

PreJSON docs



Table of contents:

- [Intro](#)
- [Basics of use](#)
 - [Expansion](#)
 - [Resolving parameter values](#)
 - [Diagnostics](#)
 - [PreJSON context](#)
 - [One-time use](#)
 - [Use with context](#)
 - [Context inheritance](#)
 - [Context options](#)
 - [Features](#)
 - [Parse](#)
 - [Format](#)
- [Syntax Features](#)
 - [Single-quoted strings](#)
 - [JavaScript number notation](#)
 - [Unquoted & Numeric keys](#)
 - [Trailing commas](#)
 - [Comments](#)
 - [Template literals](#)
 - [Time literals](#)
- [Parameters](#)
 - [Declaring and Using a Parameter](#)
 - [Default Value](#)
 - [Surrounded Parameter](#) ❌
 - [Types](#)
 - [Modifiers](#)
 - [Property access](#)
 - [Type hints & Casts](#)
 - [Syntax & usage examples](#)
 - [Type conversion table](#)
 - [Spreading](#)
 - [Spread operator rules](#)
 - [Syntax & usage examples](#)
- [Macros](#)

- [Syntax](#)
- [Examples](#)
- [PreJSON Monaco](#)
 - [Usage](#)
 - [Registering the PreJSON extension](#)
 - [Mode configuration](#)
 - [Associating a PreJSON context](#)
 - [Associating a JSON schema](#)
- [Examples](#)
- [Notation](#)
 - [Datetime](#)
 - [Fixed Datetime](#)
 - [Relative Datetime](#)
 - [Base now](#)
 - [Offset](#)
 - [Rounding](#)
 - [Date Range](#)
 - [Fixed Date Range](#)
 - [Relative Date Range](#)
- [Basics of use](#)
 - [Instantiating a time](#)
 - [Error handling](#)
 - [Expansion](#)

Intro

Basics of use

When a PreJSON string is parsed, the corresponding tree structure is created and certain semantic rules are checked, such as whether the values specified for the parameters are valid with respect to their type. The resulting PreJSON instance can be converted directly to JSON, but if it is parametric, its parameters would be replaced by `null` in the JSON.

To create a JSON where parameters are replaced by specified values, we need to **expand** the PreJSON first.

Expansion

The primary goal of the expansion is to create a new PreJSON instance by

- replacing [parameters](#) with their inline [default values](#), explicit values provided at the time of the expansion, or values specified in [the context declarations](#),
- resolving macros to the effective segment,
- and [expansion of time literals](#), which means that any relative datetime or floating date range will be evaluated according to the moment of the expansion.

Let's have the following PreJSON input and pretend it's New Year Midnight of 2023:

```
{
  time: now,
  city: ${string:city}
}
```

If we try to convert the PreJSON to a JSON without expansion, we'll get:

```
{
  time: "now",
  city: null
}
```

Notice that the relative datetime `now` is converted to a string only, and any parameters are replaced by `null`. While if we expand the PreJSON providing the value `"Prague"` for the parameter `city`, the

resulting JSON will change to:

```
{
  time: "2023-01-01T00:00:00.000Z",
  city: "Prague"
}
```

Resolving parameter values

As previously stated, values that may replace parameters during expansion are sourced from various origins, these are:

1. *explicit values* - passed by user as an argument of the expansion
2. *inline values* - values specified as [defaults](#) directly in the parsed string from which the instance was created, e.g. `${string:foo:'bar'}`
3. *context values* - values provided within [the context declarations](#)

The value for a parameter can be provided by more than one source. In such a case, PreJSON will select the one from the source with the highest priority. By default, priorities are given in the same order as listed above, i.e. explicit values have the highest priority.

CUSTOM PRIORITIES

Users can change the priority of each value source, either at the context level (specifying [expansion.valuePriority](#) option) or [explicitly during expansion](#). In the case of context in particular, be aware that changing priorities can significantly affect the behaviour of the application using PreJSON.

Diagnostics

PreJSON is designed not to throw exceptions if the parsed input is not valid. Even an invalid string will result in a PreJSON instance that logs any problems encountered during tokenization, parsing, or semantic checking. This log is intentionally not called *errors* because diagnostics may also report use of a deprecated feature or provide hints.

However, if the user attempts to expand, convert to a string, or convert to a JSON an invalid PreJSON, an exception will be thrown. Therefore, it is up to the user to always verify that the PreJSON is valid

using the available methods before its further processing.

PreJSON context

Each instance of PreJSON is created within a context that specifies options, such as formatting or optional features, or provides parameter declarations that are intended to be shared across multiple instances of PreJSON.

Even if you don't create your own `PreJSONContext` instance, PreJSON will use the default built-in one. This is useful if you only need to parse a single PreJSON without any specific options as in the following example.

