Using Node's Event Module

When I first heard about Node.js, I thought it was just a JavaScript implementation for the server. But it's actually much more: it comes with a host of built-in functions that you don't get in the browser. One of those bit of functionality is the Event Module, which has the `EventEmitter` class.

`EventEmitter`: What and Why

One last benefits to events: they are a very loose way of coupling parts of your code together.

So, what exactly does the `EventEmitter` class do? Put simply, it allows you to listen for "events" and assign actions to run when those events occur. If you're familiar with front-end JavaScript, you'll know about mouse and keyboard events that occur on certain user interactions. **These are very similar, except that we can emit events on our own, when we want to, and not necessary based on user interaction**. The principles `EventEmitter` is based on have been called the publish/subscribe model, because we can subscribe to events and then publish them. There are many front-end libraries built with pub/sub support, but Node has it build in.

The other important question is this: why would you use the event model? In Node, it's an alternative to deeply nested callbacks. A lot of Node methods are run asynchronously, which means that to run code after the method has finished, you need to pass a callback method to the function. Eventually, your code will look like a giant funnel. To prevent this, many node classes emit events that you can listen for. This allows you to organize your code the way you'd like to, and not use callbacks.

One last benefits to events: they are a very loose way of coupling parts of your code together. An event can be emitted, but if no code is listening for it, that's okay: it will just passed unnoticed. This means removing listeners (or event emissions) never results in JavaScript errors.

Using `EventEmitter`

We'll begin with the `EventEmitter` class on its own. It's pretty simple to get at: we just require the events module:

```
var events = require("events");
```

This `events` object has a single property, which is the `EventEmitter` class itself. So, let's make a simple example for starters:

```
var EventEmitter = require("events").EventEmitter;

var ee = new EventEmitter();
ee.on("someEvent", function () {
    console.log("event has occured");
});

ee.emit("someEvent");
```

We begin by creating a new `EventEmitter` object. This object has two main methods that we use for events: `on` and `emit`.

We begin with `on`. This method takes two parameters: we start with the name of the event we're listening for: in this case, that's `"someEvent"`. But of course, it could be anything, and you'll usually choose something better. The second parameter is the function that will be called when the event occurs. That's all that is required for setting up an event.

Now, to fire the event, you pass the event name to the `EventEmitter` instance's `emit` method. That's the last line of the code above. If you run that code, you'll see that we get the text printed out to the console.

That's the most basic use of an `EventEmitter`. You can also include data when firing events:

```
ee.emit("new-user", userObj);
```

That's only one data parameter, but you can include as many as you want. To use them in your event handler function, just take them as parameters:

```
ee.on("new-user", function (data) {
    // use data here
});
```

Before continuing, let me clarify part of the `EventEmitter` functionality. We can have more than one listener for each event; multiple event listeners can be assigned (all with `on`), and all functions will be called when the event is fired. By default, Node allows up to ten listeners on one event at once; if more are created, node will issue a warning. However, we can change this amount by using `setMaxListeners`. For example, if you run this, you should see a warning printed out above the output:

```
ee.on("someEvent", function () { console.log("event 1"); });
ee.on("someEvent", function () { console.log("event 2"); });
ee.on("someEvent", function () { console.log("event 3"); });
ee.on("someEvent", function () { console.log("event 4"); });
ee.on("someEvent", function () { console.log("event 5"); });
ee.on("someEvent", function () { console.log("event 6"); });
ee.on("someEvent", function () { console.log("event 7"); });
ee.on("someEvent", function () { console.log("event 8"); });
ee.on("someEvent", function () { console.log("event 9"); });
ee.on("someEvent", function () { console.log("event 10"); });
ee.on("someEvent", function () { console.log("event 11"); });

ee.emit("someEvent");
```

To set the maximum number of viewers, add this line above the listeners:

```
ee.setMaxListeners(20);
```

Now when you run it, you won't get a warning.

Other `EventEmitter` Methods

There are a few other `EventEmitter` methods you'll find useful.

Here's a neat one: `once`. It's just like the `on` method, except that it only works once. After being called for the first time, the listener is removed.

```
ee.once("firstConnection", function () { console.log("You'll never see this again"); })
ee.emit("firstConnection");
ee.emit("firstConnection");
```

If you run this, you'll only see the message once. The second emission of the event isn't picked up by any listeners (and that's okay, by the way), because the `once` listener was removed after being used once.

Speaking of removing listeners, we can do this ourselves, manually, in a few ways. First, we can remove a single listener with the `removeListener` method. It takes two parameters: the event name and the listener function. So far, we've been using anonymous functions as our listeners. If we want to be able to remove a listener later, it will need to be a function with a name we can reference. We can use this `removeListener` method to duplicate the effects of the `once` method:

```
function onlyOnce () {
    console.log("You'll never see this again");
    ee.removeListener("firstConnection", onlyOnce);
}
```

```
ee.on("firstConnection", onlyOnce)
ee.emit("firstConnection");
ee.emit("firstConnection");

```

If you run this, you'll see that it has the very same effect as `once`.

