Contents

JasperReports Template Designer Handbook

This handbook provides comprehensive guidance for template designers working with the Document Generation Service, covering both parameters and JSON data source functionality in Jaspersoft Studio 7.0.x.

Overview

The Document Generation Service supports two primary ways to pass data to JasperReports templates:

Parameters

- Simple key-value pairs passed to the template
- Ideal for configuration values, titles, dates, and single values
- Available throughout the entire template
- Accessed using `$P{parameterName}` expressions

JSON Data Source

- Complex structured data including arrays and nested objects
- Perfect for detail records, line items, and tabular data
- Automatically creates JasperReports fields from JSON structure
- Accessed using `$F{fieldName}` expressions

Hybrid Approach (Recommended)

- Combine both parameters and data source in the same template
- Use parameters for configuration and titles
- Use data source for detail records and complex structures

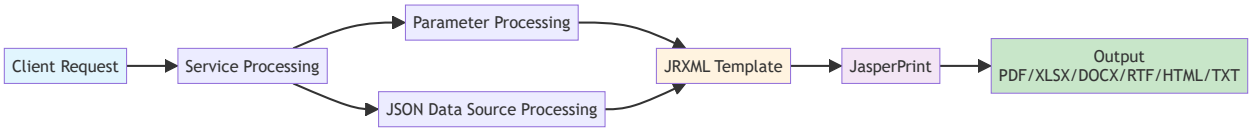
Service Integration Fundamentals

Request Structure

The service accepts requests with the following structure:

```
{
  "parameters": [
    {"name": "parameterName", "value": "parameterValue"}
  ],
  "data": {
    "fieldName": "fieldValue",
    "arrayField": [
      {"itemField": "itemValue"}
    ]
  },
  "filename": "output_filename"
}
```

Data Flow Architecture



Template Development Workflow

1. **Design**: Create template structure in Jaspersoft Studio
2. **Parameters**: Define parameters for configuration values
3. **Data Source**: Design field usage for structured data
4. **Test**: Use preview with sample data
5. **Package**: Use Muban CLI to package template with dependencies
6. **Deploy**: Upload template to service (via CLI or API)
7. **Integrate**: Test with actual service calls using CLI or HTTP client

> **Tip:** Set the report description property `com.jaspersoft.studio.report.description` in JasperSoft Studio (Report Properties panel). The service auto-extracts this value as the template description on upload if no explicit description is provided.

Parameter Usage

Defining Parameters in Jaspersoft Studio

Using the Parameters Tab

1. Open your template in Jaspersoft Studio
2. Navigate to the **Parameters** tab in the Outline view
3. Right-click and select **Create Parameter**
4. Configure parameter properties:

```
<!-- Example parameter definition -->
<parameter name="reportTitle" class="java.lang.String">
    <defaultValueExpression><![CDATA["Default Report Title"]]></defaultValueExpression>
</parameter>
```

Parameter Properties

Property	Description	Example
Name	Parameter identifier	`reportTitle`, `generatedDate`, `userRole`
Class	Java data type	`java.lang.String`, `java.lang.Integer`, `java.util.Date`
Default Value	Used when parameter is not provided	`"Draft Report"`, `new Date()`
Description	Documentation for parameter usage	“The main title displayed on the document header”

Supported Parameter Types

The service automatically converts string values to appropriate types:

```
<!-- String Parameter -->
<parameter name="title" class="java.lang.String">
    <defaultValueExpression><![CDATA["Default Title"]]></defaultValueExpression>
</parameter>

<!-- Integer Parameter -->
<parameter name="maxItems" class="java.lang.Integer">
    <defaultValueExpression><![CDATA[100]]></defaultValueExpression>
</parameter>

<!-- Date Parameter -->
<parameter name="reportDate" class="java.util.Date">
    <defaultValueExpression><![CDATA[new java.util.Date()]]></defaultValueExpression>
</parameter>

<!-- Boolean Parameter -->
<parameter name="showDetails" class="java.lang.Boolean">
    <defaultValueExpression><![CDATA[Boolean.TRUE]]></defaultValueExpression>
</parameter>

<!-- Decimal Parameter -->
<parameter name="discountRate" class="java.math.BigDecimal">
    <defaultValueExpression><![CDATA[new java.math.BigDecimal("0.10")]]></defaultValueExpression>
</parameter>
```

Using Parameters in Templates

Text Field Expressions

```
<!-- Simple parameter display -->
<textField>
    <reportElement x="0" y="0" width="200" height="20"/>
    <textFieldExpression><![CDATA[$P{reportTitle}]]></textFieldExpression>
</textField>

<!-- Parameter with formatting -->
<textField pattern="dd/MM/yyyy">
    <reportElement x="0" y="20" width="100" height="20"/>
    <textFieldExpression><![CDATA[$P{reportDate}]]></textFieldExpression>
</textField>

<!-- Conditional parameter usage -->
<textField>
    <reportElement x="0" y="40" width="200" height="20"/>
    <textFieldExpression><![CDATA[
        $P{showDetails} ? "Detailed Report" : "Summary Report"
    ]]></textFieldExpression>
</textField>
```

Mathematical Operations

```
<!-- Calculations with parameters -->
<textField pattern="#,##0.00">
    <reportElement x="0" y="60" width="100" height="20"/>
    <textFieldExpression><![CDATA[
        $P{baseAmount} * (1 + $P{taxRate})
    ]]></textFieldExpression>
</textField>
```

String Manipulation

```
<!-- String concatenation -->
<textField>
    <reportElement x="0" y="80" width="200" height="20"/>
    <textFieldExpression><![CDATA[
        "Report generated for: " + $P{customerName}
    ]]></textFieldExpression>
</textField>

<!-- String formatting -->
<textField>
    <reportElement x="0" y="100" width="200" height="20"/>
    <textFieldExpression><![CDATA[
        String.format("Discount: %.1f%%", $P{discountRate} * 100)
    ]]></textFieldExpression>
</textField>
```

Conditional Logic

```
<!-- Complex conditional expressions -->
<textField>
```

```
    <reportElement x="0" y="120" width="200" height="20"/>
    <textFieldExpression><![CDATA[
        $P{priority}.equals("HIGH") ? "[!] " + $P{title} : $P{title}
    ]]></textFieldExpression>
</textField>
```

Parameter Service Integration

When the service processes parameters, it:

1. **Extracts** parameter definitions from JRXML templates during upload
2. **Stores** parameter metadata in the database
3. **Validates** incoming parameter values against stored definitions
4. **Converts** string values to appropriate Java types
5. **Passes** converted values to JasperReports engine

Example Service Request with Parameters

```
{
  "parameters": [
    {"name": "reportTitle", "value": "Monthly Sales Summary"},
    {"name": "reportDate", "value": "2025-10-07"},
    {"name": "showDetails", "value": "true"},
    {"name": "discountRate", "value": "0.15"},
    {"name": "maxItems", "value": "50"}
  ]
}
```

JSON Data Source Usage

Understanding JSON Data Sources

The JSON data source feature converts structured data into JasperReports-compatible field collections, enabling complex document generation with arrays and nested objects.

Data Source Architecture

Data Structure Examples

Simple Object Structure

```
{
  "data": {
    "customer": {
      "name": "ACME Corporation",
      "address": "123 Business St",
      "phone": "+1-555-0123"
    },
    "invoice": {
      "number": "INV-2025-001",
      "date": "2025-10-07",
      "dueDate": "2025-11-07"
    }
  }
}
```

Template Usage:

```
<textField>
    <reportElement x="0" y="0" width="200" height="20"/>
    <textFieldExpression><![CDATA[$F{customer}]]></textFieldExpression>
</textField>
```

Array Structure (Most Common)

```
{
  "data": {
    "lineItems": [
      {
        "description": "Professional Services",
        "quantity": 40,
        "rate": 150.00,
        "amount": 6000.00
      },
      {
        "description": "Software License",
        "quantity": 1,
        "rate": 2500.00,
        "amount": 2500.00
      }
```

```
    ]
  }
}
```

Template Usage:

```
<!-- Detail band will iterate over each lineItems array element -->
<detail>
    <band height="20">
        <textField>
            <reportElement x="0" y="0" width="200" height="20"/>
            <textFieldExpression><![CDATA[$F{description}]]></textFieldExpression>
        </textField>
        <textField>
            <reportElement x="200" y="0" width="50" height="20"/>
            <textFieldExpression><![CDATA[$F{quantity}]]></textFieldExpression>
        </textField>
        <textField pattern="#,##0.00">
            <reportElement x="250" y="0" width="80" height="20"/>
            <textFieldExpression><![CDATA[$F{rate}]]></textFieldExpression>
        </textField>
        <textField pattern="#,##0.00">
            <reportElement x="330" y="0" width="80" height="20"/>
            <textFieldExpression><![CDATA[$F{amount}]]></textFieldExpression>
        </textField>
    </band>
</detail>
```

Mixed Array and Object Structure

```
{
  "data": {
    "items": [
      {"product": "Widget A", "sales": 1000, "revenue": 25000},
      {"product": "Widget B", "sales": 750, "revenue": 18750}
    ],
    "summary": {
      "totalSales": 1750,
      "totalRevenue": 43750,
      "avgPrice": 25.00
    },
    "metadata": {
      "generated": "2025-10-07T15:30:00Z",
      "region": "North America"
    }
  }
}
```

Field Access Patterns

Direct Field Access

```
<!-- Access simple fields directly -->
<textField>
    <reportElement x="0" y="0" width="100" height="20"/>
    <textFieldExpression><![CDATA[$F{product}]]></textFieldExpression>
```

```
</textField>

<textField pattern="#,##0">
    <reportElement x="100" y="0" width="80" height="20"/>
    <textFieldExpression><![CDATA[$F{sales}]]></textFieldExpression>
</textField>
```

Complex Object Field Access

```
<!-- Access nested object fields -->
<textField>
    <reportElement x="0" y="0" width="100" height="20"/>
    <textFieldExpression><![CDATA[$F{summary}]]></textFieldExpression>
</textField>

<!-- Note: The service flattens nested objects, so you access them directly -->
<textField pattern="#,##0">
    <reportElement x="100" y="0" width="80" height="20"/>
    <textFieldExpression><![CDATA[$F{totalSales}]]></textFieldExpression>
</textField>
```

Conditional Field Usage

```
<!-- Conditional field display -->
<textField>
    <reportElement x="0" y="0" width="200" height="20"/>
    <textFieldExpression><![CDATA[
        $F{revenue} > 20000 ? "[HOT] " + $F{product} : $F{product}
    ]]></textFieldExpression>
</textField>
```

Array Handling Behavior

The service implements a **UNION strategy** for handling multiple arrays. This approach vertically stacks all arrays into a single dataset, which is ideal for reports with multiple independent tables of different sizes.

UNION Strategy Overview

When multiple arrays are present, the service:

- Vertically stacks all arrays into a single dataset (like SQL UNION)
- Each row contains data from ONE array item with metadata indicating its source
- No null padding - each row is complete with its own data
- Shared (non-array) data is available in every row

```
{
  "data": {
    "insured_properties": [
      {"record_category": "property", "id": "PROP-001", "type": "building"},
      {"record_category": "property", "id": "PROP-002", "type": "garage"}
    ],
    "coverage_variants": [
      {"record_category": "variant", "code": "BASIC", "rate": 0.08},
      {"record_category": "variant", "code": "PREMIUM", "rate": 0.18}
    ],
    "pricing_summary": {
```

```
      "total_value": 9880000
    }
  }
}
```

Result: 4 rows total (2 + 2), structured as follows:

Row	_sourceArray	_sourceIndex	id / code	pricing_summary_total_value
0	insured_properties	0	PROP-001	9880000
1	insured_properties	1	PROP-002	9880000
2	coverage_variants	0	BASIC	9880000
3	coverage_variants	1	PREMIUM	9880000

Filtering by Source Array

Use `printWhenExpression` to show specific data in different template sections:

```
<!-- Show only properties -->
<detail>
    <band height="20">
        <printWhenExpression>
            <![CDATA[$F{_sourceArray}.equals("insured_properties")]]>
        </printWhenExpression>
        <textField>
            <reportElement x="0" y="0" width="100" height="20"/>
            <textFieldExpression><![CDATA[$F{id}]]></textFieldExpression>
        </textField>
    </band>
</detail>
```

Filtering by Record Category

If your data includes a `record_category` field, you can filter by it:

```
<!-- Show only variants -->
<detail>
    <band height="20">
        <printWhenExpression>
            <![CDATA[$F{record_category}.equals("variant")]]>
        </printWhenExpression>
        <textField>
            <reportElement x="0" y="0" width="100" height="20"/>
            <textFieldExpression><![CDATA[$F{code}]]></textFieldExpression>
        </textField>
    </band>
</detail>
```

UNION Metadata Fields

Each row includes metadata for filtering and navigation:

Field	Type	Description
_sourceArray	String	Name of the source array (e.g., "insured_properties")

Field	Type	Description
`_sourceIndex`	Integer	Index within the source array (0-based)
`_sourceArraySize`	Integer	Total size of the source array
`_globalIndex`	Integer	Global row index across all arrays
`_totalRows`	Integer	Total rows across all arrays
`_arrayFields`	List	List of all array field names in the data
`_originalData`	Map	Complete original data structure
`_sharedData`	Map	Non-array data (available in all rows)
`_recordType`	Object	Uses `record_category` if present, otherwise source array name

Non-Array Data with Arrays

Non-array values (shared data) are available in every row:

```
{
  "data": {
    "reportTitle": "Sales Summary",
    "items": [
      {"product": "A", "sales": 100},
      {"product": "B", "sales": 200}
    ]
  }
}
```

Result: 2 rows, each with `reportTitle` = “Sales Summary”.

Multi-Table Template Example (Insurance Offer)

This example demonstrates how to create a template with multiple independent tables using the UNION strategy:

Request Data:

```
{
  "parameters": [
    {"name": "offer_number", "value": "PROP/2026/00456"},
    {"name": "offer_date", "value": "2026-01-23"}
  ],
  "data": {
    "insured_properties": [
      {"record_category": "property", "id": "PROP-001", "type": "residential building", "value_usd":
      ↪  850000},
      {"record_category": "property", "id": "PROP-002", "type": "garage", "value_usd": 120000}
    ],
    "coverage_variants": [
      {"record_category": "variant", "code": "BASIC", "name": "Basic Coverage", "rate": 0.08},
      {"record_category": "variant", "code": "PREMIUM", "name": "Premium Coverage", "rate": 0.18}
    ],
    "additional_services": [
      {"record_category": "service", "code": "ASSIST24", "name": "24h Technical Assistance", "price":
      ↪  1200}
    ],
    "pricing_summary": {
      "total_value": 970000,
      "recommended_variant": "BASIC"
```

```
    }
  }
}
```

Generated Rows (UNION Result):

_globalIndex	_sourceArray	record_category	Key Fields
0	insured_properties	property	id=PROP-001, type=residential building
1	insured_properties	property	id=PROP-002, type=garage
2	coverage_variants	variant	code=BASIC, rate=0.08
3	coverage_variants	variant	code=PREMIUM, rate=0.18
4	additional_services	service	code=ASSIST24, price=1200

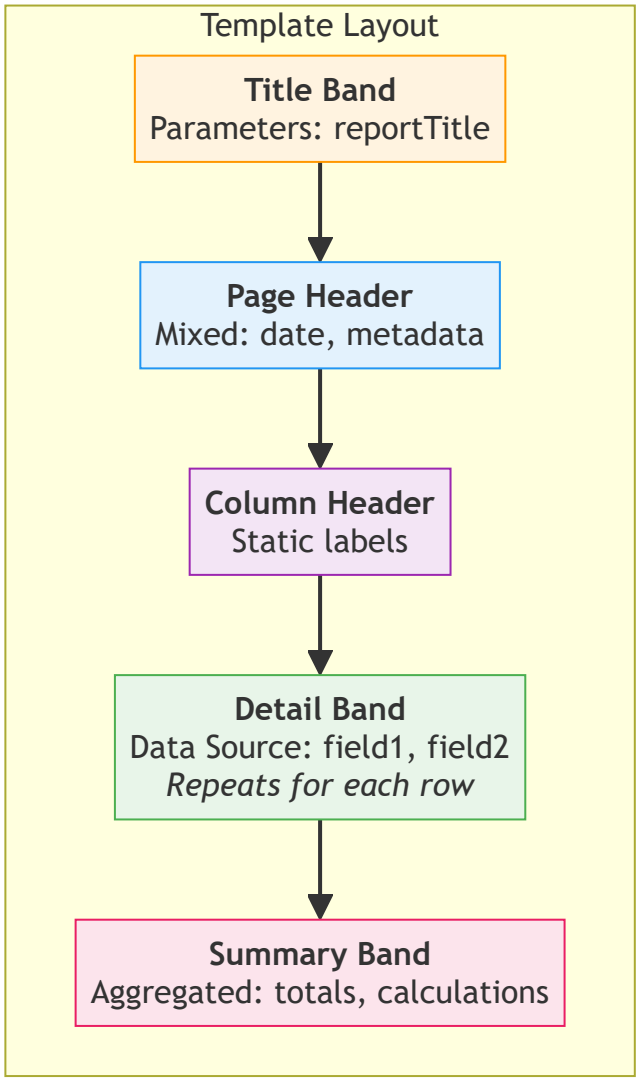
Template Design with Multiple Filtered Sections:

```
<!-- Properties Table -->
<detail>
    <band height="20">

↪   <printWhenExpression><![CDATA[$F{_sourceArray}.equals("insured_properties")]]></printWhenExpression>
        <textField>
            <reportElement x="0" y="0" width="80" height="20"/>
            <textFieldExpression><![CDATA[$F{id}]]></textFieldExpression>
        </textField>
        <textField>
            <reportElement x="80" y="0" width="150" height="20"/>
            <textFieldExpression><![CDATA[$F{type}]]></textFieldExpression>
        </textField>
        <textField pattern="$#,##0.00">
            <reportElement x="230" y="0" width="100" height="20"/>
            <textFieldExpression><![CDATA[$F{value_usd}]]></textFieldExpression>
        </textField>
    </band>
</detail>

<!-- Coverage Variants Table -->
<detail>
    <band height="20">

↪   <printWhenExpression><![CDATA[$F{_sourceArray}.equals("coverage_variants")]]></printWhenExpression>
        <textField>
            <reportElement x="0" y="0" width="80" height="20"/>
            <textFieldExpression><![CDATA[$F{code}]]></textFieldExpression>
        </textField>
        <textField>
            <reportElement x="80" y="0" width="150" height="20"/>
            <textFieldExpression><![CDATA[$F{name}]]></textFieldExpression>
        </textField>
        <textField pattern="0.00%">
            <reportElement x="230" y="0" width="100" height="20"/>
            <textFieldExpression><![CDATA[$F{rate}]]></textFieldExpression>
        </textField>
    </band>
</detail>
```

```
<!-- Additional Services Table -->
<detail>
    <band height="20">
        <printWhenExpression><![CDATA[$F{_sourceArray}.equals("additional_services")]]></
↪   printWhenExpression>
        <textField>
            <reportElement x="0" y="0" width="80" height="20"/>
            <textFieldExpression><![CDATA[$F{code}]]></textFieldExpression>
        </textField>
        <textField>
            <reportElement x="80" y="0" width="150" height="20"/>
            <textFieldExpression><![CDATA[$F{name}]]></textFieldExpression>
        </textField>
        <textField pattern="$#,##0.00">
            <reportElement x="230" y="0" width="100" height="20"/>
            <textFieldExpression><![CDATA[$F{price}]]></textFieldExpression>
        </textField>
    </band>
</detail>

<!-- Summary (uses shared data - available in all rows, show once) -->
<summary>
    <band height="40">
        <textField pattern="$#,##0.00">
            <reportElement x="0" y="0" width="200" height="20"/>
            <textFieldExpression><![CDATA["Total Value: " +
↪   $F{pricing_summary_total_value}]]></textFieldExpression>
        </textField>
        <textField>
            <reportElement x="0" y="20" width="200" height="20"/>
            <textFieldExpression><![CDATA["Recommended Variant: " +
↪   $F{pricing_summary_recommended_variant}]]></textFieldExpression>
        </textField>
    </band>
</summary>
```

Template Design for Data Sources

Template Structure Planning



Band	Data Source	Expression Examples
Title Band	Parameters	`$P{reportTitle}`
Page Header	Mixed	`$P{date}`, `$F{metadata}`
Column Header	Static	Static text labels
Detail Band	Data Source (iterates)	`$F{field1}`, `$F{field2}`
Summary Band	Aggregated	`$F{totals}`, calculations

Band Design Strategies

Title Band - Use Parameters:

```
<title>
    <band height="50">
        <textField>
            <reportElement x="0" y="0" width="400" height="30"/>
```

```
            <textElement>
                <font size="18" isBold="true"/>
            </textElement>
            <textFieldExpression><![CDATA[$P{reportTitle}]]></textFieldExpression>
        </textField>
        <textField pattern="MMMMM dd, yyyy">
            <reportElement x="0" y="30" width="200" height="20"/>
            <textFieldExpression><![CDATA[$P{reportDate}]]></textFieldExpression>
        </textField>
    </band>
</title>
```

Detail Band - Use Data Fields:

```
<detail>
    <band height="25">
        <textField>
            <reportElement x="0" y="0" width="120" height="20"/>
            <box><pen lineWidth="0.5" lineColor="#CCCCCC"/></box>
            <textFieldExpression><![CDATA[$F{productName}]]></textFieldExpression>
        </textField>
        <textField pattern="#,##0">
            <reportElement x="120" y="0" width="80" height="20"/>
            <box><pen lineWidth="0.5" lineColor="#CCCCCC"/></box>
            <textElement textAlignment="Right"/>
            <textFieldExpression><![CDATA[$F{quantity}]]></textFieldExpression>
        </textField>
        <textField pattern="#,##0.00">
            <reportElement x="200" y="0" width="100" height="20"/>
            <box><pen lineWidth="0.5" lineColor="#CCCCCC"/></box>
            <textElement textAlignment="Right"/>
            <textFieldExpression><![CDATA[$F{unitPrice}]]></textFieldExpression>
        </textField>
    </band>
</detail>
```

Summary Band - Use Aggregated Fields:

```
<summary>
    <band height="40">
        <textField pattern="#,##0.00">
            <reportElement x="200" y="10" width="100" height="20"/>
            <textElement textAlignment="Right">
                <font isBold="true"/>
            </textElement>
            <textFieldExpression><![CDATA[$F{totalAmount}]]></textFieldExpression>
        </textField>
        <staticText>
            <reportElement x="100" y="10" width="100" height="20"/>
            <textElement textAlignment="Right">
                <font isBold="true"/>
            </textElement>
            <text><![CDATA[Total:]]></text>
        </staticText>
    </band>
</summary>
```

Template Design Patterns

Pattern 1: Invoice/Order Template

Use Case: Generate invoices with header information and line items.