One-time use

```
import { PreJSON } from '@sbks/prejson'

const prejson = PreJSON.parse('{key: ${string:str:"Foobar"}}')

// Checks if diagnostics don't contain any errors
if (prejson.isValid) {
  // Returns {"key": "Foobar"}
  prejson.expand().toJSON()

  // Returns {"key": "Lorem ipsum"}
  prejson.expand({ str: 'Lorem ipsum' }).toJSON()
}
```

The static method `PreJSON.parse` creates a PreJSON instance within the default context. Notice the second expansion, where a parameter value map is provided, overriding the default value `"Foobar"` of the parameter `str`.

Use with context

When using PreJSON, you will encounter a situation where you want to parse multiple input strings under the same conditions - typically with a set of shared parameters. An example might be to create multiple requests, each fetching different data, but always filtered for the same time period. Like this:

```
{
  "filter": [
    {
      "type": "creation_date",
      "from": "2023-01-01",
      "to": "2023-03-01"
    }
  ],
  "metric": "views"
}
```

The context in this example should therefore declare a parameter for the time period.

```
import { PreJSONContext, PreJSONType } from '@sbks/prejson'

const context = new PreJSONContext()
context.declare('range', PreJSONType.DateRange, '2022/2023')
```

Any PreJSON string parsed within this context can use the `range` parameter without having to specify its type or value. At the same time, it can still contain additional parameters, but they must be declared in the string.

```
const prejson = context.parse(`
{
  filter: [
    {
      type: "creation_date",
      from: \${range | start},
      to: \${range | end}
    }
  ],
  metric: \${string:metric}
}
`)
```

To expand this PreJSON instance `prejson`, a value is required for *metric* parameter (otherwise results in `null`), while `range` is obtained from the context.

```
prejson.expand({ metric: 'views' }).toJSON()
```


In this line, we expand the `prejson` instance by replacing the parameter *metric* by the string `"views"` and expanding the *range* parameter with modifiers to the start and the end datetime of the range provided in the context. So the resulting JSON will look like this:

```
{
  "filter": [
    {
      "type": "creation_date",
      "from": "2022-01-01T00:00:00.000Z",
      "to": "2023-01-01T00:00:00.000Z"
    }
  ],
  "metric": "views"
}
```

We can reuse the `prejson` instance freely to generate other JSONs with different metrics. Of course, when passing values for expansion, we can also override the value specified for the *range* parameter in the context.

Context inheritance

Suppose you need to extend the existing PreJSON context, but want to preserve it. When you create a new instance of the context, pass the existing instance whose properties you want to inherit as the second argument. If the same properties or parameters are defined directly by the new instance, they will be overridden.

In the following example, `context2` inherits the declaration of the `p1` parameter from `context1`.

```
import { PreJSONContext, PreJSONType } from '@sbks/prejson'

const context1 = new PreJSONContext()
context1.declare('p1', PreJSONType.String, 'Lorem')

const context2 = new PreJSONContext({}, context1)
context2.declare('p2', PreJSONType.Number, 100)


// returns '{"Lorem": 100}'
context2.parse('${p1}: ${p2}').expand().toJSON()
// throws an error because parameter 'p2' is not declared in 'context1'
context1.parse('${p1}: ${p2}').expand().toJSON()
```

Context options

Option	Type	
<code>debug</code>	<code>boolean</code>	If <code>true</code> , PreJSON outputs all diagnostics to the console.

Features

Allows you to disable support for specific features.

 TIP

By default, all features are enabled, so you won't need this option unless you want to suppress one of them.

Option	Type	
<code>parameters</code>	<code>boolean</code>	Adds support for parameters . If disabled, PreJSON reports an error whenever a parameter notation appears outside a string. (The parameter notation inside a string is not recognized and is treated as plain characters.)
<code>comments</code>	<code>boolean</code>	Adds support for both single and multi-line comments.
<code>timeLiterals</code>	<code>boolean</code>	Adds support for datetime and date range literals .
<code>macros</code>	<code>boolean</code>	Adds support for macros .

Parse

Specifies default options for parsing PreJSONs. These options are also accepted as an argument for the `PreJSON.parse` and `PreJSONContext#parse` methods.

Option	Type	
<code>allowUndeclaredParameters</code>	<code>boolean</code>	If <code>true</code> , undeclared parameters are not considered to

Option	Type	
		be an error and are expanded as <code>null</code> .

Format

Specifies default options for transforming of a PreJSON to a string. These options are also accepted as an argument for the `PreJSON#format` and `PreJSON.format` methods.

