Alex Young
Marc Harter

Foreword by
Ben Noordhuis

Node.js in Practice

Includes 115 Techniques

MANNING

SAMPLE CHAPTER

Node.js in Practice

by Alex Young
and Marc Harter

Chapter 4

brief contents

4 Events: Mastering EventEmitter and beyond

This chapter covers

- Using Node's EventEmitter module
- Managing errors
- How third-party modules use EventEmitter
- How to use domains with events
- Alternatives to EventEmitter

Node's `events` module currently includes just a single class: `EventEmitter`. This class is used throughout both Node's built-in modules and third-party modules. It contributes to the overall architecture of many Node programs. Therefore it's important to understand `EventEmitter` and how to use it.

It's a simple class, and if you're familiar with DOM or jQuery events, then you shouldn't have much trouble understanding it. The major consideration when using Node is in error handling, and we'll look at this in technique 21.

`EventEmitter` can be used in various ways—it's generally used as a base class for solving a wide range of problems, from building network servers to architecting

application logic. In view of the fact that it's used as the basis for key classes in popular Node modules like Express, learning how it works can be useful for writing idiomatic code that plays well alongside existing modules.

In this chapter you'll learn how to use `EventEmitter` to make custom classes, and how it's used within Node and open source modules. You'll also learn how to solve problems found when using `EventEmitter`, and see some alternatives to it.

4.1 Basic usage

To use `EventEmitter`, the base class must be inherited from. This section includes techniques for inheriting from `EventEmitter` and mixing it into other classes that already inherit from another base class.

TECHNIQUE 19 Inheriting from EventEmitter

This technique demonstrates how to create custom classes based on `EventEmitter`. By understanding the principles in this technique, you'll learn how to use `EventEmitter`, and how to better use modules that are built with it.

Problem

You want to use an event-based approach to solve a problem. You have a class that you'd like to operate when asynchronous events occur.

Web, desktop, and mobile user interfaces have one thing in common: they're event-based. Events are a great paradigm for dealing with something inherently asynchronous: the input from human beings. To show how `EventEmitter` works, we'll use a music player as an example. It won't really play music, but the underlying concept is a great way to learn how to use events.

Solution

The canonical example of using events in Node is inheriting from `EventEmitter`. This can be done by using a simple prototype class—just remember to call `EventEmitter`'s constructor from within your new constructor.

The first listing shows how to inherit from `EventEmitter`.

Listing 4.1 Inheriting from `EventEmitter`

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

function MusicPlayer() {
  events.EventEmitter.call(this);
}

util.inherits(MusicPlayer, events.EventEmitter);
```

Using util.inherits is the idiomatic Node way to inherit from prototype classes.

Discussion

The combination of a simple constructor function and `util.inherits` is the easiest and most common way to create customized event-based classes. The next listing extends the previous listing to show how to `emit` and bind listeners using `on`.

Listing 4.2 Inheriting from `EventEmitter`

```
var util = require('util');
var events = require('events');
var AudioDevice = {
  play: function(track) {
    // Stub: Trigger playback through iTunes, mpg123, etc.
  },

  stop: function() {
  }
};

function MusicPlayer() {
  this.playing = false;
  events.EventEmitter.call(this);
}

util.inherits(MusicPlayer, events.EventEmitter);

var musicPlayer = new MusicPlayer();

musicPlayer.on('play', function(track) {
  this.playing = true;
  AudioDevice.play(track);
});

musicPlayer.on('stop', function() {
  this.playing = false;
  AudioDevice.stop();
});

musicPlayer.emit('play', 'The Roots - The Fire');

setTimeout(function() {
  musicPlayer.emit('stop');
}, 1000);
```

The class's state can be configured, and then EventEmitter's constructor can be called as required.

The inherits method copies the methods from one prototype into another—this is the general pattern for creating classes based on EventEmitter.

The emit method is used to trigger events.

This might not seem like much, but suppose we need to do something else when `play` is triggered—perhaps the user interface needs to be updated. This can be supported simply by adding another listener to the `play` event. The following listing shows how to add more listeners.

Listing 4.3 Adding multiple listeners

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

function MusicPlayer() {
  this.playing = false;
  events.EventEmitter.call(this);
}

util.inherits(MusicPlayer, events.EventEmitter);
```

```
var musicPlayer = new MusicPlayer();

musicPlayer.on('play', function(track) {
  this.playing = true;
});

musicPlayer.on('stop', function() {
  this.playing = false;
});

musicPlayer.on('play', function(track) {
  console.log('Track now playing:', track);
});

musicPlayer.emit('play', 'The Roots - The Fire');

setTimeout(function() {
  musicPlayer.emit('stop');
}, 1000);
```

New listeners can be added as needed.

