The Node Craftsman Book

An advanced Node.js tutorial

Leanpub

Manuel Kiessling

The Node Craftsman Book

An advanced Node.js tutorial

Manuel Kiessling

This book is for sale at http://leanpub.com/nodecraftsman

This version was published on 2014-09-13

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Manuel Kiessling by spreading the word about this book on Twitter!

The suggested hashtag for this book is #nodecraftsman.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#nodecraftsman

Also By Manuel Kiessling

The Node Beginner Book

Node□□

El Libro Principiante de Node

For Nora, Aaron and Melinda.

Contents

Preface

About

The aim of this book is to help beginning JavaScript programmers who already know how to write basic Node.js applications in mastering JavaScript and Node.js thoroughly.

Status

This document is work in progress and is currently about 30% done. It was last updated on September 10, 2014.

All code examples have been tested to work with Node.js v0.10.31.

Intended audience

This book will fit best for readers that are familiar with the basic concepts of JavaScript and have already written some basic Node.js applications.

As this book is a sequel to *The Node Beginner Book*, I recommend reading it before starting with this book.

Part 1: Node.js basics in detail

Introduction to Part 1

The goal of this book is to enable you to write Node.js programs of any level of complexity, from simple utilities to complex applications that use several external modules and consist of several layers of code, talking to external systems and serving thousands of users.

In order to do so, you need to learn about all the different aspects of Node.js - the tools, the methodologies, the libraries, the APIs, the best practices - and you need to learn how to put all of that together to create a working whole.

Therefore, I have split this book into two parts: A collection of different basics in Part 1, and a thorough tutorial on how to put these basics together to build a complex application in Part 2.

In Part 1, every chapter stands on its own and isn't directly related to the other chapters. Part 2 is more like a continuous story that starts from scratch and gives you a finished and working application at the end.

Thus, let's start with Part 1 and look at all the different facets of Node.js software development.

Working with NPM and Packages

We already used NPM, the *Node Package Manager*, in order to install a single package and its dependencies for the example project in *The Node Beginner Book*.

However, there is much more to NPM, and a more thorough introduction is in order.

The most useful thing that NPM does isn't installing packages. This could be done manually with slightly more hassle. What's really useful about NPM is that it handles package dependencies. A lot of packages that we need for our own project need other packages themselves. Have a look at https://npmjs.org/package/request, for example. It's the overview page for the NPM package *request*. According to its description, it provides a "simplified HTTP request client". But in order to do so, *request* not only uses its own code. It also needs other packages for doing its job. These are listed under "Dependencies": *qs*, *json-stringify-safe*, and others.

Whenever we use the NPM command line tool, *npm*, in order to install a package, NPM not only pulls the package itself, but also its dependencies, and installs those as well.

Using *npm install request* is simply a manual way to implicitly say "my project depends on *request*, please install it for me". However, there is an explicit way of defining dependencies for our own projects, which also allows to have all dependencies resolved and installed automatically.

In order to use this mechanism, we need to create a control file within our project that defines our dependencies. This control file is then used by NPM to resolve and install those dependencies.

This control file must be located at the top level folder of our project, and must be named *package.json*.

This is what a *package.json* file looks like for a project that depends on *request*:

```
{
  "dependencies": {
    "request": ""
  }
}
```

Having this file as part of our code base makes NPM aware of the dependencies of our project without the need to explicitly tell NPM what to install by hand.

We can now use NPM to automatically pull in all dependencies of our project, simply by executing *npm install* within the top level folder of our code base.

In this example, this doesn't look like much of a convenience win compared to manually installing *request*, but once we have more than a handful of dependencies in our projects, it really makes a difference.

The *package.json* file also allows us to "pin" dependencies to certain versions, i.e., we can define a version number for every dependency, which ensures that NPM won't pull the latest version automatically, but exactly the version of a package we need:

```
{
  "dependencies": {
    "request": "2.27.0"
  }
}
```

In this case, NPM will always pull *request* version 2.27.0 (and the dependencies of this version), even if newer versions are available.

Patterns are possible, too:

```
{
  "dependencies": {
    "request": "2.27.x"
  }
}
```

The *x* is a placeholder for any number. This way, NPM would pull in *request* version 2.27.0 and 2.27.5, but not 2.28.0.

The official documentation at https://npmjs.org/doc/json.html#dependencies has more examples of possible dependency definitions.

Please note that the *package.json* file does much more than just defining dependencies. We will dig deeper in the course of this book.

For now, we are prepared to use NPM for resolving the dependencies that arise in our first project - our first test-driven Node.js application.

Object-oriented JavaScript

Let's talk about object-orientation and inheritance in JavaScript.

The good news is that it's actually quite simple, but the bad news is that it works completely different than object-orientation in languages like C++, Java, Ruby, Python or PHP, making it not-quite-so simple to understand.

But fear not, we are going to take it step by step.

Blueprints versus finger-pointing

Let's start by looking at how "typical" object-oriented languages actually create objects.

We are going to talk about an object called *myCar*. *myCar* is our bits-and-bytes representation of an incredibly simplified real world car. It could have attributes like *color* and *weight*, and methods like *drive* and *honk*.

In a real application, *myCar* could be used to represent the car in a racing game - but we are going to completely ignore the context of this object, because we will talk about the nature and usage of this object in a more abstract way.

If you would want to use this myCar object in, say, Java, you need to define the blueprint of this specific object first - this is what Java and most other object-oriented languages call a *class*.

If you want to create the object *myCar*, you tell Java to "build a new object after the specification that is laid out in the class Car".

The newly built object shares certain aspects with its blueprint. If you call the method honk on your object, like so:

```
myCar.honk();
```

then the Java VM will go to the class of *myCar* and look up which code it actually needs to execute, which is defined in the *honk* method of class *Car*.

Ok, nothing shockingly new here. Enter JavaScript.

A classless society

JavaScript does not have classes. But as in other languages, we would like to tell the interpreter that it should build our *myCar* object following a certain pattern or schema or blueprint - it would be quite tedious to create every *car* object from scratch, "manually" giving it the attributes and methods it needs every time we build it.

If we were to create 30 *car* objects based on the *Car* class in Java, this object-class relationship provides us with 30 cars that are able to drive and honk without us having to write 30 *drive* and *honk* methods.

How is this achieved in JavaScript? Instead of an object-class relationship, there is an object-object relationship.

Where in Java our *myCar*, asked to *honk*, says "go look at this class over there, which is my *blueprint*, to find the code you need", JavaScript says "go look at that other object over there, which is my *prototype*, it has the code you are looking for".

Building objects via an object-object relationship is called Prototype-based programming, versus Class-based programming used in more traditional languages like Java.

Both are perfectly valid implementations of the object-oriented programming paradigm - it's just two different approaches.

Creating objects

Let's dive into code a bit, shall we? How could we set up our code in order to allow us to create our *myCar* object, ending up with an object that is a Car and can therefore *honk* and *drive*?

Well, in the most simple sense, we can create our object completely from scratch, or ex nihilo if you prefer the boaster expression.

It works like this:

```
1  var myCar = {};
2
3  myCar.honk = function() {
4    console.log('honk honk');
5  };
6
7  myCar.drive = function() {
8    console.log('vrooom...');
9  };
```

This gives us an object called *myCar* that is able to *honk* and *drive*:

```
myCar.honk();   // outputs "honk honk"
myCar.drive();  // outputs "vrooom..."
```

However, if we were to create 30 cars this way, we would end up defining the honk and drive behaviour of every single one, something we said we want to avoid.

In real life, if we made a living out of creating, say, pencils, and we don't want to create every pencil individually by hand, then we would consider building a pencil-making machine, and have this machine create the pencils for us.

After all, that's what we implicitly do in a class-based language like Java - by defining a class Car, we get the car-maker for free:

```
Car myCar = new Car();
```

will build the *myCar* object for us based on the *Car* blueprint. Using the *new* keyword does all the magic for us.

JavaScript, however, leaves the responsibility of building an object creator to us. Furthermore, it gives us a lot of freedom regarding the way we actually build our objects.

In the most simple case, we can write a function which creates "plain" objects that are exactly like our "ex nihilo" object, and that don't really share any behaviour - they just happen to roll out of the factory with the same behaviour copied onto every single one, if you want so.

Or, we can write a special kind of function that not only creates our objects, but also does some behind-the-scenes magic which links the created objects with their creator. This allows for a true sharing of behaviour: functions that are available on all created objects point to a single implementation. If this function implementation changes after objects have been created, which is possible in JavaScript, the behaviour of all objects sharing the function will change accordingly.

Let's examine all possible ways of creating objects in detail.

Using a simple function to create plain objects

In our first example, we created a plain *myCar* object out of thin air - we can simply wrap the creation code into a function, which gives us a very basic object creator:

```
1 var makeCar = function() {
2   var newCar = {};
3   newCar.honk = function() {
4     console.log('honk honk');
5   };
6 };
```

For the sake of brevity, the drive function has been omitted.

We can then use this function to mass-produce cars:

```
var makeCar = function() {
  var newCar = {}
  newCar.honk = function() {
    console.log('honk honk');
  };
  return newCar;
};

myCar1 = makeCar();
myCar2 = makeCar();
myCar3 = makeCar();
```

One downside of this approach is efficiency: for every *myCar* object that is created, a new *honk* function is created and attached - creating 1,000 objects means that the JavaScript interpreter has to allocate memory for 1,000 functions, although they all implement the same behaviour. This results in an unnecessarily high memory footprint of the application.

Secondly, this approach deprives us of some interesting opportunities. These *myCar* objects don't share anything - they were built by the same creator function, but are completely independent from each other.

It's really like with real cars from a real car factory: They all look the same, but once they leave the assembly line, they are totally independent. If the manufacturer should decide that pushing the horn on already produced cars should result in a different type of honk, all cars would have to be returned to the factory and modified.

In the virtual universe of JavaScript, we are not bound to such limits. By creating objects in a more sophisticated way, we are able to magically change the behaviour of all created objects at once.

Using a constructor function to create objects

In JavaScript, the entities that create objects with shared behaviour are functions which are called in a special way. These special functions are called *constructors*.

Let's create a constructor for cars. We are going to call this function *Car*, with a capital *C*, which is common practice to indicate that this function is a constructor.

In a way, this makes the constructor function a class, because it does some of the things a class (with a constructor method) does in a traditional OOP language. However, the approach is not identical, which is why constructor functions are often called *pseudo-classes* in JavaScript. I will simply call them classes or constructor functions.

Because we are going to encounter two new concepts that are both necessary for shared object behaviour to work, we are going to approach the final solution in two steps.

Step one is to recreate the previous solution (where a common function spilled out independent car objects), but this time using a constructor:

```
var Car = function() {
  this.honk = function() {
    console.log('honk honk');
  };
};
```

When this function is called using the *new* keyword, like so:

```
var myCar = new Car();
```

it implicitly returns a newly created object with the *honk* function attached.

Using *this* and *new* makes the explicit creation and return of the new object unnecessary - it is created and returned "behind the scenes" (i.e., the *new* keyword is what creates the new, "invisible" object, and secretly passes it to the *Car* function as its *this* variable).

You can think of the mechanism at work a bit like in this pseudo-code:

```
// Pseudo-code, for illustration only!

var Car = function(this) {
  this.honk = function() {
    console.log('honk honk');
  };
  return this;
};

var newObject = {};
var myCar = Car(newObject);
```

As said, this is more or less like our previous solution - we don't have to create every car object manually, but we still cannot modify the *honk* behaviour only once and have this change reflected in all created cars.

But we laid the first cornerstone for it. By using a constructor, all objects received a special property that links them to their constructor:

```
var Car = function() {
  this.honk = function() {
    console.log('honk honk');
  };
};

var myCar1 = new Car();
var myCar2 = new Car();

console.log(myCar1.constructor);  // outputs [Function: Car]
console.log(myCar2.constructor);  // outputs [Function: Car]
```

All created *myCars* are linked to the *Car* constructor. This is what actually makes them a *class* of related objects, and not just a bunch of objects that happen to have similar names and identical functions.

Now we have finally reached the moment to get back to the mysterious prototype we talked about in the introduction.

Using prototyping to efficiently share behaviour between objects

As stated there, while in class-based programming the class is the place to put functions that all objects will share, in prototype-based programming, the place to put these functions is the object which acts as the prototype for our objects at hand.

But where is the object that is the prototype of our *myCar* objects - we didn't create one!

It has been implicitly created for us, and is assigned to the *Car.prototype* property (in case you wondered, JavaScript functions are objects too, and they therefore have properties).

Here is the key to sharing functions between objects: Whenever we call a function on an object, the JavaScript interpreter tries to find that function within the queried object. But if it doesn't find the function within the object itself, it asks the object for the pointer to its prototype, then goes to the prototype, and asks for the function there. If it is found, it is then executed.

This means that we can create *myCar* objects without any functions, create the *honk* function in their prototype, and end up having *myCar* objects that know how to honk - because everytime the interpreter tries to execute the *honk* function on one of the myCar objects, it will be redirected to the prototype, and execute the *honk* function which is defined there.

Here is how this setup can be achieved:

```
var Car = function() {};

Car.prototype.honk = function() {
  console.log('honk honk');
};

var myCar1 = new Car();
var myCar2 = new Car();

myCar1.honk();  // executes Car.prototype.honk() and outputs "honk honk"
myCar2.honk();  // executes Car.prototype.honk() and outputs "honk honk"
```

Our constructor is now empty, because for our very simple cars, no additional setup is necessary.

Because both *myCar*s are created through this constructor, their prototype points to *Car.prototype* - executing *myCar1.honk()* and *myCar2.honk()* always results in *Car.prototype.honk()* being executed.

Let's see what this enables us to do. In JavaScript, objects can be changed at runtime. This holds true for prototypes, too. Which is why we can change the *honk* behaviour of all our cars even after they have been created:

```
var Car = function() {};

Car.prototype.honk = function() {
  console.log('honk honk');
};

var myCar1 = new Car();
var myCar2 = new Car();

myCar1.honk();  // executes Car.prototype.honk() and outputs "honk honk"
myCar2.honk();  // executes Car.prototype.honk() and outputs "honk honk"

Car.prototype.honk = function() {
  console.log('meep meep');
};

myCar1.honk();  // executes Car.prototype.honk() and outputs "meep meep"
myCar2.honk();  // executes Car.prototype.honk() and outputs "meep meep"
```

Of course, we can also add additional functions at runtime:

```
var Car = function() {};

Car.prototype.honk = function() {
  console.log('honk honk');
};

var myCar1 = new Car();
var myCar2 = new Car();

Car.prototype.drive = function() {
  console.log('vrooom...');
};

myCar1.drive();  // executes Car.prototype.drive() and outputs "vrooom..."
myCar2.drive();  // executes Car.prototype.drive() and outputs "vrooom..."
```

But we could even decide to treat only one of our cars differently:

```
var Car = function() {};

Car.prototype.honk = function() {
  console.log('honk honk');
};

var myCar1 = new Car();
var myCar2 = new Car();

myCar1.honk();  // executes Car.prototype.honk() and outputs "honk honk"
myCar2.honk();  // executes Car.prototype.honk() and outputs "honk honk"

myCar2.honk = function() {
  console.log('meep meep');
};

myCar1.honk();  // executes Car.prototype.honk() and outputs "honk honk"
myCar2.honk();  // executes myCar2.honk() and outputs "meep meep"
```

It's important to understand what happens behind the scenes in this example. As we have seen, when calling a function on an object, the interpreter follows a certain path to find the actual location of that function.

While for *myCar1*, there still is no *honk* function within that object itself, that no longer holds true for *myCar2*. When the interpreter calls *myCar2.honk()*, there now is a function within *myCar2* itself.

Therefore, the interpreter no longer follows the path to the prototype of *myCar2*, and executes the function within *myCar2* instead.

That's one of the major differences to class-based programming: while objects are relatively "rigid" e.g. in Java, where the structure of an object cannot be changed at runtime, in JavaScript, the prototype-based approach links objects of a certain class more loosely together, which allows to change the structure of objects at any time.

Also, note how sharing functions through the constructor's prototype is way more efficient than creating objects that all carry their own functions, even if they are identical. As previously stated, the engine doesn't know that these functions are meant to be identical, and it has to allocate memory for every function in every object. This is no longer true when sharing functions through a common prototype - the function in question is placed in memory exactly once, and no matter how many *myCar* objects we create, they don't carry the function themselves, they only refer to their constructor, in whose prototype the function is found.

To give you an idea of what this difference can mean, here is a very simple comparison. The first example creates 1,000,000 objects that all have the function directly attached to them:

```
var C = function() {
  this.f = function(foo) {
    console.log(foo);
  };
};

var a = [];
for (var i = 0; i < 1000000; i++) {
  a.push(new C());
}
```

In Google Chrome, this results in a heap snapshot size of 328 MB. Here is the same example, but now the function is shared through the constructor's prototype:

```
var C = function() {};

C.prototype.f = function(foo) {
  console.log(foo);
};

var a = [];
for (var i = 0; i < 1000000; i++) {
  a.push(new C());
}
```

This time, the size of the heap snapshot is only 17 MB, i.e., only about 5% of the non-efficient solution.

