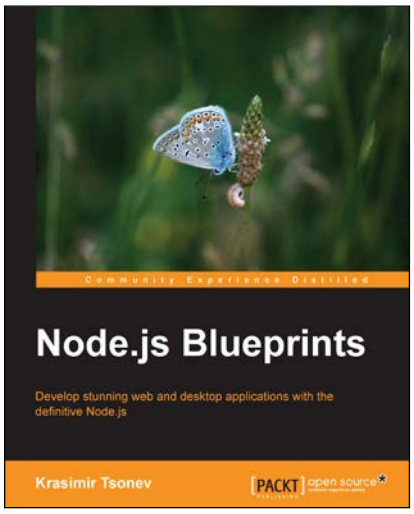
[PACKT] open source
community experience distilled
PUBLISHING

Node.js Blueprints

Krasimir Tsonev



Chapter No. 1
"Common Programming Paradigms"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.1 "Common Programming Paradigms"

A synopsis of the book’s content

Information on where to buy this book

About the Author

Krasimir Tsonev is a coder with over 10 years of experience in web development. With a strong focus on quality and usability, his interests lie in delivering cutting-edge applications. He enjoys working in the industry and has a passion for creating and discovering new and effective digital experiences. Currently, Krasimir works with technologies such as HTML5 or CSS3, JavaScript, PHP, and Node.js, although he started off as a graphic designer. Later, he spent several years as a flash developer using ActionScript3 and frameworks such as RobotLegs. After that, he continued delivering, as a freelancer, full-stack web services for his clients' graphic design as well as frontend and backend programming. With the present surge in mobile development, Krasimir is enthusiastic to work on responsive applications targeted at various devices. Having lived and worked in Bulgaria, he graduated from The Technical University of Varna with a Bachelor's and Master's degree in Computer Science.

Node.js Blueprints

As you probably know, the big things in our sphere are those that are moved by the community. Node.js is a technology that has become really popular. Its ecosystem is well-designed and brings with it the flexibility we need. With the rise of mobile development, JavaScript occupies a big part of the technology stack nowadays. The ability to use JavaScript on the server side is really interesting. It's good to know how Node.js works and where and when to use it, but it is more important to see some examples. This book will show you how this wonderful technology handles real use cases.

What This Book Covers

Chapter 1, Common Programming Paradigms, introduces us to the fact that Node.js is a JavaScript-driven technology, and we can apply common design patterns known in JavaScript in Node.js as well.

Chapter 2, Developing a Basic Site with Node.js and Express, discusses how ExpressJS is one of the top frameworks on the market. ExpressJS was included because of its fundamental importance in the Node.js world. At the end of the chapter, you will be able to create applications using the built-in Express modules and also add your own modules.

Chapter 3, Writing a Blog Application with Node.js and AngularJS, teaches you how to use frontend frameworks such as AngularJS with Node.js. The chapter's example is actually a dynamic application that works with real databases.

Chapter 4, Developing a Chat with Socket.IO, explains that nowadays, every big web app uses real-time data. It's important to show instant results to the users. This chapter covers the creation of a simple real-time chat. The same concept can be used to create an automatically updatable HTML component.

Chapter 5, Creating a To-do Application with Backbone.js, illustrates that Backbone.js was one of the first frameworks that introduced data binding at the frontend of applications. This chapter will show you how the library works. The to-do app is a simple example, but perfectly illustrates how powerful the framework is.

Chapter 6, Using Node.js as a Command-line Tool, covers the creation of a simple CLI program. There are a bunch of command-line tools written in Node.js, and the ability to create your own tool is quite satisfying. This part of the book will present a simple application which grabs all the images in a directory and uploads them to Flickr.

Chapter 7, Showing a Social Feed with Ember.js, describes an Ember.js example that will read a Twitter feed and display the latest posts. That's actually a common task of every developer because a lot of applications need to visualize social activity.

