CWEB280 -wk2

# Recommended Quiz: 20 minutes

Review : <https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Introduction>

In your own words try an explain the difference between Static Sites and Dynamic Sites

# ~~Recommended Tools: ESLint~~

~~Webstorm and ESLint can enforce a coding style for all our JS files and can fix/ point out issues like extra spaces or missing semi colons~~

~~Install es lint – type the following command in the terminal (cweb2021\lo1serveronly folder)~~

~~npm i eslint~~

~~node node\_modules\eslint\bin\eslint –init~~

~~Follow the Prompts and select the bolded option~~

~~? How would you like to use ESLint? ...~~

~~To check syntax only~~

~~To check syntax and find problems~~

**~~> To check syntax, find problems, and enforce code style~~**

~~? What type of modules does your project use? ...~~

~~JavaScript modules (import/export)~~

**~~> CommonJS (require/exports)~~**

~~None of these~~

~~? Which framework does your project use? ...~~

~~React~~

~~Vue.js~~

**~~> None of these~~**

~~? Does your project use TypeScript? »~~ **~~No~~** ~~/ Yes~~

~~? Where does your code run? ... (Press <space> to select, <a> to toggle all, <i> to invert selection)~~

~~Browser~~

**~~√ Node~~**

~~? How would you like to define a style for your project? ...~~

**~~> Use a popular style guide~~**

~~Answer questions about your style~~

~~Inspect your JavaScript file(s)~~

~~? Which style guide do you want to follow? ...~~

~~Airbnb: https://github.com/airbnb/javascript~~

~~Standard: https://github.com/standard/standard~~

**~~> Google: https://github.com/google/eslint-config-google~~**

~~XO: https://github.com/xojs/eslint-config-xo~~

~~? What format do you want your config file to be in? ...~~

~~JavaScript~~

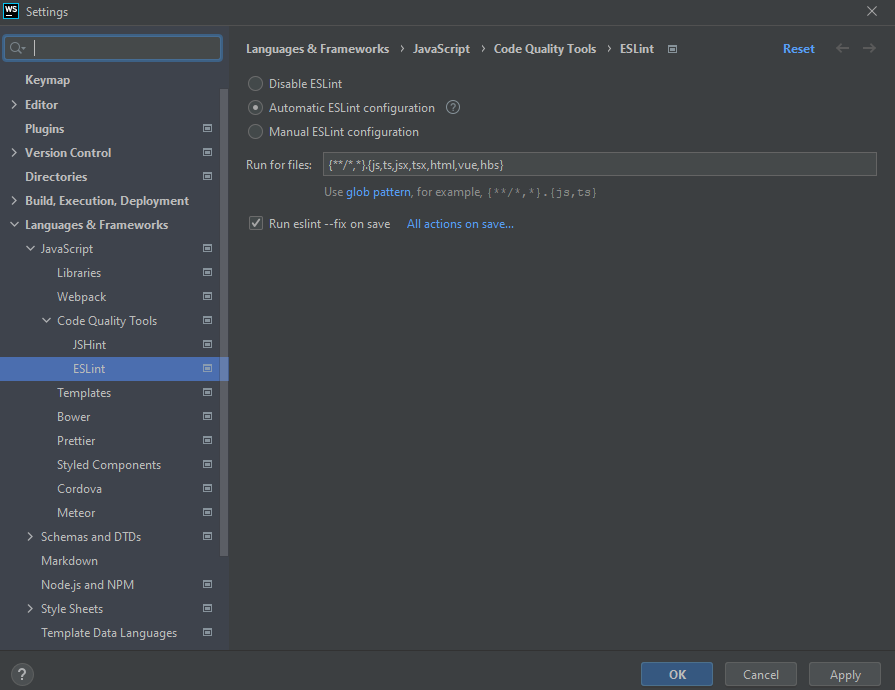
~~YAML~~

**~~> JSON~~**

~~eslint-config-google@latest eslint@>=5.16.0~~

~~? Would you like to install them now with npm? No /~~ **~~Yes~~**

~~In Webstorm – Open the settings menu Ctrl+Alt+S and find the ESLint settings and set them to the picture below – select Auto ESLint configuration and check ‘Run eslint –fix on save’~~

~~~~

~~No when you save a file Webstorm will automatically run ESLint on you code and fix any issues it can find.~~

# Template Engines in Express

Express can support several types of template engine. A template engine works in a rather simple manner:

* create a template (with the appropriate syntax) example {[title}}
* create an appropriate route to render the template,
* send “options” (or parameters) to the template that correspond to the placeholders in the template

## Handlebars

In this course we use the **Handlebars** template engine:

Learn more about Handlebars Syntax: <https://handlebarsjs.com/guide/>

Handlebars is an enhanced version of Mustache template engine. Handlebars allow for some built in helpers where as Moustache does not all any logic (no Conditional, Loops or variable declaration)