Listeners can be removed as well. `emitter.removeListener` removes a listener for a specific event, whereas `emitter.removeAllListeners` removes all of them. You'll need to store the listener in a variable to be able to reference it when removing a specific listener, which is similar to removing timers with `clearTimeout`. The next listing shows this in action.

Listing 4.4 Removing listeners

```
function play(track) {
  this.playing = true;
}

musicPlayer.on('play', play);

musicPlayer.removeListener('play', play);
```

A reference to the listener is required to be able to remove it.

`util.inherits` works by wrapping around the ES5 method `Object.create`, which inherits the properties from one prototype into another. Node's implementation also sets the superconstructor in the `super_` property. This makes accessing the original constructor a lot easier—after using `util.inherits`, your prototype class will have access to `EventEmitter` through `YourClass.super_`.

You can also respond to an event once, rather than every time it fires. To do that, attach a listener with the `once` method. This is useful where an event can be emitted multiple times, but you only care about it happening a single time. For example, you could update listing 4.3 to track if the play event has ever been triggered:

```
musicPlayer.once('play', {
  this.audioFirstStarted = new Date();
});
```

When inheriting from `EventEmitter`, it's a good idea to use `events.EventEmitter.call(this)` in your constructor to run `EventEmitter`'s constructor. The reason for this is because it'll attach the instance to a `domain` if domains are being used. To learn more about domains, see technique 22.

The methods we've covered here—`on`, `emit`, and `removeListener`—are fundamental to Node development. Once you've mastered `EventEmitter`, you'll find it cropping up everywhere: in Node's built-in modules and beyond. Creating TCP/IP servers with `net.createServer` will return a server based on `EventEmitter`, and even the `process` global object is an instance of `EventEmitter`. In addition, popular modules like `Express` are based around `EventEmitter`—you can actually create an Express `app` object and call `app.emit` to send messages around an Express project.

TECHNIQUE 20 Mixing in EventEmitter

Sometimes inheritance isn't the right way to use `EventEmitter`. In these cases, mixing in `EventEmitter` may work.

Problem

This is an alternative option to technique 19. Rather than using `EventEmitter` as a base class, it's possible to copy its methods into another class. This is useful when you have an existing class and can't easily rework it to inherit directly from `EventEmitter`.

Solution

Using a for-in loop is sufficient for copying the properties from one `prototype` to another. In this way you can copy the necessary properties from `EventEmitter`.

Discussion

This example might seem a little contrived, but sometimes it really is useful to copy `EventEmitter`'s properties rather than inherit from it in the usual way. This approach is more akin to a mixin, or multiple inheritance; see this demonstrated in the following listing.

Listing 4.5 Mixing in `EventEmitter`

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

function MusicPlayer(track) {
  this.track = track;
  this.playing = false;

  for (var methodName in EventEmitter.prototype) {
    this[methodName] = EventEmitter.prototype[methodName];
  }
}

MusicPlayer.prototype = {
  toString: function() {
    if (this.playing) {
      return 'Now playing: ' + this.track;
    } else {
      return 'Stopped';
```

This is the for-in loop that copies the relevant properties.

```
    }
  }
};

var musicPlayer = new MusicPlayer('Girl Talk - Still Here');

musicPlayer.on('play', function() {
  this.playing = true;
  console.log(this.toString());
});

musicPlayer.emit('play');
```

One example of multiple inheritance in the wild is the Connect framework.[1] The core `Server` class inherits from multiple sources, and in this case the Connect authors have decided to make their own property copying method, shown in the next listing.

Listing 4.6 `utils.merge` from Connect

