Chapter 2. Your First SPDY App

I’m not gonna mess around in this book, so let’s jump right into a SPDY `hello world` app:

```
app.get('/hello', function(req, res){
  res.render('hello', {
    title: 'Hello World!'
  });
});
```

If you are familiar with node.js, you will likely recognize that bit of code.

If you have never done node.js, that code is still pretty easy to follow thanks to a simple DSL (Domain Specific Language). This DSL describes how to route an HTTP `GET` request for the `/hello` resource. In this case, the response merely renders the `hello` template, passing along a `title` parameter of "Hello World".

Now you might be thinking to yourself, "I plunked down good money to learn SPDY and he’s talking about serving up HTTP with node.js?! How do I get a refund?" First off, all sales are final—no refunds. Just kidding.

Actually, this is lesson #1:

> **Tip**
>
> *Lesson #1:* SPDY serves up HTTP applications. As we will see, it definitely changes a few things here or there. The SPDY protocol builds on HTTP, it does not replace it.

So is that it? Nothing else is needed to run a SPDY application?

2.1. OK I Lied

The `hello world` sample app is not *really* SPDY. Yet.

The DSL that the sample app uses comes from the express.js framework. If you have ever seen the Sinatra framework in Ruby, then you understand what express.js is meant for—a simple way to write HTTP applications.

So what do we need to do to SPDY-ize it?

Two things. We need an adapter layer, `express-spdy` in this case, and some configuration:

```
// Require necessary modules
var express = require('express-spdy')
  , fs = require('fs');

// Create an instance of an application
// server -- SPDY-ized
var app = express.createServer({
  key: fs.readFileSync('keys/spdy-key.pem'),
  cert: fs.readFileSync('keys/spdy-cert.pem'),
  ca: fs.readFileSync('keys/spdy-csr.pem'),
  NPNProtocols: ['spdy/2', 'http/1.1']
}

app.get('/hello', function(req, res){
  res.render('hello', {
    title: 'Hello World!'
  });
});
```

In a regular express.js application, the `express` variable would come from requiring the `express` module. The `express-spdy` module takes the express.js framework and adds a few features to it—kinda like SPDY adds a few things to HTTP.

The API for express-spdy is unchanged from express.js, including the call to `createServer` that returns an express.js application object. Here, we are returning a SPDY-ized version of the express.js application object.

The first three configuration options that are passed to `createServer` — `key`, `cert`, and `ca` — are SSL (Secure Socket Layer) options. In fact, those options come straight out of express.js. They are, of course, not required in express.js, but with SPDY, SSL is **mandatory**. We will cover SSL in depth in Chapter 8, *SPDY and SSL*.

Tip

Lesson #2: SPDY is served over SSL. In an age where session hijacking is trivial thanks to tools like Firesheep, all non-trivial applications should be served with SSL. There is a performance hit by using SSL, but SPDY acknowledges this as a price of writing serious, modern applications.

The next configuration option, `NPNProtocols`, is the real difference between an express.js app and an express-spdy app. NPN, or Next Protocol Negotiation, is a relatively new feature of SSL. NPN is a vehicle for web servers to communicate to browsers the protocols that they can speak.

In this case, we are advertising that our `hello world` application can speak both `spdy/2` (SPDY, version 2) and `http/1.1`. Browsers that do not understand NPN or SPDY can still make regular HTTP requests of our `hello world` server.