Chapter 8, Developing Web App Workflow with Grunt and Gulp, shows that there are a bunch of things to do before you can deliver the application to the users, such as concatenation, minification, templating, and so on. Grunt is the de facto standard for such tasks. The described module optimizes and speeds up your workflow. The chapter presents a simple application setup, including managing JavaScript, CSS, HTML, and cache manifests.

Chapter 9, Automate Your Testing with Node.js, signifies that tests are really important for every application nowadays. Node.js has some really great modules for this. If you are a fan of test-driven development, this chapter is for you.

Chapter 10, Writing Flexible and Modular CSS, introduces the fact that two of the most popular CSS preprocessors are written in Node.js. This chapter is like a little presentation on them and, of course, describes styling a simple web page.

Chapter 11, Writing a REST API, states that Node.js is a fast-working technology, and it is the perfect candidate for building a REST API. You will learn how to create a simple API to store and retrieve data for books, that is, an online library.

Chapter 12, Developing Desktop Apps with Node.js, shows that Node.js is not just a web technology—you can also create desktop apps with it. It's really interesting to know that you can use HTML, CSS, and JavaScript to create desktop programs. Creating a simple file browser may not be such a challenging task, but it will give you enough knowledge to build your own applications.

1
Common Programming Paradigms

Node.js is a JavaScript-driven technology. The language has been in development for more than 15 years, and it was first used in Netscape. Over the years, they've found interesting and useful design patterns, which will be of use to us in this book. All this knowledge is now available to Node.js coders. Of course, there are some differences because we are running the code in different environments, but we are still able to apply all these good practices, techniques, and paradigms. I always say that it is important to have a good basis to your applications. No matter how big your application is, it should rely on flexible and well-tested code. The chapter contains proven solutions that guarantee you a good starting point. Knowing design patterns doesn't make you a better developer because in some cases, applying the principles strictly won't work. What you actually get is ideas, which will help you in thinking out of the box. Sometimes, programming is all about managing complexity. We all meet problems, and the key to a well-written application is to find the best suitable solutions. The more paradigms we know, the easier our work is because we have proven concepts that are ready to be applied. That's why this book starts with an introduction to the most common programming paradigms.

Node.js fundamentals

Node.js is a single-threaded technology. This means that every request is processed in only one thread. In other languages, for example, Java, the web server instantiates a new thread for every request. However, Node.js is meant to use asynchronous processing, and there is a theory that doing this in a single thread could bring good performance. The problem of the single-threaded applications is the blocking I/O operations; for example, when we need to read a file from the hard disk to respond to the client. Once a new request lands on our server, we open the file and start reading from it. The problem occurs when another request is generated, and the application is still processing the first one. Let's elucidate the issue with the following example:

```
var http = require('http');
var getTime = function() {
  var d = new Date();
  return  d.getHours() + ':' + d.getMinutes() + ':' +
      d.getSeconds() + ':' + d.getMilliseconds();
}
var respond = function(res, str) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end(str + '\n');
  console.log(str + ' ' + getTime());
}
var handleRequest = function (req, res) {
  console.log('new request: ' + req.url + ' - ' + getTime());
  if(req.url == '/immediately') {
    respond(res, 'A');
  } else {
    var now = new Date().getTime();
    while(new Date().getTime() < now + 5000) {
      // synchronous reading of the file
    }
    respond(res, 'B');
  }
}
http.createServer(handleRequest).listen(9000, '127.0.0.1');
```

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

The `http` module, which we initialize on the first line, is needed for running the web server. The `getTime` function returns the current time as a string, and the `respond` function sends a simple text to the browser of the client and reports that the incoming request is processed. The most interesting function is `handleRequest`, which is the entry point of our logic. To simulate the reading of a large file, we will create a `while` cycle for 5 seconds. Once we run the server, we will be able to make an HTTP request to `http://localhost:9000`. In order to demonstrate the single-thread behavior we will send two requests at the same time. These requests are as follows:

- One request will be sent to `http://localhost:9000`, where the server will perform a synchronous operation that takes 5 seconds
- The other request will be sent to `http://localhost:9000/immediately`, where the server should respond immediately

The following screenshot is the output printed from the server, after pinging both the URLs:

```
$ node .\server-test.js
new request: / - 16:58:30:434
B 16:58:35:437
new request: /immediately - 16:58:35:440
A 16:58:35:441
```

As we can see, the first request came at `16:58:30:434`, and its response was sent at `16:58:35:440`, that is, 5 seconds later. However, the problem is that the second request is registered when the first one finishes. That's because the thread belonging to Node.js was busy processing the `while` loop.

