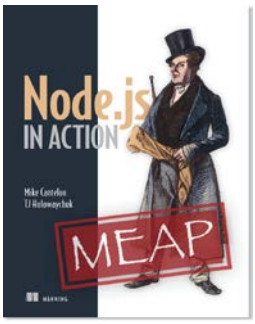
MANNING PUBLICATIONS



Node.js in Action
By Mike Cantelon and TJ Holowaychuk

If you've done front-end web programming in which interface events trigger logic, then you've done asynchronous programming. There are two popular models in the Node world for managing response logic: callbacks and event emitters. This article, based on chapter 3 of Node.js in Action, discusses handling one-off events with callbacks and responding to repeating events with event emitters.

You may also be interested in…

Asynchronous Programming Techniques

If you've done front-end web programming in which interface events, such as mouse clicks, trigger logic, then you've done asynchronous programming. Server-side asynchronous programming is no different: events can trigger response logic. There are two popular models in the Node world for managing response logic: callbacks and event emitters.

Callbacks generally define logic for *one-off* responses. If you perform a database query, for example, you can specify a callback to determine what to do with the query results. The callback may display the database results, do a calculation based on the results, or just store the results in memory for later reference.

Event emitters, on the other hand, provide a framework for organizing callbacks. Event emitters define behavior by specifying response logic that's executed when a given event type occurs: either once or every time the event occurs. The different event types an event emitter supports are often conceptually related, which makes event emitters useful for organizing and reusing code. Node's HTTP server, for example, is implemented as an event emitter because it has to use the same callback logic to repeatedly respond to requests.

So now that we've established that response logic is generally organized in one of two ways in Node, let's jump right in to it by learning first how to handle one-off events with callbacks and then how to respond to repeating events with event emitters.

Handling one-off events with callbacks

A callback is an anonymous function, passed as an argument to an asynchronous function that describes what to do after the asynchronous operation has completed. Following is an example of the use of a callback provided as an argument to the `asyncFunction` function. In the example, the text "I am the voice of the callback!" is logged to the console after 200 milliseconds.

```
function asyncFunction(callback) {
  setTimeout(function() {
    callback();
  }, 200);
}

asyncFunction(function() {
  console.log('I am the voice of the callback!');
});
```

In Node development, you may need to create asynchronous functions that require multiple callbacks as arguments: to handle success or failure, for example. The following example shows this idiom. In the example, the text "I handle success." is logged to the console.

Listing 1 An example of the use of multiple callbacks as arguments to a single asynchronous function

```
function doSomething() {
  return true;
}

function asyncFunction(err, success) {
  if (doSomething()) {
    success();
    } else {
      err();
    }
  }

asyncFunction(
  function() { console.log('I handle failure.'); },
  function() { console.log('I handle success.'); }
);
```

Using anonymous functions as callback arguments can be messy. The following example has three levels of nested callbacks. Three levels isn't bad but, once you reach, say, seven levels of callbacks, then things can look quite cluttered.

```
someAsyncFunction('data', function(text) {
  anotherAsyncFunction(text, function(text) {
    yetAnotherAsyncFunction(text, function(text) {
      console.log(text);
    });
  });
});
```

By creating functions that handle the individual levels of callback nesting you can express the same logic in a way that requires more lines of code but could be considered easier to read, depending on your tastes.

Listing 2 An example of reducing nesting by creating intermediary functions

```
function handleResult(text) {
  console.log(text);
}

function innerLogic(text) {
  yetAnotherAsyncFunction(text, handleResult);
}

function outerLogic(text) {
  anotherAsyncFunction(text, innerLogic);
}

someAsyncFunction('data', outerLogic);
```

Now that you've learned how to use callbacks to handle one-off events—used to define responses for database queries, web server requests, reading files and more—we're going to move on to how to handle repeated events using event emitters.

Handling repeating events with event emitters

Event emitters are entities suited to respond to repeating events with asynchronous logic. Support for them is built into Node and some important Node API components, such as HTTP servers, TCP/IP servers, and streams, are implemented as event emitters.

Event responses are defined through the use of *listeners*. A listener is the association of an event type with an asynchronous callback that gets triggered each time the event type occurs.

