehler
- `2`: Semantischer Fehler
- `3`: Geschäftsprozess-Fehler
- `7`: Nachricht akzeptiert

□□ Prüfidentifikatoren verstehen (RFF+Z13)

Was ist ein Prüfidentifikator?

Der **Prüfidentifikator** (im Segment `RFF+Z13`) ist die **wichtigste Kennung** in einer MaKo-Nachricht. Er identifiziert eindeutig den Geschäftsvorfall und bestimmt:

- Welcher Prozess läuft (Anmeldung, Abmeldung, etc.)
- Welche Segmente Pflicht sind
- Welche Validierungsregeln gelten
- Welche Antwort-Nachrichten zu erwarten sind

Format:

```
RFF+Z13:44001'
     │   │
     │   └─ Prüfidentifikator (5-stellig)
     └─ Qualifier Z13
```

Die wichtigsten Prüfidentifikatoren

GPKE (Strom)

Prüf-ID	Prozess	Richtung	Typische Antwort
44001	Anmeldung Netznutzung	Lieferant → NB	APERAK (44002/44003)
44002	Bestätigung Anmeldung	NB → Lieferant	-
44003	Ablehnung Anmeldung	NB → Lieferant	-
44004	Abmeldung Netznutzung	Lieferant → NB	APERAK (44005/44006)
44005	Bestätigung Abmeldung	NB → Lieferant	-

Prüf-ID	Prozess	Richtung	Typische Antwort
44006	Ablehnung Abmeldung	NB → Lieferant	-
44016	Kündigung beim alten LF	NB → Lieferant	APERAK (44017/44018)
44017	Bestätigung Kündigung	Lieferant → NB	-
44018	Ablehnung Kündigung	Lieferant → NB	-
44112	Stammdatenänderung	NB → Lieferant	-
44123	Bila.rel. Änderung	NB → Lieferant	-

Messwerte

Prüf-ID	Prozess	Verwendung
13002	Zählerstand	MSCONS
13008	Lastgang	MSCONS
13009	Energiemenge	MSCONS
13007	Gasbeschaffenheit	MSCONS (Gas)
13006	Messwert Storno	MSCONS

Stammdaten-Anfragen

Prüf-ID	Prozess	Verwendung
17101	Anfrage Stammdaten MaLo	ORDERS
17102	Anfrage Werte	ORDERS
19101	Ablehnung Anfrage Stammdaten	ORDRSP
19102	Ablehnung Anfrage Werte	ORDRSP

WiM (Messwesen)

Prüf-ID	Prozess	Verwendung
17009	Gerätewechsel	ORDERS
19015	Bestätigung Gerätewechsel	ORDRSP
21039	Auftragsstatus Sperren	IFTSTA
21040	Info Entsperrauftrag	IFTSTA

Rechnungen

Prüf-ID	Prozess	Verwendung
31001	Abschlagsrechnung	INVOIC
31002	NN-Rechnung	INVOIC
31003	WiM-Rechnung	INVOIC
31004	Stornorechnung	INVOIC
33001	Bestätigung	REMADV
33002	Abweisung	REMADV

Code-Beispiel: Prüfidentifikator auswerten

```
const json = transformer.transform(edifactString);
const pruefId = json.metadata.pruefidentifikator.id;

switch(pruefId) {
  case '44001':
    console.log('Anmeldung zur Netznutzung');
    // Validiere Pflichtfelder für Anmeldung
    validateAnmeldung(json);
    break;

  case '44004':
    console.log('Abmeldung vom Netz');
    validateAbmeldung(json);
    break;

  case '13008':
    console.log('Lastgangdaten');
    processLastgang(json.body.messwerte);
    break;

  default:
    console.warn(`Unbekannte Prüf-ID: ${pruefId}`);
}
```

Helper-Funktion

```
const { isGPKEProcess } = require('edifact-json-transformer');

// Prozess-Zuordnung prüfen
if (isGPKEProcess(edifactString)) {
  console.log('GPKE-Prozess erkannt');
}

// Alle Prüfidentifikatoren verfügbar
const { pruefidentifikatoren } = require('edifact-json-transformer');
console.log(pruefidentifikatoren['44001']);
// → "Anmeldung NN"
```