Data Structure:

```
{
  "parameters": [
    {"name": "invoiceNumber", "value": "INV-2025-001"},
    {"name": "customerName", "value": "ACME Corp"}
  ],
  "data": {
    "lineItems": [
      {"description": "Product A", "qty": 2, "price": 50.00, "total": 100.00},
      {"description": "Product B", "qty": 1, "price": 75.00, "total": 75.00}
    ],
    "totals": {
      "subtotal": 175.00,
      "tax": 17.50,
      "total": 192.50
    }
  }
}
```

Template Design:

```
<!-- Title: Use parameters -->
<title>
    <band height="60">
        <staticText>
            <reportElement x="0" y="0" width="100" height="20"/>
            <text><![CDATA[Invoice #:]]></text>
        </staticText>
        <textField>
            <reportElement x="100" y="0" width="200" height="20"/>
            <textFieldExpression><![CDATA[$P{invoiceNumber}]]></textFieldExpression>
        </textField>
        <staticText>
            <reportElement x="0" y="20" width="100" height="20"/>
            <text><![CDATA[Customer:]]></text>
        </staticText>
        <textField>
            <reportElement x="100" y="20" width="200" height="20"/>
            <textFieldExpression><![CDATA[$P{customerName}]]></textFieldExpression>
        </textField>
    </band>
</title>

<!-- Detail: Use data source -->
<detail>
    <band height="20">
        <textField>
            <reportElement x="0" y="0" width="200" height="18"/>
            <textFieldExpression><![CDATA[$F{description}]]></textFieldExpression>
        </textField>
        <textField>
```

```
            <reportElement x="200" y="0" width="50" height="18"/>
            <textElement textAlignment="Center"/>
            <textFieldExpression><![CDATA[$F{qty}]]></textFieldExpression>
        </textField>
        <textField pattern="#,##0.00">
            <reportElement x="250" y="0" width="80" height="18"/>
            <textElement textAlignment="Right"/>
            <textFieldExpression><![CDATA[$F{price}]]></textFieldExpression>
        </textField>
        <textField pattern="#,##0.00">
            <reportElement x="330" y="0" width="80" height="18"/>
            <textElement textAlignment="Right"/>
            <textFieldExpression><![CDATA[$F{total}]]></textFieldExpression>
        </textField>
    </band>
</detail>

<!-- Summary: Use aggregated data -->
<summary>
    <band height="80">
        <textField pattern="#,##0.00">
            <reportElement x="330" y="20" width="80" height="18"/>
            <textElement textAlignment="Right"/>
            <textFieldExpression><![CDATA[$F{subtotal}]]></textFieldExpression>
        </textField>
        <textField pattern="#,##0.00">
            <reportElement x="330" y="40" width="80" height="18"/>
            <textElement textAlignment="Right"/>
            <textFieldExpression><![CDATA[$F{tax}]]></textFieldExpression>
        </textField>
        <textField pattern="#,##0.00">
            <reportElement x="330" y="60" width="80" height="18"/>
            <textElement textAlignment="Right">
                <font isBold="true"/>
            </textElement>
            <textFieldExpression><![CDATA[$F{total}]]></textFieldExpression>
        </textField>
    </band>
</summary>
```

Pattern 2: Financial Template with Multiple Sections

Use Case: Financial documents with multiple data sections (quarterly data, departmental data, etc.).

Data Structure:

```
{
  "parameters": [
    {"name": "reportYear", "value": "2025"},
    {"name": "reportType", "value": "Annual Financial Summary"}
  ],
  "data": {
    "quarters": [
      {"name": "Q1", "revenue": 1000000, "expenses": 800000, "profit": 200000},
      {"name": "Q2", "revenue": 1200000, "expenses": 850000, "profit": 350000},
      {"name": "Q3", "revenue": 1100000, "expenses": 900000, "profit": 200000},
      {"name": "Q4", "revenue": 1300000, "expenses": 950000, "profit": 350000}
```

```
    ],
    "departments": [
      {"name": "Sales", "budget": 500000, "actual": 485000, "variance": -15000},
      {"name": "Marketing", "budget": 300000, "actual": 295000, "variance": -5000},
      {"name": "R&D", "budget": 800000, "actual": 825000, "variance": 25000}
    ],
    "summary": {
      "totalRevenue": 4600000,
      "totalExpenses": 3500000,
      "netProfit": 1100000,
      "profitMargin": 23.9
    }
  }
}
```

Pattern 3: Master-Detail Template

Use Case: Documents with grouped data (customers with their orders, categories with products).

Data Structure:

```
{
  "data": {
    "customers": [
      {
        "name": "ACME Corp",
        "orders": [
          {"orderNo": "ORD-001", "date": "2025-10-01", "amount": 1500.00},
          {"orderNo": "ORD-002", "date": "2025-10-05", "amount": 2300.00}
        ]
      },
      {
        "name": "TechCorp Ltd",
        "orders": [
          {"orderNo": "ORD-003", "date": "2025-10-02", "amount": 3200.00}
        ]
      }
    ]
  }
}
```

Note: For complex master-detail relationships, consider flattening the data structure or using subreports.

Nested Data Structures

The service provides enhanced support for flexible nested data structures that go beyond simple tabular data. This powerful feature allows you to work with complex hierarchical data while maintaining both standard iteration patterns and deep object navigation capabilities.

Overview of Nested Data Support

The enhanced data source implementation provides a **hybrid approach** that supports:

1. **Standard Tabular Access**: For detail bands and iterative processing
2. **Nested Object Navigation**: For accessing deep hierarchical structures
3. **Utility Methods**: For safe data manipulation and calculations
4. **Flattened Field Access**: For convenient dot-notation field usage

Data Structure Processing

When you send complex nested data to the service, it undergoes intelligent processing:

Original Data Preservation

Your complete nested structure is preserved in the `_originalData` field:

```
{
  "data": {
    "customer": {
      "name": "ACME Corp",
      "address": {
        "street": "123 Main St",
        "city": "Springfield",
        "country": "USA"
      },
      "contacts": [
        {"name": "John Doe", "email": "john@acme.com", "role": "Manager"},
        {"name": "Jane Smith", "email": "jane@acme.com", "role": "Developer"}
      ]
    },
    "orders": [
      {
        "id": "ORD-001",
        "date": "2024-01-15",
        "items": [
          {"product": "Widget A", "quantity": 5, "price": 25.99},
          {"product": "Widget B", "quantity": 3, "price": 15.50}
        ],
        "total": 176.45
      }
    ]
  }
}
```

Flattened Field Creation

The service automatically creates flattened fields with dot notation:

Flattened Field	Original Path	Value
`customer_name`	`customer.name`	“ACME Corp”
`customer_address_street`	`customer.address.street`	“123 Main St”
`customer_address_city`	`customer.address.city`	“Springfield”
`customer_address_country`	`customer.address.country`	“USA”

Metadata Fields

Each data row includes helpful metadata for navigation and filtering:

Field	Type	Description
`_originalData`	Map	Complete original nested structure
`_sharedData`	Map	Non-array data available in all rows
`_sourceArray`	String	Name of the source array for this row
`_sourceIndex`	Integer	Index within the source array (0-based)
`_sourceArraySize`	Integer	Size of the source array
`_globalIndex`	Integer	Global row index across all arrays
`_totalRows`	Integer	Total number of rows in the dataset
`_arrayFields`	List	List of field names that contain arrays
`_recordType`	Object	Uses `record_category` if present, otherwise source array name

JasperSoft Studio Setup

To use the utility methods in JasperSoft Studio 7.x, you need to add the utility JAR to your classpath. The utility class is automatically available when generating reports through the Muban service, but for design-time preview in JasperSoft Studio, you must configure it manually.

> **Important:** Do NOT place the JAR in JasperSoft Studio’s `plugins` folder. That folder is for Eclipse/OSGi plugins only, not for regular Java libraries.

Step 1: Obtain the Utility JAR

You have two options to get the utility JAR file:

Option A: **Download Pre-built JAR (Recommended for Non-Developers)**

Download the `jasperreports-utils.jar` file from your organization’s artifact repository or request it from your development team.

Option B: **Build from Source (For Developers)**

If you have access to the Muban project source code and Maven installed:

```
# Navigate to the project root directory
cd <project-root>

# Compile the project
mvn compile

# Create the utility JAR file
jar -cf jasperreports-utils.jar -C target/classes me/muban/jasperreports/utils/
```

This creates a file named `jasperreports-utils.jar` in your current directory.

Step 2: Add JAR to JasperSoft Studio

Choose ONE of the following methods:

Method 1: **Global Classpath (Recommended)**

This makes the utility available for ALL reports in JasperSoft Studio.

1. Open JasperSoft Studio
2. Go to menu: `Window` -> `Preferences`
3. In the left panel, expand `JasperSoft Studio` and click on `Classpath`
4. Click the `Add JARs...` button
5. Navigate to and select your `jasperreports-utils.jar` file
6. Click `Open`
7. Click `Apply and Close`
8. **Restart JasperSoft Studio** to ensure the changes take effect

Method 2: **Project-Specific Classpath**

This makes the utility available only for a specific JasperSoft Studio project.

1. In the `Project Explorer` panel, right-click on your project
2. Select `Properties`
3. In the left panel, click on `Java Build Path`
4. Click the `Libraries` tab
5. Click `Add External JARs...`
6. Navigate to and select your `jasperreports-utils.jar` file
7. Click `Open`
8. Click `Apply and Close`

Method 3: **Project lib Folder**

This is useful when sharing projects with team members.

1. Create a folder named `lib` inside your JasperSoft Studio project folder
2. Copy `jasperreports-utils.jar` into the `lib` folder
3. In JasperSoft Studio, right-click on your project -> `Refresh`
4. Right-click on your project -> `Properties`
5. Go to `Java Build Path` -> `Libraries` tab
6. Click `Add JARs...` (not “Add External JARs”)
7. Navigate to `your-project/lib/jasperreports-utils.jar`
8. Click `OK`, then `Apply and Close`

> **Tip:** When using Method 3, the JAR is stored within the project. If you share the project (e.g., via Git), team members will have the JAR automatically.

Step 3: Verify the Setup

After adding the JAR, verify it’s working correctly:

1. Open any template (`.jrxml` file) in the Design view
2. Add a new Text Field to your template
3. In the Expression Editor, type the following expression:

```
me.muban.jasperreports.utils.JasperReportsUtils.getNestedValue($P{REPORT_PARAMETERS_MAP}, "test", "It
↪  works!")
```

1. If the setup is correct:
   - No red error underline appears under the class name
   - Auto-completion suggests methods when you type `JasperReportsUtils.`
2. If you see an error like “Class not found” or red underline:
   - Restart JasperSoft Studio and try again
   - Verify the JAR file path is correct
   - Check that you clicked “Apply” in the preferences

Troubleshooting

Problem	Solution
“Class not found” error	Restart JasperSoft Studio after adding the JAR
No auto-completion for methods	Ensure you added the JAR to the correct classpath (global or project)
JAR not visible in Add JARs dialog	Check the file has `.jar` extension and is not corrupted
Changes not taking effect	Close all template editors, restart JasperSoft Studio
“NoClassDefFoundError” at runtime	The JAR is not on the classpath; re-add it using the steps above

Note on Service vs Designer

- **In JasperSoft Studio (Designer):** You must manually add the JAR as described above
- **In the Muban Service (Runtime):** The utility is automatically available - no configuration needed

When you upload your template to the Muban service, the utility methods will work automatically because the class is part of the service’s classpath.

Nested Field Access Patterns

Simple Flattened Fields

Access nested properties using flattened field names:

```
<!-- Customer information -->
<textField>
    <reportElement uuid="field-uuid-001" x="0" y="0" width="200" height="20"/>
    <textFieldExpression><![CDATA[$F{customer_name}]]></textFieldExpression>
</textField>

<textField>
    <reportElement uuid="field-uuid-002" x="0" y="25" width="300" height="20"/>
    <textFieldExpression><![CDATA[
        $F{customer_address_street} + ", " +
        $F{customer_address_city} + ", " +
        $F{customer_address_country}
    ]]></textFieldExpression>
</textField>
```

Utility Method Access

Use built-in utility methods for safe data navigation:

```
<!-- Safe nested value access -->
<textField>
    <reportElement uuid="field-uuid-003" x="0" y="50" width="200" height="20"/>
    <textFieldExpression><![CDATA[
        me.muban.jasperreports.utils.JasperReportsUtils.getNestedValue(
            $F{_originalData}, "customer.address.city", "N/A"
        )
    ]]></textFieldExpression>
</textField>

<!-- Array element access -->
<textField>
    <reportElement uuid="field-uuid-004" x="0" y="75" width="200" height="20"/>
    <textFieldExpression><![CDATA[
        me.muban.jasperreports.utils.JasperReportsUtils.getArrayElement(
            $F{_originalData}, "customer.contacts", 0, "name", "No Contact"
        )
    ]]></textFieldExpression>
</textField>
```

Available Utility Methods

The service provides nine static utility methods for template use:

getNestedValue(object, path, defaultValue)

Safely retrieves nested values using dot notation:

```
<textField>
    <reportElement uuid="utility-uuid-001" x="0" y="0" width="150" height="18"/>
    <textFieldExpression><![CDATA[
        me.muban.jasperreports.utils.JasperReportsUtils.getNestedValue(
            $F{_originalData}, "product.specifications.weight", "Unknown"
        )
    ]]></textFieldExpression>
</textField>
```

Parameters:

- `object`: The data object (usually `$F{_originalData}`)
- `path`: Dot-notation path (e.g., “customer.address.city”)
- `defaultValue`: Value returned if path doesn’t exist

getArrayElement(object, arrayPath, index, property, defaultValue)

Access specific elements from nested arrays:

```
<textField>
    <reportElement uuid="utility-uuid-002" x="0" y="20" width="200" height="18"/>
    <textFieldExpression><![CDATA[
        me.muban.jasperreports.utils.JasperReportsUtils.getArrayElement(
            $F{_originalData}, "orders.items", 0, "product", "No Product"
        )
    ]]></textFieldExpression>
</textField>
```

Parameters:

- `object`: The data object
- `arrayPath`: Path to the array (e.g., "orders.items")
- `index`: Array index (0-based)
- `property`: Property name within array element (null for direct element access)
- `defaultValue`: Default if element/property doesn't exist

calculateSum(object, arrayPath, property)

Calculate sums from array properties:

```
<textField pattern="#,##0.00">
    <reportElement uuid="utility-uuid-003" x="0" y="40" width="100" height="18"/>
    <textFieldExpression><![CDATA[
        me.muban.jasperreports.utils.JasperReportsUtils.calculateSum(
            $F{_originalData}, "orders.items", "price"
        )
    ]]></textFieldExpression>
</textField>
```

Parameters:

- `object`: The data object containing the array
- `arrayPath`: Path to the array within the data object
- `property`: Property name to sum within array elements
- **Returns:** Sum of all numeric values, or 0.0 if no valid values found

calculateAverage(object, arrayPath, property)

Calculate averages from array properties:

```
<textField pattern="#,##0.00">
    <reportElement uuid="utility-uuid-004" x="110" y="40" width="100" height="18"/>
    <textFieldExpression><![CDATA[
        me.muban.jasperreports.utils.JasperReportsUtils.calculateAverage(
            $F{_originalData}, "orders.items", "quantity"
        )
    ]]></textFieldExpression>
</textField>
```

Parameters:

- `object`: The data object containing the array
- `arrayPath`: Path to the array within the data object
- `property`: Property name to average within array elements
- **Returns:** Average of all numeric values, or 0.0 if no valid values found

countValues(object, arrayPath, property, targetValue)

Count occurrences of specific values:

```
<textField>
    <reportElement uuid="utility-uuid-005" x="220" y="40" width="80" height="18"/>
    <textFieldExpression><![CDATA[
        me.muban.jasperreports.utils.JasperReportsUtils.countValues(
            $F{_originalData}, "orders.items", "category", "Electronics"
```

```
        )
    ]]></textFieldExpression>
</textField>
```

Parameters:

- `object`: The data object containing the array
- `arrayPath`: Path to the array within the data object
- `property`: Property name to check within array elements
- `targetValue`: Value to count occurrences of
- **Returns:** Number of elements where property equals targetValue

countNonNullValues(object, arrayPath, property)

Count non-null values in array properties:

```
<textField>
    <reportElement uuid="utility-uuid-006" x="0" y="60" width="100" height="18"/>
    <textFieldExpression><![CDATA[
        me.muban.jasperreports.utils.JasperReportsUtils.countNonNullValues(
            $F{_originalData}, "orders.items", "discount"
        )
    ]]></textFieldExpression>
</textField>
```

Parameters:

- `object`: The data object containing the array
- `arrayPath`: Path to the array within the data object
- `property`: Property name to check within array elements
- **Returns:** Number of elements where property is not null

getArraySize(object, arrayPath)

Get the size of an array or collection:

```
<textField>
    <reportElement uuid="utility-uuid-007" x="110" y="60" width="100" height="18"/>
    <textFieldExpression><![CDATA[
        "Items: " + me.muban.jasperreports.utils.JasperReportsUtils.getArraySize(
            $F{_originalData}, "orders.items"
        )
    ]]></textFieldExpression>
</textField>
```

Parameters:

- `object`: The data object containing the array
- `arrayPath`: Path to the array within the data object
- **Returns:** Size of the array, or 0 if not found or not an array

findMaxValue(object, arrayPath, property)

Find the maximum numeric value in array properties:

```
<textField pattern="#,##0.00">
    <reportElement uuid="utility-uuid-008" x="220" y="60" width="100" height="18"/>
    <textFieldExpression><![CDATA[
        me.muban.jasperreports.utils.JasperReportsUtils.findMaxValue(
            $F{_originalData}, "orders.items", "price"
        )
    ]]></textFieldExpression>
</textField>
```

Parameters:

- `object`: The data object containing the array
- `arrayPath`: Path to the array within the data object
- `property`: Property name to find maximum within array elements
- **Returns:** Maximum numeric value, or null if no valid values found

findMinValue(object, arrayPath, property)

Find the minimum numeric value in array properties:

```
<textField pattern="#,##0.00">
    <reportElement uuid="utility-uuid-009" x="330" y="60" width="100" height="18"/>
    <textFieldExpression><![CDATA[
        me.muban.jasperreports.utils.JasperReportsUtils.findMinValue(
            $F{_originalData}, "orders.items", "price"
        )
    ]]></textFieldExpression>
</textField>
```

Parameters:

- `object`: The data object containing the array
- `arrayPath`: Path to the array within the data object
- `property`: Property name to find minimum within array elements
- **Returns:** Minimum numeric value, or null if no valid values found

Advanced Usage Patterns

Master-Detail with Nested Arrays

Create sophisticated reports that combine tabular iteration with nested data access:

```
<!-- Master section using flattened fields -->
<band height="60">
    <textField>
        <reportElement uuid="master-uuid-001" x="0" y="0" width="200" height="20"/>
        <textElement>
            <font size="14" isBold="true"/>
        </textElement>
        <textFieldExpression><![CDATA["Order: " + $F{order_id}]]></textFieldExpression>
    </textField>

    <textField>
        <reportElement uuid="master-uuid-002" x="0" y="25" width="300" height="20"/>
        <textFieldExpression><![CDATA[$F{customer_name} + " - " +
↪   $F{customer_address_city}]]></textFieldExpression>
    </textField>
```

```
    <!-- Calculated totals using utility methods -->
    <textField pattern="$#,##0.00">
        <reportElement uuid="master-uuid-003" x="400" y="25" width="100" height="20"/>
        <textElement textAlignment="Right">
            <font isBold="true"/>
        </textElement>
        <textFieldExpression><![CDATA[
            me.muban.jasperreports.utils.JasperReportsUtils.calculateSum(
                $F{_originalData}, "items", "total"
            )
        ]]></textFieldExpression>
    </textField>
</band>
```

Conditional Logic with Nested Data

Implement complex conditional logic using nested data:

```
<textField>
    <reportElement uuid="conditional-uuid-001" x="0" y="0" width="200" height="20"/>
    <textFieldExpression><![CDATA[
        (me.muban.jasperreports.utils.JasperReportsUtils.getNestedValue(
            $F{_originalData}, "customer.type", "regular"
        ).equals("premium")) ? "[*] Premium Customer" : "Regular Customer"
    ]]></textFieldExpression>
</textField>
```

Dynamic Content Based on Array Length

Adjust content based on the size of nested arrays:

```
<textField>
    <reportElement uuid="dynamic-uuid-001" x="0" y="0" width="300" height="20"/>
    <textFieldExpression><![CDATA[
        ($F{_originalData} instanceof java.util.Map &&
         ((java.util.Map)$F{_originalData}).get("items") instanceof java.util.List &&
         ((java.util.List)((java.util.Map)$F{_originalData}).get("items")).size() > 5) ?
        "Large Order (" + ((java.util.List)((java.util.Map)$F{_originalData}).get("items")).size() + "
↪   items)" :
        "Standard Order"
    ]]></textFieldExpression>
</textField>
```

Best Practices for Nested Data

Data Structure Design

- **Keep it Logical**: Structure your data to match your template layout
- **Avoid Deep Nesting**: More than 4-5 levels can become difficult to manage
- **Use Consistent Naming**: Follow consistent property naming conventions
- **Include Metadata**: Add helpful metadata fields for calculations

Template Design

- **Use Flattened Fields**: For simple property access, use flattened field names
- **Leverage Utility Methods**: For complex operations, use the provided utility methods

- **Add Null Checks**: Always provide default values for safe navigation
- **Test Edge Cases**: Test with empty arrays and missing properties

Performance Considerations

- **Minimize Deep Access**: Cache frequently accessed nested values in variables
- **Batch Calculations**: Use utility methods rather than manual loops
- **Optimize Expressions**: Keep field expressions simple and readable

