### What is Pug?

To understand Pug, you need to remember that the browser reads HTML and CSS and then displays formatted images and text to the client based on what that HTML and CSS tells it to do.

**Pug** is the middleman.

**Pug** is a template engine for **Node.js**.

A template engine allows us to inject data and then produce HTML.

In short: **At run time, Pug (and other template engines) replace variables in our file with actual values, and then send the resulting HTML string to the client.**

### How do we use Pug?

The same way we use other template engines.

Recall the following code from my: Build a Weather Website in 30 minutes with Node.js + Express + OpenWeather tutorial where we used **EJS** as our template language:

// TERMINAL  
npm install ejs --save

// SERVER.JS  
app.set('view engine', 'ejs')

Using Pug is just as easy. Here are the three steps:

1. Install Pug into your project: npm install pug --save
2. Set up your view engine: app.set(‘view engine’, ‘pug’)
3. Create a .pug file

### So, what does a .pug file look like?

Great question! Before we look at a .pug file, lets take a look at what our .ejs file looked like in my last tutorial:

**With EJS**

### Pug is awesome

* Switching from EJS to Pug brought our code down from 27 lines to just 17!
* When you write with Pug, you write code that looks like paragraphs. This greatly improves code-readability and streamlines projects with multiple developers.
* There are no closing tags with Pug. Pug makes use of indentation to determine the nesting of tags. There are even shorthands for classes (.) and IDs (#)
* Most importantly, we can write JavaScript that actually (almost/kind of) looks like JavaScript within our pug files.

### Pug is Bad

* With Pug, white-space matters. And it matters big time. The slightest mistake in your formatting/indenting/spacing means big problems for your code.
* With Pug, you can’t copy HTML from anywhere, you have to convert everything to Pug before you can use it.

### Project Reference (you can download fro github)

You can view the weather website full code (using Pug) on [Github](https://github.com/bmorelli25/simple-nodejs-weather-app/tree/pug" \t "_blank).

Hopefully, you can now understand what Pug.js is, what a .pug file looks like, and why a template engine is important in web development.

**Using template engines with Express**

A ***template engine*** enables you to use static template files in your application. At runtime, the template engine replaces variables in a template file with actual values, and transforms the template into an HTML file sent to the client. This approach makes it easier to design an HTML page.

Some popular template engines that work with Express are [Pug](https://pugjs.org/api/getting-started.html), [Mustache](https://www.npmjs.com/package/mustache), and [EJS](https://www.npmjs.com/package/ejs). The [Express application generator](https://expressjs.com/en/starter/generator.html) uses [Jade](https://www.npmjs.com/package/jade) as its default, but it also supports several others.

See [Template Engines (Express wiki)](https://github.com/strongloop/express/wiki#template-engines) for a list of template engines you can use with Express. See also [Comparing JavaScript Templating Engines: Jade, Mustache, Dust and More](https://strongloop.com/strongblog/compare-javascript-templates-jade-mustache-dust/).

**Note**: Jade has been renamed to [Pug](https://www.npmjs.com/package/pug). You can continue to use Jade in your app, and it will work just fine. However if you want the latest updates to the template engine, you must replace Jade with Pug in your app.

To render template files, set the following [application setting properties](https://expressjs.com/en/4x/api.html#app.set), set in app.js in the default app created by the generator:

* views, the directory where the template files are located. Eg: app.set('views', './views'). This defaults to the viewsdirectory in the application root directory.
* view engine, the template engine to use. For example, to use the Pug template engine: app.set('view engine', 'pug').

Then install the corresponding template engine npm package; for example to install Pug:

$ npm install pug --save

Express-compliant template engines such as Jade and Pug export a function named \_\_express(filePath, options, callback), which is called by the res.render() function to render the template code.

Some template engines do not follow this convention. The [Consolidate.js](https://www.npmjs.org/package/consolidate) library follows this convention by mapping all of the popular Node.js template engines, and therefore works seamlessly within Express.

After the view engine is set, you don’t have to specify the engine or load the template engine module in your app; Express loads the module internally, as shown below (for the above example).

app.set('view engine', 'pug')

Create a Pug template file named index.pug in the views directory, with the following content:

html

head

title= title

script

function test()

alert(‘hi’)

body

h1= message

input

type=text

keypress=test()

div

button

Then create a route to render the index.pug file. If the view engine property is not set, you must specify the extension of the viewfile. Otherwise, you can omit it.

app.get('/', function (req, res) {

res.render('index', { title: 'Hey', message: 'Hello there!' })

})

When you make a request to the home page, the index.pug file will be rendered as HTML.

Note: The view engine cache does not cache the contents of the template’s output, only the underlying template itself. The view is still re-rendered with every request even when the cache is on.

To learn more about how template engines work in Express, see: [“Developing template engines for Express”](https://expressjs.com/en/advanced/developing-template-engines.html).

