ScriptKnapper – User Guide:

What is ScriptKnapper?:

ScriptKnapper was created to aid code generation, in the hope of making the building and maintaining of your code/scripts easier.

ScriptKnapper is a JavaScript library. You can make use of the library in NodeJS scripts, or within your browser via the Web UI.

To use ScriptKnapper, you create a list of templates, and then a list of markup instructions (that will feed data into the templates). Both the templates and the markup instructions are represented in JSON, which you then feed into the library's *scriptKnapperMain* function to generate your script/code.

Contents:

JSON - A Guide for Non-Programmers:

JSON stands for "JavaScript Object Notation", and is often used by developers as a way of representing data in a text format (just like XML, if that is something you are familiar with). The data can be transferred between applications, or loaded from a .JSON file, and then converted into an "object" that a programming language (often, but not exclusively, JavaScript) can then use to access the individual data values represented within it.

JSON can represent multiple types of data:
- Strings (which is a programmer word, meaning text).
- Numbers (including decimal numbers)
- Booleans (This is another programmer word. It means either true or false.).
- Null (which means it has no value).
- Objects. These are containers for related pieces of data (You will learn more about objects in a moment.).
- Arrays (Think of this as a list of any of the above types).

Let's say we wanted to represent a car in an object. There are multiple properties we may want to represent:
- numberOfDoors
- numberOfWheels
- engineType
- lastMOTDate
- isStolen
- driver

We can represent this in an object in the following way:

```
{
        "numberOfDoors": 3,
        "numberOfWheels": 4,
        "engineType": "diesel",
        "lastMOTDate": "08/02/2019",
        "isStolen": false,
        "driver": null
}
```

The handlebar braces ("{" and "}") tell the JSON interpretor (known as a JSON "parser") that this is an object, and that the data in between the braces are the "properties" that belong to the object.

Each property has 2 parts: a key, and a value.

In JSON, a key is always set within double quotes (Single quotes cannot be used around either keys or values in JSON.), and must not contain spaces. The key is the name of the property, and it is how the application will reference this data.

The value is represented differently depending on its type. In the example

above, both engineType and lastMOTDate are string values, so they are represented in double quotes. The numberOfWheels is a number, so it does not require the quotes. The isStolen value is a boolean, so it does not require quotes, and neither does driver, as that is currently null.

Each property is separated by a comma.

Objects as Values in JSON:

A property can also have another object as its value:

```
{
        "numberOfDoors": 5,
        "numberOfWheels": 4,
        "engineType": "diesel",
        "lastMOTDate": "08/02/2019",
        "isStolen": false,
        "driver": {
                "name": "Matthew",
                "Age": 28,
                "hasLicense": true
        }
}
```

Here, the object has an internal object for the driver property.

Arrays in JSON:

Often, a JSON file will not contain a single object, but an array of objects, separated by commas:

```
[
        {
                "numberOfDoors": 5,
                "numberOfWheels": 4,
                "engineType": "diesel",
                "lastMOTDate": "08/02/2019",
                "isStolen": false,
                "driver": null
        },
        {
                "numberOfDoors": 3,
                "numberOfWheels": 3,
                "engineType": "petrol",
                "lastMOTDate": "01/01/2010",
                "isStolen": true,
                "driver": "The really nasty thief"
        }
]
```

Here, the objects are separated by commas, and put inside square braces. This would give an application multiple car objects to work with.

You can also use an array as a value for a property:

```json
{
        "numberOfDoors": 5,
        "numberOfWheels": 4,
        "engineType": "diesel",
        "lastMOTDate": "08/02/2019",
        "isStolen": false,
        "drivers": [
                { "name": "Matthew", "Age": 28, "hasLicense": true },
                { "name": "Mum", "Age": null, "hasLicense": true },
                { "name": "Dad", "Age": null, "hasLicense": true }
        ]
}
```

Here, the object has multiple drivers, each represented by their own object.

Generating an Example Script:

As an example, let's see how we may generate a Limerick with ScriptKnapper.

There once was a man from Nantucket.
He had a chicken nugget.
He ate it neat.
It tasted sweet.
Then he went and bought a whole bucket.

