

# Hortonworks DataFlow

## Apache NiFi RecordPath Guide

(January 31, 2018)

## Hortonworks DataFlow: Apache NiFi RecordPath Guide

Copyright © 2012-2018 Hortonworks, Inc. Some rights reserved.

Hortonworks DataFlow (HDF) is powered by Apache NiFi. A version of this documentation originally appeared on the [Apache NiFi website](#).

HDF is the first integrated platform that solves the real time challenges of collecting and transporting data from a multitude of sources and provides interactive command and control of live flows with full and automated data provenance. HDF is a single combined platform that provides the data acquisition, simple event processing, transport and delivery mechanism designed to accommodate the diverse dataflows generated by a world of connected people, systems and things.

Unlike other providers of platforms built using Apache Hadoop, Hortonworks contributes 100% of our code back to the Apache Software Foundation. Hortonworks DataFlow is Apache-licensed and completely open source. We sell only expert technical support, training and partner-enablement services. All of our technology is, and will remain free and open source.

Please visit the [Hortonworks](#) page for more information on Hortonworks technology. For more information on Hortonworks services, please visit either the [Support](#) or [Training](#) page. Feel free to [Contact Us](#) directly to discuss your specific needs.



Except where otherwise noted, this document is licensed under  
**Creative Commons Attribution ShareAlike 4.0 License.**  
<http://creativecommons.org/licenses/by-sa/4.0/legalcode>

# Table of Contents

1. Apache NiFi RecordPath Guide .....	1
1.1. Apache NiFi RecordPath Guide .....	1
1.1.1. Structure of a RecordPath .....	2
1.1.2. Child Operator .....	2
1.1.3. Descendant Operator .....	3
1.1.4. Filters .....	3
1.1.5. Function Usage .....	3
1.1.6. Functions .....	6
1.1.7. Standalone Functions .....	7
1.1.8. Filter Functions .....	11

# 1. Apache NiFi RecordPath Guide

## 1.1. Apache NiFi RecordPath Guide

One class of data that is often processed by NiFi is record-oriented data. When we say record-oriented data, we are often (but not always) talking about structured data such as JSON, CSV, and Avro. There are many other types of data that can also be represented as "records" or "messages," though. As a result, a set of Controller Services have been developed for parsing these different data formats and representing the data in a consistent way by using the RecordReader API. This allows data that has been written in any data format to be treated the same, so long as there is a RecordReader that is capable of producing a Record object that represents the data.

When we talk about a Record, this is an abstraction that allows us to treat data in the same way, regardless of the format that it is in. A Record is made up of one or more Fields. Each Field has a name and a Type associated with it. The Fields of a Record are described using a Record Schema. The Schema indicates which fields make up a specific type of Record. The Type of a Field will be one of the following:

- String
- Boolean
- Byte
- Character
- Short
- Integer
- Long
- BigInt
- Float
- Double
- Date - Represents a Date without a Time component
- Time - Represents a Time of Day without a Date component
- Timestamp - Represents a Date and Time
- Embedded Record - Hierarchical data, such as JSON, can be represented by allowing a field to be of Type Record itself.
- Choice - A field may be any one of several types.
- Array - All elements of an array have the same type.

- Map - All Map Keys are of type String. The Values are of the same type.

Once a stream of data has been converted into Records, the RecordWriter API allows us to then serialize those Records back into streams of bytes so that they can be passed onto other systems.

Of course, there's not much point in reading and writing this data if we aren't going to do something with the data in between. There are several processors that have already been developed for NiFi that provide some very powerful capabilities for routing, querying, and transforming Record-oriented data. Often times, in order to perform the desired function, a processor will need input from the user in order to determine which fields in a Record or which values in a Record should be operated on.

Enter the NiFi RecordPath language. RecordPath is intended to be a simple, easy-to-use Domain-Specific Language (DSL) to specify which fields in a Record we care about or want to access when configuring a processor.