Example: Complex Invoice Template

Here's a comprehensive example showing nested data usage in a complex invoice template:


```
{
  "data": {
    "invoice": {
      "number": "INV-2024-001",
      "date": "2024-01-15",
      "dueDate": "2024-02-15"
    },
    "vendor": {
      "name": "ABC Supplies Inc.",
      "address": {
        "street": "456 Business Ave",
        "city": "Commerce City",
        "state": "CO",
        "zip": "80022"
      },
      "contact": {
        "phone": "(555) 123-4567",
        "email": "billing@abcsupplies.com"
      }
    },
    "customer": {
      "name": "XYZ Manufacturing",
      "account": "CUST-789",
      "address": {
        "street": "789 Industrial Blvd",
        "city": "Factory Town",
        "state": "TX",
        "zip": "75001"
      }
    },
    "lineItems": [
      {
        "sku": "WIDGET-001",
        "description": "Premium Widget Assembly",
        "quantity": 10,
        "unitPrice": 24.99,
        "discount": 0.05,
        "total": 237.41
      },
      {
        "sku": "PART-205",
        "description": "Standard Component Kit",
        "quantity": 25,
        "unitPrice": 8.75,
        "discount": 0.00,
```

```
        "total": 218.75
      }
    ],
    "totals": {
      "subtotal": 456.16,
      "tax": 36.49,
      "shipping": 15.00,
      "grandTotal": 507.65
    },
    "terms": {
      "paymentDue": 30,
      "lateFee": 0.015,
      "discountTerms": "2/10 net 30"
    }
  }
}
```

Template Usage:

```
<!-- Invoice Header -->
<textField>
    <reportElement uuid="header-uuid-001" x="0" y="0" width="200" height="24"/>
    <textElement>
        <font size="18" isBold="true"/>
    </textElement>
    <textFieldExpression><![CDATA["Invoice " + $F{invoice_number}]]></textFieldExpression>
</textField>

<!-- Vendor Information using flattened fields -->
<textField>
    <reportElement uuid="vendor-uuid-001" x="0" y="40" width="200" height="60"/>
    <textFieldExpression><![CDATA[
        $F{vendor_name} + "\n" +
        $F{vendor_address_street} + "\n" +
        $F{vendor_address_city} + ", " + $F{vendor_address_state} + " " + $F{vendor_address_zip}
    ]]></textFieldExpression>
</textField>

<!-- Customer Information using utility methods -->
<textField>
    <reportElement uuid="customer-uuid-001" x="300" y="40" width="200" height="60"/>
    <textFieldExpression><![CDATA[
        me.muban.jasperreports.utils.JasperReportsUtils.getNestedValue(
            $F{_originalData}, "customer.name", "Unknown Customer"
        ) + "\n" +
        "Account: " + me.muban.jasperreports.utils.JasperReportsUtils.getNestedValue(
            $F{_originalData}, "customer.account", "N/A"
        ) + "\n" +
        me.muban.jasperreports.utils.JasperReportsUtils.getNestedValue(
            $F{_originalData}, "customer.address.street", ""
        ) + "\n" +
        me.muban.jasperreports.utils.JasperReportsUtils.getNestedValue(
            $F{_originalData}, "customer.address.city", ""
        ) + ", " +
        me.muban.jasperreports.utils.JasperReportsUtils.getNestedValue(
            $F{_originalData}, "customer.address.state", ""
        ) + " " +
```

```
        me.muban.jasperreports.utils.JasperReportsUtils.getNestedValue(
            $F{_originalData}, "customer.address.zip", ""
        )
    ]]></textFieldExpression>
</textField>

<!-- Line Items Count -->
<textField>
    <reportElement uuid="items-uuid-001" x="0" y="120" width="300" height="20"/>
    <textFieldExpression><![CDATA[
        "Line Items: " +
        ($F{_originalData} instanceof java.util.Map &&
          ((java.util.Map)$F{_originalData}).get("lineItems") instanceof java.util.List) ?
         ((java.util.List)((java.util.Map)$F{_originalData}).get("lineItems")).size() : 0)
    ]]></textFieldExpression>
</textField>

<!-- Totals using flattened fields -->
<textField pattern="$#,##0.00">
    <reportElement uuid="total-uuid-001" x="400" y="200" width="100" height="20"/>
    <textElement textAlignment="Right">
        <font size="14" isBold="true"/>
    </textElement>
    <textFieldExpression><![CDATA[$F{totals_grandTotal}]]></textFieldExpression>
</textField>
```

This nested data structure approach provides unprecedented flexibility while maintaining simplicity for basic use cases. You can seamlessly combine standard tabular data processing with sophisticated nested object navigation to create rich, dynamic documents.

Advanced Techniques

Calculated Fields

Create calculated fields using expressions:

```
<!-- Calculate percentage -->
<textField pattern="#,##0.0%">
    <reportElement x="200" y="0" width="80" height="18"/>
    <textElement textAlignment="Right"/>
    <textFieldExpression><![CDATA[
        $F{actualSales}.doubleValue() / $F{targetSales}.doubleValue()
    ]]></textFieldExpression>
</textField>

<!-- Conditional formatting -->
<textField>
    <reportElement x="0" y="0" width="100" height="18"/>
    <textElement>
        <font isBold="true"/>
    </textElement>
    <textFieldExpression><![CDATA[
        $F{variance} < 0 ? "[!] Under Budget" : "[OK] On Target"
    ]]></textFieldExpression>
</textField>

<!-- Complex calculations -->
<textField pattern="#,##0.00">
    <reportElement x="300" y="0" width="100" height="18"/>
    <textElement textAlignment="Right"/>
    <textFieldExpression><![CDATA[
        ($F{revenue} - $F{expenses}) / $F{revenue} * 100
    ]]></textFieldExpression>
</textField>
```

Conditional Formatting

```
<!-- Conditional colors -->
<textField>
    <reportElement x="0" y="0" width="100" height="18">
        <conditionalStyle>
            <conditionExpression><![CDATA[$F{profit} < 0]]></conditionExpression>
            <style forecolor="#FF0000" isBold="true"/>
        </conditionalStyle>
        <conditionalStyle>
            <conditionExpression><![CDATA[$F{profit} > 100000]]></conditionExpression>
            <style forecolor="#00AA00" isBold="true"/>
        </conditionalStyle>
    </reportElement>
    <textFieldExpression><![CDATA[$F{profit}]]></textFieldExpression>
</textField>
```

Variable Usage with Data Sources

```
<!-- Define variables for calculations -->
<variable name="runningTotal" class="java.math.BigDecimal" calculation="Sum">
```

```
    <variableExpression><![CDATA[$F{amount}]]></variableExpression>
</variable>

<variable name="averageAmount" class="java.math.BigDecimal" calculation="Average">
    <variableExpression><![CDATA[$F{amount}]]></variableExpression>
</variable>

<!-- Use variables in expressions -->
<textField pattern="#,##0.00">
    <reportElement x="0" y="0" width="100" height="18"/>
    <textElement textAlignment="Right"/>
    <textFieldExpression><![CDATA[$V{runningTotal}]]></textFieldExpression>
</textField>
```

Null Handling

```
<!-- Safe null handling -->
<textField>
    <reportElement x="0" y="0" width="100" height="18"/>
    <textFieldExpression><![CDATA[
        $F{optionalField} != null ? $F{optionalField} : "N/A"
    ]]></textFieldExpression>
</textField>

<!-- Null-safe calculations -->
<textField pattern="#,##0.00">
    <reportElement x="100" y="0" width="80" height="18"/>
    <textElement textAlignment="Right"/>
    <textFieldExpression><![CDATA[
        ($F{quantity} != null && $F{price} != null)
            ? $F{quantity}.doubleValue() * $F{price}.doubleValue()
            : 0.0
    ]]></textFieldExpression>
</textField>
```

Data Source Configuration in JasperSoft Studio

Understanding Data Source Declaration

When working with the Document Generation Service, you need to configure your JasperSoft Studio template to expect data in the correct format. The service automatically creates a `JRBeanCollectionDataSource` from your JSON data, but you need to set up your template properly.

Data Source Setup Methods

Method 1: Empty Data Source (Recommended for Service Integration)

This is the most common approach when designing templates for the Document Generation Service:

1. **Open Template Query Designer**:
   - Right-click on your template in the Outline view
   - Select **Dataset and Query** -> **Edit Query**
2. **Configure Empty Data Source**:
   - Language: Select **"Empty datasource"**
   - Query: Leave completely empty
   - Records: Set to a reasonable number for preview (e.g., 5-10)
3. **Add Fields Manually**:
   - In the Outline view, expand **Fields**
   - Right-click **Fields** -> **Create Field**
   - Configure each field with the exact name from your JSON data

Example Field Configuration:

```
Field Name: customerName
Class: java.lang.String
Description: Customer full name

Field Name: orderTotal
Class: java.math.BigDecimal
Description: Order total amount

Field Name: lineItems
Class: java.util.List
Description: List of order line items
```

Method 2: JSON Data Source (For Design-Time Preview)

Use this method to test your template with sample data during design:

1. **Create Sample JSON File**:

```json
{
  "items": [
    {"product": "Widget A", "price": 25.99, "quantity": 2},
    {"product": "Widget B", "price": 15.50, "quantity": 1}
  ]
}
```

2. **Configure JSON Data Source**:
   - Language: Select **"JSON"**

- Data Source Expression: `"file:sample_data.json"`
- JSON Expression: `$.items` (for array data) or `$` (for root object)

3. **Auto-Generate Fields**:
   - Click **“Read Fields”** button
   - JasperSoft Studio will automatically detect and create fields

Method 3: JavaBean Collection (Alternative Service Method)

This method mimics exactly how the service provides data:

1. **Configure Bean Collection**:
   - Language: Select **“JavaBean datasource”**
   - Query: Leave empty
   - The service will inject the data source at runtime
2. **Manual Field Definition**:
   - Create fields that match your expected JSON structure
   - Use exact field names as they appear in your data

Data Source Naming Conventions

The Document Generation Service follows these naming patterns:

JSON Field Mapping

Your JSON data structure determines the field names available in the template:

```
{
  "data": {
    "customer": {
      "name": "ACME Corp",
      "address": "123 Main St"
    },
    "items": [
      {"product": "Widget", "price": 25.99}
    ]
  }
}
```

Resulting JasperReports Fields:

JSON Path	Field Name	Field Type	Usage
`customer.name`	`customer_name`	String	`$F{customer_name}`
`customer.address`	`customer_address`	String	`$F{customer_address}`
`items[].product`	`product`	String	`$F{product}`
`items[].price`	`price`	BigDecimal	`$F{price}`

Field Naming Rules

1. **Nested Objects**: Flattened with underscores
   - `customer.address.street` -> `customer_address_street`
   - `order.billing.city` -> `order_billing_city`
2. **Array Elements**: Direct field access in detail bands

- Arrays create rows, properties become fields
- `items[0].product` -> `$F{product}` in detail band

3. **Special Fields**: Service-generated metadata (UNION strategy)
   - `_originalData` -> Complete nested data structure
   - `_sourceArray` -> Name of the source array for this row
   - `_sourceIndex` -> Index within the source array (0-based)
   - `_globalIndex` -> Global row index across all arrays
   - `_totalRows` -> Total number of rows
   - `_recordType` -> Record category or source array name (for filtering)

Service Integration Examples

Example 1: Invoice with Line Items

Service Request Data:

```
{
  "parameters": [
    {"name": "invoiceNumber", "value": "INV-2025-001"}
  ],
  "data": {
    "customer": {
      "name": "ACME Corporation",
      "email": "billing@acme.com"
    },
    "lineItems": [
      {"description": "Widget A", "quantity": 2, "price": 25.99, "total": 51.98},
      {"description": "Widget B", "quantity": 1, "price": 35.00, "total": 35.00}
    ],
    "totals": {
      "subtotal": 86.98,
      "tax": 8.70,
      "total": 95.68
    }
  }
}
```

JasperSoft Studio Field Setup:

```
Parameters:
- invoiceNumber (String)

Fields (for empty datasource):
- customer_name (String)
- customer_email (String)
- description (String)     // From lineItems array
- quantity (Integer)       // From lineItems array
- price (BigDecimal)       // From lineItems array
- total (BigDecimal)       // From lineItems array
- subtotal (BigDecimal)    // From totals object
- tax (BigDecimal)         // From totals object
- grandTotal (BigDecimal) // From totals object (maps to totals.total)
```

Example 2: Using _originalData for Complex Access

Template Expression Examples:

```
<!-- Access nested data using utility methods -->
<textField>
    <textFieldExpression><![CDATA[
        me.muban.jasperreports.utils.JasperReportsUtils.getNestedValue(
            $F{_originalData}, "customer.address.street", "N/A"
        )
    ]]></textFieldExpression>
</textField>

<!-- Calculate array sums -->
<textField pattern="#,##0.00">
    <textFieldExpression><![CDATA[
        me.muban.jasperreports.utils.JasperReportsUtils.calculateSum(
            $F{_originalData}, "lineItems", "total"
        )
    ]]></textFieldExpression>
</textField>
```

Troubleshooting Data Source Issues

Common Problems

1. **Field Not Found Error**

   ```
   Problem: net.sf.jasperreports.engine.JRException: Field 'fieldName' not found

   Solution:
   - Check field name spelling and case sensitivity
   - Verify JSON structure matches field definitions
   - Ensure data array is not empty
   ```

2. **Empty Document**

   ```
   Problem: Document generates but shows no data rows

   Solution:
   - Check that detail band height > 0
   - Verify data contains array for iteration
   - Test with simple data structure first
   ```

3. **Data Source Connection Issues**

   ```
   Problem: Template works in Studio but fails in service

   Solution:
   - Use "Empty datasource" instead of JSON datasource
   - Remove file references from data source expression
   - Ensure field types match service data types
   ```

Best Practices for Data Source Setup

1. **Use Empty Data Source**: Always use “Empty datasource” for service integration
2. **Match Field Names Exactly**: Field names must match JSON property names precisely
3. **Test with Sample Data**: Create sample JSON files for design-time testing

4. **Document Field Mapping**: Keep a reference of JSON -> Field mappings
5. **Use Consistent Naming**: Follow consistent naming conventions in your JSON data
6. **Handle Null Values**: Always include null checks in field expressions

Testing and Debugging

Using Sample Data in Jaspersoft Studio

Create JSON Sample Data

Create a file `sample_data.json`:

```
{
  "items": [
    {"product": "Widget A", "quantity": 10, "price": 25.00, "total": 250.00},
    {"product": "Widget B", "quantity": 5, "price": 50.00, "total": 250.00}
  ],
  "summary": {
    "subtotal": 500.00,
    "tax": 50.00,
    "total": 550.00
  }
}
```

Configure Data Source in Studio

Method 1: **JSON Data Source (Recommended)**

1. Right-click on your template -> **Edit Query**
2. Select **JSON** as query language
3. Set **Data Source Expression**: `"sample_data.json"`
4. Configure field mapping
5. Use **Preview** to test

Method 2: **Bean Collection Data Source (For Service Integration)**

1. In the **Template Inspector**, right-click on your template
2. Select **Edit Query**
3. Choose **JavaBean** as the query language
4. Leave the query field empty (the service will provide the data source)
5. Create fields manually or use the service's automatic field detection

Method 3: **Empty Data Source with Manual Fields**

1. Right-click your template -> **Edit Query**
2. Select **Empty datasource** as query language
3. Manually add fields in the **Fields** section of the Outline view
4. This method requires you to know the exact field structure beforehand

Alternative: Bean Collection Data Source

```
// Create test data collection
List<Map<String, Object>> testData = Arrays.asList(
    Map.of("product", "Widget A", "quantity", 10, "price", 25.00),
    Map.of("product", "Widget B", "quantity", 5, "price", 50.00)
);

// Use in Jaspersoft Studio preview
```

Field Mapping Verification

```
<!-- Add debug fields to verify data access -->
<textField>
    <reportElement x="0" y="0" width="200" height="20"/>
    <textFieldExpression><![CDATA[
        "Debug: product=" + $F{product} + ", qty=" + $F{quantity}
    ]]></textFieldExpression>
</textField>
```

Service Testing with curl

Basic Parameter Test

```
curl -X POST "http://localhost:8080/api/v1/templates/{id}/pdf" \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer <token>" \
  -d '{
    "parameters": [
      {"name": "title", "value": "Test Report"}
    ]
  }' \
  --output test_report.pdf
```

Data Source Test

```
curl -X POST "http://localhost:8080/api/v1/templates/{id}/pdf" \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer <token>" \
  -d '{
    "parameters": [
      {"name": "reportTitle", "value": "Test Data Report"}
    ],
    "data": {
      "testItems": [
        {"name": "Item 1", "value": 100},
        {"name": "Item 2", "value": 200}
      ]
    }
  }' \
  --output test_data_report.pdf
```

Debugging Common Issues

Field Not Found Errors

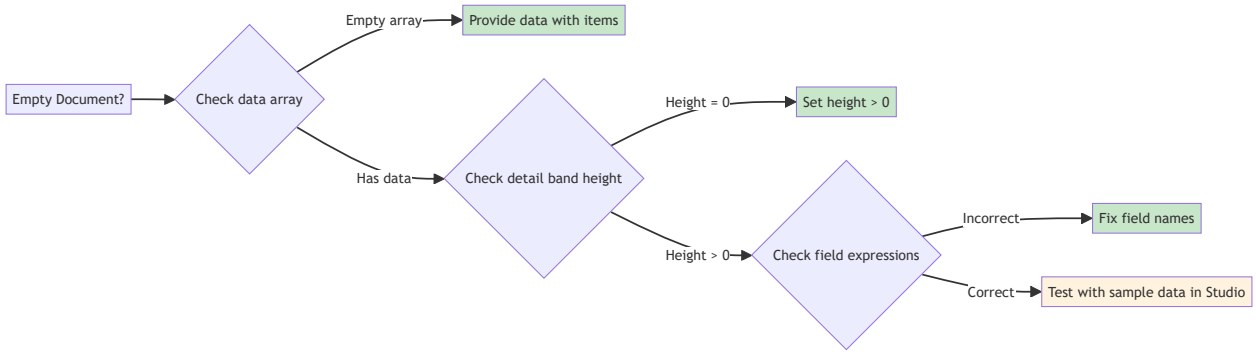
Error: `Field 'productName' not found`

Scenario	Expression	Data	Result
Correct	`$F{productName}`	`{"productName": "Widget"}`	Works
Wrong	`$F{product_name}`	`{"productName": "Widget"}`	Field not found

Solution: Field names are case-sensitive and must match exactly.

Empty Documents

Problem: Document generates but shows no data



Parameter Type Errors

Error: `Cannot convert 'abc' to Integer`

Scenario	Parameter Definition	Value	Result
Correct	`Integer count`	`{"name": "count", "value": "123"}`	Works
Wrong	`Integer count`	`{"name": "count", "value": "abc"}`	Conversion error

Solution: Ensure parameter values match the expected type.

Font Embedding and PDF/A Conformance

Introduction to Font Embedding

The Document Generation Service provides automatic font embedding and PDF/A conformance validation for professional-quality PDF generation. This feature ensures that generated PDF documents are archival-compliant and display correctly across all platforms and devices.

Key Features

- **Automatic Font Loading**: Custom fonts are automatically loaded from template packages
- **PDF/A Validation**: Pre-generation conformance checking for PDF/A-1b standard
- **Manifest-Based Configuration**: Simple XML descriptor for font metadata
- **Error Prevention**: Early validation prevents generation of non-compliant PDFs

Font Manifest Structure

Creating fonts.xml

Place a `fonts.xml` file in the root of your template package (alongside your .jrxml files):

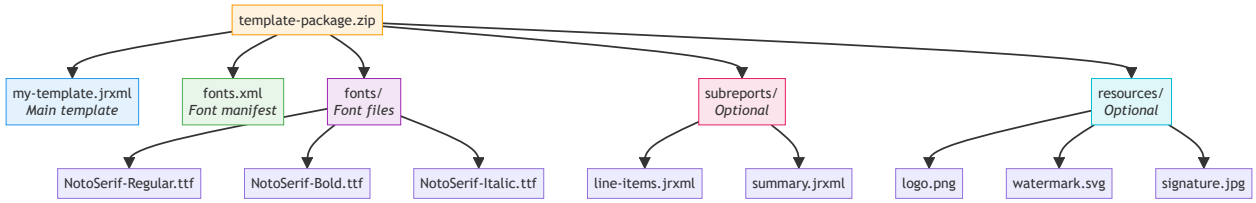
```
<?xml version="1.0" encoding="UTF-8"?>
<fontFamilies>
    <fontFamily name="DejaVu Sans">
        <normal>fonts/DejaVuSans.ttf</normal>
        <bold>fonts/DejaVuSans-Bold.ttf</bold>
        <italic>fonts/DejaVuSans-Oblique.ttf</italic>
        <boldItalic>fonts/DejaVuSans-BoldOblique.ttf</boldItalic>
        <pdfEncoding>Identity-H</pdfEncoding>
        <pdfEmbedded>true</pdfEmbedded>
    </fontFamily>

    <fontFamily name="Noto Serif">
        <normal>fonts/NotoSerif-Regular.ttf</normal>
        <bold>fonts/NotoSerif-Bold.ttf</bold>
        <italic>fonts/NotoSerif-Italic.ttf</italic>
        <boldItalic>fonts/NotoSerif-BoldItalic.ttf</boldItalic>
        <pdfEncoding>Identity-H</pdfEncoding>
        <pdfEmbedded>true</pdfEmbedded>
    </fontFamily>
</fontFamilies>
```

Manifest Elements

Element	Required	Description
`fontFamily`	Yes	Container for a single font family
`name` attribute	Yes	Font family name used in JRXML templates
`normal`	Yes	Path to regular weight font file (relative to template root)
`bold`	Optional	Path to bold weight font file
`italic`	Optional	Path to italic style font file
`boldItalic`	Optional	Path to bold italic font file
`pdfEncoding`	Yes	Must be “Identity-H” for PDF/A compliance
`pdfEmbedded`	Yes	Must be “true” for PDF/A compliance

Template Package Structure

Organize your template package with the following structure:



The REPORTS_DIR Parameter

The `REPORTS_DIR` parameter is **critical** for properly referencing resources within your template package. The service automatically injects this parameter with the absolute path to your extracted template directory at runtime.