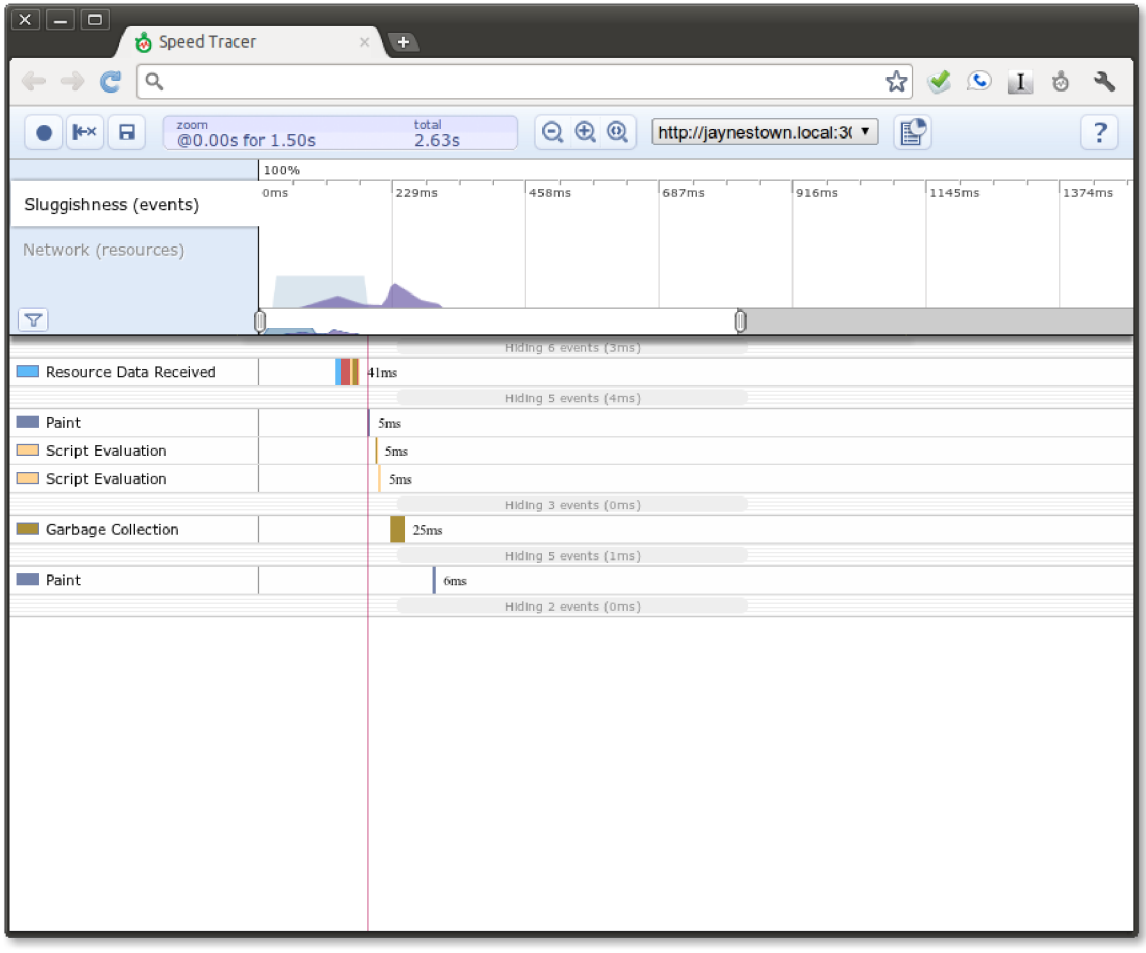
But browsers that understand NPN and SPDY, such as Google's Chrome, will speak SPDY to the server. And it will make a *huge* difference.

Let's find out how…

2.2. Without SPDY

As we explore SPDY, we will need to be able to compare it with vanilla HTTP and, later, alternative protocols. The Speed Tracer extension for Chrome produces some very nice graphs for just this purpose.