## PUG

Probably the most popular template engine is called PUG (formerly Jade)

PUG allows for logic block in the template (variable declaration, conditionals, loops etc)

PUG syntax does not use standard html tags - *example: p= 'Hello World' instead of <p>Hello World</p>*

Learn more about PUG syntax: <https://pugjs.org/language/code.html>

## Partials

Template engines also allow for reusable blocks called partials. Partials can place content on multiple templates, so we do not have to include the same mark up in all templates.

An alternative to partials is the \views\layouts.hbs file in the lo1serveronly project. This file is used as a wrapper to all the templates in the view folder

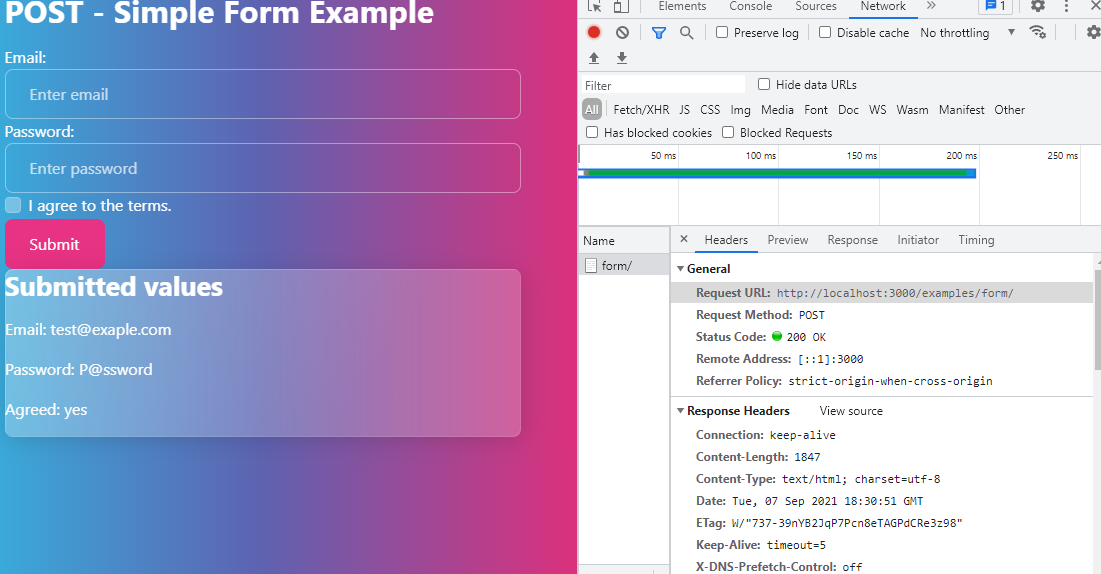
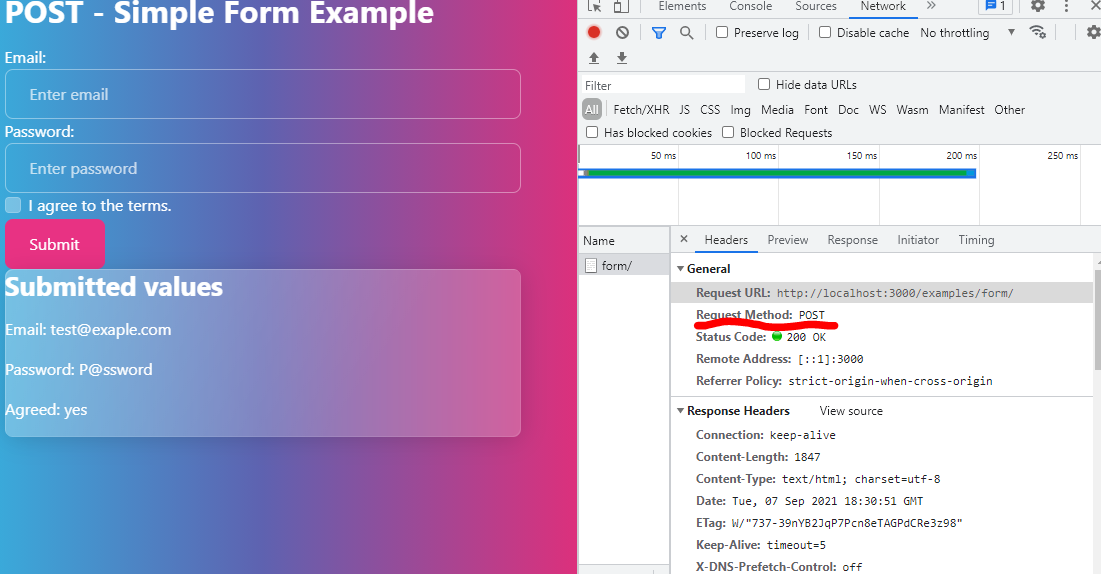
Learn more about popular template engines:   
<https://blog.logrocket.com/top-express-js-template-engines-for-dynamic-html-pages/>

# HTML Forms to interact with the Server

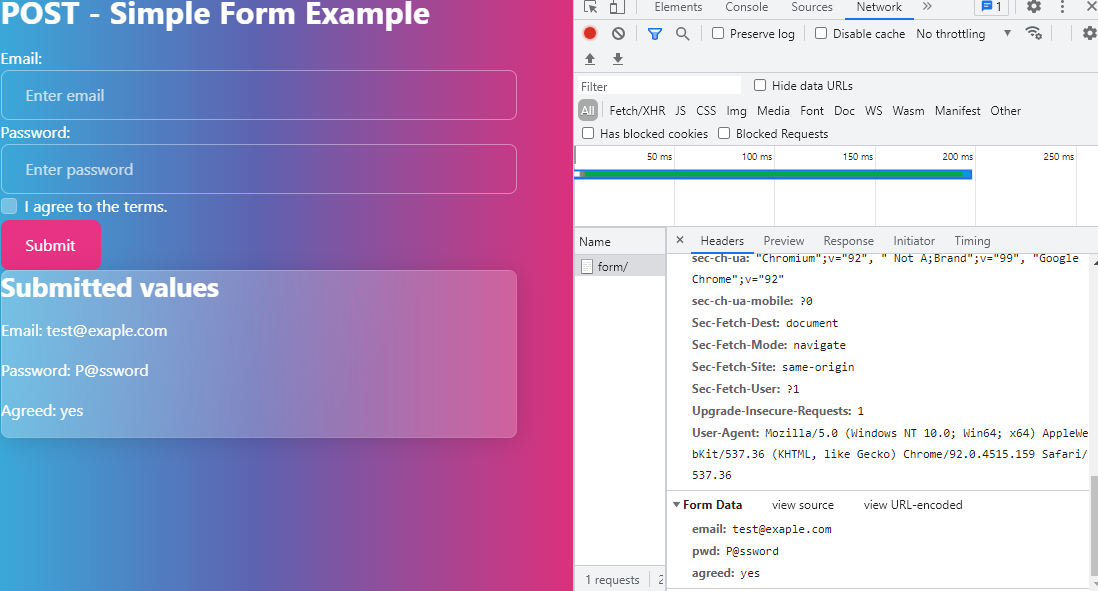
One of the ways users can interact with websites is to submit html forms. The user submits a form and the browser encodes the data in a way that the server can interpret.

Developers can then use the user submitted data to adjust the response to the request.

Form data usually send use the POST request method. To see the browser submitting the form to the server open the developer tools. In the Network tab -> select the file and examine the “Headers” section



If you scroll down further in the window you will notice the **Form Data**



# Create an Interactive Form

Create a new file in the view folder call it **form-example.hbs**

**IMPORTANT: the input tag MUST have a name attribute for the form to work**

**\view\form-example.hbs – add the following code to**

<form action="/examples/form/" method="post" >

<h1>{{ title }}</h1>

<div class="form-group">

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

<!--IMPORTANT the input tag MUST have a name attribute to post to the server ie. name="email"-->

<input type="email" class="form-control" placeholder="Enter email" name="email" id="email" required/>

</div>

<div class="form-group">

<label for="pwd">Password:</label>

<input type="password" class="form-control" placeholder="Enter password" name="pwd" id="pwd" required/>

</div>

<div class="form-group form-check">

<label class="form-check-label">

<input class="form-check-input" type="checkbox" name="agreed" value="yes" required/> I agree to the terms.

</label>

</div>

<button type="submit" class="btn btn-primary">Submit</button>

</form>

{{#if isSubmitted}}

<div class="card p-2">

<h2>Submitted values</h2>

<p>Email: {{submittedEmail}}</p>

<p>Password: {{submittedPassword}}</p>

<p>Agreed: {{submittedAgreed}}</p>

</div>

{{/if}}