### 1.1.1. Structure of a RecordPath

A Record in NiFi is made up of (potentially) many fields, and each of these fields could actually be itself a Record. This means that a Record can be thought of as having a hierarchical, or nested, structure. We talk about an "inner Record" as being the child of the "outer Record." The child of an inner Record, then, is a descendant of the outer-most Record. Similarly, we can refer to an outer Record as being an ancestor of an inner Record.

### 1.1.2. Child Operator

The RecordPath language is structured in such a way that we are able to easily reference fields of the outer-most Record, or fields of a child Record, or descendant Record. To accomplish this, we separate the names of the children with a slash character (/), which we refer to as the `child` operator. For example, let's assume that we have a Record that is made up of two fields: `name` and `details`. Also, assume that `details` is a field that is itself a Record and has two Fields: `identifier` and `address`. Further, let's consider that `address` is itself a Record that contains 5 fields: `number`, `street`, `city`, `state`, and `zip`. An example, written here in JSON for illustrative purposes may look like this:

```
{
  "name": "John Doe",
  "details": {
    "identifier": 100,
    "address": {
      "number": "123",
      "street": "5th Avenue",
      "city": "New York",
      "state": "NY",
      "zip": "10020"
    }
  }
}
```

We can reference the `zip` field by using the RecordPath: `/details/address/zip`. This tells us that we want to use the `details` field of the "root" Record. We then want to reference the `address` field of the child Record and the `zip` field of that Record.

### 1.1.3. Descendant Operator

In addition to providing an explicit path to reach the `zip` field, it may sometimes be useful to reference the `zip` field without knowing the full path. In such a case, we can use the descendant operator (`//`) instead of the child operator (`/`). To reach the same `zip` field as above, we can accomplish this by simply using the path `//zip`.

There is a very important distinction, though, between the child operator and the descendant operator: the descendant operator may match many fields, whereas the child operator will match at most one field. To help understand this, consider the following Record:

```
{
  "name": "John Doe",
  "workAddress": {
    "number": "123",
    "street": "5th Avenue",
    "city": "New York",
    "state": "NY",
    "zip": "10020"
  },
  "homeAddress": {
    "number": "456",
    "street": "116th Avenue",
    "city": "New York",
    "state": "NY",
    "zip": "11697"
  }
}
```

Now, if we use the RecordPath `/workAddress/zip`, we will be referencing the `zip` field that has a value of "10020." The RecordPath `/homeAddress/zip` will reference the `zip` field that has a value of "11697." However, the RecordPath `//zip` will reference both of these fields.

### 1.1.4. Filters

With the above examples and explanation, we are able to easily reference a specific field within a Record. However, in real scenarios, the data is rarely as simple as in the examples above. Often times, we need to filter out or refine which fields we are referencing. Examples of when we might want to do this are when we reference an Array field and want to only reference some of the elements in the array; when we reference a Map field and want to reference one or a few specific entries in the Map; or when we want to reference a Record only if it adheres to some criteria. We can accomplish this by providing our criteria to the RecordPath within square brackets (using the `[` and `]` characters). We will go over each of these cases below.

### 1.1.5. Function Usage

In addition to retrieving a field from a Record, as outlined above in the [Filters](#) section, we sometimes need to refine which fields we want to select. Or we may want to return a modified version of a field. To do this, we rely on functions. The syntax for a function is `<function name> <open parenthesis> <args> <close parenthesis>`, where `<args>` represents

one or more arguments separated by commas. An argument may be a string literal (such as 'hello') or a number literal (such as 48), or could be a relative or absolute RecordPath (such as ./name or /id). Additionally, we can use functions within a filter. For example, we could use a RecordPath such as /person[ isEmpty('name') ]/id to retrieve the id field of any person whose name is empty. A listing of functions that are available and their corresponding documentation can be found below in the [Functions](#) section.

