Build a Complete MVC Website With ExpressJS

Krasimir Tsonev on Aug 15th 2013 with 58 Comments

Tutorial Details

-
- **Difficulty**: Intermediate
- **Completion Time**: 60 Minutes

In this article we'll be building a complete website with a front-facing client side, as well as a control panel for managing the site's content. As you may guess, the final working version of the application contains a lot of different files. I wrote this tutorial step by step, following the development process, but I didn't include every single file, as that would make this a very long and boring read. However, the source code is available on GitHub and I strongly recommend that you take a look.

Introduction

Express is one of the best frameworks for Node. It has great support and a bunch of helpful features. There are a lot of great articles out there, which cover all of the basics. However, this time I want to dig in a little bit deeper and share my workflow for creating a complete website. In general, this article is not only for Express, but for using it in combination with some other great tools that are available for Node developers.

I assume that you are familiar with Nodejs, have it installed on your system, and that you have probably built some applications with it already.

At the heart of Express is Connect. This is a middleware framework, which comes with a lot of useful stuff. If you're wondering what exactly a middleware is, here is a quick example:

```
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);
```

Middleware is basically a function which accepts `request` and `response` objects and a `next` function. Each middleware can decide to respond by using a `response` object or pass the flow to the next function by calling the `next` callback. In the example above, if you remove the `next()` method call in the second middleware, the `hello world` string will never be sent to the browser. In general, that's how Express works. There are some predefined middlewares, which of course, save you a lot of time. Like for example, `Body parser` which parses request bodies and supports application/json, application/x-www-form-urlencoded, and multipart/form-data. Or the `Cookie parser`, which parses cookie headers and populates `req.cookies` with an object keyed by the cookie's name.

Express actually wraps Connect and adds some new functionality around it. Like for example, routing logic, which makes the process much smoother. Here's an example of handling a GET request:

```
app.get('/hello.txt', function(req, res){
    var body = 'Hello World';
    res.setHeader('Content-Type', 'text/plain');
    res.setHeader('Content-Length', body.length);
    res.end(body);
});
```

Setup

There are two ways to setup Express. The first one is by placing it in your `package.json` file and running `npm install` (there's a joke that `npm` means *no problem man* :)).

```
{
    "name": "MyWebSite",
    "description": "My website",
    "version": "0.0.1",
    "dependencies": {
        "express": "3.x"
    }
}
```

The framework's code will be placed in `node_modules` and you will be able to create an instance of it. However, I prefer an alternative option, by using the command line tool. Just install Express globally with `npm install -g express`. By doing this, you now have a brand new CLI instrument. For example if you run:

```
express --sessions --css less --hogan app
```

Express will create an application skeleton with a few things already configured for you. Here are the usage options for the `express(1)` command:

```
Usage: express [options]
Options:
  -h, --help          output usage information
  -V, --version       output the version number
  -s, --sessions      add session support
  -e, --ejs           add ejs engine support (defaults to jade)
  -J, --jshtml        add jshtml engine support (defaults to jade)
  -H, --hogan         add hogan.js engine support
  -c, --css   add stylesheet  support (less|stylus) (defaults to p
  -f, --force         force on non-empty directory
```

As you can see, there are just a few options available, but for me they are enough. Normally I'm using less as the CSS preprocessor and hogan as the templating engine. In this example, we will also need session support, so the `--sessions` argument solves that problem. When the above command finishes, our project looks like the following:

