Extracted from:

Node.js the Right Way

Practical, Server-Side JavaScript That Scales

The Pragmatic Bookshelf

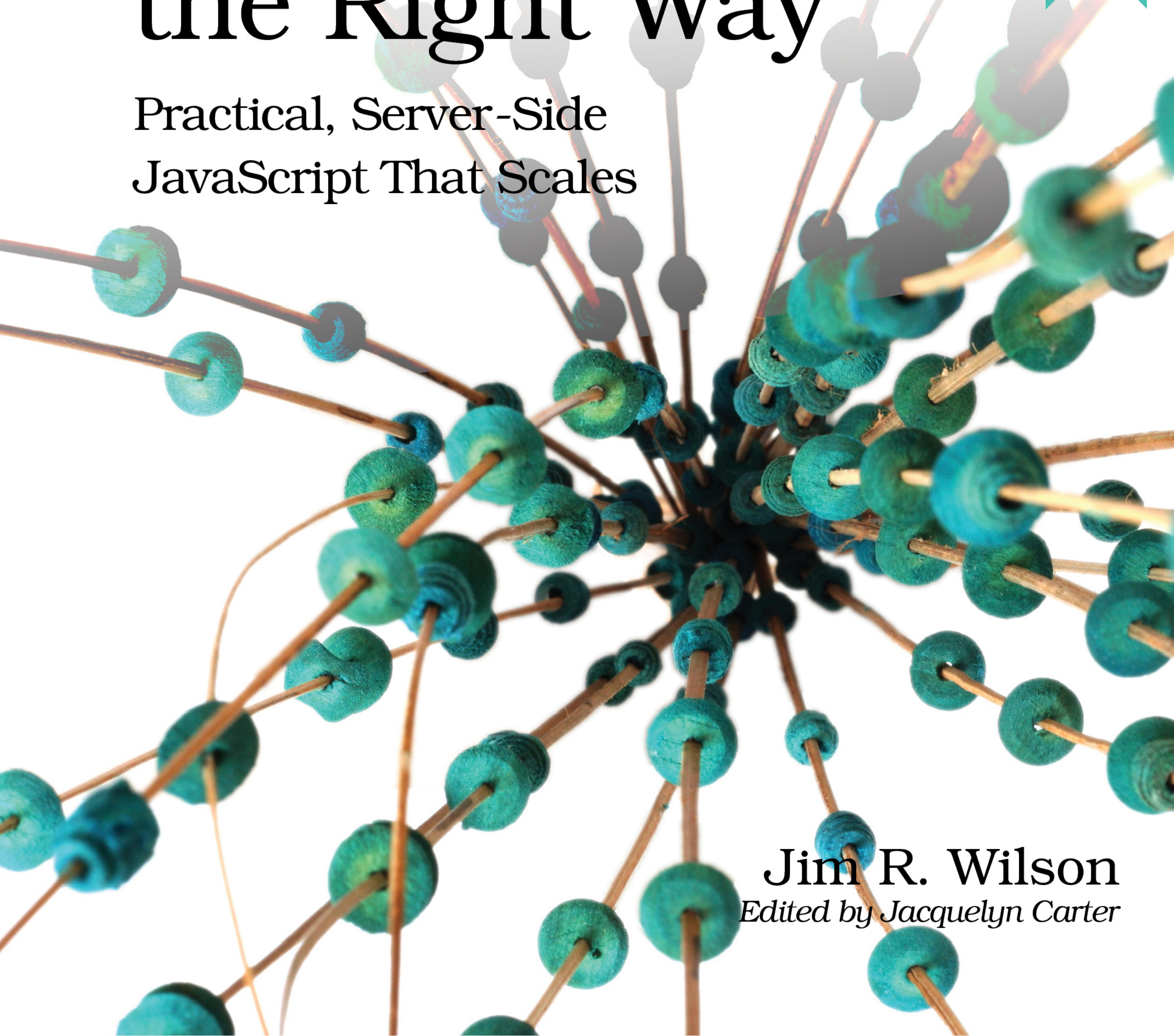
Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Pragmatic exPress

Node.js the Right Way

Practical, Server-Side JavaScript That Scales



Jim R. Wilson

Edited by Jacquelyn Carter

Node.js the Right Way

Practical, Server-Side JavaScript That Scales

Jim R. Wilson

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The team that produced this book includes:

Jacquelyn Carter (editor)
Candace Cunningham (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-937785-73-4
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—December 2013

Node.js is built from the ground up to do networked programming. In this chapter, we'll explore Node's built-in support for low-level socket connections. TCP sockets form the backbone of modern networked applications, and understanding them will serve you well as we do more complex networking through the rest of the book.