Why It Matters:

- Templates are extracted to a cache directory during document generation
- All relative paths (images, subreports, resources) must be resolved from this location
- The service sets `REPORTS_DIR` automatically — you don't need to provide it in the request

Declaring REPORTS_DIR in Your Template:

```
<parameter name="REPORTS_DIR" class="java.lang.String">
    <defaultValueExpression><![CDATA[""]]></defaultValueExpression>
</parameter>
```

Using REPORTS_DIR for Resources:

```
<!-- Image from resources folder -->
<element kind="image">
    <expression><![CDATA[$P{REPORTS_DIR} + "resources/logo.png"]]></expression>
</element>

<!-- Subreport reference -->
<element kind="subreport">
    <expression><![CDATA[$P{REPORTS_DIR} + "subreports/line-items.jasper"]]></expression>
</element>
```

Organizing Resources

The `resources/` folder (or any custom-named folder like `images/`, `assets/`) stores static files used by the template: logos, watermarks, background images, icons, signatures, etc.

Key Points:

- **Folder Name**: Customizable — use `resources/`, `images/`, `assets/`, or any convention
- **Supported Formats**: PNG, JPG, JPEG, GIF, SVG (depending on output format)
- **Path Resolution**: Always use `$P{REPORTS_DIR}` prefix for reliable path resolution
- **Nested Structure**: You can create subfolders for better organization

Example Resource References:

```
<!-- Logo in header -->
<element kind="image" x="0" y="0" width="150" height="50">
    <expression><![CDATA[$P{REPORTS_DIR} + "resources/company-logo.png"]]></expression>
</element>

<!-- Conditional watermark -->
<element kind="image" x="100" y="200" width="400" height="400">
    <expression><![CDATA[$P{isDraft} ? $P{REPORTS_DIR} + "resources/draft-watermark.png" :
↪  null]]></expression>
</element>
```

Organizing Subreports

When your template uses subreports, place them in a dedicated subfolder within the package. The folder name is customizable—common conventions include `subreports/`, `includes/`, or `components/`.

Key Points:

- **Folder Location**: Subreport templates must be placed below the main template in the directory structure (e.g., in a `subreports/` folder)
- **Custom Folder Names**: You can name the subreports folder according to your project conventions
- **Relative Paths**: Reference subreports using relative paths from the main template location
- **Multiple Levels**: You can create nested folder structures for complex template hierarchies

Subreport Expression Example:

In your main template, reference subreports using `$P{REPORTS_DIR}`:

```
<element kind="subreport">
    <expression><![CDATA[$P{REPORTS_DIR} + "subreports/line-items.jasper"]]></expression>
    <!-- or use a parameter for dynamic subreport selection -->
    <expression><![CDATA[$P{REPORTS_DIR} + "subreports/" + $P{subreportName} + ".jasper"]]></expression>
</element>
```

Important: The subreport expression must reference the compiled `.jasper` file (not `.jrxml`). You only need to include the source `.jrxml` files in your package—the service handles compilation automatically.

Using Fonts in JRXML Templates

JasperReports 7.0.x Syntax

Use the `fontName` attribute on text elements:

```
<element kind="staticText" uuid="..." x="0" y="0" width="200" height="30"
         fontSize="14.0" bold="true" fontName="DejaVu Sans">
    <text><![CDATA[Report Title]]></text>
</element>

<element kind="textField" uuid="..." x="0" y="40" width="300" height="20"
         fontSize="10.0" fontName="Noto Serif">
    <expression><![CDATA[$F{customerName}]]></expression>
</element>
```

Important Notes:

- In JasperReports 7.x, you **do not** need to manually specify `pdfFontName`, `pdfEncoding`, or `isPdfEmbedded` in your template JRXML

- The font injection system automatically adds these attributes during report compilation based on the manifest
- Simply use the `fontName` attribute matching the name in your manifest, and the system will inject the necessary PDF embedding attributes before compilation

Legacy JRXML Syntax (Pre-7.0)

If working with older JRXML files, remove PDF-specific attributes:

```
<!-- Old style - NOT needed in 7.x -->
<font fontName="DejaVu Sans" size="12" pdfFontName="fonts/DejaVuSans.ttf"
      pdfEncoding="Identity-H" isPdfEmbedded="true"/>

<!-- New style - Correct for 7.x -->
<element kind="textField" fontName="DejaVu Sans" fontSize="12.0">
    <expression><![CDATA[$F{fieldName}]]></expression>
</element>
```

PDF/A Conformance Levels

The service supports the following PDF/A conformance levels:

PDF/A-1b (ISO 19005-1:2005, Level B)

- **Use Case**: Basic archival requirements
- **Features**:
  - Device-independent appearance
  - Embedded fonts required
  - Basic metadata
- **Generation**: Set `pdfaConformance` to “PDF/A-1b”

Request Example with PDF/A

```
{
  "documentId": "550e8400-e29b-41d4-a716-446655440000",
  "format": "PDF",
  "parameters": [
    {"name": "reportTitle", "value": "Annual Financial Report"},
    {"name": "reportDate", "value": "2025-11-01"}
  ],
  "data": {
    "transactions": [
      {"date": "2025-01-15", "description": "Payment", "amount": 1500.00},
      {"date": "2025-02-20", "description": "Invoice", "amount": 2300.00}
    ]
  },
  "pdfExportOptions": {
    "pdfaConformance": "PDF/A-1b",
    "iccProfile": "sRGB"
  }
}
```

Request Example with Font Embedding Substitution

If the generated PDF contains non-embedded base-14 fonts (Helvetica, Courier, Times), use `fontEmbeddingSubstitute` to replace them with embedded TrueType fonts:

```
POST /api/v1/templates/{id}/generate/pdf
```

```
{
  "parameters": [
    {"name": "reportTitle", "value": "Annual Financial Report"}
  ],
  "pdfExportOptions": {
    "fontEmbeddingSubstitute": "DejaVu Sans"
  }
}
```

> **Note:** JasperReports typically handles font embedding natively via Font Extensions. This parameter is most useful for the DOCX→PDF pipeline where Apache FOP may inject non-embedded base-14 fonts. It can also be combined with `pdfaConformance` to ensure full PDF/A compliance.

Request Example with Image Compression

If the generated PDF contains large lossless-encoded images (e.g., embedded photos), use `imageCompressionQuality` to re-compress them as JPEG for smaller file sizes:

```
POST /api/v1/templates/{id}/generate/pdf
```

```
{
  "parameters": [
    {"name": "reportTitle", "value": "Annual Financial Report"}
  ],
  "pdfExportOptions": {
    "imageCompressionQuality": 0.85
  }
}
```

> **Note:** Quality ranges from 0.0 to 1.0 — **0.85** is recommended for print, **0.75** for mass printing. Only large colour RGB images are re-compressed; small icons, masks, and already-JPEG images are left untouched. Can be combined with `fontEmbeddingSubstitute` and `pdfaConformance`.

Request Example with CMYK Conversion

For print-shop delivery requiring CMYK colour space, use `cmykConversionProfile` to convert RGB raster images to DeviceCMYK:

```
POST /api/v1/templates/{id}/generate/pdf
```

```
{
  "parameters": [
    {"name": "reportTitle", "value": "Annual Financial Report"}
  ],
  "pdfExportOptions": {
    "cmykConversionProfile": "coated-fogra39"
  }
}
```

Note: Available profiles: **coated-fogra39** (ISO Coated v2, general commercial print), **pso-coated-v3** (PSO Coated v3, modern offset), **pso-uncoated-v3** (PSO Uncoated v3, uncoated paper). Only raster images are converted — vector graphics and text remain unchanged.

Request Example with Transparency Flattening

To speed up print-shop RIP processing, use `flattenTransparency` to remove redundant transparency groups from pages:

```
POST /api/v1/templates/{id}/generate/pdf
```

```
{
  "parameters": [
    {"name": "reportTitle", "value": "Annual Financial Report"}
  ],
  "pdfExportOptions": {
    "flattenTransparency": true
  }
}
```

Note: When enabled, a two-step process is applied: (1) images with `/SMask` (alpha channels from PNG sources) are composited onto a white background and the soft mask is removed; (2) pages that declare `/Group /Transparency` but use no real transparency (no remaining SMask, no BlendMode, no alpha) have the entry removed. The result is a fully opaque PDF optimised for RIP processing. No visible effect on output for documents with white/light backgrounds. Can be combined with all other PDF export options.

Request Example with Full Print-Shop Optimisation

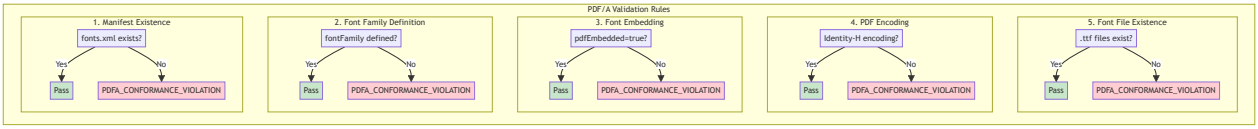
Combine all print-optimisation features for maximum RIP efficiency:

```
POST /api/v1/templates/{id}/generate/pdf
```

```
{
  "parameters": [
    {"name": "reportTitle", "value": "Annual Financial Report"}
  ],
  "pdfExportOptions": {
    "fontEmbeddingSubstitute": "DejaVu Sans",
    "imageCompressionQuality": 0.85,
    "cmykConversionProfile": "coated-fogra39",
    "flattenTransparency": true
  }
}
```

Validation Rules

The service automatically validates the following before PDF generation:



PDF/A Error Handling

Conformance Violation Error

When PDF/A validation fails, the service returns a detailed error:

```
{
  "meta": {
    "timestamp": "2025-11-01T12:00:00Z",
    "correlationId": "abc-123-def-456",
    "status": "ERROR"
  },
  "data": null,
  "errors": [
    {
      "code": "PDFA_CONFORMANCE_VIOLATION",
      "message": "PDF/A conformance validation failed: 2 fonts without proper encoding found",
      "details": "The following fonts do not use Identity-H encoding required for PDF/A: [DejaVu Sans,
      ↪  Noto Serif]"
    }
  ]
}
```

Common Error Scenarios

Scenario	Error Code	Solution
Missing manifest	PDFA_CONFORMANCE_VIOLATION	Add fonts.xml to template root
No fonts defined	PDFA_CONFORMANCE_VIOLATION	Define at least one fontFamily in manifest
Font not embedded	PDFA_CONFORMANCE_VIOLATION	Set pdfEmbedded="true" for all fonts
Wrong encoding	PDFA_CONFORMANCE_VIOLATION	Use pdfEncoding="Identity-H" for all fonts
Missing font file	PDFA_CONFORMANCE_VIOLATION	Include all referenced .ttf files in ZIP

Checking Server-Supported Fonts

Before packaging custom fonts with your template, check which fonts are already available on the server. Fonts pre-installed on the server don't need to be included in your template package.

Using Muban CLI:

```
# List all fonts available on the server
muban fonts
```

Example output:

```
Fonts (18 total):

Name                    | Faces                           | PDF Embedded
------------------------------------------------------------------------
DejaVu Sans             | normal, bold, italic, boldItalic | Yes
DejaVu Sans Condensed   | normal, bold, italic, boldItalic | Yes
DejaVu Sans Mono        | normal, bold, italic, boldItalic | Yes
DejaVu Serif            | normal, bold, italic, boldItalic | Yes
Fira Sans               | normal, bold, italic, boldItalic | Yes
Lato                    | normal, bold, italic, boldItalic | Yes
Poppins                 | normal, bold, italic, boldItalic | Yes
```

```
SansSerif              |                              | No
Serif                  |                              | No
```

When to package fonts:

Scenario	Action
Font listed by `muban fonts`	Use directly in template, no packaging needed
Font NOT listed	Include in template ZIP with `fonts.xml` manifest
Need specific variant (e.g., Light, SemiBold)	Package only if that variant is missing on server

Tip: Using server-provided fonts reduces template package size and ensures consistent rendering across all templates.

Font Selection Guidelines

Recommended Fonts for PDF/A

Open Source Fonts (Royalty-free, suitable for commercial use):

1. **DejaVu Font Family** - Comprehensive Unicode coverage
   - DejaVu Sans (sans-serif)
   - DejaVu Serif (serif)
   - DejaVu Sans Mono (monospace)
   - Download: https://dejavu-fonts.github.io/
2. **Noto Font Family** - Google's universal font family
   - Noto Sans (sans-serif)
   - Noto Serif (serif)
   - Noto Sans Mono (monospace)
   - Download: https://fonts.google.com/noto
3. **Liberation Font Family** - Metrically compatible with major proprietary fonts
   - Liberation Sans (Arial compatible)
   - Liberation Serif (Times New Roman compatible)
   - Liberation Mono (Courier New compatible)
   - Download: https://github.com/liberationfonts/liberation-fonts
4. **Source Font Family** - Adobe's open source fonts
   - Source Sans Pro (sans-serif)
   - Source Serif Pro (serif)
   - Source Code Pro (monospace)
   - Download: https://adobe-fonts.github.io/source-sans/

Font Licensing Considerations

Status	License Type	Examples
Allowed	Open source fonts (OFL, MIT, Apache licenses)	DejaVu, Noto, Liberation
Allowed	Fonts with commercial use permission	Adobe Source fonts
Allowed	Custom corporate fonts with proper licensing	Internal company fonts
Not Allowed	System fonts without redistribution rights	Windows/macOS system fonts

Status	License Type	Examples
Not Allowed	Proprietary fonts without embedding license	Commercial fonts without license
Not Allowed	Trial/demo fonts with restrictions	Evaluation fonts

Testing Font Embedding

Preview in Jaspersoft Studio

1. Open your template in Jaspersoft Studio
2. Use the **Preview** tab with sample data
3. Verify fonts render correctly
4. Check PDF output for proper embedding

Service Testing Workflow

```
1. Create template with custom fonts
2. Package with fonts.xml and .ttf files
3. Upload to service
4. Generate test PDF with PDF/A-1b conformance
5. Validate output in PDF reader
6. Check font embedding properties in PDF metadata
```

Verification Tools

Adobe Acrobat:

- File -> Properties -> Fonts tab
- All fonts should show “(Embedded Subset)”

PDF-XChange Viewer:

- File -> Document Properties -> Fonts
- Verify all fonts are embedded

pdffonts command (Linux/Mac):

```
pdffonts generated-report.pdf
# Output should show "emb" column as "yes" for all fonts
```

Migration from External Configuration

If you previously used the external jasperreports.properties configuration:

Old Approach (External Configuration)

```
# jasperreports.properties
⌋
 ↪  net.sf.jasperreports.extension.registry.factory.fonts=net.sf.jasperreports.engine.fonts.SimpleFontExtensionRegis
net.sf.jasperreports.extension.simple.font.families.MyFont=fonts/MyFont.ttf
net.sf.jasperreports.extension.simple.font.families.MyFont.bold=fonts/MyFont-Bold.ttf
```

New Approach (Manifest-Based)

```
<!-- fonts.xml in template package -->
<fontFamilies>
    <fontFamily name="MyFont">
        <normal>fonts/MyFont.ttf</normal>
        <bold>fonts/MyFont-Bold.ttf</bold>
        <pdfEncoding>Identity-H</pdfEncoding>
        <pdfEmbedded>true</pdfEmbedded>
    </fontFamily>
</fontFamilies>
```

Benefits of Manifest Approach:

- Template self-contained (no server configuration needed)
- Version control friendly (fonts travel with template)
- Multiple templates can use different fonts independently
- Automatic PDF/A validation
- Easier deployment and testing

Best Practices for Font Embedding

1. **Always Include All Font Weights**

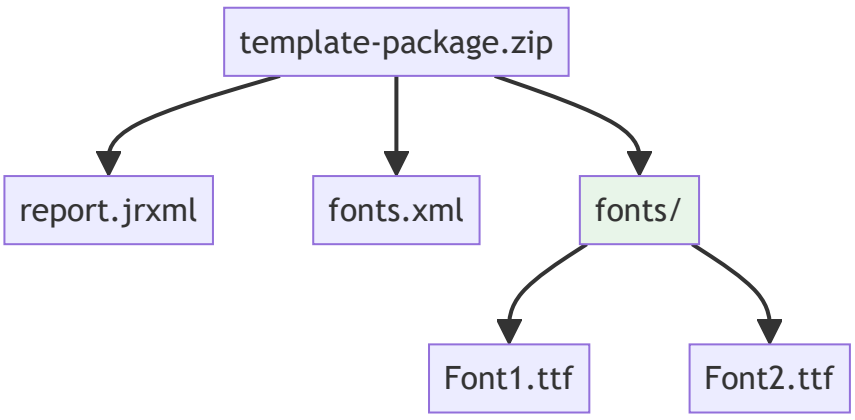
   ```
   <!-- Include normal, bold, italic, and boldItalic when available -->
   <fontFamily name="DejaVu Sans">
       <normal>fonts/DejaVuSans.ttf</normal>
       <bold>fonts/DejaVuSans-Bold.ttf</bold>
       <italic>fonts/DejaVuSans-Oblique.ttf</italic>
       <boldItalic>fonts/DejaVuSans-BoldOblique.ttf</boldItalic>
       <!-- ... -->
   </fontFamily>
   ```

2. **Use Consistent Font Names**

   ```
   <!-- Match fontName in JRXML with manifest name exactly -->
   <fontFamily name="DejaVu Sans">  <!-- Exact match required -->
   ```

3. **Organize Font Files**

   Keep fonts in a dedicated subdirectory:



4. **Test with PDF/A Validation Early**

```
Always test PDF/A generation during development, not just before production
```

5. **Document Font Sources**

```
<!-- Add comments in manifest for maintenance -->
<!-- DejaVu Sans - Downloaded from https://dejavu-fonts.github.io/ -->
<!-- License: DejaVu Fonts License (Free) -->
<fontFamily name="DejaVu Sans">
    ...
</fontFamily>
```

Troubleshooting Font Issues

Font Not Found in PDF

Problem: PDF generates but uses fallback font

Step	Check	Action
1	fontName match	Verify fontName in JRXML matches manifest exactly
2	Manifest location	Check fonts.xml is in template root
3	Font files exist	Ensure font files (.ttf) exist in ZIP package
4	Service logs	Review service logs for font loading errors

PDF/A Validation Fails

Problem: Error “PDFA_CONFORMANCE_VIOLATION” returned

Step	Check	Action
1	Embedding	Verify all fonts have `pdfEmbedded="true"`
2	Encoding	Ensure all fonts use `pdfEncoding="Identity-H"`
3	File paths	Check all font file paths are correct
4	XML validity	Confirm fonts.xml is well-formed XML

Mixed Font Rendering

```
Problem: Some text uses correct font, others use fallback

Solutions:
1. Ensure all required font weights are included (bold, italic)
2. Verify font family has all variants referenced in JRXML
3. Check for typos in fontName attributes
```

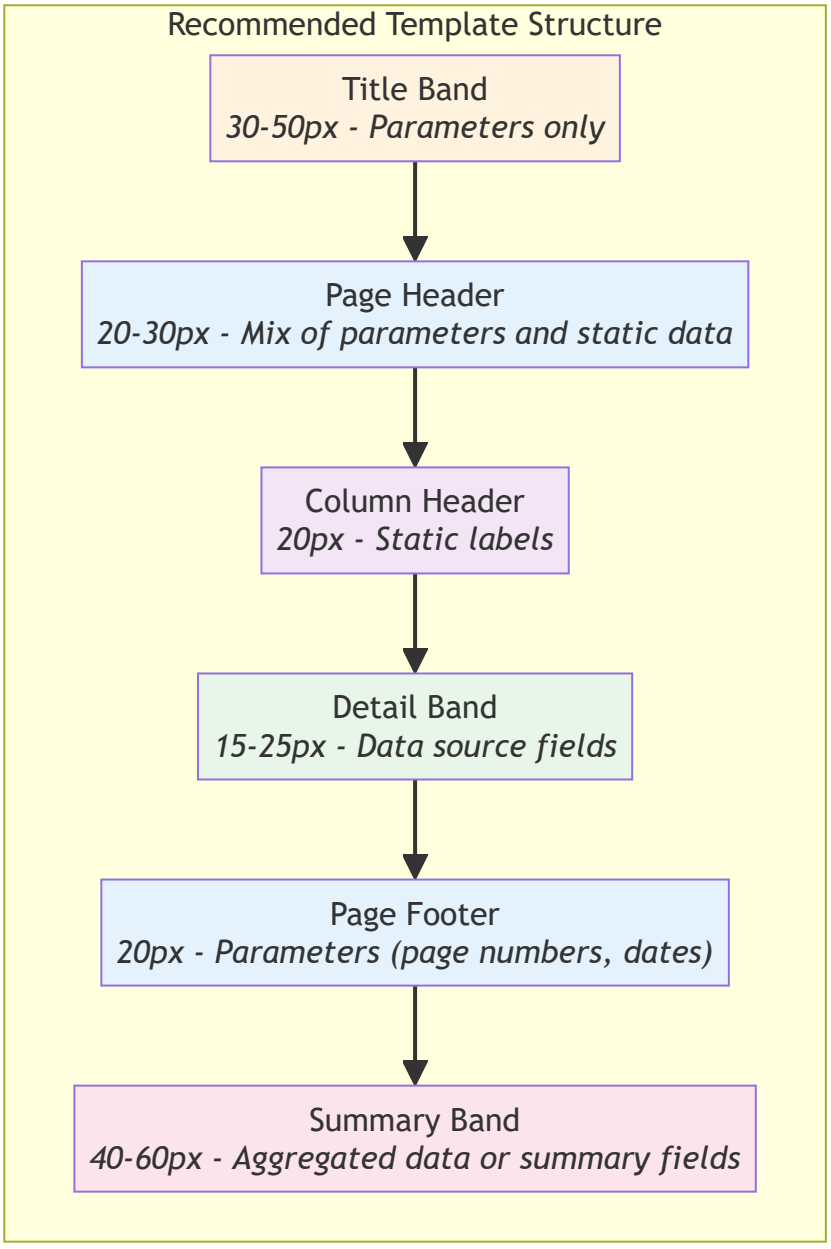
Large PDF File Size

```
Problem: PDF file size larger than expected

Solutions:
1. Use font subsets (automatic in JasperReports 7.x)
2. Limit number of font families (2-3 is usually sufficient)
3. Consider font file size when selecting fonts
4. DejaVu Sans: ~300KB, Noto Sans: ~500KB per file
```