```
exports.merge = function(a, b){
  if (a && b) {
    for (var key in b) {
      a[key] = b[key];
    }
  }
  return a;
};
```

This technique may be useful when you already have a well-established class that could benefit from events, but can't easily be a direct descendant of `EventEmitter`.

Once you've inherited from `EventEmitter` you'll need to handle errors. The next section explores techniques for handling errors generated by `EventEmitter` classes.

4.2 Error handling

Although most events are treated equally, `error` events are a special case and are therefore treated differently. This section looks at two ways of handling errors: one attaches a listener to the `error` event, and the other uses domains to collect errors from groups of `EventEmitter` instances.

TECHNIQUE 21 Managing errors

Error handling with `EventEmitter` has its own special rules that must be adhered to. This technique explains how error handling works.

Problem

You're using an `EventEmitter` and want to gracefully handle when errors occur, but it keeps raising exceptions.

[1] See http://www.senchalabs.org/connect/.

Solution

To prevent `EventEmitter` from throwing exceptions whenever an `error` event is emitted, add a listener to the `error` event. This can be done with custom classes or any standard class that inherits from `EventEmitter`.

Discussion

To handle errors, bind a listener to the `error` event. The following listing demonstrates this by building on the music player example.

Listing 4.7 Event-based errors

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

function MusicPlayer() {
  events.EventEmitter.call(this);
}

util.inherits(MusicPlayer, events.EventEmitter);

var musicPlayer = new MusicPlayer();

musicPlayer.on('play', function(track) {
  this.emit('error', 'unable to play!');
});

musicPlayer.on('error', function(err) {
  console.error('Error:', err);
});

setTimeout(function() {
  musicPlayer.emit('play', 'Little Comets - Jennifer');
}, 1000);
```

Listening for an error event

This example is perhaps simple, but it's useful because it should help you realize how `EventEmitter` handles errors. It feels like a special case, and that's because it is. The following excerpt is from the Node documentation:

> *When an `EventEmitter` instance experiences an error, the typical action is to emit an `error` event. Error events are treated as a special case in Node. If there is no listener for it, then the default action is to print a stack trace and exit the program.*

You can try this out by removing the `'error'` handler from listing 4.7. A stack trace should be displayed in the console.

This makes sense semantically—otherwise the absence of an error handler would lead to potentially dangerous activity going unnoticed. The event name, or *type* as it's referred to internally, has to appear exactly as `error`—extra spaces, punctuation, or uppercase letters won't be considered an `error` event.

This convention means there's a great deal of consistency across event-based error-handling code. It might be a special case, but it's one worth paying attention to.

TECHNIQUE 22 Managing errors with domains

Dealing with errors from multiple instances of `EventEmitter` can feel like hard work ... unless domains are used!

Problem

You're dealing with multiple non-blocking APIs, but are struggling to effectively handle errors.

Solution

Node's `domain` module can be used to centralize error handling for a set of asynchronous operations, and this includes `EventEmitter` instances that emit unhandled `error` events.

Discussion

Node's `domain` API provides a way of wrapping existing non-blocking APIs and exceptions with error handlers. This helps centralize error handling, and is particularly useful in cases where multiple interdependent I/O operations are being used.

Listing 4.8 builds on the music player example by using two `EventEmitter` descendants to show how a single error handler can be used to handle errors for separate objects.

Listing 4.8 Managing errors with `domain`

```
var util = require('util');
var domain = require('domain');
var events = require('events');
var audioDomain = domain.create();

function AudioDevice() {
  events.EventEmitter.call(this);
  this.on('play', this.play.bind(this));
}

util.inherits(AudioDevice, events.EventEmitter);

AudioDevice.prototype.play = function() {
  this.emit('error', 'not implemented yet');
};

function MusicPlayer() {
  events.EventEmitter.call(this);

  this.audioDevice = new AudioDevice();
  this.on('play', this.play.bind(this));

  this.emit('error', 'No audio tracks are available');
}

util.inherits(MusicPlayer, events.EventEmitter);
```

The Domain module must be loaded, and then a suitable instance created with the create method.

This error and any other errors will be caught by the same error handler.

```
MusicPlayer.prototype.play = function() {
  this.audioDevice.emit('play');
  console.log('Now playing');
};

audioDomain.on('error', function(err) {
  console.log('audioDomain error:', err);
});

audioDomain.run(function() {
  var musicPlayer = new MusicPlayer();
  musicPlayer.play();
});
```

Any code that raises errors inside this callback will be covered by the domain.

Domains can be used with `EventEmitter` descendants, networking code, and also the asynchronous file system methods.

To visualize how domains work, imagine that the `domain.run` callback wraps around your code, even when the code inside the callback triggers events that occur outside of it. Any errors that are thrown will still be caught by the domain. Figure 4.1 illustrates this process.

Without a domain, any errors raised using `throw` could potentially place the interpreter in an unknown state. Domains avoid this and help you handle errors more gracefully.

