**Introduction**

In today’s data-driven world, managing and automating data flows efficiently is critical. Apache NiFi simplifies this process with its intuitive visual interface and powerful processors, but what truly unlocks its potential is the **NiFi Expression Language** (NFL). This language acts as the “glue” that connects static workflows to dynamic, intelligent data pipelines, allowing you to adapt to ever-changing data requirements without rewriting entire processes.

Imagine needing to **route** files based on their **names**, **filter** sensitive data by checking metadata, or dynamically **rename** files as they move through your system. Manually coding these behaviors would be time-consuming and rigid. Instead, the NiFi Expression Language empowers you to inject logic directly into your workflows using simple, expressive syntax.

**Why NiFi Expression Language**

The Expression Language is used heavily throughout the NiFi application for configuring **Processor** **properties**. Not all Processor properties support the Expression Language, however. Whether or not a Property supports the Expression Language is determined by the developer of the Processor when the Processor is written.  However, the application strives to clearly illustrate for each Property whether or not the Expression Language is supported.

This tutorial will guide you through the fundamentals of the **NiFi Expression Language**, starting with its core concepts and building up to practical use cases. By the end, you’ll be able to:

* Reference and manipulate **FlowFile attributes** (e.g., ${filename}, ${path}).
* Apply **functions** (e.g., string manipulation, math operations, date formatting).
* Build **conditional logic** (e.g., ${size > 1000000:toBoolean()}).
* Design smarter workflows that react to data in real time.

Whether you’re filtering logs, transforming JSON, or orchestrating multi-step data pipelines, mastering the NiFi Expression Language will transform how you work with data in NiFi. Let’s begin by exploring how this language turns static workflows into dynamic, intelligent systems!

**Where To Use Expression Language**

When configuring a component property, the User Interface provides an Information icon (*see image below*) next to the name of the Property. Hovering over this icon with the mouse will provide a tooltip that provides helpful information about the Property. This information includes:

* **Description** of the Property, the default value (if any)
* **Historically** configured values (if any)
* **Evaluation** scope of this property for expression language.

[A screenshot of a computer

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2025/04/nifi-processors-info-icons.png)

There are three values and the evaluation scope of the expression language is hierarchical:

**NONE, #VARIABLE\_REGISTRY, #FLOWFILE\_ATTRIBUTES.**

* **NONE** – expression language is not supported for this property
* **VARIABLE\_REGISTRY** is hierarchically constructed as below:
  + Variables defined at process group level and then, recursively, up to the higher process group until the root process group.
  + Variables defined in custom properties files by the administrator.
  + Environment variables defined at JVM level and system properties.
* **FLOWFILE\_ATTRIBUTES** – will use attributes of each individual flow file, as well as those variables defined by the Variable Registry, as described above.

**Expression Language Editor**

When configuring the **value** of a Processor property, the NiFi User Interface provides help with the Expression Language using the Expression Language editor. Once an Expression is begin by typing **${**, the editor begins to **highlight parentheses** and **braces** so that the user is easily able to tell which opening parenthesis or brace matches which closing parenthesis or brace.

The editor also supplies **context-sensitive** help by providing a list of **all functions** that can be used at the current cursor position. To activate this feature, press **Ctrl+Space** on the keyboard. The user is also able to type part of a function name and then press **Ctrl+Space** to see all functions that can be used that start with the same prefix.

For example, if we type into the editor **${filename:to** and then press **Ctrl+Space**, we are provided a pop-up that lists six different functions: **toDate, toLower, toNumber, toRadix, toString, and toUpper**. We can then continue typing to narrow which functions are shown, or we can select one of the functions from the list by double-clicking it with the mouse or using the arrow keys to highlight the desired function and pressing Enter.

The following figure shows the Expression Language editor.

[A screenshot of a computer

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2025/04/nifi-el-editor.png)

The following image shows the context-sensitive help (by pressing **Ctrl + Space**).

[A screenshot of a computer

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2025/04/nifi-el-intellisens.png)

**Expression Language Data Types**

Each **argument** to a function and each **value** returned from a function has a **specific data type**. The Expression Language supports **five** different data types:

