
# RBMS Template Engine

**Version 20.8.1, 10 August 2020**

| Registered Address | Support | Sales |
|---|---|---|
| 26, Kingston Terrace, Princeton, New Jersey 08540, United States | | |
| | | +91 80 4850 5445 |
| http://www.rtbrick.com | support@rtbrick.com | sales@rtbrick.com |

# Table of Contents

# 1. RBMS Template Engine

The RBMS Template Engine is an execution engine for templates. A folder in the filesystem serves as template storage for the engine. The content of the folder follows a convention.

## 1.1. Template Folder structure

*Template folder structure*

```
templates
 |-- includes
 |   |-- <include-template>.gojson
 |-- <template name>
     |-- config.yaml
     |-- <include-template>.gojson
     |-- <main-template>.gojson
```

The template engine uses one templates folder where all the templates are stored. Each template resides in his own folder, the folder name is the template name. The config.yaml file inside a template folder indicates that this folder is a template. In this file also other configurations for the template engine can be made.

The template folder contains one main-template and can contain multiple `include-templates. The include-templates can be included into the main template.

Folders that don't contain a config.yaml are not treated as templates. This folders can be used as containers for other include-template files.

## 1.2. Template config

This section describes the config.yaml file. Image this folder structure for the next examples.

*Simple example folder structure*

```
templates
|-- includes
|   |-- global_include.gojson
|-- sample
    |-- config.yaml
    |-- local_include.gojson
    |-- main.gojson
```

Here there is a main-template which includes the local_include-template and the global_include-template. The config.yaml is used by the template engine to parse the right files, so that the include-directives work.

*templates/sample/config.yaml*

```
engine: golang
main_template: "main.gojson"
main_pattern: "*.gojson"
include_pattern: "includes/*.gojson"
post_processors:
  - removeTrailingCommas
  - prettyJSON
```

*Table 1. config.yaml attributes*

| Attribute | Default | Description |
|---|---|---|
| engine | golang | selects the template engine, at the moment only golang is supported |
| main_template | none | points to the entrypoint of the rendering process, this template is used as the top most, it hast to be included in the main pattern. |
| main_pattern | none | describes which files the engine should parse from the template folder. |
| include_pattern | none | describes which files the engine should additionally parse relative to the templates folder. |
| post_processors | none | allows to specify post processors that are used in that order on top of the generated output. |

*Table 2. Post processors*

| Attribute | Description |
|---|---|
| removeTrailingCommas | removes in json files the commas which are not valid, this makes the template much easier |
| removeEmptyLines | removes empty lines |
| prettyJSON | Pretty converts the input json into a more human readable format where each element is on it’s own line with clear indentation |
| uglyJSON | Ugly removes insignificant space characters from the input json byte slice and returns the compacted result. |

## 1.3. GO Lang Template Engine

The default engine is the golang template engine. This gives some links to more detailed information.

The GO Lange template engine is based on:

- GoLang test template
  The golang text template engine. This allows evaluating arguments, execute actions and include other templates.
- sprig functions
  Beside of the default functions golang already provides, the sprig function library is added to the engine.

## 1.4. Example

This section shows a simple example, that covers a lot of functionality of the templates.

The example uses the following folder structure. Each file will be described in more detail.

*Full example folder structure*

```
templates
|-- includes
|   |-- global_include.gojson
|-- sample
    |-- config.yaml
    |-- example_variables.json
    |-- local_include.gojson
    |-- main.gojson
```

The template is called sample, because there is a config.yaml in the folder sample.

*templates/sample/config.yaml*

```
engine: golang
main_template: "main.gojson"
main_pattern: "*.gojson"
include_pattern: "includes/*.gojson"
post_processors:
  - removeTrailingCommas
  - prettyJSON
```

The config.yaml file states that the main_template is called main.gojson, so thats the entrypoint for the generation.

The main_pattern defines this files templates/sample/*.gojson should be parsed into the template engine, so also the main_pattern is included.

The include_patterns defines this files templates/includes/*.gojson should be parsed into the template engine.

The post_processors are used to remove the trailing commas and make the JSON output more readable.

Let's expect the following example variables structure.

*templates/sample/example_variables.json*

```
{
  "description": "sample",
  "interfaces": [
    {
      "name": "ifp_0/0/1",
      "ipv4": "127.0.0.1",
      "x": 5,
      "y": 3
    },    {
      "name": "ifp_0/0/2",
      "ipv4": "127.0.0.2",
      "x": 4,
      "y": 4
    }
  ]
}
```

The this variables can be used to fill a template.

*templates/sample/main.gojson*

```
{{define "t1"}}
    "hostname": "static",
{{end}}
{
    {{template "t1"}}
    "description": "{{.description}}",
    "interfaces": {
        {{template "local_include.gojson" .interfaces}}
    },
    "list": {{template "global_include.gojson" .}}
}
```

This templates starts with a definition of a new template t1 that will be used in this template.

This template t1 is included immediately after {.

Then the description is added, the selection from the variable is done via the .description.

For the interfaces we use the local template local_include.gojson, the variables that are forwarded to the template are the .interfaces so only the array of the original variable set.

To render the list we include the global_include.gojson template and forward the original variable set.

*templates/sample/local_include.gojson*

```
{{range .}}
"{{.name}}": {
    "ip": "{{.ipv4}}",
    "1000/x*y": {{div 10000 (mul .x .y) }},
},
{{end}}
```

The local_include.gojson iterates over the interfaces list and prints the name and ip-address of the interface.

Also a simple computation is done by using the sprig functions div and mul.

*templates/includes/global_include.gojson*

```
[
{{range .interfaces}}"{{.name}}",{{end}}
]
```

The global_include.gojson iterates over the interfaces list and prints in an array.

This json template does not create a valid json. The commas are not set correct. The document is not well formatted. Therefore it is easier to create the templates. To create a syntactically correct and well formatted document we use post processors. The syntax is corrected by removeTrailingCommas` post processor. The format is corrected by the prettyJSON post processor.

The next source block shows the expected outcome when applying the variables from above to the template.

*templates/sample/example_result.json*

```
{
  "description": "sample",
  "hostname": "static",
  "interfaces": {
    "ifp_0/0/1": {
      "1000/x*y": 666,
      "ip": "127.0.0.1"
    },
    "ifp_0/0/2": {
      "1000/x*y": 625,
      "ip": "127.0.0.2"
    }
  },
  "list": [
    "ifp_0/0/1",
    "ifp_0/0/2"
  ]
}
```

## 1.5. TestKit

In order to do a fast template prototyping we developed a test kit. The test kit allows to execute a template with a given variable set and validate the outcome against an expected result.

To execute we have to specify:

- templatePath: Template main folder (default ".")
- template: Template name
- format: File format [txt, json, json5] (default "txt")

So for example if we execute template-engine-test -template sample -test example -format json inside the templates folder, this command will execute the sample template with the content of the example_variables.json file as input variables. After execution the outcome is stored in the example_got.json file, and validated against the example_result.json file. The format not only specifies the file endings, it also specifies how the validation is done. So for example the json format does not care about ordering of whitespace differences.