Option	Type	
<code>minify</code>	<code>boolean</code>	If <code>true</code> , all other format options are ignored and a minimal string representing the PreJSON is produced.
<code>indent</code>	<code>string</code>	String used as the indentation.
<code>trailingComma</code>	<code>boolean</code>	If <code>true</code> , adds extra comma after the last array item or object entry.
<code>quotedKeys</code>	<code>'always' 'needed'</code>	If <code>"always"</code> , double-quotes all the object keys. If <code>"needed"</code> , double-quotes only keys that don't match JS identifier notation and would otherwise cause a syntax error.

Syntax Features

Overview of what PreJSON offers compared to JSON.

! INFO

PreJSON is a superset of JSON and so any valid JSON is also a valid PreJSON.

Single-quoted strings

In addition to the standard double quotes, single quotes can also be used to delimit a string.

```
{  
  // ✅  
  "city": "Prague",  
  'country': 'CZ'  
  
  // ❌ Mismatching single and double quotes  
  'continent': "Europe"  
}
```

JavaScript number notation

Standard JSON does not support the same number notation as JavaScript. For example, a truncated decimal `.5` or a plus-signed number `+1` are considered invalid. PreJSON supports these alternatives as well.

```
[  
  // ✅  
  .5,  
  +.1e-1,  
  -1  
]
```

Unquoted & Numeric keys

Object key names don't need to be quoted unless absolutely necessary, i.e. to satisfy JavaScript identifier constraints. In addition, numeric values can also be used as keys. [Parameters](#) can also appear as keys, but only if their resulting value after expansion is either a number or a string.

```
{
  // ✅ The key satisfies JS identifier constraints
  city: "Prague",
  $hash: "4a5f5e6c5",

  // ✅ Quoted if does not satisfy
  "country-code": "CZ",
  "12em": false,

  // ✅ Numeric key
  0.5: false,
  100: true,
  .4e4: false,
  +1: true,
  -2: false,

  // ❌
  country-code: "CZ",
  100/5: "CZ",

  // ✅ Parameters expanding to a number/string
  ${number:x}: false,
  ${number:x|asString}: false,
  ${string:str}: false,
  ${datetime:dt|toISO}: false,

  // ❌
  ${datetime:dt}: false,
  ${boolean:b}: false,
  ${number:x|gt(10)}: false,
}
```

Trailing commas

Trailing commas in objects and arrays are accepted as same as in the plain JavaScript. Unlike JavaScript, PreJSON does not support multiple trailing commas that create a [sparse array](#) to prevent

easily overlooked unwanted behavior.

```
// ✅  
{ city: "Prague", country: "CZ", }  
[ 0, 1, 1, 2, 3, 5, 8, 13, 21, ]  
  
// ❌ Sparse arrays not allowed  
[ 0, 1, 1, 2, , 5, 8, 13, , ]
```

Comments

Both line and multiline comments are supported anywhere in a PreJSON.

```
// ✅  
{ /* 2 */ city: /* 3 */ "Prague", // 4  
  /* 5 */ country /* 6 */: "CZ" /* 7 */  
  // 8  
}  
// 9
```



TIP

Comments support can be disabled by passing `{features: {comments: false}}` option to [the PreJSON context](#).

Template literals

Like JavaScript, template literals allow string interpolation, except that in PreJSON, only [parameters](#) appear in placeholders. Another advantage over JSON is that template literals are a convenient way to specify multiline strings.

When parameters used in a template literal are expanded, their value is automatically converted to a string. Therefore, they can be safely used as object keys without any parameter type restrictions.

```
// ✅ Used for a plain string  
{ city: `Prague` }
```

```
// ✅ Despite the parameter type, the key is a string
// because the parameter is inside the template literal
{ `${datetime:x}`: true }

// ✅ Multiline
{ content: `
- Item 1
- ${number:x}
- Item 3
` }
```

Time literals

PreJSON extends the standard type literals with two new ones – datetime and daterange. These types follow the syntax specified by [PreJSON Time](#) package and can be used as a value (not as an object key).

```
{
  // ✅
  time1: now,
  time2: now[sD],
  time3: now[sD]-T25M,
  time4: 2020-01-01,
  time4: 2020-01-01T12:35,
  range1: 2020/now,
  range2: P10W/now[sW],
  range3: 2020/2021,

  // ❌ Not as a key
  now: true,
  2020-01-01: true,
  now/now[eD]: true,
}
```



TIP

Time literals can be disabled by passing `{features: {timeLiterals: false}}` option to [the PreJSON context](#).

Parameters

A PreJSON that contains parameters can produce a completely different JSON object each time it is expanded, depending on what values were provided for each parameter at the time of expansion.

Declaring and Using a Parameter

Parameters can be declared using syntax `${type:identifier}` at almost any position in a PreJSON.