* **String**: A String is a sequence of characters that can consist of numbers, letters, white space, and special characters.
* **Number**: A Number is an whole number comprised of one or more digits (0 through 9). When converting to numbers from Date data types, they are represented as the number of milliseconds since midnight GMT on January 1, 1970.
* **Decimal**: A Decimal is a numeric value that can support decimals and larger values with minimal loss of precision. More precisely it is a double-precision 64-bit IEEE 754 floating point. Due to this minimal loss of precision this data type should not be used for very precise values, such as currency. For more documentation on the range of values stored in this data type refer to this [link](https://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html#jls-4.2.3). The following are some examples of the forms of literal decimals that are supported in expression language (the “E” can also be lower-case):  
  – 1.1  
  – .1E1  
  – 1.11E-12
* **Date**: A Date is an object that holds a Date and Time. Utilizing the Date Manipulation and Type Coercion functions this data type can be converted to/from Strings and numbers. If the whole Expression Language expression is evaluated to be a date then it will be converted to a String with the format: ” :: “. Also expressed as “E MMM dd HH:mm:ss z yyyy” in Java  SimpleDateFormat format. For example: “Wed Dec 31 12:00:04 UTC 2016”.
* **Boolean**: A Boolean is one of either true or false.

After evaluating expression language functions, all attributes are stored as type **String**.

The Expression Language is generally able to **automatically coerce** a value of one data type to the appropriate data type for a function. However, functions do exist to manually coerce a value into a specific data type.

Hex values are supported for **Number** and **Decimal** types but they must be quoted and prepended with **“0x”** when being interpreted as literals. For example these two expressions are valid (without the quotes or **“0x”** the expression would fail to run properly):

* **${literal(“0xF“):toNumber()}**
* **${literal(“0xF.Fp10“):toDecimal()}**

**Structure of Expression Language**

The NiFi Expression Language (EL) is a lightweight, powerful syntax designed to **dynamically reference** FlowFile attributes, **manipulate** values, and **control** data routing in Apache NiFi. Its structure combines simplicity with flexibility, enabling users to embed logic directly into processor configurations without writing complex code.

The NiFi Expression Language always begins with the start delimiter **${** and ends with the end delimiter**}**. Between the start and end delimiters is the text of the Expression itself. In its most basic form, the Expression can consist of just an attribute name. For example, **${filename}** will return the **value** of the filename attribute.

In a slightly more complex example, we can instead return a manipulation of this value. We can, for example, return an all upper-case version of the filename by calling the toUpper function: **${filename:toUpper()}**. In this case, we reference the filename attribute and then manipulate this value by using the **toUpper** function.

**A Function Has Five Elements :**

1. There is the function call delimiter
2. There is the name of the function
3. Opened parenthesis
4. Function arguments. There would usually be function arguments right here, but the **toUpper** function has no arguments.
5. Closed parenthesis

[A diagram of a diagram

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2025/04/nifi-nel-structure.jpeg)

There are many different functions that are supported by the Expression Language to achieve many different goals. Some functions provide **String** (text) manipulation, such as the **toUpper** function. Others, such as the **equals** and **matches**functions, provide comparison functionality. Functions also exist for manipulating **dates** and **times** and for performing **mathematical** operations.

**Unleash the Power of Expression Language**

Now that you’ve explored the structure and capabilities of the NiFi Expression Language (EL), it’s time to put theory into action! In this hands-on section, you’ll experiment with real-world scenarios to solidify your understanding. You’ll learn how to:

* **Dynamically rename files** using timestamps and metadata.
* **Route data** based on conditions like file size, content type, or custom attributes.
* **Transform and enrich data** on the fly with string manipulation and arithmetic.
* **Troubleshoot expressions** using NiFi’s built-in EL preview tools.

Key rules:

* **Case-sensitive**: ${filename} ≠ ${FileName}.
* **Whitespace**: Avoid spaces unless inside quotes for strings.
* **Nesting**: Expressions can be nested (e.g., ${${sensor.type}:toLower()}).

Grab your NiFi instance, roll up your sleeves, and let’s dive into practical exercises that will turn abstract syntax into muscle memory. By the end, you’ll see firsthand how a few clever expressions can automate complex workflows—no custom code required! 🛠️

**Functions**

EL **Functions** provide a convenient way to **manipulate** and **compare** values of attributes. The Expression Language provides many different **functions** to meet the needs of a automated dataflow. Each function takes zero or more arguments and returns a single value. These functions can then be **chained** together to create powerful Expressions to **evaluate conditions** and **manipulate values**.

NiFi EL includes built-in functions to transform or **evaluate** data. Functions are chained using **Colon Punctuation ‘ : ‘.**

**Common Function Categories:**

| **Category** | **Example Functions** | **Usage** |
| --- | --- | --- |
| **String** | toUpper(), substring(1,5), replace('old','new') | Modify text: ${filename:toUpper()} |
| **Math** | plus(5), multiply(10), mod(3) | Calculate values: ${size:plus(100)} |
| **Date/Time** | now(), format('yyyy-MM-dd'), toDate() | Format timestamps: ${now():format('HH:mm')} |
| **Logical** | isNull(), not(), equals('test') | Evaluate conditions: ${size:gt(1000)} |

**Operators**

EL supports **logical**, **comparison**, and **arithmetic** operators:

| **Type** | **Operators** | **Example** |
| --- | --- | --- |
| **Comparison** | ==, !=, >, <, >=, <= | ${size > 1000000} (returns true/false) |
| **Logical** | and, or, not | ${size > 1000 and filename:contains('log')} |
| **Arithmetic** | +, -, \*, /, % | ${fileCount:plus(1)} |

**Attributes**

EL directly accesses **FlowFile attributes** (metadata) using their names.