If you want to remove all the listeners bound to a given event, you can use `removeAllListeners`; just pass it the name of the event:

```
ee.removeAllListeners("firstConnection");
```

To remove all listeners for all events, call the function without any parameters.

```
ee.removeAllListeners();
```

There's one last method: `listener`. This method takes an event name as a parameter and returns an array of all the functions that are listening for that event. Here's an example of that, based on our `onlyOnce` example:

```
function onlyOnce () {
    console.log(ee.listeners("firstConnection"));
    ee.removeListener("firstConnection", onlyOnce);
    console.log(ee.listeners("firstConnection"));
}

ee.on("firstConnection", onlyOnce)
ee.emit("firstConnection");
ee.emit("firstConnection");
```

Our `EventEmitter` instance itself actually fires two events of its own, which we can listen for: one when we create new listeners, and one when we remove them. See here:

```
ee.on("newListener", function (evtName, fn) {
    console.log("New Listener: " + evtName);
});

ee.on("removeListener", function (evtName) {
    console.log("Removed Listener: " + evtName);
});

function foo () {}

ee.on("save-user", foo);
ee.removeListener("save-user", foo);
```

Running this, you'll see our listeners for both new listeners and removed listeners have been run, and we get the messages we expected.

So, now that we've seen all the methods that an `EventEmitter` instance has, let's see how it works in conjunction with other modules.

`EventEmitter` Inside Modules

Since the `EventEmitter` class is just regular JavaScript, it makes perfect sense that it can be used within other modules. Inside your own JavaScript modules, you can create `EventEmitter` instances, and use them to handle internal events. That's simple, though. More interestingly, would be to create a module that inherits from `EventEmitter`, so we can use its functionality part of the public API.

Actually, there are built-in Node modules that do exactly this. For example, you may be familiar with the `http` module; this is the module that you'll use to create a web server. This basic example shows how the `on` method of the `EventEmitter` class has become part of the `http.Server` class:

```
var http = require("http");
var server = http.createServer();

server.on("request", function (req, res) {
    res.end("this is the response");
});

server.listen(3000);
```

If you run this snippet, the process will wait for a request; you can go to `http://localhost:3000` and you'll get the response. When the server instance gets the request from your browser, it emits a `"request"` event, an event that our listener will receive and can act upon.

So, how can we go about creating a class that will inherit from `EventEmitter`? It's actually not that difficult. We'll create a simple `UserList` class, which handles user objects. So, in a `userlist.js` file, we'll start with this:

```
var util         = require("util");
var EventEmitter = require("events").EventEmitter;
```

We need the `util` module to help with the inheriting. Next, we need a database: instead of using an actual database, though, we'll just use an object:

```
var id = 1;
var database = {
    users: [
        { id: id++, name: "Joe Smith",  occupation: "developer"  },
        { id: id++, name: "Jane Doe",   occupation: "data analyst" },
        { id: id++, name: "John Henry", occupation: "designer"   }
    ]
};
```

Now, we can actually create our module. If you aren't familiar with Node modules, here's how they work: any JavaScript we write inside this file is only readable from inside the file, by default. If we want to make it part of the module's public API, we make it a property of `module.exports`, or assign a whole new object or function to `module.exports`. Let's do this:

```
function UserList () {
    EventEmitter.call(this);
}
```

This is the constructor function, but it isn't your usual JavaScript constructor function. What we're doing here is using the `call` method on the `EventEmitter` constructor to run that method on the new `UserList` object (which is `this`). If we need to do any other initialization to our object, we could do it inside this function, but that's all we'll do for now.

Inheriting the constructor isn't enough though; we also need to inherit the prototype. This is where the `util` module comes in.

```
util.inherits(UserList, EventEmitter);
```

This will add everything that's on `EventEmitter.prototype` to `UserList.prototype`; now, our `UserList` instances will have all the methods of an `EventEmitter` instance. But we want to add some more, of course. We'll add a `save` method, to allow us to add new users.

```
UserList.prototype.save = function (obj) {
    obj.id = id++;
    database.users.push(obj);
    this.emit("saved-user", obj);
};
```

This method takes an object to save to our `"database"`: it adds an `id` and pushes it into the users array. Then, it emits the `"saved-user"` event, and passes the object as data. If this were a real database, saving it would probably be an asynchronous task, meaning that to work with the saved record we would need to accept a callback. The alternative to this is to emit an event, as we're doing. Now, if we want to do something with the saved record, we can just listen for the event. We'll do this in a second. Let's just close up the `UserList`

```
UserList.prototype.all = function () {
    return database.users;
};

module.exports = UserList;
```

I've added one more method: a simple one that returns all the users. Then, we assign `UserList` to `module.exports`.

Now, let's see this in use; in another file, say `test.js`. Add the following:

```
var UserList = require("./userlist");
var users = new UserList();

users.on("saved-user", function (user) {
    console.log("saved: " + user.name + " (" + user.id + ")");
});

users.save({ name: "Jane Doe", occupation: "manager" });
users.save({ name: "John Jacob", occupation: "developer" });
```

After requiring our new module and creating an instance of it, we listen for the `"saved-user"` event. Then, we can go ahead and save a few users. When we run this, you'll see that we get two messages, printing out the names and ids of the records we saved.

```
saved: Jane Doe (4)
saved: John Jacob (5)
```