**node-windows**

This is a standalone module, originally designed for internal use in [NGN](http://github.com/thinkfirst/NGN). However; it is capable of providing the same features for Node.JS scripts independently of NGN.

## Overview

The following features are available in node-windows:

* **Service Management**: Run Node.js scripts as native Windows services. Includes monitoring.
* **Event Logging**: Create logs in the Event log.
* **Commands**:
  + Elevated Permissions: Run a command with elevated privileges (may prompt user for acceptance)
  + Sudo: Run an exec command as a sudoer.
  + Identify Administrative Privileges: Determines whether the current user has administrative privileges.
  + List Tasks: A method to list running windows tasks/services.
  + Kill Task: A method to kill a specific windows service/task (by PID).

## Installation

The recommended way to install node-windows is with npm, using the global flag:

npm install -g node-windows

Then, in your project root, run:

npm link node-windows

However; it is possible to use node-windows without the global flag (i.e. install directly into the project root). More details regarding why this is not the recommended approach are available throughout this Readme.

## NO NATIVE MODULES

Using native node modules on Windows can suck. Most native modules are not distributed in a binary format. Instead, these modules rely on npm to build the project, utilizing [node-gyp](https://github.com/TooTallNate/node-gyp). This means developers need to have Visual Studio (and potentially other software) installed on the system, just to install a native module. This is portable, but painful... mostly because Visual Studio itself is over 2GB.

**node-windows does not use native modules.** There are some binary/exe utilities, but everything needed to run more complex tasks is packaged and distributed in a readily usable format. So, no need for Visual Studio... at least not for this module.

# Windows Services

node-windows has a utility to run Node.js scripts as Windows services. Please note that like all Windows services, creating one requires administrative privileges. To create a service with node-windows, prepare a script like:

var Service **=** require('node-windows').Service;

*// Create a new service object*

var svc **=** **new** Service({

  name**:**'Hello World',

  description**:** 'The nodejs.org example web server.',

  script**:** 'C:\\path\\to\\helloworld.js'

});

*// Listen for the "install" event, which indicates the*

*// process is available as a service.*

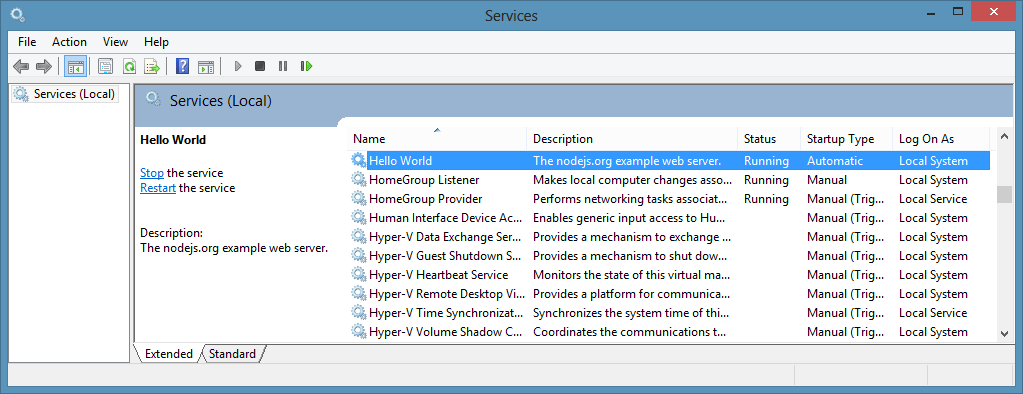
svc.on('install',function(){

  svc.start();

});

svc.install();

The code above creates a new Service object, providing a pretty name and description. The script attribute identifies the Node.js script that should run as a service. Upon running this, the script will be visible from the Windows Services utility.



The Service object emits the following events:

* install - Fired when the script is installed as a service.
* alreadyinstalled - Fired if the script is already known to be a service.
* invalidinstallation - Fired if an installation is detected but missing required files.
* uninstall - Fired when an uninstallation is complete.
* start - Fired when the new service is started.
* stop - Fired when the service is stopped.
* error - Fired in some instances when an error occurs.

In the example above, the script listens for the install event. Since this event is fired when a service installation is complete, it is safe to start the service.

Services created by node-windows are similar to most other services running on Windows. They can be started/stopped from the windows service utility, via NET START or NET STOPcommands, or even managed using the [sc](http://technet.microsoft.com/en-us/library/dd228922(v=ws.10).aspx) utility.

