Chapter 1: Architecture

In this chapter, the basic architecture of the user interface is explained.

We have some key design goals we followed while creating this architecture:

- *Extensibility:* TYPO3 is famous for its extensibility, and this of course should also apply to the TYPO3 Phoenix User Interface. In this chapter, the basic extensibility principles are explained.
- *Loading Order Independence:* Often, big JavaScript applications are very fragile, or even break, because of wrong inclusion order of their JavaScript files. Because our system is highly extensible, much work has been done to make sure the whole system has only very small loading order dependencies, i.e. for most files, the loading order is completely irrelevant, but still providing predictable results.

1 JavaScript Modules

All JavaScript is encapsulated in so-called *modules*, which are packages of (JavaScript) functionality. Modules are built in a way such that if you add or remove a module, there will be no JavaScript errors (if the module itself does not contain syntax errors of course).

The base directory where all built-in TYPO3 Backend modules are located is in `TYPO3/Public/Backend/JavaScript`.

Each module has a *Module Descriptor* file (TODO: Name?) which is named *`Modulename`*`Module.js`, i.e. the module descriptor for the *Content* module can be found inside `Content/ContentModule.js`. The Module Descriptor is a *singleton* and also acts as an *event bridge* for the module.

Every module descriptor can have the following two methods:

- `configure(F3.TYPO3.Core.Registry)`: this method should do all changes to the central registry (which gets passed as argument) inside this method. See the sectioon on the Registry for more explanation what it does. Essentially, the registry is just an extensible JSON structure which is used at many places to configure the TYPO3 Backend User Interface.
- `initialize(F3.TYPO3.Core.Application)`: in this method, the module descriptor can register itself to be called after another dependent module has been initialized, using the `application.afterInitializationOf(`*`moduleName`*`, `*`callback`*`, `*`scope`*`)` method.
- Additionally, the module descriptor exposes the public API of this module.
- And, of course it can also have private methods used to structure the code.

Let's look at an example for a module descriptor:

Example Module Descriptor

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

F3.TYPO3.Core.Application.createModule('F3.TYPO3.Dummy.DummyModule', {

 configure: function(registry) {
  registry.append('menu[main]', 'report', {(1)
   title: 'Report',
   itemId: 'report'
  });
 },

 initialize: function(application) {
  application.afterInitializationOf('F3.TYPO3.UserInterface.UserInterfaceModule', functio
   userInterfaceModule.addContentArea('report', 'dummy', {(3)
    xtype: 'F3.TYPO3.Dummy.DummyContentArea',
    name: 'Report'
   });
   userInterfaceModule.contentAreaOn('menu[main]/report', 'report', 'dummy');(4)
 }
});
```

(1) Inside the `configure` method, a new module called *report* is being added to the registry.

(2) The `DummyModule` needs the `UserInterfaceModule` to work correctly; that's why it expresses a dependency inside the `initialize` method, using the `application.afterInitializationOf(...)` method. The specified callback function is only executed if the `UserInterfaceModule` is present and has been loaded.

(3) Now follows the other module's specific initialization code. The `UserInterfaceModule` has a method `addContentArea(`*`mainTabName`*`, `*`name`*`, `*`configuration`*`)` which adds a new content area to the specified main tab name.

(4) The `UserInterfaceModule` has another method `contentAreaOn(`*`nameOfActivatedElement`*`, `*`mainTabName`*`, `*`name`*`)`, which is used to show the content area if a certain element, specified through the first parameter, is activated.

TODO: Separate Core from the rest, and move to other package!

TODO: add doc comments to example module descriptor

2 The Registry

The TYPO3 backend is mainly extensible because of a central component which stores how the User Interface is built together: the *registry*. This is a singleton object available inside `F3.TYPO3.Core.Registry`, and is thus accessible in all components.

The Registry is an extensible JSON structure.

It works in two steps: First, all modules can add / change things inside the registry inside the `configure` method of their Module Descriptor. After that, the registry is *compiled*, and after that, is just a regular JSON structure. Let's follow an example what the registry does, so you can gain an intuitive understanding of it:

```
// Inside a Module Descriptor
configure: function(registry) {
 registry.set('some/path/foo', {title: 'Hello'});
 // this is the same as:
 registry.set('some/path/foo/title', 'Hello');
}
// after the registry is compiled, it looks like:
{ some: { path: {foo: {title: 'Hello'}}}}
```

So far, this is what you would expect. Now follows an example which shows the interaction between multiple modules:

```
// Inside a Module Descriptor
configure: function(registry) {
 registry.set('some/path/foo/title', 'World', 10);
}
// Inside another module descriptor:
configure: function(registry) {
 registry.set('some/path/foo', {title: 'Hello'});
}

// after the registry is compiled, it looks like:
{ some: { path: {foo: {title: 'World'}}}}
```

If two modules try to set the same key, somehow it has to be determined which value "wins" -- and as we do not want to depend on the loading order, we have added the concept of *priorities* to the registry. In the above example, you see that in the result, the string `World` is displayed because it has a priority of 10, and the other call to set the same key has no priority set (which defaults to a priority value of zero). If the two `registry.set` calls were executed in a different order, the result after the compilation would still be the same. So, *priorities make sure the registry behaves deterministically*.

Let's say we want to remove an element from the registry -- for that, there is the `registry.remove` call:

```
// Inside a Module Descriptor
configure: function(registry) {
 registry.remove('some/path/foo/description', 'foo');
}
// Inside another module descriptor:
configure: function(registry) {
 registry.set('some/path/foo', {
  title: 'Hello',
  description: ' This is a long description'
 });
}

// after the registry is compiled, it looks like:
{ some: { path: {foo: {title: 'Hello'}}}}
```

What happened here? Although the element is removed before it is actually inserted, the result is still what one would expect. If you look closely, you will notice that we did *not* specify a priority for `registry.remove` -- and still the result is what we want. The reason is that the registry processes all operations in a strictly defined order -- and the `delete` operation is processed after all other operations. For a single operation, the priorities are then used to determine the processing order.

Note All registry operations can deal with priorities, but we left them out to not complicate the examples below.

So far, you have seen how to build JSON objects with the registry -- and now we will look at creating arrays with the registry. In the next example, you will see a simple invocation of the `registry.append` method.