The following code defines an echo server that will, when a user connects to it using Telnet, simply echo back whatever gets sent to it. The server is an event emitter with a listener defined for *data* events:

```
var net = require('net');

var server = net.createServer(function(socket) {
  socket.on('data', function(data) {
    socket.write(data);
  });
});
```

```
server.listen(8888);
```

You can define listeners that repeatedly respond to events, as the previous example showed, or define listeners that respond only once. The following code modifies the previous echo server example to echo only the first chunk of data sent to it:

```
var net = require('net');

var server = net.createServer(function(socket) {
  socket.once('data', function(data) {
    socket.write(data);
  });
});

server.listen(8888);
```

In the previous example, we used a built-in Node API that leverages event emitters. Node's built-in *events* module, however, allows you to create your own event emitters. The following code defines a *channel* event emitter with a single listener that responds when someone joins the channel. Note that you use `on` (or, alternatively, the longer form `addListener`) to add a listener to an event emitter.

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

var channel = new events.EventEmitter();

channel.on('join', function() {
  console.log("Welcome!");
});
```

The above code, when run, won't do anything because there is nothing to trigger an event. You could add a line to the previous example to would trigger an event using the emit function:

```
channel.emit('join');
```

Going farther with this, you can create simple publish/subscribe logic that you can use as the foundation for a chat application. If you run the following script you'll have a simple chat server.

Listing 3 A simple publish/subscribe system using an event emitter.

```
var events = require('events')
  , net = require('net');

var channel = new events.EventEmitter();
channel.clients = {};
channel.subscriptions = {};

channel.on('join', function(id, client) {
  this.clients[id] = client;          #1
  this.subscriptions[id] = function(senderId, message) {
    if (id != senderId) {             #2
      this.clients[id].write(message);
    }
  }
  this.on('broadcast', this.subscriptions[id]);    #3
});

var server = net.createServer(function (client) {
  var id = client.remoteAddress + ':' + client.remotePort;
  client.on('connect', function() {
    channel.emit('join', id, client);              #4
  });
  client.on('data', function(data) {
    data = data.toString();
    channel.emit('broadcast', id, data);           #5
  });
});
server.listen(8888);
```

#1 Stores reference to this client
#2 Ignores data if it's from the receiving client
#3 Adds send function as broadcast event listener
#4 Emits channel join event when client connects to server
#5 Emits channel broadcast event when data received by server

When a user connects to the server (#4), a *join* event is emitted to the channel, with the user's remote address/port used as an identifier. The user's connection data is stored (#1) and a *broadcast* listener for that user is created (#3) ignoring any messages broadcast by the user (#2). When any user sends data to the server, a broadcast event is emitted to the channel specifying the user who created it and the message.

Once you've got the chat server running, open a new command-line and enter the following to enter the chat. If you open up a few command lines you'll see that anything typed in one command-line is echoed to the others.

```
telnet 127.0.0.1 8000
```

The problem with this chat server, however, is that anyone who closes their connection, leaving the chat room, leaves behind a listener that will attempt to write to a client that is no longer connected. This will, of course, generate an error. To fix this issue, we add the following listener to the channel event emitter and added logic to the server's *close* event listener to emit the channel's *leave* event.

The leave event, essentially, removes the broadcast listener originally added for the client.

```
...
channel.on('leave', function(id) {
  channel.removeListener('broadcast', this.subscriptions[id]);
  channel.emit('broadcast', id, id + " has left the chat.\n");
});

var server = net.createServer(function (client) {
  ...
  client.on('close', function() {
    channel.emit('leave', id);
  });
});
server.listen(8888);
```

If, for whatever reason, you wanted to prevent chat without shutting down the server, you could use the event `removeAllListeners` emitter method to remove all listeners of a given type. The following code shows how this could be implemented for our chat server example.

```
channel.on('shutdown', function() {
  channel.emit('broadcast', '', "Chat has shut down.\n");
  channel.removeAllListeners('broadcast');
});
```

Error handling

A convention in the creation of event emitters is to emit an 'error' type event, providing an error object as an argument, instead of directly throwing an error. This allows custom event response logic to be defined by setting one or more listeners for this event type. Following is an example of an error listener handling an emitted error by logging to the console.

```
var events = require('events');
var myEmitter = new events.EventEmitter();

myEmitter.on('error', function(err) {
  console.log('ERROR: ' + err.message);
});

myEmitter.emit('error', new Error('Something is wrong.'));
```

If no listener for this event type is defined, however, when the event type *error* is emitted the event emitter will output a stack trace (a list of program instructions that executed up the point where the error occurred) and halt execution. The stack trace will indicate an error of the type specified by the emit call's second argument. This behavior is unique to 'error' type events (when other event types are emitted but have no listeners, nothing happens).

If 'error' is emitted without an error object supplied as the second argument a stack trace will result indicating an *Uncaught, unspecified error event* error and your application will halt. You can define your own response to this error type, however, by defining a global handler using the following code.

