widget.js Documentation

Release 0.6.0

Nicolas Vanhoren

August 19, 2015

Contents

This is the documentation of widget.js, the lightweight javascript widget framework.

Contents:

CHAPTER 1

Tutorial

1.1 Presentation of widget.js

widget.js is a lightweight framework in JavaScript to separate web applications in multiples manageable components called widgets.

widget.js is not a full featured framework that can handle all aspects of a web application like network communications or routing, there already exists good libraries for that. widget.js only handles one aspect of web development: separation of visual components into independant entities. So it provides only features to serve that goal, namely widgets, lifecycle management and events.

1.2 Quickstart

The easiest way to start a widget.js application is to checkout the sample application. Using git, do this:

```
git clone https://github.com/nicolas-van/widget.js-starter.git -b 0.6.0
```

This sample application uses bower, npm and grunt to download the dependencies and launch a small web server. Type these lines to download everything and start the server:

```
bower install
npm install
grunt
```

Then head your web browser to `http://localhost:9000` and you will see the *Hello World* message outputed by the application.

1.2.1 A Word About Template Engines

widget.js is a lightweight framework. As such, it doesn't impose a particular template engine to render HTML. The sample application uses Nunjucks from Mozilla as it is a high quality and full featured template engine in JavaScript, but you are free to replace it by any other template engine.

1.3 A First Widget

Take a look at the `src/js/app.js` file to have an example of your first widget:

```
myapp.Widget1 = widget.Widget.$extend({
    render: function() {
        return nunjucks.render('widget1.html');
    },
});
```

We can see that `Widget1` is a simple subclass of `widget.Widget`. It overrides the method `render()` to return some HTML code rendered using Nunjucks (the `widget1.html` file is located in the `views` folder). The `render()` method is a convenience used to indicate to the `widget.Widget` class the base HTML that should be appended to our widget.

In the `index.html` file we can see how this widget is instantiated and appended into the DOM:

```
var widget1 = new myapp.Widget1();
widget1.appendTo($("body"));
```

The widget is instantiated and then we call the `appendTo()` method by passing a jQuery object pointing to the `<body>` element. `appendTo()` is one of the multiple methods allowing to manipulate the location of widgets.

1.4 The Widget's Root Element

When a widget is created, its root element is created with it. By default it's always a `<div>` but it's possible to change that behavior.

You can access the root element of a widget by using the `$()` method:

```
var MyWidget = widget.Widget.$extend({
    render: function() {
        return "<p>Hi, I'm a widget!</p>";
    },
});
console.log(new MyWidget().$());
// Prints a jQuery object pointing to this element:
// <div>
//   <p>Hi, I'm a widget!</p>
// </div>
```

As we can see, the `render()` is simply called during the widget's creation to fill the root element. The generation of the root element can be customized using the `tagName`, `attributes` and `className` attributes:

```
var MyWidget = widget.Widget.$extend({
    tagName: "span",
    className: "mywidget",
    attributes: {
        "style": "display: block",
    },
    render: function() {
        return "<p>Hi, I'm a widget!</p>";
    },
});
console.log(new MyWidget().$());
// Prints a jQuery object pointing to this element:
// <span class="mywidget" style="display: block">
//   <p>Hi, I'm a widget!</p>
// </span>
```

The `$()` method also allows to search through the widget's root element. Pass it a jQuery selector and it will return the matching elements, but only those contained under the root element.

Of course you can modify any part of the widget at any time.

```
var MyWidget = widget.Widget.$extend({
    render: function() {
        return "<p>Hi, I'm a widget!</p>";
    },
    changeText: function(newText) {
        this.$("p").text(newText);
    },
});
var x = new MyWidget();
x.changeText("I'm still a widget!");
console.log(x.$());
// Prints a jQuery object pointing to this element:
// <div>
//   <p>I'm still a widget!</p>
// </div>
```

1.5 Appending Widgets Into The DOM

By instanciating a widget you initialize it with its root element. But it's still detached from the DOM. To insert it into the DOM you can use one of the methods like `appendTo()`:

```
var MyWidget = widget.Widget.$extend({
    render: function() {
        return "<p>Hi, I'm a widget!</p>";
    },
});
new MyWidget().appendTo($("body"));
```

The `appendTo()` method has a similar behavior to the jQuery's appendTo() method. Multiple other methods exist to serve the same purpose, with some difference regarding the place where the root element will be inserted:

- `appendTo()`
- `prependTo()`
- `insertAfter()`
- `insertBefore()`
- `replace()`
- `detach()` (this one removes the widget from the DOM)

Warning: It is not recommended to use the `appendTo()` and similar methods directly on the jQuery object returned by `$()`. Doing so will disable some of widget.js's features that will be explained later.