Now that you know how to inherit from `EventEmitter` and handle errors, you should be starting to see all kinds of useful ways that it can be used. The next section broadens these techniques by introducing some advanced usage patterns and higher-level solutions to program structure issues relating to events.

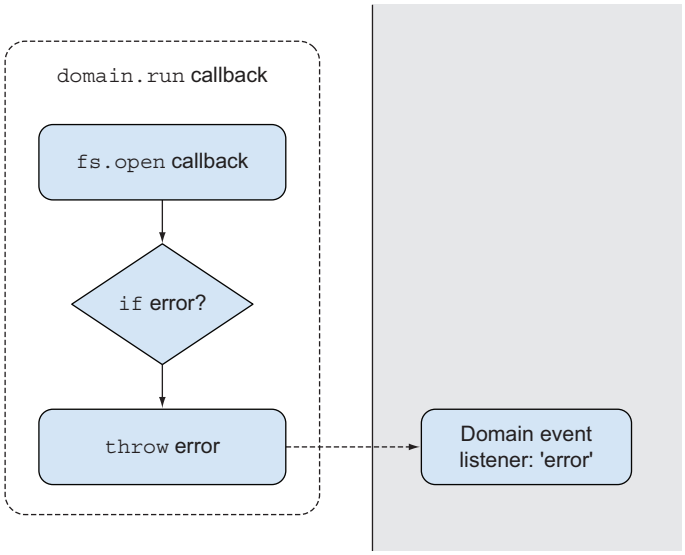

Figure 4.1 Domains help catch errors and handle them with an `EventEmitter`-style API.

4.3 Advanced patterns

This section offers some best practice techniques for solving structural issues found when using `EventEmitter`.

TECHNIQUE 23 Reflection

Sometimes you need to dynamically respond to changes to an instance of an `EventEmitter`, or query its listeners. This technique explains how to do this.

Problem

You need to either catch when a listener has been added to an emitter, or query the existing listeners.

Solution

To track when listeners are added, `EventEmitter` emits a special event called `newListener`. Listeners added to this event will receive the event name and the listener function.

Discussion

In some ways, the difference between writing good Node code and *great* Node code comes down to a deep understanding of `EventEmitter`. Being able to correctly reflect on `EventEmitter` objects gives rise to a whole range of opportunities for creating more flexible and intuitive APIs. One dynamic way of doing this is through the `newListener` event, emitted when listeners are added using the `on` method. Interestingly, this event is emitted by using `EventEmitter` itself—it's implemented by using `emit`.

The next listing shows how to track `newListener` events.

Listing 4.9 Keeping tabs on new listeners

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

function EventTracker() {
  events.EventEmitter.call(this);
}

util.inherits(EventTracker, events.EventEmitter);

var eventTracker = new EventTracker();

eventTracker.on('newListener', function(name, listener) {
  console.log('Event name added:', name);
});

eventTracker.on('a listener', function() {
  // This will cause 'newListener' to fire
});
```

Track whenever new listeners are added.

Even though `'a listener'` is never explicitly emitted in this example, the `newListener` event will still fire. Since the listener's callback function is passed as well as the event name, this is a great way to create simplified public APIs for things that require

access to the original listener function. Listing 4.10 demonstrates this concept by automatically starting a timer when listeners for `pulse` events are added.

Listing 4.10 Automatically triggering events based on new listeners

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

function Pulsar(speed, times) {
  events.EventEmitter.call(this);

  var self = this;
  this.speed = speed;
  this.times = times;

  this.on('newListener', function(eventName, listener) {
    if (eventName === 'pulse') {
      self.start();
    }
  });
}

util.inherits(Pulsar, events.EventEmitter);

Pulsar.prototype.start = function() {
  var self = this;
  var id = setInterval(function() {
    self.emit('pulse');
    self.times--;
    if (self.times === 0) {
      clearInterval(id);
    }
  }, this.speed);
};

var pulsar = new Pulsar(500, 5);

pulsar.on('pulse', function() {
  console.log('.');
});
```

Display a dot for each pulse.

We can go a step further and query `EventEmitter` objects about their listeners by calling `emitter.listeners(event)`. A list of *all* listeners can't be returned in one go, though. The entire list is technically available within the `this._events` object, but this property should be considered private. The `listeners` method currently returns an `Array` instance. This could be used to iterate over multiple listeners if several have been added to a given event—perhaps to remove them at the end of an asynchronous process, or simply to check if any listeners have been added.

