Express.js Tutorial

Posted on March 25th, 2012 under Express.js, Node.js
Tags: Express.js, git, GitHub, node.js

Looking for a good tutorial on Express.js to help you get quickly productive in it? You have come to the right place.

> This tutorial is for Express 3, the current version is Express 4. Unless you want to use Express 3, consider the tutorial outdated.

In this tutorial I will run you through the process setting up an Express.js app and making it do what a basic website might do. You will learn the basics of routes, views, Jade templates, Stylus CSS engine, handling POST and GET requests.

Let's create an online store to sell Ninja items. We will call it the Ninja Store.

Installing Express

First of all install Express.js as a global module:

```
$ npm install express -g
```

If that fails:

```
$ sudo npm install express -g
```

Now, create a directory and create an Express app in it:

```
$ mkdir ninja-store
$ cd ninja-store
$ express --sessions --css stylus
```

If you like shortcuts, you could accomplish the same this way too:

```
$ express ninja-store --sessions --css stylus && cd ninja-sto
```

We want to have support for sessions, hence the `--sessions` option. Using `--css`, we specified that we would be using the Stylus CSS engine. If you are a Less fan, you can specify `--css less`. Not specifying the `--css` option will default to the plain vanilla CSS we all are familiar with.

Then install the dependencies for the app:

```
$ npm install
```

That will install a bunch of modules used by the app. With that the base of the app is ready. It is already a working app. Let's see what it outputs. Start the app:

```
$ node app
Express server listening on port 3000
```

Then load `http://localhost:3000/` in your browser. You will see a simple webpage with the title "Express", which looks something like this:

So looks like the app is working. Time to find out how it works.

Request flow in Express

This is how a request to an Express server flows:

Route → Route Handler → Template → HTML

The route defines the URL schema. It captures the matching request and passed on control to the corresponding route handler. The route handler processes the request and passes the control to a template. The template constructs the HTML for the response and sends it to the browser.

The route handler need not always pass the control to a template, it can optionally send the response to a request directly .

To understand how Express works, let's get working on the Ninja Store app.

Setting up the Routes

We will modify some of the stuff Express generated for us to make it more suited for our app, and more logical so that you can understand the inner workings of Express better.

Rename the `index.jade` file in the `views` folder to `home.jade`. This is actually a part of setting up views in Express.js, I will get there in the next section.

Delete the `index.js` and `user.js` from the `routes` folder. We don't need them

and their presence could be potentially confusing. Create a new file called `store.js` in the `routes` folder, and put the following code in it:

```
exports.home = function(req, res){
  res.render('home', { title: 'Ninja Store' })
};
```

Then we are going to modify `app.js`:

Delete these

```
  , routes = require('./routes')
  , user = require('./routes/user')
```

Add the following right before `var app = express();`.

```
var store = require('./routes/store');
```

Delete these

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

and add this

```
app.get('/', store.home);
```

With that we have set up our own route for the home page.

Routes are URL schema for a website. In Express you define them using `app.get()`, `app.post()`, `app.delete()` etc. The get, post, delete methods are derived from their corresponding HTTP verbs.

So far we created a single route for our app. Very soon we will be creating more, but before that I'll tell you about the files in the `routes` directory.

The `routes` directory is a convention, not a compulsion. In the `routes` directory we create appropriately named files which will handle the routes we define in the `app.js` file. We will import these files into our app and assign the functions defined in them to handle various routes.

The imported file becomes sort of like an instance of the class of the route handler file. In our case `./routes/store.js` is the class, `store` is instance, and `store.home` is a method of the instance.

Again as a recommended convention, we create an appropriately named variable for the imported file from the `routes` directory. Then we pass one of its functions as the second parameter for the route. For eg:

```
app.get('/', store.home);
```

From that you might probably guess, we can pass any function as the second parameter for the route. You are correct, even this would work:

```
app.get('/', function(req, res) {
    res.render('home', { title: 'Ninja Store' });
});
```

We don't need to render HTML pages either:

```
app.get('/', function(req, res) {
    res.send('Look ma, no HTML!');
});
```

Now run the app again and see what you get. Title has changed to "Ninja Store" and you see:

We are making progress. Time to spice up the home page

Rendering Views

You have seen `res.render()` and `res.send()` in action already and probably have a fair idea about what they do. `res.send()` will directly send a string of text to the browser and `res.render()` will render a Jade template.