### 1.1.5.1. Arrays

When we reference an Array field, the value of the field may be an array that contains several elements, but we may want only a few of those elements. For example, we may want to reference only the first element; only the last element; or perhaps the first, second, third, and last elements. We can reference a specific element simply by using the index of the element within square brackets (the index is 0-based). So let us consider a modified version of the Record above:

```
{
  "name": "John Doe",
  "addresses": [
    "work": {
      "number": "123",
      "street": "5th Avenue",
      "city": "New York",
      "state": "NY",
      "zip": "10020"
    },
    "home": {
      "number": "456",
      "street": "116th Avenue",
      "city": "New York",
      "state": "NY",
      "zip": "11697"
    }
  ]
}
```

We can now reference the first element in the `addresses` array by using the RecordPath `/addresses[0]`. We can access the second element using the RecordPath `/addresses[1]`. There may be times, though, that we don't know how many elements will exist in the array. So we can use negative indices to count backward from the end of the array. For example, we can access the last element as `/addresses[-1]` or the next-to-last element as `/addresses[-2]`. If we want to reference several elements, we can use a comma-separated list of elements, such as `/addresses[0, 1, 2, 3]`. Or, to access elements 0 through 8, we can use the range operator (`..`), as in `/addresses[0..8]`. We can also mix these, and reference all elements by using the syntax `/addresses[0..-1]` or even `/addresses[0, 1, 4, 6..-1]`. Of course, not all of the indices referenced here will match on the Record above, because the `addresses` array has only 2 elements. The indices that do not match will simply be skipped.

### 1.1.5.2. Maps

Similar to an Array field, a Map field may actually consist of several different values. RecordPath gives us the ability to select a set of values based on their keys. We do this by using a quoted String within square brackets. As an example, let's re-visit our original Record from above:

```
{
  "name": "John Doe",
  "details": {
    "identifier": 100,
    "address": {
      "number": "123",
      "street": "5th Avenue",
      "city": "New York",
      "state": "NY",
      "zip": "10020"
    }
  }
}
```

Now, though, let's consider that the Schema that is associated with the Record indicates that the `address` field is not a Record but rather a Map field. In this case, if we attempt to reference the `zip` using the RecordPath `/details/address/zip` the RecordPath will not match because the `address` field is not a Record and therefore does not have any Child Record named `zip`. Instead, it is a Map field with keys and values of type String. Unfortunately, when looking at JSON this may seem a bit confusing because JSON does not truly have a Type system. When we convert the JSON into a Record object in order to operate on the data, though, this distinction can be important.

In the case laid out above, we can still access the `zip` field using RecordPath. We must now use the a slightly different syntax, though: `/details/address['zip']`. This is telling the RecordPath that we want to access the `details` field at the highest level. We then want to access its `address` field. Since the `address` field is a Map field we can use square brackets to indicate that we want to specify a Map Key, and we can then specify the key in quotes.

Further, we can select more than one Map Key, using a comma-separated list: `/details/address['city', 'state', 'zip']`. We can also select all of the fields, if we want, using the Wildcard operator (`:`): `/details/address[ ]`. Map fields do not contain any sort of ordering, so it is not possible to reference the keys by numeric indices.

### 1.1.5.3. Predicates

Thus far, we have discussed two different types of filters. Each of them allows us to select one or more elements out from a field that allows for many values. Often times, though, we need to apply a filter that allows us to restrict which Record fields are selected. For example, what if we want to select the `zip` field but only for an `address` field where the state is not New York? The above examples do not give us any way to do this.

RecordPath provides the user the ability to specify a Predicate. A Predicate is simply a filter that can be applied to a field in order to determine whether or not the field should be included in the results. Like other filters, a Predicate is specified within square brackets. The syntax of the Predicate is `<Relative RecordPath> <Operator> <Expression>`. The `Relative RecordPath` works just like any other RecordPath but must start with a `.` (to reference the current field) or a `..` (to reference the current field's parent) instead of a slash and references fields relative to the field that the Predicate applies to. The `Operator` must be one of:

- Equals (`=`)
- Not Equal (`!=`)



- Greater Than (>)
- Greater Than or Equal To (>=)
- Less Than (<)
- Less Than or Equal To (#)

The `Expression` can be a literal value such as `50` or `Hello` or can be another `RecordPath`.

