ETemplate2 JavaScript Documentation

Abstract

In contrast to our last approach, the new *ETemplate2*-System renders the ETemplate templates completely on the client. When a template should be displayed, the URL to the template definition XET file, plus the initial content array is sent to the client. The client downloads the XET file (most of the time it will come from the browser cache), parses and renders it and sets the values of the widget according to the ones defined in the content array.

JS Inheritance System

Instead of using the jQuery-Class system as we did before, a new inheritance system has been introduced (see *et2_inheritance.js*). It features:

- Basic class inheritance
- Support for interfaces
- Support for attributes

Basic inheritance

To create a class write

```
var MyClass = Class.extend([interfaces, ] functions/attributes);
```

where "interfaces" is a single interface or an array of interfaces (see below) and functions/attributes an object containing the functions the class implements. You can extend *MyClass* again by writing

```
var MyOtherClass = MyClass.extend([interfaces, ]
                       functions/attributes);
```

The *extend* function only extends the prototype and is only run once, when the JS file is intially parsed. Creating classes using the inheritance system introduces nearly no additional speed impact.

Defining functions

Simple functions: To define a function simply add it to the functions object:

```
var MyClass = Class.extend({
    myFunction: function(_param) {
     console.log("myFunction has been " +
         "called with param", param);
    }
});
```

Override functions: To override functions in another class write:

```
var MyOtherClass = MyClass.extend({
    myFunction: function(_param1, _param2) {
        // If this._super is not found inside
        // the function, the code overhead for
        // overriding functions is not created
        this._super(_param1);

        console.log("And _param2 is ", _param2);
    }
});
```

Constructor and destructor: The constructor function is named "*init*". It underlies the same overriding-mechanism as all other functions. When overriding the init function it is the best (as it probably is with all other functions too) to call the inherited super function with

```
this._super.apply(this, arguments);
```

The destructor is not really part of the inheritance system but used inside the widget system. The destructor is named "*destroy*" and should ALWAYS be written in order to free all references to other objects, delete all DOM-Nodes and unbind all event handers. When deriving from some higher-level widget classes it will be enough to override the "detatchFromDOM" method and sometimes even to do nothing (see below).

Interfaces

Using interfaces: Interfaces can be used to check whether a class implements a certain set of functions. You can create an interface declaration by writing something like:

```
var IBreathingObject = new Interface({
    breath: function() {}
});
```

The construct with the empty function is only to make the interface declaration look syntactically pleasing, other objects are not allowed inside the interface declaration.

To make a class extending a interface simply write

```
var MyClass = new Class.extend([IBreathingObject, ...], {
    [...]
});
```

or in the case of only one interface

```
var MyClass = Class.extend(IBreathingObject, {
    [...]
});
```

If a class does not implement all functions declared in the interface, it is marked as *abstract* and an attempt on creating it will throw an exception.

Extended type check: You can check whether a class/object derrives from a certain interface by using the *implements* function (*implements* does NOT check whether the functions are really implemented, just whether the class/object has been defined with that interface.

Example:

```
var MyClass = Class.extend(IBreathingObject, {
[...]
});

MyClass.prototype.implements(IBreathingObject); // true

var obj = new MyClass();
obj.implements(IBreathingObject); // true
```

Implements only checks for interfaces - if you want to check whether an object is instance of another class or implements a certain interface, you can use the *instanceOf* function. In the above example:

```
obj.instanceOf(Class); // true
obj.instanceOf(MyClass); // true;
obj.instanceOf(IBreathingObject); // true
```

Attributes

Basic usage: Attributes are used to declare a certain set of variables, getters and setters which are automatically used when (de-)serializing the object. Additionally the attributes can be used to add some documentation and type

	default value	**special conversions**
string	`""`	
integer	`0`	strings are parsed to ints
floats	`0.0`	strings are parsed to floats
boolean	`false`	the strings `"false"`, `"true"`, and `""`
any	`null`	

Table 1: Overview over all available attribute types, their standard default values and special conversions

safety to the classes. To define attributes, simply add an “attribute”-object to your class definition:

```
var MyClass = Class.extend({
    attributes: {
        "color": {
            "name": "Color",
            "type": "string",
            "default": "red",
            "description": "This is just an example"
        }
    }
});
```

Default attribute values: When the object is created, after the ”init” functions have been called, the default value of the attribute will be set, without overriding existing object-variables which have the same name as the attribute. If no default is given, the default value defaults to an default value defined for the given type. The type defaults to “any” which means, that no type-check is done (see Table). If you do not wish the default value to be automatically set, set the default value to the “et2_no_default” object.

Setters and getters: You can manually *set* an attribute by calling the function

```
obj.setAttribute(name, value);
```

`setAttribute` checks for the existance of the attribute with the given name and typechecks the given value. When setting an attribute, the code checks, whether a setter function named “`set_[name]`” is found. If yes, the function is called with the given value as parameter. If the function does not exist, the following code is executed

```
obj[name] = value;
```

You can manually *get* an attribute by calling the function

```
obj.getAttribute(name);
```

Just like the `setAttribute` function, `getAttribute` searches for a getter function named “`get_[name]`” first and simply returns `obj[name]` if the getter function is not found.

