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

**Coding Conventions**

**Authors**

Timo NotheisenVladislav Chumak

Christoph Brutscher

# Table of Contents

Table of Contents 2

1 Code documentation 3

1.1 Modules 3

1.2 Functions 4

1.2.1 Function with no parameters and no return 4

1.2.2 Function with parameters and return 4

1.2.3 Function with custom types as parameter/return value 4

1.2.4 Functions with callback as parameter 5

2 Backend Conventions 6

2.1 File Meta Structure 6

2.1.1 File Location 6

2.1.2 File/Folder Names 6

2.2 Internal File Structure 6

2.2.1 Naming Conventions 6

2.2.2 Private/Anonymous Functions 7

2.2.3 Synchronous Code 7

2.2.4 Return Values (Functions vs. Promises) 7

3 Frontend Conventions 8

3.1 File Meta Structure 8

3.1.1 File Location 8

3.1.2 File/Folder Names 9

3.2 Internal File Structure 9

3.2.1 JavaScript 9

3.2.2 Naming Conventions 9

3.2.3 JSHint 9

3.3 HTML/CSS 9

3.4 AngularJS 10

3.4.1 Controllers 10

3.4.2 Directives 10

3.4.3 Services 11

3.4.4 Routes 11

3.4.5 $rootScope 11

Version History 12

# Code documentation

Since all code should follow the Clean Code principle, it oughtn’t to 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

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 frontend and 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 named functions 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.

# Backend Conventions

This section describes all conventions relevant for backend developers.

## File Meta Structure

### File Location

All custom production code that is written for the server of Metis 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 be written in snake\_case. In case that a file contains the definition of a ‘class’, its name must start 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.

### 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 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.

# Frontend Conventions

This section describes all conventions relevant for frontend developers.

## File Meta Structure

### File Location

All custom Files (including but not limited to: JavaScript, HTML, SCSS, txt and config files) which will be published to the client should be put into the **public** folder. All config files required for development and deployment (e.g. gulp, bower) are stored in the root directory. The directories **npm modules** and **bower modules** should not be touched as they are overwritten by the respective package management tools.

The structure of the code which will later be executed by the browser is as follows:

* **public**
  + **modules**
    - Contains all AngularJS modules defined by the developer
    - **core**
      * The core module contains all views, directives, controllers and services needed to run the knowledgebase front end as well as AngularJS startup configurations.
      * **css**
        + Contains all styles of the front-end in the SASS (.scss) format
      * **js**
        + Contains folders for directives, services and controllers as well as configuration-files for the routes
      * **resources**
        + All static resources such as images or vector graphics should be allocated here
      * **views**
        + Views for routes as well as templates for directives are to be located here.
    - **tests**
      * The tests folder contains all files used to execute karma test cases

### File/Folder Names

Folder names should be written as snake\_case and file names in camelCase. If a file is a special (e.g. a directive template) then the name should reflect that with a classname appropriate to the subtype appended to the filename (f.e. componentX.template-html).

## Internal File Structure

### JavaScript

The content of JavaScript Files should have the following ordering:

* Module documentation
* definition of constants
* definition of variables
* definition of public functions
* definition of private functions

### Naming Conventions

* All variable and function names are written in camelCase

### JSHint

This project makes use of the JSHint framework to further enforce good coding conventions. The following options have been activated:

|  |  |
| --- | --- |
| Bitwise | Prohibits use of bitwise operators (^, |, &) |
| Camelcase | Enforces camelCase variable syntax |
| Curly | Enforces the use of {} around codeblocks |
| Eqeqeq | Prohibits the use of != and == |
| Undef | Prohibits the use of nonexplicitely declared variables |
| Strict | Enforces ECMAScript 5 strict mode |

## HTML/CSS

* All <input>, <a> and all other interactable html-tags need an id attribute to be easily addressable by selenium tests
* SASS is the preferable way to define styles
* Inline css is strictly forbidden
* Colors and other style variables with global relevance should be allocated in a separate .scss file
* !important clauses and overriding styles from frameworks such as bootstrap is discouraged

## AngularJS

Since AngularJS Framework is at the heart of the front-end, yet does not enforce any hard coding standards is reason to further define conventions for handling code inside the framework.

### Controllers

#### Naming

Controller’s names are always written in CamelCase and end with a “Ctrl”

#### Usage & Functionality

A controller should contain no functionality to directly manipulate the DOM. Its sole purpose is the providing, transformation and manipulation of the data within the model ($scope).

A controller by its own should only be used as a root management tool within a view to distribute data between directives as well as orchestrating them.

#### Dependency Injection

To share code between different controllers the built in AngularJS dependency injections has to be used. Dependency has to be declasred as a string as well as a parameter, for example:

***Angular.module(“core”)***.controller(“ArticleCtrl”, [“ServiceX”, **function**(ServiceX){  
 *//...*This way any issues arising from minifying/uglifying the code can be circumvented.

### Directives

The UI should be separated into components (directives) as much as possible and as long as it does not hinder productivity.

#### Scope

Directives should always use an isolated scope if possible bindings as loosely coupled possible (string-binding -> one-way-binding -> two-way-binding).

Scope variables which are bound should always have the same name as their respective attribute, for example:

***scope : {***

***variable1 : “@”,***

***variable2 : “=”***

***}***

***//…***

***<directive variable1=”{{String}}” variable2=”scopeVar”>***

#### Template binding

Templates if necessary are not allowed to be directly declared inside the directives javascript code. Rather the template is to be declared in a separate file and bound by the ***templateURL*** attribute.

### Services

Services should be used to connect the front to the back end. All REST calls are to be declared within services to make them globally available. A REST call should be defined within the service so it returns a promise to the calling controller.

### Routes

A route should be self-descriptive and informative on its own. Every state of a view is to be reflected in its url. The controller managing a view has to be declared inside router and not the view.

### $rootScope

The $rootScope should be viewed as a read only directory and only used for values which stay constant throughout the browser session. If information has to be shared please use either cookies, services or $broadcasts.

# Version History

|  |  |  |
| --- | --- | --- |
| **Version** | **Date** | **Description** |
| 1.0 | 14.06.2016 | Document created |

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)