Chapter 1: JavaScript Coding Guidelines

Here, you will find an explanation of the JavaScript Coding Guidelines we use. Generally, we strive to follow the FLOW3 Coding Guidelines as closely as possible, with exceptions which make sense in the JavaScript context.

This guideline explains mostly how we want JavaScript code to be formatted; and it does *not* deal with the TYPO3 Phoenix User Interface structure. If you want to know more about the TYPO3 Phoenix User Interface architecture, have a look into the "Phoenix User Interface Development" book.

1 Naming Conventions

- one class per file, with the same naming convention as FLOW3.

  This means all classes are built like this: `F3.`*`PackageKey`*`.`*`SubNamespace`*`.`*`ClassName`*, and this class is implemented in a JavaScript file located at *`...`*
  `Package``/.../JavaScript/`*`SubNamespace`*`/`*`ClassName`*`.js`.

  Right now, the base directory for JavaScript in FLOW3 packages `Resources/Public/JavaScript`, but this might still change.

  > **Note** In TYPO3 v4, the prefix is `TYPO3`. This means an extension should use `TYPO3.ExtensionKey` as base namespace, while the TYPO3 Core should use `TYPO3.Backend` as namespace.

- We suggest that the base directory for JavaScript files is `JavaScript`.
- Files have to be encoded in UTF-8.
- Classes and namespaces are written in `UpperCamelCase`, while properties and methods are written in `lowerCamelCase`.
- The xtype of a class is always the fully qualified class name. Every class which can be instanciated needs to have an xtype declaration.
- Never create a class which has classes inside itself. Example: if the class `F3.TYPO3.Foo` exists, it is prohibited to create a class `F3.TYPO3.Foo.Bar`.You can easily check this: If a directory with the same name as the JavaScript file exists, this is prohibited.

  Here follows an example:

  ```
  F3.TYPO3.Foo.Bar // implemented in .../Foo/Bar.js
  F3.TYPO3.Foo.Bar = ...

  F3.TYPO3.Foo // implemented in ...Foo.js
  F3.TYPO3.Foo = ..... overriding the "Bar" class
  ```

  So, if the class `F3.TYPO3.Foo.Bar` is included *before* `F3.TYPO3.Foo`, then the second class definition completely overrides the `Bar` object. In order to prevent such issues, this constellation is forbidden.

- Every class, method and class property should have a doc comment.
- Private methods and properties should start with an underscore (_) and have a `@private` annotation.

2 Doc Comments

Generally, doc comments follow the following form:

```
/**
 *
 */
```

See the sections below on which doc comments are available for the different elements (classes, methods, ...).

We are using http://code.google.com/p/ext-doc/ for rendering an API documentation from the code, that's why types inside `@param`, `@type` and `@cfg` have to be written in curly brackets like this:

```
@param {String} theFirstParameter A Description of the first parameter
@param {My.Class.Name} theSecondParameter A description of the second parameter
```

Generally, we do not use `@api` annotations, as private methods and attributes are marked with `@private` and prefixed with an underscore. So, *everything which is not marked as private belongs to the public API!*

We are not sure yet if we should use `@author` annotations at all. (TODO Decide!)

TODO: find out how to make cross-references, or how to nicely format multiline-text inside doc comments (@link?).

3 Class Definitions

Classes can be declared singleton or prototype. A class is *singleton*, if only one instance of this class will exist at any given time. An class is of type *prototype*, if more than one object can be created from the class at run-time. Most classes will be of type *prototype*.

You will find examples for both below.

