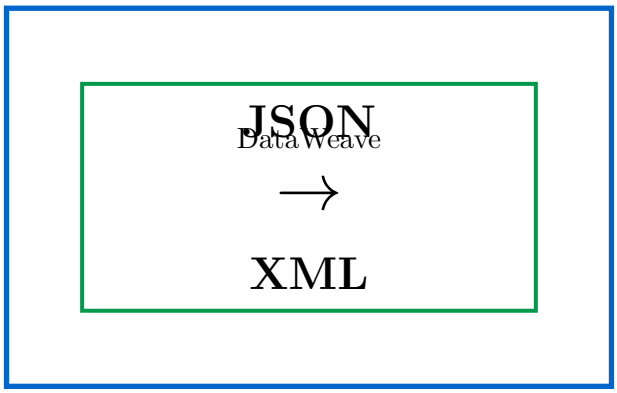
DataWeave

Complete Guide

From Beginner to Advanced



A Comprehensive Tutorial for MuleSoft Developers

D Charan Jeet

PAT - 2025

February 9, 2026

Contents

Preface

Welcome to the complete guide to DataWeave! This book is designed to take you from absolute beginner to advanced practitioner, teaching you not just the syntax, but the underlying concepts and architectural thinking needed to become proficient in DataWeave.

Who This Book Is For

This book is for:

- MuleSoft developers starting their journey
- Integration architects needing a comprehensive reference
- Students preparing for MuleSoft certification
- Anyone transforming data in enterprise systems

How to Use This Book

Read chapters sequentially if you're new to DataWeave. Each chapter builds on previous concepts. If you're experienced, use this as a reference guide and focus on advanced chapters.

Practice every example. Type the code yourself. Modify it. Break it. Fix it. This is how you truly learn.

Philosophy

This book teaches you to think like a DataWeave architect. You'll understand not just *how* to write transformations, but *why* certain approaches work better than others. You'll learn patterns, best practices, and the reasoning behind them.

Let's begin your journey to DataWeave mastery.

— *D Charan Jeet*

Chapter 1

Introduction to DataWeave

1.1 What is DataWeave?

DataWeave is a powerful transformation language created by MuleSoft for transforming data from one format to another. Think of it as a translator that speaks many languages — JSON, XML, CSV, Java objects, and more.

Concept

DataWeave is MuleSoft's expression language for data transformation. It's declarative, functional, and designed specifically for integration scenarios where data needs to be reshaped, filtered, or converted between formats.

1.1.1 Why DataWeave Exists

In enterprise integration, systems speak different languages. Your CRM system might output JSON, while your legacy ERP expects XML. Your database returns rows, but your REST API needs nested JSON objects. DataWeave solves this fundamental integration problem.

Before DataWeave, developers used:

- Java code (verbose, complex)
- XSLT for XML (limited, hard to maintain)
- Custom scripts (inconsistent, error-prone)

DataWeave provides a unified, concise way to transform data regardless of format.

1.1.2 The Role in MuleSoft

In MuleSoft's Anypoint Platform, DataWeave is the primary tool for:

1. **Transform Message component** — The main transformation component
2. **Expression evaluation** — Setting variables, conditions, logging
3. **Data validation** — Checking payload structure

4. **Dynamic routing** — Deciding flow paths based on data

1.2 How DataWeave Processes Data

DataWeave follows a simple model:

1. **Read** the input data
2. **Transform** using expressions and functions
3. **Write** to output format

Example

Here's a minimal DataWeave script:

```
%dw 2.0
output application/json
---
{
    message: "Hello, DataWeave!"
}
```

Output:

```
{
    "message": "Hello, DataWeave!"
}
```

1.2.1 The Three Parts of a DataWeave Script

Every DataWeave script has three parts:

1. **Header** — Directives like version and output format
2. **Separator** — Three dashes (`---`)
3. **Body** — The transformation logic

```
%dw 2.0                      // Header: Version
output application/json      // Header: Output format
---                          // Separator
{                            // Body: Transformation
    result: "Success"
}
```

Teacher's Tip

Think of the header as setup instructions and the body as the actual work. The separator (---) tells DataWeave: "Setup done, transformation starts now."

1.3 The Execution Model

DataWeave is:

- **Declarative** — You describe what you want, not how to get it
- **Functional** — Transformations don't modify data, they create new data
- **Lazy** — DataWeave only processes what's needed

1.3.1 Declarative Nature

In imperative languages (like Java), you write step-by-step instructions:

```
List<String> names = new ArrayList<>();
for (Person p : people) {
    names.add(p.getName());
}
```

Java Approach

In DataWeave, you declare the desired result:

```
payload map $.name
```

DataWeave Approach

1.3.2 Functional Paradigm

DataWeave functions don't have side effects. They always produce new data:

```
var numbers = [1, 2, 3]
var doubled = numbers map ($ * 2)
// numbers is still [1, 2, 3]
// doubled is [2, 4, 6]
```

Warning

DataWeave variables are immutable. Once assigned, they cannot change. This prevents bugs but requires a different thinking approach than traditional programming.

1.4 Common Use Cases

DataWeave excels at:

Use Case	Example
Format Conversion	JSON to XML, CSV to JSON
Data Filtering	Remove inactive customers
Data Enrichment	Add calculated fields
Aggregation	Sum orders by customer
Restructuring	Flatten nested objects
Validation	Check required fields

Table 1.1: Common DataWeave Use Cases

1.5 Your First DataWeave Transformation

Let's write a complete transformation:

Transform Customer Data

Input (JSON):

```
{
    "firstName": "John",
    "lastName": "Doe",
    "age": 30
}
```

DataWeave Script:

```
%dw 2.0
output application/json
---
{
    fullName: payload.firstName ++ " " ++ payload.lastName,
    isAdult: payload.age >= 18
}
```

Output:

```
{
    "fullName": "John Doe",
    "isAdult": true
}
```

1.5.1 Understanding the Example

Let's break it down:

- `%dw 2.0` — We're using DataWeave version 2.0
- `output application/json` — Output will be JSON format
- `---` — Separator between header and body
- `payload` — The input data

- `++` — String concatenation operator
- `>=` — Comparison operator

Teacher's Tip

The `payload` keyword always refers to the input data. Think of it as "the data coming in the door."

1.6 Summary

Summary

In this chapter, you learned:

- DataWeave is MuleSoft's transformation language
- It solves the integration problem of converting between data formats
- Scripts have three parts: header, separator, body
- DataWeave is declarative and functional
- The `payload` keyword represents input data

1.7 Practice Exercises

1. Write a DataWeave script that outputs your name as JSON
2. Modify the customer example to include an email field
3. Create a transformation that converts Celsius to Fahrenheit

Chapter 2

Syntax Fundamentals

2.1 Structure of a DataWeave Script

Every DataWeave script follows a consistent structure. Understanding this structure is crucial because it's the foundation everything else builds upon.

```
%dw 2.0                              // Version directive
output application/json              // Output format
var greeting = "Hello"               // Variable declaration
fun addExclamation(text) = text ++ "!"  // Function
---
{
    message: addExclamation(greeting)
}
```

Complete Script Anatomy

2.1.1 The Header Section

The header contains directives and declarations:

- **Version directive** — Always `%dw 2.0`
- **Output directive** — Format of the result
- **Variables** — Declared with `var`
- **Functions** — Declared with `fun`
- **Imports** — External modules
- **Type definitions** — Custom types

Concept

Think of the header as your workspace setup. You're declaring tools (variables and functions) and specifying how the final product should look (output directive).

2.1.2 Output Directives

The output directive tells DataWeave what format to produce:

```
output application/json          // JSON output
output application/xml           // XML output
output application/csv           // CSV output
output text/plain               // Plain text
output application/java          // Java objects
```

Different Output Formats

Same transformation, different outputs:

JSON Output:

```
%dw 2.0
output application/json
---
{
    name: "Alice",
    age: 25
}
```

Result: `{"name":"Alice","age":25}`

XML Output:

```
%dw 2.0
output application/xml
---
{
    person: {
        name: "Alice",
        age: 25
    }
}
```

Result:

```
<?xml version='1.0' encoding='UTF-8'?>
<person>
    <name>Alice</name>
    <age>25</age>
</person>
```

2.2 Variables

Variables store values for reuse. They're declared in the header with `var`.

2.2.1 Variable Declaration

```
%dw 2.0
output application/json
var username = "john_doe"
var userAge = 30
var isActive = true
---
{
    user: username,
    age: userAge,
    active: isActive
}
```

2.2.2 Why Use Variables?

Variables improve:

1. **Readability** — Name complex expressions
2. **Reusability** — Use the same value multiple times
3. **Performance** — Calculate once, use many times
4. **Maintainability** — Change in one place

Variables for Clarity

Without variables (harder to read):

```
%dw 2.0
output application/json
---
{
    totalPrice: (payload.items map $.price) reduce ($$ + $),
    discountedPrice: ((payload.items map $.price) reduce ($$
       + $)) * 0.9
}
```

With variables (clear and DRY):

```
%dw 2.0
output application/json
var total = (payload.items map $.price) reduce ($$ + $)
var discountRate = 0.1
---
{
    totalPrice: total,
    discountedPrice: total * (1 - discountRate)
}
```

Teacher's Tip

Use descriptive variable names. `var t = 100` is unclear. `var totalAmount = 100` is self-documenting.

2.2.3 Scoped Variables

Variables can be declared in the body using `do`:

```
%dw 2.0
output application/json
---
{
    result: do {
        var x = 10
        var y = 20
        ---
        x + y
    }
}
```

2.3 Expressions

Everything in DataWeave is an expression — a piece of code that produces a value.

2.3.1 Types of Expressions

Type	Example	Result
Literal	`42`	Number 42
String	`"Hello"`	String Hello
Object	`{a: 1}`	Object with field a
Array	`[1, 2, 3]`	Array of numbers
Field Access	`payload.name`	Value of name field
Function Call	`upper("abc")`	ABC
Operator	`5 + 3`	8

Table 2.1: Expression Types

2.3.2 Combining Expressions

Expressions can be nested and combined:

```
%dw 2.0
output application/json
---
{
    // Combining field access with function
```

```
    upperName: upper(payload.name),

    // Combining arithmetic and function
    doubleAge: (payload.age * 2) as String,

    // Nested object expressions
    address: {
        full: payload.street ++ ", " ++ payload.city
    }
}
```

2.4 Operators

DataWeave provides operators for various operations.

2.4.1 Arithmetic Operators

Operator	Description	Example
+	Addition	5 + 3 = 8
-	Subtraction	5 - 3 = 2
*	Multiplication	5 * 3 = 15
/	Division	10 / 2 = 5

Table 2.2: Arithmetic Operators

2.4.2 String Operators

```
%dw 2.0
output application/json
---
{
    // Concatenation
    fullName: "John" ++ " " ++ "Doe",

    // Interpolation
    greeting: "Hello, $(payload.name)!",

    // Multi-line
    description: "This is
    a multi-line
    string"
}
```

Teacher's Tip

String interpolation with `$()` is cleaner than concatenation for complex strings. Use it when embedding variables in text.