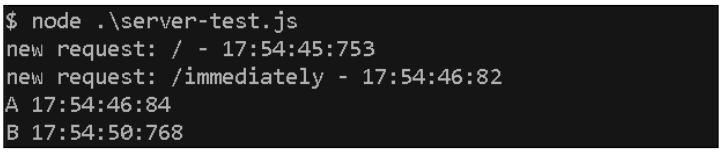
Of course, Node.js has a solution for the blocking I/O operations. They are transformed to asynchronous functions that accept callback. Once the operation finishes, Node.js fires the callback, notifying that the job is done. A huge benefit of this approach is that while it waits to get the result of the I/O, the server can process another request. The entity that handles the external events and converts them into callback invocations is called the `event` loop. The `event` loop acts as a really good manager and delegates tasks to various workers. It never blocks and just waits for something to happen; for example, a notification that the file is written successfully.

Now, instead of reading a file synchronously, we will transform our brief example to use asynchronous code. The modified example looks like the following code:

```
var handleRequest = function (req, res) {
  console.log('new request: ' + req.url + ' - ' + getTime());
  if(req.url == '/immediately') {
    respond(res, 'A');
```

```
    } else {
      setTimeout(function() {
        // reading the file
        respond(res, 'B');
        }, 5000);
      }
  }
```

The `while` loop is replaced with the `setTimeout` invocation. The result of this change is clearly visible in the server's output, which can be seen in the following screenshot:



The first request still gets its response after 5 seconds. However, the second one is processed immediately.

Organizing your code logic in modules

If we write a lot of code, sooner or later, we will start realizing that our logic should be split into different modules. In most languages, this is done through classes, packages, or some other language-specific syntax. However, in JavaScript, we don't have classes natively. Everything is an object, and in practice, objects inherit other objects. There are several ways to achieve object-oriented programming within JavaScript. You can use prototype inheritance, object literals, or play with function calls. Thankfully, Node.js has a standardized way of defining modules. This is approached by implementing **CommonJS**, which is a project that specifies an ecosystem for JavaScript.

So, you have some logic, and you want to encapsulate it by providing useful API methods. If you reach that moment, you are definitely in the right direction. This is really important, and maybe it is one of the most challenging aspects of programming nowadays. The ability to split our applications into different parts and delegate functions to them is not always an easy task. Very often, this is undervalued, but it's the key to good architecture. If a module contains a lot of dependencies, operates with different data storages, or has several responsibilities, then we are doing something wrong. Such code cannot be tested and is difficult to maintain. Even if we take care about these two things, it is still difficult to extend the code and continue working with it. That's why it's good to define different modules for different functionalities. In the context of Node.js, this is done via the `exports` keyword, which is a reference to `module.exports`.

Building a car construction application

Let's elucidate the process with a simple example. Assume that we are building an application that constructs a car. We need one main module (`car`) and a few other modules, which are responsible for the different parts of the car (`wheels`, `windows`, `doors`, and so on). Let's start with the definition of a module representing the wheels of the car, with the following code:

```
// wheels.js
var typeOfTires;
exports.init = function(type) {
    typeOfTires = type;
}
exports.info = function() {
  console.log("The car uses " + typeOfTires + " tires.");
}
```

