|  |  |
| --- | --- |
|  |  |

**Coding Conventions (Server)**

**Dokumentverantwortliche**Timo Notheisen

# Inhaltsverzeichnis

Inhaltsverzeichnis 2

1 File Meta Structure 3

2 Internal File Structure 3

2.1 JavaScript Source Files 3

2.1.1 Naming Conventions 3

2.1.2 Private/Anonymous Functions 3

2.1.3 Synchronous Code 3

2.1.4 Comments 4

2.1.5 Functions vs. Promises 4

Version History 5

# File Meta Structure

## File Location

All custom production code that is written for the server of Knownana should be located in the lib folder. All server-tests should be written into the file server.spec.js, which is located in the root folder of the server (repository path: development/server).

## File/Folder Names

Folder names as well as file names should written in snake\_case. In case that a file contains the definition of a ‘class’, its name starts with a capital letter (e.g. ‘Class\_name.js’)

# Internal File Structure

The content of JavaScript Files should have the following ordering:

* Module documentation (see chapter 2.4.1.1)
* nodeJS require-statements
* definition of constants
* definition of variables that store data
* definition of public functions
* definition of private functions
* assignments to the export object.

## Naming Conventions

* All variable and function names are written in camelCase
* Constructor names are written in PascalCase
* ‘Class’names are written in PascalCase

## Private/Anonymous Functions

In order to enable a good test coverage of the server code with unit tests, private and anonymous functions should be avoided. By making them public, they can be mocked by the tester.

## Synchronous Code

In absolutely no case write synchronous code that accesses the file system or does any other time-consuming operations. Since nodeJS is working in a single thread, this may cause the server to slow down significantly.

One exception is code that is only executed once during the startup/shutdown of the server.

## Comments

Since all code should follow the Clean Code principle, it oughtn’t be necessary to write any comments that explain the behavior of the code. But because JavaScript lacks of a type safe way to define functions, we decided to prescribe the documentation of all methods by using JSDoc[[1]](#footnote-1). This documentation should contain the following information:

* Is the function **synchronous** or **asynchronous**?
* Is the function meant to be called **locally** or **publicly**?
* What is the expected type of the different parameters? (Number, String, Object, Array, Function)

In case of a function: how does the expected signature of the method look like?

* How is the return value of the function passed to the caller? (Callback, Promise, Return Value, Nothing)
* If a JSON object is expected as parameter, or returned as return value, the format of the JSON has to be explained

This chapter describes, which documentation tool is used and how comments have to look like when using this tool.

### JSDoc

In the context of the knowledgebase project, JSDoc3 is used as the code documentation tool. The syntax of JSDoc is well supported by the PhpStorm IDE which is used by all knowledgebase developers. This document lists all guidelines for code documentation which must be followed by backend developers.

#### Modules

The module documentation must start in the first line of its JS-file and should look similar to the following example:

*/\*\*  
 \* Search engine wrapper that is used in order to communicate*

*\* with the search engine. This is the only module which may*

*\* access the search engine api directly.  
 \*  
 \** ***@module*** *lib/search\_engine\_connector  
 \** ***@author*** *Vladislav Chumak  
 \*/*

Between the description and the first @-Tag must be an empty line. If the module residents in subdirectories, the names of all subdirectories have to be included in the module name separated by slashes like in the example above.

#### Functions

All functions which are exported by a module have to be documented. The following examples should be used as documentation guideline.

**Function with no parameters and no return**

*/\*\*  
 \* Instructs the search engine to update its search index.  
 \*  
 \** ***@function*** *updateIndex  
 \** ***@static*** *\*/****searchEngineConnector***.updateIndex = **function** () {  
 *//...*}

Between the description and the first @-tag must be an empty line. The name of the function must follow right after the @function tag delimited by one space. Otherwise JSDoc will not recognize the statement above as a function. @static tag is also required, because it indicates that there is no need to instantiate anything with new operator in order to use this function.