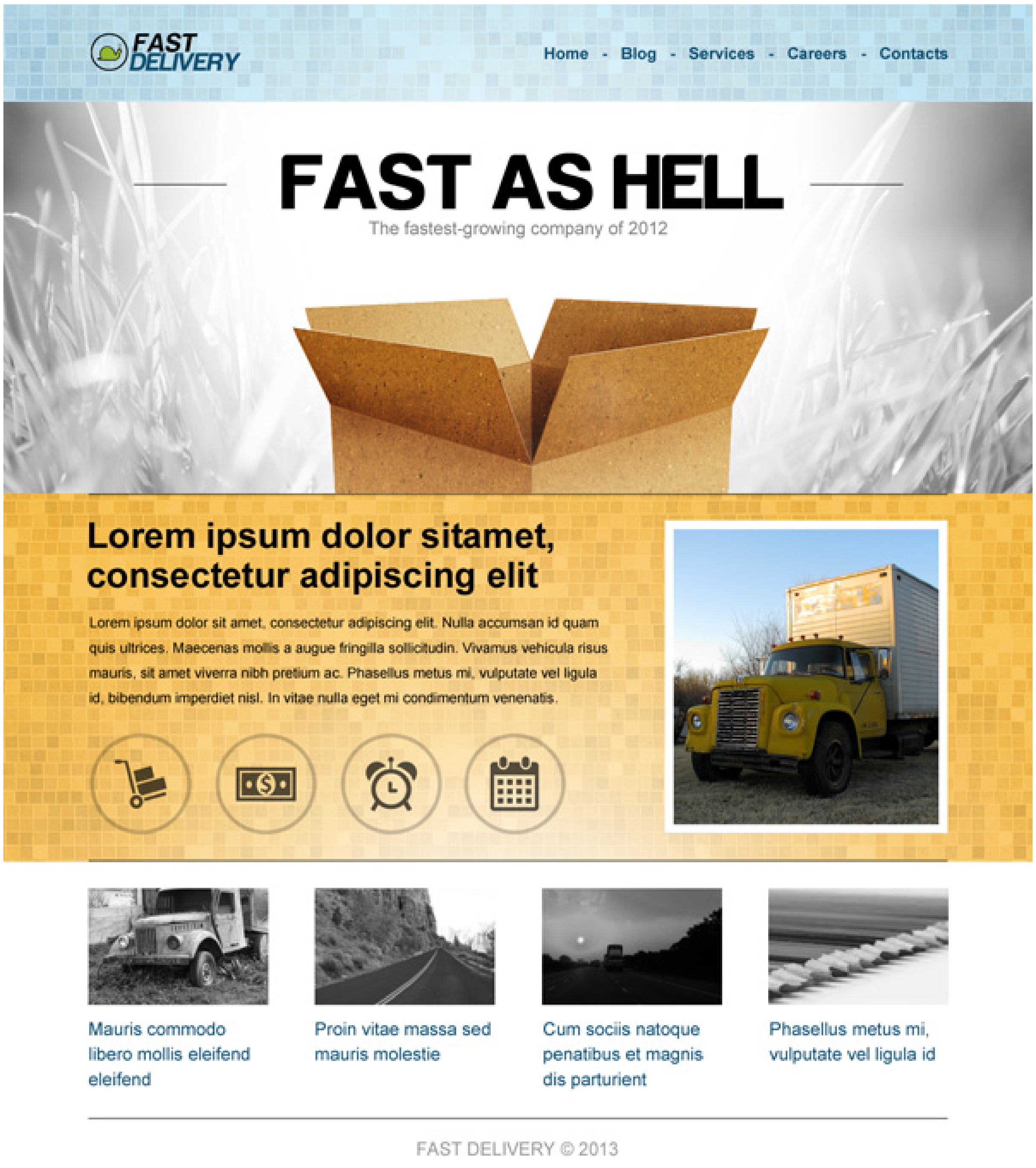
```
/public
    /images
    /javascripts
    /stylesheets
/routes
    /index.js
    /user.js
/views
    /index.hjs
/app.js
/package.json
```

If you check out the `package.json` file, you will see that all the dependencies which we need are added here. Although they haven't been installed yet. To do so, just run `npm install` and then a `node_modules` folder will pop up.

I realize that the above approach is not always appropriate. You may want to place your route handlers in another directory or something something similar. But, as you'll see in the next few chapters, I'll make changes to the already generated structure, which is pretty easy to do. So you should just think of the `express(1)` command as a boilerplate generator.

FastDelivery

For this tutorial, I designed a simple website of a fake company named FastDelivery. Here's a screenshot of the complete design:


FAST DELIVERY
Home - Blog - Services - Careers - Contacts
FAST AS HELL
The fastest-growing company of 2012
Lorem ipsum dolor sitamet, consectetur adipiscing elit
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla accumsan id quam quis ultrices. Maecenas mollis a augue fringilla sollicitudin. Vivamus vehicula risus mauris, sit amet viverra nibh pretium ac. Phasellus metus mi, vulputate vel ligula id, bibendum imperdiet nisl. In vitae nulla eget mi condimentum venenatis.
Mauris commodo libero mollis eleifend eleifend
Proin vitae massa sed mauris molestie
Cum sociis natoque penatibus et magnis dis parturient
Phasellus metus mi, vulputate vel ligula id
FAST DELIVERY © 2013

At the end of this tutorial, we will have a complete web application, with a working control panel. The idea is to manage every part of the site in separate restricted areas. The layout was created in Photoshop and sliced to CSS(less) and HTML(hogan) files. Now, I'm not going to be covering the slicing process, because it's not the subject of this article, but if you have any questions regarding this, don't hesitate to ask. After the slicing, we have the following files and app structure:

```
/public
```

```
    /images (there are several images exported from Photoshop)
    /javascripts
    /stylesheets
        /home.less
        /inner.less
        /style.css
        /style.less (imports home.less and inner.less)
/routes
    /index.js
/views
    /index.hjs (home page)
    /inner.hjs (template for every other page of the site)
/app.js
/package.json
```

Here is a list of the site's elements that we are going to administrate:

- Home (the banner in the middle – title and text)
- Blog (adding, removing and editing of articles)
- Services page
- Careers page
- Contacts page

Configuration

There are a few things that we have to do before we can start the real implementation. The configuration setup is one of them. Let's imagine that our little site should be deployed to three different places – a local server, a staging server and a production server. Of course the settings for every environment are different and we should implement a mechanism which is flexible enough. As you know, every node script is run as a console program. So, we can easily send command line arguments which will define the current environment. I wrapped that part in a separate module in order to write a test for it later. Here is the `/config/index.js` file:

```
var config = {
    local: {
        mode: 'local',
        port: 3000
    },
    staging: {
        mode: 'staging',
        port: 4000
    },
    production: {
        mode: 'production',
        port: 5000
    }
}
module.exports = function(mode) {
    return config[mode || process.argv[2] || 'local'] || config.lo
}
```