Best Practices

Best Practices for Template Design

Structure Organization



Field Naming Conventions

```
// Use clear, consistent field names
Good:
{
  "customerName": "ACME Corp",
  "orderDate": "2025-10-07",
  "lineItems": [
    {"productName": "Widget A", "unitPrice": 25.00}
  ]
```

```
}

Avoid:
{
  "cust": "ACME Corp",
  "dt": "2025-10-07",
  "items": [
    {"prod": "Widget A", "price": 25.00}
  ]
}
```

Performance Optimization

Efficient Data Structures

```
// Minimize nesting depth
Efficient:
{
  "orders": [
    {"id": "ORD-001", "amount": 1500, "customerName": "ACME"}
  ]
}

Inefficient:
{
  "orders": [
    {
      "id": "ORD-001",
      "details": {
        "financial": {
          "amount": 1500
        },
        "customer": {
          "info": {
            "name": "ACME"
          }
        }
      }
    }
  ]
}
```

Array Size Considerations

- Limit arrays to reasonable sizes (< 1000 items for PDF, < 10000 for XLSX)
- Consider pagination for large datasets
- Use summary data instead of detail when possible

Error Handling

Null-Safe Expressions

```
<!-- Always handle potential null values -->
<textField>
    <reportElement x="0" y="0" width="100" height="20"/>
    <textFieldExpression><![CDATA[
```

```
        ($F{optionalField} != null) ? $F{optionalField}.toString() : ""
    ]]></textFieldExpression>
</textField>

<!-- Safe numeric operations -->
<textField pattern="#,##0.00">
    <reportElement x="100" y="0" width="80" height="20"/>
    <textFieldExpression><![CDATA[
        (($F{quantity} != null) && ($F{price} != null))
            ? $F{quantity}.doubleValue() * $F{price}.doubleValue()
            : 0.0
    ]]></textFieldExpression>
</textField>
```

Parameter Validation

```
<!-- Validate parameter values -->
<textField>
    <reportElement x="0" y="0" width="200" height="20"/>
    <textFieldExpression><![CDATA[
        ($P{title} != null && !$P{title}.trim().isEmpty())
            ? $P{title}
            : "Untitled Report"
    ]]></textFieldExpression>
</textField>
```

Formatting Standards

Consistent Number Formatting

```
<!-- Currency formatting -->
<textField pattern="¤#,##0.00">
    <reportElement x="0" y="0" width="100" height="20"/>
    <textElement textAlignment="Right"/>
    <textFieldExpression><![CDATA[$F{amount}]]></textFieldExpression>
</textField>

<!-- Percentage formatting -->
<textField pattern="#,##0.0%">
    <reportElement x="100" y="0" width="80" height="20"/>
    <textElement textAlignment="Right"/>
    <textFieldExpression><![CDATA[$F{rate}]]></textFieldExpression>
</textField>

<!-- Integer formatting -->
<textField pattern="#,##0">
    <reportElement x="180" y="0" width="60" height="20"/>
    <textElement textAlignment="Right"/>
    <textFieldExpression><![CDATA[$F{quantity}]]></textFieldExpression>
</textField>
```

Date Formatting

```
<!-- Standard date formats -->
<textField pattern="MM/dd/yyyy">
    <reportElement x="0" y="0" width="100" height="20"/>
```

```
    <textFieldExpression><![CDATA[$P{reportDate}]]></textFieldExpression>
</textField>

<textField pattern="MMMMM dd, yyyy">
    <reportElement x="100" y="0" width="150" height="20"/>
    <textFieldExpression><![CDATA[$P{reportDate}]]></textFieldExpression>
</textField>

<textField pattern="yyyy-MM-dd HH:mm:ss">
    <reportElement x="250" y="0" width="120" height="20"/>
    <textFieldExpression><![CDATA[$F{timestamp}]]></textFieldExpression>
</textField>
```

Accessibility and Usability

Clear Visual Hierarchy

```
<!-- Use consistent fonts and sizes -->
<textField>
    <reportElement x="0" y="0" width="200" height="25"/>
    <textElement>
        <font fontName="Arial" size="18" isBold="true"/>
    </textElement>
    <textFieldExpression><![CDATA[$P{reportTitle}]]></textFieldExpression>
</textField>

<!-- Consistent spacing -->
<textField>
    <reportElement x="0" y="30" width="200" height="20"/>
    <textElement>
        <font fontName="Arial" size="12"/>
    </textElement>
    <textFieldExpression><![CDATA[$P{subtitle}]]></textFieldExpression>
</textField>
```

Color and Contrast

```
<!-- Use appropriate colors for different data types -->
<textField>
    <reportElement x="0" y="0" width="100" height="20">
        <conditionalStyle>
            <conditionExpression><![CDATA[$F{status}.equals("ERROR")]]></conditionExpression>
            <style forecolor="#CC0000" isBold="true"/>
        </conditionalStyle>
        <conditionalStyle>
            <conditionExpression><![CDATA[$F{status}.equals("SUCCESS")]]></conditionExpression>
            <style forecolor="#00AA00"/>
        </conditionalStyle>
    </reportElement>
    <textFieldExpression><![CDATA[$F{status}]]></textFieldExpression>
</textField>
```

Troubleshooting Common Issues

Common Issues and Solutions

Field Access Problems

Problem: `Field 'fieldName' not found`

Cause	Solution
Field name mismatch	Verify data structure matches field names exactly
Case sensitivity issue	Check for case sensitivity (JSON is case-sensitive)
Typo in field expression	Use debug expressions to print available fields
Data structure not converted	Test with simple data structure first

Debug Template:

```
<textField>
    <reportElement x="0" y="0" width="400" height="20"/>
    <textFieldExpression><![CDATA[
        "Available fields: " + $F{fieldName} + " (check console for full list)"
    ]]></textFieldExpression>
</textField>
```

Empty Document Issues

Problem: Document generates but shows no data

Cause	Solution
Empty data array	Verify data array has elements
Detail band height is 0	Set detail band height > 0
Incorrect field references	Check field expressions
Data source not properly created	Test with simple array structure

Test Data:

```
{
  "data": {
    "testItems": [
      {"name": "Test Item 1", "value": "Test Value 1"},
      {"name": "Test Item 2", "value": "Test Value 2"}
    ]
  }
}
```

Parameter Type Conversion Errors

Problem: `Cannot convert 'value' to ExpectedType`

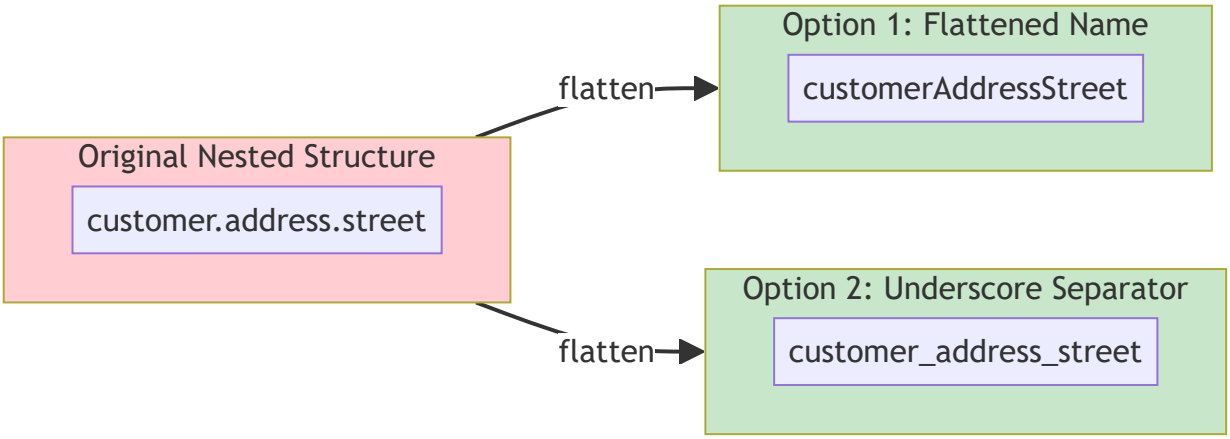
```
Common Type Issues:
- String "123" to Integer: Automatic conversion
- String "abc" to Integer: Error
- String "true" to Boolean: Automatic conversion
- String "2025-10-07" to Date: Automatic conversion
```

```
Solutions:
1. Ensure parameter values are convertible
2. Use default values in parameter definitions
3. Add validation in client applications
```

Complex Data Structure Issues

Problem: Nested objects not accessible

The service flattens complex nested structures into simple fields. Use one of these approaches:



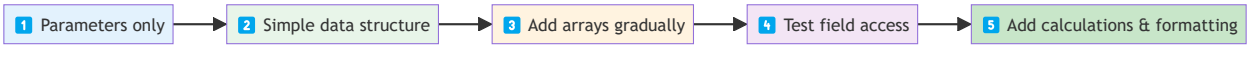
Performance Issues

Problem: Report generation is slow

Cause	Solution
Large data arrays (>1000 items)	Limit array sizes
Complex calculations in expressions	Pre-calculate complex values in client
Multiple subreports	Reduce subreport count or combine
Inefficient data structures	Optimize field expressions
Large datasets	Consider pagination

Debugging Strategies

Progressive Testing



Use Debug Fields

```
<!-- Add temporary debug fields -->
<textField>
    <reportElement x="0" y="100" width="400" height="60"/>
    <textElement>
        <font size="8"/>
    </textElement>
```

```
    <textFieldExpression><![CDATA[
        "Debug Info:\n" +
        "Field1: " + $F{field1} + "\n" +
        "Field2: " + $F{field2} + "\n" +
        "Parameter1: " + $P{param1}
    ]]></textFieldExpression>
</textField>
```

Service Testing

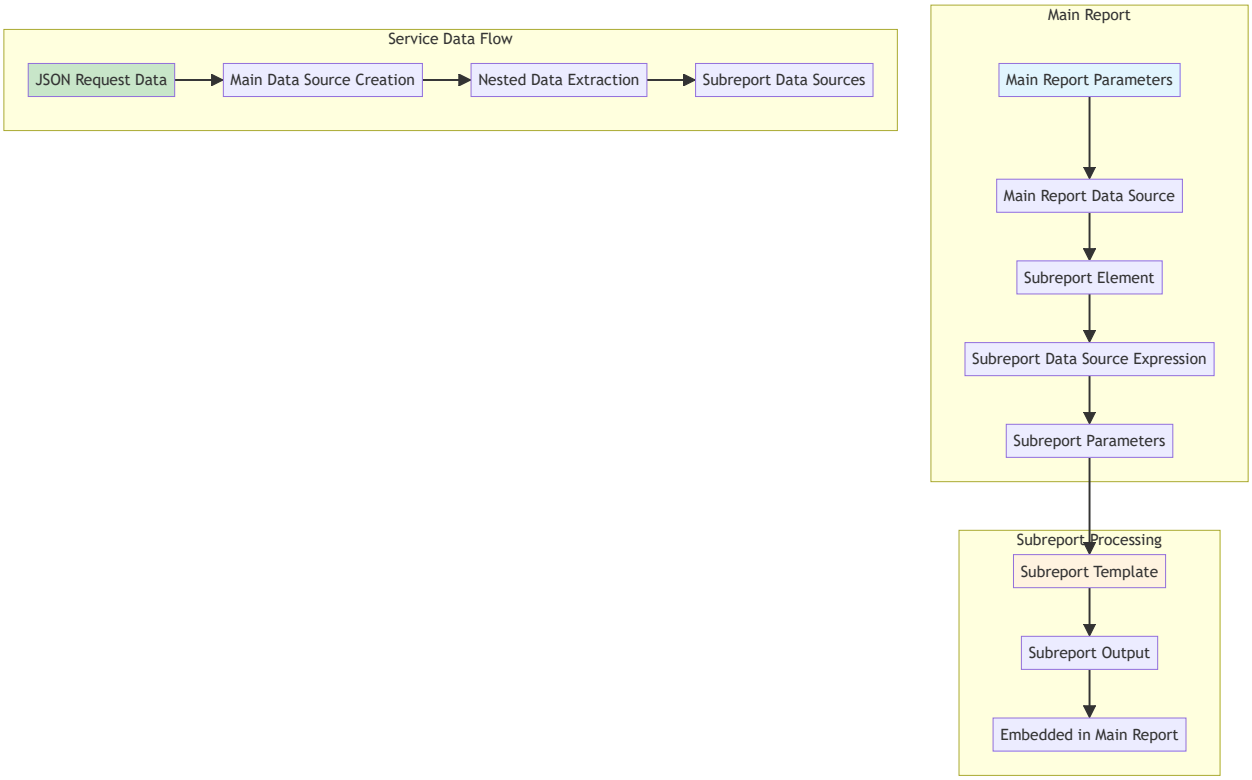
```
# Test with minimal data
curl -X POST "http://localhost:8080/api/v1/templates/{id}/pdf" \
  -H "Content-Type: application/json" \
  -d '{
    "parameters": [{"name": "test", "value": "working"}],
    "data": {"items": [{"name": "test1"}]}
  }' \
  --output debug.pdf

# Gradually increase complexity
```

Subreport Integration with Complex Data

Understanding Subreports in the Service Context

Subreports allow you to create modular, reusable report components that can handle complex nested data structures. The Document Generation Service supports subreports by passing appropriate data subsets to each subreport component.

Subreport Architecture Overview



Data Structure for Subreports

When designing reports with subreports, structure your JSON data to support hierarchical processing:

Example: Order Document with Line Items Subreport

Service Request Structure:

```
{
  "parameters": [
    {"name": "reportTitle", "value": "Order Summary"},
    {"name": "reportDate", "value": "2025-10-12"}
  ],
  "data": {
    "orders": [
      {
        "orderId": "ORD-001",
        "customerName": "ACME Corp",
        "orderDate": "2025-10-01",
        "total": 1500.00,
        "lineItems": [
```

```
          {
            "product": "Widget A",
            "quantity": 10,
            "unitPrice": 25.00,
            "total": 250.00
          },
          {
            "product": "Widget B",
            "quantity": 25,
            "unitPrice": 50.00,
            "total": 1250.00
          }
        ],
        "customerDetails": {
          "address": "123 Business St",
          "phone": "+1-555-0123",
          "email": "orders@acme.com"
        }
      },
      {
        "orderId": "ORD-002",
        "customerName": "TechCorp Ltd",
        "orderDate": "2025-10-02",
        "total": 750.00,
        "lineItems": [
          {
            "product": "Service Package",
            "quantity": 1,
            "unitPrice": 750.00,
            "total": 750.00
          }
        ],
        "customerDetails": {
          "address": "456 Tech Ave",
          "phone": "+1-555-0456",
          "email": "billing@techcorp.com"
        }
      }
    ]
  }
}
```

Creating Subreport Templates

Step 1: Design the Subreport Template

Create a separate JRXML file for the subreport (e.g., `LineItemsSubreport.jrxml`):

```
<?xml version="1.0" encoding="UTF-8"?>
<jasperReport xmlns="http://jasperreports.sourceforge.net/jasperreports"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              name="LineItemsSubreport"
              pageWidth="500" pageHeight="842"
              columnWidth="500" leftMargin="0" rightMargin="0"
              topMargin="0" bottomMargin="0">

    <!-- Subreport Parameters -->
```

```
    <parameter name="orderId" class="java.lang.String"/>
    <parameter name="customerName" class="java.lang.String"/>

    <!-- Subreport Fields (from lineItems array) -->
    <field name="product" class="java.lang.String"/>
    <field name="quantity" class="java.lang.Integer"/>
    <field name="unitPrice" class="java.math.BigDecimal"/>
    <field name="total" class="java.math.BigDecimal"/>

    <!-- Column Header -->
    <columnHeader>
        <band height="25">
            <staticText>
                <reportElement x="0" y="0" width="200" height="20"/>
                <box>
                    <bottomPen lineWidth="1.0"/>
                </box>
                <textElement>
                    <font isBold="true"/>
                </textElement>
                <text><![CDATA[Product]]></text>
            </staticText>

            <staticText>
                <reportElement x="200" y="0" width="60" height="20"/>
                <box>
                    <bottomPen lineWidth="1.0"/>
                </box>
                <textElement textAlignment="Center">
                    <font isBold="true"/>
                </textElement>
                <text><![CDATA[Qty]]></text>
            </staticText>

            <staticText>
                <reportElement x="260" y="0" width="120" height="20"/>
                <box>
                    <bottomPen lineWidth="1.0"/>
                </box>
                <textElement textAlignment="Right">
                    <font isBold="true"/>
                </textElement>
                <text><![CDATA[Unit Price]]></text>
            </staticText>

            <staticText>
                <reportElement x="380" y="0" width="120" height="20"/>
                <box>
                    <bottomPen lineWidth="1.0"/>
                </box>
                <textElement textAlignment="Right">
                    <font isBold="true"/>
                </textElement>
                <text><![CDATA[Total]]></text>
            </staticText>
        </band>
    </columnHeader>
```

```
    <!-- Detail Band -->
    <detail>
        <band height="20">
            <textField>
                <reportElement x="0" y="0" width="200" height="18"/>
                <box>
                    <pen lineWidth="0.5" lineColor="#CCCCCC"/>
                </box>
                <textElement>
                    <font size="10"/>
                </textElement>
                <textFieldExpression><![CDATA[$F{product}]]></textFieldExpression>
            </textField>

            <textField>
                <reportElement x="200" y="0" width="60" height="18"/>
                <box>
                    <pen lineWidth="0.5" lineColor="#CCCCCC"/>
                </box>
                <textElement textAlignment="Center">
                    <font size="10"/>
                </textElement>
                <textFieldExpression><![CDATA[$F{quantity}]]></textFieldExpression>
            </textField>

            <textField pattern="#,##0.00">
                <reportElement x="260" y="0" width="120" height="18"/>
                <box>
                    <pen lineWidth="0.5" lineColor="#CCCCCC"/>
                </box>
                <textElement textAlignment="Right">
                    <font size="10"/>
                </textElement>
                <textFieldExpression><![CDATA[$F{unitPrice}]]></textFieldExpression>
            </textField>

            <textField pattern="#,##0.00">
                <reportElement x="380" y="0" width="120" height="18"/>
                <box>
                    <pen lineWidth="0.5" lineColor="#CCCCCC"/>
                </box>
                <textElement textAlignment="Right">
                    <font size="10"/>
                </textElement>
                <textFieldExpression><![CDATA[$F{total}]]></textFieldExpression>
            </textField>
        </band>
    </detail>
</jasperReport>
```

Step 2: Configure Main Report to Use Subreport

In your main report template, add a subreport element:

```
<detail>
    <band height="200">
```

```
        <!-- Order Header Information -->
        <textField>
            <reportElement x="0" y="0" width="150" height="20"/>
            <textElement>
                <font size="12" isBold="true"/>
            </textElement>
            <textFieldExpression><![CDATA["Order: " + $F{orderId}]]></textFieldExpression>
        </textField>

        <textField>
            <reportElement x="200" y="0" width="200" height="20"/>
            <textElement>
                <font size="12"/>
            </textElement>
            <textFieldExpression><![CDATA[$F{customerName}]]></textFieldExpression>
        </textField>

        <textField pattern="MM/dd/yyyy">
            <reportElement x="400" y="0" width="100" height="20"/>
            <textElement>
                <font size="10"/>
            </textElement>
            <textFieldExpression><![CDATA[$F{orderDate}]]></textFieldExpression>
        </textField>

        <!-- Subreport Element -->
        <subreport>
            <reportElement x="0" y="30" width="500" height="100"/>

            <!-- Pass parameters to subreport -->
            <subreportParameter name="orderId">
                <subreportParameterExpression><![CDATA[$F{orderId}]]></subreportParameterExpression>
            </subreportParameter>

            <subreportParameter name="customerName">
                <subreportParameterExpression><![CDATA[$F{customerName}]]></subreportParameterExpression>
            </subreportParameter>

            <!-- Data source for subreport -->
            <dataSourceExpression><![CDATA[
                new net.sf.jasperreports.engine.data.JRBeanCollectionDataSource(
                    (java.util.Collection) $F{lineItems}
                )
            ]]></dataSourceExpression>

            <!-- Subreport expression -->
            <subreportExpression><![CDATA["subreports/LineItemsSubreport.jasper"]]></subreportExpression>
        </subreport>

        <!-- Order Total -->
        <textField pattern="$#,##0.00">
            <reportElement x="400" y="140" width="100" height="20"/>
            <textElement textAlignment="Right">
                <font size="12" isBold="true"/>
            </textElement>
            <textFieldExpression><![CDATA[$F{total}]]></textFieldExpression>
        </textField>
```

```
    </band>
</detail>
```

Advanced Subreport Patterns

Pattern 1: Nested Subreports (Master-Detail-Detail)

For complex hierarchical data like Orders -> Line Items -> Item Components:

```
{
  "data": {
    "orders": [
      {
        "orderId": "ORD-001",
        "lineItems": [
          {
            "product": "Assembly Kit",
            "components": [
              {"part": "Screw A", "quantity": 10},
              {"part": "Bracket B", "quantity": 2}
            ]
          }
        ]
      }
    ]
  }
}
```

Implementation:

1. **Main Report**: Iterates through orders
2. **Line Items Subreport**: Shows line items, includes components subreport
3. **Components Subreport**: Shows individual components

Pattern 2: Conditional Subreports

Show different subreports based on data conditions:

```
<subreport>
    <reportElement x="0" y="50" width="500" height="100">
        <!-- Only show if order has line items -->
        <printWhenExpression><![CDATA[
            $F{lineItems} != null &&
            ((java.util.List)$F{lineItems}).size() > 0
        ]]></printWhenExpression>
    </reportElement>

    <dataSourceExpression><![CDATA[
        new net.sf.jasperreports.engine.data.JRBeanCollectionDataSource(
            (java.util.Collection) $F{lineItems}
        )
    ]]></dataSourceExpression>

    <subreportExpression><![CDATA["subreports/LineItemsSubreport.jasper"]]></subreportExpression>
</subreport>
```

Pattern 3: Multiple Subreports per Record

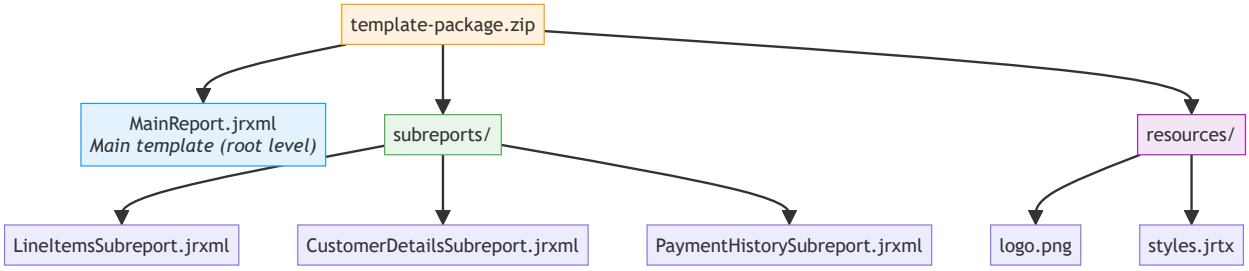
Display different aspects of complex data using multiple subreports:

```
<detail>
    <band height="400">
        <!-- Customer Info Subreport -->
        <subreport>
            <reportElement x="0" y="0" width="250" height="150"/>
            <subreportParameter name="customerId">
                <subreportParameterExpression><![CDATA[$F{customerId}]]></subreportParameterExpression>
            </subreportParameter>
            <dataSourceExpression><![CDATA[
                new net.sf.jasperreports.engine.data.JRBeanCollectionDataSource(
                    java.util.Arrays.asList($F{customerDetails})
                )
            ]]></dataSourceExpression>

↪   <subreportExpression><![CDATA["subreports/CustomerDetailsSubreport.jasper"]]></subreportExpression>
        </subreport>

        <!-- Order Items Subreport -->
        <subreport>
            <reportElement x="0" y="160" width="500" height="120"/>
            <dataSourceExpression><![CDATA[
                new net.sf.jasperreports.engine.data.JRBeanCollectionDataSource(
                    (java.util.Collection) $F{lineItems}
                )
            ]]></dataSourceExpression>
           <subreportExpression><![CDATA["subreports/LineItemsSubreport.jasper"]]></subreportExpression>
        </subreport>

        <!-- Payment History Subreport -->
        <subreport>
            <reportElement x="0" y="290" width="500" height="100"/>
            <dataSourceExpression><![CDATA[
                new net.sf.jasperreports.engine.data.JRBeanCollectionDataSource(
                    (java.util.Collection) $F{paymentHistory}
                )
            ]]></dataSourceExpression>

↪   <subreportExpression><![CDATA["subreports/PaymentHistorySubreport.jasper"]]></subreportExpression>
        </subreport>
    </band>
</detail>
```

Service Integration Considerations

Template Upload Structure

When uploading templates with subreports to the service, include all files in your ZIP:



> **Important:** The main report must be the only `.jrxml` file in the root of the ZIP archive. Subreports can be placed in any subdirectory (e.g., `subreports/`). The system looks only at the root level to identify the main template.