Ignoring attributes: Sometimes you may want the `setAttribute` function to simply ignore a certain attribute - in ETemplate2 for example, the *span* attribute is ignored, as it is read by the grid class. To mark an attribute as “ignored” simply add:

```
"ignore": true
```

to the attribute definition. Setting an attribute to be ignored can also be done at runtime - but this only effects objects of exactly the same class, as attribute definitions are copied between prototypes and not referenced.

Attribute inheritance: All classes automatically inherit the attributes of their parent class. You can change the attribute definition for a new class by simply defining a new entry for it. The entry can also be partial. So in our above example, we could simply change the color-attribute to be an integer value, as e.g. the new class directly uses the binary-representation of the color:

```
var MyOtherClass = MyClass.extend({
    attributes: {
        "color": {
            "type": "integer"
        }
    }
});
```

JS Dependencies: The JS dependency management is done on the server-side by EGW, so it cannot be used or tested when using the standalone test method (see below). However, you can mark dependencies by using the

```
/*egw:uses
    jquery.jquery;
    et2_baseWidget;
*/
```

syntax. Note that this comment has to start in the first 16 lines. More information on that topic can be found in the preamble of

`class.egw_include_mgr.inc.php`

in phpgwapi.

ETemplate2 class structure

Overview

The following UML class diagram (see figure 1) gives an overview over the *ETemplate2* class structure. Basically the widgets form a tree and when loading the XET-file, the widget objects are automatically created according to the XML-Tag. Widget classes can be registered using the

```
et2_register_widget(class_prototype, array_of_tagnames);
```

function.

et2_widget

Overview: *et2_widget* is the base widget class. It spans the widget tree and introduces features for managing the child widgets, like adding and destryoing them. The constructor takes to parameters, the parent and the actual tag name the widget was created for. The tag name is stored in the "type" object-variable. The widget automatically adds itself to the parent children list and removes itself from the list when it is destroyed.

XML-Handling: *et2_widget* has three important functions for XML handling:

```
et2_widget::loadFromXML(_node)
```

Reads the attributes for this widget from the given XML-DOM-Node. All child widgets are created and the LoadFromXML-function is recursively called for them.

```
et2_widget::loadAttributes(_attrs)
```

Is called by `LoadFromXML` with the XML-DOM-Attributes object. The base implementation simply calls the "`setAttribute`" function for every attribute found.

```
et2_widget::loadContent(_text)
```

Is called whenever `loadFromXML` finds a text-node inside the XML data. The base implementation does nothing when reaching a text node.

Cloning widgets: *et2_widget* introduces a mechanism which can be used to clone widgets and to assign the attributes of other widgets to them. The clone function

```
et2_widget::clone(_parent);
```

takes the parent widget the cloned widget should be added to.

Updating the widget: *et2_widget* has a function called `update` which reads all attributes and setts them again.

```
et2_widget::update()
```

This function is e.g. important if you replace the DOM-Node of a widget (see below) and want to call all attribute setters. The `upate` function does that automatically.

