X12 to JSON Conversion Specification

Overview

This document outlines the approach for converting X12 EDI format into JSON for easier database storage and processing.

Core Conversion Principles

1. Delimiter Parsing

- **Segment Terminator**: `~` separates segments
- **Element Separator**: `*` separates data elements within segments
- **Subelement Separator**: `:` separates composite elements
- **Repetition Separator**: `^` (if present) separates repeated elements

2. Hierarchical Structure Preservation

X12 uses loops and hierarchical levels (HL segments) that must be preserved in JSON:

- Envelope structure (ISA/GS/ST)
- Hierarchical levels (providers, subscribers, claims)
- Loop structures (2000A, 2000B, 2300, 2400 for 837)

3. Segment Identification

Each segment begins with a 2-3 character identifier:

- `ISA` = Interchange Control Header
- `GS` = Functional Group Header
- `ST` = Transaction Set Header
- `NM1` = Name/Entity
- `CLM` = Claim Information
- etc.

JSON Structure Approach

Option 1: Literal Mapping (Preserves X12 Structure)

```json
```

```
{
  "interchangeControlHeader": {
    "segmentId": "ISA",
    "authorizationQualifier": "00",
    "authorizationInformation": "          ",
    "securityQualifier": "00",
    "securityInformation": "          ",
    ...
  }
}
```

Pros: Direct 1:1 mapping, easy to reverse **Cons**: Not very human-readable, preserves X12 complexity

Option 2: Semantic Mapping (Human-Readable)

```json
{
  "interchange": {
    "senderId": "SUBMITTER123",
    "receiverId": "RECEIVER456",
    "date": "2025-01-15",
    "time": "14:30",
    "controlNumber": "000000001"
  },
  "claims": [
    {
      "claimId": "PATIENT001",
      "totalCharge": 250.00,
      "patient": {
        "firstName": "JOHN",
        "lastName": "SMITH",
        "dateOfBirth": "1980-05-15"
      }
    }
  ]
}
```

Pros: Human-readable, easy to work with **Cons**: Requires business logic, harder to reverse

Option 3: Hybrid Approach (Recommended)

Combine both approaches with metadata:

```json
{
  "metadata": {
    "transactionType": "837P",
    "version": "005010X222A1",
    "generatedAt": "2025-01-15T14:30:00Z"
  },
  "raw": {
    // Literal segment mapping for audit trail
  },
  "parsed": {
    // Semantic, human-readable structure
  }
}
```

Key Conversion Challenges

1. Qualifiers and Codes

X12 uses qualifiers that determine the meaning of subsequent elements:

- `NM1*IL` = Individual/Insured
- `NM1*PR` = Payer
- `NM1*85` = Billing Provider

Solution: Create lookup tables for qualifiers and transform to descriptive keys

2. Composite Elements

Some elements contain subelements separated by `:`:

- `HC:99213` means Healthcare Code (HC) with value 99213

Solution: Parse composites into nested objects or arrays

3. Repeating Segments

Multiple segments of the same type (e.g., multiple `REF` or `HI` segments):

Solution: Use arrays in JSON structure

4. Hierarchical Loops

HL segments create parent-child relationships:

```
HL*1**20*1~     (Level 1 - Billing Provider)
HL*2*1*22*0~    (Level 2 - Subscriber, parent is 1)
```

Solution: Build tree structure with parent references

5. Situational Elements

Many X12 elements are optional/situational:

Solution: Only include present elements, or use `null` for missing required fields

Implementation Considerations

Data Types

- Dates: Convert `YYYYMMDD` to ISO 8601 format
- Times: Convert `HHMM` to ISO 8601 format
- Amounts: Convert to decimal/float
- Codes: Keep as strings but validate against code sets

Validation

- Segment order validation
- Required vs. optional elements
- Element length validation
- Code set validation

Error Handling

- Invalid delimiters
- Missing required segments
- Malformed data
- Unknown segment types

Performance

- Stream processing for large files
- Incremental JSON generation
- Memory-efficient parsing

Example Mapping: NM1 Segment

X12 Format:

```
NM1*IL*1*SMITH*JOHN*A***MI*123456789~
```

Parsed Elements:

1. Segment ID: NM1
2. Entity Identifier Code: IL (Insured/Individual)
3. Entity Type Qualifier: 1 (Person)
4. Last Name: SMITH
5. First Name: JOHN
6. Middle Name: A
7. Name Prefix: (empty)
8. Name Suffix: (empty)
9. ID Code Qualifier: MI (Member Identification Number)
10. ID Code: 123456789

JSON Output (Semantic):

```json
{
  "entityType": "insured",
  "person": {
    "lastName": "SMITH",
    "firstName": "JOHN",
    "middleName": "A"
  },
  "identifier": {
    "type": "memberID",
    "value": "123456789"
  }
}
```

Next Steps for Implementation

1. **Create JSON Schema**: Define the target JSON structure
2. **Build Parser**: Implement X12 tokenizer and segment parser
3. **Implement Mapping Logic**: Create transformation rules
4. **Add Validation**: Implement X12 and JSON validation
5. **Handle Edge Cases**: Deal with variations and errors
6. **Add Logging/Audit**: Track transformation process

7. **Performance Optimization**: Stream processing for large files

8. **Testing**: Validate against real-world X12 files

Trading Partner Considerations

Different payers and clearinghouses may:

- Use different implementation guides
- Have varying requirements for optional segments
- Expect specific code sets
- Have custom validation rules

Solution: Configuration-driven approach with partner-specific mappings