Object-orientation, prototyping, and inheritance

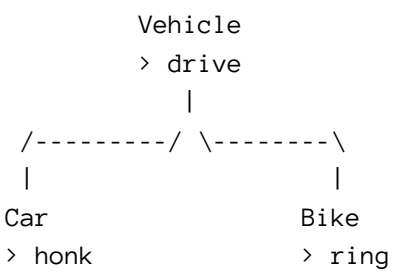
So far, we haven't talked about inheritance in JavaScript, so let's do this now.

It's useful to share behaviour within a certain class of objects, but there are cases where we would like to share behaviour between different, but similar classes of objects.

Imagine our virtual world not only had cars, but also bikes. Both drive, but where a car has a horn, a bike has a bell.

Being able to drive makes both objects vehicles, but not sharing the *honk* and *ring* behaviour distinguishes them.

We could illustrate their shared and local behaviour as well as their relationship to each other as follows:



Designing this relationship in a class-based language like Java is straightforward: We would define a class *Vehicle* with a method *drive*, and two classes *Car* and *Bike* which both extend the *Vehicle* class, and implement a *honk* and a *ring* method, respectively.

This would make the *car* as well as *bike* objects inherit the *drive* behaviour through the inheritance of their classes.

How does this work in JavaScript, where we don't have classes, but prototypes?

Let's look at an example first, and then dissect it. To keep the code short for now, let's only start with a car that inherits from a vehicle:

```
1  var Vehicle = function() {};
2
3  Vehicle.prototype.drive = function() {
4    console.log('vrooom...');
5  };
6
7
8  var Car = function() {};
9
10 Car.prototype = new Vehicle();
```

```
11
12 Car.prototype.honk = function() {
13   console.log('honk honk');
14 };
15
16
17 var myCar = new Car();
18
19 myCar.honk();   // outputs "honk honk"
20 myCar.drive();  // outputs "vrooom..."
```

In JavaScript, inheritance runs through a chain of prototypes.

The prototype of the *Car* constructor is set to a newly created vehicle object, which establishes the link structure that allows the interpreter to look for methods in "parent" objects.

The prototype of the *Vehicle* constructor has a function *drive*. Here is what happens when the *myCar* object is asked to *drive()*:

- The interpreter looks for a *drive* method within the *myCar* object, which does not exist
- The interpreter then asks the *myCar* object for its prototype, which is the prototype of its constructor *Car*
- When looking at *Car.prototype*, the interpreter sees a *vehicle* object which has a function *honk* attached, but no *drive* function
- Thus, the interpreter now asks this *vehicle* object for its prototype, which is the prototype of its constructor *Vehicle*
- When looking at *Vehicle.prototype*, the interpreter sees an object which has a *drive* function attached - the interpreter now knows which code implements the *myCar.drive()* behaviour, and executes it

A classless society, revisited

We just learned how to emulate the traditional OOP inheritance mechanism. But it's important to note that in JavaScript, that is only one valid approach to create objects that are related to each other.

It was Douglas Crockford who came up with another clever solution, which allows objects to inherit from each other directly. It's a native part of JavaScript by now - it's the *Object.create()* function, and it works like this:

```
Object.create = function(o) {
  var F = function() {};
  F.prototype = o;
  return new F();
};
```

We learned enough now to understand what's going on. Let's analyze an example:

```
var vehicle = {};
vehicle.drive = function () {
  console.log('vrooom...');
};

var car = Object.create(vehicle);
car.honk = function() {
  console.log('honk honk');
};

var myCar = Object.create(car);

myCar.honk();   // outputs "honk honk"
myCar.drive();  // outputs "vrooom..."
```

While being more concise and expressive, this code achieves exactly the same behaviour, without the need to write dedicated constructors and attaching functions to their prototype. As you can see, *Object.create()* handles both behind the scenes, on the fly. A temporary constructor is created, its prototype is set to the object that serves as the role model for our new object, and a new object is created from this setup.

Conceptually, this is really the same as in the previous example where we defined that *Car.prototype* shall be a *new Vehicle()*;.

But wait! We created the functions *drive* and *honk* within our objects, not on their prototypes - that's memory-inefficient!

Well, in this case, it's actually not. Let's see why:

```
var vehicle = {};
vehicle.drive = function () {
  console.log('vrooom...');
};

var car = Object.create(vehicle);
car.honk = function() {
  console.log('honk honk');
};

var myVehicle = Object.create(vehicle);
var myCar1 = Object.create(car);
var myCar2 = Object.create(car);

myCar1.honk();  // outputs "honk honk"
myCar2.honk();  // outputs "honk honk"

myVehicle.drive();  // outputs "vrooom..."
myCar1.drive();     // outputs "vrooom..."
myCar2.drive();     // outputs "vrooom..."
```

We have now created a total of 5 objects, but how often do the *honk* and *drive* methods exist in memory? Well, how often have they been defined? Just once - and therefore, this solution is basically as efficient as the one where we built the inheritance manually. Let's look at the numbers:

```
var c = {};
c.f = function(foo) {
  console.log(foo);
};

var a = [];
for (var i = 0; i < 1000000; i++) {
  a.push(Object.create(c));
}
```

Turns out, it's not *exactly* identical - we end up with a heap snapshot size of 40 MB, thus there seems to be some overhead involved. However, in exchange for much better code, this is probably more than worth it.

Summary

By now, it's probably clear what the main difference between classical OOP languages and JavaScript is, conceptually: While classical languages like Java provide *one* way to manage object creation and

behaviour sharing (through classes and inheritance), and this way is enforced by the language and "baked in", JavaScript starts at a slightly lower level and provides building blocks that allow us to create several different mechanisms for this.

Whether you decide to use these building blocks to recreate the traditional class-based pattern, let your objects inherit from each other directly, with the concept of classes getting in the way, or if you don't use the object-oriented paradigm at all and just solve the problem at hand with pure functional code: JavaScript gives you the freedom to choose the best methodology for any situation.

Test-Driven Node.js Development

The code examples in *The Node Beginner Book* only described a toy project, and we came away with not writing any tests for it. If writing tests is new for you, and you have not yet worked on software in a test-driven manner, then I invite you to follow along and give it a try.

We need to decide on a test framework that we will use to implement our tests. A lack of choice is not an issue in the JavaScript and Node.js world, as there are dozens of frameworks available. Personally, I prefer *Jasmine*, and will therefore use it for my examples.

Jasmine is a framework that follows the philosophy of *behaviour-driven development*, which is kind of a "subculture" within the community of test-driven developers. This topic alone could easily fill its own book, thus I'll give only a brief introduction.

The idea is to begin development of a new software unit with its specification, followed by its implementation (which, by definition, must satisfy the specification).

Let's make up a real world example: we order a table from a carpenter. We do so by *specifying* the end result:

"I need a table with a top that is 6 x 3 ft. The height of the top must be adjustable between 2.5 and 4.0 ft. I want to be able to adjust the top's height without standing up from my chair. I want the table to be black, and cleaning it with a wet cloth should be possible without damaging the material. My budget is $500."

Such a specification allows to share a goal between us and the carpenter. We don't have to care for how exactly the carpenter will achieve this goal. As long as the delivered product fits our specification, both of us can agree that the goal has been reached.

With a test-driven or behaviour-driven approach, this idea is applied to building software. You wouldn't build a piece of software and then define what it's supposed to do. You need to know in advance what you expect a unit of software to do. Instead of doing this vaguely and implicitly, a test-driven approach asks you to do the specification exactly and explicitly. Because we work on software, our specification can be software, too: we only need to write functions that check if our unit does what it is expected to do. These check functions are unit tests.

Let's create a software unit which is covered by tests that describe its expected behaviour. In order to actually drive the creation of the code with the tests, let's write the tests first. We then have a clearly defined goal: making the tests pass by implementing code that fulfills the expected behaviour, and nothing else.

In order to do so, we create a new Node.js project with two folders in it:

```
src/
spec/
```

spec is where our test cases go - in Jasmine lingo, these are called "specifications", hence "spec". The *spec* folder mirrors the file structure under *src*, i.e., a source file at *src/foo.js* is mirrored by a specification at *spec/fooSpec.js*.

Following the tradition, we will test and implement a "Hello World" code unit. Its expected behaviour is to return a string "Hello World, Joe" if called with "Joe" as its first and only parameter. This behaviour can be specified by writing a unit test.

To do so, we create a file *spec/greetSpec.js*, with the following content:

```
1  'use strict';
2
3  var greet = require('../src/greet');
4
5  describe('greet', function() {
6
7    it('should greet the given name', function() {
8      expect(greet('Joe')).toEqual('Hello Joe!');
9    });
10
11   it('should greet no-one special if no name is given', function() {
12     expect(greet()).toEqual('Hello world!');
13   });
14
15 });
```

This is a simple, yet complete specification. It is a programmatical description of the behaviour we expect from a yet-to-be-written function named *greet*.

The specification says that if the function *greet* is called with *Joe* as its first and only parameter, the return value of the function call should be the string "Hello Joe!". If we don't supply a name, the greeting should be generic.

As you can see, Jasmine specifications have a two-level structure. The top level of this structure is a *describe* block, which consists of one or more *it* blocks.

An *it* block describes a single expected behaviour of a single unit under test, and a *describe* block summarizes one or more blocks of expected behaviours, therefore completely specifying all expected behaviours of a unit.

Let's illustrate this with a real-world "unit" described by a Jasmine specification:

```
describe('A candle', function() {

  it('should burn when lighted', function() {
    // ...
  });

  it('should grow smaller while burning', function() {
    // ...
  });

  it('should no longer burn when all wax has been burned', function() {
    // ...
  });

  it('should go out when no oxygen is available to it', function() {
    // ...
  });

});
```

As you can see, a Jasmine specification gives us a structure to fully describe how a given unit should behave.

Not only can we *describe* expected behaviour, we can also *verify* it. This can be done by running the test cases in the specification against the actual implementation.

After all, our Jasmine specification is just another piece of JavaScript code which can be executed. The NPM package *jasmine-node* ships with a test case runner which allows us to execute the test case, with the added benefit of a nice progress and result output.

Let's create a *package.json* file that defines *jasmine-node* as a dependency of our application - then we can start running the test cases of our specification.

As described earlier, we need to place the *package.json* file at the topmost folder of our project. Its content should be as follows:

```
{
  "devDependencies": {
    "jasmine-node": ""
  }
}
```

We talked about the *dependencies* section of *package.json* before - but here we declare *jasmine-node* in a *devDependencies* block. The result is basically the same: NPM knows about this dependency

and installs the package and its dependencies for us. However, dev dependencies are not needed to run our application - as the name suggests, they are only needed during development.

NPM allows to skip dev dependencies when deploying applications to a production system - we will get to this later.

In order to have NPM install *jasmine-node*, please run

```
npm install
```

in the top folder of your project.

We are now ready to test our application against its specification.

Of course, our *greet* function cannot fulfill its specification yet, simply because we have not yet implemented it. Let's see how this looks by running the test cases. From the root folder of our new project, execute the following:

```
./node_modules/jasmine-node/bin/jasmine-node spec/greetSpec.js
```

As you can see, Jasmine isn't too happy with the results yet. We refer to a Node module in *src/greet.js*, a file that doesn't even exist, which is why Jasmine bails out before even starting the tests:

```
Exception loading: spec/greetSpec.js
{ [Error: Cannot find module '../src/greet'] code: 'MODULE_NOT_FOUND' }
```

Well, let's create the module, in file *src/greet.js*:

```
1 'use strict';
2
3 var greet = function() {};
4
5 module.exports = greet;
```

Now we have a general infrastructure, but of course we do not yet behave as the specification wishes. Let's run the test cases again:

```
FF

Failures:

  1) greet should greet the given name
   Message:
     TypeError: object is not a function
   Stacktrace:
     TypeError: object is not a function
    at null.<anonymous> (./spec/greetSpec.js:8:12)

  2) greet should greet no-one special if no name is given
   Message:
     TypeError: object is not a function
   Stacktrace:
     TypeError: object is not a function
    at null.<anonymous> (./spec/greetSpec.js:12:12)

Finished in 0.015 seconds
2 tests, 2 assertions, 2 failures, 0 skipped
```

Jasmine tells us that it executed two test cases that contained a total of two assertions (or expectations), and because these expectations could not be satisfied, the test run ended with two failures.

It's time to satisfy the first expectation of our specification, in file *src/greet.js*:

```
1  'use strict';
2
3  var greet = function(name) {
4    return 'Hello ' + name + '!';
5  };
6
7  module.exports = greet;
```

Another test case run reveals that we are getting closer:

```
.F

Failures:

  1) greet should greet no-one special if no name is given
   Message:
     Expected 'Hello undefined!' to equal 'Hello world!'.
   Stacktrace:
     Error: Expected 'Hello undefined!' to equal 'Hello world!'.
    at null.<anonymous> (spec/greetSpec.js:12:21)

Finished in 0.015 seconds
2 tests, 2 assertions, 1 failure, 0 skipped
```

Our first test case passes - *greet* can now correctly greet people by name. We still need to handle the case where no name was given:

```
1  'use strict';
2
3  var greet = function(name) {
4    if (name === undefined) {
5      name = 'world';
6    }
7    return 'Hello ' + name + '!';
8  };
9
10 module.exports = greet;
```

And that does the job:

```
..

Finished in 0.007 seconds
2 tests, 2 assertions, 0 failures, 0 skipped
```

We have now created a piece of software that behaves according to its specification.

You'll probably agree that our approach to create this unbelievably complex unit of software - the *greet* function - in a test-driven way doesn't prove the greatness of test-driven development in any way. That's not the goal of this chapter. It merely sets the stage for what's to come. We are going to create real, comprehensive software through the course of this book, and this is where the advantages of a test-driven approach can be experienced.

Synchronous and Asynchronous operations explained

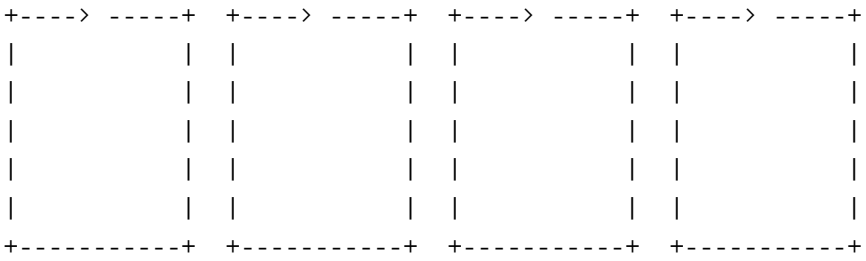
Visualizing the Node.js execution model

For the chapters that follow it's important to fully understand what it means, conceptually, that a Node.js application has synchronous and asynchronous operations, and how both operations interact with each other.

Let's try to build this understanding step by step.

The first concept that we need to understand is that of the Node.js event loop. The event loop is the execution model of a running Node.js application.

We can visualize this model as a row of loops:



I've drawn boxes because circles look really clumsy in ascii art. So, these here look like rectangles, but please imagine them as circles - circles with an arrow, which means that one circle represents one iteration through the event loop.

Another visualization could be the following pseudo-code:

```
while (I still have stuff to do) {
    do stuff;
}
```

Conceptually, at the very core of it, it's really that simple: Node.js starts, loads our application, and then loops until there is nothing left to do - at which point our application terminates.

What kind of stuff is happening inside one loop iteration? Let's first look at a very simple example, a Hello World application like this:

```
console.log('Hello');
console.log('World');
```

This is the visualization of the execution of this application in our ascii art:

```
+----> ---------+
|               |
| Write         |
| 'Hello'       |
| to the screen |
|               |
| Write         |
| 'World'       |
| to the screen |
|               |
+---------------+
```

Yep, that's it: Just one iteration, and then, exit the application.

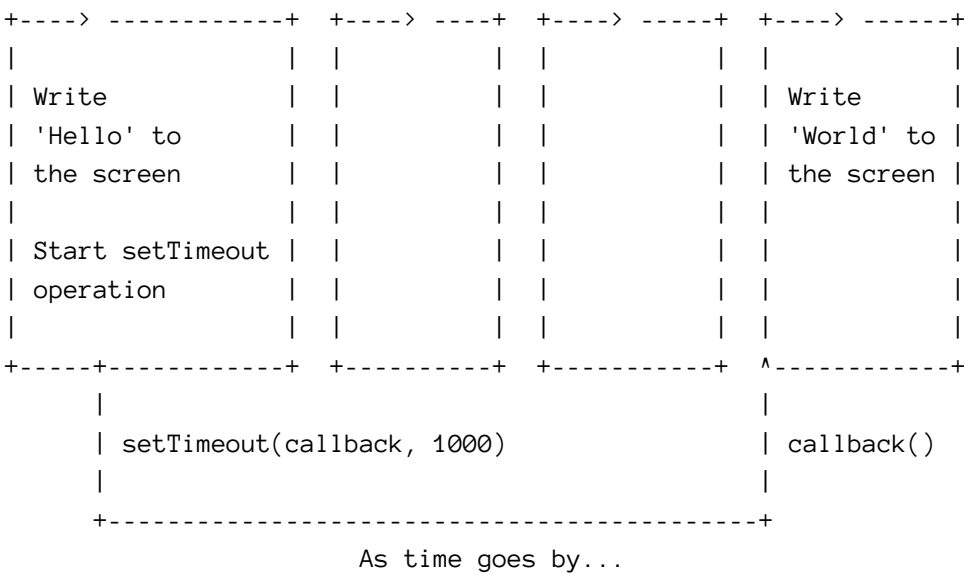
The things we asked our application to do - writing text to the screen, and then writing another text to the screen, using *console.log* - are *synchronous* operations. They both happen within the same (and in this case, only) iteration through the event loop.