**Example**:

| **Attribute** | **Description** |
| --- | --- |
| ${filename} | // Name of the original file |
| ${uuid} | // Unique identifier of the FlowFile |
| ${path} | // Directory path of the source file |
| ${mime.type} | // MIME type of the content (e.g., “text/csv”) |

**Handling Missing Attributes**:  
Use **default()** to avoid errors if an attribute is missing:

| **Attribute** | **Description** |
| --- | --- |
| ${nonExistentAttribute:default('unknown')} | // Returns “unknown” |

**Type Handling**

EL **implicitly** converts values to appropriate types (e.g., string, number, boolean). Use functions for **explicit** casting:

**Example**:

| **Attribute** | **Description** |
| --- | --- |
| ${timestamp:toNumber()} | // Convert to numeric (e.g., epoch time) |
| ${'100':toNumber()} | // String “100” → number 100 |
| ${size:gt(5000):toString()} | // Boolean → string “true” |

**Boolean Logic**

With EL, you are able to **compare** the value of an attribute against the value of another attribute. For example, the “**equals()**” function is used very widely as it determines if another String value is equal to the subject. In other words, it gives you a **direct comparison** of both String values. You will receive a Boolean response.

For example, if you wish to **find** out if a filename is called “**hello.txt**” you could check if the value of the filename attribute is **equal** to the value of the “hello.txt” string. Here is a quick examples of how Boolean logic may be applied.

| **Expression** | **Description** |
| --- | --- |
| ${filename:equals( 'hello.txt')} | // Check if the filename is equal to “hello.txt” |
| ${my\_attribute:equals( ${filename})} | // Check if the filename is equal to the value of  my\_attribute (attribute) |

**Working with Json**

EL provide functions to manipulate JSON and XML. If you reference JSONs with processors, then you must utilize the **JSONPath** **expression language**. Below is an example of a using **JsonPath** to evaluate Json attribues.

Let’s consider the following Json input file:

{

"name": "Emily Smith",

"address": {

"street": "456 Oak Rd",

"state": "NY",

"zipcode": "10001"

},

"phones": [

{

"type": "home",

"number": "555-1111"

},

{

"type": "office",

"number": "555-2222"

}

],

"children": [

{

"name": "Lucy",

"age": 12

},

{

"name": "Tom",

"age": 7

}

],

"spouse": null

}

**Example 1: Accessing a Simple Attribute**

In an **EvaluateJsonPath** processor, map **$.name** to an attribute like parent\_name.

| **Expression** | **Description** | **Result** |
| --- | --- | --- |
| $.name | Extract the parent’s name. | "Emily Smith" |

**Example 2: Accessing a Nested JSON Node**

Use **$.address.street** to populate an attribute like **street\_address**.

| **Expression** | **Description** | **Result** |
| --- | --- | --- |
| $.address.street | Extract the street from the address. | "456 Oak Rd" |

**Example 3: Accessing a JSON Array**

To evaluate Json arrays elements, use indexes (e.g **[0]**) .

| **Expression** | **Description** | **Result** |
| --- | --- | --- |
| $.phones[0].number | Returns a **Scalar** value. Extract the first phone number. | "555-2222" |

**Example 4: Filtering an Array**

To filter Json arrays elements, use filters (**[?(@.condition)]**).

| **Expression** | **Description** | **Destination Type** | **Result** |
| --- | --- | --- | --- |
| $.phones[?(@.type == 'office')].number | Returns a **Array**value. Extract the first phone number. | **flowfile-content** | ["555-2222"] |
| $.children[?(@.age > 8)].name | Returns an **Array**. Extract the office phone number. | **flowfile-content** | ["Lucy"] |

**Example 5: Accessing Attributes using jsonPath()**

You can also evaluate a Json Path and assign the result to an attribute using the **jsonPath()** function. Here we assume that the input Json is stored in an attribute called **myJson**. Following are some examples:

| **Expression** | **Description** | **Result** |
| --- | --- | --- |
| ${myJson:jsonPath('$.name')} | Extract the parent’s name. | "Emily Smith" |
| ${myJson:jsonPath('$.address.zipcode')} | Extract the zipcode. | “10001” |
| ${myJson:jsonPath('$.phones')} | Extract the phones node. | [ { “type”: “home”,  “number”: “555-1111”  },     {“type”: “office”,  “number”: “555-2222”   }  ] |

**Common Scenarios**

These examples show how you manipulate and transform attributes values which is indispensable for dynamic workflows in NiFi!These examples show how you manipulate and transform attributes values which is indispensable for dynamic workflows in NiFi!