A First ScriptKnapper Script:

Firstly, let's create some simple template JSON for this Limerick, that doesn't require any data.

```
[
        {
                "name": "limerickTemplate",
                "template": "There once was a man from Nantucket.\nHe had a chicken nugget.\nHe ate
it neat.\nIt tasted sweet.\nThen he went and bought a whole bucket."
        }
]
```

There are a few things to note about the above JSON:
- In ScriptKnapper, the templates have to be within an array (The markup JSON has this same rule.).
- To ScriptKnapper, all property values should be an array, object, or string. You cannot use number/boolean/null types.
- The template JSON object must have both a name property, and a template property.

There is something else that needs to be noted about the above JSON. The template property's value is actually on a single line. This is important, as JSON cannot accept line breaks in a value. Instead, the line breaks have been replaced with "\n". This is an escape code that will get converted into a new line character in the script. There is a String Preparation tool in the ScriptKnapper Web UI that will allow you to replace new lines and tabs with escape characters, so be sure to make use of this.

Next, we need the markup JSON (an array of JSON, giving instructions on what templates to use, and what data to pass to them). There is no data to pass, so here we are passing an empty data array in the markup (We could actually omit the data property in this instance, if we wanted to, and this would still work.):

```
[
        {
                "template": "limerickTemplate",
                "data": []
        }
]
```

The template property tells ScriptKnapper what template to use, and the data array is where you will pass in data to the template (which we will do soon).

Try using the above JSON with the ScriptKnapper Web UI. You should get the below output:

There once was a man from Nantucket.
He had a chicken nugget.
He ate it neat.
It tasted sweet.
Then he went and bought a whole bucket.

Just as we'd expected, we received the original Limerick. However, what if we wanted the ability to change the final words of lines 3 and 4? To do so, we would need to pass parameters to the template, and then have ScriptKnapper insert that data.

Passing Parameters to a Template:

Firstly, let's change our template:

There once was a man from Nantucket.
He had a chicken nugget.
He ate it {: lineThreeRhyme :}.
It tasted {: lineFourRhyme :}.
Then he went and bought a whole bucket.

Here, we have replaced the final words on the 2 lines with parameter names, surrounded by "{:" and ":}". This is how you reference a parameter that has been passed into the template (On a side note, if you actually want to use "{:" or ":}" in your generated script – without them being interpreted as braces for a parameter call – you can use "@ohb:" in place of "{:", and "@chb:" in place of ":}". ohb stands for "Opening HandleBar", and chb stands for "Closing HandleBar".).

Make the above changes to your template JSON. Also, we are about to make multiple calls to this template (something you will see in the new markup JSON below), so add 2 line-breaks ("\n\n") to the end of the template as well. Finally, replace your markup JSON with the below:

```
[
    {
        "template": "limerickTemplate",
        "data": [
            { "lineThreeRhyme": "neat", "lineFourRhyme": "sweet" },
            { "lineThreeRhyme": "quick", "lineFourRhyme": "sick" }
        ]
    }
]
```

Here, we are doing 2 things differently to the previous markup. Firstly, we are passing data into the template through objects. Secondly, as we are passing 2

data objects, we are actually calling the template twice. Therefore, ScriptKnapper will generate one after the other (this is why I suggested you add 2 line-break escape codes – "\n" – to the end of the template).

   Most of the time, you will want to have multiple template calls to different templates in your script. In order to do this, you just pass more template call objects in the markup JSON. See the below example:

```
[
        {
                "template": "aFirstTemplate",
                "data": [
                        { "greeting": "hello" }
                ]
        },
        {
                "template": "aSecondTemplate",
                "data": [
                        { "partingPhrase": "goodbye" }
                ]
        }
]
```

   If you now generate the Limerick with the new JSON, you should get the below result:

There once was a man from Nantucket.
He had a chicken nugget.
He ate it neat.
It tasted sweet.
Then he went and bought a whole bucket.

There once was a man from Nantucket.
He had a chicken nugget.
He ate it quick.
It tasted sick.
Then he went and bought a whole bucket.

   Here, we can see that the parameter calls have been replaced with the data we passed in.

Calling Templates from within Templates:

   Let's add a bit more customisation to our template:

There once was a man from Nantucket.
He had a chicken nugget.
{: lineThreeBeginning :} {: lineThreeRhyme :}.
{: lineFourBeginning :} {: lineFourRhyme :}.
Then he went and bought a whole bucket.