3.1 Prototype Class Definitions

Example of a prototype class definition

```
Ext.ns("F3.TYPO3.Content");(1)

/*                                                                    *
 * This script belongs to the FLOW3 package "TYPO3".                  *
 *                                                                    *
 * It is free software; you can redistribute it and/or modify it under *
 * the terms of the GNU General Public License as published by the Free *
 * Software Foundation, either version 3 of the License, or (at your  *
 * option) any later version.                                         *
 *                                                                    *
 * This script is distributed in the hope that it will be useful, but *
 * WITHOUT ANY WARRANTY; without even the implied warranty of MERCHAN- *
 * TABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General  *
 * Public License for more details.                                   *
 *                                                                    *
 * You should have received a copy of the GNU General Public License  *
 * along with the script.                                             *
 * If not, see http://www.gnu.org/licenses/gpl.html                   *
 *                                                                    *
 * The TYPO3 project - inspiring people to share!                     *
 *                                                                    */

/**
 * @class F3.TYPO3.Content.FrontendEditor(2)
 *
 * The main frontend editor.(3)
 *
 * @namespace F3.TYPO3.Content(4)
 * @extends Ext.Container
 */
F3.TYPO3.Content.FrontendEditor = Ext.extend(Ext.Container, {(5)
 // here comes the class contents
});
Ext.reg('F3.TYPO3.Content.FrontendEditor', F3.TYPO3.Content.FrontendEditor);(6)
```

(1) At the very beginning of the file is the namespace declaration of the class, followed by a newline.

(2) Then follows the class documentation block, which *must* start with the `@class` declaration in the first line.

(3) Now comes a description of the class, possibly with examples.

(4) Afterwards *must* follow the namespace of the class and the information about object extension.

(5) Now comes the actual class definition, using `Ext.extend`.

(6) As the last line of the class, it follows the xType registration. We always use the fully qualified class name as xtype

Usually, the constructor of the class receives a hash of parameters. The possible configuration options need to be documented inside the class with the `@cfg` annotation:

```
F3.TYPO3.Content.FrontendEditor = Ext.extend(Ext.Container, {
 /**
  * An explanation of the configuration option followed
  * by a blank line.
  *
  * @cfg {Number} configTwo
  */
 configTwo: 10
 ...
}
```

3.2 Singleton Class Definitions

Now comes a singleton class definition. You will see that it is very similar to a prototype class definition, we will only highlight the differences.

Example of a singleton class definition

```
Ext.ns("F3.TYPO3.Core");

/*                                                                    *
 * This script belongs to the FLOW3 package "TYPO3".                  *
 *                                                                    *
 * It is free software; you can redistribute it and/or modify it under *
 * the terms of the GNU General Public License as published by the Free *
 * Software Foundation, either version 3 of the License, or (at your  *
 * option) any later version.                                         *
 *                                                                    *
 * This script is distributed in the hope that it will be useful, but *
 * WITHOUT ANY WARRANTY; without even the implied warranty of MERCHAN- *
 * TABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General  *
 * Public License for more details.                                   *
 *                                                                    *
 * You should have received a copy of the GNU General Public License  *
 * along with the script.                                             *
 * If not, see http://www.gnu.org/licenses/gpl.html                   *
 *                                                                    *
 * The TYPO3 project - inspiring people to share!                     *
 *                                                                    */

/**
 * @class F3.TYPO3.Core.Application
 *
 * The main entry point which controls the lifecycle of the application.
 *
 * @namespace F3.TYPO3.Core
 * @extends Ext.util.Observable
 * @singleton(1)
 */
F3.TYPO3.Core.Application = Ext.apply(new Ext.util.Observable, {(2)
 // here comes the class contents
```

```
});(3)
```

(1) You should add a `@singleton` annotation to the class doc comment after the `@namespace` and `@extends` annotation

(2) In singleton classes, you use `Ext.apply`. Note that you need to use `new` to instanciate the base class.

(3) There is *no xType* registration in singletons, as they are available globally anyhow.