Data Source Expression Patterns

For service compatibility, use these data source expression patterns:

Simple Array Field:

```
<dataSourceExpression><![CDATA[
    new net.sf.jasperreports.engine.data.JRBeanCollectionDataSource(
        (java.util.Collection) $F{arrayFieldName}
    )
]]></dataSourceExpression>
```

Nested Object Arrays:

```
<dataSourceExpression><![CDATA[
    new net.sf.jasperreports.engine.data.JRBeanCollectionDataSource(
        (java.util.Collection)
        ((java.util.Map)$F{_originalData}).get("nested.arrayField")
    )
]]></dataSourceExpression>
```

Single Object as Collection:

```
<dataSourceExpression><![CDATA[
    new net.sf.jasperreports.engine.data.JRBeanCollectionDataSource(
        java.util.Arrays.asList($F{singleObjectField})
    )
]]></dataSourceExpression>
```

Complex Data Flow Examples

Example: Multi-Level Invoice Document

Data Structure:

```
{
  "parameters": [
    {"name": "reportTitle", "value": "Detailed Invoice"}
```

```
  ],
  "data": {
    "invoices": [
      {
        "invoiceId": "INV-2025-001",
        "customer": {
          "name": "ACME Corporation",
          "billingAddress": {
            "street": "123 Business Ave",
            "city": "Commerce City",
            "state": "CO",
            "zip": "80022"
          },
          "contacts": [
            {"name": "John Doe", "role": "Manager", "email": "john@acme.com"},
            {"name": "Jane Smith", "role": "AP", "email": "jane@acme.com"}
          ]
        },
        "lineItems": [
          {
            "itemId": "ITEM-001",
            "description": "Professional Services Package",
            "quantity": 1,
            "unitPrice": 5000.00,
            "serviceDetails": [
              {"service": "Consulting", "hours": 40, "rate": 150.00},
              {"service": "Implementation", "hours": 60, "rate": 125.00}
            ]
          }
        ],
        "paymentTerms": {
          "dueDate": "2025-11-15",
          "discountTerms": "2/10 Net 30",
          "lateFeeRate": 1.5
        }
      }
    ]
  }
}
```

Main Report Fields:

```
Fields for main report (invoices array):
- invoiceId (String)
- customer (Object) -> flattened to customer_name, customer_billingAddress_street, etc.
- lineItems (List) -> passed to subreport
- paymentTerms (Object) -> flattened fields
```

Line Items Subreport Fields:

```
Fields for line items subreport:
- itemId (String)
- description (String)
- quantity (Integer)
- unitPrice (BigDecimal)
- serviceDetails (List) -> passed to nested subreport
```

Service Details Subreport Fields:

```
Fields for service details subreport:
- service (String)
- hours (Integer)
- rate (BigDecimal)
```

Troubleshooting Subreports

Common Subreport Issues

1. **Subreport Not Found**

   ```
   Error: Could not load subreport

   Solutions:
   - Ensure .jasper files are compiled and in ZIP
   - Check subreport expression path
   - Verify file names match exactly
   ```

2. **Data Source Issues**

   ```
   Error: Cannot create data source for subreport

   Solutions:
   - Verify field contains Collection or List
   - Check for null values in array fields
   - Use conditional printWhenExpression for optional subreports
   ```

3. **Parameter Passing Problems**

   ```
   Error: Parameter not found in subreport

   Solutions:
   - Ensure parameter names match exactly
   - Define parameters in subreport template
   - Check parameter data types
   ```

Debugging Subreports

Add Debug Information:

```
<!-- In main report, show data being passed to subreport -->
<textField>
    <reportElement x="0" y="200" width="400" height="40"/>
    <textElement>
        <font size="8"/>
    </textElement>
    <textFieldExpression><![CDATA[
        "Debug: Subreport data size = " +
        ($F{lineItems} != null) ?
         ((java.util.List)$F{lineItems}).size() : "null")
    ]]></textFieldExpression>
</textField>
```

Best Practices for Subreport Design

1. **Keep Subreports Simple**: Focus each subreport on a single data aspect
2. **Minimize Parameter Passing**: Pass only essential parameters
3. **Use Consistent Naming**: Match field names between main report and service data
4. **Handle Null Data**: Always check for null collections before creating data sources
5. **Optimize Performance**: Limit subreport complexity for large datasets
6. **Test Incrementally**: Test main report first, then add subreports one by one

Example Templates

Example 1: Simple Invoice Template

JRXML Structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<jasperReport xmlns="http://jasperreports.sourceforge.net/jasperreports"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://jasperreports.sourceforge.net/jasperreports
                                  http://jasperreports.sourceforge.net/xsd/jasperreport.xsd"
              name="SimpleInvoice" pageWidth="595" pageHeight="842"
              columnWidth="535" leftMargin="30" rightMargin="30"
              topMargin="20" bottomMargin="20">

    <!-- Parameters -->
    <parameter name="invoiceNumber" class="java.lang.String"/>
    <parameter name="customerName" class="java.lang.String"/>
    <parameter name="invoiceDate" class="java.util.Date"/>

    <!-- Title -->
    <title>
        <band height="80">
            <staticText>
                <reportElement x="0" y="0" width="200" height="30"/>
                <textElement>
                    <font size="20" isBold="true"/>
                </textElement>
                <text><![CDATA[INVOICE]]></text>
            </staticText>

            <textField>
                <reportElement x="0" y="40" width="150" height="20"/>
                <textElement>
                    <font size="12" isBold="true"/>
                </textElement>
                <textFieldExpression><![CDATA["Invoice #: " + $P{invoiceNumber}]]></textFieldExpression>
            </textField>

            <textField pattern="MM/dd/yyyy">
                <reportElement x="200" y="40" width="100" height="20"/>
                <textElement>
                    <font size="12"/>
                </textElement>
                <textFieldExpression><![CDATA[$P{invoiceDate}]]></textFieldExpression>
            </textField>

            <textField>
                <reportElement x="0" y="60" width="200" height="20"/>
                <textElement>
                    <font size="12"/>
                </textElement>
                <textFieldExpression><![CDATA["Customer: " + $P{customerName}]]></textFieldExpression>
            </textField>
        </band>
    </title>

    <!-- Column Header -->
```

```
    <columnHeader>
        <band height="25">
            <staticText>
                <reportElement x="0" y="0" width="200" height="20"/>
                <box>
                    <bottomPen lineWidth="1.0"/>
                </box>
                <textElement>
                    <font isBold="true"/>
                </textElement>
                <text><![CDATA[Description]]></text>
            </staticText>

            <staticText>
                <reportElement x="200" y="0" width="60" height="20"/>
                <box>
                    <bottomPen lineWidth="1.0"/>
                </box>
                <textElement textAlignment="Center">
                    <font isBold="true"/>
                </textElement>
                <text><![CDATA[Qty]]></text>
            </staticText>

            <staticText>
                <reportElement x="260" y="0" width="80" height="20"/>
                <box>
                    <bottomPen lineWidth="1.0"/>
                </box>
                <textElement textAlignment="Right">
                    <font isBold="true"/>
                </textElement>
                <text><![CDATA[Unit Price]]></text>
            </staticText>

            <staticText>
                <reportElement x="340" y="0" width="80" height="20"/>
                <box>
                    <bottomPen lineWidth="1.0"/>
                </box>
                <textElement textAlignment="Right">
                    <font isBold="true"/>
                </textElement>
                <text><![CDATA[Total]]></text>
            </staticText>
        </band>
    </columnHeader>

    <!-- Detail -->
    <detail>
        <band height="20">
            <textField>
                <reportElement x="0" y="0" width="200" height="18"/>
                <textFieldExpression><![CDATA[$F{description}]]></textFieldExpression>
            </textField>

            <textField>
```

```
            <reportElement x="200" y="0" width="60" height="18"/>
            <textElement textAlignment="Center"/>
            <textFieldExpression><![CDATA[$F{quantity}]]></textFieldExpression>
        </textField>

        <textField pattern="#,##0.00">
            <reportElement x="260" y="0" width="80" height="18"/>
            <textElement textAlignment="Right"/>
            <textFieldExpression><![CDATA[$F{unitPrice}]]></textFieldExpression>
        </textField>

        <textField pattern="#,##0.00">
            <reportElement x="340" y="0" width="80" height="18"/>
            <textElement textAlignment="Right"/>
            <textFieldExpression><![CDATA[$F{totalPrice}]]></textFieldExpression>
        </textField>
    </band>
</detail>

<!-- Summary -->
<summary>
    <band height="100">
        <line>
            <reportElement x="260" y="10" width="160" height="1"/>
        </line>

        <staticText>
            <reportElement x="260" y="20" width="80" height="18"/>
            <textElement textAlignment="Right">
                <font isBold="true"/>
            </textElement>
            <text><![CDATA[Subtotal:]]></text>
        </staticText>

        <textField pattern="#,##0.00">
            <reportElement x="340" y="20" width="80" height="18"/>
            <textElement textAlignment="Right"/>
            <textFieldExpression><![CDATA[$F{subtotal}]]></textFieldExpression>
        </textField>

        <staticText>
            <reportElement x="260" y="40" width="80" height="18"/>
            <textElement textAlignment="Right">
                <font isBold="true"/>
            </textElement>
            <text><![CDATA[Tax:]]></text>
        </staticText>

        <textField pattern="#,##0.00">
            <reportElement x="340" y="40" width="80" height="18"/>
            <textElement textAlignment="Right"/>
            <textFieldExpression><![CDATA[$F{tax}]]></textFieldExpression>
        </textField>

        <staticText>
            <reportElement x="260" y="60" width="80" height="18"/>
            <textElement textAlignment="Right">
```

```
                    <font size="12" isBold="true"/>
                </textElement>
                <text><![CDATA[TOTAL:]]></text>
            </staticText>

            <textField pattern="#,##0.00">
                <reportElement x="340" y="60" width="80" height="18"/>
                <textElement textAlignment="Right">
                    <font size="12" isBold="true"/>
                </textElement>
                <textFieldExpression><![CDATA[$F{total}]]></textFieldExpression>
            </textField>
        </band>
    </summary>
</jasperReport>
```

Sample Request:

```
{
  "parameters": [
    {"name": "invoiceNumber", "value": "INV-2025-001"},
    {"name": "customerName", "value": "ACME Corporation"},
    {"name": "invoiceDate", "value": "2025-10-07"}
  ],
  "data": {
    "lineItems": [
      {
        "description": "Professional Consulting Services",
        "quantity": 40,
        "unitPrice": 150.00,
        "totalPrice": 6000.00
      },
      {
        "description": "Software License (Annual)",
        "quantity": 1,
        "unitPrice": 2500.00,
        "totalPrice": 2500.00
      }
    ],
    "subtotal": 8500.00,
    "tax": 850.00,
    "total": 9350.00
  },
  "filename": "Invoice_ACME_2025_001"
}
```

Example 2: Sales Document Template

This example demonstrates a more complex template with grouped data and calculations.

JRXML Structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<jasperReport xmlns="http://jasperreports.sourceforge.net/jasperreports"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://jasperreports.sourceforge.net/jasperreports
                                  http://jasperreports.sourceforge.net/xsd/jasperreport.xsd"
```

```
            name="SalesReport" pageWidth="595" pageHeight="842"
            columnWidth="535" leftMargin="30" rightMargin="30"
            topMargin="20" bottomMargin="20">

    <!-- Parameters -->
    <parameter name="REPORTS_DIR" class="java.lang.String">
        <defaultValueExpression><![CDATA[""]]></defaultValueExpression>
    </parameter>
    <parameter name="reportTitle" class="java.lang.String"/>
    <parameter name="quarter" class="java.lang.String"/>
    <parameter name="generatedBy" class="java.lang.String"/>

    <!-- Fields from salesData array -->
    <field name="region" class="java.lang.String"/>
    <field name="salesperson" class="java.lang.String"/>
    <field name="product" class="java.lang.String"/>
    <field name="quantity" class="java.lang.Integer"/>
    <field name="revenue" class="java.math.BigDecimal"/>

    <!-- Shared fields from summary object -->
    <field name="_sharedData.summary.totalQuantity" class="java.lang.Integer"/>
    <field name="_sharedData.summary.totalRevenue" class="java.math.BigDecimal"/>
    <field name="_sharedData.summary.topRegion" class="java.lang.String"/>

    <!-- Group by region -->
    <group name="RegionGroup">
        <groupExpression><![CDATA[$F{region}]]></groupExpression>
        <groupHeader>
            <band height="30">
                <textField>
                    <reportElement x="0" y="5" width="200" height="20"
                                   backcolor="#E0E0E0" mode="Opaque"/>
                    <textElement>
                        <font size="12" isBold="true"/>
                    </textElement>
                    <textFieldExpression><![CDATA["Region: " + $F{region}]]></textFieldExpression>
                </textField>
            </band>
        </groupHeader>
    </group>

    <!-- Title -->
    <title>
        <band height="70">
            <textField>
                <reportElement x="0" y="0" width="400" height="30"/>
                <textElement>
                    <font size="18" isBold="true"/>
                </textElement>
                <textFieldExpression><![CDATA[$P{reportTitle}]]></textFieldExpression>
            </textField>

            <textField>
                <reportElement x="0" y="35" width="150" height="20"/>
                <textElement>
                    <font size="11"/>
                </textElement>
```

```
            <textFieldExpression><![CDATA["Period: " + $P{quarter}]]></textFieldExpression>
        </textField>

        <textField>
            <reportElement x="200" y="35" width="200" height="20"/>
            <textElement>
                <font size="11"/>
            </textElement>
           <textFieldExpression><![CDATA["Generated by: " + $P{generatedBy}]]></textFieldExpression>
        </textField>
    </band>
</title>

<!-- Column Header -->
<columnHeader>
    <band height="25">
        <staticText>
            <reportElement x="0" y="0" width="120" height="20"/>
            <box><bottomPen lineWidth="1.0"/></box>
            <textElement><font isBold="true"/></textElement>
            <text><![CDATA[Salesperson]]></text>
        </staticText>
        <staticText>
            <reportElement x="120" y="0" width="120" height="20"/>
            <box><bottomPen lineWidth="1.0"/></box>
            <textElement><font isBold="true"/></textElement>
            <text><![CDATA[Product]]></text>
        </staticText>
        <staticText>
            <reportElement x="240" y="0" width="80" height="20"/>
            <box><bottomPen lineWidth="1.0"/></box>
            <textElement textAlignment="Right"><font isBold="true"/></textElement>
            <text><![CDATA[Quantity]]></text>
        </staticText>
        <staticText>
            <reportElement x="320" y="0" width="100" height="20"/>
            <box><bottomPen lineWidth="1.0"/></box>
            <textElement textAlignment="Right"><font isBold="true"/></textElement>
            <text><![CDATA[Revenue]]></text>
        </staticText>
    </band>
</columnHeader>

<!-- Detail -->
<detail>
    <band height="20">
        <textField>
            <reportElement x="0" y="0" width="120" height="18"/>
            <textFieldExpression><![CDATA[$F{salesperson}]]></textFieldExpression>
        </textField>
        <textField>
            <reportElement x="120" y="0" width="120" height="18"/>
            <textFieldExpression><![CDATA[$F{product}]]></textFieldExpression>
        </textField>
        <textField>
            <reportElement x="240" y="0" width="80" height="18"/>
            <textElement textAlignment="Right"/>
```

```
                <textFieldExpression><![CDATA[$F{quantity}]]></textFieldExpression>
            </textField>
            <textField pattern="#,##0.00">
                <reportElement x="320" y="0" width="100" height="18"/>
                <textElement textAlignment="Right"/>
                <textFieldExpression><![CDATA[$F{revenue}]]></textFieldExpression>
            </textField>
        </band>
    </detail>

    <!-- Summary -->
    <summary>
        <band height="80">
            <line>
                <reportElement x="200" y="10" width="220" height="1"/>
            </line>
            <staticText>
                <reportElement x="200" y="20" width="120" height="18"/>
                <textElement textAlignment="Right"><font isBold="true"/></textElement>
                <text><![CDATA[Total Quantity:]]></text>
            </staticText>
            <textField>
                <reportElement x="320" y="20" width="100" height="18"/>
                <textElement textAlignment="Right"/>

↪   <textFieldExpression><![CDATA[$F{_sharedData.summary.totalQuantity}]]></textFieldExpression>
            </textField>
            <staticText>
                <reportElement x="200" y="40" width="120" height="18"/>
                <textElement textAlignment="Right"><font isBold="true"/></textElement>
                <text><![CDATA[Total Revenue:]]></text>
            </staticText>
            <textField pattern="#,##0.00">
                <reportElement x="320" y="40" width="100" height="18"/>
                <textElement textAlignment="Right"><font size="12" isBold="true"/></textElement>

↪   <textFieldExpression><![CDATA[$F{_sharedData.summary.totalRevenue}]]></textFieldExpression>
            </textField>
            <textField>
                <reportElement x="0" y="60" width="300" height="18"/>
                <textElement><font isItalic="true"/></textElement>
                <textFieldExpression><![CDATA["Top performing region: " +
↪   $F{_sharedData.summary.topRegion}]]></textFieldExpression>
            </textField>
        </band>
    </summary>
</jasperReport>
```

Sample Request:

```
{
  "parameters": [
    {"name": "reportTitle", "value": "Quarterly Sales Summary"},
    {"name": "quarter", "value": "Q3 2025"},
    {"name": "generatedBy", "value": "Sales Manager"}
  ],
  "data": {
```

```
    "salesData": [
      {
        "region": "North",
        "salesperson": "John Smith",
        "product": "Widget A",
        "quantity": 150,
        "revenue": 15000.00
      },
      {
        "region": "North",
        "salesperson": "Jane Doe",
        "product": "Widget B",
        "quantity": 120,
        "revenue": 18000.00
      },
      {
        "region": "South",
        "salesperson": "Bob Johnson",
        "product": "Widget A",
        "quantity": 200,
        "revenue": 20000.00
      }
    ],
    "summary": {
      "totalQuantity": 470,
      "totalRevenue": 53000.00,
      "averageOrderSize": 112.77,
      "topRegion": "South",
      "topProduct": "Widget A"
    }
  }
}
```

Muban CLI Tool

The **Muban CLI** is a command-line interface that simplifies template development, testing, and deployment workflows. It works on Windows, macOS, and Linux.