   This is a fine solution, however the third and fourth lines are both doing the exact same thing (taking and displaying 2 strings, separated by a space, and

then ending with a period). You may want to put these parts of the template into their own separate template. Let's do that in our template JSON:

```
[
	{
		"name": "limerickTemplate",
		"template": "There once was a man from Nantucket.\nHe had a chicken
nugget.\n{{: \"template\": \"middleLine\", \"data\": [{ \"lineBeginning\": \"He ate
it\", \"lineRhyme\": \"neat\" }, { \"lineBeginning\": \"It tasted\", \"lineRhyme\": \"sweet\" }] :}}Then he
went and bought a whole bucket.\n\n"
	},
	{
		"name": "middleLine",
		"template": "{: lineBeginning :} {: lineRhyme :}.\n"
	}
]
```

Firstly, notice that we've added a new "middleLine" template. Secondly, you can see that we've made some changes to the "limerickTemplate", so let's take a further look at that (and let's put it on multiple lines, with tabs, so it is easier to read):

```
"template": "There once was a man from Nantucket.\n
He had a chicken nugget.\n
{{:
	\"template\": \"middleLine\",
	\"data\": [
			{
				\"lineBeginning\": \"He ate it\",
				\"lineRhyme\": \"neat\"
			},
			{
				\"lineBeginning\": \"It tasted\",
				\"lineRhyme\": \"sweet\"
			}
	]
:}}
Then he went and bought a whole bucket.\n\n"
```

The first thing you may notice is the "{{:" and ":}}". These tell ScriptKnapper that this is a call to another template (If you actually want to use these strings in the template, then replace "{{:" with "@odhb:", and ":}}" with "@cdhb;". odhb stands for "opening double handlebars", and cdhb stands for "closing double handlebars").

You will have also noticed that all of the double quotation marks within the inner-template call are preceded by backslashes. The need for the backslashes is due to a limitation of the JSON format. In JSON, you have to use double quotes, not single quotes. However, this means you cannot then use double quotes within your string, as the parser would not know which ones were within the string, and which one was the end of the string. Because of this, JSON will only allow double quotes within a string if they are proceeded by a backslash (this means you are "escaping" the characters). Therefore, in order to use a template call within a JSON property, you have to use the backslashes.

Coming back to the template itself, you may notice that we are feeding in the

end of the lines manually into the inner-template. However, we have already passed those values into the limerickTemplate, so couldn't we just feed those values in?

The answer is yes.

Passing Parameters Down to Inner Templates:

Any parameter passed to a template can then be passed down to the templates called within that template. However, you should be aware that if there is a parameter with the same name passed manually into the inner-template, then that value will be passed down, not the value fed into the outer-template.

```
"template": "There once was a man from Nantucket.\n
He had a chicken nugget.\n
{{
        \"template\": \"middleLine\",
        \"data\": [
                        {
                                \"lineBeginning\": \"He ate it\",
                                \"lineRhyme\": \"{:lineThreeRhyme:}\"
                        },
                        {
                                \"lineBeginning\": \"It tasted\",
                                \"lineRhyme\": \"{:lineFourRhyme:}\"
                        }
        ]
}}
Then he went and bought a whole bucket.\n\n"
```

As you can see, we have now fed parameter calls into the template as data (Note that these are still within quotes.). And, as the data is passed down to the inner-templates, this will work as long as the inner-template call doesn't also have a parameter named "lineThreeRhyme" or "lineFourRhyme".

Of course, if we can feed a parameter into a template call, you may be wondering if we can pass a template call into a template as a parameter?

Passing Template Calls as Parameters:

You can achieve what we have achieved above, but in a slightly different way. Instead of placing the template calls within the template, you can place them within the data.

```
[
        {
                "template": "template1",
                "data": [
                        {
                                "exampleData":"Data - {{: \"template\": \"template2\", \"data\":
[] :}}. That was the second template."
                        }
                ]
        }
]
```

Here, the data contains a template call. This follows the same rules that we followed when putting a template call inside another template: the template call is within double quotes, and any double quotes that are within the call must be "escaped" with a backslash. You can also see that the template call can be surrounded by other text.

The ScriptKnapper Web UI:

   The tools in the ScriptKnapper Web UI are simple to use. First, there is the main tool for generating scripts out of template and markup JSON. Then, there is the string preparation tool, and the template JSON building tool.