### Environment Variables

Sometimes you may want to provide a service with static data, passed in on creation of the service. You can do this by setting environment variables in the service config, as shown below:

var svc **=** **new** Service({

  name**:**'Hello World',

  description**:** 'The nodejs.org example web server.',

  script**:** 'C:\\path\\to\\helloworld.js',

  env**:** {

    name**:** "HOME",

    value**:** process.env["USERPROFILE"] *// service is now able to access the user who created its' home directory*

  }

});

You can also supply an array to set multiple environment variables:

var svc **=** **new** Service({

  name**:**'Hello World',

  description**:** 'The nodejs.org example web server.',

  script**:** 'C:\\path\\to\\helloworld.js',

  env**:** [{

    name**:** "HOME",

    value**:** process.env["USERPROFILE"] *// service is now able to access the user who created its' home directory*

  },

  {

    name**:** "TEMP",

    value**:** path.join(process.env["USERPROFILE"],"/temp") *// use a temp directory in user's home directory*

  }]

});

### User Account Attributes

If you need to specify a specific user or particular credentials to manage a service, the following attributes may be helpful.

The user attribute is an object with three keys: domain,account, and password. This can be used to identify which user the service library should use to perform system commands. By default, the domain is set to the local computer name, but it can be overridden with an Active Directory or LDAP domain. For example:

**app.js**

var Service **=** require('node-windows').Service;

*// Create a new service object*

var svc **=** **new** Service({

  name**:**'Hello World',

  script**:** require('path').join(\_\_dirname,'helloworld.js')

});

svc.user.domain **=** 'mydomain.local';

svc.user.account **=** 'username';

svc.user.password **=** 'password';

**...**

Both the account and password must be explicitly defined if you want the service module to run commands as a specific user. By default, it will run using the user account that launched the process (i.e. who launched node app.js).

The other attribute is sudo. This attribute has a single property called password. By supplying this, the service module will attempt to run commands using the user account that launched the process and the password for that account. This should only be used for accounts with administrative privileges.

**app.js**

var Service **=** require('node-windows').Service;

*// Create a new service object*

var svc **=** **new** Service({

  name**:**'Hello World',

  script**:** require('path').join(\_\_dirname,'helloworld.js')

});

svc.sudo.password **=** 'password';

**...**

### Cleaning Up: Uninstall a Service

Uninstalling a previously created service is syntactically similar to installation.

var Service **=** require('node-windows').Service;

*// Create a new service object*

var svc **=** **new** Service({

  name**:**'Hello World',

  script**:** require('path').join(\_\_dirname,'helloworld.js')

});

*// Listen for the "uninstall" event so we know when it's done.*

svc.on('uninstall',function(){

  console.log('Uninstall complete.');

  console.log('The service exists: ',svc.exists);

});

*// Uninstall the service.*

svc.uninstall();

The uninstall process only removes process-specific files. **It does NOT delete your Node.js script!**

### What Makes node-windows Services Unique?

Lots of things!

**Long Running Processes & Monitoring:**

The built-in service recovery for Windows services is fairly limited and cannot easily be configured from code. Therefore, node-windows creates a wrapper around the Node.js script. This wrapper is responsible for restarting a failed service in an intelligent and configurable manner. For example, if your script crashes due to an unknown error, node-windows will attempt to restart it. By default, this occurs every second. However; if the script has a fatal flaw that makes it crash repeatedly, it adds unnecessary overhead to the system. node-windows handles this by increasing the time interval between restarts and capping the maximum number of restarts.

**Smarter Restarts That Won't Pummel Your Server:**

Using the default settings, node-windows adds 25% to the wait interval each time it needs to restart the script. With the default setting (1 second), the first restart attempt occurs after one second. The second occurs after 1.25 seconds. The third after 1.56 seconds (1.25 increased by 25%) and so on. Both the initial wait time and the growth rate are configuration options that can be passed to a new Service. For example:

var svc **=** **new** Service({

  name**:**'Hello World',

  description**:** 'The nodejs.org example web server.',

  script**:** 'C:\\path\\to\\helloworld.js',

  wait**:** 2,

  grow**:** .5

});

In this example, the wait period will start at 2 seconds and increase by 50%. So, the second attempt would be 3 seconds later while the fourth would be 4.5 seconds later.

**Don't DOS Yourself!**

Repetitive recycling could potentially go on forever with a bad script. To handle these situations, node-windows supports two kinds of caps. Using maxRetries will cap the maximum number of restart attempts. By default, this is unlimited. Setting it to 3 would tell the process to no longer restart a process after it has failed 3 times. Another option is maxRestarts, which caps the number of restarts attempted within 60 seconds. For example, if this is set to 3 (the default) and the process crashes/restarts repeatedly, node-windows will cease restart attempts after the 3rd cycle in a 60 second window. Both of these configuration options can be set, just like wait or grow.

Finally, an attribute called abortOnError can be set to true if you want your script to **not**restart at all when it exits with an error.

### How Services Are Made

node-windows uses the [winsw](https://github.com/kohsuke/winsw) utility to create a unique .exe for each Node.js script deployed as a service. A directory called daemon is created and populated with myappname.exe and myappname.xml. The XML file is a configuration for the executable. Additionally, winsw will create some logs for itself in this directory (which are viewable in the Event log).