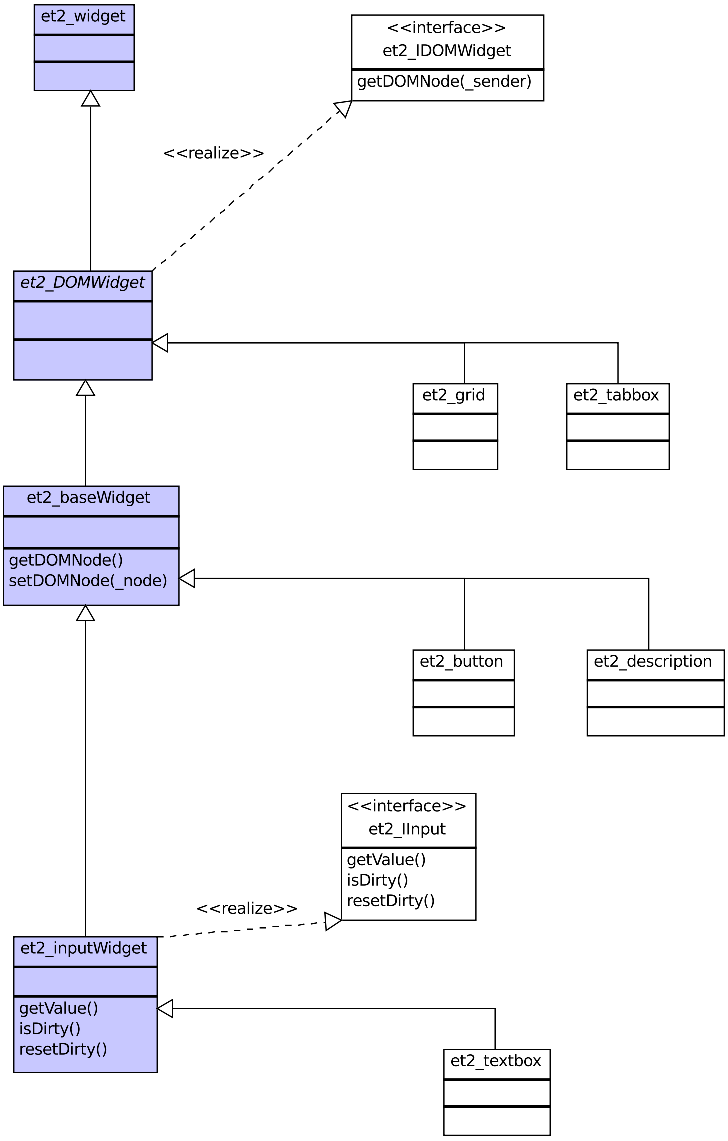

Figure 1: Overview over the ETemplate2 JavaScript side class structure. The blue classes are the ones you normally derrive other classes from.

et2_IDOMNode interface

The *et2_IDOMNode* interface declares the

`et2_IDOMNode::getDOMNode(_sender)`

function, which has to be implemented by all widgets which have a representation in the HTML-DOM-Tree. `getDOMNode` should return the DOM-Node of the current widget. The return value has to be a plain DOM node, not a jQuery object. The `_sender` parameter defines which widget is asking for the DOMNode. Depending on that, the widget may return different nodes. This is used in the grid or the tab. Normally the _sender parameter can be omitted in most implementations of the getDOMNode function. However, you should always provide the `_sender` parameter when calling `getDOMNode`!

et2_DOMWidget

The *et2_DOMWidget* class is derrived from *et2_IDOMNode* and *et2_widget* without implementing the `getDOMNode` funcition. It introduces an mechanism which automatically inserts the DOM-Node of this widget into the DOM-Node of the parent widget (if the parent widget implements *et2_IDOMWidget*. The two functions

```
et2_DOMWidget::attachToDOM()
et2_DOMWidget::detatchFromDOM()
```

have to be extended by any class which derrives from *et2_DOMwidget* and attaches event-handlers to the DOM. All event handlers have to be detached in the `detatchFromDOM` function and (re-)attached in the `attachToDOM` function. See the *et2_baseWidget* statustext-code for an example. The *et2_DOMWidget* also automatically sets the *id* property of its DOM-Node.

Directly derrive a widget from the *et2_DOMWidget* class, if you have to return different DOM-Nodes depending on the widget which asked for the DOM-Node and therefore need an own implementation of the `getDOMNode` function.

et2_baseWidget

et2_baseWidget, derrived from *et2_DOMWidget* is in most cases the best choice whe implementing an widget. It introduces the function

`et2_DOMWidget::setDOMNode(_node)`

and implements `getDOMNode`. Calling `setDOMNode` detatches the current node from the DOM-Tree and attaches the given node. Very basic widgets only have to call `setDOMNode` once in the constructor. Widgets which can change their underlying DOM-Node can call `setDOMNode` e.g. from setter functions. Remember that you have to reapply all changes you've made to the DOM-Node, so probably you'll have to call `update()` after having set the new DOM-Node.

et2_IInput Interface

et2_IInput is the base interface for all widgets which can return a value. The three functions, which have to be implemented are rather self-explanatory:

```
et2_IInput::getValue()
et2_IInput::isDirty()
et2_IInput::resetDirty()
```

et2_inputWidget

The *et2_inputWidget* derrives from *et2_baseWidget* and *et2_IInput*. This class automatically reads its content from the content-array when its id is set (behaviour might be changed). It has a function

```
et2_inputWidget::getInputWidget()
```

which as a default returns the DOM-Node of the widget. In more complicated widgets where the base DOM-Node is not equally to the input-DOM-Node you can override this function.

The setters and getters for the value attribute of the widget utilize the jQuery “`val()`” function for the DOM-Node returned by

```
et2_inputWidget::getInputWidget()
```

In some complicated cases it might be better not to derrive directly from `et2_inputWidget` but from `et2_baseWidget` or even `et_DOMWidget` and simply implement `et2_IInput`.

Conventions and testing

New widgets should be in their own JS-File named like the widget. They can by easily tested by including the JS-File in `test_xml.html` found in the “test” folder. You can simply add new tests using the same scheme as with the existing tests.

Please always test, whether your widget also works properly if it is wrapped in a template, as this causes the widget to be cloned, which creates a completely different situation in comparison to reading the widget directly from XML. A test XET-File for this may look like that:

```
<overlay>
    <!-- Test without template -->
    <mywidget/>

    <!-- Test with template -->
    <template id="test">
        <mywidget/>
    </template>
    <!-- Reference the template above -->
```

```
    <template id="test"/>
</overlay>
```

There are also many more or less helpfull debug messages in the code. You can regulate the amout of messages by setting the `ET2_DEBUGLEVEL` variable in *et2_common.js*.