To illustrate this, let's take the following `Record` as an example:

```
{
  "name": "John Doe",
  "workAddress": {
    "number": "123",
    "street": "5th Avenue",
    "city": "New York",
    "state": "NY",
    "zip": "10020"
  },
  "homeAddress": {
    "number": "456",
    "street": "Grand St",
    "city": "Jersey City",
    "state": "NJ",
    "zip": "07304"
  },
  "details": {
    "position": "Dataflow Engineer",
    "preferredState": "NY"
  }
}
```

Now we can use a `Predicate` to choose only the fields where the state is not New York. For example, we can use `/[./state != 'NY']`. **This will select any Record field that has a state field if the state does not have a value of "NY". Note that the details Record will not be returned because it does not have a field named state. So in this example, the RecordPath will select only the homeAddress field. Once we have selected that field, we can continue on with our RecordPath. As we stated above, we can select the zip field: `/[./state != 'NY']/zip`.** This `RecordPath` will result in selecting the `zip` field only from the `homeAddress` field.

We can also compare the value in one field with the value in another field. For example, we can select the address that is in the person's preferred state by using the `RecordPath` `/*[./state = /details/preferredState]`. In this example, this `RecordPath` will retrieve the `workAddress` field because its `state` field matches the value of the `preferredState` field.

Additionally, we can write a `RecordPath` that references the "city" field of any record whose state is "NJ" by using the parent operator (`..`): `/*[./state = 'NJ']/city[../state = 'NJ']`.

## 1.1.6. Functions

In the [Function Usage](#) section above, we describe how and why to use a function in `RecordPath`. Here, we will describe the different functions that are available, what they

do, and how they work. Functions can be divided into two groups: **Standalone Functions**, which can be the 'root' of a RecordPath, such as `substringAfter( /name, ' ' )` and **Filter Functions**, which are to be used as a filter, such as `/name[ contains('John') ]`. A Standalone Function can also be used within a filter but does not return a `boolean` (true or false value) and therefore cannot itself be an entire filter. For example, we can use a path such as `/name[ substringAfter(., ' ') = 'Doe' ]` but we cannot simply use `/name[ substringAfter(., ' ') ]` because doing so doesn't really make sense, as filters must be boolean values.

Unless otherwise noted, all of the examples below are written to operate on the following Record:

```
{
  "name": "John Doe",
  "workAddress": {
    "number": "123",
    "street": "5th Avenue",
    "city": "New York",
    "state": "NY",
    "zip": "10020"
  },
  "homeAddress": {
    "number": "456",
    "street": "Grand St",
    "city": "Jersey City",
    "state": "NJ",
    "zip": "07304"
  },
  "details": {
    "position": "Dataflow Engineer",
    "preferredState": "NY",
    "employer": "",
    "vehicle": null,
    "phrase": " "
  }
}
```

## 1.1.7. Standalone Functions

### 1.1.7.1. substring

The `substring` function returns a portion of a String value. The function requires 3 arguments: The value to take a portion of, the 0-based start index (inclusive), and the 0-based end index (exclusive). The start index and end index can be 0 to indicate the first character of a String, a positive integer to indicate the *n*th index into the string, or a negative integer. If the value is a negative integer, say `-n`, then this represents the *n*<sup>th</sup> character for the end. A value of `-1` indicates the last character in the String. So, for example, `substring( 'hello world', 0, -1 )` means to take the string `hello`, and return characters 0 through the last character, so the return value will be `hello world`.

RecordPath	Return value
<code>substring( /name, 0, -1 )</code>	John Doe
<code>substring( /name, 0, -5 )</code>	John
<code>substring( /name, 1000, 1005 )</code>	<empty string>

<code>substring( /name, 0, 1005)</code>	John Doe
<code>substring( /name, -50, -1)</code>	<empty string>

### 1.1.7.2. substringAfter

Returns the portion of a String value that occurs after the first occurrence of some other value.

RecordPath	Return value
<code>substringAfter( /name, '' )</code>	Doe
<code>substringAfter( /name, 'o' )</code>	hn Doe
<code>substringAfter( /name, ' ' )</code>	John Doe
<code>substringAfter( /name, 'xyz' )</code>	John Doe

### 1.1.7.3. substringAfterLast

Returns the portion of a String value that occurs after the last occurrence of some other value.