The preceding code could be the content of `wheels.js`. It contains two methods. The first method, `init`, should be called first and accepts one setting, that is, the type of the wheels' tires. The second method simply outputs some information. In our main file, `car.js`, we have to get an instance of the wheels and use the provided API methods. This can be done as follows:

```
// cars.js
  var wheels = require("./wheels.js");
  wheels.init("winter");
  wheels.info();
```

When you run the application with `node car.js`, you will get the following output:

```
The car uses winter tires.
```

So, everything that you want to expose to the outside world should be attached to the `export` object. Note that `typeOfTires` is a local variable for the module. It is available only in `wheels.js` and not in `car.js`. It's also a common practice to apply an object or a function to the `exports` object directly, as shown in the following code for example:

```
// engine.js
var Class = function() {
    // ...
}
Class.prototype = {
  forward: function() {
    console.log("The car is moving forward.");
  },
```

```
    backward: function() {
      console.log("The car is moving backward.");
    }
  }
  module.exports = Class;
```

In JavaScript, everything is an object and that object has a `prototype` property. It's like a storage that keeps the available variables and methods. The `prototype` property is heavily used during inheritance in JavaScript, because it provides a mechanism for transferring logic.

We will also clear the difference between `module.exports` and `exports`. As you can see, in `wheels.js`, we assigned two functions, `init` and `info`, directly to the `exports` global object. In fact, that object is a reference to `module.exports`, and every function or variable attached to it is available to the outside world. However, if we assign a new object or function directly to the `export` object, we should not expect to get an access to it after requiring the file. This should be done with `module.exports`. Let's take the following code as an example:

```
// file.js
module.exports.a = 10;
exports.b = 20;

// app.js
var file = require('./file');
console.log(file.a, file.b);
```

Let's say that both the files, `app.js` and `file.js`, are in the same directory. If we run `node app.js`, we will get `10 20` as the result. However, consider what would happen if we changed the code of `file.js` to the following code:

```
module.exports = { a: 10 };
exports.b = 20;
```

Then, in this case, we would get `10 undefined` as the result. That's because `module.exports` has a new object assigned and `exports` still points to the old one.

Using the car's engine

Let's say that the module in `engine.js` controls the car. It has methods for moving the car forward and backward. It is a little different because the logic is defined in a separate class and that class is directly passed as a value of `module.exports`. In addition, as we are exporting a function, and not just an object, our instance should be created with the `new` keyword. We will see how the car's engine works with the `new` keyword as shown in the following code:

```
var Engine = require("./engine.js");
var e = new Engine();
e.forward();
```

There is a significant difference between using JavaScript functions as constructors and calling them directly. When we call the function as a constructor, we get a new object with its own prototype. If we miss the `new` keyword, the value which we get at the end is the result of the function's invocation.

Node.js caches the modules returned by the `require` method. It's done to prevent the blocking of the `event` loop and increase the performance. It's a synchronous operation, and if there is no cache, Node.js will have to do the same job repeatedly. It's also good to know that we can call the method with just a folder name, but there should be a `package.json` or an `index.js` file inside the directory. All these mechanisms are described well in the official documentation of Node.js at `http://nodejs.org/`. What is important to note here is that the environment encourages modular programming. All we need is native implementation into the system, and we don't have to use a third-party solution that provides modularity.

Like in the client-side code, every Node.js module can be extended. Again, as we are writing the code in plain JavaScript, we can use the well-known approaches for inheritance. For example, take a look at the following code:

```
var Class = function() { }
Class.prototype = new require('./engine.js')();
Class.prototype.constructor = Class;
```

Node.js even offers a helper method for this purpose. Let's say that we want to extend our `engine.js` class and add API methods to move the car in the left and right directions. We can do this with the following piece of code:

```
// control.js
var util = require("util");
var Engine = require("./engine.js");
var Class = function() { }
util.inherits(Class, Engine);
Class.prototype.left = function() {
  console.log("The car is moving to left.");
};
Class.prototype.right = function() {
  console.log("The car is moving to right.");
}
module.exports = Class;
```