1.6 Widget Events

Events is one of the main features of widget.js, and an incredibly useful tool in all modern UI libraries. Widget events are separate from DOM events like `click` or `submit`. They are used to define your own custom events. Example:

```
var MyWidget = widget.Widget.$extend({
    doSomething: function() {
        // some code...
        this.trigger("someEvent", "hello");
    },
});

var x = new MyWidget();
x.on("someEvent", function(txt) {
    console.log("an event occured " + txt);
});

x.doSomething();
// prints "an event occured hello"
```

`on()` is used to register event handlers, `trigger()` is used to trigger one and `off()` can be used to unregister if you need to.

See also:

If you want to use events outside of widgets you can use the `widget.EventDispatcher` class.

1.7 Properties

Properties are similar to class attributes, but they will trigger events when their value change. There are two possible ways to use properties: simple properties and accessors.

1.7.1 Simple Properties

Simply use the `set()` and `get()` methods on `widget.Widget` instances. They will set and get the asked property and trigger `change:xxx` events where `xxx` is the property name. Example:

```
var MyWidget = widget.Widget.$extend({
    constructor: function(parent) {
        this.$super(parent);
        this.on("change:color", function() {
            this.$().css("background-color", this.get("color"));
        }.bind(this));
    },
});

var x = new MyWidget();
x.set("color", "#000000");
console.log(x.$().css("background-color"));
// prints "#000000"
```

Note: This is the first time we override the constructor of the widget. The `$super()` here is simply used to call the super method in the parent class. The `parent` argument that is passed to the constructor will be explained in the life cycle chapter.

In this example we use a property named `color`. When this property is modified the widget will change the background color of its root element. This way the `color` widget property and the `background-color` CSS property are syncronized.

1.7.2 Accessors

You can also define getter and setter for properties. The downside is that you must always think about triggering the `change:xxx` event by yourself.

```
var MyWidget = widget.Widget.$extend({
    getColor: function() {
        return this.$().css("background-color");
    },
    setColor: function(color) {
        var previous = this.getColor();
        this.$().css("background-color", color);
        if (previous != this.getColor())
            this.trigger("change:color");
    },
});

var x = new MyWidget();
x.on("change:color", function() {
    console.log("current color is " + x.getColor());
});
x.setColor("#FFFFFF");
// prints "current color is #FFFFFF"
x.set("color", "#000000");
// prints "current color is #000000"
```

Note: When using accessors it is still possible to use `get()` and `set()`. `get("color")` will call `getColor()` and `set("color")` will call `setColor()`.

See also:

If you want to use properties outside of widgets you can use the `widget.Properties` class.

1.8 Widget Life Cycle

1.8.1 Widget Destruction

We saw how to create widgets, now it is time to destroy them. To do so just call the `destroy()` method:

```
var x = new widget.Widget();
x.appendTo($("body"));
x.destroy();
// the root element of x has been removed from the DOM
```

Once `destroy()` has been called on a widget it is considered as a dead object. Its root element is destroyed and all its event handlers are removed.

See also:

Removing the event handlers when an widget is destroyed simplifies the task of the garbage collector as events tend to generate a lot of circular references that make objects removal difficult.

It is also common to override the `destroy()` method to add some cleanup code. Remember: widgets are independant visual components. Aside from displaying HTML code they could encapsulate any kind of behavior like animations, network communication, etc... They are always susceptible to reserve ressources that should be freed or run background processes that should be stopped.