Let's look at the model when we bring asynchronous operations into the game, like this:

```
console.log('Hello');

setTimeout(function() {
  console.log('World');
}, 1000);
```

This still prints *Hello* and then *World* to the screen, but the second text is printed with a delay of 1000 ms.

setTimeout, you may have guessed it, is an *asynchronous* operation. We pass the code to be executed in the body of an anonymous function - the so-called *callback function*. Why do we do so? The visualization helps to understand why:



This, again at the very core of it, is what calling an asynchronous function does: It starts an operation *outside the event loop*. Conceptually, Node.js starts the asynchronous operation and makes a mental note that when this operations triggers an event, the anonymous function that was passed to the operation needs to be called.

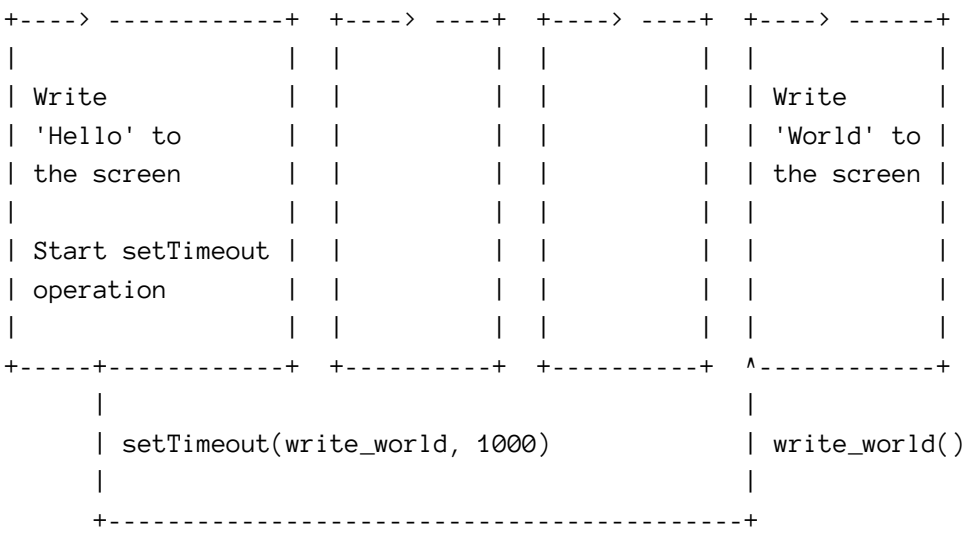
Hence, the *event loop*: as long as asynchronous operations are ongoing, the Node.js process loops, waiting for events from those operations. As soon as no more asynchronous operations are ongoing, the looping stops and our application terminates.

Note that the visualization isn't detailed enough to show that Node.js checks for outside events *between* loop iterations.

Just to be clear: callback functions don't need to be anonymous inline functions:

```
var write_world = function() {
  console.log('World');
};

console.log('Hello');

setTimeout(write_world, 1000);
```

results in:

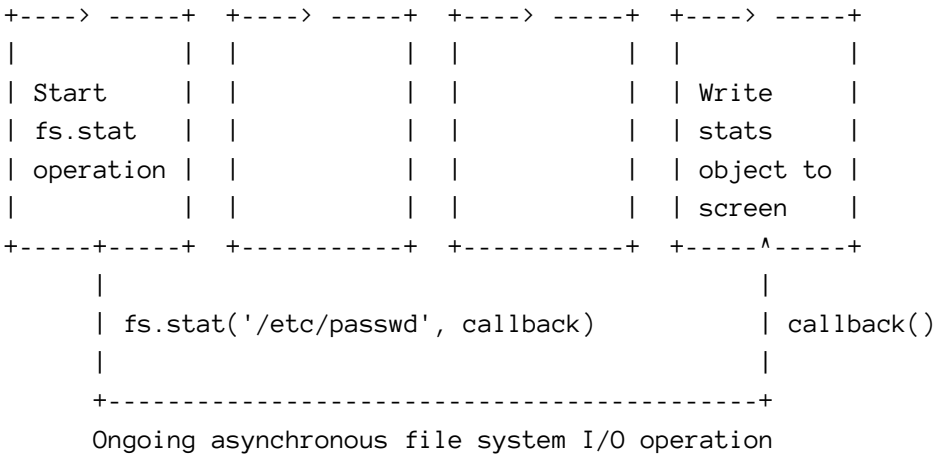


It's just that more often than not, declaring a named function and passing it as the callback isn't worth the hassle, because very often we need the functionality described in the function only once.

Let's look at a slightly more interesting asynchronous operation, the *fs.stat* call - it starts an IO operation that looks at the file system information for a given path, that is, stuff like the inode number and the size etc.:

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

fs.stat('/etc/passwd', function(err, stats) {
  console.dir(stats);
});
```

The visualization:

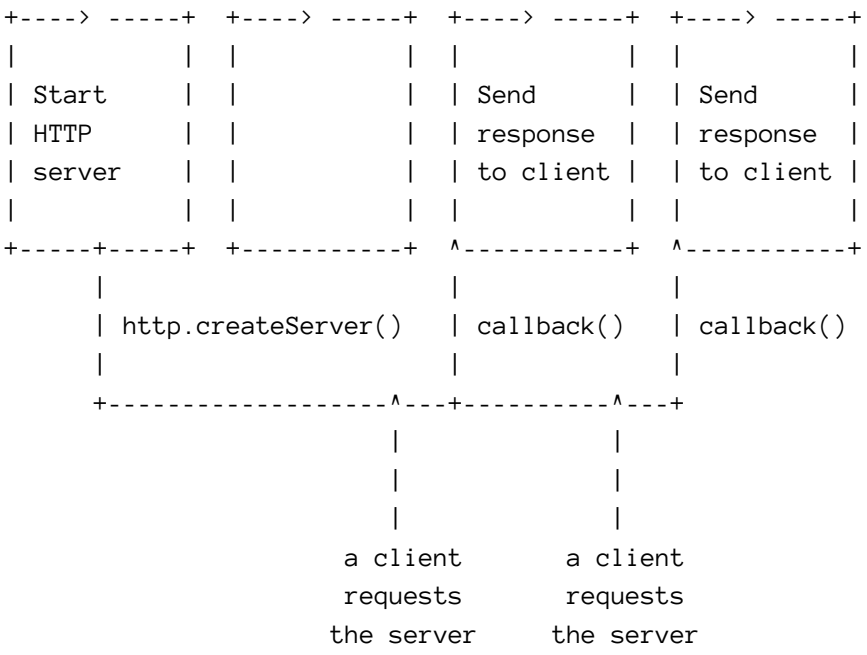


Really not that different - instead of Node.js merely counting milliseconds in the background, it starts an IO operation; IO operations are expensive, and several loop iterations go by where nothing happens. Then, Node.js has finished the background IO operation, and triggers the callback we passed in order to jump back - right into the current loop iteration. We then print - very synchronously - the *stats* object to the screen.

Another classical example: When we start an HTTP server, we create a background operation which continuously waits for HTTP requests from clients:

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

http.createServer(function(request, response) {
  response.writeHead(200, {'Content-Type': 'text/html'});
  response.write('<b>Hello World</b>');
  response.end();
}).listen(8080);
```

Whenever this event occurs, the passed callback is triggered:



Blocking and non-blocking operations

From the understanding of this conceptual model, we can get to understanding the difference between blocking and non-blocking operations.

The key to understanding these is the realization that *every* synchronous operation results in a blocking operation. That's right, even an innocent

```
console.log('Hello World');
```

results in a blocking operation - while the Node.js process writes the text to the screen, this is the *only* thing that it does. There is only one single piece of JavaScript code that can be executed within the event loop at any given time.

The reason this doesn't result in a problem in the case of a `console.log()` is that it is an extremely cheap operation. In comparison, even the most simple IO operations are way more expensive. Whenever the network or harddrives (yes, including SSDs) are involved, expect things to be *incredibly much slower* compared to operations where only the CPU and RAM are involved, like `var a = 2 * 21`. Just how much slower? The following table gives you a good idea - it shows actual times for different kind of computer operations, and shows how they relate to each other compared to how time spans that human beings experience relate to each other:

```
1  1 CPU cycle                       0.3 ns     1 s
2  Level 1 cache access              0.9 ns     3 s
3  Level 2 cache access              2.8 ns     9 s
4  Level 3 cache access             12.9 ns    43 s
5  Main memory access                120 ns     6 min
6  Solid-state disk I/O          50-150 µs   2-6 days
7  Rotational disk I/O             1-10 ms  1-12 months
8  Internet: SF to NYC               40 ms     4 years
9  Internet: SF to UK                81 ms     8 years
10 Internet: SF to Australia        183 ms    19 years
11 OS virtualization reboot           4 s     423 years
12 SCSI command time-out             30 s    3000 years
13 Hardware virtualization reboot    40 s    4000 years
14 Physical system reboot             5 m      32 millenia
```

So, the difference between setting a variable in your code and reading even a tiny file from disk is like the difference between preparing a sandwich and going on vacation for a week. You can prepare a lot of sandwichs during one week of vacation.

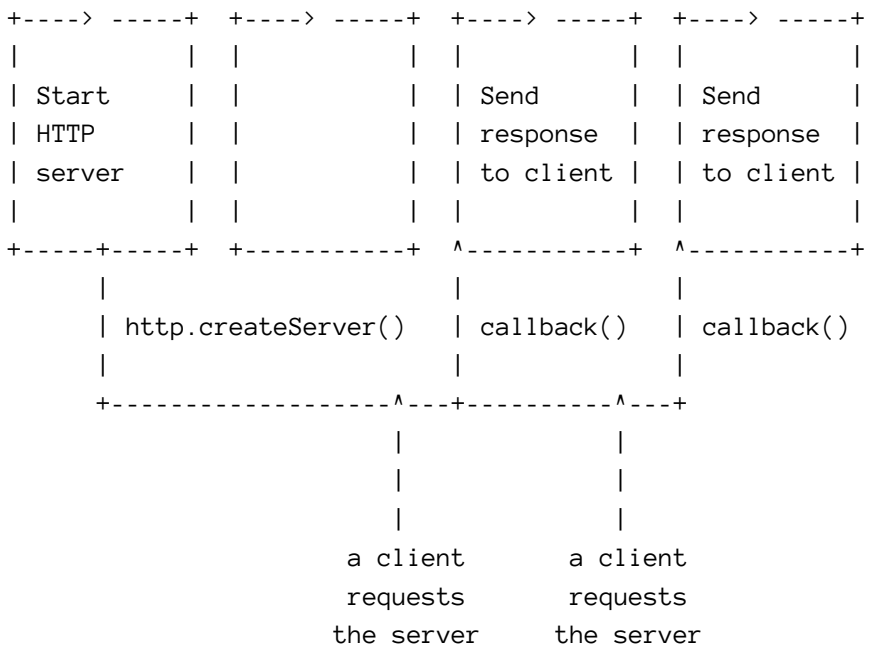
And that's the sole reason why all those Node.js functions that result in IO operations also happen to work asynchronously: it's because the event loop needs being kept free of any long-running operations during which the Node.js process would practically stall, which would result in unresponsive applications.

Just to be clear: we can easily stall the event loop even if no IO operations are involved and we only use cheap synchronous functions - if only there are enough of them within one loop iteration. Take the following code for example:

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

http.createServer(function(request, response) {
  console.log('Handling HTTP request');
  response.writeHead(200, {'Content-Type': 'text/html'});
  response.write('<b>Hello World</b>');
  response.end();
}).listen(8080);

var a;
for (var i=0; i < 10000000000; i += 1) {
  a = i;
}

console.log('For loop has finished');
```

It's our minimalistic web server again, plus a for loop with 10,000,000,000 iterations.

Our event loop visualization basically looks the same:



But here is the problem: The HTTP server will start listening in the background as an asynchronous operation, but we will stay within the first loop operation as long as the *for* loop is running. And although we only have a very cheap operation happening within the for loop, it happens 10,000,000,000 times, and on my machine, this takes around 20 seconds.

When you start the server application and then open your browser at http://localhost:8080/, you won't get an answer right away. External events, like our HTTP request, are only handled between one loop iteration and the next; however, the first loop iteration takes 20 seconds because of our *for* loop, and only then Node.js switches to the next iteration and has a chance to handle our request by calling the HTTP server callback.

As you will see, the application will print *For loop has finished*, and right after that, it will answer the HTTP request and print *Handling HTTP request.* This demonstrates how external events from asynchronous operations are handled at the beginning of a new event loop iteration.

You've probably heard time and again how Node.js isn't suited for writing applications with CPU intensive tasks - as we can see, the reason for this is the event loop model.

From this, we can distill the two most important rules for writing reponsive Node.js applications:

- Handle IO-intensive operations through asynchronous operations
- Keep your own code (that is, everything that happens synchronously within event loop iterations) as lean as possible

This leaves the question: what are sensible solutions if you *have* to do expensive CPU-bound operations within your JavaScript code? As we will learn in later chapters, we can mitigate the problem that Node.js itself simply isn't particularly suited for these kinds of operations.

Using and creating Event Emitters

Introduction

By now, you are probably more than familiar with this:

```
someFunction(function(err) {
  if (!err) {
    console.log('Hey, looks like someFunction has finished and called me.');
  } else {
    console.log('Oh no, something terrible happened!');
  }
});
```

We call a function, *someFunction* in this case, which does something asynchronously in the background, and calls the anonymous function we passed in (the *callback*) once it has finished, passing an *Error* object if something went wrong, or *null* if all went fine. That's the standard callback pattern, and you will encounter it regularly when writing Node.js code.

For simple use cases, where we call a function that finishes its task some time in the future, either successfully or with an error, this pattern is just fine.

For more complex use cases, this pattern does not scale well. There are cases where we call a function which results in multiple events happening over time, and also, different types of events might happen.

One example is reading the contents of a file through a *ReadStream*. The details of handling files are discussed in a later chapter, but we will use this example to illustrate event emitters.

This snippet demonstrates how to read data from a large file using a *ReadStream*:

```
'use strict';

var fs = require('fs');

fs.createReadStream('/path/to/large/file');
```

When reading data from a file, two different things can happen: We either receive content, or we reach the end of the file. Both cases *could* be handled using the callback pattern - for example, by using one callback with two parameters, one with the data and another one that is *false* as long as

the end of the file has not been reached, and *true* once it has been reached. Or we could provide two separate callback functions, one that is called when content is retrieved and one that is called when the end of the file has been reached.

But there is a more elegant way. Instead of working with classical callbacks, *createReadStream* allows us to use an *Event Emitter*. That's a special object which can be used to attach callback functions to different events. Using it looks like this:

```
1  'use strict';
2
3  var fs = require('fs');
4
5  var stream = fs.createReadStream('/path/to/large/file');
6
7  stream.on('data', function(data) {
8    console.log('Received data: ' + data);
9  });
10
11 stream.on('end', function() {
12   console.log('End of file has been reached');
13 });
```

Here is what happens in detail:

- On line 5, we create a read stream that will start to retrieve the contents of file */path/to/large/-file*. The call to *fs.createReadStream* does not take a function argument to use it as a callback. Instead, it returns an object, which we assign as the value of the variable *stream*.
- On line 7, we attach a callback to one type of events the *ReadStream* emits: *data* events
- On line 11, we attach another callback to another type of event the ReadStream emits: the *end* event

The object that is returned by *fs.createReadStream* is an *Event Emitter*. These objects allow us to attach different callbacks to different events while keeping our code readable and sane.

A *ReadStream* retrieves the contents of a file in chunks, which is more efficient than to load the whole data of potentially huge files into memory at once in one long, blocking operation.

Because of this, the *data* event will be emitted multiple times, depending on the size of the file. The callback that is attached to this event will therefore be called multiple times.

When all content has been retrieved, the *end* event is fired once, and no other events will be fired from then on. The *end* event callback is therefore the right place to do whatever we want to do after we retrieved the complete file content. In practice, this would look like this:

```
'use strict';

var fs = require('fs');

var stream = fs.createReadStream('/path/to/large/file');

var content = '';

stream.on('data', function(data) {
  content = content + data;
});

stream.on('end', function() {
  console.log('File content has been retrieved: ' + content);
});
```

It doesn't make too much sense to efficiently read a large file's content in chunks, only to assign the whole data to a variable and therefore using the memory anyways. In a real application, we would read a file in chunks to, for example, send every chunk to a web client that is downloading the file via HTTP. We will talk about this in more detail in a later chapter.

The *Event Emitter* pattern itself is simple and clearly defined: a function returns an event emitter object, and using this object's *on* method, callbacks can be attached to events.

However, there is no strict rule regarding the events themselves: an event *type*, like *data* or *end*, is just a string, and the author of an event emitter can define any name she wants. Also, it's not defined what arguments are passed to the callbacks that are triggered through an event - the author of the event emitter should define this through some kind of documentation.

There is one recurring pattern, at least for the internal Node.js modules, and that is the *error* event: Most event emitters emit an event called "error" whenever an error occurs, and if we don't listen to this event, the event emitter will raise an exception.

You can easily test this by running the above code: as long as you don't happen to have a file at */path/to/large/file*, Node.js will bail out with this message:

```
events.js:72
        throw er; // Unhandled 'error' event
              ^
Error: ENOENT, open '/path/to/large/file'
```

But if you attach a callback to the *error* event, you can handle the error yourself:

```
'use strict';

var fs = require('fs');

var stream = fs.createReadStream('/path/to/large/file');

var content = '';

stream.on('error', function(err) {
  console.log('Sad panda: ' + err);
});

stream.on('data', function(data) {
  content = content + data;
});

stream.on('end', function() {
  console.log('File content has been retrieved: ' + content);
});
```

Instead of using *on*, we can also attach a callback to an event using *once*. Callbacks that are attached this way will be called the first time that the event occurs, but will then be removed from the list of event listeners and not be called again:

```
stream.once('data', function(data) {
  console.log('I have received the first chunk of data');
});
```

Also, it's possible to detach an attached callback manually. This only works with named callback functions:

```
var callback = function(data) {
  console.log('I have received a chunk of data: ' + data);
}

stream.on('data', callback);

stream.removeListener('data', callback);
```

And last but not least, you can remove all attached listeners for a certain event:

```
stream.removeAllListeners('data');
```

Creating your own Event Emitter object

We can create event emitters ourselves. This is even supported by Node.js by inheriting from the built-in *events.EventEmitter* class. But let's first implement a simple event emitter from scratch, because this explains the pattern in all its details.

For this, we are going to create a module whose purpose is to regularly watch for changes in the size of a file. Once implemented, it can be used like this:

```
'use strict';

watcher = new FilesizeWatcher('/path/to/file');

watcher.on('error', function(err) {
  console.log('Error watching file:', err);
});

watcher.on('grew', function(gain) {
  console.log('File grew by', gain, 'bytes');
});

watcher.on('shrank', function(loss) {
  console.log('File shrank by', loss, 'bytes');
});

watcher.stop();
```

As you can see, the module consists of a class *FilesizeWatcher* which can be instantiated with a file path and returns an event emitter. We can listen to three different events from this emitter: *error*, *grew*, and *shrank*.

Let's start by writing a spec that describes how we expect to use our event emitter. To do so, create a new project directory, and add the following *package.json*:

```
{
  "devDependencies": {
    "jasmine-node": ""
  }
}
```

Afterwards, run `npm install`.

Now create a file *FilesizeWatcherSpec.js*, with the following content:

```
'use strict';

var FilesizeWatcher = require('./FilesizeWatcher');
var exec = require('child_process').exec;

describe('FilesizeWatcher', function() {

  var watcher;

  afterEach(function() {
    watcher.stop();
  });

  it('should fire a "grew" event when the file grew in size', function(done) {

    var path = '/var/tmp/filesizewatcher.test';
    exec('rm -f ' + path + ' ; touch ' + path, function() {
      watcher = new FilesizeWatcher(path);

      watcher.on('grew', function(gain) {
        expect(gain).toBe(5);
        done();
      });

      exec('echo "test" > ' + path, function() {});

    });

  });

  it('should fire a "shrank" event when the file grew in size', function(done) {

    var path = '/var/tmp/filesizewatcher.test';
    exec('rm -f ' + path + ' ; echo "test" > ' + path, function() {
      watcher = new FilesizeWatcher(path);

      watcher.on('shrank', function(loss) {
        expect(loss).toBe(3);
        done();
      });

      exec('echo "a" > ' + path, function() {});
```

```
43
44      });
45
46    });
47
48    it('should fire "error" if path does not start with a slash', function(done) {
49
50      var path = 'var/tmp/filesizewatcher.test';
51      watcher = new FilesizeWatcher(path);
52
53      watcher.on('error', function(err) {
54        expect(err).toBe('Path does not start with a slash');
55        done();
56      });
57
58    });
59
60  });
```

Because this is just an example application, we will not create a *spec* and a *src* directory, but instead just put both the specification file and the implementation file in the top folder of our project.

Before we look at the specification itself in detail, let's discuss the *done()* call we see in each of the *it* blocks. The *done* function is a callback that is passed to the function parameter of an *it* block by Jasmine.

This pattern is used when testing asynchronous operations. Our emitter emits events asynchronously, and Jasmine cannot know by itself when events will fire. It needs our help by being told *"now the asynchronous operation I expected to occur* did *actually occur"* - and this is done by triggering the callback.

Now to the specification itself. The first expectation is that when we write "test" into our testfile, the *grew* event is fired, telling us that the file gained 5 bytes in size.

Note how we use the *exec* function from the *child_process* module to manipulate our test file through shell commands within the specification.

Next, we specify the behaviour that is expected if the monitored test file shrinks in size: the *shrank* event must fire and report how many bytes the file lost.

At last, we specify that if we ask the watcher to monitor a file path that doesn't start with a slash, an error event must be emitted.

I'm creating a very simplified version of a file size watcher here for the sake of brevity - for a realworld implementation, more sophisticated checks would make sense.

We will create two different implementations which both fulfill this specification.

First, we will create a version of the file size watcher where we manage the event listener and event emitting logic completely ourselves. This way, we experience first hand how the event emitter pattern works.

Afterwards, we will implement a second version where we make use of existing Node.js functionality in order to implement the event emitter pattern without the need to reinvent the wheel.

The following shows a possible implementation of the first version, where we take care of the event listener callbacks ourselves:

```
1  'use strict';
2
3  var fs = require('fs');
4
5  var FilesizeWatcher = function(path) {
6    var self = this;
7
8    self.callbacks = {};
9
10   if (/^\//.test(path) === false) {
11     self.callbacks['error']('Path does not start with a slash');
12     return;
13   }
14
15   fs.stat(path, function(err, stats) {
16       self.lastfilesize = stats.size;
17   });
18
19   self.interval = setInterval(
20     function() {
21       fs.stat(path, function(err, stats) {
22         if (stats.size > self.lastfilesize) {
23           self.callbacks['grew'](stats.size - self.lastfilesize);
24           self.lastfilesize = stats.size;
25         }
26         if (stats.size < self.lastfilesize) {
27           self.callbacks['shrank'](self.lastfilesize - stats.size);
28           self.lastfilesize = stats.size;
29         }
```

```
30        }, 1000);
31      });
32  };
33
34  FilesizeWatcher.prototype.on = function(eventType, callback) {
35    this.callbacks[eventType] = callback;
36  };
37
38  FilesizeWatcher.prototype.stop = function() {
39    clearInterval(this.interval);
40  };
41
42  module.exports = FilesizeWatcher;
```

Let's discuss this code.

On line 3, we load the *fs* module - we need its *stat* function to asynchronously retrieve file information.

On line 5 we start to build a constructor function for FilesizeWatcher objects. They are created by passing a path to watch as a parameter.

On line 6, we assign the object instance variable to a local *self* variable - this way we can access our instantiated object within callback functions, where *this* would point to another object.

We then create the *self.callbacks* object - we are going to use this as an associative array where we will store the callback to each event.

Next, on line 10, we check if the given path starts with a slash using a regular expression - if it doesn't, we trigger the callback associated with the *error* event.

If the check succeeds, we start an initial *stat* operation in order to store the file size of the given path - we need this base value in order to recognize future changes in file size.

The actual watch logic starts on line 19. We set up a 1-second interval where we call *stat* on every interval iteration and compare the current file size with the last known file size.

Line 22 handles the case where the file grew in size, calling the event handler callback associated with the *grew* event; line 26 handles the opposite case. In both cases, the new file size is saved.

Event handlers can be registered using the *FilesizeWatcher.on* method which is defined on line 34. In our implementation, all it does is to store the callback under the event name in our *callbacks* object.

Finally, line 38 defines the *stop* method which cancels the interval we set up in the constructor function.

Let's see if this implementation works by running `./node_modules/jasmine-node/bin/jasmine-node ./FilesizeWatcherSpec.js`:

```
..F

Failures:

  1) FilesizeWatcher should fire "error" if the path does not start with a slash
   Message:
     TypeError: Object #<Object> has no method 'error'
   Stacktrace:
     TypeError: Object #<Object> has no method 'error'
    at new FilesizeWatcher (FilesizeWatcher.js:11:28)
    at null.<anonymous> (FilesizeWatcherSpec.js:51:15)
    at null.<anonymous> (...mine-node/lib/jasmine-node/async-callback.js:45:37)

Finished in 0.087 seconds
3 tests, 3 assertions, 1 failure, 0 skipped
```

Well... nearly. The core functionality works: the *grew* and *shrank* events are fired as expected.

But the file path check makes problems. According to the message, the problem arises on line 11:

```
self.callbacks['error']('Path does not start with a slash');
```

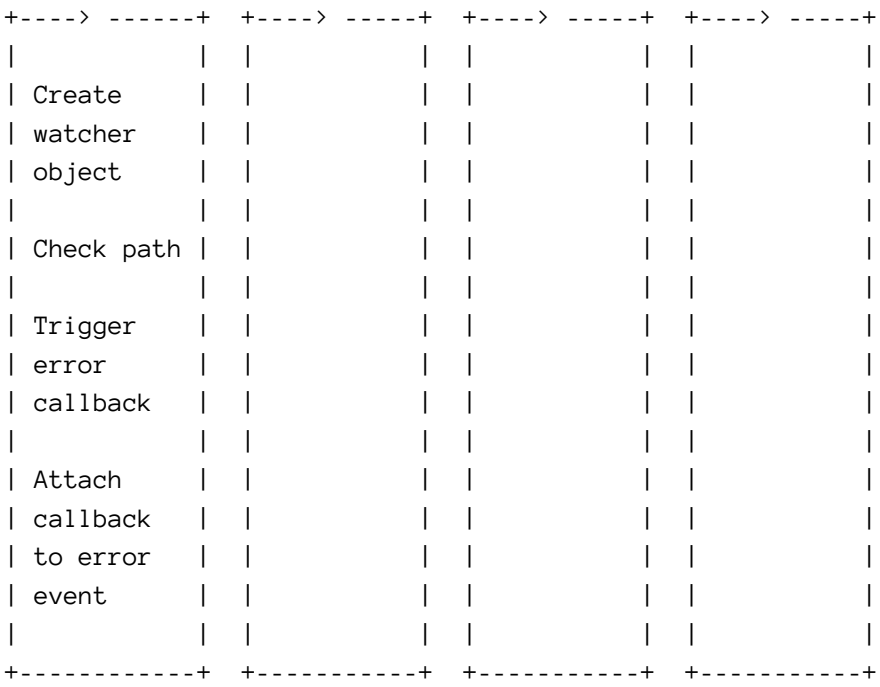
The error message says that the *self.callbacks* object doesn't have a method called *error*. But in our specification, we defined a callback for this event, just as we did for the other events, on line 53 of *FilesizeWatcherSpec.js*:

```
watcher.on('error', function(err) {
```

The problem here is quite subtle and can only be understood if one remembers how the execution model of Node.js works. Let's recall the chapter on synchronous and asynchronous operations. Our specification code looks like this:

```
watcher = new FilesizeWatcher(path);

watcher.on('error', function(err) {
  expect(err).toBe('Path does not start with a slash');
  done();
});
```

According to the visualization technique we used to explain the inner workings of the event loop model, this is what actually happens:



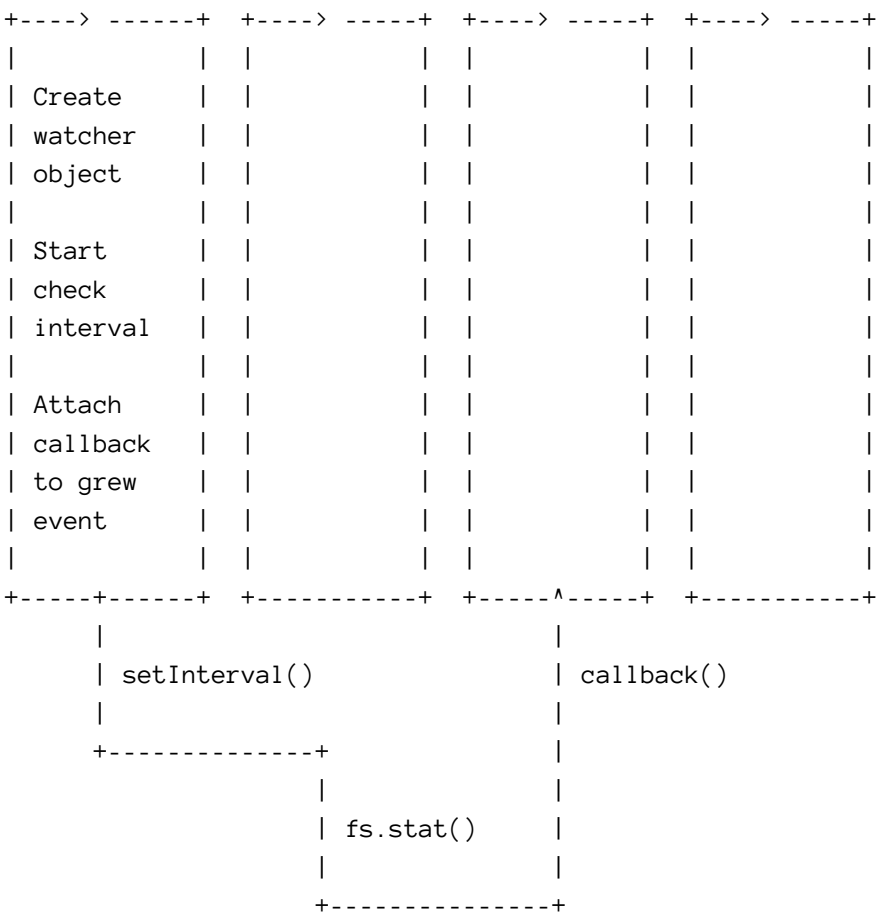
Now it becomes clear that the order in which things are happening is simply wrong: We first call the error callback, and then we attach it. This can't work.

When creating the *watcher* object, we call the *Filesizewatcher* constructor function. Up to the point where it checks the path and calls the *error* event handler callback, everything happens synchronously:

```
var FilesizeWatcher = function(path) {
  var self = this;

  self.callbacks = {};

  if (/^\//.test(path) === false) {
    self.callbacks['error']('Path does not start with a slash');
    return;
  }
```

The constructor function returns and Node.js execution continues with the main code, all the while we are still in the first event loop iteration. Only now - afterwards - we attach the event handler callback for the *error* event; but the constructor already tried to call it!

Why doesn't this problem arise with the *grew* and *shrank* events? Here is the visualization:



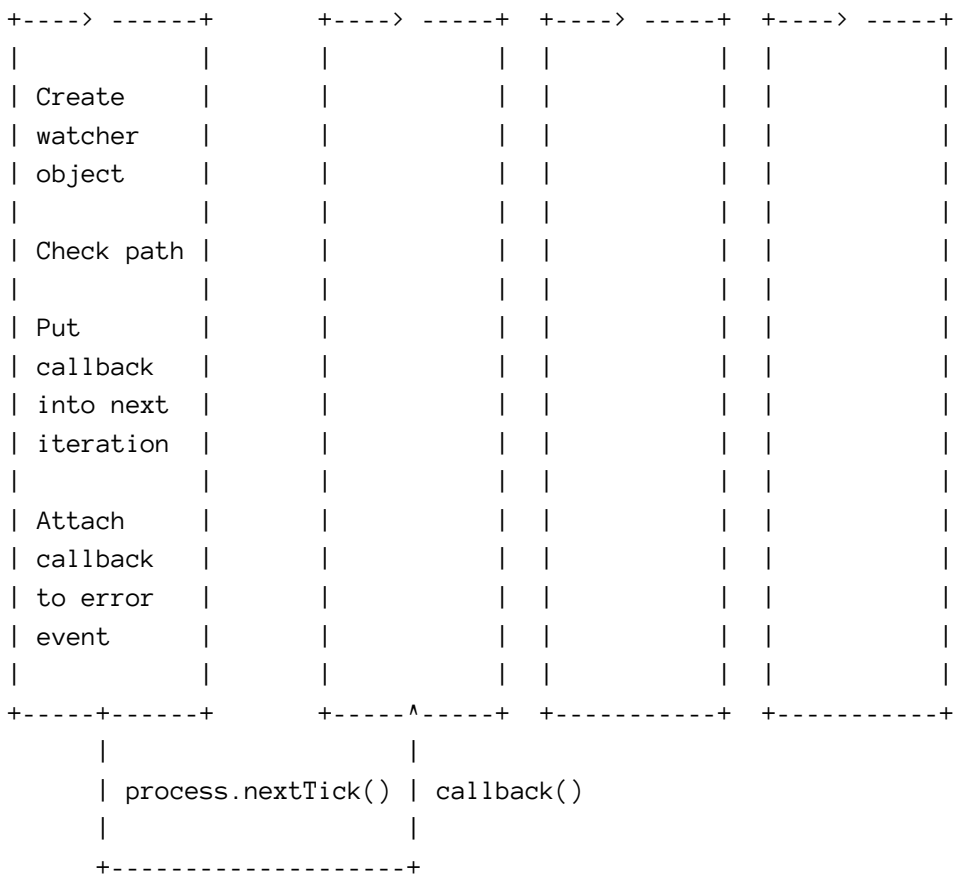
This shows how our main code in the first event loop starts the asynchronous *setInterval* operation which in turn starts an asynchronous *stat* operation. Only several event loop iterations later, when the callback for the *grew* event has long been attached, is this callback triggered. No problem arises.

How can we solve the problem for our *error* event? The solution is to put the triggering of the not-yet-attached callback into the future, into the next event loop iteration; this way, the callback will be triggered in the second event loop iteration *after* it has been attached in the first event loop iteration.

Moving code execution into the next event loop iteration is simple thanks to the *process.nextTick* method. Here is how the relevant part in file *FilesizeWatcher.js* needs to be rewritten:

```
if (/^\//.test(path) === false) {
  process.nextTick(function() {
    self.callbacks['error']('Path does not start with a slash');
  });
  return;
}
```

With this, we put the callback triggering for the *error* event listener into a callback function that will be executed by *process.nextTick* on the next iteration of the event loop:



Thus we ensure that it is called *after* it has been defined.

As mentioned, Node.js supports us in creating event emitters, sparing us from writing the callback handling code for our own emitters.

We can use this support by inheriting from an existing class *EventEmitter* in module *events*.

Because the expected behaviour of our *MyEmitter* doesn't change from an outside perspective, we will leave the spec unchanged, and only change our implementation from using a self-made solution to utilizing the existing Node.js class:

```
1  'use strict';
2
3  var fs = require('fs');
4  var util = require('util');
5  var EventEmitter = require('events').EventEmitter;
6
7  var FilesizeWatcher = function(path) {
8    var self = this;
9
```

```
10    if (/^\//.test(path) === false) {
11      process.nextTick(function() {
12        self.emit('error', 'Path does not start with a slash');
13      });
14      return;
15    }
16
17    fs.stat(path, function(err, stats) {
18        self.lastfilesize = stats.size;
19    });
20
21    self.interval = setInterval(
22      function() {
23        fs.stat(path, function(err, stats) {
24          if (stats.size > self.lastfilesize) {
25            self.emit('grew', stats.size - self.lastfilesize);
26            self.lastfilesize = stats.size;
27          }
28          if (stats.size < self.lastfilesize) {
29            self.emit('shrank', self.lastfilesize - stats.size);
30            self.lastfilesize = stats.size;
31          }
32        }, 1000);
33      });
34  };
35
36  util.inherits(FilesizeWatcher, EventEmitter);
37
38  FilesizeWatcher.prototype.stop = function() {
39    clearInterval(this.interval);
40  };
41
42  module.exports = FilesizeWatcher;
```

The interesting part here is line 36. We use a helper function that ships with Node.js, *util.inherits*. We use it to make our *FilesizeWatcher* class a descendant of *events.EventEmitter*, which also ships with Node.js. This class implements all the logic that is needed to act as an event emitter, and our *FilesizeWatcher* class inherits this logic. This way, our own code now concentrates on implementing its specific logic, without having to care about the details of registering callbacks and checking if these can be called etc.

According to our tests, *FilesizeWatcher* still behaves as it did before:

```
...

Finished in 0.052 seconds
3 tests, 3 assertions, 0 failures, 0 skipped
```

Instead of using *util.inherits*, we can also create the inheritance by placing a new *events.EventEmitter* object on the prototype of *FilesizeWatcher*:

```
FilesizeWatcher.prototype = new EventEmitter();
```

Although *util.inherits* adds some extra properties to our constructor, the result is the same, behaviour-wise. Give it a try if you like and see if the tests still pass.

Summary

The asynchronous development style using callbacks that is a recurring pattern in JavaScript and Node.js applications can at times result in a code structure that is hard to understand. Applying the event emitter pattern comes in handy in situations where more than one callback event must be handled, resulting in a cleaner code structure that is easier to follow and reason about. The pattern is so useful that we should use it for our own objects and functions where it makes sense, something that's made easy by utilizing the existing *util.inherits* class.

Optimizing code performance and control flow management using the *async* library

Writing or using functions or methods in your code that are executed one after the other gets you a long way in your applications.

Sometimes, those functions are simple synchronous steps:

```
console.log('Starting calculation...');
var result = 5 + 4;
console.log('The result is', result);
```

Often, callbacks are used when operations are executed in the background and jump back into our code's control flow asynchronously at a later point in time:

```
console.log('Starting calculation...');
startExpensiveCalculation(5, 4, function(err, result) {
  if (!err) {
    console.log('The result is', result);
  }
});
```

If those asynchronous background operations bring forth a more complex callback behaviour, they might be implemented as an event emitter:

```
console.log('Starting calculation...');

calculation = Calculator.start(5, 4);

calculation.on('error', function(err) {
  console.log('An error occured:', err);
});

calculation.on('interim result', function(result) {
  console.log('Received interim result:', result);
```

```
});

calculation.on('done', function(result) {
  console.log('Received final result:', result);
});
```

Handling expensive operations asynchronously in the background, especially if they are IO-bound, is an important key to making Node.js applications perform efficiently - reading a large file or writing a lot of records to a database will always be a costly procedure, but handling it asynchronously at least ensures that the other parts of our application won't be blocked while that procedure is going on.

Nevertheless, there is often potential for optimization within our own code and its control flow.

Executing expensive asynchronous background tasks in parallel

Let's consider an example where our application queries several different remote web services, presenting the retrieved data on the console.

We are not going to query real remote web services, instead we will write a very simple Node.js HTTP server that will serve as a dummy web service. Our web server doesn't really do anything significant, and therefore we will make it react to requests a bit slower than neccessary, in order to simulate a real web service that has a certain workload - as you will see, this makes it easier for us to show the performance gain in our own code optimizations.

Please create a new project folder and create a file *server.js* with the following content:

```
1  'use strict';
2
3  var http = require('http');
4  var url = require('url');
5  var querystring = require('querystring');
6
7  http.createServer(function(request, response) {
8
9    var pathname = url.parse(request.url).pathname;
10   var query = url.parse(request.url).query;
11   var id = querystring.parse(query)['id'];
12
13   var result = {
14     'pathname': pathname,
```

```
15      'id': id,
16      'value': Math.floor(Math.random() * 100)
17    };
18
19    setTimeout(function() {
20      response.writeHead(200, {"Content-Type": "application/json"});
21      response.end(JSON.stringify(result));
22    }, 2000 + Math.floor(Math.random() * 1000));
23
24  }).listen(
25    8080,
26    function() {
27      console.log('Echo Server listening on port 8080');
28    }
29  );
```

This gives us a very simple "echo" server - if we request the URL http://localhost:8080/getUser?id=4, we receive `{"pathname":"/getUser","id":"4","value":67}` as the response. This is good enough to give us the simulation of a remote webservice API to play around with.

But alas, it's a slow webservice! Someone didn't do his optimization homework, and we now have to deal with an API where every response takes between 2 to 3 seconds (this is simulated with the *setTimeout* construct on lines 19-22).

This allows to show how different request patterns will result in different runtime characteristics.

We will now write a Node.js client for this webserver. This client will make two consecutive requests to the server, and print the output for both requests on the command line:

```
1  'use strict';
2
3  var request = require('request');
4
5  request.get(
6    'http://localhost:8080/getUserName?id=1234',
7    function(err, res, body) {
8      var result = JSON.parse(body);
9      var name = result.value;
10
11     request.get(
12       'http://localhost:8080/getUserStatus?id=1234',
13       function(err, res, body) {
14         var result = JSON.parse(body);
15         var status = result.value;
```

```
16
17          console.log('The status of user', name, 'is', status);
18    });
19
20  });
```

Save this in a file called *client.js*. Also, you need to install the *request* module via `npm install request`.

This is probably the most straight-forward solution. We start the first request, wait for it to finish, print the result, then start the second request, wait for it to finish, and print that result, too.

How long does that take? Let's see:

```
$ time node client.js
The status of user 62 is 68

real  0m5.810s
user  0m0.172s
sys   0m0.033s
```

Don't forget to start the server via `node server.js` beforehand!

Not surprisingly, it takes around 5-6 seconds, because we only start the second request after the first request has been completed, and each request takes around 2-3 seconds.

We can't do anything about the terribly slow remote webservice, but our own code isn't exactly optimal either. Our two requests don't inherently depend on each other, and yet, we are executing them serially.

Of course starting these requests in parallel is simple, because both are asynchronous operations:

```
'use strict';

var request = require('request');

var name, status;

request.get(
  'http://localhost:8080/getUserName?id=1234',
  function(err, res, body) {
    var result = JSON.parse(body);
    name = result.value;
});

request.get(
  'http://localhost:8080/getUserStatus?id=1234',
  function(err, res, body) {
    var result = JSON.parse(body);
    status = result.value;
});

console.log('The status of user', name, 'is', status);
```

No, wait, sorry! That's not going to work - *console.log* will execute within the first event loop iteration while the request callbacks are triggered in later iterations. Mh, how about...

```
'use strict';

var request = require('request');

var name, status;

request.get(
  'http://localhost:8080/getUserName?id=1234',
  function(err, res, body) {
    var result = JSON.parse(body);
    name = result.value;
});

request.get(
  'http://localhost:8080/getUserStatus?id=1234',
  function(err, res, body) {
    var result = JSON.parse(body);
```

```
    status = result.value;

    console.log('The status of user', name, 'is', status);
});
```

No, that's not good either: We *start* both request in parallel, but we have no guarantee that they will finish at the same time. We risk printing

```
The status of user undefined is 75
```

if the second request finishes earlier than the first. Well, looks like we need some additional code to synchronize our finished calls. How about this:

```
'use strict';

var request = require('request');

var name, status;
var firstHasFinished, secondHasFinished = false;

request.get(
  'http://localhost:8080/getUserName?id=1234',
  function(err, res, body) {
    var result = JSON.parse(body);
    name = result.value;
    markFinished('first');
});

request.get(
  'http://localhost:8080/getUserStatus?id=1234',
  function(err, res, body) {
    var result = JSON.parse(body);
    status = result.value;
    markFinished('second');
});

function markFinished(step) {
  if (step == 'first') {
    firstHasFinished = true;
  }

  if (step == 'second') {
```

```
    secondHasFinished = true;
  }

  if (firstHasFinished && secondHasFinished) {
    console.log('The status of user', name, 'is', status);
  }
}
```

Seriously now - that's not even funny! What if we need to synchronize dozens or hundreds of operations? We could use an array where we store the state of each operation... no, this whole thing doesn't feel right.

async to the rescue, I say!

async is a clever little module that makes managing complex control flows in our code a breeze.

After installing the module via `npm install async`, we can write our client like this:

```
'use strict';

var request = require('request');
var async = require('async');

var name, status;

var getUsername = function(callback) {
  request.get(
    'http://localhost:8080/getUserName?id=1234',
    function(err, res, body) {
      var result = JSON.parse(body);
      callback(err, result.value);
    });
};

var getUserStatus = function(callback) {
  request.get(
    'http://localhost:8080/getUserStatus?id=1234',
    function (err, res, body) {
      var result = JSON.parse(body);
      callback(err, result.value);
    });
};

async.parallel([getUsername, getUserStatus], function(err, results) {
```

```
27   console.log('The status of user', results[0], 'is', results[1]);
28 });
```

Let's analyze what we are doing here.

On line 4, we load the async library. We then wrap our requests into named functions. These functions will be called with a *callback* parameter. Inside our functions, we trigger this callback when our operation has finished - in this case, when the requests have been answered.

We call the callbacks with two parameters: an *error* object (which is *null* if no errors occured), and the result value.

The orchestration happens on lines 26-28. We use the *parallel* method of the *async* object and pass an array of all the functions we want to run in parallel. Additionally, we pass a callback function which expect two parameters, *err* and *results*.

async.parallel will trigger this callback as soon as the slowest of the parallel operations has finished (and called its callback), or as soon as one of the operations triggers its callback with an error.

Let's see what this does to the total runtime of our script:

```
$ time node client.js
The status of user 95 is 54

real    0m3.176s
user    0m0.240s
sys     0m0.044s
```

As one would expect, the total runtime of our own code matches the runtime of one request because both requests are started in parallel and will finish roughly at the same time.

Optimizing code structure with *async*

async offers several other mechanisms for managing the control flow of our code. These are interesting even if our concern isn't performance optimization. Let's investigate them.

For these cases, let's remove the artificial slowness from the API server by removing the *setTimeout* operation on line 19 and 22, making the server respond immediately:

```
'use strict';

var http = require('http');
var url = require('url');
var querystring = require('querystring');

http.createServer(function(request, response) {

  var pathname = url.parse(request.url).pathname;
  var query = url.parse(request.url).query;
  var id = querystring.parse(query)['id'];

  var result = {
    'pathname': pathname,
    'id': id,
    'value': Math.floor(Math.random() * 100)
  };

  response.writeHead(200, {"Content-Type": "application/json"});
  response.end(JSON.stringify(result));

}).listen(
  8080,
  function() {
    console.log('Echo Server listening on port 8080');
  }
);
```

Sometimes we want to run operations in series. This is of course possible by putting method calls into the callback functions of previous method calls, but the code quickly becomes ugly if you do this with a lot of methods:

```
'use strict';

var request = require('request');

var url = 'http://localhost:8080/';

request.get(url + 'getUserName?id=1234', function(err, res, body) {
  console.log('Name:', JSON.parse(body).value);

  request.get(url + 'getUserStatus?id=1234', function(err, res, body) {
```

```
    console.log('Status:', JSON.parse(body).value);

    request.get(url + 'getUserCountry?id=1234', function(err, res, body) {
      console.log('Country:', JSON.parse(body).value);

      request.get(url + 'getUserAge?id=1234', function(err, res, body) {
        console.log('Age:', JSON.parse(body).value);
      });

    });

  });

});
```

This is already starting to look messy, and we haven't even added any notable "business logic" to our code.

Note how our code is intended another level with every method call, creating the so-called "boomerang pattern" that is typical for multi-level nested callback control flows.

We can use *async.series* to achieve the same control flow with much cleaner code:

```
'use strict';

var request = require('request');
var async = require('async');

var url = 'http://localhost:8080/';

async.series([

  function(callback) {
    request.get(url + 'getUserName?id=1234', function(err, res, body) {
      console.log('Name:', JSON.parse(body).value);
      callback(null);
    });
  },

  function(callback) {
    request.get(url + 'getUserStatus?id=1234', function(err, res, body) {
      console.log('Status:', JSON.parse(body).value);
      callback(null);
```

```
    });
  },

  function(callback) {
    request.get(url + 'getUserCountry?id=1234', function(err, res, body) {
      console.log('Country:', JSON.parse(body).value);
      callback(null);
    });
  },

  function(callback) {
    request.get(url + 'getUserAge?id=1234', function(err, res, body) {
      console.log('Age:', JSON.parse(body).value);
      callback(null);
    });
  }

]);
```

Just as with *async.parallel*, we can use *async.series* to collect the results of each step and do something with them once all steps have finished. This is again achieved by passing the result of each step to the callback each step triggers, and by providing a callback function to the *async.series* call which will receive an array of all results:

```
'use strict';

var request = require('request');
var async = require('async');

var url = 'http://localhost:8080/';

async.series([

  function(callback) {
    request.get(url + 'getUserName?id=1234', function(err, res, body) {
      callback(null, 'Name: ' + JSON.parse(body).value);
    });
  },

  function(callback) {
    request.get(url + 'getUserStatus?id=1234', function(err, res, body) {
      callback(null, 'Status: ' + JSON.parse(body).value);
```

```
    });
  },

  function(callback) {
    request.get(url + 'getUserCountry?id=1234', function(err, res, body) {
      callback(null, 'Country: ' + JSON.parse(body).value);
    });
  },

  function(callback) {
    request.get(url + 'getUserAge?id=1234', function(err, res, body) {
      callback(null, 'Age: ' + JSON.parse(body).value);
    });
  }

],

  function(err, results) {
    for (var i=0; i < results.length; i++) {
      console.log(results[i]);
    }
  }

);
```

In case that one of the series steps passes a non-null value to its callback as the first parameter, the series is immediately stopped, and the final callback is triggered with the results that have been collected to far, and the *err* parameter set to the error value passed by the failing step.

async.waterfall is similar to *async.series*, as it executes all steps in series, but it also enables us to access the results of a previous step in the step that follows:

```
'use strict';

var request = require('request');
var async = require('async');

var url = 'http://localhost:8080/';

async.waterfall([

    function(callback) {
      request.get(url + 'getSessionId', function(err, res, body) {
```

```
12          callback(null, JSON.parse(body).value);
13        });
14      },
15
16      function(sId, callback) {
17        request.get(url + 'getUserId?sessionId=' + sId, function(err, res, body) {
18          callback(null, sId, JSON.parse(body).value);
19        });
20      },
21
22      function(sId, uId, callback) {
23        request.get(url + 'getUserName?userId=' + uId, function(err, res, body) {
24          callback(null, JSON.parse(body).value, sId);
25        });
26      }
27
28    ],
29
30    function(err, name, sId) {
31      console.log('Name:', name);
32      console.log('SessionID:', sId);
33    }
34
35  );
```

Note how for each step function, *callback* is passed as the last argument. It follows a list of arguments for each parameter that is passed by the previous function, minus the error argument which each step function always passes as the first parameter to the callback function.

Also note the difference in the final callback: instead of *results*, it too expects a list of result values, passed by the last waterfall step.

async provides several other interesting methods which help us to bring order in our control flow and allows us to orchestrate tasks in an efficient manner. Check out the async README[1] for more details.

[1] https://github.com/caolan/async/blob/master/README.md

Node.js and MySQL

Using the node-mysql library

Node.js is able to connect to MySQL database servers, but support for this is not built in. We need to make use of a Node.js package that handles the low level details of establishing a connection and talking to the database. Additionally, such a package should enable us to issue statements to the database written in SQL.

The most widely accepted package for this is *node-mysql* by Node.js core contributor Felix Geisendörfer.

In order to use it, let's start by declaring *mysql* as a dependency for our project in our *package.json* file:

```
{
  "dependencies": {
    "mysql": "~2.0.0"
  }
}
```

As always, *npm install* pulls in everything we need to start working with a MySQL database.

For what follows, I assume that you have a working MySQL database up and running on your local machine. You should be able to connect to it locally. You can verify that this is the case by running

```
mysql -h127.0.0.1 -uroot -p
```

on your command line. Provide the database root password if needed - you will also need this password in your Node.js code in order to establish a connection.

A first database application

Once you are up and running, let's start with a first, simple database application. Create a file *index.js* in the directory where you installed the *node-mysql* module, and fill it with the following code:

```
1  'use strict';
2
3  var mysql = require('mysql');
4
5  var connection = mysql.createConnection({
6    host: 'localhost',
7    user: 'root',
8    password: 'root'
9  });
10
11 connection.query(
12   'SELECT "foo" AS first_field, "bar" AS second_field',
13   function (err, results, fields) {
14     console.log(results);
15     connection.end();
16   }
17 );
```

Let's see what exactly happens here. First, we include Felix' external *mysql* library (line 3). We then create a new object dubbed *connection* on line 5.

This object is a representation of our connection with the database. No connection without connecting, which is why we call the *createConnection* function with the parameters necessary to connect and authenticate against the database server: the host name, the username, and the password.

Note that there is a parameter *port* which defaults to 3306 if omitted. You only need to set this parameter if your database server does not run on this default port.

For a list of all possible parameters (a total of 17 are available), see the *node-mysql* documentation at https://github.com/felixge/node-mysql#connection-options.

Through the newly created connection object, we can now talk to the database server, which is what we do on line 11. To do so, the *connection* object provides a method named *query*.

We provide two parameters to the method call. The first is an SQL statement, the second is an anonymous function. This function gets called when Node.js receives the answer to our query from the server - a *callback*.

This is once again the classical asynchronous pattern of JavaScript applications that makes sense whenever our code triggers an operation whose execution could potentially block the program flow if executed synchronously: Imagine the database needing 5 seconds before it has compiled the answer to our query - that's not unlikely for more complex queries. If we wouldn't wait for the database's answer asynchronously, then our whole application would have to wait for it. No other code could be executed for these 5 seconds, rendering our whole application unusable during this time.

Line 12 shows which parameters the *query* method passes to our callback function: *err*, *results*, and *fields*.

For now, let's concentrate on *results*, because that's where the fun is.

What kind of thing is *results*? It's simply an array of objects, with each object representing one row of the result set and each attribute of such an object representing one field of the according row.

Let's run our application and examine its output to see how this looks. Please run `node index.js`. The result should look like this:

```
[ { first_field: 'foo', second_field: 'bar' } ]
```

Let's add a legend:

```
|-------------- An array ----------------------|
|                                               |
| |------------ with an object -------------| |
| |                                         | |
| |    |------- that has an attribute       | |
| |    |                                    | |
| |    |        with a value                | |
| |    |           |                        | |
| | -----------   ---                        | |
[ { first_field: 'foo', second_field: 'bar' } ]
| | -----------   ---                        | |
| |    |           |                        | |
| |    |        with a value                | |
| |    |                                    | |
| |    |------- that has a field            | |
| |                                         | |
| |------------ with a row -----------------| |
|                                               |
|-------------- A result set -----------------|
```

(Note that we didn't query for any real data from any real database table, but relational databases are really stubborn and treat everything as a set of rows of fields, even if we only query for some text strings mapped to made up fields like we just did.)

According to this structure we can of course access the value of any field in any row directly, by accessing the right attribute of the right object:

```
console.log(results[0].first_field); // Outputs 'foo';
```

Let's examine the other parameters that are passed to our anonymous callback function.

err is either an error object, if a problem occured, or *null* if everything went well. Let's modify our code a bit to see how this works:

```
1  'use strict';
2
3  var mysql = require('mysql');
4
5  var connection = mysql.createConnection({
6    host: 'localhost',
7    user: 'root',
8    password: 'root'
9  });
10
11 connection.query(
12   'SLECT "foo" AS first_field, "bar" AS second_field',
13   function (err, results, fields) {
14     console.log(err);
15     console.log(results);
16     connection.end();
17   }
18 );
```

Note how I changed the SQL statement in line 12 and added logging of the *err* parameter on line 14. The altered SQL statement is invalid and results in the database answering the query with an error. *node-mysql* translates this error into an error object which is then passed as the first parameter to our callback function.

If you run the new code, a string representation of the error object is written to the console:

```
{ [Error: ER_PARSE_ERROR: You have an error in your SQL syntax; check
   the manual that corresponds to your MySQL server version for the
   right syntax to use near 'SLECT "foo" AS first_field, "bar" AS
   second_field' at line 1]
  code: 'ER_PARSE_ERROR',
  errno: 1064,
  sqlState: '42000',
  index: 0 }
```

The fact that the *err* parameter is *null* if no error occured makes error handling in our code very simple:

```
'use strict';

var mysql = require('mysql');

var connection = mysql.createConnection({
  host: 'localhost',
  user: 'root',
  password: 'root'
});

connection.query(
  'SLECT "foo" AS first_field, "bar" AS second_field',
  function (err, results, fields) {
    if (err) {
      console.log('A database error occured!');
    } else {
      console.log(results);
    }
    connection.end();
  }
);
```

That's a very typical pattern in a lot of Node.js applications: Call a library function while providing a callback, and check if the first parameter passed to this callback is *null* in order to decide if an error occured, continuing otherwise.

fields returns an array with one object for every field that was requested in the query - each object contains meta information about the according field. It isn't of any significant use for us here. Feel free to *console.log()* the *fields* parameter if you like.

Using node-mysql's Streaming API

Next, we will query some real data from existing tables. For this, we first need to create a database, add a table to it, and insert some rows. Let's do this through a Node.js application. Please create a file *create.js* with the following content:

```
'use strict';

var mysql = require('mysql');

var connection = mysql.createConnection({
  host: 'localhost',
  user: 'root',
  password: 'root'
});

connection.query('CREATE DATABASE node', function(err) {
  if (err) {
    console.log('Could not create database "node".');
  }
});

connection.query('USE node', function(err) {
  if (err) {
    console.log('Could not switch to database "node".');
  }
});

connection.query('CREATE TABLE test ' +
                 '(id INT(11) AUTO_INCREMENT, ' +
                 ' content VARCHAR(255), ' +
                 ' PRIMARY KEY(id))',
  function(err) {
    if (err) {
      console.log('Could not create table "test".');
    }
  }
);

connection.query('INSERT INTO test (content) VALUES ("Hello")');
connection.query('INSERT INTO test (content) VALUES ("World")');

connection.end();
```

This program is meant to be executed just once in order to set up a new database with some example data. Please run it now.

Let's rewrite our *index.js* file and try to query for this newly added data:

```
1  'use strict';
2
3  var mysql = require('mysql');
4
5  var connection = mysql.createConnection({
6    host: 'localhost',
7    user: 'root',
8    password: 'root',
9    database: 'node'
10 });
11
12 connection.query(
13   'SELECT id, content FROM test',
14   function (err, results, fields) {
15     if (err) {
16       console.log('A database error occured!');
17     } else {
18       console.log(results);
19     }
20     connection.end();
21   }
22 );
```

Please note how on line 9 I have added an additional connection parameter named *database*. This way, a default database is already selected for subsequent queries on this connection, and doesn't have to be selected through a *USE database* query.

If we execute the script, the result should look like this:

```
[ { id: 1, content: 'Hello' }, { id: 2, content: 'World' } ]
```

As one would expect, we now receive an array with two objects, because the resultset consists of two rows.

This gives us the opportunity to discuss an even better way to handle resultsets from queries.

I have explained how the classical callback pattern should be used on the *query* call because otherwise expensive database operations could block the whole Node.js application.

However, there is another source of potentially blocking behaviour: large resultsets.

Imagine that our test database consists of 1,000,000 rows instead of just two, and we would like to output the content of the whole resultset to the console like this:

```
connection.query(
  'SELECT id, content FROM test',
  function (err, results, fields) {
    for (var i = 0; i < results.length; i++) {
      console.log('Content of id ' + results[i].id +
                            ' is ' + results[i].content);
    }
  }
);
```

Using the callback pattern ensures that our Node.js application won't block while the database retrieves these 1,000,000 rows. However, once we receive the resultset with these rows in our application - that is, within the callback - and we start operating on them, executing these operations also will take a long time.

During this time, our application will block, too. Remember that in Node.js, only one piece of our JavaScript code is executed at any given time. If we iterate over these 1,000,000 array entries and do something with each entry, and this takes, say, 10 seconds, then our application cannot do something else during these 10 seconds.

The solution: instead of using the all-or-nothing callback pattern, we can utilize *node-mysql*'s streaming API.

Instead of providing a callback function to the *query* call, we only provide the SQL statement, but we use the return value of the *query* function: a *Query* object.

This object emits several events that happen during the lifetime of the query: *error*, *field*, *row* and *end*.

Let's concentrate on *row* and see how a more efficient version of our 1,000,000 row output application would look like:

```
var query = connection.query('SELECT id, content FROM test');

query.on('row', function(row) {
  console.log('Content of id ' + row.id + ' is ' + row.content);
});
```

Now we have a piece of our code running 1,000,000 times, once for each row. While this doesn't make our application faster in the narrower sense (it will still take the assumed 10 seconds until all rows have been handled), it does make it way more efficient. Our application now only blocks during the handling of one row, immediately returning control to the event loop of our application, allowing to execute other code in between.

Of course, this isn't critical for an application that is used by only one person on one shell console. But it is *very* critical if such a database operation takes place within a multi-user client-server application:

Imagine you use a website to retrieve a small amount of data from a database - you wouldn't want to wait 10 seconds for your page to load only because another user requested a page that retrieved 1,000,000 rows for him from the database.

Let's rewrite our *index.js* script once again to demonstrate usage of the complete streaming API of *node-mysql*:

```
1  'use strict';
2
3  var mysql = require('mysql');
4
5  var connection = mysql.createConnection({
6    host: 'localhost',
7    user: 'root',
8    password: 'root',
9    database: 'node'
10 });
11
12 var query = connection.query('SELECT id, content FROM test');
13
14 query.on('error', function(err) {
15   console.log('A database error occured:');
16   console.log(err);
17 });
18
19 query.on('fields', function(fields) {
20   console.log('Received fields information.');
21 });
22
23 query.on('result', function(result) {
24   console.log('Received result:');
25   console.log(result);
26 });
27
28 query.on('end', function() {
29   console.log('Query execution has finished.');
30   connection.end();
31 });
```

It's pretty much self explanatory. Note how the order of the *on* blocks doesn't say anything about when (and if) they are executed - it's up to *node-mysql* to decide when each block (or, to be more precise: the anonymous callback function associated with each block) will be called.

When executing this script, this is what the output should look like:

```
Received fields information.
Received result:
{ id: 1, content: 'Hello' }
Received result:
{ id: 2, content: 'World' }
Query execution has finished.
```

Making SQL queries secure against attacks

Let's now go full circle and create a simple web application that allows to insert data into our table and also reads and displays the data that was entered.

We need to start a web server with two routes (one for displaying data, one for taking user input), and we need to pass user input to the database and database results to the webpage. Here is the application in one go:

```
1  'use strict';
2
3  var mysql       = require('mysql'),
4      http        = require('http'),
5      url         = require('url'),
6      querystring = require('querystring');
7
8
9  // Start a web server on port 8888. Requests go to function handleRequest
10
11 http.createServer(handleRequest).listen(8888);
12
13
14 // Function that handles http requests
15
16 function handleRequest(request, response) {
17
18   // Page HTML as one big string, with placeholder "DBCONTENT" for data from
19   // the database
20   var pageContent = '<html>' +
21                     '<head>' +
22                     '<meta http-equiv="Content-Type" ' +
23                     'content="text/html; charset=UTF-8" />' +
24                     '</head>' +
25                     '<body>' +
26                     '<form action="/add" method="post">' +
```

```
27                    '<input type="text" name="content">' +
28                    '<input type="submit" value="Add content" />' +
29                    '</form>' +
30                    '<div>' +
31                    '<strong>Content in database:</strong>' +
32                    '<pre>' +
33                    'DBCONTENT' +
34                    '</pre>' +
35                    '</div>' +
36                    '<form action="/" method="get">' +
37                    '<input type="text" name="q">' +
38                    '<input type="submit" value="Filter content" />' +
39                    '</form>' +
40                    '</body>' +
41                    '</html>';
42
43  // Parsing the requested URL path in order to distinguish between
44  // the / page and the /add route
45  var pathname = url.parse(request.url).pathname;
46
47  // User wants to add content to the database (POST request to /add)
48  if (pathname == '/add') {
49    var requestBody = '';
50    var postParameters;
51    request.on('data', function (data) {
52      requestBody += data;
53    });
54    request.on('end', function() {
55      postParameters = querystring.parse(requestBody);
56      // The content to be added is in POST parameter "content"
57      addContentToDatabase(postParameters.content, function() {
58        // Redirect back to homepage when the database has finished adding
59        // the new content to the database
60        response.writeHead(302, {'Location': '/'});
61        response.end();
62      });
63    });
64
65  // User wants to read data from the database (GET request to /)
66  } else {
67    // The text to use for filtering is in GET parameter "q"
68    var filter = querystring.parse(url.parse(request.url).query).q;
```

```
69      getContentsFromDatabase(filter, function(contents) {
70        response.writeHead(200, {'Content-Type': 'text/html'});
71        // Poor man's templating system: Replace "DBCONTENT" in page HTML with
72        // the actual content we received from the database
73        response.write(pageContent.replace('DBCONTENT', contents));
74        response.end();
75      });
76    }
77  }
78
79
80  // Function that is called by the code that handles the / route and
81  // retrieves contents from the database, applying a LIKE filter if one
82  // was supplied
83
84  function getContentsFromDatabase(filter, callback) {
85    var connection = mysql.createConnection({
86      host: 'localhost',
87      user: 'root',
88      password: 'root',
89      database: 'node'
90    });
91    var query;
92    var resultsAsString = '';
93
94    if (filter) {
95      query = connection.query('SELECT id, content FROM test ' +
96                               'WHERE content LIKE "' + filter + '%"');
97    } else {
98      query = connection.query('SELECT id, content FROM test');
99    }
100
101   query.on('error', function(err) {
102     console.log('A database error occured:');
103     console.log(err);
104   });
105
106   // With every result, build the string that is later replaced into
107   // the HTML of the homepage
108   query.on('result', function(result) {
109     resultsAsString += 'id: ' + result.id;
110     resultsAsString += ', content: ' + result.content;
```

```
111     resultsAsString += '\n';
112   });
113
114   // When we have worked through all results, we call the callback
115   // with our completed string
116   query.on('end', function(result) {
117     connection.end();
118     callback(resultsAsString);
119   });
120 }
121
122
123 // Function that is called by the code that handles the /add route
124 // and inserts the supplied string as a new content entry
125
126 function addContentToDatabase(content, callback) {
127   var connection = mysql.createConnection({
128     host: 'localhost',
129     user: 'root',
130     password: 'root',
131     database: 'node',
132   });
133
134   connection.query('INSERT INTO test (content) ' +
135                    'VALUES ("' + content + '")',
136     function(err) {
137       if (err) {
138         console.log('Could not insert content "' + content +
139                     '" into database.');
140       }
141       callback();
142     });
143 }
```

Simply start the application by running `node index.js`, open up its homepage in your browser by visiting http://localhost:8888/, and play around with it.

So, that's quite a bit of code here. Let's walk through the important parts. If the user requests the homepage, here is what happens:

- The HTTP server that is started on line 11 receives the requests and calls the *handleRequest* function that is declared on line 16.

- On line 45 the variable *pathname* is declared and initialized with the requested path, which is / in this case, by parsing the request URL.
- The code continues on line 66, where the request for / is handled.
- The handling starts by parsing the URL for the value of the GET parameter *q* which is later used to filter the database content.
- On line 69 things start to get interesting. We call *getContentsFromDatabase* with two parameters: The *filter* string we just parsed from the URL, and an anonymous callback function.
- Execution continues on line 84, starting with the creation of a new database connection.
- Then, on lines 94 to 99, the query to retrieve the contents from the database is started.
- Each retrieved row triggers the execution of the *result* event handler on line 108, where we extend the string *resultsAsString* with the contents of each row.
- Execution jumps to line 116 once all rows have been retrieved. Here, in the *end* event handler, we close the connection to the database and then call the callback we passed into the *getContentsFromDatabase* function. This returns control back to the request handler, in line 70.
- There, we start responding to the request, first by sending a status 200 header, followed by the page content - with the database content replaced into the HTML at the proper place on line 74.

With this, the request has been handled and responded to.

When the user uses the *Filter content* form, the same procedure is executed, with the only difference that now the GET parameter *q* has a value which is then used to filter the database results using the SQL *LIKE* clause.

When using the *Add content* button, a different execution path is taken:

- The HTTP server that is started on line 11 receives the requests and calls the *handleRequest* function that is declared on line 16.
- On line 45 the variable *pathname* is declared and initialized with the requested path, which is */add* in this case, by parsing the request URL.
- The code continues on line 48, where the request for */add* is handled.
- Then, on line 51, the body of the request (the POST data) is retrieved step-by-step, in the same way that large resultsets from a database can be retrieved when using the streaming API. With every step, a variable is extended which finally holds the complete request body.
- Once the request has been fully received, execution continues on line 54 when the *end* event is triggered for the request.
- We then parse the request body in order to translate the POST data into a JavaScript object which allows us to easily access the value of POST parameters, as is done one line 57 where we call the *addContentToDatabase* function with the string that is to be added to the database as its first parameter, followed by a callback function that we expect to be triggered once the data has been inserted.

- With this, execution continues on line 126 where we start by connecting to the database, followed by an *INSERT* query on line 134. The query function triggers our anonymous callback defined on line 135 when the query execution has finished, which in turn triggers the callback that was passed in from the request handler function.
- This leads to execution continuing on line 60, where we redirect the user's browser back to the homepage.

So, nice application, does what it's supposed to do. And has a huge problem.

Why? Because the way we treat the incoming data that is provided by the user puts our database at risk. Our application is vulnerable to so-called SQL injection attacks. Let's demonstrate how.

To do so, let's create another table within our database *node*, one that's supposed to carry sensitive information that's not meant for any user's eyes. We will just pretend that our nice little content-viewer app is part of a bigger application that somewhere also stores user passwords.

We will call this table *passwords*, and you need to create this within the *node* database on your local server, either using the MySQL command line, a MySQL GUI client you happen to use, or, if you like, you can simply rewrite the *create.js* script we used earlier, like this:

```
1  'use strict';
2
3  var mysql = require('mysql');
4
5  var connection = mysql.createConnection({
6    host: 'localhost',
7    user: 'root',
8    password: 'root'
9  });
10
11 connection.query('USE node', function(err) {
12   if (err) {
13     console.log('Could not switch to database "node".');
14   }
15 });
16
17 connection.query('CREATE TABLE passwords ' +
18                  '(id INT(11) AUTO_INCREMENT, ' +
19                  ' password VARCHAR(255), ' +
20                  ' PRIMARY KEY(id))',
21   function(err) {
22     if (err) {
23       console.log('Could not create table "passwords".');
24     }
```

```
25    }
26  );
27
28  connection.query('INSERT INTO passwords (password) VALUES ("secret")');
29  connection.query('INSERT INTO passwords (password) VALUES ("dont_tell")');
30
31  connection.end();
```

Execute *create.js* through Node.js in order to create the new table.

Now, here is the scenario: We have a table *test* that contains public information, and a table *passwords* that contains sensitive information. We have written an application that operates only on the *test* table, thus allowing public access to the public information in that table, but not allowing access to any sensitive information like the passwords in the *passwords* table.

And yet, the passwords can be retrieved through the application. Here's how to do that:

- Start the application with *node index.js*
- Visit http://localhost:8888 in your browser
- Enter the following text into the second text input field and hit *Filter content*

```
%" UNION SELECT id, password AS content FROM passwords WHERE password LIKE "
```

Note that you need to enter the text *exactly* as shown above, starting with a percentage sign and ending with a quotation mark.

You will receive a list of data from the test table (if any content has been added before), but at the end of this list, you will also see the two passwords we have added to the *passwords* table, like this:

```
id: 1, content: foo bar
id: 1, content: secret
id: 2, content: dont_tell
```

Whoops. How is that possible? It's because a set of SQL commands has been injected into our application's SQL, although only data - the string to filter for - was meant to be injected. Hence an SQL injection attack.

If you look at the critical piece of our code, what happens becomes clear. On lines 95 and 96, our SQL query is constructed:

```
query = connection.query('SELECT id, content FROM test ' +
                         'WHERE content LIKE "' + filter + '%"');
```

If the content of the *filter* variable is "foo", this is what the final SQL looks like:

```
SELECT id, content FROM test WHERE content LIKE "foo%"
```

But if the malicious filter text is entered into the web page, we end up with the following SQL query:

```
SELECT id, content FROM test WHERE content LIKE "%" UNION
  SELECT id, password AS content FROM passwords WHERE password LIKE "%"
```

(I've added a line break in order to stay within the boundaries of the text page.)

The malicious filter text simply closes the LIKE query of the original SQL and continues with an additional SELECT on the *passwords* table, combining the results of both SELECTs into a single (and identically structured) result set using the UNION clause. Another LIKE clause is added at the end of the malicious code, which makes sure that the whole SQL is syntactically correct.

This is a made up example for demo purposes, but that's how actual attacks look like. A real attack might need a lot more effort - e.g., the attacker needs to find out the structure of the passwords table, but supporting information can be gathered by querying meta data from the database; if an attacker finds a part of the application that allows to extend harmless SQL queries into more potent ones, finding sensitive information is only a question of time and patience.

What can we, as application developers, do against such attacks? There is no 100% secure system, as the saying goes, however, there are a lot of ways to raise the bar, making attacks more difficult.

In our scenario, a very effective counter-measure is to use placeholders with value assignment when building SQL statements, instead of hand-crafting the final query by concatenating strings.

This way, we can guarantee that any part of your query that is supposed to be treated as a value is indeed interpreted by *node-mysql* as a value, escaping it in a way that doesn't allow the value to end up at the database server as a command.

To achieve this, we need a way to create a query that can unambiguously differentiate between the command parts of a query and the value parts.

This is done by creating a query where the value parts are not directly included within the query string - instead, their position is marked with a special character, the quotation mark (?), as a placeholder.

This is how our query looks when using this syntax:

```
SELECT id, content FROM test WHERE content LIKE ?
```

Now *node-mysql* knows that whatever is going to be used at the position marked by the *?* placeholder is a value, treating it accordingly.

Of course, when issueing the query, we also need a way to tell what value to use. This is done by mapping the value onto the query-with-placeholder, like so:

```
filter = filter + '%';
query = connection.query('SELECT id, content FROM test WHERE content LIKE ?',
                         filter);
```

In our specific case, we need to add the search wildcard, *%*, to the filter parameter. We then construct the query with the placeholder, passing the value for this placeholder as the second parameter. The rest of the code remains unchanged.

We should change the INSERT query that is used for adding content, too, by rewriting line 135 to

```
connection.query('INSERT INTO test (content) VALUES (?)', content,
  function(err) {
  ...
```

This changes the SQL query to placeholder syntax, and adds the *content* variable as the second parameter. The callback function is now the third parameter.

Mapping multiple values onto multiple placeholders is done by providing an array of values:

```
query('SELECT foo FROM bar WHERE id = ? AND name = ?', [42, 'John Doe']);
```

Here, "42" maps to the *id* placeholder, and "John Doe" to the *name* placeholder.

If you restart the application ad try to attack it again, you will see that the attack no longer works. Mission accomplished.

Summary

In this chapter we learned about the *node-mysql* module, how it enables us to connect to relational databases, how using asynchronous callbacks and the Streaming API enable us to write efficient applications even when large resultsets need to be handled, and analyzed how database applications can be attacked and, more importantly, how we can protect our applications against these kinds of attacks.

Node.js and MongoDB

Some MongoDB basics

MongoDB is a document-oriented NoSQL database that stores objects in the following hierarchy:

```
Server
      \_Database
                \_Collection
                            \_Document
                            \_Document
                            \_Document
                            \_ ...
                \_Collection
                            \_Document
                            \_Document
                            \_Document
                            \_ ...
```

If you are coming from a relational SQL database background (I do), then it might help to think of collections as tables, and of documents as rows in that table. It's not a perfect analogy, however; the key idea of a document-oriented database like MongoDB is that documents within the same collection do not need to have the same structure, while each row in a table of a relational database does have the same structure.

Here is how actual data in this hierarchy might actually look like:

```
localhost
     \_accounting
          \_customers
               \_ { _id: 53c6c2, name: 'Jane Doe' }
               \_ { _id: 9dc42e, name: 'John Doe', age: 24 }
               \_ { _id: 63a76d, name: 'Jim Doe', options: { billtime: 3 } }
          \_invoices
               \_ { _id: 98ef5a, value: 192.87 }
               \_ { _id: f4eb21, value: 192.87, recurring: true }
```

This example represents the case where one has a MongoDB server running on localhost, with two databases *customers* and *invoices*. Each database contains some documents that are similar, but not identical, in structure.

As you can see, I have represented the document objects in JavaScript object notation - this is how we are going to encounter them when working with a MongoDB server from within Node.js.

Every document has an *_id* attribute, which is a random 24-digit hex value - I have shortened them for the illustration.

We will now examine how to create and work with such a structure.

Applying CRUD operations with the low-level mongodb driver

Again, I won't cover installation of the server software. Please refer to the official installation instructions[2] in order to get a MongoDB server up and running on your machine.

Once this is done, create a new project folder called *mongodb-test*, and in there create an initial *package.json* as follows:

```
{
  "name": "mongodb-test",
  "version": "0.0.1",
  "description": "",
  "dependencies": {
    "mongodb": "^1.4.7"
  },
  "devDependencies": {}
}
```

As always, we pull in the dependencies with

```
npm install
```

Now we can create a first program in file *index.js*:

[2] http://docs.mongodb.org/manual/installation/

```
1  'use strict';
2
3  var MongoClient = require('mongodb').MongoClient;
4
5  MongoClient.connect(
6      'mongodb://127.0.0.1:27017/accounting',
7      function(err, connection) {
8        var collection = connection.collection('customers');
9
10       collection.insert({'name': 'Jane Doe'}, function(err, count) {
11
12         collection.find().toArray(function(err, documents) {
13           console.dir(documents);
14           connection.close();
15         });
16
17       });
18
19     });
```

Let's see what this code does. On line 3, we require the *MongoClient* class from the *mongodb* library. We use the *connect* method of this class on line 5 to establish a connection to our MongoDB server, which is listening on *localhost*, port *27017*. Upon connecting, we declare that we want to use the *accounting* database.

We then create an object *collection* on line 8 which represents the *customers* collection within the *accounting* database.

Using the *insert* method of the collection object, we add a new document to the *customers* collection on line 10.

When the insert operation has finished, we query the collection for all existing documents using the *find* method on line 12. Our callback receives an array with the documents, prints them to the screen, and then closes the database connection.

Running this code by executing

```
node index.js
```

will result in output similar to

```
[ { _id: 53c8ba1b517c86d3c9df71e6, name: 'Jane Doe' } ]
```

and running it again will result in output similar to

```
[ { _id: 53c8ba1b517c86d3c9df71e6, name: 'Jane Doe' },
  { _id: 53c8ba21d13eb8dbc96e2c19, name: 'Jane Doe' } ]
```

As you can see, each run creates a new document in the database. The *name* attribute is set by our code, the *_id* is generated by MongoDB.

Let's rewrite our code a bit in order to update existing documents:

```
1  'use strict';
2
3  var MongoClient = require('mongodb').MongoClient;
4
5  MongoClient.connect(
6    'mongodb://127.0.0.1:27017/accounting',
7    function(err, connection) {
8      var collection = connection.collection('customers');
9
10     collection.update({}, {'$set': {'age': 24}}, function(err, count) {
11
12       console.log('Updated', count, 'documents');
13
14       collection.find().toArray(function(err, documents) {
15         console.dir(documents);
16         connection.close();
17       });
18
19     });
20
21   });
```

The change is on line 10. Instead of inserting a new document, we want to update our documents. The first parameter is the filter that defines which documents to match when applying the update. In this case, we pass an empty object, that is, we don't want to match any specific document and instead we want to apply the update to all documents in the collection.

The second parameter defines what to update within each matched document. In this case, we want to set the *age* attribute of the documents to *24*. The documents we already stored do not have an *age* attribute yet, but that's not a problem: because MongoDB works schema-less, it will happily add attributes that did not yet exist on a given document.

This is how the content of our collection looks like after running the new code:

```
Updated 1 documents
[ { _id: 53c8ba1b517c86d3c9df71e6, name: 'Jane Doe' },
  { _id: 53c8ba21d13eb8dbc96e2c19, name: 'Jane Doe', age: 24 } ]
```

Oh, well. Not quite what we expected - we didn't define a filter to the *update* statement because we wanted to match all existing documents, and yet only one of our two documents has been updated with the *age* attribute.

The reason is that this is the default behaviour for MongoDB updates: only the first match for a given filter - even if it matches every document - is updated.

By changing line 10 from

```
collection.update({}, {'$set': {'age': 24}}, function(err, count) {
```

to

```
collection.update(
  {},
  {'$set': {'age': 24}},
  {'multi': true},
  function(err, count) {
```

(that is, by adding a parameter `{'multi': true}`), we override this behaviour, which then results in MongoDB updating all matching documents.

Let's have a look at how actual filters that match only some of our documents look like. We are going to rewrite our script because we need some more documents to play with:

```
1  'use strict';
2
3  var MongoClient = require('mongodb').MongoClient;
4
5  MongoClient.connect(
6    'mongodb://127.0.0.1:27017/accounting',
7    function (err, connection) {
8      var collection = connection.collection('customers');
9
10     var doFind = function (callback) {
11       collection.find().toArray(function (err, documents) {
12         console.dir(documents);
13         callback();
14       });
```

```
15      };
16
17      var doInsert = function (i) {
18        if (i < 20) {
19          var value = Math.floor(Math.random() * 10);
20          collection.insert(
21            {'n': '#' + i, 'v': value},
22            function (err, count) {
23              doInsert(i + 1);
24            });
25        } else {
26          console.log();
27          console.log('Inserted', i, 'documents:');
28          doFind(function () {
29            doUpdate();
30          });
31        }
32      };
33
34      var doUpdate = function () {
35        collection.update(
36          {'v': {'$gt': 5}},
37          {'$set': {'valuable': true}},
38          {'multi': true},
39          function (err, count) {
40            console.log();
41            console.log('Updated', count, 'documents:');
42            doFind(function () {
43              collection.remove({}, function () {
44                connection.close();
45              });
46            });
47          });
48      };
49
50      doInsert(0);
51
52    });
```

As you can see, we created three functions, *doFind*, *doInsert* and *doUpdate*. The script starts by calling *doInsert*, which is a recursive function that inserts 20 documents into the collection, and does so in a serial fashion by recursively calling itself (on line 23) every time the previous insert operation

has finished. Each documents has an *n* attribute with the serial number of its insert operation, and a *v* value, which is a random number between 0 and 9.

As soon as the limit of 20 inserts has been reached, the function calls *doFind* (on line 28) which prints all existing documents to the screen. We pass *doFind* a callback function which calls *doUpdate*. This function adds the field *valuable* to some, but not all, documents.

Whether a document is updated or not depends on it's *v* value: using the *$gt* operator (on line 36), we filter the documents that are affected by the update to those whose value is *greater than* 5.

After the update operation has finished, *doFind* is called once again, printing the results of the update to the screen. We then delete all documents using *remove* (on line 43), again using a filter that matches any document - however, *remove* doesn't need a *multi* parameter in order to work on all matched documents; it does so by default.

Here is a typical result for a run of this script:

```
Inserted 20 documents:
[ { _id: 53d60cc676371927029f95bd, n: '#0', v: 5 },
  { _id: 53d60cc676371927029f95be, n: '#1', v: 8 },
  { _id: 53d60cc676371927029f95bf, n: '#2', v: 0 },
  { _id: 53d60cc676371927029f95c0, n: '#3', v: 4 },
  { _id: 53d60cc676371927029f95c1, n: '#4', v: 6 },
  { _id: 53d60cc676371927029f95c2, n: '#5', v: 2 },
  { _id: 53d60cc676371927029f95c3, n: '#6', v: 4 },
  { _id: 53d60cc676371927029f95c4, n: '#7', v: 1 },
  { _id: 53d60cc676371927029f95c5, n: '#8', v: 0 },
  { _id: 53d60cc676371927029f95c6, n: '#9', v: 7 },
  { _id: 53d60cc676371927029f95c7, n: '#10', v: 5 },
  { _id: 53d60cc676371927029f95c8, n: '#11', v: 8 },
  { _id: 53d60cc676371927029f95c9, n: '#12', v: 2 },
  { _id: 53d60cc676371927029f95ca, n: '#13', v: 8 },
  { _id: 53d60cc676371927029f95cb, n: '#14', v: 1 },
  { _id: 53d60cc676371927029f95cc, n: '#15', v: 0 },
  { _id: 53d60cc676371927029f95cd, n: '#16', v: 1 },
  { _id: 53d60cc676371927029f95ce, n: '#17', v: 4 },
  { _id: 53d60cc676371927029f95cf, n: '#18', v: 4 },
  { _id: 53d60cc676371927029f95d0, n: '#19', v: 6 } ]

Updated 6 documents:
[ { _id: 53d60cc676371927029f95bd, n: '#0', v: 5 },
  { _id: 53d60cc676371927029f95bf, n: '#2', v: 0 },
  { _id: 53d60cc676371927029f95c0, n: '#3', v: 4 },
  { _id: 53d60cc676371927029f95c2, n: '#5', v: 2 },
  { _id: 53d60cc676371927029f95c3, n: '#6', v: 4 },
```