The myappname.exe file launches the node-windows wrapper, which is responsible for monitoring and managing the script. Since this file is a part of node-windows, moving the node-windows directory could result in the .exe file not being able to find the Node.js script. However; this should not be a problem if node-windows is installed globally, per the recommended installation instructions.

All of these daemon-specific files are created in a subdirectory called daemon, which is created in the same directory where the Node.js script is saved. Uninstalling a service will remove these files.

Event Logging

Services created with node-windows have two event logs that can be viewed through the Windows Event Viewer. A log source named myappname.exe provides basic logging for the executable file. It can be used to see when the entire service starts/stops or has errors. A second log, named after your service name (i.e. My App Name), is used by the node-windows monitor. It is possible to write to this log from the Node.js script using the node-windows Event Logging.

# Event Logging

New as of v0.1.0 is a non-C++ based event logging utility. This utility can write to the event log, making your logs visible from the Event Viewer.

To create a logger:

var EventLogger **=** require('node-windows').EventLogger;

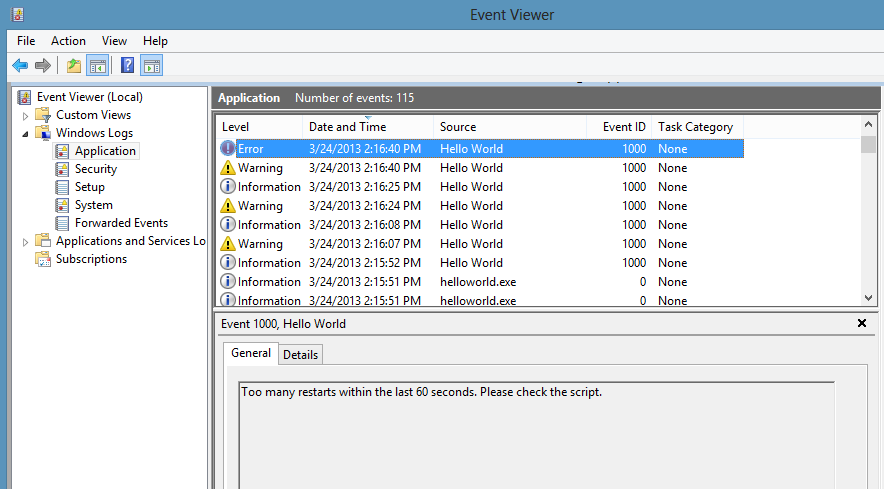
var log **=** **new** EventLogger('Hello World');

log.info('Basic information.');

log.warn('Watch out!');

log.error('Something went wrong.');

Looks similar to:



Some lesser-used options are also available through node-windows event logging.

log.auditSuccess('AUser Login Success');

log.auditFailure('AUser Login Failure');

Each log type (info, warn, error, auditSuccess, and auditFailure) method optionally accepts two additional arguments, including a code and callback. By default, the event code is 1000 if not otherwise specified. To provide a custom event code with a log message and write that message to the console, the following code could be used:

log.info('Something different happened!', 1002, function(){

  console.log('Something different happened!');

});

By default, event logs are all part of the APPLICATION scope. However; it is also possible to use the SYSTEM log. To do this, a configuration object must be passed to the new log:

var EventLogger **=** require('node-windows').EventLogger;

var log **=** **new** EventLogger({

  source**:** 'My Event Log',

  eventLog**:** 'SYSTEM'

});

# Commands

node-windows ships with several commands to simplify tasks on MS Windows.

## elevate

Elevate is similar to sudo on Linux/Mac. It attempts to elevate the privileges of the current user to a local administrator. Using this does not require a password, but it does require that the current user have administrative privileges. Without these privileges, the command will fail with a access denied error.

On systems with UAC enabled, this may prompt the user for permission to proceed:

**Syntax**:

elevate(cmd[,options,callback])

* cmd: The command to execute with elevated privileges. This can be any string that would be typed at the command line.
* options (optional): Any options that will be passed to require('child\_process').exec(cmd,<OPTIONS>,callback).
* callback (optional): The callback function passed to require('child\_process').exec(cmd,options,<CALLBACK>).

## sudo

Sudo acts similarly to sudo on Linux/Mac. Unlike elevate, it requires a password, but it will not prompt the user for permission to proceed. Like elevate, this still requires administrative privileges for the user, otherwise the command will fail. The primary difference between this and elevate() is the prompt.

**Syntax**:

sudo(cmd,password[,options,callback])

* cmd: The command to execute with elevated privileges. This can be any string that would be typed at the command line.
* password: The password of the user
* options (optional): Any options that will be passed to require('child\_process').exec(cmd,<OPTIONS>,callback).
* callback (optional): The callback function passed to require('child\_process').exec(cmd,options,<CALLBACK>).