3.3 Class Doc Comments

Class Doc Comments should always be in the following order:

- `@class` *`Name.Of.Class`* (required)
- Then follows a description of the class, which can span multiple lines. Before and after this description should be a a blank line.
- `@namespace` *`Name.Of.Namespace`* (required)
- `@extends` *`Name.Of.BaseClass`* (required)
- `@singleton` (required if the class is a singleton)

If the class has a non-empty constructor, the following doc comments need to be added as well, after a blank line:

- `@constructor`
- `@param {`*`type`*`}` *`nameOfParameter description of parameter`* for every parameter of the constructor

Example of a class doc comment without constructor

```
/**
 * @class F3.TYPO3.Foo.Bar
 *
 * Some Description of the class,
 * which can possibly span multiple lines
 *
 * @namespace F3.TYPO3.Foo
 * @extends F3.TYPO3.Core.SomeOtherClass
 */
```

Example of a class doc comment with constructor

```
/**
 * @class F3.TYPO3.Foo.ClassWithConstructor
 *
 * This class has a constructor!
 *
 * @namespace F3.TYPO3.Foo
 * @extends F3.TYPO3.Core.SomeOtherClass
 *
 * @constructor
 * @param {String} id The ID which to use
 */
```

4 Method Definitions

Methods should be documented the following way:

Example of a method comment

```
...
F3.TYPO3.Core.Application = Ext.apply(new Ext.util.Observable, { // this is just an examp
```

```
 ... propery definitions ...
 /**
  * This is a method declaration; and the
  * explanatory text is followed by a newline.
  *
  * @param {String} param1 Parameter name
  * @param {String} param2 (Optional) Optional parameter
  * @return {Boolean} Return value
  */
 aPublicMethod: function(param1, param2) {
  return true;
 },(1)

 /**
  * this is a private method of this class,
  * the private anotation marks them an prevent that they
  * are listed in the api doc. As they are private, they
  * have to start with an underscore as well.
  *
  * @return {void}
  * @private
  */
 _sampleMethod: function() {
 }
}
...
```

(1) There should be a blank line between methods.

Contrary to what is defined in the FLOW3 PHP Coding Guidelines, methods which are public *automatically belong to the public API*, without an `@api` annotation. Contrary, methods which do *not belong to the public API* need to begin with an underscore and have the `@private` annotation.

5 Property Definitions

All properties of a class need to be properly documented as well, with an `@type` annotation. If a property is private, it should start with an underscore and have the `@private` annotation at the last line of its doc comment.

```
...
F3.TYPO3.Core.Application = Ext.apply(new Ext.util.Observable, { // this is just an examp
 /**
  * Explanation of the property
  * which is followed by a newline
  *
  * @type {String}
  */
 propertyOne: 'Hello',

 /**
  * Now follows a private property
  * which starts with an underscore.
  *
  * @type {Number}
  * @private
  */
 _thePrivateProperty: null,
 ...
}
```

6 Code Style

- use single quotes(') instead of double quotes(") for string quoting
- Multiline strings (using \) are forbidden. Instead, multiline strings should be written like this:

```
'Some String' +
' which spans' +
' multiple lines'
```

- There is no limitation on line length.
- JavaScript Constants (true, false, null) should be written lowercase, and not uppercase.
- Custom JavaScript Constants should be avoided.
- Use a single `var` statement at the top of a method to declare all variables. Example:

```
function() {
 var myVariable1, myVariable2, someText;
 // now, use myVariable1, ....
}
TODO: Discuss the following (inline assignment):
function() {
 var myVariable1 = Ext.getComponent(...),
     variable2, variable3;
}
-> i feel it makes the code more unreadable, so I would suggest to use variant 1.
```

- We use *a single TAB* for indentation.
- Use inline comments sparingly, they are often a hint that a new method must be introduced.

  Inline Comments must be indented *one level deeper* than the current nesting level. Example:

```
function() {
 var foo;
  // Explain what we are doing here.
 foo = '123';
}
```

- White Spaces around control structures like `if`, `else`, ... should be inserted like in the FLOW3 CGLs:

```
if (myExpression) {
 // if part
} else {
 // Else Part
}
```

- Arrays and Objects should *never* have a trailing comma after their last element,
- Arrays and objects should be formatted in the following way:

```
[
 {
  foo: 'bar'
 }, {
  x: y
 }
]
```

- Method calls should be formatted the following way:

```
// for simple parameters:
new Ext.blah(options, scope, foo);
object.myMethod(foo, bar, baz);

// when the method takes a single parameter of type object as argument, and this obje
new Ext.Panel({
 a: 'b',
 c: 'd'
});

// when the method takes more parameters, and one is a configuration object which is
```

```
new Ext.blah(
 {
  foo: 'bar'
 },
 scope,
 options
);
```

-> TODO: are there JS Code Formatters / Indenters, maybe the Spket JS Code Formatter?

