ĐẠI HỌC TÔN ĐỨC THẮNG
TON DUC THANG UNIVERSITY

503106

ADVANCED WEB PROGRAMMING

CHAPTER 4: FORM HANDLING

LESSON 04 – FORM HANDLING

OUTLINE

1. Sending Client Data to the Server
2. HTML Forms
3. Encoding
4. Different Approaches to Form Handling
5. Form Handling with Express
6. File Uploads
7. Using Fetch to Send Form Data
8. File Uploads with Fetch

Sending Client Data to the Server

- Two options for sending client data to the server: the querystring and the request body
- The query string: GET request
- The request body: POST request
- POST is generally recommended for form submission

HTML Forms

```
<form action="/process" method="POST">
  <input type="hidden" name="hush" val="hidden, but not secret!">
  <div>
    <label for="fieldColor">Your favorite color: </label>
    <input type="text" id="fieldColor" name="color">
  </div>
  <div>
    <button type="submit">Submit</button>
  </div>
</form>
```

Encoding

- When the form is submitted (either by the browser or via AJAX), it must be encoded
- It defaults to *application/x-wwwform-urlencoded*
- If you need to upload files, you're forced to use the *multipart/form-data* encoding type, which is not handled directly by Express (we will be discussing an alternative shortly)

Different Approaches to Form Handling

- There are two things to consider when processing forms:
  - what path handles the form (the action)
  - and what response is sent to the browser
- It is quite common to use the same path for displaying the form and processing the form: these can be distinguished because the former is a GET request, and the latter is a POST request
  - If you take this approach, you can omit the action attribute on the form
- The other option is to use a separate path to process the form
  - This approach might be preferred if you have multiple URLs that use the same submission mechanism

Different Approaches to Form Handling

- Whatever path you use to process the form, you have to decide what response to send back to the browser. Here are your options:
  - Direct HTML response
  - 303 redirect. This is the recommended method for responding to a form submission request

Different Approaches to Form Handling

- Where does the redirection point to?
  - Redirect to dedicated success/failure pages
  - Redirect to the original location with a flash message
  - Redirect to a new location with a flash message

Form Handling with Express

- If you're using POST, you'll have to link in middleware to parse the URL-encoded body
- Install the body-parser middleware (**npm install body-parser**)
- Then, link it in:

```
const bodyParser = require('body-parser')

app.use(bodyParser.urlencoded({ extended: true }))
```

- http://expressjs.com/en/resources/middleware/body-parser.html
- **req.body** now becomes available for you, and that's where all of your form fields will be made available

Form Handling with Express

```
<h2>Sign up for our newsletter to receive news and specials!</h2>
<form class="form-horizontal" role="form" action="/newsletter-signup/process?form=newsletter"
method="POST">
  <input type="hidden" name="_csrf" value="{{csrf}}">
  <div class="form-group">
    <label for="fieldName" class="col-sm-2 controllabel">Name</label>
    <div class="col-sm-4">
      <input type="text" class="form-control" id="fieldName" name="name">
    </div>
  </div>
  <div class="form-group">
    <label for="fieldEmail" class="col-sm-2 controllabel">Email</label>
    <div class="col-sm-4">
      <input type="email" class="form-control" required id="fieldEmail" name="email">
    </div>
  </div>
  <div class="form-group">
    <div class="col-sm-offset-2 col-sm-4">
      <button type="submit" class="btn btn-primary">Register</button>
    </div>
  </div>
</form>
```

Form Handling with Express

```
const express = require('express')
const expressHandlebars =
require('express-handlebars')
const app = express()
// configure Handlebars view
engine
app.engine('handlebars',
expressHandlebars({
        defaultLayout: 'main',
}))
app.set('view engine',
'handlebars')
app.use(express.static(__dirname +
'/public'))
app.use(express.json());
app.use(express.urlencoded());
```

```
app.get('/newsletter-signup', (req, res) => {
  // we will learn about CSRF later...for now, we just provide
a dummy value
  res.render('newsletter-signup', { csrf: 'CSRF token goes
here'})
})
app.post('/newsletter-signup/process', (req, res) => {
  console.log('Form (from querystring): ' + req.query.form)
  console.log('CSRF token (from hidden form field): ' +
req.body._csrf)
  console.log('Name (from visible form field): ' +
req.body.name)
  console.log('Email (from visible form field): ' +
req.body.email)
  res.redirect(303, '/newsletter-signup/thank-you')
})
app.get('/newsletter-signup/thank-you', (req, res) =>
res.render('newsletter-signup-thank-you'))
```

File Uploads

- There are four popular and robust options for multipart form processing: busboy, multiparty, formidable, and multer
- In this demo we will use **multiparty (npm install multiparty)**
- Let’s create a file upload form
- Note that we must specify *enctype="multipart/form-data"* to enable file uploads.

```
<h2>Vacation Photo Contest</h2>
<form class="form-horizontal" role="form" enctype="multipart/form-data"
method="POST" action="/vacation-photo">
  <input type="hidden" name="_csrf" value="{{csrf}}">
  …
  <div class="form-group">
    <label for="fieldPhoto" class="col-sm-2 controllabel">Vacation
photo</label>
    <div class="col-sm-4">
      <input type="file" class="form-control" required accept="image/*"
id="fieldPhoto" name="photo">
    </div>
  </div>
  <div class="form-group">
    <div class="col-sm-offset-2 col-sm-4">
      <button type="submit" class="btn btn-primary">Register</button>
    </div>
  </div>
</form>
```

File Uploads

```
const multiparty = require('multiparty')
app.get('/vacation-photo', (req, res) => {
  res.render('vacation-photo')
})
app.post('/vacation-photo', (req, res) => {
  const form = new multiparty.Form()
  form.parse(req, (err, fields, files) => {
    if (err) return res.status(500).send(err.message)
    console.log('field data: ', fields)
    console.log('files: ', files)
    res.redirect(303, '/vacation-photo-thank-you')
  })
})
```