RecordPath	Return value
<code>substringAfterLast( /name, '' )</code>	Doe
<code>substringAfterLast( /name, 'o' )</code>	e
<code>substringAfterLast( /name, ' ' )</code>	John Doe
<code>substringAfterLast( /name, 'xyz' )</code>	John Doe

### 1.1.7.4. substringBefore

Returns the portion of a String value that occurs before the first occurrence of some other value.

RecordPath	Return value
<code>substringBefore( /name, '' )</code>	John
<code>substringBefore( /name, 'o' )</code>	J
<code>substringBefore( /name, ' ' )</code>	John Doe
<code>substringBefore( /name, 'xyz' )</code>	John Doe

### 1.1.7.5. substringBeforeLast

Returns the portion of a String value that occurs before the last occurrence of some other value.

RecordPath	Return value
<code>substringBeforeLast( /name, '' )</code>	John
<code>substringBeforeLast( /name, 'o' )</code>	John D
<code>substringBeforeLast( /name, ' ' )</code>	John Doe
<code>substringBeforeLast( /name, 'xyz' )</code>	John Doe

### 1.1.7.6. replace

Replaces all occurrences of a String with another String.

RecordPath	Return value
<code>replace( /name, 'o', 'x' )</code>	Jxhn Dxe
<code>replace( /name, 'o', 'xyz' )</code>	Jxyzhn Dxyze
<code>replace( /name, 'xyz', 'zyx' )</code>	John Doe
<code>replace( /name, 'Doe', /workAddress/city )</code>	John New York

### 1.1.7.7. replaceRegex

Evaluates a Regular Expression against the contents of a String value and replaces any match with another value. This function requires 3 arguments: the String to run the regular expression against, the regular expression to run, and the replacement value. The replacement value may optionally use back-references, such as `$1` and `${named_group}`

RecordPath	Return value
<code>replaceRegex( /name, 'o', 'x' )</code>	Jxhn Dxe
<code>replaceRegex( /name, 'o', 'xyz' )</code>	Jxyzhn Dxyze
<code>replaceRegex( /name, 'xyz', 'zyx' )</code>	John Doe
<code>replaceRegex( /name, '\s+.*', /workAddress/city )</code>	John New York
<code>replaceRegex(/name, '([JD])', '\$1x')</code>	Jxohn Dxoe
<code>replaceRegex(/name, '(?&lt;hello&gt;[JD])', '\${hello}x')</code>	Jxohn Dxoe

### 1.1.7.8. concat

Concatenates all the arguments together.

RecordPath	Return value
<code>concat( /name, ' lives in ', /homeAddress/city )</code>	John Doe lives in Jersey City

### 1.1.7.9. fieldName

Normally, when a path is given to a particular field in a Record, what is returned is the value of that field. It can sometimes be useful, however, to obtain the name of the field instead of the value. To do this, we can use the `fieldName` function.

RecordPath	Return value
<code>fieldName(//city/..)</code>	workAddress and homeAddress
<code>//city[not(startsWith(fieldName(..), 'work'))]</code>	Jersey City

In the above example, the first RecordPath returns two separate field names: "workAddress" and "homeAddress". The second RecordPath, in contrast, returns the value of a "city" field and uses the `fieldName` function as a predicate. The second RecordPath finds a "city" field whose parent does not have a name that begins with "work". This means that it will return the value of the "city" field whose parent is "homeAddress" but not the value of the "city" field whose parent is "workAddress".

### 1.1.7.10. toDate

Converts a String to a date. For example, given a schema such as:

```
{
  "type": "record",
  "name": "events",
  "fields": [
    { "name": "name", "type": "string" },
    { "name": "eventDate", "type": "string" }
  ]
}
```

and a record such as:

```
{
  "name" : "My Event",
  "eventDate" : "2017-10-20T'11:00:00'Z"
}
```

The following record path would parse the eventDate field into a Date:

```
toDate( /eventDate, "yyyy-MM-dd'T'HH:mm:ss'Z'")
```

### 1.1.7.11. format

Converts a Date to a String in the given format.