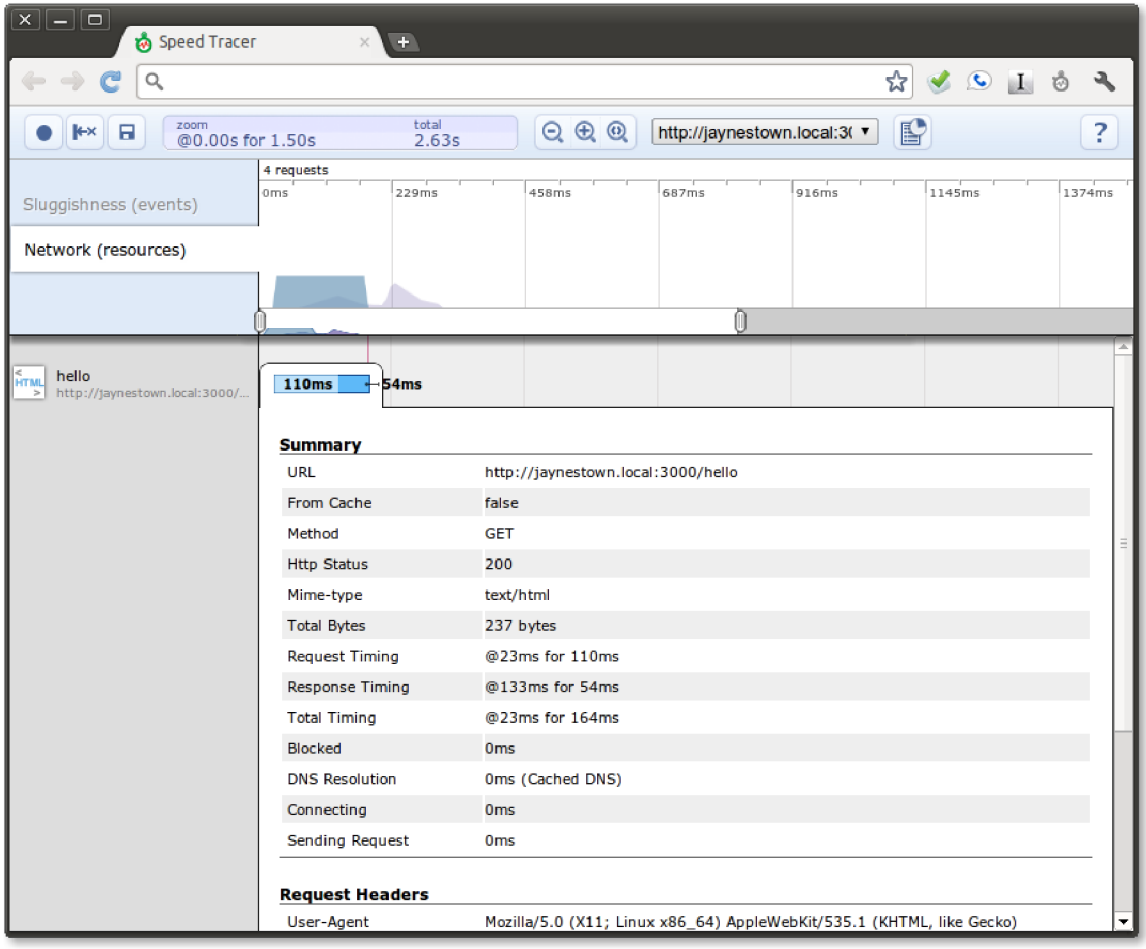
The `hello world` app *without* SPDY looks like this in Speed Tracer:



Speed Tracer presents a nice overview of just about anything that might affect page load time including: network overhead, parsing of CSS/HTML/Javascript, rendering, even internal garbage collection.

From this Speed Tracer overview, we see that most of the time is spent receiving data from the server. It looks as though even more time is spent waiting for the data in the first place —it only takes 41ms to process the data, but there is almost 100ms before data is even seen here. What gives?

The answer to that can be found in the "Network (resources)" tab of Speed Tracer:



The request took 110ms while the response took another 54ms. Why so long? Network latency is causing a delay here. Real world connections are not instantaneous and SPDY is optimized for the real world.

Local connections usually have round trip times of a couple milliseconds. Great connections on the internet are on the order of tens of milliseconds. Round trip times of 100ms are not great, but also not uncommon.