□□ Praktische Tipps für den Alltag

1. Schnell-Check: Was ist das für eine Nachricht?

```
// Nur Metadaten ohne vollständige Transformation
const { EdifactTransformer } = require('edifact-json-transformer');

function quickInfo(edifactString) {
  const transformer = new EdifactTransformer({
    validateStructure: false,  // Schneller
    parseTimestamps: false
  });

  const json = transformer.transform(edifactString);

  return {
    typ: json.metadata.message_type,
    name: json.metadata.message_name,
    pruefId: json.metadata.pruefidentifikator?.id,
    prozess: json.metadata.applicable_processes,
    absender: json.parties.sender?.id,
    empfaenger: json.parties.receiver?.id
```

```
  };
}

console.log(quickInfo(edifactString));
// { typ: 'UTILMD', name: 'Stammdaten', pruefId: '44001', ... }
```

2. Marktlokationen schnell extrahieren

```
const { extractAllMarktlokationIds } = require('edifact-json-transformer');

// Super-schnelle Extraktion
const maloIds = extractAllMarktlokationIds(edifactString);
console.log(maloIds); // ['12345678901', '98765432109']

// In der Datenbank nachschlagen
maloIds.forEach(id => {
  const kunde = db.findByMaloId(id);
  console.log(`Kunde: ${kunde.name} (${id})`);
});
```

3. Zeitscheiben in MSCONS verarbeiten

```
function processTimeSeries(msconsString) {
  const json = transformer.transform(msconsString);
  const zeitreihen = json.body.messwerte.zeitreihen;

  // Viertelstundenwerte
  const viertelstundenwerte = zeitreihen.map((seq, idx) => {
    const messwert = json.body.messwerte.messwerte[idx];
    return {
      sequenz: seq.sequence_number,
      wert: messwert.value,
      einheit: messwert.unit,
      zeitpunkt: calculateTimeSlot(seq.sequence_number, startDate)
    };
  });
```

```
  return viertelstundenwerte;
}

function calculateTimeSlot(sequenz, startDate) {
  // Viertelstunde = 15 Minuten
  const minutes = (sequenz - 1) * 15;
  return new Date(startDate.getTime() + minutes * 60000);
}
```

4. Validierung mit Fehler-Report

```
const { validateAHB } = require('edifact-json-transformer');

function validateAndReport(edifactString) {
  const report = validateAHB(edifactString);

  if (!report.is_valid) {
    console.error('⚠ Validierungsfehler gefunden:');

    report.errors.forEach(error => {
      console.error(`  [${error.category}] ${error.message}`);
    });

    report.warnings.forEach(warning => {
      console.warn(`  ⚠ [${warning.category}] ${warning.message}`);
    });

    return false;
  }

  console.log('✓ Nachricht ist valide');
  return true;
}
```

5. Datumswerte lesen

```
const json = transformer.transform(edifactString);

// Alle Datumswerte
console.log(json.dates);
// {
//   message_date: '2024-10-19',
//   start_date: '2024-01-01',
//   end_date: '2024-12-31'
// }

// Spezifische Prüfungen
const { start_date, end_date } = json.dates;

if (new Date(start_date) > new Date(end_date)) {
  console.error('Start liegt nach Ende!');
}

// Dauer berechnen
const days = (new Date(end_date) - new Date(start_date)) / (1000 * 60 * 60 * 24);
console.log(`Zeitraum: ${days} Tage`);
```

6. Graph-Beziehungen für Analyse

```
const transformer = new EdifactTransformer({
  generateGraphRelations: true
});

const json = transformer.transform(edifactString);

// Neo4j Cypher generieren
const { convertToNeo4jCypher } = require('edifact-json-transformer');
const statements = convertToNeo4jCypher(edifactString);

statements.forEach(stmt => {
  console.log(stmt.cypher);
  console.log(stmt.parameters);
});
```

```
// Oder direkt aus JSON
json.graph_relations.forEach(rel => {
  console.log(`${rel.from.type}:${rel.from.id} -[${rel.relationship}]->
${rel.to.type}:${rel.to.id}`);
});
```