Notice that the form method="post" – otherwise the browser will submit the form as part of the URL query string.

**\routes\examples.js – add the following code above the last line that reads -- module.exports = router;**

/\* GET content for path: http://localhost:3000/examples/form \*/

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

res.render('form-example', {

title: 'GET - Simple Form Example',

});

});

/\* POST submit form data to path : http://localhost:3000/examples/form \*/

router.post('/form', function(req, res, next) {

res.render('form-example', {

title: 'POST - Simple Form Example',

isSubmitted: req.body.agreed === 'yes', // check to see if user checked the checkbox <input name="agreed" value="yes" />

// example <input name="someName" /> then we use req.body.someName access the user input value

submittedEmail: req.body.email, // <input name="email" /> then we use: req.body.email

submittedPassword: req.body.pwd, // <input name="pwd" /> then we use: req.body.pwd

submittedAgreed: req.body.agreed, // <input name="agreed" value="yes" /> then we use: req.body.agreed

});

});

Restart the webserver for the changes to take effect - Navigate to <http://localhost:3000/examples/form/>

Exercise:   
What happens when you specify the ***value*** attribute in the **email** input tag?   
value="{{submittedEmail}}"

What happens when you specify the ***value*** attribute in the **password** input tag? Why does it not work?  
value="{{submittedPassword}}"

Exercise: What happens when you add the ***checked*** attribute in the **checkbox** input tag?   
{{#if isSubmitted}}checked{{/if}}

# Query Strings to interact with the Server

Web browsers can also send data to the server using query strings in the address bar.

Example URL: <http://localhost:3000/examples/form/?email=t%40t.ca&pwd=SAGFSD&agreed=yes>

* Query strings are appended to the end of the website path and start with a ‘?’ question mark character
* Query Strings are split up into the name value pairs with the use of ‘&’(amperes and)
* The name is on the left of the = equal sign and the value is in the right. (i.e*. some\_name****=****some+value*)

# Code support for Query Strings

Query strings are usually handed by the GET request hander. So lets modify the router.get call.

**\routes\examples.js – add the code highlighted in yellow**

/\* GET content for path: http://localhost:3000/examples/form/ \*/

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

res.render('form-example', {

title: 'GET - Simple Form Example',  
// check to see if the agreed url param is yes - http://localhost:3000/examples/form/?agreed=yes

isSubmitted: req.query.agreed === 'yes',

// example …/form/?someName=someValue then we use req.query.someName access the value of the URL parameter

submittedEmail: req.query.email, // …/form/?someName=someValue&email=t%40t.ca then we use:req.query.email