1.8.2 Parent-Children Relationship

In the constructor of `widget.Widget` there is an argument we didn't used until now: `parent`.

```
var MyWidget1 = widget.Widget.$extend({
    constructor: function(parent) {
        this.$super(parent);
        this.otherWidget = new MyWidget2(this).appendTo(this.$());
    }
});
var MyWidget2 = widget.Widget.$extend({
    // another widget
});
var x = new MyWidget1().appendTo($("body"));
x.destroy();
console.log(x.getDestroyed());
// prints true
console.log(x.otherWidget.getDestroyed());
// prints true
```

All widgets should have as first argument of their constructor `parent` and forward it to the super method. It serves to identify parent-children relationship. In this example, the `MyWidget1` instance is the parent and the `MyWidget2` is the child. Parent-children relationship is deeply related to life cycle management: whenever a widget is destroyed, all its children are also destroyed. So in the example we call `destroy()` on the `MyWidget1` instance and we can see that the `MyWidget2` instance was also destroyed.

Life cycle management using parent-children relationship is useful in big applications where a lot of widgets contain other widgets. If relationship are correctly defined, whenever you destroy a widget all the widgets it created will be destroyed. By extension all ressources that were directly or indirectly reserved by that widget will also be freed.

See also:

If you want to use life cycle management outside of widgets you can use the `widget.LifeCycle` class.

1.9 Putting It All Together

widget.js is just a toolbox that gives some indications on how to define good components. It is still necessary to use common sense and good practices to create scalable and maintainable applications.

Widgets should be considered as black boxes from the outside. A widget's HTML should only be modified by that same widget and be invisible from other components of the application.

As example, only a widget should register DOM events on one of its own elements. If you have a widget containing a `<form>` element, never register the `submit` event from outside the widget by doing something like `theWidget.$("form").on("submit", ...)`. Here is a more correct way to do it:

```
var MyWidget = widget.Widget.$extend({
    render: function() {
        return nunjucks.render('myform.html');
    },
    constructor: function(parent) {
        this.$super(parent);
        this.$("form").on("submit", this.formSubmit.bind(this));
    },
    formSubmit: function() {
        this.trigger("formCompleted");
    },
});
```

Here we forward the `submit` DOM event to a method that will trigger a `formCompleted` widget event. The difference is that the `submit` DOM event is only a technical detail about how a HTML `<form>` works. The `formCompleted` widget event is much more meaningful as a high level event: it identifies when the user has finished completing the form. If later we want to add validation to our widget, add complex asynchronous operations or transform the widget into something completely different like a wizard we can do so without modifying the external API of our widget. So any piece of code in our application that already used the `MyWidget` class will not see the difference. To sum it: `MyWidget` is a component that correctly encapsulates its behavior.

1.10 Tools and Shortcuts

The previous parts of this tutorial presented the main features of widget.js, but there are still many shortcuts that can be used to reduce the amount of code:

1.10.1 Setting Multiple Properties

The *set()* method can also receive a dictionary to set multiple properties with one call:

```
mywidget.set({
    "property1": "value1",
    "property2": "value2",
    "property3": "value3",
});
```

Of course all the `change:xxx` events will correctly be triggered.

1.10.2 Widget Events Static Definition

When a widget wants to register events on itself it can be boring to always call the `on()` method. To simplify it you can add event handlers in the `events` attribute:

```
var MyWidget = widget.Widget.$extend({
    events: {
        "change:color": function() {
            console.log("color changed");
        },
        "change:size": "sizeChanged",
    },
    sizeChanged: function() {
        console.log("size changed");
    },
});
```

Keys of the `events` attribute can be functions or strings that reference a specific method of the widget.

1.10.3 DOM Events Static Definition

Just like widget events static definition, there is an alternative to calling `$().on()`:

```
var MyWidget = widget.Widget.$extend({
    domEvents: {
        "mouseenter": function() {
            console.log("mouse entered widget");
        },
```

```
        "click button": "buttonClicked",
    },
    render: function() {
        return "<button>Click Me</button>";
    },
    buttonClicked: function() {
        console.log("button was clicked");
    },
});
```

The syntax is similar to widget events static definition except you can specify sub elements on which the event should be registered by using a key in the format `eventName cssSelector` like in the example `click button` (that binds the click event only on the button contained in the widget).

Warning: It is perfectly normal to define both `events` and `domEvents` but don't confuse them! They are different features.

1.10.4 Standard Widget Events

Some events are automatically triggered by widgets:

- `destroying` will be triggered when the widget is destroyed.
- `appendedToDom` will be triggered when the widget is appended in the DOM and it not anymore in a detached state. This is useful as example if you need to position elements using absolute positionning or start an animation.
- `removedFromDom` will be triggered if the widget is removed from the DOM, usually because the `detach()` method has been called.