

A Better Structure For Express/node.js Apps

Dec 29, 2017

If you'd like to start a new Express app, there are thousands of articles on how to do so. Most of them, targeting beginners, include all the code in one or two files. Although, technically correct, these articles are quite lax on good practices. In no particular order, I'd like to address the address a couple of them.

App Start Up Issues

The API/web endpoint accepts requests before a database connection was established. Usually, the offending code would be structured like this:

```
// app.js

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

// ...
mongoose.connect('mongodb://localhost/db');

// ...
app.use(require('./routes'));

// ...
var server = app.listen( process.env.PORT || 3000, function(){
  console.log(`Listening on port ${server.address().port}`);
});
```

Because, in node.js, almost all the calls are asynchronously, your app will start listening for connections before the database connection was established. Although this might not be a problem when developing the app, it's a different story when deployed in production. A fast start up time might be 300-500 ms, connecting to redis and mongodb, might take 600 ms or more. Are you sure there are no incoming connections in those 300-600 ms your app takes to start?

A minimal fix, would delay the listening for connections part until the database connection was established.

```
mongoose.connection.on('open', function(){
  var server = app.listen( process.env.PORT || 3000, function(){
    console.log(`Listening on port ${server.address().port}`);
  });
});
```

No Setup For Connecting to Multiple Databases/Services

Most examples connect to zero

```
app.get('/', function(req, res, next){
  return res.send('Hello World!');
});
```

or one database/service.

```
// routes/articles.js
var router = require('express').Router();
var mongoose = require('mongoose');
var Article = mongoose.model('Article');

router.get('/', auth.optional, function(req, res, next) {
  Article.find({}, function(err, articles){
    if (err) {
      return next(err);
    }
    return res.json(articles || []);
  });
});
```

These are fine, simple examples, however I'd like to make them a bit more complex. First, your router should not require your model. If really what to do that query in your router, at minimum, you should pass in the router and model as a dependency.

```
// routes/articles.js
module.exports = function(dependencies) {
  var router = dependencies.router;
  var Article = dependencies.Article;
  // var redisClient = dependencies.redisClient;

  router.get('/', function(req, res, next) {
    Article.find({}, function(err, articles){
      if (err) {
```

```

        return next(err);
    }
    return res.json(articles || []);
  });
});
};

// app.js
var mongoose = require('mongoose');
var dependencies = {
  router: require('express').Router(),
  Article: mongoose.model('Article')
};
var articlesRoute = require('routes/articles')(dependencies);
app.use(articlesRoute);

```

If you're inclined to, you'd have a chance to mock router and/or the model, to test them independently.

If one would like to go further, model and/or the redis client can be further refactored into a `store.js` file that will handle the databases interaction. For example, return a cached version of the articles, and only if none is found, query the database. Maybe also store the last result for 10 seconds.

```

// lib/store.js
module.exports = function(dependencies) {
  var Article = dependencies.Article;
  var redisClient = dependencies.redisClient;

  function articles(callback) {
    return redisClient.get('articles', function(err, items){
      // if err ... unhandled
      if (items) {
        return callback(err, items);
      }
      Article.find({}, function(err, articles){
        // if err ...
        var ttl = 10; // expire cache after 10 seconds
        return redisClient.set('articles', articles, 'ex', ttl,
          function(err, result){
            return callback(err, articles);
          });
      });
    });
  }
};

```

```

    return {
      articles: articles
    }
  }
}

// app.js
var Article; // from require or dependencies
var redisClient;

var store = require('lib/store')({
  Article,
  redisClient
});

var articlesRoute = require('routes/articles')({
  router: require('express').Router(),
  store
});

// routes/articles.js
module.exports = function(dependencies) {
  var store = dependencies.store;

  router.get('/', function(req, res, next) {
    return store.articles(function(err, items){
      if (err) {
        return next(err);
      }
      return res.json(articles || []);
    })
  })
}

```

To write a unit test for the above code, one would start with `store.js`. The npm packages `mockgoose` and `redis-js` could be used to setup an in-memory database for mongo and redis.

Using the same technique, one can inject any type of service that requires configuration and setup (eg. RabbitMQ, Push Notifications Services, websockets). If at a later point, one would like to replace *redis* with *memcache*, it shouldn't be a `require('redis')` in every file.