While it is possible to create a website entirely using `res.send()`, it is certainly not recommended to do so, because you will only end up with a complex and dirty looking codebase. Jade is the default templating engine for Express. Let's find out the basics of `res.render()` and the Jade templates.

Open the file named `home.jade` in the `views` directory. Let's examine its content:

```
extends layout

block content
  h1= title
  p Welcome to #{title}
```

`extend layout` means, this view file should look for another view file named `layout.jade` in the same directory and fill in the `block` placeholders defined in `layout.jade`.

Now, let's take a look at the contents of `layout.jade`.

```
doctype 5
html
  head
    title= title
```

```
    link(rel='stylesheet', href='/stylesheets/style.css')
  body
    block content
```

It does indeed have a `block` named content, the content for which is defined in `home.jade`. Note, you can use anything for `block` names; "content" is just a convention.

What you are seeing is Jade code. The very basics of Jade is this:

i. HTML tag names to create tags, but without the angular brackets
ii. Spaces or tabs for indentation, but don't mix them
iii. `#` to assign an `id` to a tag
iv. `.` to assign a class to a tag, or create a div with a class if not already created
v. (property='value') to create attributes and assign values to a tag

Jade is out of scope of this tutorial, so you might want to read more about it here. However, as we proceed with the tutorial, I will explain you the relevant Jade code we come across.

The `doctype 5` you see in `layout.jade` is doing a doctype declaration of HTML5. Also notice the relatively 'cryptic' `title= title`, the code is explained below.

`title= title`: The view expects a variable called `title` from the route handler. The variable will be set as the content of the `title` tag AFTER escaping special characters. To not-escape, use `title!= title`.

You can send data to views via the variables object in the `res.render()` method. Eg:

```
res.render('home', { title: 'Ninja Store' });
```

The `title` variable can be accessed using `=title`, `!=title` or `#{title}` in the views.

```
  p Welcome to #{title}
  p= title
  p!= title
```

By default `res.send()` and `res.render()` send the HTTP status code of 200. You can specify your own HTTP status code.

Custom status code example for `res.send()`:

```
res.send('File not Found', 404);
```

Custom status code example for `res.render()` (make sure you have created a file named `404.jade` in the `views` directory):

```
res.status(404).render('404', { title: 'File not Found'});
```

Ok, now that we know the basic of views, let's revamp our homepage!

Put the following code in the `home.jade` file:

```
extends layout

block content
  #wrapper
    #logo
        img(src='/images/logo.png')
    #display
        #login
            form(method='post')
                | Enter your name if you want to be a ninja
                div
                    input(type='text', name='username')
                    input(type='submit', value='Log In')

    #footer
        div Copyright © Ninja Store #{+new Date().getFullYea
        a(href='/page?name=about') About
        | |
        a(href='/page?name=contact') Contact Us
```

Make sure you create a nice logo for the store, name it `logo.png`, and keep it in the `/public/images/` directory.

Put the following code in the `style.styl` file in the `/public/stylesheets/` directory:

```
body
  padding: 0
  font: 14px "Lucida Grande", Helvetica, Arial, sans-serif

a
  color: #0069FF

#wrapper
  width: 450px
  margin: 0 auto
  padding: 40px 20px
  background: #fff

#logo
  text-align: center

#display
  margin: 20px 0 50px
```

```
#userbar
  margin-bottom: 10px
```

Aren't we supposed to edit the `style.css` file? Well, `style.styl` is the Stylus file which generates the `style.css` file. Even though we code our CSS in the `.styl` file, we always include the `.css` file in the Jade template:

```
link(rel='stylesheet', href='/stylesheets/style.css')
```

So what is the point of using Stylus? Stylus offers a dynamic and efficient syntax for generating CSS. The details is out of scope of this tutorial, you can learn more about Stylus here.

Note: just like Jade templates, make sure you either uses spaces or tabs consistently for indenting your Stylus code.

Changes made to the `views` and everything in the `public` directory will be updated immediately when you refresh the browser. Refresh the browser and see what you get. In rare cases, the view may not be update - feel free to restart the server.

Wow! There you have, a log in form.

Try logging in.

Meh! Hit by an error:

```
Cannot POST /
```

Let me explain what's going on.

Remember the route

```
app.get('/', store.home);
```

we created?

That was for GET requests to the root of the website. Since our form uses the POST method, the route is not capturing it. We need to create a POST router for the home page to process the form.