2.4.3 Comparison Operators

Operator	Description	Example
==	Equals	5 == 5 → true
!=	Not equals	5 != 3 → true
>	Greater than	5 > 3 → true
<	Less than	3 < 5 → true
>=	Greater or equal	5 >= 5 → true
<=	Less or equal	3 <= 5 → true

Table 2.3: Comparison Operators

2.4.4 Logical Operators

```
%dw 2.0
output application/json
---
{
    // AND operator
    isAdultActive: payload.age >= 18 and payload.active,

    // OR operator
    needsReview: payload.age < 18 or payload.score < 50,

    // NOT operator
    isInactive: not payload.active
}
```

2.4.5 Selector Operators

These access parts of data structures:

Operator	Description	Example
.	Single-value selector	payload.name
.*	Multi-value selector	payload.users.*name
..	Descendant selector	payload..email
[]	Index/filter selector	payload.items[0]
[&]	Key-value selector	payload[&]

Table 2.4: Selector Operators

Selector Examples

Input:

```
{
    "company": {
        "departments": [
```

```
            {
                "name": "IT",
                "employees": [
                    {"name": "Alice", "email": "alice@co.com"},
                    {"name": "Bob", "email": "bob@co.com"}
                ]
            }
        ]
    }
}
```

Transformations:

```
%dw 2.0
output application/json
---
{
    // Single-value: get first department name
    dept: payload.company.departments[0].name,

    // Multi-value: get all employee names
    allNames: payload.company.departments.*employees.*name,

    // Descendant: get all emails anywhere
    allEmails: payload..email
}
```

Output:

```
{
    "dept": "IT",
    "allNames": [["Alice", "Bob"]],
    "allEmails": ["alice@co.com", "bob@co.com"]
}
```

2.4.6 Default Value Operator

The `default` operator provides fallback values:

```
%dw 2.0
output application/json
---
{
    username: payload.username default "anonymous",
    age: payload.age default 0,
    tags: payload.tags default []
}
```

Warning

`default` only triggers when the value is `null`. Empty strings or zero values won't trigger the default. Use `isEmpty()` for those cases.

2.5 Comments

Comments document your code:

```
%dw 2.0
output application/json
// This is a single-line comment
---
{
    /* This is a
       multi-line comment */
    name: payload.firstName  // Inline comment
}
```

Teacher's Tip

Write comments that explain *why*, not *what*. The code shows what it does. Comments should explain business logic or complex decisions.

2.6 Common Syntax Mistakes

2.6.1 Missing Separator

Warning

Wrong:

```
%dw 2.0
output application/json
{
    name: "Alice"
}
```

Correct:

```
%dw 2.0
output application/json
---
{
    name: "Alice"
}
```

The `---` separator is mandatory between header and body.

2.6.2 Incorrect String Concatenation

Warning

Wrong:

```
name: "Hello" + "World"  // + doesn't work for strings
```

Correct:

```
name: "Hello" ++ "World"  // Use ++ for concatenation
```

2.6.3 Accessing Non-Existent Fields

```
// If payload doesn't have 'address' field
address: payload.address  // Returns null, might cause issues

// Better approach
address: payload.address default "Not provided"
```

2.7 Summary

Summary

In this chapter, you learned:

- DataWeave scripts have header, separator, and body
- Variables are declared with `var`
- Everything is an expression that produces a value
- Operators include arithmetic, string, comparison, and logical
- Selectors access nested data structures
- The `default` operator provides fallback values
- Comments use `//` for single-line and `/* */` for multi-line

2.8 Practice Exercises

1. Write a script that concatenates first and last name with a space
2. Create a transformation that calculates price after 15% discount
3. Use the `default` operator to provide default values for missing fields
4. Extract all email addresses from a nested object structure

Chapter 3

Data Types

3.1 Understanding Types in DataWeave

DataWeave is strongly typed, meaning every value has a specific type. Understanding types helps you write correct transformations and avoid runtime errors.

Concept

Types define what kind of data you're working with and what operations are valid. You can't multiply a string by a number, but you can concatenate strings or add numbers. Types enforce these rules.

3.2 Core Data Types

DataWeave supports these fundamental types:

- String
- Number
- Boolean
- Null
- Object
- Array
- Date/Time types
- Binary
- Range

3.3 Strings

Strings represent text data.

3.3.1 String Literals

```
%dw 2.0
output application/json
---
{
    // Double quotes
    name: "Alice",

    // Single quotes
    title: 'Developer',

    // Multi-line string
    description: "This is a
    multi-line
    string",

    // Empty string
    empty: ""
}
```

3.3.2 String Interpolation

Embed expressions inside strings:

```
%dw 2.0
output application/json
var firstName = "John"
var age = 30
---
{
    // String interpolation with $()
    message: "Hello, $(firstName)! You are $(age) years old.",

    // Expressions inside interpolation
    yearsToRetirement: "$(65 - age) years until retirement"
}
```

Output:

```
{
    "message": "Hello, John! You are 30 years old.",
    "yearsToRetirement": "35 years until retirement"
}
```

3.3.3 String Functions

```
%dw 2.0
output application/json
---
```

```
{
    uppercase: upper("hello"),            // "HELLO"
    lowercase: lower("HELLO"),            // "hello"
    capitalized: capitalize("hello"),     // "Hello"
    trimmed: trim("  hello  "),           // "hello"
    length: sizeOf("hello"),              // 5
    substring: "hello"[0 to 2],           // "hel"
    contains: "hello" contains "ell",     // true
    startsWith: "hello" startsWith "he", // true
    endsWith: "hello" endsWith "lo"       // true
}
```

String Manipulation

Transform customer names to proper format:

Input:

```
{
    "firstName": "  john  ",
    "lastName": "  DOE  "
}
```

DataWeave:

```
%dw 2.0
output application/json
---
{
    fullName: capitalize(trim(payload.firstName)) ++
              " " ++
              capitalize(lower(trim(payload.lastName)))
}
```

Output:

```
{
    "fullName": "John Doe"
}
```

3.4 Numbers

DataWeave supports integers and decimals.

3.4.1 Number Literals

```
%dw 2.0
output application/json
---
{
    integer: 42,
```

```
    decimal: 3.14,
    negative: -10,
    scientific: 1.5e3,      // 1500
    zero: 0
}
```

3.4.2 Arithmetic Operations

```
%dw 2.0
output application/json
var price = 100
var quantity = 3
var taxRate = 0.08
---
{
    subtotal: price * quantity,                // 300
    tax: price * quantity * taxRate,           // 24
    total: (price * quantity) * (1 + taxRate), // 324
    average: (100 + 200 + 300) / 3,           // 200
    remainder: 17 mod 5                        // 2
}
```

3.4.3 Number Functions

```
%dw 2.0
output application/json
---
{
    // Rounding
    rounded: round(3.7),         // 4
    floor: floor(3.7),           // 3
    ceiling: ceil(3.2),          // 4

    // Absolute value
    absolute: abs(-42),          // 42

    // Min/Max
    minimum: min([3, 1, 4, 1, 5]),  // 1
    maximum: max([3, 1, 4, 1, 5]),  // 5

    // Random
    randomNum: random()          // Random between 0 and 1
}
```

Teacher's Tip

When working with money, be careful with decimal precision. Use explicit rounding to avoid floating-point errors. Consider using integers for cents rather than decimals for dollars.

3.5 Booleans

Booleans represent true or false values.

3.5.1 Boolean Literals

```
%dw 2.0
output application/json
---
{
    isActive: true,
    isDeleted: false,
    hasAccess: payload.role == "admin"
}
```

3.5.2 Boolean Logic

```
%dw 2.0
output application/json
var age = 25
var isStudent = true
---
{
    // AND - both must be true
    qualifiesForDiscount: age < 30 and isStudent,  // true

    // OR - at least one must be true
    needsVerification: age < 18 or age > 65,       // false

    // NOT - inverts the value
    isAdult: not (age < 18),                       // true

    // Complex logic
    canVote: age >= 18 and not isStudent           // false
}
```

3.5.3 Truthiness and Falsiness

Warning

In DataWeave, only explicit `false` and `null` are falsy. Empty strings, zero, and empty arrays are **truthy**.

```
"" as Boolean        // true (not false!)
0 as Boolean         // true (not false!)
[] as Boolean        // true (not false!)
null as Boolean      // false
false as Boolean     // false
```

This is different from JavaScript or Python. Be explicit in conditions.

3.6 Null

`null` represents the absence of a value.

3.6.1 Working with Null

```
%dw 2.0
output application/json
---
{
    // Accessing non-existent field returns null
    missing: payload.nonExistent,

    // Using default for null values
    username: payload.username default "anonymous",

    // Checking for null
    hasEmail: payload.email != null,

    // Null-safe navigation with default
    city: payload.address.city default "Unknown"
}
```

Teacher's Tip

Always use `default` when accessing fields that might not exist. This prevents `null` from propagating through your transformation and causing issues.

3.7 Objects

Objects are key-value pairs, similar to JSON objects or dictionaries.

3.7.1 Object Literals

```
%dw 2.0
output application/json
---
{
    // Simple object
    name: "Alice",
    age: 30,

    // Nested object
    address: {
```

```
        street: "123 Main St",
        city: "Boston",
        country: "USA"
    },

    // Dynamic keys
    (payload.keyName): "dynamicValue",

    // Conditional fields
    (email: payload.email) if payload.email != null
}
```

3.7.2 Accessing Object Fields

```
%dw 2.0
output application/json
var person = {
    name: "Bob",
    age: 25,
    address: {
        city: "NYC"
    }
}
---
{
    // Dot notation
    personName: person.name,

    // Bracket notation (for dynamic keys)
    personAge: person["age"],

    // Nested access
    city: person.address.city,

    // Safe navigation with default
    zipCode: person.address.zip default "00000"
}
```

3.7.3 Object Functions

```
%dw 2.0
output application/json
var person = {
    name: "Alice",
    age: 30,
    city: "Boston"
}
---
{
```

```
    // Get all keys
    keys: keysOf(person),                    // ["name", "age", "city
        "]

    // Get all values
    values: valuesOf(person),                // ["Alice", 30, "Boston
        "]

    // Number of fields
    fieldCount: sizeOf(person),              // 3

    // Check if key exists
    hasAge: person.age != null               // true
}
```

3.7.4 Building Objects Dynamically

Dynamic Object Construction

Input:

```
{
    "fields": [
        {"key": "username", "value": "alice123"},
        {"key": "email", "value": "alice@example.com"}
    ]
}
```

DataWeave:

```
%dw 2.0
output application/json
---
{
    // Convert array of key-value pairs to object
    (payload.fields map {
        ($.key): $.value
    })
}
```

Output:

```
{
    "username": "alice123",
    "email": "alice@example.com"
}
```

3.8 Arrays

Arrays are ordered collections of values.

3.8.1 Array Literals

```
%dw 2.0
output application/json
---
{
    // Simple array
    numbers: [1, 2, 3, 4, 5],

    // Mixed types (valid but not recommended)
    mixed: [1, "hello", true, null],

    // Empty array
    empty: [],

    // Array of objects
    users: [
        {name: "Alice", age: 30},
        {name: "Bob", age: 25}
    ]
}
```

3.8.2 Accessing Array Elements

```
%dw 2.0
output application/json
var numbers = [10, 20, 30, 40, 50]
---
{
    // Index access (0-based)
    first: numbers[0],          // 10
    third: numbers[2],          // 30
    last: numbers[-1],          // 50 (negative index from end)

    // Range selection
    firstThree: numbers[0 to 2],    // [10, 20, 30]
    lastTwo: numbers[-2 to -1],     // [40, 50]

    // Array size
    count: sizeOf(numbers)      // 5
}
```

3.8.3 Array Functions

```
%dw 2.0
output application/json
var numbers = [3, 1, 4, 1, 5, 9, 2, 6]
---
{
```

```
    // Check if empty
    isEmpty: isEmpty(numbers),          // false

    // Contains
    hasThree: numbers contains 3,       // true

    // Distinct values
    unique: distinctBy(numbers, $),     // [3, 1, 4, 5, 9, 2, 6]

    // Join to string
    asString: numbers joinBy ", ",      // "3, 1, 4, 1, 5, 9, 2,
       6"

    // Flatten nested arrays
    flattened: flatten([[1,2], [3,4]]), // [1, 2, 3, 4]

    // Reverse
    reversed: [1, 2, 3] reverse,        // [3, 2, 1]

    // Sorted
    sorted: orderBy(numbers, $)         // [1, 1, 2, 3, 4, 5, 6,
       9]
}
```

3.8.4 Concatenating Arrays

```
%dw 2.0
output application/json
---
{
    // Using ++ operator
    combined: [1, 2, 3] ++ [4, 5, 6],   // [1, 2, 3, 4, 5, 6]

    // Using flatten
    flattened: flatten([[1, 2], [3, 4], [5]]) // [1, 2, 3, 4, 5]
}
```

3.9 Date and Time Types

DataWeave has specialized types for dates and times.