Tight Code Coupling

Or as seen in almost every online example `require` the same local or external module in every file.

```
// router.js
var mongoose = require('mongoose');
var Article = mongoose.model('Article');
```

can be replaced with

```
// router.js
module.exports = function(dependencies) {
  var Article = dependencies.Article;
  // var models = dependencies.models; // or better yet
}
```

The same can be applied to app.js.

```
// app.js
module.exports = function(dependencies) {
  var config = dependencies.config;
  var models = dependencies.models;
  var redisClient = dependencies.redisClient;

  var app = express();

  // ...
  return app;
}
```

And even to config.js. Beside all the connection strings (mongo, redis, etc), one can use this file to assign a release version to the current app. This maybe be useful if one needs to pass in that info to other services, like sentry.io.

```
// config.js
module.exports = function(dependencies) {
  var process = dependencies.process;
  var os = dependencies.os;
  var p = require('../package.json'); // or from dependencies

  var config = {
    port: process.env.NODE_PORT || 3000,
    hostname: process.env.NODE_PORT || os.hostname(),
    mongoUrl: process.env.NODE_MONGO_URL || 'mongodb://localhost/db',
    release: p.name + '-' + p.version
  };
};
```

```
    return config;
}
```

Using the presented building blocks, when setting up an app, a start module can be build, so all the services start only after the required connections were established.

```
// lib/start.js
module.exports = function(dependencies) {
    var config = dependencies.config;
    var process = dependencies.process;
    var mongoose = dependencies.mongoose;

    function redisOnConnect(startTime, mongooseOnOpen) {
        return function redisOnConnectFunc() {
            console.log(`Redis connected after ${new Date() - startTime} ms.`);

            mongoose.connect(config.mongoUrl, {
                useMongoClient: true
            });
            mongoose.connection.on('error', onError);
            mongoose.connection.on('disconnected', function(){
                debug('Mongoose connection disconnected');
                process.exit();
            });
            mongoose.connection.on('open', mongooseOnOpen(config, startTime));
        };
    };

    function onError(error) {
        console.log('Connection error: ' + err);
        process.exit();
    }

    function serverOnListening(startTime, server) {
        return function onListeningFunc() {
            var address = server.address();
            console.log(`Server listening on ${address.address}:${address.port}`);
            console.log(`Start time: ${new Date() - startTime} ms.`);
        };
    }

    return {
        redisOnConnect,
        onError,
        serverOnListening
    };
};
```

```

}

// bin/service.js
var mongoose = require('mongoose');
var config = require('lib/config')({
  process,
  os: require('os')
});
var start = require('lib/start')({
  config,
  process,
  mongoose
});
function mongooseOnOpen(config, startTime) {
  return function mongooseOnOpenFunc() {
    console.log(`Mongoose connection open in ${new Date() - startTime}`);

    var app = require('../app')({
      config,
      store
    });

    var http = require('http');
    var server = http.createServer(app);

    server.listen(config.port);
    server.on('error', start.onError);
    server.on('listening', start.serverOnListening(startTime, server));
  };
};

var redisClient = redis.client(config);
redisClient.on('ready', start.redisOnConnect(startTime, mongooseOnOpen))

```

It is a good practise that when writing a new service, one should answer the question, how should this new code be tested? Today, all the cool kids use *mongo*, but tomorrow, they go hipster and switch to *FoxPro*.

OpenSSL s_client vs PHP stream_socket_client

Mar 20, 2016

We, as developers, craft bespoke apps that connect to remote servers, over encrypted connections. Those connections are authenticated using API keys, username and passwords or public certificate and private keys. Sometimes, those connections work out of the box, sometimes we'll spend 6 hours figuring out what's wrong.

The easiest way to establish a plain text connection from the terminal to a server, is to use telnet

```
telnet google.com 80
```

then issue a HTTP command like `GET /` and you'll get Google's homepage, which will be a redirect to a HTTPS version of the same page.

To establish a SSL connection, you can use the `openssl` Swiss knife.

```
openssl s_client -connect google.com:443
```

and then issue the same HTTP command `GET /` and you'll get the HTTPS version of the Google homepage.