! INFO

A parameter used as an object key must be of the string or number type or must use such [modifiers](#) to ensure that it is expanded to values of that type.

```
// ✓  
${string:x}  
{ key: ${string:x} }  
{ ${string:x}: true }  
  
// ✗  
{ ${number:x} foo: "bar" }  
{ ${array:x}: "bar" }
```

Once a parameter is declared, it can be used in other parts of the object without specifying the type (optional). Note that it doesn't really matter in what order the parameters are used and declared.

```
[  
  ${x},  
  ${string:x},  
  ${x},  
  ${string:x},  
  // Throws type mismatch since `x` has been already specified as a string at line 3  
  ${number:x},  
]
```

Parameter declaration behaves like any other parameter occurrence, and so, when a PreJSON is being expanded, the declarations are expanded as well.

Default Value

The examples above would require the user to set the value of the parameter `x`, and would expand the parameter as `null` until the user did so. To avoid this behavior, the parameter declaration can be extended with a default value that matches the type.

```
[
  ${x},
  ${string:x:"Lorem ipsum"}
]
```

Default values are only accepted when using syntax that includes type. If the default value occurs in multiple places in the JSON, only the first occurrence is used.

Surrounded Parameter ❌

⚠️ DEPRECATED

Using parameters inside a standard string is deprecated. Use [template literals](#) to place parameters in strings.

All of the examples so far have used parameters to represent a key or value in the resulting JSON – the `${..}` syntax hasn't been surrounded by any other content. However, PreJSON supports placing parameters inside a string. The following example

```
{
  "key0": "${string:name:'Omni'}",
  "key1": "${number:num:10}",
  "key2": "${name} responded to ${num} requests."
}
```

expands to

```
{
  "key0": "Omni",
  "key1": 10,
  "key2": "Omni responded to 10 requests."
}
```

As you can see in the example

- If the parameter syntax is surrounded by other content, its value is converted to a string,
- Otherwise, the value expands to a JSON value, i.e., boolean, string, number, or `null`.

Types

Type	Default value syntax
<code>boolean</code>	<code>true</code> or <code>false</code>
<code>string</code>	a single/double-quoted string
<code>number</code>	any valid number literals
<code>datetime</code>	PreJSON Datetime notation or a string
<code>daterange</code>	PreJSON Date range notation or a string
<code>array</code>	array whose elements are of any supported parameter type
<code>object</code>	object whose values are of any supported parameter type

Modifiers

Sometimes it is necessary to modify the value of a parameter before its expansion. A typical example is a date range, where users need to access only one part of the range rather than the entire range.

Modifiers are used in a pipeline that is appended to the parameter notation. PreJSON checks for type continuity in the pipeline, and if the output of one modifier is incompatible with the expected input of the next modifier, it reports an error.

```
[
  ${number:x|plus(3)|div(2)|mod(3)},
  ${datetime:day|plus(1, 'month')|startOf('year')},
  ${number:y|plus(3)|replace('10', '20')},
```

```
    ${daterange:dr|end},  
  ]
```

LEARN MORE

See the [Modifier reference](#) for a detailed list of built-in modifiers categorized by the expected input type.

Property access

When working with parameters whose values are complex structures such as nested arrays and objects, it is often necessary to traverse such a structure to find the desired value. That's why PreJSON allows parameters of type object or array to use dot-bracket notation to access deeper into the parameter value.

In the following prejson, the first occurrence of the `o` parameter expands to the object `{x: [10, 30]}` but the second only accesses a specific value of the structure and expands to `30`.

```
[  
  ${object:o:{x: [10, 30]}},  
  ${o.x[1]}  
]
```

INFO

Property access can only be used without specifying the default value. `${object:o:{x: 1}.x}` is not a valid PreJSON syntax.

Type hints & Casts

Property access has a shortcoming – PreJSON cannot identify the type of the value accessed by the dot-bracket notation.

This is not a problem if we only need to expand a nested value without further processing (``Count: ${o.count}``). However, if we want to use modifiers, it's not possible because **PreJSON cannot ensure type continuity**.

```
// split cannot identify type of the input
${o.word | split("")}
```

For this reason, users are expected to provide a hint to ensure that PreJSON will be able to process the parameter appropriately. The type hint determines the resulting type of the parameter value regardless of its actual type. If they do not match, PreJSON implicitly converts the value, so **the hint works also as a cast if needed**.

Updating the previous example, the second expansion will result in `"30"` instead of `30`.

```
[
  ${object:o:{x: [10, 30]}},
  ${((string) o.x[1])}
]
```

Once the property access includes a type hint, the modifier pipeline can be used safely.

! INFO

Type hinting and casting can only be used with property access. PreJSON does not allow to cast parameter values themselves.