There are only two settings (for now) – `mode` and `port`. As you may guess, the application uses different ports for the different servers. That's why we have to update the entry point of the site, in `app.js`.

```
...
var config = require('./config')();
...
http.createServer(app).listen(config.port, function(){
    console.log('Express server listening on port ' + config.port);
});
```

To switch between the configurations, just add the environment at the end. For example:

```
node app.js staging
```

Will produce:

```
Express server listening on port 4000
```

Now we have all our settings in one place and they are easily manageable.

Tests

I'm a big fan of TDD. I'll try to cover all the base classes used in this article. Of course, having tests for absolutely everything will make this writing too long, but in general, that's how you should proceed when creating your own apps. One of my favorite frameworks for testing is jasmine. Of course it's available in the npm registry:

```
npm install -g jasmine-node
```

Let's create a `tests` directory which will hold our tests. The first thing that we are going to check is our configuration setup. The spec files must end with `.spec.js`, so the file should be called `config.spec.js`.

```
describe("Configuration setup", function() {
    it("should load local configurations", function(next) {
        var config = require('../config')();
        expect(config.mode).toBe('local');
        next();
    });
    it("should load staging configurations", function(next) {
        var config = require('../config')('staging');
        expect(config.mode).toBe('staging');
        next();
    });
    it("should load production configurations", function(next) {
        var config = require('../config')('production');
        expect(config.mode).toBe('production');
        next();
    });
});
```

Run `jasmine-node ./tests` and you should see the following:

```
Finished in 0.008 seconds
3 tests, 6 assertions, 0 failures, 0 skipped
```

This time, I wrote the implementation first and the test second. That's not exactly the TDD way of doing things, but over the next few chapters I'll do the opposite.

I strongly recommend spending a good amount of time writing tests. There is nothing better than a fully tested application.

A couple of years ago I realized something very important, which may help you to produce better programs. Each time you start writing a new class, a new module, or just a new piece of logic, ask yourself:

> How can I test this?

The answer to this question will help you to code much more efficiently, create better APIs, and put everything into nicely separated blocks. You can't write tests for spaghetti code. For example, in the configuration file above (`/config/index.js`) I added the possibility to send the `mode` in the module's constructor. You may wonder, why do I do that when the main idea is to get the mode from the command line arguments? It's simple ... because I needed to test it. Let's imagine that one month later I need to check something in a `production` configuration, but the node script is run with a `staging` parameter. I won't be able to make this change without that little improvement. That one previous little step now actually prevents problems in the future.

Database

Since we are building a dynamic website, we need a database to store our data in. I chose to use mongodb for this tutorial. Mongo is a NoSQL document database. The installation instructions can be found here and because I'm a Windows user, I followed the Windows installation instead. Once you finish with the installation, run the MongoDB daemon, which by default listens on port `27017`. So, in theory, we should be able to connect to this port and communicate with the mongodb server. To do this from a node script, we need a mongodb module/driver. If you downloaded the source files for this tutorial, the module is already added in the `package.json` file. If not, just add `"mongodb": "1.3.10"` to your dependencies and run `npm install`.

Next, we are going to write a test, which checks if there is a mongodb server running. `/tests/mongodb.spec.js` file:

```
describe("MongoDB", function() {
    it("is there a server running", function(next) {
        var MongoClient = require('mongodb').MongoClient;
        MongoClient.connect('mongodb://127.0.0.1:27017/fastdelivery
            expect(err).toBe(null);
            next();
        });
    });
});
```

The callback in the `.connect` method of the mongodb client receives a `db` object. We will use it later to manage our data, which means that we need access to it inside our models. It's not a good idea to create a new `MongoClient` object every time when we have to make a request to the database. That's why I moved the running of the express server inside the callback of the `connect` function:

```
MongoClient.connect('mongodb://127.0.0.1:27017/fastdelivery', func
    if(err) {
        console.log('Sorry, there is no mongo db server running.')
    } else {
        var attachDB = function(req, res, next) {
            req.db = db;
            next();
        };
        http.createServer(app).listen(config.port, function(){
            console.log('Express server listening on port ' + conf
        });
    }
});
```

Even better, since we have a configuration setup, it would be a good idea to place the mongodb host and port in there and then change the connect URL to:

```
'mongodb://' + config.mongo.host + ':' + config.mongo.port + '/fast
```

Pay close attention to the middleware: `attachDB`, which I added just before the call to the `http.createServer` function. Thanks to this little addition, we will populate a `.db` property of the request object. The good news is that we can attach several functions during the route definition. For example:

```
app.get('/', attachDB, function(req, res, next) {
    ...
})
```

So with that, Express calls `attachDB` beforehand to reach our route handler. Once this happens, the request object will have the `.db` property and we can use it to access the database.