## isAdminUser

This asynchronous command determines whether the current user has administrative privileges. It passes a boolean value to the callback, returning true if the user is an administrator or false if it is not.

**Example**

var wincmd **=** require('node-windows');

wincmd.isAdminUser(function(isAdmin){

**if** (isAdmin) {

    console.log('The user has administrative privileges.');

  } **else** {

    console.log('NOT AN ADMIN');

  }

});

## list

The list method queries the operating system for a list of running processes.

var wincmd **=** require('node-windows');

wincmd.list(function(svc){

  console.log(svc);

},true);

This returns an array of running processes. Supplying the optional true argument in the above example provides a list with verbose output. The output is specific to the version of the operating system. Here is an example of verbose output on a Windows 8 computer.

[{

  ImageName**:** 'cmd.exe',

  PID**:** '12440',

  SessionName**:** 'Console',

  'Session#'**:** '1',

  MemUsage**:** '1,736 K',

  Status**:** 'Unknown',

  UserName**:** 'Machine\\Corey',

  CPUTime**:** '0:00:00',

  WindowTitle**:** 'N/A'

},{

  ImageName**:** 'tasklist.exe',

  PID**:** '1652',

  SessionName**:** 'Console',

  'Session#'**:** '1',

  MemUsage**:** '8,456 K',

  Status**:** 'Unknown',

  UserName**:** 'Machine\\Corey',

  CPUTime**:** '0:00:00',

  WindowTitle**:** 'N/A'

}]

The regualar (non-verbose) output typically provides the ImageName,PID,SessionName,Session#, MemUsage, and CPUTime.

## kill

This method will kill a process by PID.

var wincmd **=** require('node-windows');

wincmd.kill(12345,function(){

  console.log('Process Killed');

});

In this example, process ID 12345 would be killed. It is important to note that the user account executing this node script may require administrative privileges.

# NodeJS : Creating HTML Form with POST method

The Below example Shows creating HTML form in NodeJS with POST method and GET methods to display input.

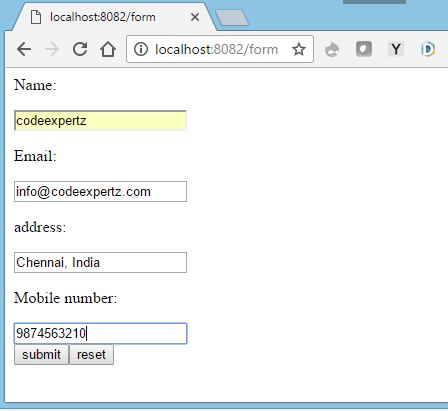
**Execution :**

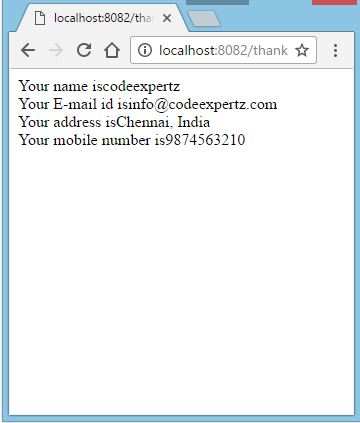
// Goto the directory that form\_program.js and run below command in command prompt or terminal.

Drive://Path to Source file >node form\_program.js

Example app listening at :::8082 Port

**Output:**





**Example:**

var http = require("http");

var express = require('express');

var app = express();

var bodyParser = require('body-parser');

var urlencodedParser = bodyParser.urlencoded({ extended: **true** });

*// Running Server Details.*

var server = app.listen(8082, function () {

var host = server.address().address

var port = server.address().port

console.log("Example app listening at %s:%s Port", host, port)

});

app.get('/form', function (req, res) {

var html='';

html +="<body>";

html += "<form action='/thank' method='post' name='form1'>";

html += "Name:</p><input type= 'text' name='name'>";

html += "Email:</p><input type='text' name='email'>";

html += "address:</p><input type='text' name='address'>";

html += "Mobile number:</p><input type='text' name='mobilno'>";

html += "<input type='submit' value='submit'>";

html += "<INPUT type='reset' value='reset'>";

html += "</form>";

html += "</body>";

res.send(html);

});

app.post('/thank', urlencodedParser, function (req, res){

var reply='';

reply += "Your name is" + req.body.name;

reply += "Your E-mail id is" + req.body.email;

reply += "Your address is" + req.body.address;

reply += "Your mobile number is" + req.body.mobilno;

res.send(reply);