Handling Forms - POST, GET Requests

Add this to app.js:

```
app.post('/', store.home_post_handler);
```

Now we need to define the `home_post_handler()` function in `store.js`

module. Add the following to `store.js`:

```
// handler for form submitted from homepage
exports.home_post_handler = function(req, res) {
    // if the username is not submitted, give it a default o
    username = req.body.username || 'Anonymous';
    // store the username as a session variable
    req.session.username = username;
    // redirect the user to homepage
    res.redirect('/');
};
```

So handling POST data is pretty easy - just look for them in the `req.body` object.

Refresh the browser.

You still get `Cannot POST /`. That's because for changes you make in .js files to take effect, you need to restart the server. So restart the server and try logging in again.

This time the form submission works fine, but you still see the log in form. Did we log in or not? We did log in, but our route handler made it confusing. We'll change it now. Edit the `exports.home()` function:

```
// handler for homepage
exports.home = function(req, res) {
    // if user is not logged in, ask them to login
    if (typeof req.session.username == 'undefined') res.rende
    // if user is logged in already, take them straight to t
    else res.redirect('/items');
};
```

Restart the server and log in again.

```
Cannot GET /items
```

An error again. But it looks more promising this time, the app is trying to load an undefined route. If we define the route, we would have it working.

Before we get to defining the missing route, let's optimize the views a little bit. Some components of the view will be common to all the views, so it is a good idea to modularize them. The Jade template engine has a command called `include` using which you can include files into a view.

Create a file called `footer.jade` in the `views` directory with the following content:

```
#footer
  div Copyright © Ninja Store #{+new Date().getFullYear()}
  a(href='/page?name=about') About
  | |
  a(href='/page?name=contact') Contact Us
```

Create a file called `userbar.jade` in the `views` directory with the following content:

```
#userbar
  | Welcome
  b #{username}
  | |
  a(href='/items') Items
  | |
  a(href='/logout') Log Out
```

Now edit the `home.jade` file:

```
extends layout

block content
  #wrapper
    #logo
      a(href='/')
        img(src='/images/logo.png')
    #display
      #login
        form(method='post')
          | Enter your name if you want to be a ninja
          div
            input(type='text', name='username')
            input(type='submit', value='Log In')

    include footer
```

We will be defining the missing `items/` and a related route in a while. In the process, we will also find out how to handle GET requests in Express.js.

There are two types of parametric GET requests in Express.js - `req.params` and `req.query`.

The `req.params` object contains request parameters right in the URL. It uses the clean URL scheme. Eg: `http://example.com/product/1274`.

The `req.query` object contains request parameters in the GET query. It uses the conventional GET request scheme. Eg: `http://example.com?product=1274`.

For displaying the items we will use the `req.params` method.

Add these routes to `app.js`:

```
// display the list of item
app.get('/items', store.items);
// show individual item
app.get('/item/:id', store.item);
```

`/items` for listing the items of the store.

`/item/:name` for displaying the individual items.

We will use a simple hard-coded array as the data source. Add the following to the `store.js` file.

```
// our 'database'
var items = {
    SKN:{name:'Shuriken', price:100},
    ASK:{name:'Ashiko', price:690},
    CGI:{name:'Chigiriki', price:250},
    NGT:{name:'Naginata', price:900},
    KTN:{name:'Katana', price:1000}
};
```

Now create the corresponding route handlers for the routes we added. Add the following to `store.js`:

```
// handler for displaying the items
exports.items = function(req, res) {
    // don't let nameless people view the items, redirect the
    if (typeof req.session.username == 'undefined') res.redi
    else res.render('items', { title: 'Ninja Store - Items',
};

// handler for displaying individual items
exports.item = function(req, res) {
    // don't let nameless people view the items, redirect the
    if (typeof req.session.username == 'undefined') res.redi
    else {
        var name = items[req.params.id].name;
        var price = items[req.params.id].price;
        res.render('item', { title: 'Ninja Store - ' + name,
    }
};
```

Time to create the views for the routes.

View for listing all the items on the store. Name it `items.jade`.

```
extends layout

block content
  #wrapper
    #logo
      img(src='/images/logo.png')
    #display
      include userbar

      -for (var id in items)
```

```
        - var item = items[id]
        div
          a(href='/item/#{id}') #{item.name} - $#{item.price
    include footer
```

View for listing individual items of the store. Name it `item.jade`.

```
extends layout

block content
  #wrapper
    #logo
      img(src='/images/logo.png')
    #display
      include userbar

      p The #{name.toLowerCase()} is one of the must-have ite
      p Buy it today!

    include footer
```

Our footer links uses the conventional GET style links, let's find out how to handle those kind of requests.