3.9.1 Date/Time Literals

```
%dw 2.0
output application/json
---
{
    // Current date/time
```

```
    now: now(),

    // Date literal
    birthDate: |2000-01-15|,

    // DateTime literal
    appointmentTime: |2024-03-15T14:30:00|,

    // Time literal
    meetingTime: |14:30:00|,

    // With timezone
    utcTime: |2024-03-15T14:30:00Z|
}
```

3.9.2 Date/Time Functions

```
%dw 2.0
output application/json
---
{
    // Current timestamp
    currentTime: now(),

    // Format date
    formatted: now() as String {format: "yyyy-MM-dd"},

    // Extract components
    year: now().year,
    month: now().month,
    day: now().day,
    hour: now().hour,

    // Date arithmetic
    tomorrow: now() + |P1D|,    // Plus 1 day
    nextWeek: now() + |P7D|,    // Plus 7 days

    // Difference between dates
    daysBetween: (|2024-12-31| - |2024-01-01|).days
}
```

Date Formatting

```
%dw 2.0
output application/json
---
{
    // Different formats
    iso: now() as String {format: "yyyy-MM-dd"},
    us: now() as String {format: "MM/dd/yyyy"},
```

```
    readable: now() as String {format: "MMMM dd, yyyy"},
    withTime: now() as String {format: "yyyy-MM-dd HH:mm:ss"}
}
```

Output:

```
{
    "iso": "2024-03-15",
    "us": "03/15/2024",
    "readable": "March 15, 2024",
    "withTime": "2024-03-15 14:30:45"
}
```

3.10 Type Coercion

Converting between types using the `as` operator.

3.10.1 Explicit Coercion

```
%dw 2.0
output application/json
---
{
    // String to Number
    numberFromString: "42" as Number,           // 42

    // Number to String
    stringFromNumber: 42 as String,             // "42"

    // String to Boolean
    boolFromString: "true" as Boolean,          // true

    // Date to String
    dateString: |2024-03-15| as String,         // "2024-03-15"

    // String to Date
    dateFromString: "2024-03-15" as Date,       // |2024-03-15|

    // Object to String (JSON)
    jsonString: {name: "Alice"} as String       // '{"name":"
       Alice"}'
}
```

3.10.2 Coercion with Format

```
%dw 2.0
output application/json
```

```
---
{
    // Format number with decimal places
    formatted: 3.14159 as String {format: "0.00"},

    // Format date with pattern
    dateFormatted: |2024-03-15| as String {format: "MM/dd/yyyy"},

    // Parse date with pattern
    parsedDate: "15/03/2024" as Date {format: "dd/MM/yyyy"}
}
```

3.10.3 Type Checking

```
%dw 2.0
output application/json
var value = "hello"
---
{
    // Check type
    isString: value is String,          // true
    isNumber: value is Number,          // false
    isObject: value is Object,          // false
    isArray: value is Array             // false
}
```

Warning

Failed coercion throws an error. Always validate data before coercing:

```
// This will fail if input is not numeric
age: payload.age as Number

// Better approach
age: if (payload.age is Number)
        payload.age
    else
        payload.age as Number
```

3.11 Range Type

Ranges represent sequences of numbers.

```
%dw 2.0
output application/json
---
{
    // Create array from range
    numbers: 1 to 5,                    // [1, 2, 3, 4, 5]
```

```
7
8       // Use in iteration
9       squares: (1 to 5) map $ * $,          // [1, 4, 9, 16, 25]
10
11      // Reverse range
12      countdown: 5 to 1,                    // [5, 4, 3, 2, 1]
13
14      // Range with step
15      evens: (0 to 10) filter ($ mod 2 == 0)  // [0, 2, 4, 6, 8,
            10]
16  }
```

3.12 Common Type Mistakes

3.12.1 String vs Number Comparison

Warning

```
1  // Wrong - comparing different types
2  age: "30" > 25           // Error or unexpected result
3
4  // Correct - coerce first
5  age: ("30" as Number) > 25    // true
```

3.12.2 Null Propagation

Warning

```
1  // If payload.user is null, this fails
2  name: upper(payload.user.name)
3
4  // Correct - handle null
5  name: if (payload.user != null)
6            upper(payload.user.name)
7        else
8            "Unknown"
9
10 // Or use default
11 name: upper(payload.user.name default "Unknown")
```

3.13 Summary

Summary

In this chapter, you learned:

- DataWeave has strong typing: String, Number, Boolean, Null, Object, Array
- String interpolation uses `$()`
- Objects are key-value pairs, accessed with dot or bracket notation
- Arrays are ordered collections, indexed from 0
- Date/Time types have specialized functions and formatting
- Use `as` operator for explicit type coercion
- Always handle potential `null` values with `default`
- Type checking uses `is` operator

3.14 Practice Exercises

1. Convert a string date ”2024-03-15” to Date type and format as ”March 15, 2024”
2. Create an object dynamically from an array of key-value pairs
3. Extract unique values from an array and sort them
4. Calculate the number of days between two dates
5. Build a full name from first/middle/last name fields, handling null values

Chapter 4

Core Functions

4.1 Introduction to Functional Programming

DataWeave is a functional language. Instead of loops and mutations, you use functions that transform data. This chapter covers the essential functions every DataWeave developer must master.

Concept

Functional programming treats data transformation as a pipeline. Data flows through functions, each transforming it in some way. Think of it like an assembly line where each station modifies the product.

4.2 Map Function

`map` transforms each element in an array.

4.2.1 Basic Syntax

```
array map expression
```

Inside `map`:

- `$` — Current element
- `$$` — Current index (0-based)
- `$$$` — Total number of elements

4.2.2 Simple Mapping

Double Each Number

Input:

```
[1, 2, 3, 4, 5]
```

DataWeave:

```
%dw 2.0
output application/json
---
payload map ($ * 2)
```

Output:

```
[2, 4, 6, 8, 10]
```

4.2.3 Mapping Objects

Extract Names from Users

Input:

```
{
    "users": [
        {"id": 1, "name": "Alice", "age": 30},
        {"id": 2, "name": "Bob", "age": 25},
        {"id": 3, "name": "Charlie", "age": 35}
    ]
}
```

DataWeave:

```
%dw 2.0
output application/json
---
payload.users map $.name
```

Output:

```
["Alice", "Bob", "Charlie"]
```

4.2.4 Transforming Object Structure

Reshape User Data

Input:

```
{
    "users": [
        {"firstName": "Alice", "lastName": "Smith", "age":
           30},
        {"firstName": "Bob", "lastName": "Jones", "age": 25}
    ]
}
```

DataWeave:

```
%dw 2.0
output application/json
---
payload.users map {
    fullName: $.firstName ++ " " ++ $.lastName,
    age: $.age,
    isAdult: $.age >= 18
}
```

Output:

```
[
    {
        "fullName": "Alice Smith",
        "age": 30,
        "isAdult": true
    },
    {
        "fullName": "Bob Jones",
        "age": 25,
        "isAdult": true
    }
]
```

4.2.5 Using Index and Total

```
%dw 2.0
output application/json
---
["a", "b", "c"] map {
    value: $,
    index: $$,
    total: $$$,
    position: "$($$+1) of $($$$)"
}
```

Output:

```
[
    {"value": "a", "index": 0, "total": 3, "position": "1 of 3"},
    {"value": "b", "index": 1, "total": 3, "position": "2 of 3"},
    {"value": "c", "index": 2, "total": 3, "position": "3 of 3"}
]
```

Teacher's Tip

Use named parameters for clarity when `map` logic gets complex:

```
users map (user, index) -> {
    name: user.name,
```

```
    position: index + 1
}
```

4.3 Filter Function

`filter` keeps only elements that match a condition.

4.3.1 Basic Syntax

```
array filter condition
```

4.3.2 Simple Filtering

Filter Even Numbers

Input:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

DataWeave:

```
%dw 2.0
output application/json
---
payload filter ($ mod 2 == 0)
```

Output:

```
[2, 4, 6, 8, 10]
```

4.3.3 Filtering Objects

Filter Active Users

Input:

```
{
    "users": [
        {"name": "Alice", "active": true, "age": 30},
        {"name": "Bob", "active": false, "age": 25},
        {"name": "Charlie", "active": true, "age": 35}
    ]
}
```

DataWeave:

```
%dw 2.0
output application/json
```

```
---
payload.users filter $.active
```

Output:

```
[
    {"name": "Alice", "active": true, "age": 30},
    {"name": "Charlie", "active": true, "age": 35}
]
```

4.3.4 Complex Filters

```
%dw 2.0
output application/json
---
{
    // Multiple conditions with AND
    adults: payload.users filter ($.age >= 18 and $.active),

    // Multiple conditions with OR
    special: payload.users filter ($.age < 18 or $.age > 65),

    // Field existence
    withEmail: payload.users filter ($.email != null),

    // String matching
    searchResults: payload.users filter ($.name contains "Alice")
}
```

4.3.5 Combining Map and Filter

Filter Then Transform

Get names of active adult users:

```
%dw 2.0
output application/json
---
payload.users
    filter ($.active and $.age >= 18)
    map $.name
```

Output:

```
["Alice", "Charlie"]
```

Teacher's Tip

Order matters! Filter before map to reduce processing:

```
// Efficient - filter reduces data first
users filter $.active map transform($)

// Inefficient - map processes everything
users map transform($) filter $.active
```

4.4 Reduce Function

`reduce` combines all elements into a single value (aggregation).

4.4.1 Basic Syntax

```
array reduce ((item, accumulator) -> expression)
```

- `item` — Current element (or `$`)
- `accumulator` — Running total (or `$$`)

4.4.2 Sum Example

Calculate Total

Input:

```
[10, 20, 30, 40, 50]
```

DataWeave:

```
%dw 2.0
output application/json
---
{
    total: payload reduce ($$ + $)
}
```

Output:

```
{
    "total": 150
}
```

How it works:

```
Step 1: accumulator = 10, item = 20  -> 10 + 20 = 30
Step 2: accumulator = 30, item = 30  -> 30 + 30 = 60
Step 3: accumulator = 60, item = 40  -> 60 + 40 = 100
Step 4: accumulator = 100, item = 50 -> 100 + 50 = 150
```

4.4.3 Reduce with Initial Value

```
%dw 2.0
output application/json
---
{
    // Start from 100
    totalWithBonus: ([10, 20, 30] reduce ($$ + $)) + 100
}
```

4.4.4 Building Objects with Reduce

Convert Array to Object

Input:

```
[
    {"key": "name", "value": "Alice"},
    {"key": "age", "value": 30},
    {"key": "city", "value": "Boston"}
]
```

DataWeave:

```
%dw 2.0
output application/json
---
payload reduce ((item, acc = {}) ->
    acc ++ {(item.key): item.value}
)
```

Output:

```
{
    "name": "Alice",
    "age": 30,
    "city": "Boston"
}
```

4.4.5 Common Reduce Patterns

```
%dw 2.0
output application/json
var numbers = [5, 2, 8, 1, 9, 3]
var items = [
    {name: "A", price: 10},
    {name: "B", price: 20},
    {name: "C", price: 15}
]
---
```

```
{
    // Sum
    sum: numbers reduce ($$ + $),

    // Product
    product: numbers reduce ($$ * $),

    // Max value
    max: numbers reduce (
        if ($ > $$) $ else $$
    ),

    // Concatenate strings
    names: items map $.name reduce ($$ ++ ", " ++ $),

    // Total price
    totalPrice: (items map $.price) reduce ($$ + $)
}
```

Teacher's Tip

For simple operations, consider using built-in functions instead of `reduce`:

```
// Instead of reduce for sum
numbers reduce ($$ + $)

// Use sum function
sum(numbers)

// Instead of reduce for max
numbers reduce (if ($ > $$) $ else $$)

// Use max function
max(numbers)
```

4.5 Pluck Function

`pluck` transforms an object into an array by accessing keys and values.