6.1 Using JSLint to validate your JavaScript

JSLint is a JavaScript program that looks for problems in JavaScript programs. It is a code quality tool. When C was a young programming language, there were several common programming errors that were not caught by the primitive compilers, so an accessory program called `lint` was developed that would scan a source file, looking for problems. `jslint` is the same for JavaScript.

JavaScript code ca be validated on-line at http://www.jslint.com/. When validating the JavaScript code, "The Good Parts" family options should be set. For that purpose, there is a button "The Good Parts" to be clicked.

Instead of using it online, you can also use JSLint locally, which is now described. For the sake of convenience, the small tutorial bellow demonstrates how to use JSlint with the help of CLI wrapper to enable recursive validation among directories which streamlines the validation process.

- Download Rhino from http://www.mozilla.org/rhino/download.html and put it for instance into /Users/john/WebTools/Rhino.
- Download JSLint.js (@see attachment "jslint.js", line 5667-5669 contains the configuration we would like to have, still to decide) (TODO)
- Download jslint.php (@see attachment "jslint.php" TODO), for example into /Users/fudriot/WebTools/JSLint
- Open and edit path in jslint.php -> check variable $rhinoPath and $jslintPath
- Add an alias to make it more convenient in the terminal:

```
alias jslint '/Users/fudriot/WebTools/JSLint/jslint.php
```

Now, you can use JSLint locally:

```
// scan one file or multi-files
jslint file.js
jslint file-1.js file-2.js

//scan one directory or multi-directory
jslint directory
jslint directory-1 directory-2

//scan current directory
jslint .
```

It is also possible to adjust the validation rules JSLint uses. At the end of file "jslint.js", it is possible to customize the rules to be checked by JSlint by changing options' value. By default, the options are taken over the book "JavaScript: The Good Parts" which is written by the same author of JSlint.

Below are the options we use for TYPO3 v5:

```
bitwise: true, eqeqeq: true, immed: true,newcap: true, nomen: false, onevar: true, pluspl
```

In case some files needs to be evaluated with special rules, it is possible to add a comment on the top of file which can override the default ones:

```
/*jslint white: true, evil: true, laxbreak: true, onevar: true, undef: true, nomen: true,
```

More information about the meaning and the reasons of the rules can be found at
http://www.jslint.com/lint.html

6.2 Event Handling

When registering an event handler, always use explicit functions instead of inline functions to allow overriding of the event handler.

Additionally, this function needs to be prefixed with `on` to mark it as event handler function. Below follows an example for good and bad code.

Good Event Handler Code

```
F3.TYPO3.Application.on('theEventName', this._onCustomEvent, this);
```

Bad Event Handler Code

```
F3.TYPO3.Application.on(
  'theEventName',
  function() {
    alert('Text');
  },
  this
);
```

All events need to be explicitely documented inside the class where they are fired onto with an `@event` annotation.

Here follows an example:

```
F3.TYPO3.Core.Application = Ext.apply(new Ext.util.Observable, { // this is just an examp
 /**
  * @event eventOne Event declaration
  */

 /**
  * @event eventTwo Event with parameters
  * @param {String} param1 Parameter name
  * @param {Object} param2 Parameter name
  * <ul>
  * <li><b>property1:</b> description of property1</li>
  * <li><b>property2:</b> description of property2</li>
  * </ul>
  */
 ...
}
```

Additionally, make sure to document if the scope of the event handler is not set to `this`, i.e. does not point to its class, as the user expects this.

7 ExtJS specific things

TODO

-> explain initializeObject

-> how to extend Data Stores (for example)

-> how to extend Ext components

-> datastore extended?

-> can be extended by using constructer() not initComponets() like it is for panels and so on

8 Unit Testing

TODO: http://developer.yahoo.com/yui/3/test/