The Main Tool:

ScriptKnapper Web UI

Enter the template JSON, and the markup JSON, and then select Transpile to generate your script.

Template JSON

```
[
    {
        "name": "limerickTemplate",
        "template": "There once was a man from
Nantucket.\nHe had a chicken nugget.\nHe ate it
{:lineThreeRhyme:}.\nIt tasted {:lineFourRhyme:}.\nThen
he went and bought a whole bucket."
    }
]
```

Markup JSON

```
[
    {
        "template": "limerickTemplate",
        "data": [
            { "lineThreeRhyme": "neat",
"lineFourRhyme": "sweet" },
            { "lineThreeRhyme": "quick",
"lineFourRhyme": "sick" }
        ]
    }
]
```

Output

Transpile

   Enter your template JSON in the first textbox, and your markup JSON in the second textbox. Then click the Transpile button, and the output textbox will return the result. If there was an error, the output section will turn red, and the textbox will contain the error message and details. If the script was generated successfully, the output box will turn green, and the textbox will contain your script.
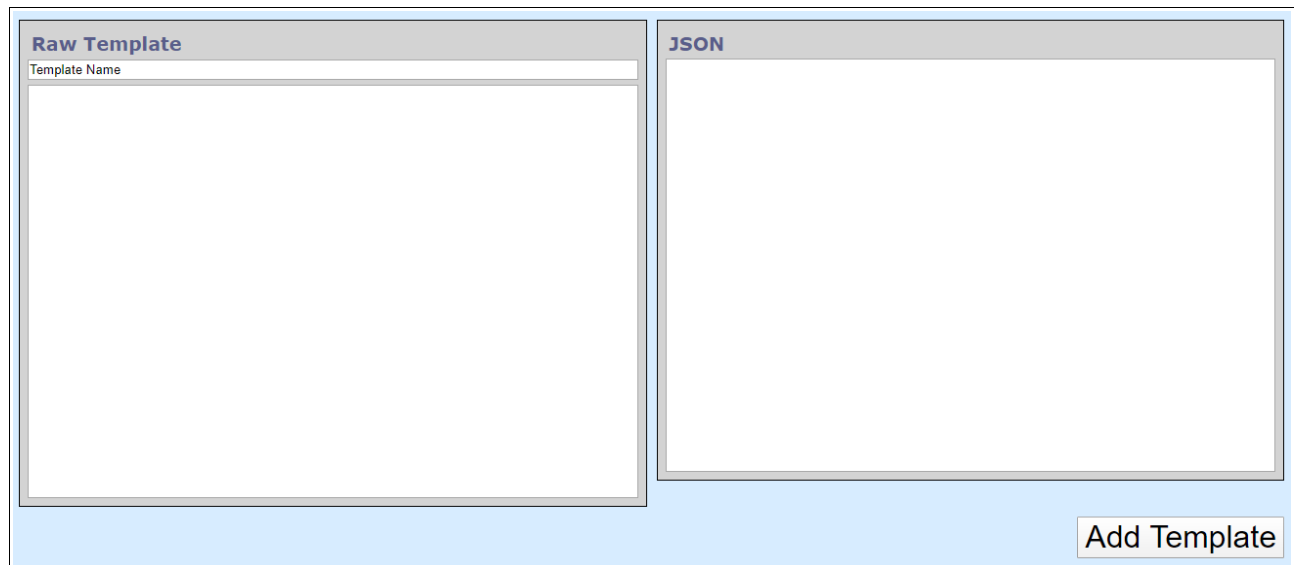
The String Preparation Tool:

1. [No change to whitespace ▼]   2. ☐ Escape double quotes   3. ☐ Replace tag strings

Input String

Prepared String

Prepare String

The string preparation tool has 3 options you can set. The first is a drop-down list of replacements for tabs and line breaks (You can choose to do nothing with them, remove them, replace them with spaces, or replace them with escape codes – such as \n and \t.). The next option determines whether or not the tool will escape any double quotation marks. The final option determines whether or not the tool will replace the handlebar brace tags with their replacement strings (such as "@ohb:" for "{:", and "@chb:" for ":}").

The Template JSON Building Tool:

This is a tool you can use to quickly turn your templates into JSON. Just enter the name of the template, and then enter the template (without escaping any double-quotes).