Syntax & usage examples

```
✓ ${array:a[0][1]}
✓ ${object:o.x["Foo Bar"].y}

// hints a type of the property access result to use modifiers
✓ ${((string) array:a[5] | split(":"))}
✓ ${((boolean) object:o.x.y | neg)}

// cannot use property access + default value
✗ ${object:o.x: {x: 10} .x}

// unknown type of property access
✗ ${object:o.x | split(":")}

// invalid type of property access
✗ ${((number) object:o.x | split(":"))}
```

```
// unknown type
```

```
❌ ${({foo}) object:o.x}
```

```
// attempt to cast value directly
```







```
❌ ${({foo}) object:o}
```

Type conversion table

NOTE

Conversions with the symbol  differ from the JavaScript conventions.

Undefined result is notated as "–".

Value	Boolean	Number	String	Datetime
<code>null</code>	<code>null</code>	<code>null</code>	<code>null</code>	<code>null</code>
<code>false</code>	<code>false</code>	<code>0</code>	<code>"false"</code>	–
<code>true</code>	<code>true</code>	<code>1</code>	<code>"true"</code>	–
<code>0</code>	<code>false</code>	<code>0</code>	<code>"0"</code>	<code>1970-01-01 ... 00</code>
<code>1</code>	<code>true</code>	<code>1</code>	<code>"1"</code>	<code>1970-01-01 ... 00</code>
<code>"0"</code>	<code>true</code>	<code>0</code>	<code>"0"</code>	<code>null</code>
<code>"20"</code>	<code>true</code>	<code>20</code>	<code>"20"</code>	<code>0020-01-01 ... 00</code>
<code>[]</code>	<code>true</code>	– 	<code>"[]"</code> 	–
<code>[20]</code>	<code>true</code>	– 	<code>"[20]"</code> 	–
<code>["two"]</code>	<code>true</code>	–	<code>'["two"]'</code> 	–
<code>{}</code>	<code>true</code>	–	<code>"{}"</code> 	–

Value	Boolean	Number	String	Datetime
<code>{a: 1}</code>	<code>true</code>	-	<code>'{"a":1}'</code> 🚩	-
<code>2023-01</code>	<code>true</code>	<code>1672531200000</code>	<code>"2023-01-01T ... 000Z"</code>	<code>2023-01-01T ... 0</code>
<code>2023/2024</code>	<code>true</code>	-	<code>"2023-01-01T ... /2024-01-01T ... "</code>	-

Spreading

Typically, parameters placed in an array (object) are expanded as its single element (key/value). This may not be desirable when the parameters are of the array/object type.

Let's have the following prejson:

```
{ letters: [ "A", "B", ${array:letters}] }
```

When we expand it, providing `["X", "Y", "Z"]` as the value of `letters` parameter, we'll get

```
{ "letters": ["A", "B", ["X", "Y", "Z"]] }
```

But that is probably not the result we want. We expect the letters passed as parameter value to be added as elements of the parent array. And here's the turn of **the spread operator**. Let's slightly update the original prejson:

```
{ letters: [ "A", "B", ${ ... array:letters}] }
```

With this small change, the prejson now expands as expected:

```
{ "letters": ["A", "B", "X", "Y", "Z"] }
```

Spread operator rules

- Spread can only be used with **object** and **array** parameters, or parameters whose modifiers or property access result in an object/array.
- The resulting type of a spread parameter must match the parent entity type. That means, you cannot spread an array parameter inside an object.
- If the value of a spread parameter is not defined at the time of expansion, or is null, expanding the parameter is omitted. The result is the same as if an empty array/object was spread.
- When spreading in objects, the last added property always takes priority over those added before.

The following example expands to `{x: 20, y: 20}`:

```
{
  y: 10,
  ${... object:o:{x: 10, y: 20}},
  x: 20
}
```

Syntax & usage examples

```
✓ [ 10, ${... array:x} ]
✓ { x: 10, ${... object:o} }
✓ { @includeIf (${object:o|defined}) <${... o}> }
```

```
// split converts string to array
```

```
✓ [ ${... string:s|split("/")}] ]
```

```
// cast in a modifier changes the type
```

```
✓ { ${... number:n|(object) expr("{x: input}")} }
```

```
// attempt to spread invalid type
```

```
✗ [ ${... string:s} ]
```

```
✗ [ ${... boolean:b} ]
```

```
✗ [ ${... array:a|join(",")} ]
```

```
// parameter type does not match the parent type
```

```
✗ [ ${... object:o} ]
```

```
✗ { ${... array:a} }
```

```
✗ { @includeIf (true) <${... array:o}> }
```

Macros

Macros allow you to conditionally include specific segments of PreJSON during expansion. Such conditions consist of combination of parameter values and/or literal values. Let's illustrate with an example.