The first argument to this function must be a Date or a Number, and the second argument must be a format String that follows the Java SimpleDateFormat.

For example, given a schema such as:

```
{
  "type": "record",
  "name": "events",
  "fields": [
    { "name": "name", "type": "string" },
    { "name": "eventDate", "type": { "type": "long", "logicalType": "timestamp-millis" } }
  ]
}
```

and a record such as:

```
{
  "name" : "My Event",
  "eventDate" : 1508457600000
}
```

The following record path expressions would format the date as a String:

RecordPath	Return value
<code>format( /eventDate, "yyyy-MM-dd'T'HH:mm:ss'Z'")</code>	2017-10-20T'11:00:00'Z'
<code>format( /eventDate, "yyyy-MM-dd")</code>	2017-10-20

In the case where the field is declared as a String, the toDate function must be called before formatting.

For example, given a schema such as:

```
{
  "type": "record",
  "name": "events",
  "fields": [
    { "name": "name", "type": "string" },
    { "name": "eventDate", "type": "string" }
  ]
}
```

and a record such as:

```
{
  "name" : "My Event",
  "eventDate" : "2017-10-20'T'11:00:00'Z'"
}
```

The following record path expression would re-format the date String:

RecordPath	Return value
<code>format( toDate(/eventDate, "yyyy-MM-dd'T'HH:mm:ss'Z'"), 'yyyy-MM-dd')</code>	2017-10-20

## 1.1.8. Filter Functions

### 1.1.8.1. contains

Returns `true` if a String value contains the provided substring, `false` otherwise

RecordPath	Return value
<code>/name[contains(., 'o')]</code>	John Doe
<code>/name[contains(., 'x')]</code>	<returns no results>
<code>/name[contains( ../workAddress/state, /details/preferredState )]</code>	John Doe

### 1.1.8.2. matchesRegex

Evaluates a Regular Expression against the contents of a String value and returns `true` if the Regular Expression exactly matches the String value, `false` otherwise. This function requires 2 arguments: the String to run the regular expression against, and the regular expression to run.

RecordPath	Return value
<code>/name[matchesRegex(., 'John Doe')]</code>	John Doe
<code>/name[matchesRegex(., 'John')]</code>	<returns no results>
<code>/name[matchesRegex(., '.* Doe' )]</code>	John Doe

### 1.1.8.3. startsWith

Returns `true` if a String value starts with the provided substring, `false` otherwise

RecordPath	Return value
<code>/name[startsWith(., 'J')]</code>	John Doe
<code>/name[startsWith(., 'x')]</code>	<returns no results>

/name[startsWith(., 'xyz')]	<returns no results>
/name[startsWith(., '')]	John Doe

### 1.1.8.4. endsWith

Returns `true` if a `String` value ends with the provided substring, `false` otherwise

RecordPath	Return value
/name[endsWith(., 'e')]	John Doe
/name[endsWith(., 'x')]	<returns no results>
/name[endsWith(., 'xyz')]	<returns no results>
/name[endsWith(., '')]	John Doe

### 1.1.8.5. not

Inverts the value of the function or expression that is passed into the `not` function.

RecordPath	Return value
/name[not(endsWith(., 'x'))]	John Doe
/name[not(contains(., 'x'))]	John Doe
/name[not(endsWith(., 'e'))]	<returns no results>

### 1.1.8.6. isEmpty

Returns `true` if the provided value is either null or is an empty string.

RecordPath	Return value
/name[isEmpty(/details/employer)]	John Doe
/name[isEmpty(/details/vehicle)]	John Doe
/name[isEmpty(/details/phase)]	<returns no results>
/name[isEmpty(.)]	<returns no results>

### 1.1.8.7. isBlank

Returns `true` if the provided value is either null or is an empty string or a string that consists only of white space (spaces, tabs, carriage returns, and new-line characters).

RecordPath	Return value
/name[isBlank(/details/employer)]	John Doe
/name[isBlank(/details/vehicle)]	John Doe
/name[isBlank(/details/phase)]	John Doe
/name[isBlank(.)]	<returns no results>

Last updated 2018-01-09 02:06:38 -07:00