When you then select the Add Template button, the template will be converted into a JSON object, within a JSON array. It will have any whitespace replaced with the correct code ("\n", "\t", etc), and any double-quote characters will be escaped, as is required by the JSON format. You can then enter the next template, which will be added to the end of the array.

ScriptKnapper Functions for Automating Code Generation (Using NodeJS):

There are a couple of functions within the ScriptKnapper library that will give you the functionality of the Web UI tools when writing NodeJS scripts for automatic code generation. Once the templates and/or data JSON is prepared, it can be fed into the main ScriptKnapper entry-point function, to then generate your code.

```javascript
const myTextFileReader = require("./mod/myTextFileReader/myTextFileReader");
const myTextFileSaver = require("./mod/myTextFileSaver/myTextFileSaver");
const scriptKnapper = require("./dist/scriptknapper.bundle.js");

const templateName = "myTemplate";
const unpreparedTemplate = {
    templateName,
    template: myTextFileReader("./template1.txt")
}
const preparedTemplateJSON =
    scriptKnapper.buildTemplateJSON([ unpreparedTemplate ], false);

const message = "message here";

const data = JSON.stringify([{
    template: templateName,
    data: [ { message } ]
}]);

let result;
try
{
    result = scriptKnapper.scriptKnapperMain(data, preparedTemplateJSON);
}
catch(err)
{
    result = err.message;
}

myTextFileSaver(result);
```

See the function descriptions below.

prepareTemplateString():

Inputs:
- passedTemplateString: The string to be changed.
- whitespaceChange: How should the whitespace (not including space characters) in the string be handled? There are 4 options:
    1. "none": Whitespace will not be changed.
    2. "remove": Whitespace will just be removed.
    3. "spaceReplace": Whitespace will be replaced with space characters.
    4. "escapeCodeReplace": Whitespace will be replaced with escape codes (\n, \t, etc).
- escapeDoubleQuotes: A boolean value. Should double-quote characters be escaped?
- replaceTags: A boolean value. Should ScriptKnapper tags be replaced with their respective replacement strings?

Output:
- This function will return a string, that has been changed based on the input settings.

  This is a utility function to help you prepare your templates for the process. The function will make the required replacements on the inputted string (based on the provided parameters), and then return the result.

buildTemplateJSON():

Inputs:
- templates: An array of objects. Each object will have 2 properties:
  a "templateName" and a "template" (which is the text of the template).
- templatesAlreadyPrepared: A boolean value. Has the template text already
  been prepared?

Output:
- This function will return a JSON string, which will contain all of the
  given templates, in the structure required by scriptKnapperMain().
  If templatesAlreadyPrepared is false, double quote characters will be
  escaped, and whitespace characters (such as tabs and new-lines –
  but not spaces) will be replaced with escape codes.

  This is a utility function to help you prepare your templates for the process.
The function will take the given objects, and use them to generate the
template JSON string that can then be saved for later use, or passed directly
into scriptKnapperMain().

scriptKnapperMain():

Inputs:
- markupObjectsJSON: This is the JSON with the data to be entered into the template.
- templateObjectsJSON: This is the JSON with the different templates.

Output:
- This function will throw if there is an error.
- This function will return a string. This will be the generated code/script.

This is the entry point for generating code/scripts with ScriptKnapper.

The template objects JSON will be an array of objects. Each object will have a "name" property (which is the name of the template), and a "template" property (which is the template string).

The markup objects JSON will be an array of objects. These "markup" objects will be processed one after the other, to generate the final script. Each markup object will have a "template" property (which is the name of the template to use), and a "data" property (which will be an array of "data" objects).

Each "data" object in the markup object's data array will contain the data properties that are required by the template. The template will then be generated multiple times, once for each data object, using that data.

Extra Functionality:

Adding Data to the Data Object from within a Template:

   You can add extra data to the data object being used to populate the template, with a data addition tag. See the below example:

{+ "data1": "Hello World!", "data2": "Hello Again!" +}

   Placing this tag in a template would then add 2 properties to the data object: data1 and data2. If there was an inner-template call in this template, and you wanted to be sure it received these values for those properties, then using this tag would ensure that. This is because, if the data object already had those 2 properties, the values in the data addition tag would replace those pre-existing values.

   If you would like to actually use the "{+" or "+}" strings within your script, then you will need to replace them with "@ohb+" and "@chb+", respectively.