ScriptKnapper – Developer Guide

ScriptKnapper build version: 3.0.0

ScriptKnapper is a Javascript function library for code/script generation, using templates stored in a JSON format.

Important Note: *Please read the user guide before continuing with this developer guide, as that will help you to fully understand all of the software's features (which are not re-explained here).*

Contents:

Configuring and Making a New Build:

Once you have finished any work on the project's separate functions, you can bring them all together into a single file with the below command:

> npm run build

As defined in *package.json,* this command will then run the below command:

> node make_build.js

The *make_build.js* script is located in the root folder of the project. When running, it will inform you if a build file already exists, and ask you to confirm that you would like to overwrite it. If you would like the script to overwrite the file without asking, you can use the silent flag (either "-s" or "--silent") while running the *make_build.js* script directly.

However, before you build the library, you may want to reconfigure the build options. To do this, you will need to update *build_config.json. The build_config.json* file has 7 options:

1. buildFileName (string) – This is the name of the build file (such as "scriptKnapper.js")
2. sourceDirectory (string) – This is the location of the source files. All .js files (but not any .test.js files) within this directory will be brought into the build file.
3. buildDirectory (string) – This is the location to store the build file when it is created.
4. version (string) – The version number for this build.
5. openingComment (string) – Here you can set the text that will be displayed – within a multi-line comment – at the top of the build (You do not need to include the comment tags within this).
6. removeSingleLineComments (boolean) – This value tells the build script whether or not it should include any single-line comments it finds in the source files, when adding that code to the build.
7. removeMultiLineComments (boolean) – This value tells the build script whether or not it should include any multi-line comments it finds in the source files, when adding that code to the build.
8. exports (array) – This is an array of objects. These objects tell the build script which functions should be added to the exports. Each object will have 2 properties: "functionName" (the name of the function to be added to the exports object), and "exportName" (the name of the property on the exports object). An example of the

generated line is: *“exports.thisIsTheExportName = thisIsTheFunctionName;”.*

Important Note: *If you are wanting to use your new build file with the ScriptKnapper WebUI, you will need to replace the version used by the UI with your new build file. This is not done automatically by the build script.*

Running Tests:

The test suites for this project use a Javascript testing framework called Jest (https://jestjs.io).

To run these tests, use the below command:

> npm test

As defined in *package.json,* this command will then run the below command:

> jest --watch

This will run the .test.js files for each function in the /src folder. Visit the Jest website to learn more about how to use Jest.

The ScriptKnapper WebUI:

The web UI is made up of 4 files:

1. scriptKnapperWeb.html
2. scriptKnapperStyle.css – All of the styling for the UI.
3. scriptKnapperUIScript.js – This contains the functions that are called by the different tools in the UI (These are called via onClick events.). These functions will then make use of some of the functions in the ScriptKnapper library.
4. scriptKnapper.js – The build of ScriptKnapper that is being used by the UI.

The UI has 3 tools:

1. The main ScriptKnapper tool.
2. The Prepare a String tool.
3. The Build Template JSON tool.

Descriptions of the use of each of these tools can be found in the user documentation.

The Main ScriptKnapper Tool:

This tool's action button will trigger the transpileClickAction() function within scriptKnapperUIScript.js. This function makes use of scriptKnapperMain() from the ScriptKnapper library (scriptKnapperMain() is the entry point for generating code/scripts).

The Prepare a String Tool:

This tool's action button will trigger the prepareStringClickAction() function within scriptKnapperUIScript.js. This function makes use of prepareTemplateString() from the ScriptKnapper library.

The Build Template JSON tool:

This tool's action button will trigger the buildTemplateJSONClickAction() function within scriptKnapperUIScript.js. This function also makes use of buildTemplateJSON() from the ScriptKnapper library.

Descriptions of Individual Functions:

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 return an array with 2 values, to be destructured by the caller.
- The first value will be a boolean, which is true if there was an error, or otherwise false.
- The second value will be a string. This will be the generated code/script, or an error if there was a problem during the process.

This is the entry point for generating code/scripts with ScriptKnapper. This will first attempt to parse the template objects JSON, and then the markup objects JSON. It will then pass on these parsed object arrays to the resolveAllMarkupObjects() function to be processed. Finally, it will take the result text from resolveAllMarkupObjects() and run it through replaceTagStringSubstitutions(). The resulting text is then returned.

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.

