File Upload Revisited

Storage and Serving to Users

File Upload Recap

- A few weeks ago, we looked at uploading files with the **multer** middleware
- While we had the code to do file upload, there were still some missing pieces
  - How do we store files in the database?
  - How do we serve files up in the browser?

Multer Recap - Multipart Form

- Whenever you have a file input, your enctype should be `multipart/form-data`, and your method should be post

```
<form enctype="multipart/form-data" method="post">
    <input name="photo" type="file"/>
    <button>Save Photo</button>
</form>
```

Multer Recap - Uploader

- The multer module creates an "uploader" class, that generates file handling middleware
- All of the files go in the specified dest folder
- Files don't keep their original names, or file extensions

```
const multer = require("multer");
const uploader = multer({ dest: "uploads/" });
```

Multer Recap - Uploader Middleware

- The middleware functions take in the name of the file input
- `uploader.single()` provides one file at `req.file`
- `uploader.array()` provides an array of files at `req.files`

```
app.post("/upload/one", uploader.single("photo"), function(req, res) {
    console.log("File " + req.file.filename + " is " + req.file.size + " bytes");
});


app.post("/upload/many", uploader.array("photos"), function(req, res) {
    for (let i = 0; i < req.files.length; i++) {
        console.log("File #" + i + " is " + req.files[i].size + " bytes");
    }
});
```

So Where Do We Store It?

- The user's given us a file, it's been placed in `/uploads`, but how does it get used now?
- Like any form submitted data we don't want to use, we now want to save the file to the database
- But we don't want to save the *content* of the file, we just want to save the *information*
- By storing the (newly generated) name of the file, we can reference it later
    - Think of the filename as an ID, like a user ID

Make a Table For Them

- We could have our own `files` / `images` / songs table that keeps track of the file's metadata
  - Typically we save metadata along with the file, things like `size`, `mimetype`, `originalname`
- Anything that wants to use these files can define a one-to-many or many-to-many relationship with the file table
  - It's best to not add any other relationship columns to the File than a userid, to promote reusability
  - Imagine how Facebook uses the same photo entity to provide album photos, profile photos, wall post photos etc.
  - Each of those entities have a `pictureId` column, rather than the `picture` table having columns for every entity it could have a relationship with

Make a Table (Code Example)

```
// models/file.js
const Sequelize = require("sequelize");
const sql = require("../util/sql");

const File = sql.define("file", {
    id: {
        type: Sequelize.STRING,
        primaryKey: true,
    },
    size: Sequelize.INTEGER,
    name: Sequelize.STRING,
    mime: Sequelize.STRING,
});

module.exports = File;
```

Make a Table (Code Example)

```
/* Assume regular express setup above... */
const File = require("./models/file");
const multer = require("multer");
const uploader = multer({ dest: "uploads/" });

app.post("/upload", uploader.single("file"), function(req, res) {
    File.create({
        id: req.file.filename,
        size: req.file.size,
        name: req.file.originalname,
        mime: req.file.mimetype,
    })
    .then(function(file) {
        res.send("File uploaded!");
    })
    .catch(function(err) {
        res.status(400).send("Bad file!");
    });
});
```

Making a File Publicly Accessible

- So now we've uploaded our file, and we've saved the file's ID to the database, now what?
- Our files live in /uploads, but this is **not** accessible to public traffic
  - We want it that way, so that we could have private, inaccessible documents
- But we typically *do* have a static assets folder that *is* publicly available
- If we could move things over there, then we could link to them like other assets

Making a File Publicly Accessible (Code)

```
const fs = require("fs-extra");

// Inside of app.post("/upload")
File.create(/* ... */).then(function(file) {
    const dest = "assets/files/" + req.file.filename;
    fs.copy(req.file.path, dest).then(function() {
        res.send("File uploaded!");
    });
})
.catch(function(err) {
    res.status(400).send("Bad file!");
});
```

- Now the file could be accessed at [hostname]:[port]/assets/files/[fileid]

Side Note: Using `fs-extra`

- Node's `fs` module on its own leaves a lot to be desired
  - It's missing a lot of handy functions like `copy()` and `move()`
  - It always uses node-style callbacks instead of promises
- Because of that, the community has created `fs-extra`, a version of `fs` that improves it in many ways
- If you ever see this module included, just know that it does everything `fs` does and more
- And if you copy `fs` code that `require()`s it, you'll need to install and require it too

Side Note: File Extension

- By default, multer doesn't save the file with an extension in the name
- However, when we move things to assets, we probably want that
- Fortunately the `path` module provides an easy way of adding that back in
- Calling `path.extname(file.originalname)` will get us the extension of the file they originally uploaded

```
// Top of file...
const path = require("path");

// In your route...
const ext = path.extname(file.originalname);
const dest = "assets/files/" + file.filename + ext;
fs.copy(file.path, dest).then(/* ... */);
```

Bringing It All Together

- Phew, that's a heck of a lot of code to put in a route...
- How about we *extend our Model* instead?

```
// models/file.js
const fs = require("fs-extra");
const path = require("path");
const sql = require("../util/sql");

const File = sql.define("file", /* ... */);

File.saveFile = function(file) {
    return File.create({
        id: file.filename,
        size: file.size,
        name: file.originalname,
        mime: file.mimetype,
    })
    .then(function(dbFile) {
        const ext = path.extname(file.originalname);
        const dest = "assets/files/" + file.filename + ext;
        return fs.copy(file.path, dest);
    });
};

module.exports = File;
```

Bringing It All Together (app.js)

```
/* Assume regular express setup above... */
const File = require("./models/file");
const multer = require("multer");
const uploader = multer({ dest: "uploads/" });

app.post("/upload", uploader.single("file"), function(req, res) {
    File.saveFile(req.file)
        .then(function() {
            res.send("File uploaded!");
        })
        .catch(function(err) {
            res.status(400).send("Bad file!");
        });
});
```

Challenge: DropDox

- We're creating a file sharing app called "DropDox"
- It's got basic user auth, a file model, and a file list page already implemented
- The only thing that's missing is the upload form, which is what you'll be adding
- See the readme on Github for all of the information
    - **Read the readme completely** or you will have a hard time knowing what to do!
- When you're finished, there are some challenge goals in there too

Additional Reading

- Previous `multer` Slides
- `fs-extra` Docs