4.5.1 Basic Syntax

```
object pluck expression
```

Inside `pluck`:

- `$` — Value
- `$$` — Key
- `$$$` — Index

4.5.2 Simple Pluck

Get All Values

Input:

```
{
    "name": "Alice",
    "age": 30,
    "city": "Boston"
}
```

DataWeave:

```
%dw 2.0
output application/json
---
{
    values: payload pluck $,
    keys: payload pluck $$,
    keyValuePairs: payload pluck {key: $$, value: $}
}
```

Output:

```
{
    "values": ["Alice", 30, "Boston"],
    "keys": ["name", "age", "city"],
    "keyValuePairs": [
        {"key": "name", "value": "Alice"},
        {"key": "age", "value": 30},
        {"key": "city", "value": "Boston"}
    ]
}
```

4.5.3 Pluck with Nested Objects

Extract Department Info

Input:

```
{
    "departments": {
        "IT": {"budget": 100000, "employees": 10},
        "HR": {"budget": 50000, "employees": 5},
        "Sales": {"budget": 150000, "employees": 20}
    }
}
```

DataWeave:

```
%dw 2.0
output application/json
```

```
---
payload.departments pluck {
    name: $$,
    budget: $.budget,
    employees: $.employees
}
```

Output:

```
[
    {"name": "IT", "budget": 100000, "employees": 10},
    {"name": "HR", "budget": 50000, "employees": 5},
    {"name": "Sales", "budget": 150000, "employees": 20}
]
```

Teacher's Tip

`pluck` is perfect for converting objects to arrays when you need both keys and values. It's like turning a dictionary into a list of entries.

4.6 GroupBy Function

`groupBy` organizes array elements into groups based on a criterion.

4.6.1 Basic Syntax

```
array groupBy expression
```

4.6.2 Simple Grouping

Group by Category

Input:

```
{
    "products": [
        {"name": "Laptop", "category": "Electronics", "price"
            : 1000},
        {"name": "Shirt", "category": "Clothing", "price":
            50},
        {"name": "Phone", "category": "Electronics", "price":
             500},
        {"name": "Jeans", "category": "Clothing", "price":
            80}
    ]
}
```

DataWeave:

```
%dw 2.0
output application/json
---
payload.products groupBy $.category
```

Output:

```
{
    "Electronics": [
        {"name": "Laptop", "category": "Electronics", "price": 1000},
        {"name": "Phone", "category": "Electronics", "price": 500}
    ],
    "Clothing": [
        {"name": "Shirt", "category": "Clothing", "price": 50},
        {"name": "Jeans", "category": "Clothing", "price": 80}
    ]
}
```

4.6.3 Advanced Grouping

Group and Calculate Totals

```
%dw 2.0
output application/json
---
(payload.products groupBy $.category) pluck {
    category: $$,
    items: $,
    totalValue: sum($ map $.price),
    itemCount: sizeOf($)
}
```

Output:

```
[
    {
        "category": "Electronics",
        "items": [...],
        "totalValue": 1500,
        "itemCount": 2
    },
    {
        "category": "Clothing",
        "items": [...],
        "totalValue": 130,
        "itemCount": 2
```

```
    }
]
```

4.6.4 Grouping by Expression

```
%dw 2.0
output application/json
var users = [
    {name: "Alice", age: 25},
    {name: "Bob", age: 35},
    {name: "Charlie", age: 28},
    {name: "David", age: 42}
]
---
{
    // Group by age range
    byAgeGroup: users groupBy (
        if ($.age < 30) "young"
        else if ($.age < 40) "middle"
        else "senior"
    )
}
```

4.7 DistinctBy Function

`distinctBy` removes duplicates based on a criterion.

4.7.1 Basic Syntax

```
array distinctBy expression
```

4.7.2 Simple Deduplication

Remove Duplicate Numbers

Input:

```
[1, 2, 2, 3, 3, 3, 4, 5, 5]
```

DataWeave:

```
%dw 2.0
output application/json
---
payload distinctBy $
```

Output:

```
[1, 2, 3, 4, 5]
```

4.7.3 Distinct by Field

Unique Users by Email

Input:

```
{
    "users": [
        {"name": "Alice", "email": "alice@example.com"},
        {"name": "Alice Smith", "email": "alice@example.com"
           },
        {"name": "Bob", "email": "bob@example.com"}
    ]
}
```

DataWeave:

```
%dw 2.0
output application/json
---
payload.users distinctBy $.email
```

Output:

```
[
    {"name": "Alice", "email": "alice@example.com"},
    {"name": "Bob", "email": "bob@example.com"}
]
```

4.7.4 Distinct by Expression

```
%dw 2.0
output application/json
---
{
    // Unique by lowercase name
    uniqueNames: payload.users distinctBy lower($.name),

    // Unique by first character
    uniqueFirstLetter: payload.users distinctBy $.name[0],

    // Unique by rounded age
    uniqueAgeGroups: payload.users distinctBy ($.age / 10)
}
```

4.8 Chaining Functions

The real power comes from combining functions.

Complete Data Pipeline

Task: From orders, get unique product categories where total value exceeds $100

Input:

```
{
    "orders": [
        {
            "customer": "Alice",
            "items": [
                {"product": "Laptop", "category": "
                   Electronics", "price": 1000},
                {"product": "Mouse", "category": "Electronics
                   ", "price": 25}
            ]
        },
        {
            "customer": "Bob",
            "items": [
                {"product": "Shirt", "category": "Clothing",
                   "price": 50}
            ]
        }
    ]
}
```

DataWeave:

```
%dw 2.0
output application/json
---
payload.orders
    // Flatten all items from all orders
    flatMap $.items
    // Group by category
    groupBy $.category
    // Convert to array and calculate totals
    pluck {
        category: $$,
        total: sum($ map $.price)
    }
    // Filter where total > 100
    filter $.total > 100
    // Get just the category names
    map $.category
    // Remove duplicates
    distinctBy $
```

Output:

```
["Electronics"]
```

Teacher's Tip

When chaining:

1. Filter early to reduce data volume
2. Group before aggregating
3. Transform last to avoid unnecessary processing
4. Read the chain top-to-bottom like a story

4.9 Summary

Summary

In this chapter, you learned:

- **map** — Transform each element in an array
- **filter** — Keep elements matching a condition
- **reduce** — Combine all elements into a single value
- **pluck** — Convert object to array with keys and values
- **groupBy** — Organize elements into groups
- **distinctBy** — Remove duplicates based on criterion
- Functions chain to create powerful data pipelines
- Inside functions: `$` = item, `$$` = key/accumulator, `$$$` = index/total

4.10 Practice Exercises

1. Transform an array of prices to include tax (multiply by 1.08)
2. Filter users who are active AND age between 25-35
3. Calculate average age using `reduce`
4. Convert object with department names as keys to array of department objects
5. Group products by price range: cheap (<50), medium (50-200), expensive (>200)
6. Get unique cities from array of addresses

7. Chain functions to: filter active users, group by department, calculate total salary per department

Chapter 5

Data Format Transformations

5.1 Understanding Format Conversion

One of DataWeave's primary purposes is converting data between formats. In this chapter, you'll learn to transform between JSON, XML, and CSV.

Concept

Different systems speak different formats. APIs typically use JSON, enterprise systems often use XML, and data exports frequently use CSV. DataWeave acts as the universal translator.

5.2 JSON to JSON Transformation

Even within the same format, data often needs restructuring.

5.2.1 Simple Restructuring

Flatten Nested Structure

Input:

```
{
    "user": {
        "personal": {
            "firstName": "Alice",
            "lastName": "Smith"
        },
        "contact": {
            "email": "alice@example.com",
            "phone": "555-0123"
        }
    }
}
```

DataWeave:

```
%dw 2.0
```

```
output application/json
---
{
    firstName: payload.user.personal.firstName,
    lastName: payload.user.personal.lastName,
    email: payload.user.contact.email,
    phone: payload.user.contact.phone
}
```

Output:

```
{
    "firstName": "Alice",
    "lastName": "Smith",
    "email": "alice@example.com",
    "phone": "555-0123"
}
```

5.2.2 Nested to Flat Array

Extract All Products

Input:

```
{
    "orders": [
        {
            "orderId": 1,
            "items": [
                {"product": "Laptop", "quantity": 1},
                {"product": "Mouse", "quantity": 2}
            ]
        },
        {
            "orderId": 2,
            "items": [
                {"product": "Keyboard", "quantity": 1}
            ]
        }
    ]
}
```

DataWeave:

```
%dw 2.0
output application/json
---
payload.orders flatMap ((order) ->
    order.items map {
        orderId: order.orderId,
```

```
        product: $.product,
        quantity: $.quantity
    }
)
```

Output:

```
[
    {"orderId": 1, "product": "Laptop", "quantity": 1},
    {"orderId": 1, "product": "Mouse", "quantity": 2},
    {"orderId": 2, "product": "Keyboard", "quantity": 1}
]
```

5.3 JSON to XML Transformation

Converting JSON to XML requires understanding XML structure.

5.3.1 Basic JSON to XML

Simple Conversion

Input (JSON):

```
{
    "customer": {
        "name": "Alice",
        "age": 30,
        "active": true
    }
}
```

DataWeave:

```
%dw 2.0
output application/xml
---
{
    Customer: {
        Name: payload.customer.name,
        Age: payload.customer.age,
        Active: payload.customer.active
    }
}
```

Output (XML):

```
<?xml version='1.0' encoding='UTF-8'?>
<Customer>
    <Name>Alice</Name>
    <Age>30</Age>
```

```
    <Active>true</Active>
</Customer>
```

5.3.2 XML Attributes

Use `@` to create XML attributes:

XML with Attributes

DataWeave:

```
%dw 2.0
output application/xml
---
{
    Customer @(id: "123", type: "premium"): {
        Name: "Alice",
        Age: 30
    }
}
```

Output:

```
<?xml version='1.0' encoding='UTF-8'?>
<Customer id="123" type="premium">
    <Name>Alice</Name>
    <Age>30</Age>
</Customer>
```

5.3.3 Arrays to XML

JSON Array to XML Elements

Input:

```
{
    "users": [
        {"name": "Alice", "age": 30},
        {"name": "Bob", "age": 25}
    ]
}
```

DataWeave:

```
%dw 2.0
output application/xml
---
{
    Users: {
        (payload.users map (user) -> {
```

```
            User: {
                Name: user.name,
                Age: user.age
            }
        })
    }
}
```

Output:

```
<?xml version='1.0' encoding='UTF-8'?>
<Users>
    <User>
        <Name>Alice</Name>
        <Age>30</Age>
    </User>
    <User>
        <Name>Bob</Name>
        <Age>25</Age>
    </User>
</Users>
```

Teacher's Tip

When converting arrays to XML, wrap each array element in parentheses and use `map` to create repeated elements. This prevents DataWeave from creating a single element with array values.

5.3.4 XML Namespaces

```
%dw 2.0
output application/xml
ns soap http://schemas.xmlsoap.org/soap/envelope/
ns cust http://example.com/customer
---
soap#Envelope: {
    soap#Body: {
        cust#Customer: {
            cust#Name: "Alice",
            cust#Age: 30
        }
    }
}
```

5.4 XML to JSON Transformation

Converting XML to JSON is common when exposing legacy systems via modern APIs.