```
// Inside a Module Descriptor
configure: function(registry) {
 registry.append('menu/main', 'edit', {title: 'Edit the element'});
}
// after the registry is compiled, it looks like:
{
 menu: {
  main: [
   {
    title: 'Edit the element',
    key: edit
   }
  ]
 }
}
```

So, what happened now? Because we used `append`, the registry assumes that `menu/main` is an *array*, and no object anymore. The second argument of `append` is an *array key*, which can be used to reference this array element later inside a path.

When you look at the result, you will see that menu/main is really an array now, and the array element is again an object with the title we specified. Additionally, the array element gets the array key inserted under the special name `key`. Let's look how this array key can be used to modify objects:

```
// Inside a Module Descriptor
configure: function(registry) {
 registry.append('menu/main', 'edit', {title: 'Edit the element'});
}
```

```
// Inside another module descriptor:
configure: function(registry) {
 registry.set('menu/main/edit/title', 'Element bearbeiten');
}

// after the registry is compiled, it looks like:
{
 menu: {
  main: [
   {
    title: 'Element bearbeiten',
    key: edit
   }
  ]
 }
}
```

Here, you see that `set` is used to change the title of the `edit` element inside the array (again, the order of the registry statements does not matter).

> **Note** There is also a `prepend` method which inserts an element at the beginning of an array, and not at the end like `append`.

Often, one does not want to insert an element at the end or at the beginning of an array, but somewhere before or after an element. The registry supports the two operations `insertAfter` and `insertBefore` for exactly that: To insert a sibling of a given node. Let's look at an example:

```
// Inside a Module Descriptor
configure: function(registry) {
 registry.insertAfter('menu/main/edit', 'preview', {title: 'Preview'});
}
// Inside another module descriptor:
configure: function(registry) {
 this.registry.append('menu/main', 'edit', {title: 'Edit'});
 this.registry.append('menu/main', 'delete', {title: 'Delete'}, 10);
}

// after the registry is compiled, it looks like:
{
 menu: {
  main: [{
   title: 'Edit',
   key: 'edit'
  }, {
   title: 'Preview',
   key: 'preview'
  }, {
   title: 'Delete',
   key: 'delete'
  }]
 }
}
```

Here, you see that "preview" has been inserted after "edit".

2.1 Logical objects and the "children" property

If one looks at JSON objects which form a tree structure, they often look like the following:

```
{
 menu: {
  title: 'Home',
  children: [
   {
    key: 'company',
    title: 'Our Company',
    children: [
     {
      key: 'partners',
      title: 'Partners'
     }
    ]
   }, {
    id: ...,
    title: ...
   }
  ]
 }
}
```

So basically, an object which has children often stores these in the special property `children`. Now, let's imagine we want to add a new child to `company` using the registry, then this is possible with `registry.append('menu/children/company/children', 'investor-relations', {title: 'Investor Relations'})`. However, this is quite unreadable because of the many `.../children/...` sections in the path. For that, we invented a little syntactic sugar, so you can re-write `menu/children/company/children` as `menu[]/company[]`, which is a lot more readable.

Now imagine we would rename the `children` property to `childNodes` -- in this case, instead of writing `menu/childNodes/company/childNodes`, one could write `menu[childNodes]/company[childNodes]`. Although this is semantically equivalent, it is better readable because it says: "I want to go to the `childNodes` property of the `menu` object, and from there I want to go to the `childNodes` property of the `company` object."

So, if you go to a real sub-object, use the slash as delimiter, but if you just traverse into a more complex object, use the bracket-syntax.

2.2 special paths in the registry

TODO: maybe this should be moved somewhere else, later.

This section explains the basic layout and structure of the registry.

- `menu`: Contains all menu definitions
  - `main`: Contains the main menu displayed in the top area of the Backend
- `schema/`*`ContentType`*: schema definition for the content type
  - `service`: describes the endpoints which should be used for showing, updating, ceating and deleting data.
  - `properties`: describes for each property its type and the validations which apply.
- `form`: Configuration for forms
  - `type/`*`ContentType`*: Form definitions for the given content type
    - `standard`: standard form definition which is used by default
    - There might be additional form definitions for specific places in the backend.
  - `editor/`*`Type`*: Editor configuration for the given type.

3 Events

The different components of the user interface communicate via events. A module fires some events, and other modules listen to these events and do some specific actions.

Every Module Descriptor inherits from `Ext.util.Observable`, so there can be events thrown on these objects, and one can listen to these events. Here is a quick example how that works, which should be familiar to everybody knowing `Ext.util.Observable`:

```
// Register an event listener
F3.TYPO3.Core.Application.on('logout', this._onLogout, this);

// at some other place, the event is fired, which triggers all registered event listeners
F3.TYPO3.Core.Application.fireEvent('logout');
```

Some events are only relevant to the internals of the module, and should not be exposed to other modules. These events should, by convention, start with an underscore character, and in their documentation block, have the `@private` annotation.

TODO: should we add some section about sub-namespaces inside event names?

4 Forms

[illegible]

5 Exception Handling

[illegible]

6 Custom widgets explained

-> extra chapter

6.1 BreadcrumbMenu

[illegible]

6.2 [illegible]

[illegible]