```
{ _id: 53d60cc676371927029f95c4, n: '#7', v: 1 },
{ _id: 53d60cc676371927029f95c5, n: '#8', v: 0 },
{ _id: 53d60cc676371927029f95c7, n: '#10', v: 5 },
{ _id: 53d60cc676371927029f95c9, n: '#12', v: 2 },
{ _id: 53d60cc676371927029f95cb, n: '#14', v: 1 },
{ _id: 53d60cc676371927029f95cc, n: '#15', v: 0 },
{ _id: 53d60cc676371927029f95cd, n: '#16', v: 1 },
{ _id: 53d60cc676371927029f95ce, n: '#17', v: 4 },
{ _id: 53d60cc676371927029f95cf, n: '#18', v: 4 },
{ _id: 53d60cc676371927029f95be, n: '#1', v: 8, valuable: true },
{ _id: 53d60cc676371927029f95c1, n: '#4', v: 6, valuable: true },
{ _id: 53d60cc676371927029f95c6, n: '#9', v: 7, valuable: true },
{ _id: 53d60cc676371927029f95c8, n: '#11', v: 8, valuable: true },
{ _id: 53d60cc676371927029f95ca, n: '#13', v: 8, valuable: true },
{ _id: 53d60cc676371927029f95d0, n: '#19', v: 6, valuable: true } ]
```

As you can see, 20 documents with a random *v* value are inserted. Afterwards, all documents whose value is greater than 5 have been updated to include the attribute *valuable* set to *true* - a total of 6 documents in this case.

Retrieving specific documents using filters

We have not yet performed any truly interesting queries against our collection. Up until now, all we did was to retrieve all documents contained in our collection by performing *collection.find()* without any filter. More complex filters are available of course.

Let's rewrite the existing code in order to retrieve only specific documents in our *doFind* function. First, a very simple example - we are going to retrieve all documents where the *v* attribute has a value of *5*. To do so, we need to change line 11 from

```
collection.find().toArray(function (err, documents) {
```

to

```
collection.find({'v': 5}).toArray(function (err, documents) {
```

which results in only the matching documents being printed:

```
Inserted 20 documents:
[ { _id: 53d7698d99c6107303ad204c, n: '#8', v: 5 },
  { _id: 53d7698d99c6107303ad2050, n: '#12', v: 5 } ]

Updated 10 documents:
[ { _id: 53d7698d99c6107303ad204c, n: '#8', v: 5 },
  { _id: 53d7698d99c6107303ad2050, n: '#12', v: 5 } ]
```

We can write filters that match more than just one attribute, like this:

```
collection.find(
  {
    'v': 6,
    'valuable': true
  }
).toArray(function (err, documents) {
```

Both attributes of each document in the collection must match the given values for a document to be included. In other words, this is an AND query.

For the given collection, the above query doesn't make too much sense, because after the update, every document with a *v* value of 6 has the *valuable* attribute set to *true* anyways. But we might want to filter for all *valuable* documents whose *v* value is less than 8, like so:

```
collection.find(
  {
    'v': { '$lt': 8 },
    'valuable': true
  }
).toArray(function (err, documents) {
```

We already encountered the *greater than* operator *$gt*, and *$lt* is the opposite operator, *less than*. Other available operators are:

- `$lte`: less than or equal
- `$gte`: greater than or equal
- `$ne`: not equal

Filters can also be combined into an OR query, if, for example, we want to query for all documents whose *v* value is either 5 or 8:

```
collection.find(
  {
    '$or': [
      {'v': 5},
      {'v': 8}
    ]
  }
).toArray(function (err, documents) {
```

And of course it is also possible to combine an AND query and an OR query:

```
collection.find(
  {
    'v': {'$gt': 3},
    '$or': [
      {'n': '#5'},
      {'n': '#10'}
    ]
  }
).toArray(function (err, documents) {
```

This will retrieve all documents that have a *v* value greater than *3* AND whose *n* value is *#5* OR *#10*.

Speaking of the *n* attribute: it holds a string value, which cannot be queried with operators like *$gt* or *$lt*, of course. We can, however, apply regular expressions:

```
collection.find(
  {
    'n': /^#1/
  }
).toArray(function (err, documents) {
```

This matches all documents whose *n* value starts with *#1*.

All the filter variations discussed so far can also be applied to the *update* and *remove* methods of the *collection* object. This way, you can update or remove only specific documents in a given collection.

The *find* method also takes an *options* parameter. Possible options are *limit*, *skip*, and *sort*:

```
collection.find(
  {
    'n': /^#1/
  },
  {
    'limit': 5,
    'skip' : 2,
    'sort' : 'v'
  }
).toArray(function (err, documents) {
```

This one retrieves five documents whose *n* value starts with *#1*, skips the first two matches, and sorts the result set by the *v* value:

```
Inserted 20 documents:
[ { _id: 53d7739bbe04f81c05c13b6f, n: '#19', v: 5 },
  { _id: 53d7739bbe04f81c05c13b66, n: '#10', v: 5 },
  { _id: 53d7739bbe04f81c05c13b6b, n: '#15', v: 6 },
  { _id: 53d7739bbe04f81c05c13b67, n: '#11', v: 7 },
  { _id: 53d7739bbe04f81c05c13b68, n: '#12', v: 7 } ]

Updated 9 documents:
[ { _id: 53d7739bbe04f81c05c13b6f, n: '#19', v: 5 },
  { _id: 53d7739bbe04f81c05c13b66, n: '#10', v: 5 },
  { _id: 53d7739bbe04f81c05c13b6b, n: '#15', v: 6, valuable: true },
  { _id: 53d7739bbe04f81c05c13b68, n: '#12', v: 7, valuable: true },
  { _id: 53d7739bbe04f81c05c13b67, n: '#11', v: 7, valuable: true } ]
```

We can also sort over multiple fields and in different directions:

```
collection.find(
  {
    'n': /^#1/
  },
  {
    'limit': 5,
    'skip' : 2,
    'sort' : [ ['v', 'asc'], ['n', 'desc'] ]
  }).toArray(function (err, documents) {
```

This sorts the matching documents first by their *v* value in ascending order, and documents with an identical *v* value are then ordered by their *n* value, in descending order, as can be seen here with the documents whose *v* value is *1*:

```
Inserted 20 documents:
[ { _id: 53d77475c0a6e83b0556d817, n: '#14', v: 1 },
  { _id: 53d77475c0a6e83b0556d815, n: '#12', v: 1 },
  { _id: 53d77475c0a6e83b0556d818, n: '#15', v: 3 },
  { _id: 53d77475c0a6e83b0556d81a, n: '#17', v: 4 },
  { _id: 53d77475c0a6e83b0556d814, n: '#11', v: 6 } ]

Updated 7 documents:
[ { _id: 53d77475c0a6e83b0556d817, n: '#14', v: 1 },
  { _id: 53d77475c0a6e83b0556d815, n: '#12', v: 1 },
  { _id: 53d77475c0a6e83b0556d818, n: '#15', v: 3 },
  { _id: 53d77475c0a6e83b0556d81a, n: '#17', v: 4 },
  { _id: 53d77475c0a6e83b0556d814, n: '#11', v: 6, valuable: true } ]
```

More complex update operations

We already learned how to add attributes or change the value of attributes in existing documents:

```
collection.update(
  {'n': /^#1/},
  {'$set': {'v': 5} },
  {'multi': true},
  function (err, count) {
    // ...
  }
```

This would set the value of the *v* attribute to *5* for all documents whose *n* value starts with *#1*.

This is probably the most regularly used update operation. Some other ways to update documents are available, however. Numeric attributes can be increased, for example:

```
collection.update(
  {'n': /^#1/},
  {'$inc': {'v': +1} },
  {'multi': true},
  function (err, count) {
    // ...
  }
```

There is no *$dec* operation, but of course, we can increase by *-1*:

```
collection.update(
  {'n': /^#1/},
  {'$inc': {'v': -1} },
  {'multi': true},
  function (err, count) {
    // ...
  }
```

Multiplication is possible, too:

```
collection.update(
  {'n': /^#1/},
  {'$mul': {'v': 2} },
  {'multi': true},
  function (err, count) {
    // ...
  }
```

There are several other operators:

- $rename Renames a field.
- $unset Removes the specified field from a document.
- $min Only updates the field if the specified value is less than the existing field value.
- $max Only updates the field if the specified value is greater than the existing field value.
- $currentDate Sets the value of a field to current date, either as a Date or a Timestamp.

Please refer to the MongoDB documentation[3] for a detailed discussion of each operation.

The *update* method can be used to perform an operation where documents are updated if the filter being used is able to match existing documents, but a new document is inserts instead if no existing document could be matched. This is called an *upsert*:

[3] http://docs.mongodb.org/manual/reference/operator/update-field/

```
collection.update(
  {'n': '#20'},
  {'$set': {'n': '20', 'v': 1} },
  {'multi': true, 'upsert': true},
  function (err, count) {
    // ...
  }
```

In our initial collection, only entries with *n* from *#0* to *#19* exist. Therefore, an update call with a filter set to `{'n': '#20'}` cannot match any document in the collection. Because the *upsert* option is set to *true*, a new document is inserted:

```
Inserted 20 documents:
[ { _id: 53d789d4fff187d607f03b3a, n: '#0', v: 7 },
  ...
  { _id: 53d789d4fff187d607f03b4c, n: '#18', v: 8 },
  { _id: 53d789d4fff187d607f03b4d, n: '#19', v: 5 } ]

Updated 1 documents:
[ { _id: 53d789d4fff187d607f03b3a, n: '#0', v: 7 },
  ...
  { _id: 53d789d4fff187d607f03b4c, n: '#18', v: 8 },
  { _id: 53d789d4fff187d607f03b4d, n: '#19', v: 5 },
  { _id: 53d789d44266a0f5acef6f05, n: '#20', v: 1 } ]
```

Working with indexes

Like other databases, MongoDB needs to index data sets if we want queries on large sets to perform well.

In order to experience this firsthand, let's write a script which inserts 200,000 documents instead of just 20:

```
'use strict';

var MongoClient = require('mongodb').MongoClient;

MongoClient.connect(
  'mongodb://127.0.0.1:27017/accounting',
  function (err, connection) {
    var collection = connection.collection('customers');

    var doInsert = function (i) {
      if (i < 200000) {
        var value = Math.floor(Math.random() * 10);
        collection.insert(
          {'n': '#' + i, 'v': value},
          function (err, count) {
            doInsert(i + 1);
          });
      } else {
        connection.close();
      }
    };

    doInsert(0);

  });
```

Call this script *insert.js* and execute it running `node insert.js`.

Now, we time how long it takes MongoDB to return 10 specific rows from this collection, using the following script:

```
'use strict';

var MongoClient = require('mongodb').MongoClient;

MongoClient.connect(
  'mongodb://127.0.0.1:27017/accounting',
  function (err, connection) {
    var collection = connection.collection('customers');

    collection.find(
      {'v': {'$gt': 5}},
      {
```

```
13          'skip:': 100000,
14          'limit': 10,
15          'sort': 'v'
16        }
17      ).toArray(function (err, documents) {
18        console.dir(documents);
19          connection.close();
20      });
21
22    });
```

This script queries the database for 10 documents whose *v* attribute is greater than 5, starting at document 100,000, sorting results by their *v* value.

Name this script *query.js* and run it like this:

```
time node query.js
```

This way, you'll get a poor man's performance benchmark. This is the result on my machine:

```
~$ time node query.js

[ { _id: 53d7955c3ac66b9e0af26745, n: '#36271', v: 6 },
  { _id: 53d7955c3ac66b9e0af26757, n: '#36289', v: 6 },
  { _id: 53d7955c3ac66b9e0af26761, n: '#36299', v: 6 },
  { _id: 53d7955c3ac66b9e0af2677a, n: '#36324', v: 6 },
  { _id: 53d7955c3ac66b9e0af2677b, n: '#36325', v: 6 },
  { _id: 53d7955c3ac66b9e0af26792, n: '#36348', v: 6 },
  { _id: 53d7955c3ac66b9e0af26794, n: '#36350', v: 6 },
  { _id: 53d7955c3ac66b9e0af267b2, n: '#36380', v: 6 },
  { _id: 53d7955c3ac66b9e0af267b3, n: '#36381', v: 6 },
  { _id: 53d7955c3ac66b9e0af267b6, n: '#36384', v: 6 } ]

real 0m1.078s
user 0m0.238s
sys  0m0.040s
```

The script runtime is 1 second on average. Now, write the script that adds an index on the *v* attribute:

```
'use strict';

var MongoClient = require('mongodb').MongoClient;

MongoClient.connect(
  'mongodb://127.0.0.1:27017/accounting',
  function (err, connection) {
    var collection = connection.collection('customers');

    collection.ensureIndex('v', function(err, indexName) {
      connection.close();
    });

  }
);
```

Call this one *addIndex.js* and execute it. Then, run the benchmark again. This is my result:

```
~$ time node query.js

[ { _id: 53d7955c3ac66b9e0af26745, n: '#36271', v: 6 },
  { _id: 53d7955c3ac66b9e0af26757, n: '#36289', v: 6 },
  { _id: 53d7955c3ac66b9e0af26761, n: '#36299', v: 6 },
  { _id: 53d7955c3ac66b9e0af2677a, n: '#36324', v: 6 },
  { _id: 53d7955c3ac66b9e0af2677b, n: '#36325', v: 6 },
  { _id: 53d7955c3ac66b9e0af26792, n: '#36348', v: 6 },
  { _id: 53d7955c3ac66b9e0af26794, n: '#36350', v: 6 },
  { _id: 53d7955c3ac66b9e0af267b2, n: '#36380', v: 6 },
  { _id: 53d7955c3ac66b9e0af267b3, n: '#36381', v: 6 },
  { _id: 53d7955c3ac66b9e0af267b6, n: '#36384', v: 6 } ]

real 0m0.269s
user 0m0.236s
sys  0m0.033s
```

From 1 second to 0.2 seconds - quite an improvement. This effect becomes more significant the larger the collection is. With a 2,000,000 documents collection, the query takes 5.6 seconds without an index and still only 0.2 seconds with an index on my machine.

Querying collections efficiently

When retrieving large result sets from a MongoDB collection, the same rule that applies to MySQL database result sets also applies here: reading the complete result set into our Node.js process at once

isn't going to be efficient resource-wise. Handling a result array with 2,000,000 entries will eat a lot of memory no matter what. This is what the *toArray* method, which we used until now, does:

```
collection.find().toArray(function (err, documents) {
  // We now have one large documents array
});
```

What *collection.find()* returns is a so-called *cursor object*. It can be transformed into an array, which is very convenient, but when handling lots of documents, it's better to handle each one separately using the cursor's *each* method:

```
collection.find().each(function (err, document) {
  // We retrieve one document with each callback invocation
});
```

This way, the cursor object will invoke the callback function for each document in the query result set, which means that Node.js only needs to claim memory for one document at a time.

Thus, instead of

```
collection.find().toArray(function (err, documents) {
  console.dir(documents);
});
```

we can do

```
collection.find().each(function (err, document) {
  console.dir(document);
});
```

if we want to print a whole result set to the screen. The callback will be invoked with the *document* parameter set to *null* if the end of the result set has been reached - therefore, we can rewrite our doFind-doInsert-doUpdate script like this:

```
'use strict';

var MongoClient = require('mongodb').MongoClient;

MongoClient.connect(
  'mongodb://127.0.0.1:27017/accounting',
  function (err, connection) {
    var collection = connection.collection('customers');

    var doFind = function (callback) {
      collection.find(
        {},
        {'sort': '_id'}
      ).each(function (err, document) {
        if (document === null) {
          callback();
        } else {
          console.dir(document);
        }
      });
    };

    var doInsert = function (i) {
      if (i < 20) {
        var value = Math.floor(Math.random() * 10);
        collection.insert(
          {'n': '#' + i, 'v': value},
          function (err, count) {
            doInsert(i + 1);
          });
      } else {
        console.log();
        console.log('Inserted', i, 'documents:');
        doFind(function () {
          doUpdate();
        });
      }
    };

    var doUpdate = function () {
      collection.update(
        {'n': /^#1/},
```

```
43          {'$mul': {'v': 2} },
44          {'multi': true},
45          function (err, count) {
46            console.log();
47            console.log('Updated', count, 'documents:');
48            doFind(function () {
49              collection.remove({}, function () {
50                connection.close();
51              });
52            });
53          });
54      };
55
56      doInsert(0);
57
58    });
```

The mongodb driver also provides a convenient way to access the records of a result set through a streaming API. The following code excerpt shows a version of the *doFind* function which has been rewritten to make use of this API:

```
var doFind = function (callback) {

  var stream = collection.find(
    {},
    {'sort': '_id'}
  ).stream();

  stream.on('data', function(document) {
    console.dir(document);
  });

  stream.on('close', function() {
    callback();
  });

};
```

As you can see, it's quite simple: The *stream* method of the cursor object returned by *collection.find()* returns a stream object which emits *data* and *close* events, among others.

We can then attach callback functions to these events where we either print the incoming document, or call the provided callback function in order to continue our script.

The stream object returned by *collection.find().stream()* implements the *stream.Readable* interface. See the corresponding Node.js documentation chapter[4] for more information.

Summary

In this chapter we learned about the workings of MongoDB and the *mongodb* driver, which enables us to insert, update, retrieve and remove documents from this NoSQL database. We learned about query filters and complex update operations, demonstrated how indexes can significantly accelerate query operations, and showed how to handle large result sets efficiently.

[4] http://nodejs.org/api/stream.html#stream_class_stream_readable