5.4.1 Basic XML to JSON

Simple XML to JSON

Input (XML):

```
<Customer>
    <Name>Alice</Name>
    <Age>30</Age>
    <Active>true</Active>
</Customer>
```

DataWeave:

```
%dw 2.0
output application/json
---
{
    customer: {
        name: payload.Customer.Name,
        age: payload.Customer.Age as Number,
        active: payload.Customer.Active as Boolean
    }
}
```

Output:

```
{
    "customer": {
        "name": "Alice",
        "age": 30,
        "active": true
    }
}
```

Warning

XML doesn't have native types. Everything is a string. Always coerce values to the correct type when converting to JSON:

```
// XML: <Age>30</Age>
age: payload.Age          // String "30"
age: payload.Age as Number  // Number 30
```

5.4.2 XML Attributes to JSON

Handle XML Attributes

Input:

```
<Customer id="123" type="premium">
    <Name>Alice</Name>
</Customer>
```

DataWeave:

```
%dw 2.0
output application/json
---
{
    customer: {
        id: payload.Customer.@id,
        type: payload.Customer.@type,
        name: payload.Customer.Name
    }
}
```

Output:

```
{
    "customer": {
        "id": "123",
        "type": "premium",
        "name": "Alice"
    }
}
```

5.4.3 Repeating XML Elements to Array

XML List to JSON Array

Input:

```
<Users>
    <User>
        <Name>Alice</Name>
        <Age>30</Age>
    </User>
    <User>
        <Name>Bob</Name>
        <Age>25</Age>
    </User>
</Users>
```

DataWeave:

```
%dw 2.0
```

```
output application/json
---
{
    users: payload.Users.*User map {
        name: $.Name,
        age: $.Age as Number
    }
}
```

Output:

```
{
    "users": [
        {"name": "Alice", "age": 30},
        {"name": "Bob", "age": 25}
    ]
}
```

Teacher's Tip

Use the multi-value selector `.*` to extract all elements with the same name into an array. Then use `map` to transform each element.

5.5 CSV Handling

CSV is common for data imports/exports and reports.

5.5.1 CSV to JSON

Parse CSV File

Input (CSV):

```
name,age,city
Alice,30,Boston
Bob,25,NYC
Charlie,35,LA
```

DataWeave:

```
%dw 2.0
output application/json
---
payload map {
    name: $.name,
    age: $.age as Number,
    city: $.city
}
```

Output:

```
[
    {"name": "Alice", "age": 30, "city": "Boston"},
    {"name": "Bob", "age": 25, "city": "NYC"},
    {"name": "Charlie", "age": 35, "city": "LA"}
]
```

5.5.2 JSON to CSV

Generate CSV Report

Input:

```
[
    {"name": "Alice", "age": 30, "city": "Boston"},
    {"name": "Bob", "age": 25, "city": "NYC"}
]
```

DataWeave:

```
%dw 2.0
output application/csv header=true
---
payload map {
    name: $.name,
    age: $.age,
    city: $.city
}
```

Output:

```
name,age,city
Alice,30,Boston
Bob,25,NYC
```

5.5.3 CSV with Custom Delimiter

```
%dw 2.0
output application/csv separator="|", header=true
---
payload
```

Output:

```
name|age|city
Alice|30|Boston
Bob|25|NYC
```

5.6 Complex Transformation Scenarios

5.6.1 API Response Normalization

Normalize Different API Structures

Task: Two APIs return user data differently. Normalize to common format.

API 1 Format:

```
{
    "user_name": "Alice",
    "user_email": "alice@example.com",
    "account_active": "Y"
}
```

API 2 Format:

```
{
    "fullName": "Bob Smith",
    "email": "bob@example.com",
    "isActive": true
}
```

DataWeave (handles both):

```
%dw 2.0
output application/json
---
{
    name: payload.user_name default payload.fullName,
    email: payload.user_email default payload.email,
    active: if (payload.account_active?)
                (payload.account_active == "Y")
            else
                payload.isActive
}
```

5.6.2 Data Enrichment

Enrich Order Data

Input:

```
{
    "order": {
        "items": [
            {"productId": "P1", "quantity": 2},
            {"productId": "P2", "quantity": 1}
        ]
    },
    "products": [
```

```
        {"id": "P1", "name": "Laptop", "price": 1000},
        {"id": "P2", "name": "Mouse", "price": 25}
    ]
}
```

DataWeave:

```
%dw 2.0
output application/json
var productMap = payload.products
    reduce ((item, acc={}) ->
        acc ++ {(item.id): item}
    )
---
{
    order: {
        items: payload.order.items map {
            productId: $.productId,
            productName: productMap[$.productId].name,
            unitPrice: productMap[$.productId].price,
            quantity: $.quantity,
            totalPrice: productMap[$.productId].price * $.
               quantity
        },
        orderTotal: sum(payload.order.items map (
            productMap[$.productId].price * $.quantity
        ))
    }
}
```

Output:

```
{
    "order": {
        "items": [
            {
                "productId": "P1",
                "productName": "Laptop",
                "unitPrice": 1000,
                "quantity": 2,
                "totalPrice": 2000
            },
            {
                "productId": "P2",
                "productName": "Mouse",
                "unitPrice": 25,
                "quantity": 1,
                "totalPrice": 25
            }
        ],
        "orderTotal": 2025
    }
```

```
}
```

5.7 Summary

Summary

In this chapter, you learned:

- JSON to JSON transformations restructure data within the same format
- JSON to XML requires understanding XML structure and namespaces
- XML attributes use `@` syntax
- XML to JSON requires type coercion (XML has only strings)
- CSV parsing automatically creates objects from rows
- CSV generation uses `output application/csv`
- Complex transformations combine multiple techniques
- Lookup tables improve performance in data enrichment

5.8 Practice Exercises

1. Convert a nested JSON user object to flat structure
2. Transform JSON order array to XML with attributes for order IDs
3. Parse XML product catalog to JSON array
4. Generate CSV report from JSON transaction data
5. Create transformation that handles two different API response formats
6. Enrich employee data with department information from lookup table

Chapter 6

Advanced Concepts

6.1 Pattern Matching

Pattern matching allows elegant conditional logic based on data structure.

6.1.1 Basic Match Expression

```
value match {
    case pattern1 -> result1
    case pattern2 -> result2
    else -> defaultResult
}
```

6.1.2 Type Matching

Route by Type

```
%dw 2.0
output application/json
fun processValue(val) = val match {
    case is String -> {type: "text", value: val}
    case is Number -> {type: "number", value: val}
    case is Boolean -> {type: "flag", value: val}
    case is Array -> {type: "list", count: sizeOf(val)}
    case is Object -> {type: "object", keys: keysOf(val)}
    else -> {type: "unknown", value: null}
}
---
{
    result1: processValue("hello"),
    result2: processValue(42),
    result3: processValue([1,2,3])
}
```

Output:

```
{
```

```
    "result1": {"type": "text", "value": "hello"},
    "result2": {"type": "number", "value": 42},
    "result3": {"type": "list", "count": 3}
}
```

6.1.3 Value Matching

```
%dw 2.0
output application/json
fun getDiscount(customerType) = customerType match {
    case "VIP" -> 0.20
    case "Premium" -> 0.15
    case "Regular" -> 0.05
    else -> 0
}
---
{
    vipDiscount: getDiscount("VIP"),
    regularDiscount: getDiscount("Regular")
}
```

6.1.4 Structure Matching

Match Object Structure

```
%dw 2.0
output application/json
fun categorizeEvent(event) = event match {
    case {type: "click", element: e} ->
        "User clicked on $(e)"
    case {type: "submit", form: f} ->
        "Form $(f) submitted"
    case {type: "error", code: c} ->
        "Error occurred: code $(c)"
    else -> "Unknown event"
}
---
{
    event1: categorizeEvent({type: "click", element: "button"
        }),
    event2: categorizeEvent({type: "error", code: 404})
}
```

Teacher's Tip

Pattern matching is cleaner than nested if-else for complex conditions. It makes your intent clear and reduces cognitive load.

6.2 Conditional Logic

6.2.1 If-Else Expressions

```
%dw 2.0
output application/json
---
{
    // Simple if-else
    status: if (payload.age >= 18) "adult" else "minor",

    // Multi-line if-else
    category: if (payload.score >= 90)
                  "Excellent"
              else if (payload.score >= 75)
                  "Good"
              else if (payload.score >= 60)
                  "Average"
              else
                  "Needs Improvement",

    // If without else (returns null if false)
    premium: if (payload.memberYears > 5) "Yes" else null
}
```

6.2.2 Conditional Fields

```
%dw 2.0
output application/json
---
{
    name: payload.name,
    age: payload.age,

    // Include field only if condition is true
    (email: payload.email) if payload.email != null,

    // Include field based on age
    (discount: 0.10) if payload.age >= 65,

    // Multiple conditional fields
    (vipStatus: "Gold") if payload.purchaseTotal > 10000,
    (vipStatus: "Silver") if payload.purchaseTotal > 5000
```

```
}
```

6.2.3 Unless Expression

```
%dw 2.0
output application/json
---
{
    // Execute unless condition is true
    message: "Proceed" unless payload.hasError

    // Equivalent to:
    // message: if (not payload.hasError) "Proceed" else null
}
```

6.3 Functions and Modules

6.3.1 Defining Functions

```
%dw 2.0
output application/json

// Simple function
fun greet(name: String) = "Hello, $(name)!"

// Function with multiple parameters
fun calculateTax(amount: Number, rate: Number) =
    amount * rate

// Function with default parameter
fun formatCurrency(amount: Number, symbol: String = "$") =
    "$(symbol)$(amount as String {format: "0.00"})"

// Multi-line function
fun processUser(user: Object) = {
    fullName: user.firstName ++ " " ++ user.lastName,
    ageGroup: if (user.age < 30) "Young" else "Mature",
    isActive: user.active default false
}
---
{
    greeting: greet("Alice"),
    tax: calculateTax(100, 0.08),
    price1: formatCurrency(99.99),
    price2: formatCurrency(99.99, "€"),
    user: processUser(payload)
}
```

6.3.2 Type Annotations

```
%dw 2.0
output application/json

// Function with type annotations
fun addNumbers(a: Number, b: Number): Number = a + b

// Function with complex types
fun getUserEmail(user: {name: String, email: String}): String =
    user.email

// Function returning Object
fun createResponse(success: Boolean, data: Any): Object = {
    success: success,
    data: data,
    timestamp: now()
}
---
payload
```

6.3.3 Creating Modules

Modules organize reusable functions.

File: utils.dwl

```
%dw 2.0

fun formatName(first: String, last: String) =
    capitalize(trim(first)) ++ " " ++ capitalize(trim(last))

fun calculateDiscount(price: Number, percentage: Number) =
    price * (1 - percentage)

fun isAdult(age: Number) = age >= 18
```

Using the module:

```
%dw 2.0
output application/json
import modules::utils

---
{
    name: utils::formatName(payload.first, payload.last),
    discountedPrice: utils::calculateDiscount(100, 0.15),
    canVote: utils::isAdult(payload.age)
}
```

Teacher's Tip

Create modules for:

- Business logic functions (tax calculations, validation rules)
- String utilities (formatting, sanitization)
- Date utilities (parsing, formatting)
- Common transformations used across projects

6.4 Recursion

Recursion is useful for processing nested structures of unknown depth.