Note

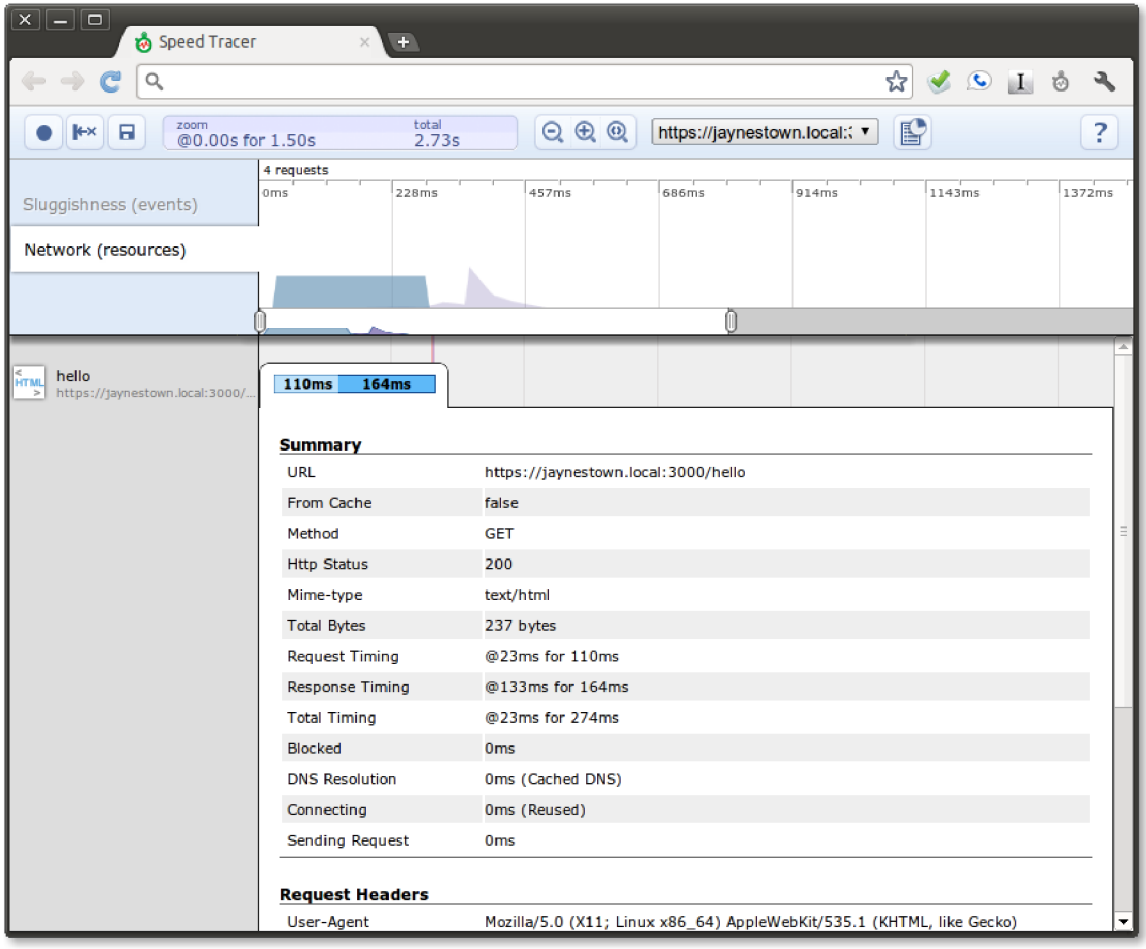
Unless otherwise stated, all examples in this book will include a one-way latency of 50ms / round trip latency of 100ms.

What this tells us is that the biggest problem facing our `hello world` app is network latency.

Surely SPDY can help! Let's see what SPDY is capable of…

2.3. The SPDY Difference

After adding `require('express-spdy')` and `NPNProtocols: ['spdy/2', 'http/1.1']` to convert the express.js app to express-spdy, Speed Tracer reports:



Now wait a second. In both the vanilla HTTP version and SPDY version of the application, the request takes 110ms. But the SPDY response takes 133ms vs 54ms in vanilla HTTP. The response time in SPDY is more than double?

No doubt you are really beginning to question the wisdom of reading this book, but have no fear. Speedy is just around the bend.

Note

The slowness in response time is due entirely to a small penalty paid in the very first frame of a SPDY session. The first frame contains compression initialization

that increases the compressed frame beyond the size of the original. Once we hit the Advanced Topics section, we will see why this is and why the penalty is very, very much worth the price.

Mostly, the slower response time boils down to an infallible truism…

2.4. Hello World Applications Are Silly

The hello world application in this book is no exception.

SPDY is overkill for simple sites like this `hello world` application, documentation repositories, or marketing sites. But if you are planning on doing anything more complex than this, SPDY will give you a huge win.

Tip

Lesson #3: SPDY is not optimized for very small applications.

In the next chapter, we will get our first taste for the power of SPDY as we add some images and simple Javascript to our hello world application.

In this chapter, we saw our first SPDY application. Happily, SPDY apps are not that much different than the HTTP apps with which web developers are familiar. To be sure there will be much to learn about writing SPDY apps, but at least we do not have to start over completely.

We also learned that SPDY is served over SSL. Always. Finally, we found that SPDY is definitely *not* meant for simple `hello world`. To find out what kind of applications SPDY is good for, keep reading. Things are going to get a lot more interesting.