In cases where an array of events is available, the `listeners` method will effectively return `this._events[type].slice(0)`. Calling `slice` on an array is a JavaScript shortcut for creating a *copy* of an array. The documentation states that this behavior may

change in the future, so if you really want to create a copy of attached listeners, then call `slice` yourself to ensure you really get a copy and not a reference to a data structure within the emitter instance.

Listing 4.11 adds a `stop` method to the `Pulsar` class. When `stop` is called, it checks to see if there are any listeners; otherwise, it raises an error. Checking for listeners is a good way to prevent incorrect usage, but you don't have to do this in your own code.

Listing 4.11 Querying listeners

```
Pulsar.prototype.stop = function() {
  if (this.listeners('pulse').length === 0) {
    throw new Error('No listeners have been added!');
  }
};

var pulsar = new Pulsar(500, 5);

pulsar.stop();
```

TECHNIQUE 24 Detecting and exploiting EventEmitter

A lot of successful open source Node modules are built on `EventEmitter`. It's useful to spot where `EventEmitter` is being used and to know how to take advantage of it.

Problem

You're working on a large project with several components and want to communicate between them.

Solution

Look for the `emit` and `on` methods whenever you're using either Node's standard modules or open source libraries. For example, the Express `app` object has these methods, and they're great for sending messages within an application.

Discussion

Usually when you're working on a large project, there's a major component that's central to your problem domain. If you're building a web application with Express, then the `app` object is one such component. A quick check of the source shows that this object mixes in `EventEmitter`, so you can take advantage of events to communicate between the disparate components within your project.

Listing 4.12 shows an Express-based example where a listener is bound to an event, and then the event is emitted when a specific route is accessed.

Listing 4.12 Reusing `EventEmitter` in Express

```
var express = require('express');
var app = express();

app.on('hello-alert', function() {
  console.warn('Warning!');
});
```

```
app.get('/', function(req, res){
  res.app.emit('hello-alert');
  res.send('hello world');
});

app.listen(3000);
```

The app object is also available in res.app.

This might seem contrived, but what if the route were defined in another file? In this case, you wouldn't have access to the `app` object, unless it was defined as a global.

Another example of a popular project built on `EventEmitter` is the Node Redis client (https://npmjs.org/package/redis). Instances of `RedisClient` inherit from `EventEmitter`. This allows you to hook into useful events, like the `error` event, as shown in the next listing.

Listing 4.13 Reusing `EventEmitter` in the `redis` module

```
var redis = require('redis'),
var client = redis.createClient();

client.on('error', function(err) {
  console.error('Error:', err);
});

client.on('monitor', function(timestamp, args) {
  console.log('Time:', timestamp, 'arguments:', args);
});

client.on('ready', function() {
  // Start app here
});
```

The monitor event emitted by the redis module for tracking when various internal activities occur

In cases where the route separation technique has been used to store routes in several files, you can actually send events by calling `res.app.emit(event)`. This allows route handlers to communicate back to the `app` object itself.

This might seem like a highly specific Express example, but other popular open source modules are also built on `EventEmitter`—just look for the `emit` and `on` methods. Remember that Node's internal modules like the `process` object and `net.createServer` inherit from `EventEmitter`, and well-written open source modules tend to inherit from these modules as well. This means there's a huge amount of scope for event-based solutions to architectural problems.

This example also highlights another benefit of building projects around `EventEmitter`—asynchronous processes can respond as soon as possible. If the `hello-alert` event performs a very slow operation like sending an email, the person browsing the page might not want to wait for this process to finish. In this case, you can render the requested page while effectively performing a slower operation in the background.

The Node Redis client makes excellent use of `EventEmitter` and the author has written documentation for what each of the methods do. This is a good idea—if

somebody joins your project, they may find it hard to get an overall picture of the events that are being used.

TECHNIQUE 25 Categorizing event names

Some projects just have too many events. This technique shows how to deal with bugs caused by mistyped event names.

Problem

You're losing track of the events in your program, and are concerned that it may be too easy to write an incorrect event name somewhere causing a difficult-to-track bug.

Solution

The easiest way to solve this problem is to use an object to act as a central dictionary for all of the event names. This creates a centralized location of each event in the project.

Discussion

It's hard to keep track of event names littered throughout a project. One way to manage this is to keep each event name in one place. Listing 4.14 demonstrates using an object to categorize event names, based on the previous examples in this chapter.