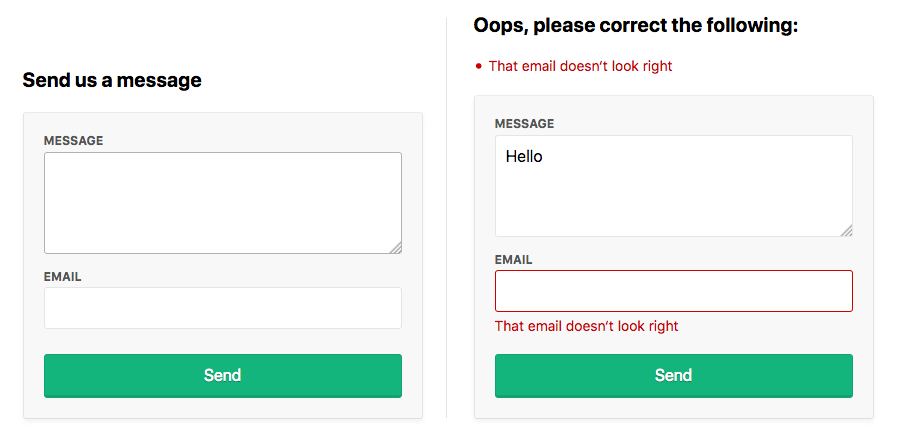
});

Typically the form handling process involves:

* displaying an empty HTML form in response to an initial GET request
* user submitting the form with data in a POST request
* validation on both the client and the server
* re-displaying the form populated with escaped data and error messages if invalid
* doing something with the sanitized data on the server if it’s all valid
* redirecting the user or showing a success message after data is processed.

Handling form data also comes with extra security considerations.

We’ll go through all of these and explain how to build them with Node.js and Express — the most popular web framework for Node. First, we’ll build a simple contact form where people can send a message and email address securely and then take a look what’s involved in processing file uploads.



## Setup

Make sure you’ve got a recent version of Node.js installed; node -v should return 8.9.0 or higher.

Download the starting code from here with git:

git clone https://github.com/sitepoint-editors/node-forms.git

cd node-forms

npm install

npm start

There’s not too much code in there. It’s just a bare-bones Express setup with EJS templates and error handlers:

// server.js

const path = require('path')

const express = require('express')

const layout = require('express-layout')

const routes = require('./routes')

const app = express()

app.set('views', path.join(\_\_dirname, 'views'))

app.set('view engine', 'ejs')

const middleware = [

layout(),

express.static(path.join(\_\_dirname, 'public')),

]

app.use(middleware)

app.use('/', routes)

app.use((req, res, next) => {

res.status(404).send("Sorry can't find that!")

})

app.use((err, req, res, next) => {

console.error(err.stack)

res.status(500).send('Something broke!')

})

app.listen(3000, () => {

console.log(`App running at http://localhost:3000`)

})

The root url / simply renders the index.ejs view.

// routes.js

const express = require('express')

const router = express.Router()

router.get('/', (req, res) => {

res.render('index')

})

module.exports = router

## Displaying the Form

When people make a GET request to /contact, we want to render a new view contact.ejs:

// routes.js

router.get('/contact', (req, res) => {

res.render('contact')

})

The contact form will let them send us a message and their email address:

<!-- views/contact.ejs -->

<div class="form-header">

<h2>Send us a message</h2>

</div>

<form method="post" action="/contact" novalidate>

<div class="form-field">

<label for="message">Message</label>

<textarea class="input" id="message" name="message" rows="4" autofocus></textarea>

</div>

<div class="form-field">

<label for="email">Email</label>

<input class="input" id="email" name="email" type="email" value="" />

</div>

<div class="form-actions">

<button class="btn" type="submit">Send</button>

</div>

</form>

See what it looks like at http://localhost:3000/contact.

## Form Submission

To receive POST values in Express, you first need to include the body-parsermiddleware, which exposes submitted form values on req.body in your route handlers. Add it to the end of the middlewares array:

// server.js

const bodyParser = require('body-parser')

const middlewares = [

// ...

bodyParser.urlencoded()

]

It’s a common convention for forms to POST data back to the same URL as was used in the initial GET request. Let’s do that here and handle POST /contact to process the user input.

Let’s look at the invalid submission first. If invalid, we need to pass back the submitted values to the view so they don’t need to re-enter them along with any error messages we want to display:

router.get('/contact', (req, res) => {

res.render('contact', {

data: {},

errors: {}

})

})

router.post('/contact', (req, res) => {

res.render('contact', {

data: req.body, // { message, email }

errors: {

message: {

msg: 'A message is required'

},

email: {

msg: 'That email doesn‘t look right'

}

}

})

})

If there are any validation errors, we’ll do the following:

* display the errors at the top of the form
* set the input values to what was submitted to the server
* display inline errors below the inputs
* add a form-field-invalid class to the fields with errors.