As you develop socket-based servers and clients, you'll learn about the following Node.js aspects.

Architecture and Core
: The asynchronous programming techniques we explored in the last chapter will be even more important here. You'll learn how to extend Node.js classes like `EventEmitter`. You'll create custom modules to house reusable code.

Patterns
: A network connection has two endpoints. A common pattern is for one endpoint to act as the server while the other is the client. We'll develop both kinds of endpoints in this chapter, as well as a JavaScript Object Notation (JSON)-based protocol for client/server communication.

JavaScriptisms
: The JavaScript language has an interesting inheritance model. You'll learn about Node's utilities for creating class-like relationships between objects.

Supporting Code
: Testing is important to ensure that our programs behave the way we expect them to. In this chapter, we'll develop a test server that behaves badly on purpose to probe edge cases of our protocol.

To begin, we'll develop a simple and complete TCP server program. Then we'll iteratively improve the server as we address concerns such as robustness, modularity, and testability.

Listening for Socket Connections

Networked services exist to do two things: connect endpoints and transmit information between them. No matter what kind of information is transmitted, a connection must first be made.

In this section, you'll learn how to create socket-based services using Node.js. We'll develop an example application that sends data to connected clients, then we'll connect to this service using standard command-line tools. By the end, you'll have a good idea of how Node does the client/server pattern.

Binding a Server to a TCP Port

TCP socket connections consist of two *endpoints*. One endpoint *binds* to a numbered port while the other endpoint *connects* to a port.

This is a lot like a telephone system. One phone binds a given phone number for a long time. A second phone places a call—it connects to the bound number. Once the call is answered, information (sound) can travel both ways.

In Node.js, the bind and connect operations are provided by the net module. Binding a TCP port to listen for connections looks like this:

```
"use strict";
const
  net = require('net'),
  server = net.createServer(function(connection) {
    // use connection object for data transfer
  });
server.listen(5432);
```

The net.createServer() method takes a callback function and returns a Server object. Node invokes the callback function whenever another endpoint connects. The connection parameter is a Socket object that you can use to send or receive data.

Calling server.listen() binds the specified port. In this case, we're binding TCP port number 5432. Figure 4, *A Node.js server binding a TCP socket for listening*, on page 7 shows this basic setup. The figure shows our one Node.js process whose server binds a TCP port. Any number of clients—which may or may not be Node.js processes—can connect to that bound port.

Our server program doesn't do anything with the connection yet. Let's fix that by using it to send some useful information to the client.

Writing Data to a Socket

In Chapter 2, *Wrangling the File System*, on page ?, we developed some simple file utilities that would take action whenever a target file changed. Let's reuse the file changes as a source of information for our example networked service. This will give us something to code against as we dig into aspects of Node.js development.

Open your favorite text editor and enter this:

networking/net-watcher.js

```
'use strict';
const
```

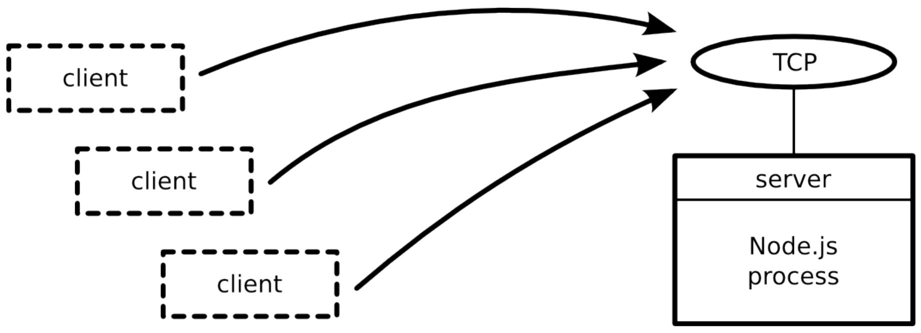


Figure 4—A Node.js server binding a TCP socket for listening

```
  fs = require('fs'),
  net = require('net'),

  filename = process.argv[2],

  server = net.createServer(function(connection) {

    // reporting
    console.log('Subscriber connected.');
    connection.write("Now watching '" + filename + "' for changes...\n");

    // watcher setup
    let watcher = fs.watch(filename, function() {
      connection.write("File '" + filename + "' changed: " + Date.now() + "\n");
    });

    // cleanup
    connection.on('close', function() {
      console.log('Subscriber disconnected.');
      watcher.close();
    });

  });

if (!filename) {
  throw Error('No target filename was specified.');
}

server.listen(5432, function() {
  console.log('Listening for subscribers...');
});
```

Save the file as net-watcher.js. Most of the code here is taken from previous examples in the book, so it should look pretty familiar. The novel parts to the net-watcher program begin inside the callback function given to createServer(). This callback function does three things:

- It reports that the connection has been established (both to the client with connection.write and to the console).
- It begins listening for changes to the target file, saving the returned watcher object. This callback sends change information to the client using connection.write.
- It listens for the connection's close event so it can report that the subscriber has disconnected and stop watching the file, with watcher.close().