```
process.on('uncaughtException', function(err){
  console.error(err.stack);
  process.exit(1);
});
```

If you wanted to provide users connecting to chat with a count of currently connected users you could use the method, listeners as shown by the following code, which returns an array of listens for a given event type.

```
channel.on('join', function(id, client) {
  var welcome = "Welcome!\n"
              + 'Guests online: ' + this.listeners('broadcast').length;
  client.write(welcome + "\n");
  ...
```

If you wanted to increase the number of listeners an event emitter has, to avoid warnings Node will display once there are more than 10 listeners, you could use the method. Using our channel event emitter `setMaxListeners` as an example, you'd use the following line to increase the number of allowed listeners:

```
channel.setMaxListeners(50);
```

Extending the event emitter

If you'd like to build upon the event emitter's behavior, you can create a new JavaScript class that inherits from the event emitter. Let's create, as an example of this, a class called `Watcher` meant to process files placed in a specified filesystem directory. You'll then use this class to create a utility that watches a filesystem directory, renaming any files placed in it to lower case, and copies these files into a separate directory.

The first thing you'd do is create a class constructor that takes as arguments the directory to monitor and the directory in which to put altered files, as shown by the following code.

```
function Watcher(watchDir, processedDir) {
  this.watchDir = watchDir;
  this.processedDir = processedDir;
}
```

Next, you'd add logic to inherit the event emitter's behavior.

```
var events = require('events');
Watcher.prototype = new events.EventEmitter();
```

Next, you'd extend the methods inherited from `EventEmitter` with two new methods.

Listing 4 Extending the event emitter's functionality

```
var fs = require('fs')
  , watchDir = './watch'
  , processedDir = './done';

Watcher.prototype.watch = function() {      #1
  var watcher = this;
  fs.readdir(this.watchDir, function(err, files) {
    if (err) throw err;
    for(index in files) {
      watcher.emit('process', files[index]);    #2
    }
  })
}

Watcher.prototype.start = function() {     #3
  var watcher = this;
  fs.watchFile(watchDir, function() {
    watcher.watch();
  });
}
```

#1 Extends EventEmitter with method that processes files
#2 Processes each file in the watch directory
#3 Extends EventEmitter with method to start watching

One method cycles through the directory (#1), processing any files found (#2). The other method starts the directory monitoring (#3). The monitoring leverages Node's function, so when something happens in the `fs.watchFile` watched directory, the `watch` method is triggered, cycling through the watched directory and emitting a *process* event for each file found.

Now that you've defined the `Watcher` class, you can put it to work by creating a `Watcher` object.

```
var watcher = new Watcher(watchDir, processedDir);
```

With your newly created `Watcher` object, you can use the `on` method, inherited from the event emitter class, to set the logic used to process each file.

```
watcher.on('process', function process(file) {
  var watchFile = this.watchDir + '/' + file;
  var processedFile = this.processedDir + '/' + file.toLowerCase();

  fs.rename(watchFile, processedFile, function(err) {
    if (err) throw err;
  });
});
```

Now that the all the necessary logic is in place, you can start the directory monitor using the following code.

```
watcher.start();
```

After putting the `Watcher` code into a script, and creating "watch" and "done" directories, you should be able to run the script using Node, drop files into the "watch" directory, and see the files pop up, renamed to lower case, in the "done" directory. This is an example of how the event emitter can be a useful class to create new classes from.

By learning how to use callbacks to define one-off asynchronous logic and how to use event emitters to dispatch asynchronous logic repeatedly, you're one step closer to mastering the control of Node application behavior. In a single callback or event emitter listener, however, you may want to include logic that performs additional asynchronous tasks. If the order in which these tasks are to be performed is important, you may be faced with a new challenge: how to control exactly when each task, in a series of asynchronous tasks, executes.

Summary

You've learned how to deal with the challenges of controlling asynchronous behavior through the use of callbacks, event emitters, and flow control.

Callbacks are appropriate for one-off asynchronous logic, but their use requires care to prevent messy code. Event emitters can be helpful for organizing asynchronous logic because they allow it to be associated with a conceptual entity and easily managed through the use of listeners.

Here are some other Manning titles you might be interested in:

Quick & Easy HTML5 and CSS3
Rob Crowther

Sass and Compass in Action
Wynn Netherland, Nathan Weizenbaum, and Chris Eppstein

Secrets of the JavaScript Ninja
John Resig and Bear Bibeault

Last updated: September 22, 2011