These examples show how you manipulate and transform attributes values which is indispensable for dynamic workflows in NiFi!

* **Reformat a date**

| **Expression** | **Description** |
| --- | --- |
| ${string\_date:toDate("yyyy-MM-DD")} | // Convert a string to a date and apply a date format (year-month-day) |

* **Mathematical operations**

| **Expression** | **Description** |
| --- | --- |
| ${amount\_owed:minus(5)} | // Remove **5** from the value of amount\_owed |

* **Multi-variable: greater than**.

| **Expression** | **Description** |
| --- | --- |
| ${variable\_one:gt(${variable\_two})} | // Check if **variable\_one** is **greater than variable\_two** |

* **Adding a new directory to a path attribute**.

| **Expression** | **Description** |
| --- | --- |
| ${path:append('/new\_directory')} | // Append a path attribute with ‘new directory’ |

* **Adding a new directory to a path attribute**.

| **Expression** | **Description** |
| --- | --- |
| ${path:append('/new\_directory')} | // Append a path attribute with ‘new directory’ |

* **Dynamically Rename Files Using Timestamps and Metadata**

| **Expression** | **Description** |
| --- | --- |
| ${filename:substringBeforeLast('.')}\_${now():  format('yyyyMMdd-HHmmss')}\_user${user.id}.${filename:  substringAfterLast('.')} | * Extracts the filename without its extension (e.g., **report** from **report.txt**). * Adds a timestamp (e.g., **20231015-153045**) and a **user.id** attribute (e.g., **user123**). * Preserves the original file extension (e.g., **.txt**).   **Result**: report\_20231015-153045\_user123.txt |

* **Route Data Based on Conditions**

**Expression in RouteOnAttribute Processor**:

| **Expression** | **Description** |
| --- | --- |
| ${size:gt(1048576)} | * Route files larger than **1MB** to specific directories. |
| ${mime.type:equals('text/csv')} | * Route **CSV** files to specific directories. |
| **Configuration** | **Outcome** |
| * Create two relationships: **large\_files** and **csv\_files**. * Use the above expressions in the processor’s dynamic properties. | * Files larger than 1MB flow to **large\_files**. * CSV files flow to **csv\_files**. |

* **Transform and Enrich Data On the Fly**

**Use Case**: Convert product names to uppercase and calculate total price (quantity × price).

**Expression in UpdateAttribute Processor**:

| **Expression** | **Description** |
| --- | --- |
| product\_name\_upper = ${product.name:toUpper()} | Assuming the product name is “widget”. This function Transforms "widget" to "WIDGET". |
| total\_price = ${quantity:toNumber():multiply(${price:toNumber()})} | Multiplies quantity (e.g., 5) by price (e.g., 10) → 50 and store the result into **total\_price** |

**Advanced Features**

* **Ternary Operator**: Conditionally return values.

| **Expression** | **Description** |
| --- | --- |
| ${size > 1000000:ifElse('large', 'small')} | // Returns “large” or “small” |

* **Regular Expressions**: Extract or match patterns.

| **Expression** | **Description** |
| --- | --- |
| ${filename:matches('^log\_.\*')} | // Returns `true` if filename starts with “log\_” |
| ${filename:replaceAll('[0-9]', 'X')} | // Replace all digits with “X” |

* **Escaping**: Use $${ to treat ${ as a literal.

| **Expression** | **Description** |
| --- | --- |
| $${filename} | // Outputs “${filename}” instead of evaluating it |

**Summary**

This tutorial introduced the NiFi Expression Language (EL) and JSONPath as essential tools for building dynamic, metadata-driven workflows in Apache NiFi. You learned how EL enables real-time data manipulation—renaming files with timestamps, routing based on conditions, and transforming attributes—using simple syntax like **${attribute:function()}**. JSONPath integration demonstrated extracting nested or array-based JSON data (e.g., addresses, phone numbers) for precise control.

Combining these tools empowers you to automate complex tasks, eliminate hardcoding, and create adaptive pipelines that scale with evolving data needs. Mastery of EL and JSONPath unlocks efficient, maintainable workflows, turning raw data into actionable insights effortlessly.