-- views/contact.ejs -->

<div class="form-header">

<% if (Object.keys(errors).length === 0) { %>

<h2>Send us a message</h2>

<% } else { %>

<h2 class="errors-heading">Oops, please correct the following:</h2>

<ul class="errors-list">

<% Object.values(errors).forEach(error => { %>

<li><%= error.msg %></li>

<% }) %>

</ul>

<% } %>

</div>

<form method="post" action="/contact" novalidate>

<div class="form-field <%= errors.message ? 'form-field-invalid' : '' %>">

<label for="message">Message</label>

<textarea class="input" id="message" name="message" rows="4" autofocus><%= data.message %></textarea>

<% if (errors.message) { %>

<div class="error"><%= errors.message.msg %></div>

<% } %>

</div>

<div class="form-field <%= errors.email ? 'form-field-invalid' : '' %>">

<label for="email">Email</label>

<input class="input" id="email" name="email" type="email" value="<%= data.email %>" />

<% if (errors.email) { %>

<div class="error"><%= errors.email.msg %></div>

<% } %>

</div>

<div class="form-actions">

<button class="btn" type="submit">Send</button>

</div>

</form>

Submit the form at http://localhost:3000/contact to see this in action. That’s everything we need on the view side.

## Validation and Sanitization

There is a handy middleware express-validator for validating and sanitizing data using the [validator.js](https://github.com/chriso/validator.js) library, let’s include it in our middlewares array:

// server.js

const validator = require('express-validator')

const middlewares = [

// ...

validator()

]

### Validation

With the [validators](https://github.com/chriso/validator.js#validators) provided we can easily check that a message and a valid email was provided:

// routes.js

const { check, validationResult } = require('express-validator/check')

router.post('/contact', [

check('message')

.isLength({ min: 1 })

.withMessage('Message is required'),

check('email')

.isEmail()

.withMessage('That email doesn‘t look right')

], (req, res) => {

const errors = validationResult(req)

res.render('contact', {

data: req.body,

errors: errors.mapped()

})

})

### Sanitization

With the [sanitizers](https://github.com/chriso/validator.js#sanitizers) provided we can trim whitespace from the start and end of the values, and normalize the email into a consistent pattern. This can help remove duplicate contacts being created by slightly different inputs. For example, ' Mark@gmail.com' and 'mark@gmail.com ' would both be sanitized into 'mark@gmail.com'.

Sanitizers can simply be chained onto the end of the validators:

const { matchedData } = require('express-validator/filter')

router.post('/contact', [

check('message')

.isLength({ min: 1 })

.withMessage('Message is required')

.trim(),

check('email')

.isEmail()

.withMessage('That email doesn‘t look right')

.trim()

.normalizeEmail()

], (req, res) => {

const errors = validationResult(req)

res.render('contact', {

data: req.body,

errors: errors.mapped()

})

const data = matchedData(req)

console.log('Sanitized:', data)

})

The matchedData function returns the output of the sanitizers on our input.

## The Valid Form

If there are errors we need to re-render the view. If not, we need to do something useful with the data and then show that the submission was successful. Typically, the person is redirected to a success page and shown a message.

HTTP is stateless, so you can’t redirect to another page and pass messages along without the help of a [session cookie](https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies) to persist that message between HTTP requests. A “flash message” is the name given to this kind of one-time-only message we want to persist across a redirect and then disappear.

There are three middlewares we need to include to wire this up:

const cookieParser = require('cookie-parser')

const session = require('express-session')

const flash = require('express-flash')

const middlewares = [

// ...

cookieParser(),

session({

secret: 'super-secret-key',

key: 'super-secret-cookie',

resave: false,

saveUninitialized: false,

cookie: { maxAge: 60000 }

}),

flash()

]

The express-flash middleware adds req.flash(type, message) which we can use in our route handlers:

// routes

router.post('/contact', [

// validation ...

], (req, res) => {

const errors = validationResult(req)

if (!errors.isEmpty()) {

return res.render('contact', {

data: req.body,

errors: errors.mapped()

})

}

const data = matchedData(req)

console.log('Sanitized: ', data)

// Homework: send sanitized data in an email or persist in a db

req.flash('success', 'Thanks for the message! I‘ll be in touch :)')

res.redirect('/')

})

The express-flash middleware adds messages to req.locals which all views have access to:

<!-- views/index.ejs -->

<% if (messages.success) { %>

<div class="flash flash-success"><%= messages.success %></div>

<% } %>

<h1>Working With Forms in Node.js</h1>

You should now be redirected to index view and see a success message when the form is submitted with valid data. Huzzah! We can now deploy this to production and be sent messages by the prince of Nigeria.

## Security considerations

If you’re working with forms and sessions on the Internet, you need to be aware of common security holes in web applications. The best security advice I’ve been given is “Never trust the client!”

### TLS over HTTPS

Always use TLS encryption over https:// when working with forms so that the submitted data is encrypted when it’s sent across the Internet. If you send form data over http://, it’s sent in plain text and can be visible to anyone eavesdropping on those packets as they journey across the Internet.

