widget.js Documentation

Release 1.2.0

Nicolas Vanhoren

February 24, 2017

Contents

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

Contents:

CHAPTER 1

Tutorial

Presentation of widget.js

widget.js is a lightweight framework in JavaScript to separate web applications in multiple reusable 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.

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 1.2.0
```

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.

A Word About Class Inheritance

widget.js allows to use ES6 class inheritance on its classes like this:

```
class MyWidget extends widget.Widget {
    constructor() {
        super();
    }
}
```

We recommend to use this ES6 feature and to use Babel when you need to support old browsers (which is what the sample application does). But if you don't want or can use Babel we also provide the `extend()` helper on all classes to simplify inheritance:

```
var MyWidget = widget.Widget.extend({
    constructor: function() {
        widget.Widget.call(this);
    }
});
```

The provided examples below all use ES6 syntax, just know that everything is usable with `extend()` too.

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.

A First Widget

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

```
myapp.Widget1 = class Widget1 extends widget.Widget {
    constructor() {
        super();
        this.el.innerHTML = nunjucks.render('widget1.html');
    }
}
```

We can see that `Widget1` is a simple subclass of `widget.Widget`. It overrides the constructor to add some HTML to the root element of the widget, `this.el`. In this example that HTML is rendered using Nunjucks (the `widget1.html` file is located in the `views` folder).

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(document.body);
```

The widget is instantiated and then we call the `appendTo()` method by passing an element. `appendTo()` is one of the multiple methods allowing to manipulate the location of widgets.

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 `el` accessor:

```
class MyWidget extends widget.Widget {
    constructor() {
        super();
        this.el.innerHTML = "<p>Hi, I'm a widget!</p>";
    }
}
console.log(new MyWidget().el);
// Prints this element:
// <div>
```

```
//   <p>Hi, I'm a widget!</p>
// </div>
```

The generation of the root element can be customized using the `tagName`, `attributes` and `className` attributes:

```
class MyWidget extends widget.Widget {
    get tagName() { return "span"; }
    get className() { return "mywidget"; }
    get attributes() { return {"style": "display: block"}; }
    constructor() {
        super();
        this.el.innerHTML = "<p>Hi, I'm a widget!</p>";
    }
}
console.log(new MyWidget().el);
// Prints this element:
// <span class="mywidget" style="display: block">
//   <p>Hi, I'm a widget!</p>
// </span>
```

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()`:

```
class MyWidget extends widget.Widget {
    constructor() {
        super();
        this.el.innerHTML = "<p>Hi, I'm a widget!</p>";
    }
}
new MyWidget().appendTo(document.body);
```

The `appendTo()` method appends the root element at the end of the provided element. Multiple other methods exist to serve the same purpose, with some differences 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 directly alter the placement of the root element by using `el`. Doing so will disable some of widget.js's features that will be explained later.

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:

```
class MyWidget extends Widget {
    doSomething() {
        // some code...
        this.trigger("someEvent");
    }
}

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

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

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

`on()` can also be used to register multiple events at once:

```
x.on({
    "someEvent": function() { ... },
    "someOtherEvent": function() { ... },
});
```

See also:

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

DOM Events

While it is perfectly feasible to call `addEventListener()` on the root element accessed using `el`, widget.js provides an easier way to listen to those DOM events:

```
var x = new MyWidget();
x.on("dom:click", function() {
    console.log("the element was clicked");
});
```

When adding `dom:` at the beggining of the event type when calling `on()` you can proxy the DOM events through the widget. It is also possible to listen to events on sub elements of the root element:

```
x.on("dom:click button", function(e) { // note that you can replace "button" by any CSS selector
    console.log("the button was clicked");
});
```

Doing so uses event bubbling. In this example the hypothetic button could be created after the call to `on()` without problems. It can also be great for performances in multiple cases. Also note that, in the above example, you can get the button element using `e.bindedTarget`.

Widget Life Cycle

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(document.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 a dead object. Its root element is detached 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.

Parent-Children Relationship

Widgets maintain a parent-children between themselves. You can see that relationship by using the `parent` and `children` attributes.

```
class MyWidget1 extends widget.Widget {
    constructor() {
        super();
        this.otherWidget = new MyWidget2().appendTo(this.el);
    }
}
class MyWidget2 extends widget.Widget {
    // another widget
}
var x = new MyWidget1().appendTo(document.body);
console.log(x.otherWidget.parent === x);
// prints true
console.log(x.children[0] === x.otherWidget);
// prints true
```

Widgets maintain their parent-children automatically. You can also specify it explicitly by setting the `parent` attribute.

When a widget is destroyed it will destroy its children recursively:

```
x.destroy();
console.log(x.destroyed);
// prints true
console.log(x.otherWidget.destroyed);
// prints true
```

Life cycle management using parent-children relationship is useful in big applications where a lot of widgets contain other widgets. When 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.

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.on("dom:submit form", ...)`. Here is a more correct way to do it:

```
class MyWidget extends widget.Widget {
    constructor() {
        super();
        this.on("dom:submit form", this.formSubmit);
        this.el.innerHTML = nunjucks.render('myform.html');
    }
    formSubmit() {
        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.

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:

Ready

The typical helper to know if the browser finished the loading of the page, if you don't plan to use jQuery:

```
widget.ready(function() {
    // put some code
});
```

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.