7. Batch-Verarbeitung

```
const fs = require('fs');
const { EdifactTransformer } = require('edifact-json-transformer');

function processBatch(directory) {
  const transformer = new EdifactTransformer({
    enableAHBValidation: true
  });

  const files = fs.readdirSync(directory);
  const results = {
    success: 0,
    errors: [],
    warnings: []
  };

  files.forEach(file => {
    try {
      const content = fs.readFileSync(`${directory}/${file}`, 'utf8');
      const json = transformer.transform(content);

      if (json.validation?.is_valid !== false) {
        results.success++;
      } else {
        results.errors.push({
          file,
          errors: json.validation.errors
        });
      }

      if (json.validation?.warnings?.length > 0) {
        results.warnings.push({
```

```
          file,
          warnings: json.validation.warnings
        });
      }
    } catch (error) {
      results.errors.push({ file, error: error.message });
    }
  });

  console.log(`
    ✅ Erfolgreich: ${results.success}
    ❌ Fehler: ${results.errors.length}
    ⚠️ Warnungen: ${results.warnings.length}
  `);

  return results;
}
```

🔧 Häufige Fehler und Lösungen

1. Syntaxfehler

Problem: Falsche Trennzeichen

```
// ❌ Falsch
"UNH+1+UTILMD:D:11A:UN:2.6'BGM+E01+12345+9'..."  // Fehlt Zeilenumbruch-Escape

// ✅ Richtig
const edifact = normalizeEdifact(rawString);  // Transformer macht das automatisch
```

Problem: Fehlende Pflichtsegmente

```
const json = transformer.transform(edifactString);

if (json.validation?.errors) {
  json.validation.errors.forEach(error => {
```

```
    if (error.message.includes('Fehlendes UNH')) {
      console.error('Nachrichtenkopf fehlt!');
    }
  });
}
```

Lösung: Prüfe die AHB-Vorgaben für den jeweiligen Nachrichtentyp.

Problem: Falsche Datenformate

```
// DTM-Segment mit falschem Format
"DTM+137:19.10.2024:102'"   // ❌ Falsch: TT.MM.JJJJ

"DTM+137:20241019:102'"     // ✅ Richtig: JJJJMMTT
```

Debugging:

```
const json = transformer.transform(edifactString, {
  includeRawSegments: true
});

// Rohe Segmente inspizieren
json.raw_segments.filter(s => s.tag === 'DTM').forEach(dtm => {
  console.log('DTM:', dtm.raw);
});
```

---

2. Semantische Fehler

Problem: Referenznummer-Mismatch

```
// ❌ UNH und UNT stimmen nicht überein
"UNH+1+UTILMD:D:11A:UN:2.6'...UNT+7+2'"

// ✅ Korrekt
"UNH+1+UTILMD:D:11A:UN:2.6'...UNT+7+1'"
```

Automatische Prüfung:

```
const json = transformer.transform(edifactString);

if (json.validation?.errors.some(e => e.message.includes('Referenznummer-Mismatch'))) {
  console.error('UNH/UNT Referenzen stimmen nicht überein!');
}
```

Problem: Ungültige MP-ID

```
const json = transformer.transform(edifactString);

Object.entries(json.parties).forEach(([role, party]) => {
  if (!party.valid_mp_id) {
    console.error(`
      ⚠ Ungültige MP-ID für ${role}
      Wert: ${party.id}
      Erwartet: 13-stellig numerisch
    `);
  }
});
```

Validierung:

```
function isValidMpId(id) {
  return /^\d{13}$/.test(id);
}
```

Problem: Start-/Enddatum vertauscht

```
const json = transformer.transform(edifactString);

if (json.validation?.errors) {
  json.validation.errors.forEach(error => {
    if (error.message.includes('Start-Datum liegt nach End-Datum')) {
      console.error('Zeitlogik ist falsch!');
      console.log('Start:', json.dates.start_date);
      console.log('Ende:', json.dates.end_date);
    }
  });
}
```

3. Geschäftsprozess-Fehler

Problem: Falscher Prüfidentifikator

```
const json = transformer.transform(edifactString);
const pruefId = json.metadata.pruefidentifikator?.id;

// Prüfen ob Prüf-ID zum Nachrichtentyp passt
const validCombinations = {
  'UTILMD': ['44001', '44002', '44003', '44004', '44005', '44006', '44016', '44112', '44123'],
  'MSCONS': ['13002', '13007', '13008', '13009'],
  'ORDERS': ['17009', '17101', '17102'],
  'INVOIC': ['31001', '31002', '31003', '31004']
};

const messageType = json.metadata.message_type;
if (!validCombinations[messageType]?.includes(pruefId)) {
  console.error(`
    ❌ Prüf-ID ${pruefId} passt nicht zu ${messageType}
    Erlaubte IDs: ${validCombinations[messageType].join(', ')}
  `);
}
```