### Wear Your Helmet

There’s a neat little middleware called [helmet](https://helmetjs.github.io/docs/) that adds some security from HTTP headers. It’s best to include right at the top of your middlewares and is super easy to include:

// server.js

const helmet = require('helmet')

middlewares = [

helmet()

// ...

]

### Cross-site Request Forgery (CSRF)

You can protect yourself against [cross-site request forgery](https://en.wikipedia.org/wiki/Cross-site_request_forgery) by generating a unique token when the user is presented with a form and then validating that token before the POST data is processed. There’s a middleware to help you out here as well:

// server.js

const csrf = require('csurf')

middlewares = [

// ...

csrf({ cookie: true })

]

In the GET request we generate a token:

// routes.js

router.get('/contact', (req, res) => {

res.render('contact', {

data: {},

errors: {},

csrfToken: req.csrfToken()

})

})

And also in the validation errors response:

router.post('/contact', [

// validations ...

], (req, res) => {

const errors = validationResult(req)

if (!errors.isEmpty()) {

return res.render('contact', {

data: req.body,

errors: errors.mapped(),

csrfToken: req.csrfToken()

})

}

// ...

})

Then we just need include the token in a hidden input:

<!-- view/contact.ejs -->

<form method="post" action="/contact" novalidate>

<input type="hidden" name="\_csrf" value="<%= csrfToken %>">

<!-- ... -->

</form>

That’s all that’s required.

We don’t need to modify our POST request handler as all POST requests will now require a valid token by the csurf middleware. If a valid CSRF token isn’t provided, a ForbiddenError error will be thrown, which can be handled by the error handler defined at the end of server.js.

You can test this out yourself by editing or removing the token from the form with your browser’s developer tools and submitting.

### Cross-site Scripting (XSS)

You need to take care when displaying user-submitted data in an HTML view as it can open you up to [cross-site scripting(XSS)](https://developer.mozilla.org/en-US/docs/Glossary/Cross-site_scripting). All template languages provide different methods for outputting values. The EJS <%= value %> outputs the HTML escaped value to protect you from XSS, whereas <%- value %> outputs a raw string.

Always use the escaped output <%= value %> when dealing with user submitted values. Only use raw outputs when you’re sure that’s is safe to do so.

## File Uploads

Uploading files in HTML forms is a special case that requires an encoding type of "multipart/form-data". See [MDN’s guide to sending form data](https://developer.mozilla.org/en-US/docs/Learn/HTML/Forms/Sending_and_retrieving_form_data#A_special_case_sending_files) for more detail about what happens with multipart form submissions.

You’ll need additional middleware to handle multipart uploads. There’s an Express package named multer that we’ll use here:

// routes.js

const multer = require('multer')

const upload = multer({ storage: multer.memoryStorage() })

router.post('/contact', upload.single('photo'), [

// validation ...

], (req, res) => {

// error handling ...

if (req.file) {

console.log('Uploaded: ', req.file)

// Homework: Upload file to S3

}

req.flash('success', 'Thanks for the message! I’ll be in touch :)')

res.redirect('/')

})

This code instructs multer to upload the file in the “photo” field into memory and exposes the File object in req.file which we can inspect or process further.

The last thing we need is to add the enctype attribute and our file input:

<form method="post" action="/contact?\_csrf=<%= csrfToken %>" novalidate enctype="multipart/form-data">

<input type="hidden" name="\_csrf" value="<%= csrfToken %>">

<div class="form-field <%= errors.message ? 'form-field-invalid' : '' %>">

<label for="message">Message</label>

<textarea class="input" id="message" name="message" rows="4" autofocus><%= data.message %></textarea>

<% if (errors.message) { %>

<div class="error"><%= errors.message.msg %></div>

<% } %>

</div>

<div class="form-field <%= errors.email ? 'form-field-invalid' : '' %>">

<label for="email">Email</label>

<input class="input" id="email" name="email" type="email" value="<%= data.email %>" />

<% if (errors.email) { %>

<div class="error"><%= errors.email.msg %></div>

<% } %>

</div>

<div class="form-field">

<label for="photo">Photo</label>

<input class="input" id="photo" name="photo" type="file" />

</div>

<div class="form-actions">

<button class="btn" type="submit">Send</button>

</div>

</form>

Unfortunately, we also needed to include \_csrf as a GET param so that the csurfmiddleware plays ball and doesn’t lose track of our token during multipart submissions.

Try uploading a file, you should see the File objects logged in the console.

### Populating File Inputs

In case of validation errors, we can’t re-populate file inputs like we did for the text inputs. A common approach to solving this problem involves these steps:

* uploading the file to a temporary location on the server
* showing a thumbnail and filename of the attached file
* adding JavaScript to the form to allow people to remove the selected file or upload a new one
* moving the file to a permanent location when everything is valid.