Suppose we want to fetch a request from an API that accepts data filters shaped as an object where the key is the filter name and the value is its config:

```
{
  "filters": {
    "date": {
      "from": " ... ",
      "to": " ... "
    },
    "country": {
      "code": "cz"
    }
  }
}
```

Since we are using PreJSON, the request payload will be parametric because we want to allow users to change the filtering. So, for example, we will get the user-selected country code via the PreJSON context as a `countryCode` [parameter](#). The PreJSON string will then look like this:

```
{
  "filters": {
    "date": {
      "from": " ... ",
      "to": " ... "
    },
    "country": {
      "code": ${countryCode}
    }
  }
}
```

This will work great as long as the user wants to filter the data by country. But what if they want to turn off the filter? Setting `countryCode` to null won't help:


```
{
  "filters": {
    "date": {
      "from": " ... ",
      "to": " ... "
    },
    "country": {
      "code": null
    }
  }
}
```

Even if we omit setting the value of `countryCode` so that it's undefined, the `country` filter will still be used but with an invalid config.

```
{
  "filters": {
    "date": {
      "from": " ... ",
      "to": " ... "
    },
    "country": {}
  }
}
```

In this situation, we will use a macro. This will include the object entry `country` in the resulting object when expanding only if a value is assigned to `countryCode`:

```
{
  "filters": {
    "date": {
      "from": " ... ",
      "to": " ... "
    },
    @includeIf (${countryCode|defined}) <
      "country": {
        "code": ${countryCode}
      }
    >
  }
}
```



TIP

To check if a parameter has a value assigned, `defined` modifier is used. But you're not limited to checking the value presence. Using other [modifiers](#), you can compare parameter value to a const etc.



LEARN MORE

See the [Macro reference](#) for a detailed list of built-in macros.

Syntax

The syntax of macros consists of the macro name preceded by the at symbol, followed by a parenthesized list of arguments and a list of segments, each enclosed in chevron symbols.



```
@macroName (...args) < segment0 > < segment1 > ... < segmentN >
```



SEGMENT SYNTAX

The inner syntax of each segment depends on the place of macro use. If a macro is placed inside an object, an object entry list is expected to be the content of segment. If placed inside an array, a list of values is expected.

Examples

```
// ✓
{ @include(${city|defined}) <city: ${city}> }
{ cities: ["Prague", "Wien", @include(${city|defined}) <${city}>] }

// ✓ Macros can be nested
{
  @includeIf(${country|defined}) <
    country: {
      code: ${country}
    }
  >
}
```

```
        @includeIf(${limit|defined}) <
            limit: ${limit}
        >
    }
>
}
```

```
// ✖ At least one segment is required
{ @includeIf(true) }
```

```
// ✖ Invalid segment syntax inside an object
{ @includeIf(true) <10> }
```

```
// ✖ Invalid segment syntax inside an array
[ @includeIf(true) <city: "Prague"> ]
```

PreJSON Monaco

This package adds PreJSON support to the Monaco editor. When registered, the following features can be used for PreJSON documents opened in the editor:

- Diagnostics – in-place reporting of PreJSON syntax and semantic errors/warnings.
- Schema validation – uses provided JSON schemas to validate PreJSON structure (where applicable)
- Completion
 - suggests possible values or object property keys based on supplied JSON schemas
 - provides hints for parameters, macros and modifiers
- Hover information
 - displays descriptions provided in JSON schemas
 - displays macro and modifier details
- Document formatting
- Syntax highlighting

Usage

Registering the PreJSON extension

Register the PreJSON extension in the entry of your application.

```
import { registerPreJSON } from '@sbks/prejson-monaco'
import * as monaco from 'monaco-editor'

if (window)
  window.MonacoEnvironment = {
    getWorker(moduleId, label) {
      switch (label) {
        case 'editorWorkerService':
          return new Worker(
            new URL(
              'monaco-editor/esm/vs/editor/editor.worker',
              import.meta.url
            )
          )
      }
    }
  }
```

```

        case 'prejson':
            return new Worker(
                new URL('@sbks/prejson-mono/lib/prejson.worker', import.meta.url)
            )
            // any other languages you need to use
    },
}

registerPreJSON(monaco)

```

If you're using the React wrapper [@monaco-editor/react](#), you'll need the registration wait for resolving Monaco instance:

```

import { registerPreJSON } from '@sbks/prejson-mono'
import { loader } from '@monaco-editor/react'

if (window)
    window.MonacoEnvironment = {
        getWorker(moduleId, label) {
            switch (label) {
                case 'editorWorkerService':
                    return new Worker(
                        new URL(
                            'monaco-editor/esm/vs/editor/editor.worker',
                            import.meta.url
                        )
                    )
                case 'prejson':
                    return new Worker(
                        new URL('@sbks/prejson-mono/lib/prejson.worker', import.meta.url)
                    )
                    // any other languages you need to use
            }
        },
    }

loader.init().then((monaco) => registerPreJSON(monaco))