Finally, notice the callback passed into server.listen() at the end. Node invokes this function after it has successfully bound port 5432 and is ready to start receiving connections.

Connecting to a TCP Socket Server with Telnet

Now let's run the net-watcher program and confirm that it behaves the way we expect. This will require a little terminal juggling.

To run and test the net-watcher program, you'll need three terminal sessions: one for the service itself, one for the client, and one to trigger changes to the watched file. In the first terminal, run the net-watcher program:

```
$ node --harmony net-watcher.js target.txt
Listening for subscribers...
```

This program creates a service listening on TCP port 5432. To connect to it, open a second terminal and use the telnet program like so:

```
$ telnet localhost 5432
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Now watching target.txt for changes...
```

Back in the first terminal, you should see this:

```
Subscriber connected.
```

Finally, to trigger a change to the watched file, open a third terminal and touch the file target.txt:

```
$ touch target.txt
```

In the telnet terminal, after a moment you should see a line like this:

```
File 'target.txt' changed: Sat Jan 12 2013 12:35:52 GMT-0500 (EST)
```

You can kill the telnet session by typing Ctrl-] and then Ctrl-C. If you do, you'll see the following line appear in the first terminal:

```
Subscriber disconnected.
```

To terminate the net-watcher service, type Ctrl-C from its terminal.

The following figure outlines the setup we just created. The net-watcher process (box) binds a TCP port and watches a file—both resources are shown as ovals.

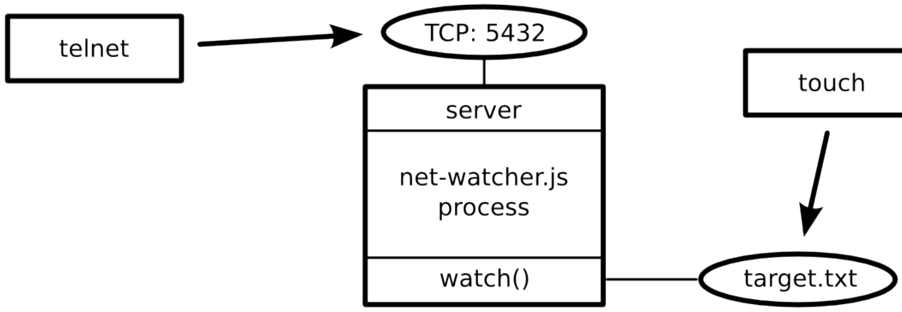


Figure 5—A Node.js program watching a file and reporting changes to connected TCP clients

More than one subscriber can connect and receive updates simultaneously. If you open additional terminals and connect to port 5432 with telnet, they'll all receive updates when you touch the target file.

TCP sockets are useful for communicating between networked computers. But if you need processes on the same computer to communicate, Unix sockets offer a more efficient alternative. The net module can create this kind of socket as well, which we'll look at next.

Listening on Unix Sockets

To see how the net module uses Unix sockets, let's modify the net-watcher program to use this kind of communication channel. Keep in mind that Unix sockets work only on Unix-like environments.

Open the net-watcher.js program and change the server.listen() section to this:

```
server.listen('/tmp/watcher.sock', function() {
  console.log('Listening for subscribers...');
});
```

Save the file as net-watcher-unix.js, then run the program as before:

```
$ node --harmony net-watcher-unix.js target.txt
Listening for subscribers...
```

To connect a client, we now need nc instead of telnet. nc is short for netcat, a TCP/UDP socket utility program that also supports Unix sockets.

```
$ nc -U /tmp/watcher.sock
Now watching target.txt for changes...
```

Unix sockets can be faster than TCP sockets because they don't require invoking network hardware. However, they're local to the machine.

That concludes the basics of creating network socket servers in Node. We discovered how to create socket servers and connect to them using common client utility programs like telnet and nc. This framework will supply the backdrop for the rest of the examples in the chapter.

Next, we'll beef up our service by transforming the data into a parsable format. This will put us in position to develop custom client applications.