MVC

We all know the MVC pattern. The question is how this applies to Express. More or less, it's a matter of interpretation. In the next few chapters I'll create modules, which act as a model, view and controller.

Model

The model is what will be handling the data that's in our application. It should have access to a `db` object, returned by `MongoClient`. Our model should also have a method for extending it, because we may want to create different types of models. For example, we might want a `BlogModel` or a `ContactsModel`. So we need to write a new spec: `/tests/base.model.spec.js`, in order to test these two model features. And remember, by defining these functionalities before we start coding the implementation, we can guarantee that our module will do only what we want it to do.

```
var Model = require("../models/Base"),
    dbMockup = {};
describe("Models", function() {
    it("should create a new model", function(next) {
        var model = new Model(dbMockup);
        expect(model.db).toBeDefined();
        expect(model.extend).toBeDefined();
        next();
    });
    it("should be extendable", function(next) {
        var model = new Model(dbMockup);
        var OtherTypeOfModel = model.extend({
            myCustomModelMethod: function() { }
        });
        var model2 = new OtherTypeOfModel(dbMockup);
        expect(model2.db).toBeDefined();
        expect(model2.myCustomModelMethod).toBeDefined();
        next();
    })
});
```

Instead of a real `db` object, I decided to pass a mockup object. That's because later, I may want to test something specific, which depends on information coming from the database. It will be much easier to define this data manually.

The implementation of the `extend` method is a little bit tricky, because we have to change the prototype of `module.exports`, but still keep the original constructor. Thankfully, we have a nice test already written, which proves that our code works. A version which passes the above, looks like this:

```
module.exports = function(db) {
    this.db = db;
};
module.exports.prototype = {
    extend: function(properties) {
        var Child = module.exports;
        Child.prototype = module.exports.prototype;
        for(var key in properties) {
            Child.prototype[key] = properties[key];
        }
        return Child;
    },
    setDB: function(db) {
        this.db = db;
    },
    collection: function() {
        if(this._collection) return this._collection;
        return this._collection = this.db.collection('fastdelivery
    }
}
```

Here, there are two helper methods. A setter for the `db` object and a getter for our database `collection`.

View

The view will render information to the screen. Essentially, the view is a class which sends a response to the browser. Express provides a short way to do this:

```
res.render('index', { title: 'Express' });
```

The response object is a wrapper, which has a nice API, making our life easier. However, I'd prefer to create a module which will encapsulate this functionality. The default `views` directory will be changed to `templates` and a new one will be created, which will host the `Base` view class. This little change now requires another change. We should notify Express that our template files are now placed in another directory:

```
app.set('views',   dirname + '/templates');
```

First, I'll define what I need, write the test, and after that, write the implementation. We need a module matching the following rules:

- Its constructor should receive a response object and a template name.
- It should have a `render` method which accepts a data object.

- It should be extendable.

You may wonder why I'm extending the `View` class. Isn't it just calling the `response.render` method? Well in practice, there are cases in which you will want to send a different header or maybe manipulate the `response` object somehow. Like for example, serving JSON data:

```
var data = {"developer": "Krasimir Tsonev"};
response.contentType('application/json');
response.send(JSON.stringify(data));
```

Instead of doing this every time, it would be nice to have an `HTMLView` class and a `JSONView` class. Or even an `XMLView` class for sending XML data to the browser. It's just better, if you build a large website, to wrap such functionalities instead of copy–pasting the same code over and over again.

Here is the spec for the `/views/Base.js`:

```
var View = require("../views/Base");
describe("Base view", function() {
    it("create and render new view", function(next) {
        var responseMockup = {
            render: function(template, data) {
                expect(data.myProperty).toBe('value');
                expect(template).toBe('template-file');
                next();
            }
        }
        var v = new View(responseMockup, 'template-file');
        v.render({myProperty: 'value'});
    });
    it("should be extendable", function(next) {
        var v = new View();
        var OtherView = v.extend({
            render: function(data) {
                expect(data.prop).toBe('yes');
                next();
            }
        });
        var otherViewInstance = new OtherView();
        expect(otherViewInstance.render).toBeDefined();
        otherViewInstance.render({prop: 'yes'});
    });
});
```

In order to test the rendering, I had to create a mockup. In this case, I created an object which imitates the Express's response object. In the second part of the test, I created another View class which inherits the base one and applies a custom render method. Here is the `/views/Base.js` class.

```
module.exports = function(response, template) {
    this.response = response;
    this.template = template;
};
module.exports.prototype = {
    extend: function(properties) {
        var Child = module.exports;
        Child.prototype = module.exports.prototype;
        for(var key in properties) {
            Child.prototype[key] = properties[key];
        }
        return Child;
    },
    render: function(data) {
        if(this.response && this.template) {
            this.response.render(this.template, data);
        }
    }
}
```

Now we have three specs in our `tests` directory and if you run `jasmine-node ./tests` the result should be:

```
Finished in 0.009 seconds
7 tests, 18 assertions, 0 failures, 0 skipped
```

Controller

Remember the routes and how they were defined?

```
app.get('/', routes.index);
```

The `'/'` after the route, which in the example above, is actually the controller. It's just a middleware function which accepts `request`, `response` and `next`.

```
exports.index = function(req, res, next) {
    res.render('index', { title: 'Express' });
};
```

Above, is how your controller should look, in the context of Express. The `express(1)` command line tool creates a directory named `routes`, but in our case, it is better for it to be named `controllers`, so I changed it to reflect this naming scheme.