Problem: Fehlende Rollen

```
// Für Anmeldung (44001) sind spezifische Rollen erforderlich
if (json.metadata.pruefidentifikator.id === '44001') {
  const hasLieferant = json.parties.lieferant || json.parties.sender;
  const hasNetzbetreiber = json.parties.netzbetreiber || json.parties.receiver;

  if (!hasLieferant) {
    console.error('❌ Lieferant-Rolle fehlt für Anmeldung');
  }

  if (!hasNetzbetreiber) {
    console.error('❌ Netzbetreiber-Rolle fehlt für Anmeldung');
  }
}
```

Problem: Ungültige Marktlokations-ID

```
const json = transformer.transform(edifactString);

json.body.stammdaten?.marktlokationen?.forEach((malo, idx) => {
  if (!malo.valid) {
    console.error(`
      ⚠️ Marktlokation ${idx + 1} ungültig
      ID: ${malo.id}
      Erwartet: 11-stellig
      Tatsächlich: ${malo.id?.length} Zeichen
    `);
  }
});

json.body.stammdaten?.messlokationen?.forEach((melo, idx) => {
  if (!melo.valid) {
    console.error(`
      ⚠️ Messlokation ${idx + 1} ungültig
      ID: ${melo.id}
      Erwartet: 33-stellig
      Tatsächlich: ${melo.id?.length} Zeichen
    `);
  }
});
```

🎯 Best Practices für Entwickler

1. Validierung auf allen Ebenen

```
class EdifactProcessor {
  constructor() {
    this.transformer = new EdifactTransformer({
      enableAHBValidation: true,
      validateStructure: true,
      validateBusinessRules: true
```

```
  });
}

async process(edifactString) {
  // 1. Syntax-Validierung
  if (!this.validateSyntax(edifactString)) {
    throw new Error('Syntax ungültig');
  }

  // 2. Transformation
  const json = this.transformer.transform(edifactString);

  // 3. Struktur-Validierung
  if (!json.validation?.is_valid) {
    this.logErrors(json.validation.errors);
    throw new Error('Struktur ungültig');
  }

  // 4. Geschäftsprozess-Validierung
  this.validateBusinessLogic(json);

  // 5. Verarbeitung
  return this.processMessage(json);
}

validateSyntax(edifactString) {
  // Grundlegende Checks
  return edifactString.includes("UNH") &&
         edifactString.includes("UNT");
}

validateBusinessLogic(json) {
  const pruefId = json.metadata.pruefidentifikator?.id;

  // Prozessspezifische Validierung
  switch(pruefId) {
    case '44001':
      this.validateAnmeldung(json);
      break;
```

```
      case '13008':
        this.validateLastgang(json);
        break;
    }
  }
}
```

2. Umfassendes Error Handling

```
class MessageValidator {
  validate(json) {
    const errors = [];
    const warnings = [];

    // Pflichtfelder prüfen
    if (!json.metadata.pruefidentifikator) {
      errors.push({
        category: 'AHB',
        message: 'Prüfidentifikator fehlt',
        severity: 'ERROR'
      });
    }

    // Marktlokationen prüfen
    json.body.stammdaten?.marktlokationen?.forEach((malo, idx) => {
      if (!malo.valid) {
        warnings.push({
          category: 'STAMMDATEN',
          message: `MaLo ${idx + 1}: ${malo.id} ist ungültig`,
          severity: 'WARNING'
        });
      }
    });

    return {
      is_valid: errors.length === 0,
      errors,
      warnings
```

```
    };
  }

  generateAPERAK(originalMessage, errors) {
    // APERAK-Nachricht generieren
    return `
      UNH+${generateRef()}+APERAK:D:11A:UN:2.0'
      BGM+${errors.length > 0 ? '27' : '7'}+${generateRef()}+9'
      ${errors.map(e => `ERC+${e.code}'`).join('')}
      UNT+${2 + errors.length}+${generateRef()}'
    `;
  }
}
```