**Function with parameters and return**

*/\*\*  
 \* Searches for article files which match the given key words.  
 \*  
 \** ***@function*** *searchArticles  
 \** ***@static*** *\** ***@param*** *{string} q - The search query which contains the key words.  
 \** ***@returns*** *{string[]} Search results.  
 \*/****searchEngineConnector***.searchArticles = **function** (q) {  
 *//...*}

After the @param tag must follow the name of the type surrounded by curly braces, the name of the parameter, and the corresponding description text. There are the following basic types available: boolean, number, string, Object, Date, RegExp, Promise, Error. The upper and lower case must be considered.

If the type is an array, square brackets should be appended to the type of array entries like in the example above.

**Function with custom types as parameter/return value**

*/\*\*  
 \* Represents a single search result entry for a search query.*

*\* The entry references a file on the file system which was matched*

*\* to the search query by search engine.  
 \*  
 \** ***@typedef*** *SearchResultEntry  
 \** ***@property*** *{number} id - ID of the article.  
 \** ***@property*** *{string} filename - Name of the file which matches the search query.  
 \** ***@property*** *{string} text - The snippet of the file content which contains the matched text for the search query. The matched key words are wrapped in &lt;b&gt;.  
 \*/  
  
/\*\*  
 \* Searches for article files which match the given key words.  
 \*  
 \** ***@function*** *searchArticles  
 \** ***@static*** *\** ***@param*** *{string} q -The search query which contains the key words.  
 \** ***@returns*** *{Promise<module:lib/search\_engine\_connector~SearchResultEntry*

*[]|Error>} Search results.  
 \*/****searchEngineConnector***.searchArticles = **function** (q) {  
 *//...*}

Complex Objects have to be documented by means of @typedef tag which requires its own documentation block. After @typedef documentation block the defined type can be used in type specifications of @param and @returns tags as types. The type which is defined in typedef should be referenced in curly brackets including its module, e.g.:

*{module:lib/search\_engine\_connector~SearchResultEntry}*

Even if the syntax above looks cumbersome, it still ensures that the generated documentation contains valid linking between type references and type definition.

**Functions with callback as parameter**

*/\*\*  
 \* Callback for error handling.  
 \*  
 \** ***@callback*** *errorCallback  
 \** ***@param*** *{Error} error - The error of the search operation.  
 \*/  
  
/\*\*  
 \* Searches for article files which match the given key words.  
 \*  
 \** ***@function*** *searchArticles  
 \** ***@static*** *\** ***@param*** *{string} q - The search query which contains the key words.  
 \** ***@param*** *{errorCallback} errorCallback - Callback which will be notified if the search operation fails.  
 \** ***@returns*** *{SearchResultEntry[]} Search results.  
 \*/****searchEngineConnector***.searchArticles = **function** (q, errorCallback) {  
 *//...*}

Callbacks have to be documented by means of @callback tag which requires its own documentation block. After @callback documentation block the name of the callback can be used in type specifications of @param and @returns tags.

### Return Values (Functions vs. Promises)

Depending on several conditions (see below), the return type of a function has to be chosen. But no matter which return type is chosen, it *has to be documented* in JSDoc above of the function.

**Functions with private purpose**

Functions that are meant to be called only locally should return their result, as well as any potential error message in a callback function. This callback function has to have the following signature:

function (err, result)

where err is an error message, if an error occurred, and null if not. result contains the actual return value – or null, if an error occurred. The name of the callback function parameter in the signature of a method should be cb.

If a private function doesn’t have any return value and no error could possibly happen, it doesn’t need to provide a way to pass a callback function.

**Functions with public purpose**

Functions that are meant to be called publicly should return their result, as well as any potential error message in form of a JavaScript Promise[[2]](#footnote-2).

If a function (private or public) doesn’t have any return value and no error could possibly happen, it doesn’t need support callbacks/Promises.

If a function (private or public) is synchronous and is not expected to throw any errors, it can as well return the return value directly.

# Version History

|  |  |  |
| --- | --- | --- |
| **Version** | **Date** | **Description** |
| 1.0 | 07.06.2016 | Document creation |

1. https://en.wikipedia.org/wiki/JSDoc [↑](#footnote-ref-1)
2. https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global\_Objects/Promise [↑](#footnote-ref-2)