```

Mode configuration

By default, all the extension features are enabled. If you'd like to disable some of them, you can overwrite the default configuration:

```
import { prejsonDefaults } from '@sbks/prejson-monaco'

// Turn off code completion
prejsonDefaults.setModeConfiguration({ completion: false })
```

Currently supported features	
<code>diagnostics</code>	Displays syntax and semantic error hints as well as schema validation results.
<code>hover</code>	Displays information extracted from schemas and documentation for PreJSON language features.
<code>completion</code>	Provides hints for values or object property keys based on schema, suggests declared parameters.
<code>format</code>	Enables formatting the document.

Associating a PreJSON context

By default, PreJSON documents are parsed in the default context. However, because the context determines important properties such as

- what the formatted output should look like
- which syntactic features are allowed
- or what parameters should be available in each instance parsed in a given context,

the extension provides a way to associate a PreJSON context with specific documents.

```
import { prejsonDefaults } from '@sbks/prejson-monaco'
import { PreJSONType } from '@sbks/prejson'

const customContext = new PreJSONContext({ format: { quotedKeys: 'always' } })
customContext.declare('user', PreJSONType.String)

prejsonDefaults.setContexts([
```

```
{
  fileMatch: ['*.prejson'],
  context,
},
])
```

This snippet will ensure that all PreJSON documents whose path ends with the *.prejson* are parsed with *user* parameter in the scope. When such a document is formatted, all object property keys are quoted as well as any completion suggestions that are applied.

Associating a JSON schema

By default, the extension only reports syntactic and semantic errors found during parsing, and if completion is requested, only provides a list of parameters or syntax feature documentation. However, similar to [vscode-json-languageservice](#), the PreJSON extension allows you to associate JSON schemas with documents. Once a schema is provided for a document, the extension reports structural differences from the schema, adds appropriate values, and displays meta information.

```
import { prejsonDefaults } from '@sbks/prejson-monaco'
import { PreJSONType } from '@sbks/prejson'

prejsonDefaults.setSchemas([
  {
    uri: 'prejson/schema.json',
    fileMatch: ['*.prejson'],
    schema,
  },
])
```

Examples

Notation	Meaning
<code>now[sD]</code>	Today midnight
<code>now[sD]+0.5D</code>	Today noon
<code>now[sY]</code>	This year's New Year midnight
<code>now[sD]/now</code>	Today from midnight until now
<code>now[sD]/now[eD]</code>	All day today
<code>P1D/now[sD]</code>	Yesterday
<code>now[sW]/now</code>	This week
<code>now[sM]/now</code>	This month
<code>now[sY]/now</code>	This year
<code>PT1H/now[sH]</code>	Last hour
<code>P1D/now[sH]</code>	Last 24 hours
<code>P1W/now[sD]</code>	Last 7 days
<code>P30D/now[sD]</code>	Last 30 days
<code>P90D/now[sD]</code>	Last 90 days
<code>P1Y/now[sY]</code>	Last year
<code>2021/2022</code> or <code>2021/P1Y</code> or <code>P1Y/2022</code>	Year 2021

Notation	Meaning
<div>2022-06/2022-07</div> or <div>2022-06/P1M</div> or <div>P1M/2022-07</div>	June 2022

Notation

How PreJSON Time differs from ISO standards and what it offers in addition.

Datetime

Fixed Datetime

Fixed moment notation conforms to [ISO 8601 Dates & Times](#) with the following notes:

- Only years in the range from 0000 to 9999 are allowed (extended notation `±000000` not supported).
- Reduced date notation (`2022-04` for *April 2022* or `2022` for start of the year 2022) is not supported. Date and month are always required.
- Date/month/year separator `-` and time separator `:` are required.
- [Week date](#) and [ordinal date](#) notations are not supported.

```
// ✅  
2020-01-01  
2020-01-01T12  
2020-01-01T12:24  
2020-01-01T12:24:35  
2020-01-01T12:24:35.456  
2020-01-01T12:24+01  
2020-01-01T12:24+01:30  
2020-01-01T12:24Z  
  
// ❌ separators required  
20200101  
2020-01-01T122435.456  
2020-01-01 12:24  
  
// ❌ Incomplete date  
2020-01  
2020  
T12:24:35.456  
  
// ❌ Unsupported ISO features  
-01000-01-01  
2020W01
```

2020-156
2020156

Relative Datetime

now	[sD]	+P1W
↓	↓	↓
base	round	offset

A relative moment consists of a base that specifies a general time point, and optionally an [offset](#) and/or a [rounding](#).