3. Strukturierte Logging-Strategie

```
const winston = require('winston');

const logger = winston.createLogger({
  level: 'info',
  format: winston.format.json(),
  transports: [
    new winston.transports.File({ filename: 'edifact-error.log', level: 'error' }),
    new winston.transports.File({ filename: 'edifact-combined.log' })
  ]
});

function processMessage(edifactString, source) {
  const startTime = Date.now();

  try {
    const json = transformer.transform(edifactString);

    logger.info({
      event: 'message_processed',
      type: json.metadata.message_type,
      pruefId: json.metadata.pruefidentifikator?.id,
      source,
```

```
      duration: Date.now() - startTime,
      validation: json.validation?.is_valid
    });

    return json;

  } catch (error) {
    logger.error({
      event: 'processing_failed',
      source,
      error: error.message,
      stack: error.stack,
      duration: Date.now() - startTime
    });
    throw error;
  }
}
```

4. Modularer Aufbau

```
// parser.js
class EdifactParser {
  parse(edifactString) {
    return transformer.transform(edifactString);
  }
}

// validator.js
class EdifactValidator {
  validateAHB(json) { /* ... */ }
  validateStructure(json) { /* ... */ }
  validateBusinessRules(json) { /* ... */ }
}

// processor.js
class MessageProcessor {
  processUTILMD(json) { /* ... */ }
  processMSCONS(json) { /* ... */ }
```

```
  processORDERS(json) { /* ... */ }
}

// main.js
class EdifactService {
  constructor() {
    this.parser = new EdifactParser();
    this.validator = new EdifactValidator();
    this.processor = new MessageProcessor();
  }

  async handleMessage(edifactString) {
    const json = this.parser.parse(edifactString);

    if (!this.validator.validateAHB(json)) {
      throw new ValidationError('AHB-Validierung fehlgeschlagen');
    }

    const messageType = json.metadata.message_type;
    return this.processor[`process${messageType}`](json);
  }
}
```

5. Testing-Strategie

```
// test/utilmd.test.js
const { EdifactTransformer } = require('edifact-json-transformer');

describe('UTILMD Transformation', () => {
  let transformer;

  beforeEach(() => {
    transformer = new EdifactTransformer({
      enableAHBValidation: true
    });
  });

  test('Anmeldung NN (44001) korrekt verarbeitet', () => {
```

```
    const edifact = loadTestFile('utilmd_anmeldung_44001.txt');
    const json = transformer.transform(edifact);

    expect(json.metadata.message_type).toBe('UTILMD');
    expect(json.metadata.pruefidentifikator.id).toBe('44001');
    expect(json.body.stammdaten.marktlokationen).toHaveLength(1);
    expect(json.validation.is_valid).toBe(true);
  });

  test('Ungültige MaLo-ID wird erkannt', () => {
    const edifact = loadTestFile('utilmd_invalid_malo.txt');
    const json = transformer.transform(edifact);

    expect(json.validation.is_valid).toBe(false);
    expect(json.validation.errors).toContainEqual(
      expect.objectContaining({
        message: expect.stringContaining('11-stellig')
      })
    );
  });

  test('Fehlender Prüfidentifikator wird gemeldet', () => {
    const edifact = loadTestFile('utilmd_no_pruefid.txt');
    const json = transformer.transform(edifact);

    expect(json.validation.errors).toContainEqual(
      expect.objectContaining({
        message: expect.stringContaining('Prüfidentifikator')
      })
    );
  });
});
```

6. Versionierung und Updates

```
class EdifactService {
  constructor() {
    this.transformers = {
```

```
      'v2.6': new EdifactTransformer({ /* config für 2.6 */ }),
      'v2.7': new EdifactTransformer({ /* config für 2.7 */ })
    };
  }

  getTransformer(version) {
    return this.transformers[`v${version}`] || this.transformers['v2.6'];
  }

  process(edifactString) {
    // Version aus UNH extrahieren
    const versionMatch = edifactString.match(/UNH\+\d+\+[^:]+:[^:]+:[^:]+:[^:]+:([^']+)'/);
    const version = versionMatch ? versionMatch[1] : '2.6';

    const transformer = this.getTransformer(version);
    return transformer.transform(edifactString);
  }
}
```