submittedPassword: req.query.pwd, // …/form/?someName=someValue&pwd=passW0rd then we use: req.query.pwd

submittedAgreed: req.query.agreed, // …/form/?someName=someValue&agreed=yes then we use: req.query.agreed

});

});

Restart the webserver for the changes to take effect - Navigate to:

<http://localhost:3000/examples/form/?email=t%40t.ca&pwd=SAGFSD&agreed=yes>

**Exercise:** What happens when you switch the form input tag from ‘post’ to ‘get’ then submit the form?

<form action="/examples/form/" method="get" >

# Upload files

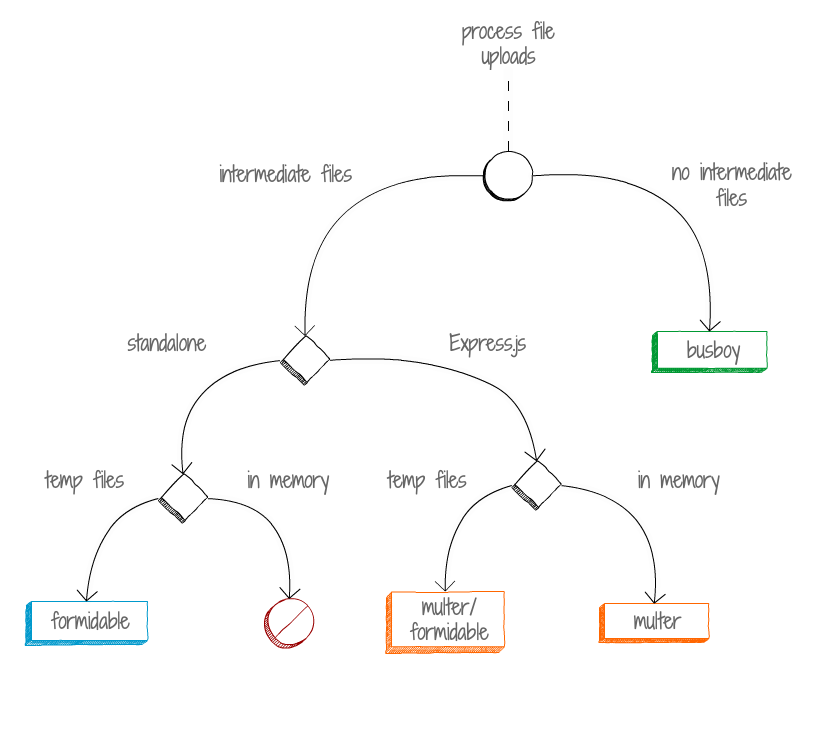
Often websites will allow users to upload files as inputs. The most common version of this is attaching photos to social media posts.

* The underlying concepts are the same regardless of the type of file being uploaded to the server
* The server then needs to deal with the uploaded file. The most common way is to store the uploaded file in a temporary directory
* Then the developer decided what to do with the file on the server

Express will work with multiple file upload packages – there are pros and cons for each package. The common ones are

* Formidable
* Busboy
* Multer
* Multiparty

The following figure is a quick overview of the differences



Learn more about file upload packages: <https://bytearcher.com/articles/formidable-vs-busboy-vs-multer-vs-multiparty/>

## Install Multer

For ease of use we will use Multer in this course. But we will need to install the package to our lo1serveronly project.

From the terminal type the following command:

npm i multer

## Using Multer to handle uploaded files.

* First you must require multer in the router file. This is usually done directly below the statement that requires Express
* Then you need to configure multer by creating an instance with a set of options.
  + In the example below we set destination directory path and set the new object to the ‘upload’ variable
  + multer will use the windows user’s temp directory if the destination is not specified

const express = require('express');

const router = express.Router();

// add packages that will handle the file uploads

// https://www.npmjs.com/package/multer#usage

const multer = require('multer');

// IMPORTANT: ensure you created the destination folder before using it below

const upload = multer({dest: 'public/uploads/'});

When using express router we need to pass in the new instance of multer (in the example below the variable containing the multer instance is called ‘upload’) to the post function.

In this case we are using the any() function to accept any uploaded files but this should be used with caution.

router.post('/upload', upload.any(),(req, res, next) => {…

* Within the handler function multer exposes a new property to the req (request object) called ‘files’ ( i.e. req.files).
* The files property is an array of file objects.
* Each file object has a numeric index