Listing 4.14 Categorizing event names using an object

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

function MusicPlayer() {
  events.EventEmitter.call(this);
  this.on(MusicPlayer.events.play, this.play.bind(this));
}

var e = MusicPlayer.events = {
  play: 'play',
  pause: 'pause',
  stop: 'stop',
  ff: 'ff',
  rw: 'rw',
  addTrack: 'add-track'
};

util.inherits(MusicPlayer, events.EventEmitter);

MusicPlayer.prototype.play = function() {
  this.playing = true;
};

var musicPlayer = new MusicPlayer();

musicPlayer.on(e.play, function() {
  console.log('Now playing');
});

musicPlayer.emit(e.play);
```

The object used to store the event list is aliased for convenience.

When adding new listeners, users of the class can refer to the events list rather than writing the event names as strings.

Although `EventEmitter` is an integral part of Node's standard library, and an elegant solution to many problems, it can be the source of a lot of bugs in larger projects where people may forget the name of a given event. One way around this is to avoid writing events as strings. Instead, an object can be used with properties that refer to the event name strings.

If you're writing a reusable, open source module, you should consider making this part of the public API so it's easy for people to get a centralized list of event names.

There are other observer pattern implementations that avoid using string event names to effectively type check events. In the next technique we'll look at a few that are available through npm.

Although `EventEmitter` provides a wide array of solutions when working on Node projects, there are alternative implementations out there. The next section includes some popular alternatives.

4.4 Third-party modules and extensions

`EventEmitter` is essentially an *observer pattern* implementation. There are other interpretations of this pattern, which can help scale Node programs to run across several processes or over a network. The next technique introduces some of the more popular alternatives created by the Node community.

TECHNIQUE 26 Alternatives to EventEmitter

`EventEmitter` has a great API and works well in Node programs, but sometimes a problem requires a slightly different solution. This technique explores some alternatives to `EventEmitter`.

Problem

You're trying to solve a problem that doesn't quite fit `EventEmitter`.

Solution

Depending on the exact nature of the problem you're trying to solve, there are several alternatives to `EventEmitter`: publish/subscribe, AMQP, and `js-signals` are some popular alternatives with good support in Node.

Discussion

The `EventEmitter` class is an implementation of the *observer pattern*. A related pattern is publish/subscribe, where publishers send messages that are characterized into classes to subscribers without knowing the details of the subscribers themselves.

The publish/subscribe pattern is often useful in cases where horizontal scaling is required. If you need to run multiple Node processes on multiple servers, then technologies like AMQP and ØMQ can help implement this. They're both specifically designed to solve this class of problem, but may not be as convenient as using the Redis publish/subscribe API if you're already using Redis.

If you need to horizontally scale across a distributed cluster, then an AMQP implementation like RabbitMQ (http://www.rabbitmq.com/) will work well. The `rabbitmq-nodejs-client` (https://github.com/adrai/rabbitmq-nodejs-client) module has a

publish/subscribe API. The following listing shows a simple example of RabbitMQ in Node.

Listing 4.15 Using RabbitMQ with Node

```
var rabbitHub = require('rabbitmq-nodejs-client');
var subHub = rabbitHub.create( { task: 'sub', channel: 'myChannel' } );
var pubHub = rabbitHub.create( { task: 'pub', channel: 'myChannel' } );

subHub.on('connection', function(hub) {
  hub.on('message', function(msg) {
    console.log(msg);
  }.bind(this));
});
subHub.connect();

pubHub.on('connection', function(hub) {
  hub.send('Hello World!');
});
pubHub.connect();
```

Print the message when it's received.

ØMQ (http://www.zeromq.org/) is more popular in the Node community. Justin Tulloss and TJ Holowaychuk's `zeromq.node` module (https://github.com/JustinTulloss/zeromq.node) is a popular binding. The next listing shows just how simple this API is.

Listing 4.16 Using ØMQ with Node

```
var zmq = require('zmq');
var push = zmq.socket('push');
var pull = zmq.socket('pull');

push.bindSync('tcp://127.0.0.1:3000');
pull.connect('tcp://127.0.0.1:3000');
console.log('Producer bound to port 3000');

setInterval(function() {
  console.log('sending work');
  push.send('some work');
}, 500);

pull.on('message', function(msg) {
  console.log('work: %s', msg.toString());
});
```

If you're already using Redis with Node, then it's worth trying out the Pub/Sub API (http://redis.io/topics/pubsub). Listing 4.17 shows an example of this using the Node Redis client (https://github.com/mranney/node_redis).

Listing 4.17 Using Redis Pub/Sub with Node