The first line gets a reference to the Node.js native `utils` module. It's full of useful functions. The fourth line is where the magic happens. By calling the `inherits` method, we have actually set a new prototype of our `Class` object. Keep in mind that every new method should use the already applied prototype. That's why the `left` and `right` methods are defined after the inheritance. At the end, our car will move in four directions, as shown in the following code snippet:

```
var Control = require("./control.js");
var c = new Control();
c.forward();
c.right();
```

Understanding inter-module communication

We've found out how to put our code logic into modules. Now, we need to know how to make them communicate with each other. Very often, people describe Node.js as an event-driven system. It's also called non-blocking because as we have seen earlier in the chapter, it can accept a new request even before the previous request is fully complete. That's very efficient and highly scalable. The events are very powerful and are good means to inform the other modules of what is going on. They bring about encapsulation, which is very important in modular programming. Let's add some events to the car example we discussed earlier. Let's say that we have air conditioning, and we need to know when it is started. The implementation of such logic consists of two parts. The first one is the air conditioning module. It should dispatch an event that indicates the start of the action. The second part is the other code that listens for that event. We will create a new file called `air.js` containing the logic responsible for the air conditioning, as follows:

```
// air.js
var util = require("util");
var EventEmitter = require('events').EventEmitter;
var Class = function() { }
util.inherits(Class, EventEmitter);
Class.prototype.start = function() {
  this.emit("started");
};
module.exports = Class;
```

Our class extends a Node.js module called `EventEmitter`. It contains methods such as `emit` or `on`, which help us to establish event-based communication. There is only one custom method defined: `start`. It simply dispatches an event that indicates that the air conditioning is turned on. The following code shows how we can attach a listener:

```
// car.js
var AirConditioning = require("./air.js");
var air = new AirConditioning();
air.on("started", function() {
  console.log("Air conditioning started");
});
air.start();
```

A new instance of the `AirConditioning` class is created. We attached an event listener and fired the `start` method. The handler is called, and the message is printed to the console. The example is a simple one but shows how two modules communicate. It's a really powerful approach because it offers encapsulation. The module knows its responsibilities and is not interested in the operations in the other parts of the system. It simply does its job and dispatches notifications (events). For example, in the previous code, the `AirConditioning` class doesn't know that we will output a message when it is started. It only knows that one particular event should be dispatched.

Very often, we need to send data during the emitting of an event. This is really easy. We just have to pass another parameter along with the name of the event. Here is how we send a `status` property:

```
Class.prototype.start = function() {
  this.emit("started", { status: "cold" });
};
```

The object attached to the event contains some information about the air conditioning module. The same object will be available in the listener of the event. The following code shows us how to get the value of the `status` variable mentioned previously:

```
air.on("started", function(data) {
  console.log("Status: " + data.status);
});
```

There is a design pattern that illustrates the preceding process. It's called the **Observer**. In the context of that pattern, our air conditioning module is called **subject**, and the car module is called the observer. The subject broadcasts messages or events to its observers, notifying them that something has changed.

If we need to remove a listener, Node.js has a method for that called `removeListener`. We can even allow a specific number of observers using `setMaxListeners`. Overall, the events are one of the best ways to wire your logical parts. The main benefit is that you isolate the module, but it is still highly communicative with the rest of your application.

Asynchronous programming

As we already learned, in nonblocking environments, such as Node.js, most of the processes are asynchronous. A request comes to our code, and our server starts processing it but at the same time continues to accept new requests. For example, the following is a simple file reading:

```
fs.readFile('page.html', function (err, content) {
  if (err) throw err;
  console.log(content);
});
```

The `readFile` method accepts two parameters. The first one is a path to the file we want to read, and the second one is a function that will be called when the operation finishes. The callback is fired even if the reading fails. Additionally, as everything can be done via that asynchronous matter, we may end up with a very long callback chain. There is a term for that—callback hell. To elucidate the problem, we will extend the previous example and do some operations with the file's content. In the following code, we are nesting several asynchronous operations:

```
fs.readFile('page.html', function (err, content) {
  if(err) throw err;
  getData(function(data) {
    applyDataToTheTemplate(content, data, function(resultedHTML) {
      renderPage(resultedHTML, function() {
        showPage(function() {
          // finally, we are done
        });
     });
  });
  });
});
```