Since we're not just building a teeny tiny application, it would be wise if we created a base class, which we can extend. If we ever need to pass some kind of functionality to all of our controllers, this base class would be the perfect place. Again, I'll write the test first, so let's define what we need:

- it should have an`extend` method, which accepts an object and returns a new child instance
- the child instance should have a `run` method, which is the old middleware function
- there should be a `name` property, which identifies the controller
- we should be able to create independent objects, based on the class

So just a few things for now, but we may add more functionality later. The test would look something like this:

```
BaseController = require("../controllers/Base");
:ribe("Base controller", function() {
it("should have a method extend which returns a child instance", functio
    expect(BaseController.extend).toBeDefined();
    var child = BaseController.extend({ name: "my child controller" });
    expect(child.run).toBeDefined();
    expect(child.name).toBe("my child controller");
    next();
});
it("should be able to create different childs", function(next) {
    var childA = BaseController.extend({ name: "child A", customProperty
    var childB = BaseController.extend({ name: "child B" });
    expect(childA.name).not.toBe(childB.name);
    expect(childB.customProperty).not.toBeDefined();
    next();
});
```

And here is the implementation of `/controllers/Base.js`:

```
var _ = require("underscore");
module.exports = {
    name: "base",
    extend: function(child) {
        return _.extend({}, this, child);
    },
    run: function(req, res, next) {

    }
}
```

Of course, every child class should define its own `run` method, along with its own logic.

FastDelivery Website

Ok, we have a good set of classes for our MVC architecture and we've covered our newly created modules with tests. Now we are ready to continue with the site, of our fake company, `FastDelivery`. Let's imagine that the site has two parts – a front-end and an administration panel. The front-end will be used to display the information written in the database to our end users. The admin panel will be used to manage that data. Let's start with our admin (control) panel.

Control Panel

Let's first create a simple controller which will serve as the administration page. `/controllers/Admin.js` file:

```
var BaseController = require("./Base"),
    View = require("../views/Base");
module.exports = BaseController.extend({
    name: "Admin",
    run: function(req, res, next) {
        var v = new View(res, 'admin');
        v.render({
            title: 'Administration',
            content: 'Welcome to the control panel'
        });
    }
});
```

By using the pre-written base classes for our controllers and views, we can easily create the entry point for the control panel. The `View` class accepts a name of a template file. According to the code above, the file should be called `admin.hjs` and should be placed in `/templates`. The content would look something like this:

```
<!DOCTYPE html>
<html>
    <head>
        <title>{{ title }}</title>
        <link rel='stylesheet' href='/stylesheets/style.css' />
    </head>
    <body>
        <div class="container">
            <h1>{{ content }}</h1>
        </div>
    </body>
</html>
```

(In order to keep this tutorial fairly short and in an easy to read format, I'm not going to show every single view template. I strongly recommend that you download the source code from GitHub.)

Now to make the controller visible, we have to add a route to it in `app.js`:

```
var Admin = require('./controllers/Admin');
...
var attachDB = function(req, res, next) {
    req.db = db;
    next();
};
...
app.all('/admin*', attachDB, function(req, res, next) {
    Admin.run(req, res, next);
});
```

Note that we are not sending the `Admin.run` method directly as middleware. That's because we want to keep the context. If we do this:

```
app.all('/admin*', Admin.run);
```

the word `this` in `Admin` will point to something else.

Protecting the Administration Panel

Every page which starts with `/admin` should be protected. To achieve this, we are going to use Express's middleware: Sessions. It simply attaches an object to the request called `session`. We should now change our Admin controller to do two additional things:

- It should check if there is a session available. If not, then display a login form.
- It should accept the data sent by the login form and authorize the user if the username and password match.

Here is a little helper function we can use to accomplish this:

```
authorize: function(req) {
    return (
        req.session &&
        req.session.fastdelivery &&
        req.session.fastdelivery === true
    ) || (
        req.body &&
        req.body.username === this.username &&
        req.body.password === this.password
    );
}
```

First, we have a statement which tries to recognize the user via the session object. Secondly, we check if a form has been submitted. If so, the data from the form is available in the `request.body` object which is filled by the `bodyParser` middleware. Then we just check if the username and password matches.