Results

```
field data:  { _csrf: [ 'my string' ] }
files:  {
  photo: [
    {
      fieldName: 'photo',
      originalFilename: '5.gif',
      path: '/var/folders/vf/2lmhd95d6rx4y34d39lk5lg40000gn/T/qgCeXs6QXDIJYfYUCel21iAw.gif',
      headers: [Object],
      size: 31662
    }
  ]
}
```

Move file

```
var fs = require('fs’)

var oldPath = 'old/path/file.txt’

var newPath = 'new/path/file.txt’

fs.rename(oldPath, newPath, function (err) {
                if (err) throw err;
                console.log('Successfully renamed - AKA moved!’);
 })
```

Using Fetch to Send Form Data

- *fetch()* allows you to make network requests similar to XMLHttpRequest (XHR)
- The main difference is that the Fetch API uses Promises
  - Promises Tutorial: https://web.dev/promises/
- Introduction to Fetch: https://developers.google.com/web/updates/2015/03/introduction-to-fetch

Using Fetch to Send Form Data

- Let's start with the frontend code.
- We don't need to specify an *action* or *method*, and we'll wrap our form in a container element that will make it easier to display our "thank you" message:

```
<div id="newsletterSignupFormContainer">
  <form class="form-horizontal role="form" id="newsletterSignupForm">
    <!-- the rest of the form contents are the same... -->
  </form>
</div>
```

Using Fetch to Send Form Data

- Then we'll have a script that intercepts the form submit event and cancels it (using *Event#preventDefault*) so we can handle the form processing ourselves

```
<script>
document.getElementById('newsletterSignupForm').addEventListener('submit', evt => {
    evt.preventDefault()
    const form = evt.target
    const body = JSON.stringify({
      _csrf: form.elements._csrf.value,
      name: form.elements.name.value,
      email: form.elements.email.value,
    })
    const headers = { 'Content-Type': 'application/json' }
    const container = document.getElementById('newsletterSignupFormContainer')

fetch('/api/newsletter-signup', { method: 'post', body, headers }).then(resp => {
        if(resp.status < 200 || resp.status >= 300)
          throw new Error(`Request failed with status ${resp.status}`)
        return resp.json()
      })
      .then(json => {
        container.innerHTML = '<b>Thank you for signing up!</b>'
      })
      .catch(err => {
        container.innerHTML = `<b>We're sorry, we had a problem ` +
          `signing you up. Please <a href="/newsletter">try again</a>`
      })
  })
</script>
```

Using Fetch to Send Form Data

- Now in our server file (app.js), make sure we're linking in middleware that can parse JSON bodies, before we specify our two endpoints:

```
app.use(bodyParser.json())
//...
app.get('/newsletter', handlers.newsletter)
app.post('/api/newsletter-signup',handlers.api.newsletterSignup)
```

Using Fetch to Send Form Data

- Now we'll add those endpoints to our lib/handlers.js file:

```
exports.newsletter = (req, res) => {
  // we will learn about CSRF later...for now, we just
  // provide a dummy value
  res.render('newsletter', { csrf: 'CSRF token goes here' })
}
exports.api = {
  newsletterSignup: (req, res) => {
    console.log('CSRF token (from hidden form field): ' + req.body._csrf)
    console.log('Name (from visible form field): ' + req.body.name)
    console.log('Email (from visible form field): ' + req.body.email)
    res.send({ result: 'success' })
  },
}
```

File Uploads with Fetch

- Happily, using *fetch* for file uploads is nearly identical to letting the browser handle it.
- Consider this JavaScript to send our form contents using fetch:

```
<script>
  document.getElementById('vacationPhotoContestForm')
    .addEventListener('submit', evt => {
      evt.preventDefault()
      const body = new FormData(evt.target)
      const container = document.getElementById('vacationPhotoContestFormContainer')
      fetch('/api/vacation-photocontest', { method: 'post', body })
        .then(…)
    })
</script>
```

- The important detail to note here is that we convert the form element to a *FormData* object

File Uploads with Fetch

- We'll add a new route to our app:

```
app.get('/vacation-photo-ajax', handlers.vacationPhotoContestAjax)
```

- In our handlers:

```
exports.vacationPhotoContestAjax = (req, res) => {
  res.render('vacation-photo-ajax')
}
```

- And our view file (vacation-photo-ajax.handlebars):

```
<h2>Vacation Photo Contest</h2>
<div id="vacationPhotoContestFormContainer">
<form class="form-horizontal" role="form" id="vacationPhotoContestForm" enctype="multipart/form-data">
  ...
</form>
</div>
```

File Uploads with Fetch

- File uploads process:

```
app.post('/api/vacation-photo-contest', (req, res) => {
  const form = new multiparty.Form()
  form.parse(req, (err, fields, files) => {
    if (err) return handlers.api.vacationPhotoContestError(req, res, err.message)
    handlers.api.vacationPhotoContest(req, res, fields, files)
  })
})
```

File Uploads with Fetch

- That’s all there is to it! our handler is almost exactly the same:

```
exports.api.vacationPhotoContest = (req, res, fields, files) =>
{
  console.log('field data: ', fields)
  console.log('files: ', files)
  res.send({ result: 'success' })
}
```

Improving File Upload UI

- Some of the most popular file upload frontends are as follows:
  - jQuery File Upload (https://github.com/blueimp/jQuery-File-Upload)
  - Uppy (https://github.com/transloadit/uppy)
  - file-upload-with-preview (https://github.com/promosis/file-upload-with-preview)
  - And many more …