Structure: req.files = [{field:'file1',…}, {field:'file2',…},{field:'file3',…}]

The best practice is to use the fields() function and specify the exact fields our app will accept

router.post('/upload', upload.fields([{name: 'file1', maxCount: 3}, {name: 'file2', maxCount: 3}]),

(req, res, next) => {…

* In this case within the handler function req.files has named object structure

Structure: req.files = {  
file1:[{field:'file1',…}, {field:'file1',…},{field:'file1',…}],   
file2:[{field:'file2',…}, {field:'file2',…},{field:'file2',…}]  
}

Learn more about Multer Usage : <https://www.npmjs.com/package/multer#usage>

# Simple File Upload and text inputs

**Create the folder lo1serveronly\public\uploads**

**\routes\examples.js – add the code in yellow to the top of the file**

const express = require('express');

const router = express.Router();

// add packages that will handle the file uploads

// https://www.npmjs.com/package/multer#usage

const multer = require('multer');

// IMPORTANT: ensure you created the destination folder before using it below

const upload = multer({dest: 'public/uploads/'});

Create a new template in the views folder to handle uploaded files.

IMPORTANT: The form tag MUST have the enctype="multipart/form-data" attribute to upload files.

**Create new file form-upload.hbs in \view\form-upload.hbs – add the following code to**

<form action="/examples/upload/" method="post" enctype="multipart/form-data" >

<h1>{{ title }}</h1>

<div class="form-group">

<label for="title1">File 1 Title:</label>

<input type="text" class="form-control" placeholder="Enter file title" name="title1" id="title1"/>

</div>

<div class="form-group">

<label for="file1">File 1:</label>

<input type="file" class="form-control" placeholder="Select file" name="file1" id="file1" />

</div>

<div class="form-group">

<label for="desc1">File 1 Description:</label>

<textarea class="form-control" placeholder="Enter file description" name="desc1" id="desc1" rows="3" ></textarea>

</div>

<div class="form-group mt-2">

<label for="title2">File 2 Title:</label>

<input type="text" class="form-control" placeholder="Enter file title" name="title2" id="title2"/>

</div>

<div class="form-group">

<label for="file2">File 3:</label>

<input type="file" class="form-control" placeholder="Select file" name="file2" id="file2" />

</div>

<div class="form-group">

<label for="desc2">File 2 Description:</label>

<textarea class="form-control" placeholder="Enter file description" name="desc1" id="desc1" rows="3" ></textarea>

</div>

<button type="submit" class="btn btn-primary">Submit</button>

</form>

{{#if isSubmitted}}

<div class="card p-2">

<h2>Submitted Files and Description</h2>

<p>File 1 Title: {{file1Title}}</p>

<p>File 1: {{file1Info.originalname}}</p>

<p>File 1 Description: {{file1Description}}</p>

<p class="mt-2">File 2 Title: {{file2Title}}</p>