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 function is not directly involved in the code-generation process. Instead, it is a utility function to help developers prepare their 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 function is not directly involved in the code-generation process. Instead, it is a utility function to help developers prepare their 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().

resolveAllMarkupObjects():

Inputs:

- markupObjects: This is an array of the markup objects
  to be acted on. Each markup object contains
  a template name, and an array of data objects
  (which are the data to feed into the templates).
- templateObjects: This is an array of template objects.

Output:

- This function will return an array with 2 values,
  to be destructured by the caller.
- The first value will be a boolean, which is true
  if there was an error, or otherwise false.
- The second value will be a string. This will be
  the generated code/script, or an error if
  there was a problem during the process.

This function will loop through the given array of “markup” objects. Each markup object will have a “template” property (the name of the template to use) and a “data” property (an array of “data” objects). For more information on how these properties are structured and used, see the description for the scriptKnapperMain() function.

The object is then passed to checkForMarkupObjectError(), to be checked for any more errors.

The object is then checked to see if it doesn't have a data array, or if the data array is empty. If this is the case, the markup object is given a data array with a single empty object (this may happen if the user is calling a template that doesn't require any data).

Then, the object is passed to feedDataObjectsIntoTemplate(), to be resolved and added to the final script.

feedDataObjectsIntoTemplate():

Inputs:

- templateObject: This is the object with the template name, and the template itself.
- markupObject: This is an array of data objects, to be fed into the template, one by one.
- allTemplateObjects: This is an array of all of the template objects (This is needed when calling resolveInnerTemplateCalls()).

Output:

- This function will return an array with 2 values, to be destructured by the caller.
- The first value will be a boolean, which is true if there was an error, or otherwise false.
- The second value will be a string. This will be the result code after it has been generated (the result for all of the data objects for this template call), or it will be an error if there was a problem during the process.

This function will loop through the given “markup” object's “data” array, attempting to populate the requested template with each data object. The result string will be returned with a populated template for each “data” object.

First, the function will pass the template and the data object to the addDataObjectAdditionsFromTemplate() function. This will search through the template for any data-addition tags (see the user guide for information on these). The new data from these tags will be added to the current data object. A new version of the template is then generated by the removeDataAdditionTags() function, which will remove these tags.

Next, the template and the data object are fed into the populateTemplateWithGivenData() function. This will search for the data tags in the template, and replace them with the correct values from the data object.

Then, the template and the data object are passed into the resolveInnerTemplateCalls()

function. This will search through the template for any inner-template calls (a template call can be included in the template itself, or fed into a template through a property in the data object). This function has to be ran after the populateTemplateWithGivenData() function. If it is ran before, and the data object had a property that contained an inner-template call, then that call would not get resolved.

populateTemplateWithGivenData():

Inputs:

- dataObject: This is the object with the data to be entered into the template.
- templateName: This is string, which is used when generating errors.
- template: This is a string, which is the template to use.

Output:

- This function will return an array with 2 values, to be destructured by the caller.
- The first value will be a boolean, which is true if there was an error, or otherwise false.
- The second value will be a string. This will be the template after it has been populated with the given data, or an error if there was a problem populating the template.

This function will go through the text of a given template and find any data tags (but it will ignore any inner-template calls). It will then extract the property name from the tag, and then replace that tag with the corresponding value from the data object.

resolveInnerTemplateCalls():

Inputs:

- thisIterationResultText: The text that may contain inner-template calls.
- dataObject: This is the data object for the current data iteration.
- templateObjects: This is the array of template objects.

Output:

- This function will return an array with 2 values, to be destructured by the caller.
- The first value will be a boolean, which is true if there was an error, or otherwise false.
- The second value will be a string. This will be the given text, with inner-templates resolved, or an error if there was a problem.

This function will return the given template string, but with any inner-template calls populated.

When this function finds an inner-template call, it first extracts any data objects from the call, and then uses the mergeObjects() function to replace each one with a new data object. Each new data object will be the inner-template's object, merged with the current data object that was passed into this function (The mergeObjects() function will prioritise the inner-template's data. This means that, if the current data object contains a property called “prop1”, and the inner-template's data object also has a property called “prop1”, the value of “prop1” in the new object will be taken from the inner-template call.).

Then, any inner-template calls (along with the merged data objects) will be fed into the resolveAllMarkupObjects() function. The results from this will then be inserted into the template string, replacing the inner-template call tags.

addDataObjectAdditionsFromTemplate():

Inputs:

- The current data object, as a single object.
- The current template, as a string.