6.4.1 Basic Recursion

Calculate Factorial

```
%dw 2.0
output application/json

fun factorial(n: Number): Number =
    if (n <= 1)
        1
    else
        n * factorial(n - 1)
---
{
    fact5: factorial(5),     // 120
    fact10: factorial(10)    // 3628800
}
```

6.4.2 Recursive Tree Traversal

Find All IDs in Nested Structure

Input:

```
{
    "id": 1,
    "name": "Root",
    "children": [
        {
            "id": 2,
            "name": "Child1",
            "children": [
```

```
                {"id": 3, "name": "Grandchild1"},
                {"id": 4, "name": "Grandchild2"}
            ]
        },
        {
            "id": 5,
            "name": "Child2"
        }
    ]
}
```

DataWeave:

```
%dw 2.0
output application/json

fun getAllIds(node: Object): Array =
    [node.id] ++ (
        if (node.children?)
            flatten(node.children map getAllIds($))
        else
            []
    )
---
{
    allIds: getAllIds(payload)
}
```

Output:

```
{
    "allIds": [1, 2, 3, 4, 5]
}
```

6.4.3 Recursive Flattening

```
%dw 2.0
output application/json

fun deepFlatten(arr: Array): Array =
    arr flatMap ((item) ->
        if (item is Array)
            deepFlatten(item)
        else
            [item]
    )
---
{
    nested: [1, [2, [3, [4, 5]], 6], 7],
    flattened: deepFlatten([1, [2, [3, [4, 5]], 6], 7])
```

```
}
```

Output:

```
{
    "nested": [1, [2, [3, [4, 5]], 6], 7],
    "flattened": [1, 2, 3, 4, 5, 6, 7]
}
```

Warning

Recursion can cause stack overflow with very deep structures. For extremely deep nesting, consider iterative approaches or ensure your data has reasonable depth limits.

6.5 Error Handling

6.5.1 Try-Catch

```
%dw 2.0
output application/json
---
{
    // Handle potential errors
    result: try(() ->
        payload.amount as Number
    ) orElse (error) -> 0,

    // With error message
    safeConversion: try(() ->
        payload.date as Date {format: "yyyy-MM-dd"}
    ) orElse (error) -> {
        success: false,
        error: error.description
    }
}
```

6.5.2 Validation Functions

```
%dw 2.0
output application/json

fun validateEmail(email: String): Boolean =
    email contains "@" and email contains "."

fun validateAge(age: Number): Object =
    if (age >= 0 and age <= 120)
        {valid: true, value: age}
    else
```

```
        {valid: false, error: "Age must be between 0 and 120"}

---
{
    emailCheck: validateEmail(payload.email),
    ageValidation: validateAge(payload.age)
}
```

6.5.3 Safe Navigation

```
%dw 2.0
output application/json
---
{
    // Without safe navigation - can fail
    // city: payload.user.address.city

    // With safe navigation using default
    city: payload.user.address.city default "Unknown",

    // Chained defaults
    contact: payload.email
        default payload.phone
        default "No contact info",

    // Check existence before access
    hasAddress: payload.user.address?
}
```

6.6 Working with Do Blocks

`do` blocks create local scopes for complex logic.

Complex Calculation with Local Variables

```
%dw 2.0
output application/json
---
{
    orderSummary: do {
        var subtotal = sum(payload.items map $.price)
        var taxRate = 0.08
        var tax = subtotal * taxRate
        var shipping = if (subtotal > 100) 0 else 10
        ---
        {
            subtotal: subtotal,
            tax: tax,
            shipping: shipping,
```

```
        total: subtotal + tax + shipping
    }
  }
}
```

Teacher's Tip

Use `do` blocks when:

- You need local variables for complex calculations
- Logic has multiple steps that build on each other
- You want to isolate scope and avoid polluting the header

6.7 Update Operator

The update operator modifies nested structures immutably.

Update Nested Field

Input:

```
{
    "user": {
        "name": "Alice",
        "settings": {
            "theme": "dark",
            "notifications": true
        }
    }
}
```

DataWeave:

```
%dw 2.0
output application/json
---
payload update {
    case .user.settings.theme -> "light"
}
```

Output:

```
{
    "user": {
        "name": "Alice",
        "settings": {
            "theme": "light",
            "notifications": true
        }
```

```
        }
    }
```

6.7.1 Multiple Updates

```
%dw 2.0
output application/json
---
payload update {
    case .user.name -> upper($.user.name)
    case .user.settings.theme -> "light"
    case .user.lastLogin -> now()
}
```

6.8 Summary

Summary

In this chapter, you learned:

- Pattern matching with `match` for elegant conditional logic
- Conditional field inclusion in objects
- Function definition with type annotations
- Creating and importing modules for code reuse
- Recursion for processing nested structures
- Error handling with `try-catch`
- `do` blocks for local scopes
- Update operator for modifying nested data immutably

6.9 Practice Exercises

1. Write a pattern match that routes events based on their structure
2. Create a module with currency formatting functions
3. Implement recursive function to sum all numbers in nested arrays
4. Write validation function with proper error handling
5. Use update operator to modify multiple fields in nested object

6. Create function that safely extracts email from various object structures

Chapter 7

Performance & Best Practices

7.1 Understanding Performance

DataWeave transformations can process large datasets. Understanding performance implications helps you write efficient code.

Concept

Performance matters when:

- Processing large arrays (thousands of items)
- Deeply nested transformations
- High-throughput integration scenarios
- Real-time API responses

7.2 Memory Efficiency

7.2.1 Avoid Unnecessary Copies

Warning

Inefficient:

```
var temp1 = payload.users map $.name
var temp2 = temp1 filter ($ != null)
var temp3 = temp2 distinctBy $
---
temp3
```

Efficient (chaining):

```
---
payload.users
    map $.name
    filter ($ != null)
    distinctBy $
```

Chaining avoids creating intermediate variables and allows DataWeave to optimize.

7.2.2 Filter Early

```
// Inefficient - transforms all then filters
payload.users
    map expensiveTransformation($)
    filter $.active

// Efficient - filters first, transforms less
payload.users
    filter $.active
    map expensiveTransformation($)
```

7.2.3 Use Lookup Tables

Efficient Lookups

Inefficient (nested loop):

```
%dw 2.0
output application/json
---
payload.orders map (order) -> {
    orderId: order.id,
    customer: (payload.customers filter $.id == order.
       customerId)[0]
}
// O(n * m) complexity
```

Efficient (lookup table):

```
%dw 2.0
output application/json
var customerMap = payload.customers reduce ((item, acc={}) ->
    acc ++ {(item.id): item}
)
---
payload.orders map (order) -> {
    orderId: order.id,
    customer: customerMap[order.customerId]
}
// O(n + m) complexity
```

Teacher's Tip

When joining data:

1. Create a lookup table (object/map) from one dataset
2. Use simple key access instead of filtering
3. This changes O(n×m) to O(n+m) complexity

7.3 Streaming

DataWeave supports streaming for large datasets.

7.3.1 When to Use Streaming

Use streaming when:

- Processing files larger than available memory
- Transforming data that flows continuously
- You only need to touch each element once

7.3.2 Streaming-Compatible Operations

```
%dw 2.0
output application/json deferred=true
---
payload map {
    // Simple transformations work with streaming
    name: $.firstName ++ " " ++ $.lastName,
    age: $.age
}
```

7.3.3 Non-Streaming Operations

These require full data in memory:

- `groupBy`
- `orderBy`
- `sizeOf` (on arrays)
- Operations that need all data at once

7.4 Optimization Techniques

7.4.1 Minimize Nested Maps

Warning

Inefficient:

```
payload.orders map (order) -> {
    items: order.items map (item) -> {
        product: payload.products map (p) ->
            if (p.id == item.productId) p else null
    }
}
```

This creates nested loops with O(n×m×p) complexity.

Better approach:

```
%dw 2.0
output application/json
var productMap = payload.products reduce ((p, acc={}) ->
    acc ++ {(p.id): p}
)
---
payload.orders map (order) -> {
    items: order.items map (item) -> {
        product: productMap[item.productId]
    }
}
```

7.4.2 Avoid Redundant Calculations

```
// Inefficient - calculates sum multiple times
{
    total: sum(payload.items map $.price),
    average: sum(payload.items map $.price) / sizeOf(payload.
       items),
    withTax: sum(payload.items map $.price) * 1.08
}

// Efficient - calculate once
%dw 2.0
output application/json
var total = sum(payload.items map $.price)
var count = sizeOf(payload.items)
---
{
    total: total,
    average: total / count,
    withTax: total * 1.08
```

```
}
```

7.4.3 Use Built-in Functions

```
// Inefficient custom implementation
payload reduce ((item, acc=0) -> acc + item)

// Efficient built-in
sum(payload)

// Inefficient
payload reduce ((item, acc=[]) ->
    if (not (acc contains item)) acc + [item] else acc
)

// Efficient built-in
payload distinctBy $
```

7.5 Code Organization Best Practices

7.5.1 Use Meaningful Names

```
// Bad
var x = payload.users filter $.a
var y = x map $.n

// Good
var activeUsers = payload.users filter $.active
var userNames = activeUsers map $.name
```

7.5.2 Break Down Complex Transformations

Organize Complex Logic

Hard to read:

```
---
payload.orders map {
    id: $.id,
    total: sum($.items map ($.price * $.qty)),
    customer: (payload.customers filter $.id == $.customerId)
        [0].name,
    status: if ($.paid and $.shipped) "complete"
            else if ($.paid) "processing"
            else "pending"
}
```

Better organized:

```
%dw 2.0
output application/json

fun calculateOrderTotal(items) =
    sum(items map ($.price * $.qty))

fun getOrderStatus(order) =
    if (order.paid and order.shipped) "complete"
    else if (order.paid) "processing"
    else "pending"

var customerMap = payload.customers reduce ((c, acc={}) ->
    acc ++ {(c.id): c}
)
---
payload.orders map {
    id: $.id,
    total: calculateOrderTotal($.items),
    customer: customerMap[$.customerId].name,
    status: getOrderStatus($)
}
```

7.5.3 Document Complex Logic

```
%dw 2.0
output application/json

/**
 * Calculates discount based on customer tier and order amount.
 * VIP: 20% on orders > $1000, 15% otherwise
 * Regular: 10% on orders > $500, 5% otherwise
 */
fun calculateDiscount(tier: String, amount: Number): Number =
    tier match {
        case "VIP" -> if (amount > 1000) 0.20 else 0.15
        case "Regular" -> if (amount > 500) 0.10 else 0.05
        else -> 0
    }
---
payload
```

7.6 Error Prevention

7.6.1 Validate Inputs

```
%dw 2.0
```

```
output application/json

fun processOrder(order) =
    if (order.items == null or isEmpty(order.items))
        {error: "Order must have items"}
    else if (order.customerId == null)
        {error: "Customer ID required"}
    else
        {
            orderId: order.id,
            total: sum(order.items map $.price),
            customerId: order.customerId
        }
---
processOrder(payload)
```

7.6.2 Use Default Values

```
%dw 2.0
output application/json
---
{
    // Safe field access
    name: payload.name default "Unknown",
    age: payload.age default 0,
    tags: payload.tags default [],

    // Safe nested access
    city: payload.address.city default "Not specified",

    // Safe arithmetic
    discount: (payload.discount default 0) * 100
}
```

7.6.3 Type Safety

```
%dw 2.0
output application/json

fun processAge(age): Number =
    if (age is Number)
        age
    else if (age is String)
        age as Number
    else
        0
---
{
    age: processAge(payload.age)
```

```
}
```

7.7 Testing Strategies

7.7.1 Test with Edge Cases

Always test transformations with:

- Empty arrays
- Null values
- Missing fields
- Unexpected types
- Very large datasets

7.7.2 Unit Test Functions

```
%dw 2.0
output application/json

fun calculateTax(amount: Number, rate: Number): Number =
    amount * rate

// Test cases
---
{
    test1: calculateTax(100, 0.08) == 8,       // true
    test2: calculateTax(0, 0.08) == 0,         // true
    test3: calculateTax(100, 0) == 0,          // true
    test4: calculateTax(-100, 0.08) == -8      // true
}
```

7.8 Common Anti-Patterns

7.8.1 Don't Mutate in Functions

Warning

DataWeave is functional - don't try to mutate:

```
// This doesn't work (variables are immutable)
var total = 0
payload.items map {
    total = total + $.price  // ERROR
    ...
}
```

```

// Use reduce instead
var total = sum(payload.items map $.price)
```