As you can see, our code looks bad. It's difficult to read and follow. There are a dozen instruments that can help us to avoid such situations. However, we can fix the problem ourselves. The very first step to do is to spot the issue. If we have more than four or five nested callbacks, then we definitely should refactor our code. There is something very simple, which normally helps, that makes the code **shallow**.
The previous code could be translated to a more friendly and readable format. For example, see the following code:

```
var onFileRead = function(content) {
  getData(function(data) {
    applyDataToTheTemplate(content, data, dataApplied);
  });
}
var dataApplied = function(resultedHTML) {
  renderPage(resultedHTML, function() {
    showPage(weAreDone);
  });
}
var weAreDone = function() {
  // finally, we are done
}
fs.readFile('page.html', function (err, content) {
  if (err) throw err;
    onFileRead(content);
});
```

Most of the callbacks are just defined separately. It is clear what is going on because the functions have descriptive names. However, in more complex situations, this technique may not work because you will need to define a lot of methods. If that's the case, then it is good to combine the functions in an external module. The previous example can be transformed to a module that accepts the name of a file and the callback function. The module is as follows:

```
var renderTemplate = require("./renderTemplate.js");
renderTemplate('page.html', function() {
  // we are done
});
```

You still have a callback, but it looks like the helper methods are hidden and only the main functionality is visible.

Another popular instrument for dealing with asynchronous code is the **promises** paradigm. We already talked about events in JavaScript, and the promises are something similar to them. We are still waiting for something to happen and pass a callback. We can say that the promises represent a value that is not available at the moment but will be available in the future. The syntax of promises makes the asynchronous code look synchronous. Let's see an example where we have a simple module that loads a Twitter feed. The example is as follows:

```
var TwitterFeed = require('TwitterFeed');
TwitterFeed.on('loaded', function(err, data) {
  if(err) {
      // ...
   } else {
      // ...
   }
});
TwitterFeed.getData();
```

We attached a listener for the `loaded` event and called the `getData` method, which connects to Twitter and fetches the information. The following code is what the same example will look like if the `TwitterFeed` class supports promises:

```
var TwitterFeed = require('TwitterFeed');
var promise = TwitterFeed.getData();
promise.then(function(data) {
  // ...
}, function(err) {
  // ...
});
```

The `promise` object represents our data. The first function, which is sent to the `then` method, is called when the `promise` object succeeds. Note that the callbacks are registered after calling the `getData` method. This means that we are not rigid to actual process of getting the data. We are not interested in when the action occurs. We only care when it finishes and what its result is. We can spot a few differences from the event-based implementation. They are as follows:

- There is a separate function for error handling.
- The `getData` method can be called before calling the `then` method. However, the same thing is not possible with events. We need to attach the listeners before running the logic. Otherwise, if our task is synchronous, the event may be dispatched before our listener attachment.
- The **promise** method can only succeed or fail once, while one specific event may be fired multiple times and its handlers can be called multiple times.

The promises get really handy when we chain them. To elucidate this, we will use the same example and save the tweets to a database with the following code:

```
var TwitterFeed = require('TwitterFeed');
var Database = require('Database');
var promise = TwitterFeed.getData();
promise.then(function(data) {
  var promise = Database.save(data);
  return promise;
}).then(function() {
  // the data is saved
  // into the database
}).catch(function(err) {
  // ...
});
```

So, if our successful callback returns a new promise, we can use `then` for the second time. Also, we have the possibility to set only one error handler. The `catch` method at the end is fired if some of the promises are rejected.