Base `now`

Represents the current millisecond at the time of datetime expansion.

Offset

"Moves" the base. Suppose we don't need to point to a current millisecond, but instead want to express the same millisecond an hour ago – so we need to "move" `now` back one hour.

The offset is appended to the *base* and is notated in the same way as [ISO Duration](#), so the mentioned example is `now-PT1H`.

```
// ✅  
now-PT1H  
now+P1M1D  
now+P0.5M  
  
// ❌ Missing offset direction  
now P1D  
  
// ❌ Invalid duration  
now-P10X  
now+P1H10D  
  
// ❌ Offset is not supported in fixed time  
2020-01-01+P1W
```


NOTE


For backward compatibility, PreJSON also supports offset notation without the `P` prefix that is required by ISO 8601. However, it's strongly recommended to include the prefix for better readability and consistency.


Rounding


In most cases, we won't be interested in the current millisecond of the expanded datetime. Instead, we may want to know when the week associated with the moment began. For this reason, PreJSON supports time rounding, which adjusts the moment to the start/end of a given time unit.


The rounding is specified by a pair of direction (start `s` or end `e`) and time unit enclosed in square brackets.

```
// 
now[sD]
now[eW]+P1D
now[em]

//  Missing direction or unit
now[s]
now[D]

//  Invalid unit
now[sX]

//  Rounding must precede offset
now+P1W[sD]

//  Rounding is not supported in fixed time
2020-01-01[sW]
```

Supported units	
<code>Y</code>	year
<code>Q</code>	quarter
<code>M</code>	month

Supported units	
W	week
D	day
H	hour
m	minute
s	second

Date Range

Fixed Date Range

Fixed date range notation conforms to [ISO 8601 Time intervals](#) with the following notes:

- Only variants *start/end*, *start/duration* and *duration/end* are supported.
- Deriving end from start in the *start/end* variant (e.g. `2022-01-01/05` for *January 1 – January 5, 2020*) is not supported, both endpoints must conform to [the datetime notation](#).
 - However, date ranges can (as opposed to datetimes) use the reduced date notation, e.g. `2020/2021` which stands for *January 1, 2020 – January 1, 2021*.

```
// ✅
2020/2021
2020-01-01/2021
2020-01-01T12/2020-01-01T13

// ❌ Invalid parts
P10W/P1D

// ❌ Unsupported derived end part
2020-01-01/02-01
2021-02/03
```

Relative Date Range

In relative date ranges, one or both of the parts are built from [a relative datetime](#).

// 

now[sW]/now

2020/now[sY]

2021-01-01T12:00/now[sD]+P1D

Basics of use

Datetimes and date ranges in PreJSON can be either fixed or relative. Fixed times are something you are used to in JS Date, Moment or Luxon, while relative times do not express any specific point in time until they're [expanded](#).

Instantiating a time

Both `PreDateTime` and `PreDateRange` support creating an instance from a string matching the [notation](#) or from a corresponding Luxon instance (`DateTime`/`Interval`). PreJSON Time instances are **always** created using static methods, not by calling the constructor – just the same way as in Luxon.

```
import { PreDateTime, PreDateRange } from '@sbks/prejson-time'

PreDateTime.fromString('now[sD]')
PreDateRange.fromString('now/now[eD]')
PreDateTime.fromLuxon(DateTime.utc())
PreDateRange.fromLuxon(Interval.fromISO('2020/2021'))
```

Error handling

PreJSON Time follows the Luxon convention and does not throw an error if its input is invalid. Instead, the validity is indicated by the `isValid` getter and the `invalidExplanation` member reports the reason for the invalidity.

```
import { PreDateTime, PreDateRange } from '@sbks/prejson-time'

const range = PreDateRange.fromString('2022/2021')

// `false`
console.log(range.isValid)

// "Reversed date range"
console.log(range.invalidExplanation)
```

Expansion

Once you want to make a relative PreJSON Time instance to a fixed one, you need to expand it. Expansion evaluates `now` base to the current millisecond and applies `offset(s)` and `rounding(s)`. The result is a new fixed instance.

PreJSON Time instances provide several ways to expand:

- `expand` method – returns a fixed PreJSON instance (or itself if already is fixed)
- `toLuxon` method - expands the instance and returns a Luxon instance representing the fixed time
- `toISO` method - expands the instance and returns its ISO representation



TIP

When using the `toLuxon` method, the returned Luxon instance is extended with a special property that carries the notation of the original PreJSON Time from which the Luxon instance was created by the expansion. It can be accessed using the `PreDateTime.ORIGINAL` (or `PreDateRange.ORIGINAL`) symbol.