7.8.2 Avoid Deep Nesting

```
// Hard to read
{
    a: {
        b: {
            c: {
                d: if (condition1) {
                    e: if (condition2) {
                        ...
                    }
                }
            }
        }
    }
}

// Better - use functions
fun processLevel1(data) = { ... }
fun processLevel2(data) = { ... }

---
{
    a: processLevel1(payload),
    b: processLevel2(payload)
}
```

7.8.3 Don't Overuse Match

```
// Overkill
value match {
    case 1 -> "one"
    case 2 -> "two"
    else -> "other"
}

// Simpler with if-else for few cases
if (value == 1) "one"
else if (value == 2) "two"
else "other"
```

7.9 Summary

Summary

Best practices for DataWeave:

- Filter before map to reduce data volume
- Use lookup tables instead of nested filters
- Chain operations instead of intermediate variables
- Leverage built-in functions
- Use meaningful variable and function names
- Document complex business logic
- Validate inputs and use defaults
- Test with edge cases
- Avoid deep nesting and mutations
- Consider streaming for large datasets

7.10 Practice Exercises

1. Refactor a nested filter-map to use lookup tables
2. Optimize a transformation with redundant calculations
3. Add input validation to a complex transformation
4. Break down a monolithic transformation into functions
5. Write test cases for a custom function

Chapter 8

Real World Use Cases

8.1 API Payload Transformation

8.1.1 REST API Response Normalization

Normalize Third-Party API

Scenario: External API returns verbose structure. Normalize for internal use.
External API Response:

```
{
    "response": {
        "status_code": 200,
        "data": {
            "user_profile": {
                "user_id": "U123",
                "first_name": "Alice",
                "last_name": "Smith",
                "email_address": "alice@example.com",
                "account_creation_date": "2020-01-15T10:30:00
                   Z",
                "subscription_tier": "premium"
            }
        }
    }
}
```

Internal Format Needed:

```
{
    "id": "U123",
    "fullName": "Alice Smith",
    "email": "alice@example.com",
    "memberSince": "2020-01-15",
    "isPremium": true
}
```

DataWeave Transformation:

```
%dw 2.0
output application/json
var user = payload.response.data.user_profile
---
{
    id: user.user_id,
    fullName: user.first_name ++ " " ++ user.last_name,
    email: user.email_address,
    memberSince: user.account_creation_date as Date as String
        {format: "yyyy-MM-dd"},
    isPremium: user.subscription_tier == "premium"
}
```

8.1.2 Request Payload Construction

Build SOAP Request from JSON

Input (from mobile app):

```
{
    "orderRequest": {
        "customerId": "C123",
        "items": [
            {"sku": "LAPTOP-001", "qty": 1},
            {"sku": "MOUSE-002", "qty": 2}
        ],
        "shippingAddress": {
            "street": "123 Main St",
            "city": "Boston",
            "state": "MA",
            "zip": "02101"
        }
    }
}
```

DataWeave (construct SOAP):

```
%dw 2.0
output application/xml
ns soap http://schemas.xmlsoap.org/soap/envelope/
ns ord http://example.com/orders
---
soap#Envelope: {
    soap#Header: null,
    soap#Body: {
        ord#CreateOrder: {
            ord#CustomerID: payload.orderRequest.customerId,
            ord#OrderItems: {
                (payload.orderRequest.items map {
```

```
                    ord#Item: {
                        ord#SKU: $.sku,
                        ord#Quantity: $.qty
                    }
                })
            },
            ord#ShippingAddress: {
                ord#Street: payload.orderRequest.
                   shippingAddress.street,
                ord#City: payload.orderRequest.
                   shippingAddress.city,
                ord#State: payload.orderRequest.
                   shippingAddress.state,
                ord#ZipCode: payload.orderRequest.
                   shippingAddress.zip
            }
        }
    }
}
```

8.2 Data Aggregation and Reporting

8.2.1 Sales Report Generation

Daily Sales Summary

Input (transaction logs):

```
[
    {"txnId": "T1", "product": "Laptop", "category": "
       Electronics", "amount": 1000, "date": "2024-03-15"},
    {"txnId": "T2", "product": "Mouse", "category": "
       Electronics", "amount": 25, "date": "2024-03-15"},
    {"txnId": "T3", "product": "Shirt", "category": "Clothing
       ", "amount": 50, "date": "2024-03-15"},
    {"txnId": "T4", "product": "Laptop", "category": "
       Electronics", "amount": 1000, "date": "2024-03-16"}
]
```

DataWeave (generate report):

```
%dw 2.0
output application/json
---
{
    reportDate: now() as String {format: "yyyy-MM-dd"},
    summary: {
        totalTransactions: sizeOf(payload),
        totalRevenue: sum(payload map $.amount),
```

```
        averageTransaction: avg(payload map $.amount)
    },
    byCategory: (payload groupBy $.category) pluck {
        category: $$,
        transactions: sizeOf($),
        revenue: sum($ map $.amount),
        topProduct: (
            $ groupBy $.product
            pluck {product: $$, count: sizeOf($)}
            orderBy -$.count
        )[0].product
    },
    byDate: (payload groupBy $.date) pluck {
        date: $$,
        transactions: sizeOf($),
        revenue: sum($ map $.amount)
    }
}
```

Output:

```
{
    "reportDate": "2024-03-17",
    "summary": {
        "totalTransactions": 4,
        "totalRevenue": 2075,
        "averageTransaction": 518.75
    },
    "byCategory": [
        {
            "category": "Electronics",
            "transactions": 3,
            "revenue": 2025,
            "topProduct": "Laptop"
        },
        {
            "category": "Clothing",
            "transactions": 1,
            "revenue": 50,
            "topProduct": "Shirt"
        }
    ],
    "byDate": [
        {"date": "2024-03-15", "transactions": 3, "revenue":
           1075},
        {"date": "2024-03-16", "transactions": 1, "revenue":
           1000}
    ]
}
```

8.3 Data Migration

8.3.1 Legacy to Modern System

Migrate Customer Data

Legacy System Format:

```
<Customers>
    <Customer CustomerID="C001" Status="A">
        <FullName>ALICE SMITH</FullName>
        <ContactInfo>
            <Phone Type="Home">555-0101</Phone>
            <Phone Type="Mobile">555-0102</Phone>
            <Email>ALICE@OLD.COM</Email>
        </ContactInfo>
        <Address>
            <Line1>123 MAIN ST</Line1>
            <City>BOSTON</City>
            <State>MA</State>
        </Address>
    </Customer>
</Customers>
```

Modern System Format:

```
{
    "customerId": "C001",
    "name": "Alice Smith",
    "status": "active",
    "contacts": {
        "phones": [
            {"type": "home", "number": "555-0101"},
            {"type": "mobile", "number": "555-0102"}
        ],
        "email": "alice@old.com"
    },
    "address": {
        "street": "123 Main St",
        "city": "Boston",
        "state": "MA"
    }
}
```

DataWeave:

```
%dw 2.0
output application/json
---
payload.Customers.*Customer map {
    customerId: $.@CustomerID,
    name: capitalize(lower($.FullName)),
    status: if ($.@Status == "A") "active" else "inactive",
```

```
    contacts: {
        phones: $.ContactInfo.*Phone map {
            type: lower($.@Type),
            number: $
        },
        email: lower($.ContactInfo.Email)
    },
    address: {
        street: capitalize(lower($.Address.Line1)),
        city: capitalize(lower($.Address.City)),
        state: $.Address.State
    }
}
```

8.4 Event Processing

8.4.1 Event Stream Transformation

Process IoT Sensor Events

Raw Sensor Data:

```
[
    {"sensorId": "S001", "type": "temperature", "value": 72.5, "timestamp": "2024-03-15T10:00:00Z", "unit": "F"},
    {"sensorId": "S001", "type": "temperature", "value": 73.2, "timestamp": "2024-03-15T10:05:00Z", "unit": "F"},
    {"sensorId": "S002", "type": "humidity", "value": 45, "timestamp": "2024-03-15T10:00:00Z", "unit": "%"},
    {"sensorId": "S001", "type": "temperature", "value": 75.8, "timestamp": "2024-03-15T10:10:00Z", "unit": "F"}
]
```

Process and Alert:

```
%dw 2.0
output application/json

fun fahrenheitToCelsius(f: Number): Number =
    (f - 32) * 5 / 9

fun checkAlert(sensor, value: Number): Object =
    if (sensor.type == "temperature" and value > 75)
        {level: "warning", message: "High temperature detected"}
    else if (sensor.type == "humidity" and value < 30)
```

```
        {level: "warning", message: "Low humidity detected"}
    else
        {level: "normal", message: "Within normal range"}
---
{
    processedAt: now(),
    events: payload map {
        sensorId: $.sensorId,
        type: $.type,
        value: if ($.unit == "F")
                    fahrenheitToCelsius($.value)
                else
                    $.value,
        unit: if ($.unit == "F") "C" else $.unit,
        timestamp: $.timestamp as DateTime,
        alert: checkAlert($, $.value)
    },
    alerts: (payload
        map {
            sensorId: $.sensorId,
            type: $.type,
            value: $.value,
            alert: checkAlert($, $.value)
        }
        filter $.alert.level != "normal"
    ),
    summary: {
        totalEvents: sizeOf(payload),
        alertCount: sizeOf(payload filter checkAlert($, $.
           value).level != "normal"),
        avgTemperature: avg(
            (payload filter $.type == "temperature")
            map fahrenheitToCelsius($.value)
        )
    }
}
```

8.5 Multi-Source Data Integration

8.5.1 Merge Data from Multiple Systems

Customer 360 View

CRM Data:

```
{
    "customers": [
        {"id": "C001", "name": "Alice Smith", "segment": "
```

```
            premium"},
        {"id": "C002", "name": "Bob Jones", "segment": "
            standard"}
    ]
}
```

Order System Data:

```
{
    "orders": [
        {"customerId": "C001", "orderId": "O100", "total":
            1500, "date": "2024-03-01"},
        {"customerId": "C001", "orderId": "O101", "total":
            200, "date": "2024-03-10"},
        {"customerId": "C002", "orderId": "O102", "total":
            500, "date": "2024-03-05"}
    ]
}
```

Support System Data:

```
{
    "tickets": [
        {"customerId": "C001", "ticketId": "T500", "status":
            "open", "priority": "high"},
        {"customerId": "C002", "ticketId": "T501", "status":
            "closed", "priority": "low"}
    ]
}
```

DataWeave (merge all):

```
%dw 2.0
output application/json
var crmData = payload[0]
var orderData = payload[1]
var supportData = payload[2]

var ordersByCustomer = orderData.orders groupBy $.customerId
var ticketsByCustomer = supportData.tickets groupBy $.
    customerId
---
{
    customer360: crmData.customers map {
        customerId: $.id,
        name: $.name,
        segment: $.segment,
        orderHistory: {
            totalOrders: sizeOf(ordersByCustomer[$.id]
                default []),
            totalSpent: sum((ordersByCustomer[$.id] default
                []) map $.total),
```

```
            lastOrderDate: max((ordersByCustomer[$.id]
                default []) map $.date)
        },
        support: {
            totalTickets: sizeOf(ticketsByCustomer[$.id]
                default []),
            openTickets: sizeOf(
                (ticketsByCustomer[$.id] default [])
                filter $.status == "open"
            ),
            highPriorityTickets: sizeOf(
                (ticketsByCustomer[$.id] default [])
                filter $.priority == "high"
            )
        },
        healthScore: do {
            var orders = sizeOf(ordersByCustomer[$.id]
                default [])
            var openTickets = sizeOf(
                (ticketsByCustomer[$.id] default [])
                filter $.status == "open"
            )
            ---
            if (orders > 5 and openTickets == 0) "excellent"
            else if (orders > 2 and openTickets < 2) "good"
            else if (openTickets > 3) "at-risk"
            else "fair"
        }
    }
}
```

8.6 Summary

Summary

Real-world DataWeave patterns:

- Normalize external API responses to internal formats
- Construct complex SOAP/XML requests from JSON
- Generate aggregated reports from transaction data
- Migrate data between legacy and modern systems
- Process and filter event streams with business logic
- Merge data from multiple sources into unified views

- Apply transformations, validations, and calculations in pipelines

Common techniques used:

- Lookup tables for efficient joins
- GroupBy for aggregation
- Custom functions for business logic
- Conditional transformations
- Type coercion for format conversions

8.7 Practice Exercises

1. Create a transformation that normalizes responses from three different weather APIs
2. Build a sales report showing top products, revenue by region, and month-over-month growth
3. Migrate XML employee records to JSON with calculated fields (tenure, next review date)
4. Process a stream of financial transactions and flag suspicious patterns
5. Merge customer data from CRM, billing, and support systems into single profile

Chapter 9

Revision & Interview Preparation

9.1 Core Concepts Review

9.1.1 DataWeave Fundamentals

- **What is DataWeave?** MuleSoft's transformation language for converting data between formats
- **Key characteristics:** Declarative, functional, type-safe
- **Script structure:** Header (directives, variables, functions) + Separator (`---`) + Body
- **Version:** Always start with `%dw 2.0`
- **Output directive:** Specifies format (JSON, XML, CSV, etc.)