Installation

Install from PyPI using pip (requires Python 3.9+):

```
pip install muban-cli
```

Configuration

```
# Configure the server URL
muban configure --server https://your-muban-server.com

# Login (if authentication is enabled)
muban login

# Check configuration
muban configure --show
```

Template Packaging

The `package` command analyzes your JRXML template and creates a deployment-ready ZIP with all dependencies:

```
# Package a template with all its dependencies (images, subreports)
muban package invoice.jrxml

# Specify output filename
muban package invoice.jrxml -o invoice-template.zip

# Dry run - see what would be included without creating the ZIP
muban package invoice.jrxml --dry-run -v
```

Features:

- Automatic asset discovery (images, subreports, fonts)
- Recursive subreport analysis
- Handles `REPORTS_DIR` parameter for path resolution
- Warns about missing files

Template Upload

```
# Upload the packaged template
muban push invoice-template.zip --name "Invoice Template" --author "Finance Team"

# List all templates on the server
muban list

# Get template details including parameters and fields
muban get TEMPLATE_ID --params --fields
```

Document Generation

Test your templates directly from the command line:

```
# Generate with parameters
muban generate TEMPLATE_ID -p title="Monthly Statement" -p date="2026-02-09"

# Different output formats
muban generate TEMPLATE_ID -F pdf -o report.pdf
muban generate TEMPLATE_ID -F xlsx -o report.xlsx
muban generate TEMPLATE_ID -F docx -o report.docx

# Using a parameter file
muban generate TEMPLATE_ID --params-file params.json

# Using JSON data source
muban generate TEMPLATE_ID --data-file data.json

# PDF/A compliance
muban generate TEMPLATE_ID --pdf-pdfa PDF/A-1b
```

Parameter file format (params.json):

```
{
  "reportTitle": "Quarterly Sales Summary",
  "reportDate": "2026-02-09",
  "department": "Sales"
}
```

Data source file format (data.json):

```
{
  "items": [
    {"productName": "Widget A", "quantity": 100, "unitPrice": 25.50},
    {"productName": "Widget B", "quantity": 50, "unitPrice": 45.00}
  ],
  "summary": {
    "totalItems": 150,
    "totalValue": 4800.00
  }
}
```

Typical Designer Workflow

```
# 1. Configure CLI (one-time setup)
muban configure --server http://localhost:8080
muban login

# 2. Design template in Jaspersoft Studio
#    ... create invoice.jrxml with images, subreports ...

# 3. Package the template
muban package invoice.jrxml -o invoice.zip

# 4. Upload to server
muban push invoice.zip --name "Invoice" --author "Designer Name"
```

```
# 5. Get the template ID
muban list --search "Invoice"

# 6. Test document generation
muban generate abc123-uuid -p customerName="ACME Corp" -p invoiceDate="2026-02-09"

# 7. Verify parameters are correctly registered
muban get abc123-uuid --params
```

Useful CLI Commands for Designers

Command	Description
`muban package <file.jrxml>`	Package template with dependencies
`muban push <file.zip>`	Upload template to server
`muban list`	List all templates
`muban get <ID> --params --fields`	Show template parameters and fields
`muban generate <ID> -p key=value`	Generate document with parameters
`muban pull <ID>`	Download template ZIP
`muban fonts`	List available fonts on server

For complete documentation, see the Muban CLI on PyPI.

Appendix

Quick Reference

Expression Syntax

Type	Syntax	Example
Field	`$F{fieldName}`	`$F{customerName}`
Parameter	`$P{parameterName}`	`$P{reportDate}`
Variable	`$V{variableName}`	`$V{PAGE_NUMBER}`
Resource	`$R{key}`	`$R{label.title}`

System Variables

Variable	Type	Description
`PAGE_NUMBER`	Integer	Current page number
`COLUMN_NUMBER`	Integer	Current column number
`REPORT_COUNT`	Integer	Number of processed records
`PAGE_COUNT`	Integer	Number of records on page
`COLUMN_COUNT`	Integer	Number of records in column

Variable Calculation Types

Type	Description
Nothing	No calculation
Count	Counts values
DistinctCount	Counts unique values
Sum	Sums values
Average	Calculates average
Lowest	Finds minimum
Highest	Finds maximum
StandardDeviation	Calculates standard deviation
Variance	Calculates variance
First	First value in group
System	System calculations

Variable Reset Types

Type	Description
Report	Reset at end of report
Page	Reset on each page
Column	Reset on each column
Group	Reset on group change
None	No reset

Output Formats

Format	Description	Use Case
PDF	Portable Document Format	Printing, archiving
XLSX	Microsoft Excel	Data analysis, editing
DOCX	Microsoft Word	Text documents
HTML	Web page	Online display
XML	Extensible Markup Language	Data exchange

Useful Links

- JasperReports Documentation
- Jaspersoft Studio - download
- Muban CLI - PyPI - Command-line tool for template management
- DejaVu Fonts
- Noto Fonts
- Liberation Fonts

Office Format Template Designer Handbook

This handbook provides practical guidance for template designers creating DOCX-based document templates for the Document Generation Service. No programming knowledge is required — templates are authored entirely in Microsoft Word.

Overview

The DOCX template engine lets you create document templates directly in Microsoft Word. You write your document as usual, inserting `${placeholder}` markers where dynamic content should appear. The service replaces these markers with actual data at generation time and produces a filled document in your chosen format (DOCX, PDF, HTML, or TXT).

Placeholders can also contain **conditional expressions** for dynamic text output — for example, choosing between “Mr.” and “Mrs.” based on gender, or selecting singular/plural forms based on a count. See the Conditional Expressions section for details.

When to Use DOCX Templates

DOCX templates are best suited for:

- **Letters and correspondence** — notices, invitations, cover letters
- **Legal documents** — contracts, agreements, terms and conditions
- **Government forms** — certificates, decisions, official correspondence
- **Business documents** — invoices, purchase orders, delivery notes
- **HR documents** — employment contracts, offer letters, payslips

What You Need

- **Microsoft Word** (2016 or later recommended) or any compatible editor (LibreOffice Writer, WPS Office)
- **A ZIP archiver** (built into Windows/macOS, or 7-Zip)
- Access to the Document Generation Service API or a front-end that integrates with it

Output Formats

Format	URL path	Description
DOCX	`/generate/docx`	Filled Word document — identical to your template but with placeholders replaced
PDF	`/generate/pdf`	Converted from the filled DOCX — suitable for printing and distribution
HTML	`/generate/html`	HTML page returned as a ZIP archive (contains `index.html` and assets)
TXT	`/generate/txt`	Plain text representation of the document

Getting Started

Step 1: Create Your Template in Word

Open Microsoft Word and create your document as you normally would. Apply all the formatting, styles, headers, footers, tables, and images you need.

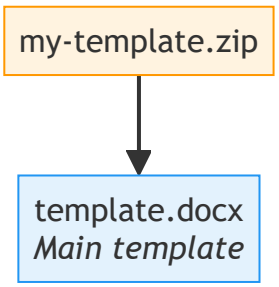
Wherever you want dynamic content, type a placeholder using the `${name}` syntax:

```
Dear ${recipientName},

We are writing to confirm your appointment on ${appointmentDate}.

Your reference number is: ${referenceNumber}
```

Save the file as a standard `.docx` document (e.g., `template.docx`).

Step 2: Package as ZIP

The service expects templates as ZIP files. Create a ZIP archive containing your DOCX file:



On Windows, select the file → right-click → **Send to** → **Compressed (zipped) folder**.

Step 3: Upload

Upload the ZIP file to the service via the API or your application's template management interface.

The service will:

1. Detect the template type as DOCX automatically
2. Scan the document for all `${placeholder}` patterns
3. Store extracted placeholders as metadata (visible via the API)
4. Auto-extract the template description from the document's core properties (`dc:description`) if none was provided during upload

   Tip: Set the document description in Microsoft Word via **File** → **Properties** → **Description** (or **File** → **Info** → **Properties** → **Advanced Properties** → **Summary** → **Comments**). This value will be used as the template description automatically.

Step 4: Generate Documents

Send a generation request with the placeholder values filled in. The output format is selected via the URL path — e.g., `POST /templates/{id}/generate/pdf`. The service returns the completed document in the requested format.

Placeholder Syntax

Basic Placeholders

A placeholder is any text wrapped in `${` and `}`. The name inside can contain letters, numbers, and underscores:

Placeholder	Valid	Notes
`${name}`	Yes	Simple name
`${firstName}`	Yes	camelCase recommended
`${case_number}`	Yes	Underscores allowed
`${address.city}`	Yes	Dot notation for nested data
`${ name }`	No	Spaces inside break detection
`${first name}`	No	Spaces not allowed in names

Where Placeholders Work

You can place `${placeholder}` markers in:

- **Body text** — paragraphs, headings, any text content
- **Headers and footers** — page headers, footers, page numbers area
- **Table cells** — any cell in any table
- **Text boxes** — floating text containers

Nested Data (Dot Notation)

When the data has a hierarchical structure, use dot notation to access nested values:

Template:

```
Company: ${company.name}
Address: ${company.address.street}, ${company.address.city}
Tax ID: ${company.taxId}
```

Data provided:

```
{
  "company": {
    "name": "ACME Sp. z o.o.",
    "taxId": "PL1234567890",
    "address": {
      "street": "Marszałkowska 1",
      "city": "Warszawa"
    }
  }
}
```

Result:

```
Company: ACME Sp. z o.o.
Address: Marszałkowska 1, Warszawa
Tax ID: PL1234567890
```

Unresolved Placeholders

If a placeholder name doesn't match any provided data, the placeholder text remains as-is in the output. For example, `${missingValue}` will appear literally as `${missingValue}` in the generated document. This makes it easy to spot missing data during testing.

Dynamic Image Replacement

Images in the template can be dynamically replaced at document generation time. This is useful for facsimile signatures, company stamps, approval seals, or any image that varies per document.

Setting Up a Placeholder Image

1. **Insert an image** into the template — this will serve as the default/example image
2. **Right-click the image** → **Edit Alt Text** (or: Format Picture → Alt Text)
3. Set the alt text to `image:` followed by a **key**, for example: `image:facsimile`
4. Position and size the image exactly as needed — the replacement will keep your layout

The `image:` prefix tells the system this is a dynamic placeholder. Everything after `image:` is the **key** — the API uses this key to identify which image to replace.

Choosing the Key

The key can be either a **simple name** or a **relative file path** within the template ZIP — each approach has different fallback behavior:

Alt text example	Key	Fallback when no API data
`image:facsimile`	`facsimile`	Keeps the original image embedded in the template
`image:assets/facsimile.png`	`assets/facsimile.png`	Loads `assets/facsimile.png` from the template ZIP

Use a simple key (e.g., `image:facsimile`) when the image is always provided via API at generation time. The original image in the template serves as a visual placeholder during design.

Use a path key (e.g., `image:assets/facsimile.png`) when you want a default image bundled in the template ZIP that gets used automatically if the API doesn't send an override.

How Replacement Works

When a document is generated, the system scans for images with `image:` alt text and resolves replacements in this order:

Priority	What happens
1. API provides image data	If the request includes a parameter matching the key (Base64 data URI, URL, or a file path), the image is replaced
2. Key used as file path	If no API data, the key itself is treated as a relative path within the template ZIP — if a file exists there, it replaces the image
3. Keep original	If nothing is found, the original image from the template stays unchanged

Providing Images via API

Option A — Base64 encoded image (per-request dynamic, e.g., a different signer each time):

```
{
  "parameters": [
    { "name": "facsimile", "value": "..." }
  ]
}
```

Option B — URL to an external image (fetched at generation time from HTTP/HTTPS):

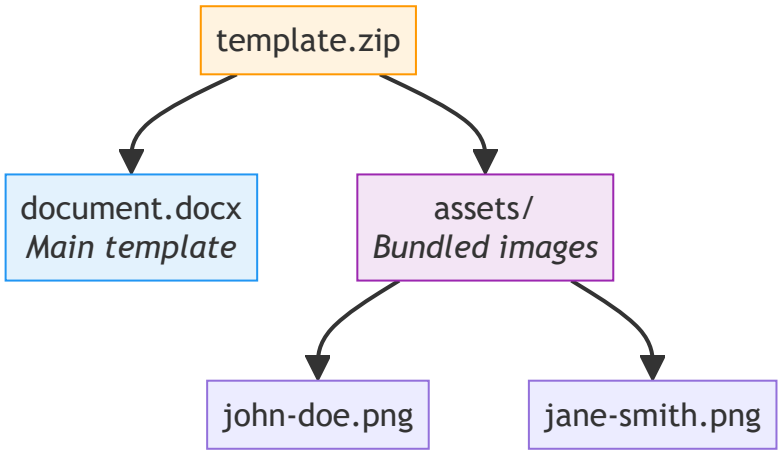
```
{
  "parameters": [
    { "name": "facsimile", "value": "https://cdn.example.com/signatures/john-doe.png" }
  ]
}
```

The URL must return a valid image content type (`image/png`, `image/jpeg`, `image/gif`, etc.). Limits: 10 s connect timeout, 10 s read timeout, 10 MB max response. If the fetch fails, the original placeholder image stays.

Option C — Path to a bundled image (select from pre-bundled options in the ZIP):

```
{
  "parameters": [
    { "name": "facsimile", "value": "assets/john-doe.png" }
  ]
}
```

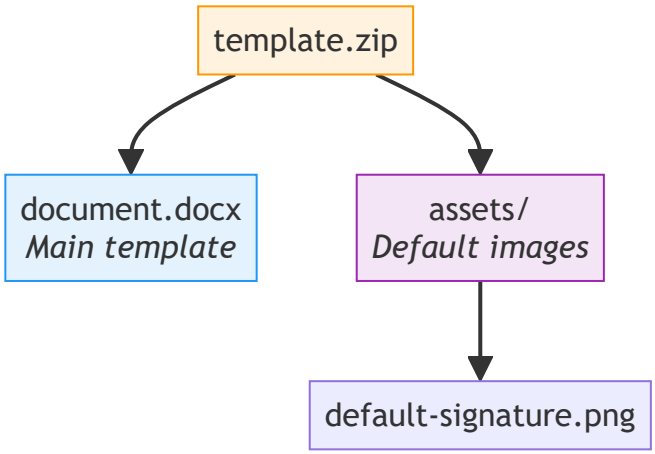
For Option C, the template ZIP must contain the referenced file:



Example: Bundled Default with Override Option

If you want a default signature that gets replaced automatically **and** can be overridden per request, use a path as the key:

Alt text: `image:assets/default-signature.png`



The `default-signature.png` is used automatically (no API parameter needed).

To override with a different image at generation time:

```
{
  "parameters": [
    { "name": "assets/default-signature.png", "value": "..." }
  ]
}
```

Tips

- The alt text **must** start with `image:` — other images (logos, decorations) are left untouched
- Both **inline** and **floating** images are supported
- Images in **headers**, **footers**, and **table cells** all work
- If no replacement data is found (and the key is not a valid path), the original image stays — design your template with a good default

Conditional Image Selection (SpEL Expressions)

The key after `image:` can contain `${...}` SpEL expressions — the same expression syntax used in text placeholders. This lets the **template designer** control which image is shown based on data values, without requiring the API caller to decide.

Gender-based image:

```
image:${ gender == 'F' ? 'assets/female.png' : 'assets/male.png' }
```

Risk-level indicator:

```
image:${ risk > 80 ? 'assets/exclamation.png' : 'assets/info.png' }
```

Department-specific stamp (mixed literal + expression):

```
image:assets/${department}/stamp.png
```

With `department = "finance"`, this resolves to `assets/finance/stamp.png` in the template ZIP.

How it works:

1. The `${...}` expressions are evaluated against the request data (same context as text placeholders)
2. The evaluated result is then resolved through the normal cascade (Base64 / URL / file path)
3. If evaluation fails or the resolved file doesn't exist, the original placeholder image stays

   Important: Use **single quotes** (`'...'`) for string literals inside expressions. Double quotes conflict with the XML attribute encoding in the DOCX file. SpEL treats single and double quotes identically, so `'female'.equals(gender)` works exactly the same as `"female".equals(gender)`.

Conditional Expressions

Placeholders can contain conditional logic using **ternary expressions**. This lets you produce different text based on the data — without creating multiple templates.

Basic Syntax

```
${condition ? "value if true" : "value if false"}
```

The expression is evaluated at generation time. If the condition is true, the first value is used; if false, the second.

Examples: Conditional Expressions

Gender-based honorific:

```
Dear ${"female".equals(customer_gender) ? "Mrs." : "Mr."} ${customer_name},
```

With `customer_gender = "female"`, `customer_name = "Anna Kowalska"` → **Dear Mrs. Anna Kowalska,** With `customer_gender = "male"`, `customer_name = "Jan Kowalski"` → **Dear Mr. Jan Kowalski,**

Quantity-dependent text:

```
You have ordered ${items.size() > 3 ? "a lot of" : "a few"} items.
```

With 5 items → **You have ordered a lot of items.** With 2 items → **You have ordered a few items.**

Singular/plural:

```
Your order contains ${items.size()} ${items.size() == 1 ? "position" : "positions"}.
```

With 1 item → **Your order contains 1 position.** With 4 items → **Your order contains 4 positions.**

Age-based classification:

```
Account type: ${age >= 18 ? "Full account" : "Junior account"}
```

Null/missing value fallback (Elvis operator):

```
Nickname: ${nickname ?: "N/A"}
```

If `nickname` is null or not provided → **Nickname: N/A**

Nested Conditions

You can nest ternary expressions using parentheses:

```
${age >= 18 ? (vip ? "VIP Adult" : "Regular Adult") : "Junior"}
```

age	vip	Result
25	true	VIP Adult
25	false	Regular Adult
15	true	Junior

Keep nesting to one level for readability. For complex logic, consider pushing the decision to the calling application.

Available Operations in Expressions

Operation	Example	Description
Comparison	`age >= 18`	`>`, `<`, `>=`, `<=`, `==`, `!=`
String equality	`"female".equals(gender)`	Safe null-proof string comparison
List size	`items.size()`	Number of elements in an array
Empty check	`items.isEmpty()`	True if the array has no elements
String methods	`name.toUpperCase()`	`toUpperCase()`, `toLowerCase()`, `length()`
Arithmetic	`price * qty`	`+`, `-`, `*`, `/`
Logical	`age > 18 && vip`	`&&` (and), `||` (or), `!` (not)
Elvis operator	`nickname ?: "N/A"`	Use right side when left is null
Nested map access	`address.city`	Access nested data fields

Expressions in Tables

Conditional expressions work inside table template rows too. In a replicated row, expressions can access the current row's fields directly:

Product	Status
`${items.name}`	`${items.quantity > 100 ? "Bulk" : "Standard"}`

Inside a replicated row, the current row's fields (e.g., `quantity`) are accessible by simple name — no `items.` prefix needed for expressions.

How Expressions Are Detected

The service automatically distinguishes between simple placeholders and expressions:

Placeholder	Type	Behavior
`${recipientName}`	Simple key	Direct value lookup
`${address.city}`	Dot-notation key	Nested value lookup
`${items.size() > 3 ? "a lot" : "a few"}`	Expression	Evaluated as logic
`${name.toUpperCase()}`	Expression	Method call evaluated

Any placeholder containing operators (`>`, `?`, `+`), method calls (`()`), or string literals (`"..."`) is automatically treated as an expression. Simple names (letters, numbers, dots, underscores) are always treated as key lookups — existing templates continue to work unchanged.

Expressions and Placeholder Discovery

When you upload a template, the service scans it for `${placeholder}` patterns and stores them as parameter metadata. Expressions are **excluded** from this discovery — only simple key placeholders appear as parameters. This is by design: expressions are logic, not input parameters.

Limitations

- **No curly braces in expression text** — the `}` character ends the placeholder, so literal `}` inside strings will break the expression. Pass brace-containing values as parameters instead:

  ```
  ${is_active ? label_active : label_inactive}
  ```

  Where `label_active = "{ACTIVE}"` is provided as a parameter value.

- **Expressions must be in a single formatting run** — just like regular placeholders, the entire `${...}` must have uniform formatting (same font, size, bold/italic). If Word splits the expression across formatting boundaries, it won't be recognized.

- **Security restrictions** — expressions can read data and call simple methods, but cannot access the server, create objects, or execute system commands. This is enforced by the template engine.

Smart Quotes (Typographic Quotes) — IMPORTANT

Microsoft Word automatically converts straight quotes (") into typographic ("smart") quotes as you type. The exact characters depend on your Word language setting:

Language	Opening	Closing	Example
English	“	”	“Hello”
Polish / German	„	”	„Hello”
French	«	»	«Hello»

Smart quotes will break expressions. The expression engine requires standard ASCII straight quotes (" U+0022). If Word replaces them with typographic quotes, the expression will fail and the placeholder will appear unresolved in the output.

You must disable smart quotes before writing expressions. Follow these steps:

1. Go to **File** → **Options** → **Proofing** → **AutoCorrect Options…**
2. Select the **AutoFormat As You Type** tab
3. Uncheck **“Straight quotes” with “smart quotes”**
4. Also select the **AutoFormat** tab and uncheck the same option
5. Click **OK**

> **Tip:** You only need to disable this while typing expressions. You can re-enable it afterward for normal document text. Alternatively, type your expressions in a plain text editor (Notepad) and paste them into Word — pasted text is not affected by AutoCorrect.

How to tell if smart quotes are the problem:

If your expression appears literally in the output (e.g., `${"female".equals(gender) ? "Mrs." : "Mr."}` is not replaced), and you're sure the data is correct, check the quotes. In Word, place your cursor on a quote character and look at the status bar or use **Insert** → **Symbol** to check its Unicode code point. Straight quotes are `U+0022`, smart quotes are `U+201C`/`U+201D`/`U+201E` etc.

Quick fix without changing settings: Select the affected smart quote characters in your expression and replace them one by one — delete the smart quote character, then type a straight quote while holding `Ctrl+Z` immediately after (this undoes Word's auto-correction of that specific character).

Note: This limitation only applies to string literals **inside expressions**. Simple placeholders like `${recipientName}` contain no quotes and are never affected. Data values (from parameters and JSON data) are also not affected — they can contain any Unicode characters including smart quotes.

Data Types in Expressions

Expressions work with the **original data types** from the request:

JSON Value	Type in Expression	Can Do
`25`	Integer	`age >= 18`
`29.99`	Decimal	`price * qty`
`"female"`	String	`"female".equals(gender)`
`true`	Boolean	`vip ? "VIP" : "Regular"`
`[...]`	List	`items.size()`, `items.isEmpty()`
`{...}`	Map (object)	`address.city`

Conditional Document Blocks

Conditional blocks let you include or exclude entire sections of a document based on data values. Unlike conditional expressions (which choose between two inline text values), conditional blocks control the **visibility of whole paragraphs, tables, and images**.

Syntax

Conditional blocks use `#{if expr}`, optional `#{else}`, and `#{fi}` markers. Each marker must be in **its own paragraph** (a separate line in the Word document):

```
#{if customer_debt > 1000}
This section appears only when the condition is true.
#{fi}
```

With an else branch:

```
#{if customer_debt > 1000}
You have a high outstanding debt.
#{else}
Your account is in good standing.
#{fi}
```

The `#{...}` delimiters are intentionally different from `${...}` placeholders to make it clear that these are **structural directives**, not value substitutions.

Examples: Conditional Blocks

Show a legal notice only for high-debt customers:

```
Dear ${customer_name},

Thank you for your continued business.

#{if customer_debt > 1000}
IMPORTANT: Your account has an outstanding balance of ${customer_debt} PLN.
Please settle the payment within 14 days to avoid legal action.
#{fi}

Kind regards,
${company_name}
```

With `customer_debt = 1500` → the notice paragraph appears. With `customer_debt = 200` → the notice paragraph is removed, no blank space left.