Output:

- If there is an error, the function will return an array. The first index will be true, to say there has been an error, and the second will be the error text.
- If there is not an error, the function will return an array. The first index will be false, to say there hasn't been an error, and the second will be the new version of the data object, with the added properties.

This function will go through the given template string, searching for data-addition tags. Once it finds a tag, it will extract the property name and value from it, and add this property to the passed “data” object. This is done using the mergeObjects() function, so the value in the new data will be prioritised (meaning that if the current data object has a “prop1” property, and the data-addition tag is trying to add a property named “prop1”, the returned object's “prop1” property will have the value from the tag, not from the original data object).

Once all tags have been found and added, then the new data object is returned.

removeDataAdditionTags():

Inputs:

- The string to remove the data addition tags from

Output:

- If there is an error, the function will return an array. The first index will be true, to say there has been an error, and the second will be the error text.
- If there is not an error, the function will return an array. The first index will be false, to say there hasn't been an error, and the second will be a string of text, with the addition tags removed

This function will go through the given template string, searching for data-addition tags, and then remove them from the string. It will then return this new string.

checkForMarkupObjectError():

Inputs:

- markupObject: The markup object to be checked for errors
- templateObjects: An array of template objects.

Output:

- This function will return an array with 2 values,
  to be destructured by the caller.
- The first value will be a boolean, which is true
  if there was an error, or otherwise false.
- The second value will be a string. This will be
  an error if there was a problem, or otherwise a
  blank string.

This function will check the given “markup” object for any errors. First, it will check that it has a “template” property. It will then check that the template exists, and that there isn't more than one template object with that template name. Finally, it will check that the object's “data” property (if it has a “data” property) is an array.

findObjectStringLength():

Inputs:

- objectText: The string of text that will contain the object. This string should begin with the opening brace.
- startBraceString: The string that represents an opening brace.
- endBraceString: The string that represents a closing brace.

Output:

- This function will return the length of the substring that makes up this object (including the braces).
- If the object in the string is not complete, the string doesn't start with the opening brace, or the string doesn't contain an object at all, this function will return -1.

This function will return the character length of the first object tag within the given string (including the tag's braces – which are passed to the function as the startBraceString and endBraceString parameters). This string will need to begin with the opening tag of the object, and then the function will go through the string until it finds the end of the object (while ignoring any inner object tags within the object string).

mergeObjects():

Inputs:

- objects: An array of objects to merge. The objects are prioritised based on their order in the array. So if the first and second objects both have a property with the same name, the value in the first object will be passed into the new object for that property.

Output:

- This function will return an object, that is a merge of the provided objects.

This function will merge a given array of objects into one object. If 2 or more objects in the array have the same property, then the new object will take the value from the first of those objects in the array (So, if an array has 2 objects, and both have a property with the same name, the value from the first object will be used for this property.).

prepareErrorMessage():

Inputs:

- message: A message to be included in the error.
- templateName (optional): If this error occurred while working on a template, the template name can be passed here, so the error can include it.
- dataJSONString (optional): The JSON object (parsed to string), that contains the data that was being used to populate the template, so the error can include it.

Output:

- This function will return a string.
- The string will contain the provided message within it.
- If the templateName was provided, the string will reference this.
- If the dataJSONString was provided, the string will reference this.

This function will take a basic error message string, the name of the current template, and the current data object (parsed into a JSON format). It will then turn this data into a detailed error message, to be returned to the UI.

replaceSubstrings():

Inputs:

- text: The string that needs characters/substrings replacing.
- replacements: An array of objects that define what needs to be replaced, and what to replace them with. Each object will have a "from" property (the substring to search for) and a "to" property (the substring to replace the "from" with).

Output:

- This function will return a string.
- The string will be the contents of the "text" parameter, except with the requested replacements made.

This function will go through each of the given replacement objects, and make the replacements in the given string. Each object has a “from” property and a “to” property, and every instance of the “from” property will be replaced by its “to” property.

replaceTagStringSubstitutions():

Inputs:

- text: The text that contains the substitutions to be replaced.

Output:

- This function will return a string, that will have any tag substitution strings replaced with the correct tags.

This function will replace any tag substitution strings within the given text. The user can use these substitution strings when they would actually like to use a string that is a tag (e.g. they could use “@odhb:” instead of “{{:”). This function is ran by the scriptKnapperMain() function after all of the tag replacements in the template strings have been made.