And now here is the `run` method of the controller, which uses our new helper. We check if the user is authorized, displaying either the control panel itself, otherwise we display the login page:

```
run: function(req, res, next) {
    if(this.authorize(req)) {
        req.session.fastdelivery = true;
        req.session.save(function(err) {
            var v = new View(res, 'admin');
            v.render({
                title: 'Administration',
                content: 'Welcome to the control panel'
            });
        });
    } else {
        var v = new View(res, 'admin-login');
        v.render({
            title: 'Please login'
        });
    }
}
```

Managing Content

As I pointed out in the beginning of this article we have plenty of things to administrate. To simplify the process, let's keep all the data in one collection. Every record will have a `title`, `text`, `picture` and `type` property. The `type` property will determine the owner of the record. For example, the Contacts page will need only one record with `type: 'contacts'`, while the Blog page will require more records. So, we need three new pages for adding, editing and showing records. Before we jump into creating new templates, styling, and putting new stuff in to the controller, we should write our model class, which stands between the MongoDB server and our application and of course provides a meaningful API.

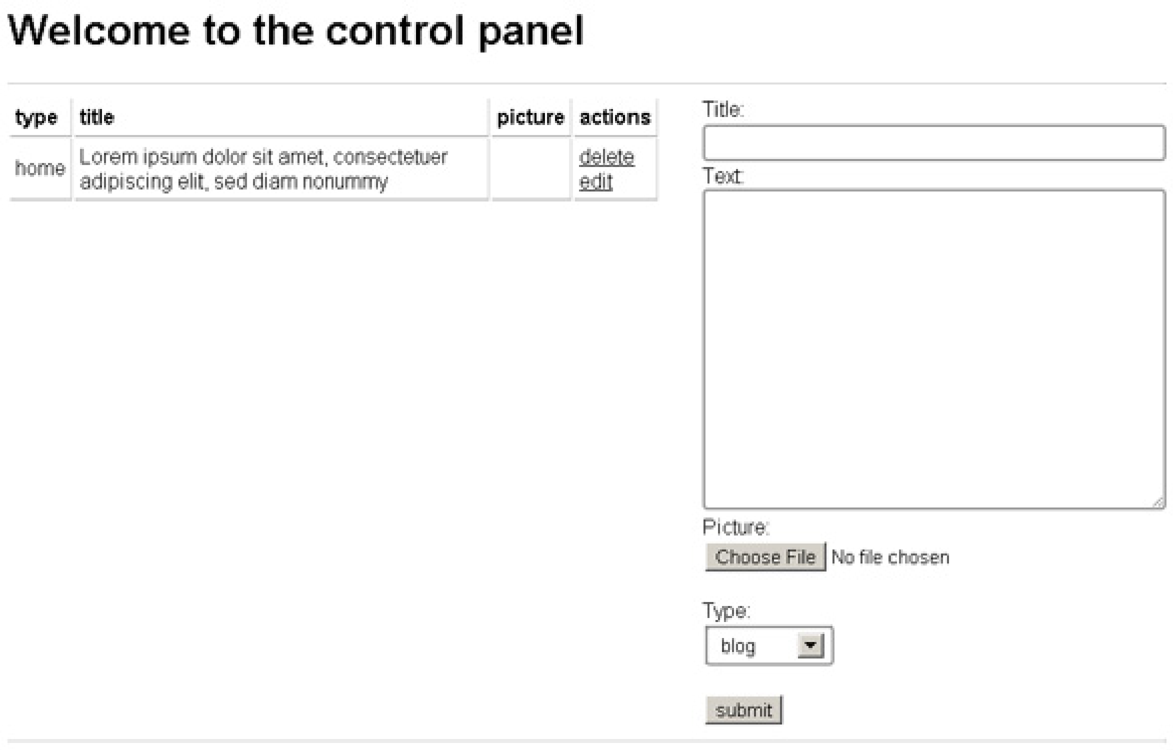
```
// /models/ContentModel.js

var Model = require("./Base"),
    crypto = require("crypto"),
    model = new Model();
var ContentModel = model.extend({
    insert: function(data, callback) {
        data.ID = crypto.randomBytes(20).toString('hex');
        this.collection().insert(data, {}, callback || function(){
    },
    update: function(data, callback) {
        this.collection().update({ID: data.ID}, data, {}, callback
    },
    getlist: function(callback, query) {
        this.collection().find(query || {}).toArray(callback);
    },
    remove: function(ID, callback) {
        this.collection().findAndModify({ID: ID}, [], {}, {remove:
    }
});
module.exports = ContentModel;
```

The model takes care of generating a unique ID for every record. We will need it in order to update the information later on.

If we want to add a new record for our Contacts page, we can simply use:

```
var model = new (require("../models/ContentModel"));
model.insert({
    title: "Contacts",
    text: "...",
    type: "contacts"
});
```

So, we have a nice API to manage the data in our mongodb collection. Now we are ready to write the UI for using this functionality. For this part, the `Admin` controller will need to be changed quite a bit. To simplify the task I decided to combine the list of the added records and the form for adding/editing them. As you can see on the screenshot below, the left part of the page is reserved for the list and the right part for the form.