9.1.2 Quick Reference Table

Function	Purpose	Example
`map`	Transform each element	`[1,2,3] map $ * 2`
`filter`	Keep matching elements	`[1,2,3] filter $ > 1`
`reduce`	Aggregate to single value	`[1,2,3] reduce ($$ + $)`
`pluck`	Object to array	`{a:1} pluck $$`
`groupBy`	Group by criterion	`users groupBy $.dept`
`distinctBy`	Remove duplicates	`items distinctBy $.id`
`flatMap`	Map and flatten	`orders flatMap $.items`
`orderBy`	Sort array	`users orderBy $.age`

Table 9.1: Essential DataWeave Functions

9.2 Common Interview Questions

9.2.1 Conceptual Questions

Q1: What is the difference between `map` and `flatMap`?

Answer: `map` transforms each element and maintains array structure. `flatMap` transforms and flattens nested arrays.

```
// map keeps nested structure
[[1,2], [3,4]] map ($ map $ * 2)
// Result: [[2,4], [6,8]]

// flatMap flattens after mapping
[[1,2], [3,4]] flatMap ($ map $ * 2)
// Result: [2, 4, 6, 8]
```

Q2: What do $, $$, and $$$ represent?
Answer:

- `$` — Current item/value
- `$$` — Current key/index or accumulator (in reduce)
- `$$$` — Total count (in iteration)

Q3: How does DataWeave handle null values?
Answer: Use `default` operator to provide fallback values:

```
name: payload.name default "Unknown"
```

Q4: What's the difference between `groupBy` and `distinctBy`?
Answer:

- `groupBy` — Returns object with groups: `{key: [items]}`
- `distinctBy` — Returns array with unique items based on criterion

9.2.2 Coding Questions

Q5: Transform array of objects to extract specific fields
Task:

```
Input: [
    {"id": 1, "name": "Alice", "age": 30, "dept": "IT"},
    {"id": 2, "name": "Bob", "age": 25, "dept": "HR"}
]

Output: [
    {"id": 1, "name": "Alice"},
    {"id": 2, "name": "Bob"}
]
```

Solution:

```
%dw 2.0
output application/json
---
payload map {
    id: $.id,
    name: $.name
}
```

Q6: Calculate total from array of prices

```
%dw 2.0
output application/json
---
{
    total: sum(payload map $.price)
}
```

Q7: Group items and calculate totals per group
Input:

```
[
    {"product": "Laptop", "category": "Electronics", "price":
       1000},
    {"product": "Mouse", "category": "Electronics", "price": 25},
    {"product": "Shirt", "category": "Clothing", "price": 50}
]
```

Solution:

```
%dw 2.0
output application/json
---
(payload groupBy $.category) pluck {
    category: $$,
    total: sum($ map $.price),
    count: sizeOf($)
}
```

Q8: Flatten nested arrays

```
%dw 2.0
output application/json
---
flatten(payload.orders map $.items)
```

Q9: Filter and transform
Task: Get names of active users over 25

```
%dw 2.0
output application/json
---
payload.users
    filter ($.active and $.age > 25)
    map $.name
```

Q10: Join two datasets
Input:

```
{
    "orders": [{"id": 1, "customerId": "C1", "total": 100}],
    "customers": [{"id": "C1", "name": "Alice"}]
}
```

Solution:

```
%dw 2.0
output application/json
var customerMap = payload.customers reduce ((c, acc={}) ->
    acc ++ {(c.id): c}
)
---
payload.orders map {
    orderId: $.id,
    customerName: customerMap[$.customerId].name,
    total: $.total
}
```

9.3 Format Conversion Checklist

9.3.1 JSON to XML

```
%dw 2.0
output application/xml
---
{
    Root: {
        Element: value,

        // Attribute
        ElementWithAttr @(id: "123"): value,

        // Repeating elements from array
        (payload.items map {
            Item: $.name
        })
    }
}
```

9.3.2 XML to JSON

```
%dw 2.0
output application/json
---
{
    // Single element
    name: payload.Root.Element,

    // Attribute
    id: payload.Root.ElementWithAttr.@id,

    // Repeating elements
    items: payload.Root.*Item map $
```

```
}
```

9.3.3 CSV Handling

```
// CSV to JSON
%dw 2.0
output application/json
---
payload map {
    name: $.name,
    age: $.age as Number
}

// JSON to CSV
%dw 2.0
output application/csv header=true
---
payload
```

9.4 Performance Tips

1. **Filter early** — Reduce dataset before expensive operations
2. **Use lookup tables** — Convert nested filters to O(1) lookups
3. **Chain operations** — Avoid intermediate variables
4. **Leverage built-ins** — Use `sum()`, `max()`, etc. instead of `reduce`
5. **Calculate once** — Store expensive calculations in variables

9.5 Common Mistakes to Avoid

Warning

1. Forgetting `---` separator
2. Using `+` instead of `++` for strings
3. Not handling null values
4. Inefficient nested loops
5. Forgetting type coercion when converting from XML
6. Mutating variables (not possible in DataWeave)
7. Not using parentheses for dynamic keys: `($.key): value`

9.6 Interview Scenarios

9.6.1 Scenario 1: API Integration

Question: "You need to transform a third-party API response to match your internal schema. How would you approach this?"

Answer:

1. Analyze both formats
2. Identify field mappings
3. Handle missing/null values with `default`
4. Apply type conversions as needed
5. Validate critical fields
6. Use functions for complex transformations

9.6.2 Scenario 2: Performance Problem

Question: "Your transformation is slow when processing 10,000 records. What would you check?"

Answer:

1. Look for nested `filter` operations
2. Replace with lookup tables
3. Filter before mapping
4. Avoid redundant calculations
5. Consider streaming with `deferred=true`
6. Check for unnecessary type conversions

9.6.3 Scenario 3: Data Quality

Question: "Input data sometimes has missing fields. How do you handle this?"

Answer:

```
%dw 2.0
output application/json
---
{
    // Provide defaults
    name: payload.name default "Unknown",

    // Conditional fields
    (email: payload.email) if payload.email != null,

    // Validation
```

```
    age: if (payload.age is Number and payload.age >= 0)
            payload.age
         else
            0
}
```

9.7 Quick Practice Problems

9.7.1 Problem 1

Convert array of numbers to array of their squares:

```
[1, 2, 3, 4, 5] map $ * $
```

9.7.2 Problem 2

Get unique department names from users:

```
payload.users distinctBy $.department map $.department
```

9.7.3 Problem 3

Calculate average age:

```
avg(payload.users map $.age)
```

9.7.4 Problem 4

Build object from key-value array:

```
payload reduce ((item, acc={}) ->
    acc ++ {(item.key): item.value}
)
```

9.7.5 Problem 5

Filter adults and get their names:

```
payload filter $.age >= 18 map $.name
```

9.8 Final Checklist

Before an interview, ensure you can:

- ☐ Explain DataWeave's functional paradigm
- ☐ Write basic transformations without reference

- ☐ Use map, filter, reduce confidently
- ☐ Handle null values properly
- ☐ Convert between JSON, XML, CSV
- ☐ Group and aggregate data
- ☐ Join datasets efficiently
- ☐ Write custom functions
- ☐ Optimize transformations
- ☐ Debug transformation errors

9.9 Summary

Summary

Key takeaways for interviews:

- Understand core functions deeply: map, filter, reduce, pluck, groupBy
- Practice format conversions (JSON XML CSV)
- Know how to handle null and missing data
- Optimize with lookup tables and early filtering
- Write clean, readable code with good naming
- Always validate inputs and provide defaults
- Use built-in functions when available
- Think functionally — no mutations!

You are now ready for DataWeave mastery!

Practice these patterns, understand the concepts, and you'll excel in any DataWeave interview or project.

Appendix: Quick Reference

Essential Functions

Function	Description	Example
`map`	Transform each element	`arr map $ * 2`
`filter`	Keep matching elements	`arr filter $ > 5`
`reduce`	Aggregate values	`arr reduce $$ + $`
`pluck`	Object → Array	`obj pluck $$`
`groupBy`	Group by key	`arr groupBy $.type`
`distinctBy`	Unique values	`arr distinctBy $.id`
`flatten`	Flatten arrays	`flatten(nestedArr)`
`flatMap`	Map + flatten	`arr flatMap $.items`
`orderBy`	Sort	`arr orderBy $.age`
`sum`	Sum numbers	`sum(numbers)`
`avg`	Average	`avg(numbers)`
`max`	Maximum	`max(numbers)`
`min`	Minimum	`min(numbers)`
`sizeOf`	Count elements	`sizeOf(arr)`
`isEmpty`	Check if empty	`isEmpty(arr)`

String Functions

Function	Example
`upper`	`upper("hello")` → ”HELLO”
`lower`	`lower("HELLO")` → ”hello”
`capitalize`	`capitalize("hello")` → ”Hello”
`trim`	`trim(" hello ")` → ”hello”
`++`	`"hello" ++ " world"` → ”hello world”
`contains`	`"hello" contains "ell"` → true
`startsWith`	`"hello" startsWith "he"` → true
`endsWith`	`"hello" endsWith "lo"` → true
`replace`	`"hello" replace "l" with "r"` → ”herro”
`split`	`"a,b,c" splitBy ","` → [”a”,”b”,”c”]

Operator	Purpose	Example
`.`	Field access	`payload.name`
`.*`	Multi-value selector	`payload.*user`
`..`	Descendant selector	`payload..email`
`[]`	Index/filter	`arr[0]`
`++`	Concatenation	`arr1 ++ arr2`
`--`	Removal	`obj -- "key"`
`default`	Fallback value	`val default 0`
`as`	Type coercion	`"42" as Number`
`is`	Type check	`val is String`
`match`	Pattern match	`val match {...}`
`update`	Modify nested	`obj update {...}`

Operators

Common Patterns

Extract field from array of objects:

```
users map $.name
```

Filter then transform:

```
users filter $.active map $.name
```

Sum field values:

```
sum(orders map $.total)
```

Group and aggregate:

```
(items groupBy $.category) pluck {
    category: $$,
    total: sum($ map $.price)
}
```

Flatten nested arrays:

```
orders flatMap $.items
```

Join datasets with lookup:

```
var lookup = dataset2 reduce ((item, acc={}) ->
    acc ++ {(item.id): item}
)
---
dataset1 map {
    field1: $.field1,
    joined: lookup[$.foreignKey]
}
```

End of DataWeave Complete Guide

Happy transforming!