```
var redis = require('redis');
var client1 = redis.createClient();
var client2 = redis.createClient();
```

```
var msg_count = 0;

client1.on('subscribe', function(channel, count) {
  client2.publish('channel', 'Hello world.');
});

client1.on('message', function(channel, message) {
  console.log('client1 channel ' + channel + ': ' + message);
  client1.unsubscribe();
  client1.end();
  client2.end();
});

client1.subscribe('channel');
```

Be sure to close client connections when using the Redis module.

Finally, if publish/subscribe isn't what you're looking for, then you may want to take a look at `js-signals` (https://github.com/millermedeiros/js-signals). This module is a messaging system that doesn't use strings for the signal names, and dispatching or listening to events that don't yet exist will raise errors.

Listing 4.18 shows how `js-signals` sends and receives messages. Notice how signals are properties of an object, rather than strings, and that listeners can receive an arbitrary number of arguments.

Listing 4.18 Using Redis Pub/Sub with Node

```
var signals = require('signals');
var myObject = {
  started: new signals.Signal()
};

function onStarted(param1, param2){
  console.log(param1, param2);
}

myObject.started.add(onStarted);
myObject.started.dispatch('hello', 'world');
```

Binding a listener to the started signal

Dispatching the signal using two parameters

`js-signals` provides a way of using properties for signal names, as mentioned in technique 25, but in this case the module will raise an error if an unregistered listener is dispatched or bound to. This approach is more like "strongly typed" events, and is very different from most publish/subscribe and event observer implementations.

4.5 Summary

In this chapter you've learned how `EventEmitter` is used through inheritance and multiple inheritance, and how to manage errors with and without domains. You've also seen how to centralize event names, how open source modules build on `EventEmitter`, and some alternative solutions.

What you should take away from this chapter is that although `EventEmitter` is usually used as a base class for inheritance, it's also possible to mix it into existing classes. Also, although `EventEmitter` is a great solution to many problems and used throughout

Node's internals, sometimes other solutions are more optimal. For example, if you're using Redis, then you can take advantage of its publish/subscribe implementation. Finally, `EventEmitter` isn't without its problems; managing large amounts of event names can cause bugs, and now you know how to avoid this by using an object with properties that act as event names.

In the next chapter we'll look at a related topic: streams. Streams are built around an event-based API, so you'll be able to use some of these `EventEmitter` techniques there as well.

WEB DEVELOPMENT/JAVASCRIPT

Node.js IN PRACTICE

Young • Harter

You've decided to use Node.js for your next project and you need the skills to implement Node in production. It would be great to have Node experts Alex Young and Marc Harter at your side to help you tackle those day-to day challenges. With this book, you can!

Node.js in Practice is a collection of 115 thoroughly tested examples and instantly useful techniques guaranteed to make any Node application go more smoothly. Following a common-sense Problem/Solution format, these experience-fueled techniques cover important topics like event-based programming, streams, integrating external applications, and deployment. The abundantly annotated code makes the examples easy to follow, and techniques are organized into logical clusters, so it's a snap to find what you're looking for.

What's Inside

- Common usage examples, from basic to advanced
- Designing and writing modules
- Testing and debugging Node apps
- Integrating Node into existing systems

Written for readers who have a practical knowledge of JavaScript and the basics of Node.js.

Marc Harter works daily on large-scale projects including high-availability real-time applications, streaming interfaces, and other data-intensive systems. **Alex Young** is a seasoned JavaScript developer who blogs regularly at DailyJS.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/Node.jsinPractice

"An in-depth tour of Node.js."
—From the Foreword by Ben Noordhuis, Cofounder of StrongLoop, Inc.

"The missing manual for Node.js, packed with real-world examples!"
—Kevin Baister
1KB Software Solutions Ltd.

"Essential recipes for the server-side JavaScript developer."
—Gregor Zurowski, Sotheby's

"Useful techniques and resources that help with problem solving, debugging, and troubleshooting."
—Michael Piscatello
MBP Enterprises, LLC

ISBN 13: 978-1-617290-93-0
ISBN 10: 1-617290-93-9
54999
9 781617 290930

MANNING $49.99 / Can $52.99 [INCLUDING eBOOK]