Welcome to the control panel

type	title	picture	actions
home	Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy		delete edit

Having everything on one page means that we have to focus on the part which renders the page or to be more specific, on the data which we are sending to the template. That's why I created several helper functions which are combined, like so:

```
var self = this;
...
var v = new View(res, 'admin');
self.del(req, function() {
    self.form(req, res, function(formMarkup) {
        self.list(function(listMarkup) {
            v.render({
                title: 'Administration',
                content: 'Welcome to the control panel',
                list: listMarkup,
                form: formMarkup
            });
        });
    });
});
```

It looks a little bit ugly, but it works as I wanted. The first helper is a `del` method which checks the current GET parameters and if it finds `action=delete&id=[id of the record]`, it removes data from the collection. The second function is called `form` and it is responsible mainly for showing the form on the right side of the page. It checks if the form is submitted and properly updates or creates records in the database. At the end, the `list` method fetches the information and prepares an HTML table, which is later sent to the template. The implementation of these three helpers can be found in the source code for this tutorial.

Here, I've decided to show you the function which handles the file upload:

```
handleFileUpload: function(req) {
    if(!req.files || !req.files.picture || !req.files.picture.name
        return req.body.currentPicture || '';
    }
    var data = fs.readFileSync(req.files.picture.path);
    var fileName = req.files.picture.name;
    var uid = crypto.randomBytes(10).toString('hex');
    var dir = __dirname + "/../public/uploads/" + uid;
    fs.mkdirSync(dir, '0777');
    fs.writeFileSync(dir + "/" + fileName, data);
    return '/uploads/' + uid + "/" + fileName;
}
```

If a file is submitted, the node script `.files` property of the request object is filled with data. In our case, we have the following HTML element:

```
<input type="file" name="picture" />
```

This means that we could access the submitted data via `req.files.picture`. In the code snippet above, `req.files.picture.path` is used to get the raw content of the file. Later, the same data is written in a newly created directory and at the end, a proper URL is returned. All of these operations are synchronous, but it's a good practice to use the asynchronous version of `readFileSync`, `mkdirSync` and `writeFileSync`.

Front-End

The hard work is now complete. The administration panel is working and we have a `ContentModel` class, which gives us access to the information stored in the database. What we have to do now, is to write the front-end controllers and bind them to the saved content.

Here is the controller for the Home page – `/controllers/Home.js`

```
module.exports = BaseController.extend({
    name: "Home",
    content: null,
    run: function(req, res, next) {
        model.setDB(req.db);
        var self = this;
        this.getContent(function() {
            var v = new View(res, 'home');
            v.render(self.content);
        })
    },
    getContent: function(callback) {
        var self = this;
        this.content = {};
        model.getlist(function(err, records) {
            ... storing data to content object
            model.getlist(function(err, records) {
                ... storing data to content object
                callback();
            }, { type: 'blog' });
        }, { type: 'home' });
    }
});
```

The home page needs one record with a type of `home` and four records with a type of `blog`. Once the controller is done, we just have to add a route to it in `app.js`:

```
app.all('/', attachDB, function(req, res, next) {
    Home.run(req, res, next);
});
```

Again, we are attaching the `db` object to the `request`. Pretty much the same workflow as the one used in the administration panel.

The other pages for our front-end (client side) are almost identical, in that they all have a controller, which fetches data by using the model class and of course a route defined. There are two interesting situations which I'd like to explain in more detail. The first one is related to the blog page. It should be able to show all the articles, but also to present only one. So, we have to register two routes:

```
app.all('/blog/:id', attachDB, function(req, res, next) {
    Blog.runArticle(req, res, next);
});
app.all('/blog', attachDB, function(req, res, next) {
    Blog.run(req, res, next);
});
```

They both use the same controller: `Blog`, but call different `run` methods. Pay attention to the `/blog/:id` string. This route will match URLs like `/blog/4e3455635b4a6f6dccfaa1e50ee71f1cde75222b` and the long hash will be available in `req.params.id`. In other words, we are able to define dynamic parameters. In our case, that's the ID of the record. Once we have this information, we are able to create a unique page for every article.

The second interesting part is how I built the Services, Careers and Contacts pages. It is clear that they use only one record from the database. If we had to create a different controller for every page then we'd have to copy/paste the same code and just change the `type` field. There is a better way to achieve this though, by having only one controller, which accepts the `type` in its `run` method. So here are the routes:

```
app.all('/services', attachDB, function(req, res, next) {
    Page.run('services', req, res, next);
});
app.all('/careers', attachDB, function(req, res, next) {
    Page.run('careers', req, res, next);
});
app.all('/contacts', attachDB, function(req, res, next) {
    Page.run('contacts', req, res, next);
});
```

And the controller would look like this:

```
module.exports = BaseController.extend({
    name: "Page",
    content: null,
    run: function(type, req, res, next) {
        model.setDB(req.db);
        var self = this;
        this.getContent(type, function() {
            var v = new View(res, 'inner');
            v.render(self.content);
        });
    },
    getContent: function(type, callback) {
        var self = this;
        this.content = {}
        model.getlist(function(err, records) {
            if(records.length > 0) {
                self.content = records[0];
            }
            callback();
        }, { type: type });
    }
});
```

Deployment

Deploying an Express based website is actually the same as deploying any other Node.js application:

- The files are placed on the server.
- The node process should be stopped (if it is running).
- An `npm install` command should be ran in order to install the new dependencies (if any).
- The main script should then be run again.