First create the routes for the footer links. Add the following to `app.js`.

```
// show general pages
app.get('/page', store.page);
```

Now, the route handler in `store.js`:

```
// handler for showing simple pages
exports.page = function(req, res) {
    var name = req.query.name;
    var contents = {
        about: 'Ninja Store sells the coolest ninja stuff in
        contact: 'You can contact us at <address><strong>Nin
    };
    res.render('page', { title: 'Ninja Store - ' + name, use
};
```

And then the view for the pages. Create a new view file named `page.jade` in the `views` directory with the following content.

```
extends layout

block content
  #wrapper
    #logo
```

```
      a(href='/')
        img(src='/images/logo.png')
    #display
      p!= content

    include footer
```

Notice how we use `p!= content` instead of `p #{content}`. That is because we don't want the Jade engine to escape the HTML content of the variable.

we need to create a route and a handler for logging out. Add this route to `app.js`:

```
app.get('/logout', function(req, res) {
    // delete the session variable
    delete req.session.username;
    // redirect user to homepage
    res.redirect('/');
});
```

Notice how we specified the route handler right in the `app.js` file with the route definition. All the route definition cares about is that the second parameter should be a function, wherever it may be defined; just make sure to pass the request object (`req`) and the response object (`res`) to it.

With that our Ninja Store website ready. Start the app and get ready to witness the brilliance of the Ninja Store in action.

Conclusion

You know what? The whole project is on GitHub I intentionally didn't tell you earlier, so that you don't get lazy and skip learning. Now that you done the hard work, feel free to clone the Ninja Store repository and go berserk on it.

Express.js is a lot more than what I covered in this tutorial. I have already, and will be covering the more advanced topics in dedicated articles on my website.

Related to this post

i. Express.js Custom Error Pages – 404 and 500
ii. Mobile Web Development in Express.js (Node.js)
iii. Express.js Jade Partials – How to use them
iv. vhost in Express.js
v. How to serve static HTML files in Express.js
vi. POST / GET Request Handling in Node.js Express
vii. WordPress: Image has failed to upload due to an error
viii. Handle File Uploads in Express (Node.js)
ix. Form Handling / Processing in Express.js
x. Kohana Route not Working: Kohana gives a 404 Error for Route

136 Responses to “Express.js Tutorial”

Gordo says:
December 2, 2013 at 9:38 pm
Great tutorial, thanks a lot!
I would include userbar in page.jade for navigating back

david says:
March 7, 2014 at 12:04 am
Thanks so much!! this is great

monish says:
March 14, 2014 at 8:17 pm
Simple and easy! Great explanation..

Michael SPohn says:
July 10, 2014 at 8:07 pm
I’m sure that I am missing something simple, but running express –sessions gives me the error: unknown option –sessions. Any help with this would be greatly appreciated.

daniel says:
August 1, 2014 at 10:30 am
Hi, I have the same problem as Michael. option –sessions is unknown. I have installed express version 3.0.0. Thank you very much.

Captain says:
August 7, 2014 at 12:50 am
The express generator of Express 4 does not have the sessions option.

« Previous 1 … 12 13 14

Make a Comment

Your Name

Your E-Mail

Submit Comment

Follow @hacksparrow

Recent Comments

bro on Web speed is slow but Torrent speed is fast – Solution!
CoursesWeb on JavaScript: get the number of properties in an object WITHOUT LOOPING
Sunjoy Jeergall on The MongoDB Tutorial
Nick Baki on Express.js HTTPS
Yogesh on TCP Socket Programming in Node.js
pradeep; on Python: split string method and examples
Marco Faustinelli on jQuery with Node.js
Hugo Volkaert on Ubuntu: how to install easy_install
Carlos Rene on Running Express.js in Production Mode
sravan kumar on Node.js Image Processing and Manipulation

Tags

Aptana array CURL database Express Express.js
Firefox git GitHub Google Google Chrome Gzip Homebrew
HTML HTTP HTTPS JavaScript javascript array
javascript for loop Linux list lol Mac Microsoft
MongoDB Mozilla Mozilla Firefox MySQL mysqldump
node.js NoSQL npm Object pagination PHP
Python Redis Regex Sitemap sitemap.xml.gz string
Thunderbird tuple Ubuntu Wordpress