VIP-only content:

```
#{if status == 'VIP'}
As a VIP member, you are entitled to a 20% discount on all services.
Please present this letter at the reception desk.
#{fi}
```

Show a section based on boolean flag:

```
#{if include_attachment_list}
The following documents are attached:
```

```
- Certificate of incorporation
- Tax clearance certificate
- Power of attorney
#{fi}
```

If/else — show different content depending on condition:

```
#{if items.size() > 1}
Below is a table with ${items.size() > 2 ? "many" : "few"} items.
| Item | Value |
${items.name}  ${items.value}
#{else}
No table to display – there is nothing to show.
#{fi}
```

With `items = [{name: "A", value: 1}, {name: "B", value: 2}]` → the table appears. With `items = [{name: "A", value: 1}]` → only the “No table” message appears.

Multiple independent blocks in one document:

```
#{if show_header_notice}
This document is confidential and intended for the addressee only.
#{fi}

Document content here...

#{if show_footer_disclaimer}
Disclaimer: This document does not constitute legal advice.
#{fi}
```

Each block is evaluated independently — you can have any combination of visible/hidden blocks.

Expressions

The condition inside `#{if ...}` supports the same SpEL expressions as `${...}` placeholders:

Expression	Meaning
`#{if is_premium}`	Boolean variable
`#{if customer_debt > 1000}`	Numeric comparison
`#{if status == 'VIP'}`	String equality
`#{if items.size() > 0}`	List method call
`#{if age >= 18 && is_citizen}`	Logical AND
`#{if nickname ?: false}`	Truthy check (non-null, non-empty)

Boolean Coercion

The expression result is coerced to true/false:

Result	Treated as
`true`	**true**
`false`	**false**
Non-zero number	**true**

Result	Treated as
`0`	**false**
Non-empty string (not `"false"`)	**true**
Empty string or `null`	**false**
Missing variable	**false**

Rules and Limitations

1. **Markers must be standalone paragraphs** — don't put other text on the same line as `#{if}`, `#{else}`, or `#{fi}`
2. **No nesting** — you cannot put a `#{if}` block inside another `#{if}` block
3. **`#{else}` is optional** — you can use just `#{if}`/`#{fi}` when you don't need an alternative branch
4. **Blocks work everywhere** — body text, headers, footers, and inside table cells
5. **Placeholders inside conditional blocks** — `${...}` placeholders between the markers are only evaluated if their branch is active. If the condition is false, the if-branch (including its placeholders) is removed without evaluation. This means you can safely reference variables that only exist for certain conditions.
6. **Smart quotes** — the same smart quote warning applies to string literals inside `#{if}` expressions
7. **Unmatched markers** — an `#{if}` without a matching `#{fi}` (or vice versa) is left as-is in the output and logged as a warning

Processing Order

Conditional blocks are processed **before** placeholder substitution. This means:

1. `#{if expr}` / `#{else}` / `#{fi}` blocks are evaluated and removed/kept
2. Then `${placeholder}` values are filled in
3. Then `image:{key}` images are replaced

This order ensures that placeholders inside a false-condition block are never evaluated.

Conditional Blocks vs. Conditional Expressions

Feature	Conditional Expressions `${...}`	Conditional Blocks `#{if}`
Scope	Single inline value	Entire paragraphs/sections
Syntax	`${condition ? "yes" : "no"}`	`#{if condition}` … `#{else}` … `#{fi}`
Use case	Choose between two words/phrases	Show/hide whole sections
Has else branch	Via ternary `? :`	`#{else}` (optional)
Can contain tables/images	No	Yes
Can be nested	Yes (via parentheses)	No

Working with Tables

Static Tables

Tables without array placeholders work like any other part of the document — placeholders in cells are simply replaced:

Template:

Detail	Value
Name	`${clientName}`
Date	`${issueDate}`
Amount	`${totalAmount}`

Result:

Detail	Value
Name	Jan Kowalski
Date	2026-02-25
Amount	15 432,00

Dynamic Tables (Row Replication)

This is the most powerful feature. When placeholders in a table row reference an **array** of data, the service automatically generates one row per array element.

How it works:

1. You design a single “template row” in your table with placeholders like `${items.name}`, `${items.qty}`
2. The `items` part identifies the data array
3. The `.name`, `.qty` parts identify which field from each array element to use
4. The service clones this row for every element in the array and fills in the values

Template:

Product	Quantity	Unit Price	Total
`${items.product}`	`${items.quantity}`	`${items.unitPrice}`	`${items.total}`

Data:

```
{
  "items": [
    {"product": "Office Chair", "quantity": 5, "unitPrice": 899.99, "total": 4499.95},
    {"product": "Standing Desk", "quantity": 3, "unitPrice": 1599.00, "total": 4797.00},
    {"product": "Monitor Arm", "quantity": 10, "unitPrice": 149.50, "total": 1495.00}
  ]
}
```

Result:

Product	Quantity	Unit Price	Total
Office Chair	5	899,99	4 499,95
Standing Desk	3	1 599,00	4 797,00
Monitor Arm	10	149,50	1 495,00

Table Design Rules

- **Header rows stay fixed** — only the row containing `${array.field}` placeholders is replicated
- **All formatting is preserved** — borders, cell shading, fonts, alignment, text color
- **Empty arrays** — if the data array is empty, the template row is removed entirely
- **Multiple tables** — you can use different arrays in different tables (e.g., `${items.name}` in one table, `${payments.date}` in another)
- **Mixed content** — static placeholders and array placeholders can coexist in the same document

Table Styling Tips

When designing tables that will be replicated:

1. **Use Word table styles** — apply a built-in or custom table style (e.g., “Grid Table 4 - Accent 1”) for consistent appearance. The service preserves banded rows, header formatting, and border styles in PDF output.
2. **Format the template row** — the formatting of your template row is cloned to every generated row. Set fonts, alignment, number formats, and cell padding on this single row.
3. **Test with varying row counts** — verify your template looks good with 1 row, 5 rows, and 50+ rows. Consider page breaks for large datasets.
4. **Column widths** — set fixed column widths rather than auto-fit. This ensures consistent layout regardless of data content.

Number Formatting and Locales

Automatic Locale Formatting

When a generation request includes a `documentLocale` parameter, numeric values are automatically formatted according to that locale's conventions:

Locale	Number	Formatted
`pl_PL` (Polish)	`1234567.89`	`1 234 567,89`
`de_DE` (German)	`1234567.89`	`1.234.567,89`
`en_US` (English)	`1234567.89`	`1,234,567.89`
`fr_FR` (French)	`1234567.89`	`1 234 567,89`
(no locale)	`1234567.89`	`1234567.89`

Locale formatting applies to all numeric values in both scalar placeholders and table data. String values are never reformatted.

Supported Locale Formats

Locales can be specified as:

- Language only: `"en"`, `"pl"`, `"de"`
- Language + country: `"pl_PL"`, `"en_US"`, `"de_DE"`
- Language + country + variant: `"no_NO_NY"`

Fonts

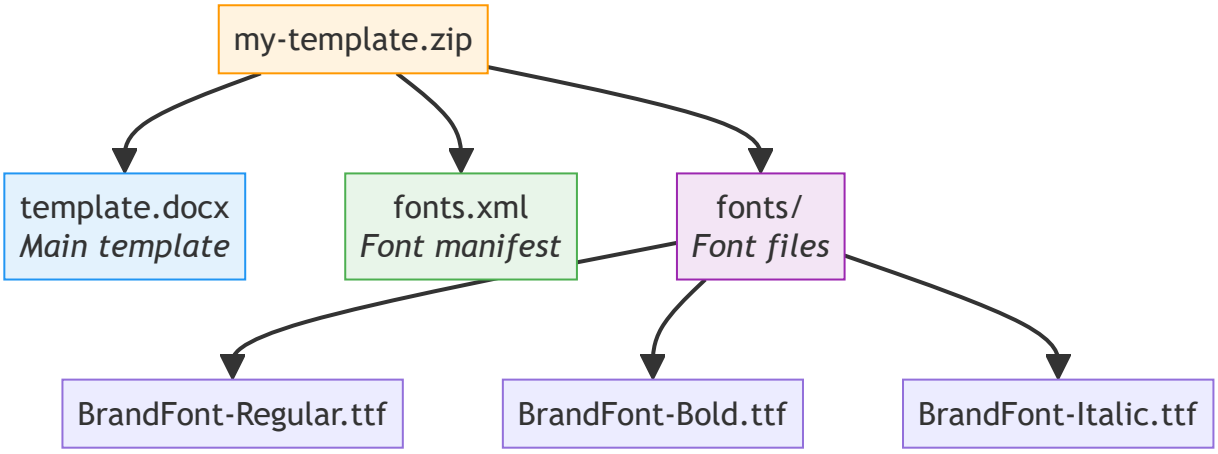
Using Standard Fonts

For maximum compatibility, use fonts that are available on the server where the service runs. Common safe choices:

- **Arial** / **Helvetica**
- **Times New Roman** / **Times**
- **Courier New** / **Courier**
- **Liberation Sans** / **Liberation Serif** (Linux-friendly alternatives)
- **DejaVu Sans** / **DejaVu Serif** (good Unicode coverage)

Bundling Custom Fonts

If your template uses a font that isn't available on the server, you can bundle it with the template. Add a `fonts.xml` manifest and a `fonts/` directory to your ZIP:



fonts.xml example:

```
<?xml version="1.0" encoding="UTF-8"?>
<fontFamilies>
  <fontFamily name="BrandFont">
    <normal>fonts/BrandFont-Regular.ttf</normal>
    <bold>fonts/BrandFont-Bold.ttf</bold>
    <italic>fonts/BrandFont-Italic.ttf</italic>
    <pdfEncoding>Identity-H</pdfEncoding>
    <pdfEmbedded>true</pdfEmbedded>
  </fontFamily>
</fontFamilies>
```

Important:

- Use `.ttf` (TrueType) or `.otf` (OpenType) static font files
- Variable fonts (`.ttf` files with variable axes) are **not supported**
- Font names must match exactly what you used in Word (case-sensitive)
- Template fonts cannot override system fonts for security reasons

Font Tips for PDF Output

- **Always embed fonts** when generating PDF — unembedded fonts may fall back to a default, changing the appearance
- **Test PDF output** early in the design process to catch font issues before finalizing the template
- **Use bold/italic font files** rather than relying on Word's synthetic bold/italic — the PDF converter maps font variants by file, not by styling

PDF Output Considerations

What Converts Well

- Text formatting (bold, italic, underline, strikethrough, font size, color)
- Paragraphs (alignment, spacing, indentation)
- Tables (borders, cell shading, merged cells, column widths)
- Headers and footers
- Page size and margins
- Bulleted and numbered lists

What May Not Convert Perfectly

- Complex text wrapping around images
- Advanced positioning (text boxes with precise anchoring)
- SmartArt and drawn objects
- Embedded OLE objects (Excel charts, etc.)
- Some advanced table features (diagonal cell borders)
- Watermarks (may need to be recreated as header images)

Design Recommendations for PDF

1. **Keep layouts simple** — straightforward linear layouts convert best
2. **Use tables for alignment** — rather than tabs or text boxes, use borderless tables to position content
3. **Set explicit page margins** — don’t rely on Word’s default margins; set them explicitly in Page Layout
4. **Test early and often** — generate a PDF after each major layout change to verify fidelity

PDF Security

When generating PDF output, you can optionally apply security settings:

Feature	Description
User password	Password required to open and view the PDF
Owner password	Password required to change security settings
Encryption	AES-128 or AES-256 bit encryption
Print permission	Allow or prevent printing
Copy permission	Allow or prevent text/image copying
Modify permission	Allow or prevent document editing

These settings are specified in the generation request, not in the template itself. The same template can produce both secured and unsecured PDFs depending on the request.

Template Design Best Practices

Placeholder Naming

Convention	Example	When to Use
`camelCase`	`${firstName}`	Simple scalar values
`dotNotation`	`${address.city}`	Nested data structures
`arrayDotField`	`${items.price}`	Table row data
Descriptive names	`${invoiceTotalAmount}`	When clarity matters

Avoid abbreviations that aren't obvious — `${rcpNm}` is harder to maintain than `${recipientName}`.

Formatting Consistency

Apply consistent formatting to each placeholder:

Each `${placeholder}` must use the same font, size, bold/italic settings throughout the placeholder text. If part of a placeholder is bold and part isn't, Word splits it into multiple XML runs internally, which may prevent the service from recognizing the placeholder.

Good:

- Type `${recipientName}` in one go, all in the same font and style

Bad:

- Type `${recipient`, then select it and make it bold, then type `Name}` in regular weight
- Copy-paste parts of a placeholder from different sources with different formatting

Fix: If a placeholder isn't being replaced, select the entire `${...}` text and reapply a uniform font/style to it.

Document Structure

1. **Design the full document first** — get the layout, headers, footers, and styling right before adding placeholders
2. **Add placeholders last** — once the design is stable, replace sample text with `${placeholder}` markers
3. **Test with realistic data** — use data that matches production lengths and formats to verify the layout handles real content
4. **Consider page breaks** — for documents with dynamic-length tables, plan where page breaks should fall

Common Pitfalls

Problem	Cause	Solution
Placeholder not replaced	Inconsistent formatting across the placeholder text	Select the entire `${...}` and apply uniform formatting
Placeholder not replaced	Extra invisible characters (e.g., zero-width space)	Retype the placeholder manually
Table rows not replicating	Placeholder doesn't use `${array.field}` dot notation	Use `${arrayName.fieldName}` format
Numbers not formatted	No `documentLocale` in the generation request	Add `"documentLocale": "pl_PL"` to the request

Problem	Cause	Solution
PDF fonts look wrong	Font not available on the server	Bundle the font in the template ZIP
PDF layout differs from Word	Complex positioning or text wrapping	Simplify the layout; use tables for alignment
Empty rows in output	Array field name doesn’t match provided data	Check field names match exactly (case-sensitive)

Complete Example

Scenario: Monthly Invoice

1. Template design (`invoice.docx`):

The document contains a company header, client details as scalar placeholders, a line items table with array placeholders, and a footer with totals.

```
                        INVOICE
                  ${company.name}
            ${company.address.street}
      ${company.address.zipCode} ${company.address.city}
             Tax ID: ${company.taxId}

Invoice No: ${invoiceNumber}
Issue Date: ${issueDate}
Due Date:   ${dueDate}

Bill To:
  ${client.name}
  ${client.address.street}
  ${client.address.zipCode} ${client.address.city}
```

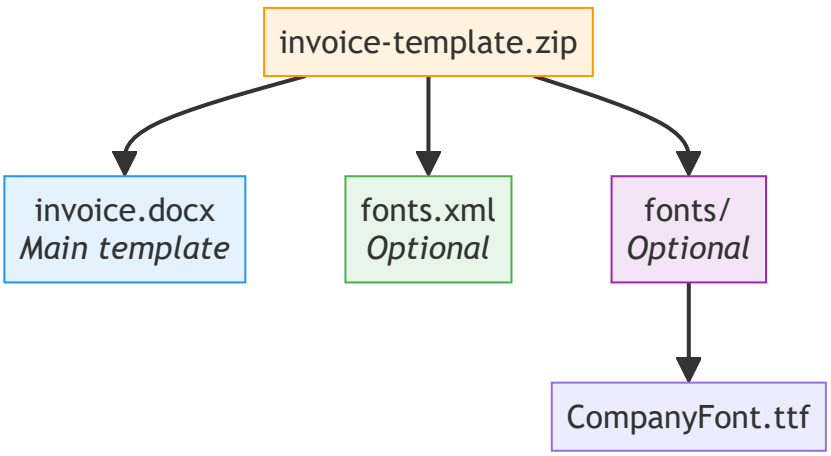
Table (formatted in Word with borders and header shading):

#	Description	Qty	Unit Price	Amount
${items.ordinal}	${items.description}	${items.quantity}	${items.unitPrice}	${items.amount}

Footer area:

```
                              Subtotal: ${subtotal}
                              Tax (23%): ${taxAmount}
                              TOTAL:     ${totalAmount}

Payment terms: ${paymentTerms}
This invoice contains ${items.size()} ${items.size() == 1 ? "position" : "positions"}.
```

2. Package:



3. Generation request:

`POST /templates/{templateId}/generate/pdf`

```
{
  "documentLocale": "pl_PL",
  "parameters": [
    {"name": "invoiceNumber", "value": "FV/2026/02/001"},
    {"name": "issueDate", "value": "2026-02-25"},
    {"name": "dueDate", "value": "2026-03-25"},
    {"name": "subtotal", "value": "10791.95"},
    {"name": "taxAmount", "value": "2482.15"},
    {"name": "totalAmount", "value": "13274.10"},
    {"name": "paymentTerms", "value": "Net 30 days"}
  ],
  "data": {
    "company": {
      "name": "ACME Sp. z o.o.",
      "taxId": "PL5261234567",
      "address": {
        "street": "Marszałkowska 1",
        "zipCode": "00-001",
        "city": "Warszawa"
      }
    },
    "client": {
      "name": "Beta Corp S.A.",
      "address": {
        "street": "Krakowska 15",
        "zipCode": "31-001",
        "city": "Kraków"
      }
    },
    "items": [
      {"ordinal": 1, "description": "Office Chair Ergonomic", "quantity": 5, "unitPrice": 899.99,
      ↪  "amount": 4499.95},
      {"ordinal": 2, "description": "Standing Desk Pro", "quantity": 3, "unitPrice": 1599.00, "amount":
      ↪  4797.00},
      {"ordinal": 3, "description": "Monitor Arm Dual", "quantity": 10, "unitPrice": 149.50, "amount":
      ↪  1495.00}
    ]
  },
  "pdfExportOptions": {
    "ownerPassword": "admin-secret",
    "canPrint": true,
    "canCopy": false,
    "canModify": false
  }
}
```

5. Font Embedding Substitution: If the generated PDF contains non-embedded base-14 fonts (Helvetica, Courier, Times) injected by Apache FOP, use `fontEmbeddingSubstitute` to replace them with an embedded TrueType font:

```
{
  "parameters": [...],
  "pdfExportOptions": {
```

```
    "fontEmbeddingSubstitute": "Arial"
  }
}
```

The specified font must be available — either bundled in the template package (via `fonts.xml`) or installed on the server.

6. Image Compression: If the generated PDF contains large lossless-encoded images (e.g., embedded photos or scans), use `imageCompressionQuality` to re-compress them as JPEG for significantly smaller file sizes:

```
{
  "parameters": [...],
  "pdfExportOptions": {
    "imageCompressionQuality": 0.85
  }
}
```

Quality ranges from 0.0 to 1.0 — **0.85** is recommended for print, **0.75** for mass printing. Only large colour RGB images are re-compressed; small icons, masks, and already-JPEG images are left untouched.

7. CMYK Conversion: For print-shop delivery requiring CMYK colour space, use `cmykConversionProfile` to convert RGB raster images to DeviceCMYK:

```
{
  "parameters": [...],
  "pdfExportOptions": {
    "cmykConversionProfile": "coated-fogra39"
  }
}
```

Available profiles: **coated-fogra39** (general commercial print), **pso-coated-v3** (modern offset), **pso-uncoated-v3** (uncoated paper). Only raster images are converted — vector graphics and text remain unchanged.

8. Transparency Flattening: To speed up print-shop RIP processing, use `flattenTransparency` to remove redundant transparency groups:

```
{
  "parameters": [...],
  "pdfExportOptions": {
    "flattenTransparency": true
  }
}
```

Pages that declare transparency but don't actually use it (no SMask, no BlendMode, no alpha) have the `/Group /Transparency` entry removed. Additionally, images with `/SMask` (alpha channels from PNG sources) are composited onto a white background and the soft mask is removed, producing a fully opaque PDF for print-shop RIPs.

9. Result: A professional PDF invoice with 3 line items, Polish number formatting (`4 499,95`), protected against copying and modification.

Troubleshooting

Placeholder Not Being Replaced

Symptom: The `${placeholder}` text appears literally in the output.

Cause: Word's spellchecker or proofing tools split the placeholder text into multiple XML fragments internally. The service has a run-merging step that fixes this in most cases, but it only works when all fragments share identical formatting.

Fix:

1. Select the entire `${placeholder}` text
2. Apply a uniform font, size, bold/italic setting
3. If this doesn't help, delete the placeholder and retype it in one go
4. Avoid applying formatting to part of a placeholder (e.g., making just the name bold)

Table Rows Not Replicating

Symptom: Only one row appears (with unresolved placeholders), instead of multiple data rows.

Checklist:

1. Placeholders must use dot notation: `${arrayName.fieldName}` — not just `${fieldName}`
2. The `arrayName` must match a key in the data that holds an array (list of objects)
3. Each field name must match a key in the array objects (case-sensitive)
4. The placeholder must be in a regular table row — not in a merged header cell

PDF Looks Different from Word

Common causes:

- Font not available on the server → bundle it in the ZIP
- Complex text wrapping or positioning → simplify to linear layout
- Synthetic bold (Word's "B" button without a bold font file) → provide actual bold `.ttf` file
- Default margins differ between Word and PDF renderer → set explicit margins

Numbers Not Formatted

Check:

1. Is `documentLocale` included in the generation request?
2. Is the value actually numeric? String values like `"1234.50"` in the parameters array are locale-formatted, but values wrapped in quotes within the data object must be numeric (not strings) to trigger formatting.

Quick Reference Card

Task	Syntax
Simple placeholder	`${name}`
Nested value	`${parent.child}`
Deep nesting	`${company.address.city}`
Table row field	`${arrayName.fieldName}`
Multiple tables	Use different array names per table
Ternary condition	`${condition ? "yes" : "no"}`
String comparison	`${"val".equals(field) ? "A" : "B"}`
List size check	`${items.size() > 3 ? "many" : "few"}`
Null fallback	`${field ?: "default"}`
Singular/plural	`${count == 1 ? "item" : "items"}`

Template Package	Required
`.docx` file	Yes (exactly one)
`fonts.xml`	Only if bundling custom fonts
`fonts/` directory	Only if bundling custom fonts
`.jrxml` file	No (presence switches to JASPER engine)

Output Format	URL Path
Word document	`POST /templates/{id}/generate/docx`
PDF document	`POST /templates/{id}/generate/pdf`
HTML (ZIP archive)	`POST /templates/{id}/generate/html`
Plain text	`POST /templates/{id}/generate/txt`
PDF with security	Add `"pdfExportOptions": {...}` to body
Locale formatting	Add `"documentLocale": "pl_PL"` to body