Keep in mind that Node is still fairly young, so not everything may work as you expected, but there are improvements being made all the time. For example, forever guarantees that your Nodejs program will run continuously. You can do this by issuing the following command:

```
forever start yourapp.js
```

This is what I'm using on my servers as well. It's a nice little tool, but it solves a big problem. If you run your app with just `node yourapp.js`, once your script exits unexpectedly, the server goes down. `forever`, simply restarts the application.

Now I'm not a system administrator, but I wanted to share my experience integrating node apps with Apache or Nginx, because I think that this is somehow part of the development workflow.

As you know, Apache normally runs on port `80`, which means that if you open `http://localhost` or `http://localhost:80` you will see a page served by your Apache server and most likely your node script is listening on a different port. So, you need to add a virtual host that accepts the requests and sends them to the right port. For example, let's say that I want to host the site, that we've just built, on my local Apache server under the `expresscompletewebsite.dev` address. The first thing that we have to do is to add our domain to the `hosts` file.

```
127.0.0.1   expresscompletewebsite.dev
```

After that, we have to edit the `httpd-vhosts.conf` file under the Apache configuration directory and add

```
# expresscompletewebsite.dev
<VirtualHost *:80>
    ServerName expresscompletewebsite.dev
    ServerAlias www.expresscompletewebsite.dev
    ProxyRequests off
    <Proxy *>
        Order deny,allow
        Allow from all
    </Proxy>
    <Location />
        ProxyPass http://localhost:3000/
        ProxyPassReverse http://localhost:3000/
    </Location>
</VirtualHost>
```

The server still accepts requests on port `80`, but forwards them to port `3000`, where node is listening.

The Nginx setup is much much easier and to be honest, it's a better choice for hosting Nodejs based apps. You still have to add the domain name in your `hosts` file. After that, simply create a new file in the `/sites-enabled` directory under the Nginx installation. The content of the file would look something like this:

```
server {
    listen 80;
    server_name expresscompletewebsite.dev
    location / {
            proxy_pass http://127.0.0.1:3000;
            proxy_set_header Host $http_host;
    }
}
```

Keep in mind that you can't run both Apache and Nginx with the above hosts setup. That's because they both require port `80`. Also, you may want to do a little bit of additional research about better server configuration if you plan to use the above code snippets in a production environment. As I said, I'm not an expert in this area.

Conclusion

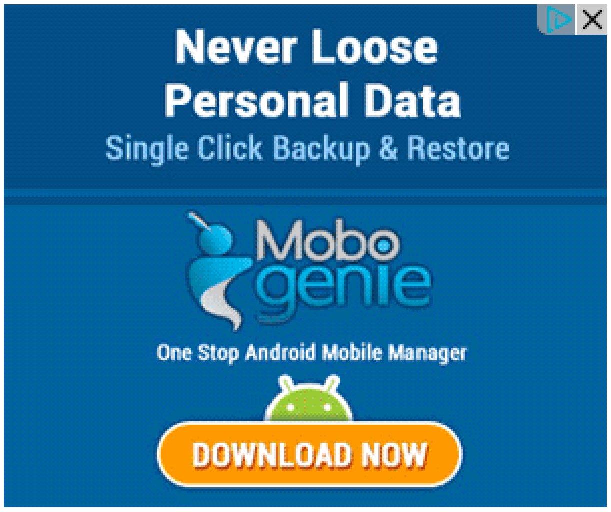
Express is a great framework, which gives you a good starting point to begin building your applications. As you can see, it's a matter of choice on how you will extend it and what you will use to build with it. It simplifies the boring tasks by using a few great middlewares and leaves the fun parts to the developer.

Source code

The source code for this sample site that we built is available on GitHub – https://github.com/tutsplus/build-complete-website-expressjs. Feel free to fork it and play with it. Here are the steps for running the site.

- Download the source code
- Go to the `app` directory
- Run `npm install`
- Run the mongodb daemon
- Run `node app.js`

Like 150 people like this. Be the first of your friends.


Never Loose
Personal Data
Single Click Backup & Restore
Mobo genie
One Stop Android Mobile Manager
DOWNLOAD NOW

Tags: expressjs

By Krasimir Tsonev

Krasimir Tsonev is a coder with over ten years of experience in web development. With a strong focus on quality and usability, he is interested in delivering cutting edge applications. Currently, with the rise of the mobile development, Krasimir is enthusiastic to work on responsive applications targeted to various devices. Living and working in Bulgaria, he graduated at the Technical University of Varna with a bachelor and master degree in computer science. If you'd like to stay up to date on his activities, refer to his blog or follow him on Twitter.

Note: Want to add some source code? Type <pre><code> before it and </code></pre> after it. Find out more