There are four states of every promise, and we should mention them here because it's a terminology that is widely used. A promise could be in any of the following states:

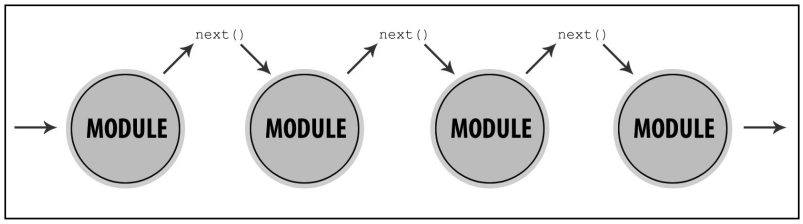
- **Fulfilled**: A promise is in the fulfilled state when the action related to the promise succeeds
- **Rejected**: A promise is in the rejected state when the action related to the promise fails
- **Pending**: A promise is in the pending state if it hasn't been fulfilled or rejected yet
- **Settled**: A promise is in a settled state when it has been fulfilled or rejected

The asynchronous nature of JavaScript makes our coding really interesting. However, it could sometimes lead to a lot of problems. Here is a wrap up of the discussed ideas to deal with the issues:

- Try to use more functions instead of closures
- Avoid the pyramid-looking code by removing the closures and defining top-level functions
- Use events
- Use promises

Exploring middleware architecture

The Node.js framework is based on the middleware architecture. That's because this architecture brings modularity. It's really easy to add or remove functionalities from the system without breaking the application because the different modules do not depend on each other. Imagine that we have several modules that are all stored in an array, and our application starts using them one by one. We are controlling the whole process, that is, the execution continues only if we want it to. The concept is demonstrated in the following diagram:



Connect (`https://github.com/senchalabs/connect`) is one of the first frameworks that implements this pattern. In the context of Node.js, the middleware is a function that accepts the request, response, and the next callbacks. The first two parameters represent the input and output of the middleware. The last one is a way to pass the flow to the next middleware in the list. The following is a short example of this:

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

var app = connect()
  .use(function(req, res, next) {
    console.log("That's my first middleware");
    next();
  })
  .use(function(req, res, next) {
    console.log("That's my second middleware");
    next();
  })
  .use(function(req, res, next) {
    console.log("end");
    res.end("hello world");
  });

http.createServer(app).listen(3000);
```

The `use` method of `connect` accepts middleware. In general, the middleware is just a simple JavaScript function. We can write whatever we want in it. What is important to do at the end is to call the `next` method. It passes the flow to the next middleware. Often, we will need to transfer data between the middleware. It's a common practice to modify the request or the response objects because they are the input and output of the module. We can attach new properties or functions, and they will be available for the next middleware in the list. As in the following code snippet, we are attaching an object to a `data` property.

```
.use(function(req, res, next) {
    req.data = { value: "middleware"};
    next();
})
.use(function(req, res, next) {
    console.log(req.data.value);
})
```

The request and response objects are identical in every function. Thus, the middleware share the same scope. At the same time, they are completely independent. This pattern provides a really flexible development environment. We can combine modules that do different tasks written by different developers.

Composition versus inheritance

In the previous section, we learned how to create modules, how to make them communicate, and how to use them. Let's talk a bit about how to architect modules. There are dozens of ways to build a good application. There are also some great books written only on this subject, but we will focus on two of the most commonly used techniques: composition and inheritance. It's really important to understand the difference between the two. They both have pros and cons. In most of the cases, their usage depends on the current project.

The `car` class from the previous sections is a perfect example of composition. The functionalities of the `car` object are built by other small objects. So, the main module actually delegates its jobs to other classes. For example, the wheels or the air conditioning of the car are controlled by externally defined modules:

```
var wheels = require("./wheels.js")();
var control = require("./control.js")();
var airConditioning = require("./air.js")();
module.export = {
  run: function() {
    wheels.init();
```