7. Performance-Optimierung

```
// Caching für wiederholte Transformationen
const NodeCache = require('node-cache');
const cache = new NodeCache({ stdTTL: 600 });

function transformWithCache(edifactString) {
  const hash = crypto.createHash('md5').update(edifactString).digest('hex');

  let json = cache.get(hash);
  if (json) {
    console.log('Cache hit');
    return json;
  }

  json = transformer.transform(edifactString);
  cache.set(hash, json);
  return json;
}
```

```
// Bulk-Processing mit Worker Threads
const { Worker } = require('worker_threads');

async function processBulk(messages) {
  const chunkSize = Math.ceil(messages.length / 4);
  const workers = [];

  for (let i = 0; i < messages.length; i += chunkSize) {
    const chunk = messages.slice(i, i + chunkSize);
    workers.push(
      new Promise((resolve) => {
        const worker = new Worker('./edifact-worker.js', {
          workerData: chunk
        });
        worker.on('message', resolve);
      })
    );
  }

  const results = await Promise.all(workers);
  return results.flat();
}
```

□□ Weiterführende Ressourcen

Offizielle Quellen

1. **EDI@Energy Portal** (BDEW)
   - URL: https://www.edi-energy.de/
   - Registrierung erforderlich
   - Enthält: AHB, MIG, Codelisten, Prozessbeschreibungen
2. **Bundesnetzagentur**
   - Festlegungen zu GPKE, GeLi Gas, WiM, MaBiS
   - URL: https://www.bundesnetzagentur.de/
3. **BDEW Codelisten**
   - Aktuelle Qualifier und Codes
   - Updates bei Prozessänderungen

Tools und Plattformen

1. **Willi Mako** – Intelligente MaKo-Plattform
   - Web-App: https://stromhaltig.de/app/
   - Open-Source Client: https://github.com/energychain/willi-mako-client
   - Regulatorisches Wissen, Beispiele, Validierung
2. **edifact-json-transformer**
   - Repository: https://github.com/energychain/edifact-to-json-transformer
   - NPM: `npm install edifact-json-transformer`
   - MIT-Lizenz, Community-Support

Lernressourcen

- **EDIFACT ISO 9735 Standard**: Grundlagen zu Segmenten und Syntax
- **BDEW Schulungen**: Workshops zu Marktkommunikation
- **Online-Communities**: Austausch mit anderen Entwickler:innen

Support

- **Issues/Bugs**: GitHub Issues
- **Pull Requests**: Contributions willkommen!
- **Maintainer**: STROMDAO GmbH (https://stromdao.de/)

□□ Zusammenfassung

Du hast jetzt gelernt:

□**Grundlagen**: Was EDIFACT ist und warum es in der Energiewirtschaft genutzt wird
□**Prozesse**: GPKE, GeLi Gas, WiM und MaBiS im Detail
□**Nachrichtentypen**: UTILMD, MSCONS, ORDERS, INVOIC, APERAK
□**Prüfidentifikatoren**: Die wichtigsten IDs und ihre Bedeutung
□**Praktische Tools**: Wie du mit dem Transformer effizient arbeitest
□**Fehlerbehandlung**: Häufige Probleme erkennen und lösen
□**Best Practices**: Professionelle Entwicklung von MaKo-Software

Nächste Schritte

1. **Installiere** den Transformer: `npm install edifact-json-transformer`
2. **Probiere** die Beispiele aus diesem Guide
3. **Validiere** echte Nachrichten aus deinem Projekt
4. **Erkunde** Willi Mako für tieferes regulatorisches Wissen
5. **Teile** deine Erfahrungen und trage zur Community bei

Denk daran

> "EDIFACT ist komplex, aber mit den richtigen Tools und etwas Übung wirst du zum MaKo-Profi!"

Viel Erfolg bei deinen ersten (oder nächsten) Schritten in der Marktkommunikation! □□

Maintainer: STROMDAO GmbH

Powered by: Willi Mako – Intelligente Marktkommunikation

Veröffentlicht auf: Corrently.io

Letzte Aktualisierung: Oktober 2024

Revision #1
Created 20 October 2025 09:58:11 by Thorsten Zoerner
Updated 20 October 2025 09:59:38 by Thorsten Zoerner