<p>File 2: {{file2Info.originalname}}</p>

<p>File 2 Description: {{file2Description}}</p>

</div>

{{/if}}

Notice that the form method="post" – otherwise the browser will submit the form as part of the URL query string.

**\routes\examples.js – add the following code above the last line that reads -- module.exports = router;**

// TODO: code to handle GET request to /examples/upload/ path

/\* GET content for path: http://localhost:3000/examples/upload \*/

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

res.render('upload-files', {

title: 'GET - Upload Form Example',

});

});

// TODO: code to handle POST request to /examples/upload/ path

/\* POST submit form data to path : http://localhost:3000/examples/upload \*/

// BEST PRACTICE: specify the only fields the app app will accept

// https://www.npmjs.com/package/multer#usage

router.post('/upload',upload.any(),

(req, res, next) => {

// output file array infor to console to see what is available

console.log('uploaded files:\n');

console.log(req.files);

res.render('upload-files', {

title: 'POST - Upload Form Example',

isSubmitted: true, // check to see if the file title is filled in

});

});

Exercise: Check the console for the structure of the req.files.

Try to display the uploaded file ‘orginalname’ in the template

Best practice using upload.fields()

**\routes\examples.js – add the following code above the last line that reads -- module.exports = router;**

// TODO: code to handle GET request to /examples/upload/ path

/\* GET content for path: http://localhost:3000/examples/upload \*/

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

res.render('upload-files', {

title: 'GET - Upload Form Example',

});

});

// TODO: code to handle POST request to /examples/upload/ path

/\* POST submit form data to path : http://localhost:3000/examples/upload \*/

// BEST PRACTICE: specify the only fields the app app will accept

// https://www.npmjs.com/package/multer#usage

router.post('/upload', upload.fields([{name: 'file1', maxCount: 1}, {name: 'file2', maxCount: 1}]),

(req, res, next) => {

// output file array infor to console to see what is available

console.log('uploaded files:\n');

console.log(req.files);

// declare variables to store the uploaded file information

let file1;

let file2;

// check to see if the corresponding files were uploaded otherwise use new object

file1 = req.files['file1'] ? req.files['file1'][0] : {originalname: 'not uploaded'};

file2 = req.files['file2'] ? req.files['file2'][0] : {originalname: 'not uploaded'};

console.log(file1);

res.render('upload-files', {

title: 'POST - Upload Form Example',

isSubmitted: true, // check to see if the file title is filled in />

file1Title: req.body.file1Title,

file2Title: req.body.file2Title,

file1Description: req.body.desc1,

file2Description: req.body.desc2,

file1Info: file1,

file2Info: file2,  
 });

});