```
        control.forward();
        airConditioning.start();
      }
    }
```

For the outside world, the car has only one method: `run`. However, what happens is that we perform three different operations, and they are defined in other modules. Often, the composition is preferred over the inheritance because while using this approach, we can easily add as many modules as we want. It's also interesting that we cannot only include modules but also other compositions.

On the other side is the inheritance. The following code is a typical example of inheritance:

```
var util = require("util");
var EventEmitter = require('events').EventEmitter;
var Class = function() { }
util.inherits(Class, EventEmitter);
```

This code implies that our class needs to be an event emitter, so it simply inherits that functionality from another class. Of course, in this case, we can still use composition and create an instance of the `EventEmitter` class, define methods such as `on` and `dispatch`, and delegate the real work. However, here it is much better to use inheritance.

The truth is somewhere in between—the composition and the inheritance should play together. They are really great tools, but each of them has its own place. It's not only black and white, and sometimes it is difficult to find the right direction. There are three ways to add behavior to our objects. They are as follows:

- Writing the functionality into the objects directly
- Inheriting the functionality from a class that already has the desired behavior
- Creating a local instance of an object that does the job

The second one is related to inheritance and the last one is actually a composition. By using composition, we are adding a few more abstraction layers, which is not a bad thing, but it could lead to unnecessary complexity.

Managing dependencies

Dependency management is one of the biggest problems in complex software. Often, we build our applications around third-party libraries or custom-made modules written for other projects. We do this because we don't want to reinvent the wheel every time.

In the previous sections of this chapter, we used the `require` global function. That's how Node.js adds dependencies to the current module. A functionality written in one JavaScript file is included in another file. The good thing is that the logic in the imported file lives in its own scope, and only the publicly exported functions and variables are visible to the host. With this behavior, we are able to separate our logic modules into Node.js packages. There is an instrument that controls such packages. It's called **Node Package Manager** (**npm**) and is available as a command-line instrument. Node.js has become so popular mainly because of the existence of its package manager. Every developer can publish their own package and share it with the community. The good versioning helps us to bind our applications to specific versions of the dependencies, which means that we can use a module that depends on other modules. The main rule to make this work is to add a `package.json` file to our project. We will add this file with the following code:

```
{
  "name": "my-awesome-module",
  "version": "0.1.10",
  "dependencies": {
    "optimist": "0.6.1",
    "colors": "0.6.2"
  }
}
```

The content of the file should be valid JSON and should contain at least the `name` and `version` fields. The `name` property should also be unique, and there should not be any other module with the same name. The `dependencies` property contains all the modules and versions that we depend on. To the same file, we can add a lot of other properties. For example, information about the author, a description of the package, the license of the project, or even keywords. Once the module is registered in the registry, we can use it as a dependency. We just need to add it in our `package.json` file, and after we run `npm install`, we will be able to use it as a dependency. Since Node.js adopts the module pattern, we don't need instruments such as the dependency injection container or service locater.

Let's write a `package.json` file for the car example used in the previous sections, as follows:

```
{
  "name": "my-awesome-car",
  "version": "0.0.1",
  "dependencies": {
    "wheels": "2.0.1",
    "control": "0.1.2",
    "air": "0.2.4"
  }
}
```

Summary

In this chapter, we went through the most common programming paradigms in Node.js. We learned how Node.js handles parallel requests. We understood how to write modules and make them communicative. We saw the problems of the asynchronous code and their most popular solutions. At the end of the chapter, we talked about how to construct our application. With all this as a basis, we can start thinking about better programs. Software writing is not an easy task and requires strong knowledge and experience. The experience usually comes after years of coding; however, knowledge is something that we can get instantly. Node.js is a young technology; nonetheless, we are able to apply paradigms and concepts from client-side JavaScript and even other languages.

In the next chapter, we will see how to use one of the most popular frameworks for Node.js, that is, Express.js, and we will build a simple website.

Where to buy this book

You can buy Node.js Blueprints from the Packt Publishing website: `http://www.packtpub.com/nodejs-blueprints/book`.

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com

For More Information:
www.packtpub.com/nodejs-blueprints/book