OpenSSL can be used to open a connection that requires certificate authentication too, just supply those as CLI options. This way you can test that the Apple Push Notification connectivity, by using your developer certificate and private key. Just make sure that they are converted to the PEM file format first.

```
openssl s_client -connect gw.example.com:1234 -cert example.com.cert -ke
```

After you've figured out which developer certificate and which private key are still active and match your remote server, you can establish the same connection from PHP.

```
$opts = array(
    'ssl' => array(
        'local_cert' => 'example.com.cert',
        'local_pk' => 'example.com.key',
        'cafile' => 'example.cacert',
        'verify_peer' => true,
    )
);

$timeout = 160;
$host = "ssl://gw.example.com:1234";

echo "Connecting\n";
```



```

$context = stream_context_create($opts);
$socket = stream_socket_client (
    $host, $errno, $errstr, $timeout,
    STREAM_CLIENT_CONNECT, $context);

if (!$socket) {
    echo "Failure $errno errstr $errstr.\n";
} else {
    echo "Success\n";
}

```

In most of the cases, after running this script, the output will be `Success` .

However, if you're unlucky, you'll get the following output

```

Connecting
PHP Warning:  stream_socket_client(): SSL operation failed with code 1.
error:14090086:SSL routines:SSL3_GET_SERVER_CERTIFICATE:certificate veri
PHP Warning:  stream_socket_client(): Failed to enable crypto in /path/t
PHP Warning:  stream_socket_client(): unable to connect to ssl://gw.exam
errno 0 errstr .

```

After you double check several times, you reach the conclusion that you're using the same certificate, key and CA certificate as above. You'll start wondering if there is a difference when the connection is made using `openssl s_client` and PHP's `stream_context_create` and `stream_socket_client` (or `curl`).

One of the ways to get a glimpse into what each program is doing, is to use [DTrace](#) if you're lucky enough to use one of the OSes where's available (like MacOS X, FreeBSD or SmartOS) or to use its poor friend [strace](#).

```

strace php example.php 2> example.php.log
strace openssl s_client -connect gw.example.com:1234 -cert example.com.c

```

After a careful check of those file's output, you'll notice that OpenSSL and PHP are using two different CA stores. Open SSL's `/usr/lib/ssl/certs/` , versus PHP's `/usr/share/ca-certificates/mozilla/` . You can spot this by checking what happens before getting the `stream_socket_client(): SSL operation failed with code 1.` error message:

PHP's trace:

```

stat("/usr/share/ca-certificates/mozilla//6b99d060.0", 0x7ffded049c80) =

```

OpenSSL's trace:

```
stat("/usr/lib/ssl/certs/6b99d060.0", {st_mode=S_IFREG|0644, st_size=164
open("/usr/lib/ssl/certs/6b99d060.0", O_RDONLY) = 4
```

After figuring this little difference, it's pretty easy to fix the PHP code. Change the `stream_context_create` to include the new CA path:

```
$opts = array(
    'ssl' => array(
        'local_cert' => 'example.com.cert',
        'local_pk' => 'example.com.key',
        'cafile' => 'example.cacert',
        'capath' => '/usr/lib/ssl/certs/',
        'verify_peer' => true,
    )
);
```

Tip o' the hat to Ben Hearsurn's [Python and SSL Certificate Verification](#) article for suggesting to use strace to figure out why do we get different results, while using the same certificates.

References:

- [Python and SSL Certificate Verification](#)
- [PHP's stream_context_create SSL context options](#)
- [Connection to URL using OpenSSL client works, but curl fails](#)

Skagen 597LSLB

May 29, 2015

A couple of years ago, I bought a nice Danish watch, Skagen 597LSLB. It wasn't a very expensive watch, but it was a pretty one. At that time, I paid about €120 for it.

Fast forward a couple of years later, some parts of the watch do not seem to be that solid. The leather band started to peel off, like an onion. It's quite unfortunate that this happened, having in mind that I did not wear this watch every day, since I got it, but only about half of those days. We all know that things are not built as they used to be and that we're living in a consumerist world, so nothing new so far. One would assume that getting a leather band replaced, shouldn't be a big deal though. However, it seems that the folks running Skagen could not be bothered to sell any watch accessories anymore. Since they do not offer a replacement on their site, your only option is eBay or a third party company.

A couple of days later, the battery ran out (for the third time since I got the watch). That's fine, they are not designed to last forever. Let's check Skagen's site again, to find out what battery should I buy. Again, I was out of luck, as I could not find that info online.



To save others the trouble of researching this bit, I've written this blog post. The Skagen 597LSLB watch model is using a SR616SW/Renata 321/Energizer 321 cell.

Here is a photo of the watch's internals, in case you're curious about the built quality.



Marius' Square

Marius Ursache
marius@marius.me.uk

 [bamse16](#)
 [bamse](#)

Software developer, working with firefighters to help manage incidents easier.