User Accounts with Secure Authentication

- We used **Express** to handle http requests/responses formatting for our server. If we did not have this then we would have to parse the incoming request for the route and request type. We would also have to build our own http response but express handles the incoming route and request type by “app.get” for get request, “app.post” for post request. “app.get” takes in a string and a handler where the string will be the route we will accept and the handler will handle what we need. This handler takes in a (request, and response) where the request is the incoming request to the server and response is the response from the server. We can build a response to send back to the client by simply doing response.send(content) to build a response to send content back to the client. If we did not use Express, we would have to build a http response from scratch. In order to get the html pages, we used res.render(html file) which sends the html pages. Express overall helps us not worry about the small things such as building a response,parsing a response and etc so we can write things that will improve the logic and functionality of our app.
- Features we used Express to handle for User Account:
    - Sign up
        - Express will handle routing for the http request/response for “/signup” when a user first creates an account.
        - When the user first creates an account it will take in the request body:
            - Email
            - Password
            - Confirm password
            - First name, Last name
        - We will fetch that information with a POST request and confirm if the email exists already in the MongoDB database.
        - For secure authentication for the passwords first we compare the password and confirm password to see if they match. If the passwords match then we hash the password using **bcrypt** library which will hash passwords for us. After hashing, we compare the hashed password and if it's the same then the user is created.
    - Sign in
        - Express will handle routing for the http/request response for “/login” when a user attempts to sign in
        - When a user attempts to log in it will first find the email in the database and then compare the input password password from the user to the hashed password that we stored in our MongoDB database.
        - If the password is the same then we will remove the token that is already associated with the user and create a new token for that user. This token will be attached to every single request and will be only valid for 24 hrs.
        - Users won’t be able to visit certain pages without a valid token.
- For this part pug does not render data or fill in templates for login so it will just send the login html file.

- Body-parser parsed the body which was sent in the initial post request. The request was made using ajax so the body was structured in json format. Body-parser let us access the body as a javascript object.
- Cookie-parse cookie parser easily reads and writes the cookies. Cookie parser turns the cookies into javascript objects so it can be easily read. It also sets cookies automatically by adding a new field to the object. We used this to check if there is a token and validate it when they come to the website. We also set the cookies when a user successfully created an account or logs in.

Users to see all users who are currently logged in

- Pug implementation
    - Pug will get data on all current logged in users and will display each one on the left of the page. A for loop is used on the data to map all the years.

Users can send direct messages (DM) to other users with notifications when a DM is received

- Socket io implementation
    - Socket io lets users talk in real time with one another.
    - Each user has a socket started when they log in (rooms). The socket will keep track of incoming and outgoing messages. So if someone messages the user, then the user will receive a frame with the message and javascript will display the message or notify them.
- Pug implementation
    - Pug is going to template the home page with user information. Pus will get user information, direct messages, online users, and current dm receiver.
    - With the data described above, Pug will conditionally template the data into an HTML file. For example, in the chat the user's message will be on the right whereas the recipient will be on the left.
    - Pug will also load the user's name, image, etc in the HTML file.

Users can share some form of multimedia content which is stored and hosted on your server

- Profile Pictures
    - When a user first creates their account, they are given a default profile picture. This picture is attached and can be seen at the top left corner of the home screen and with their messages. Each user has the ability to change their profile picture by clicking on their current profile picture at the top left corner of the home screen. Here they can upload any photo of their choice and save it as their profile picture. Once it has been uploaded, the new picture is now attached and can be seen at the top left corner of the home screen and with their messages.These

changes are all done by our database. Each user object has a profile picture variable that is automatically set to the default profile picture. The image requests that are requested at the top left corner of the home screen and the messages are all a request for the profile picture variable for the logged in user. When the user makes a change to their profile picture, we find that user in the database and update its profile picture variable to the path of the now locally stored image that they just uploaded. Now, when the user goes back to the home page, the page makes a request for their profile picture variable, and since it has been set in the database as the path of the uploaded image, their new profile picture is displayed and will be displayed on any request that requests their profile picture.

- Multer Implementation
    - Multer is middleware for node.js that handles multipart/form data. We are using multer specifically for image upload and local storage. When a user wants to change their profile picture, the upload and local storage process is taken care of by multer. We chose to use multer so we would not have to code our own image processing code. We do not have to worry about reading the bytes until we get all of the image data (like we had to do for homework 3), multer takes care of that.

Live interactions between users via WebSockets (Cannot be text)

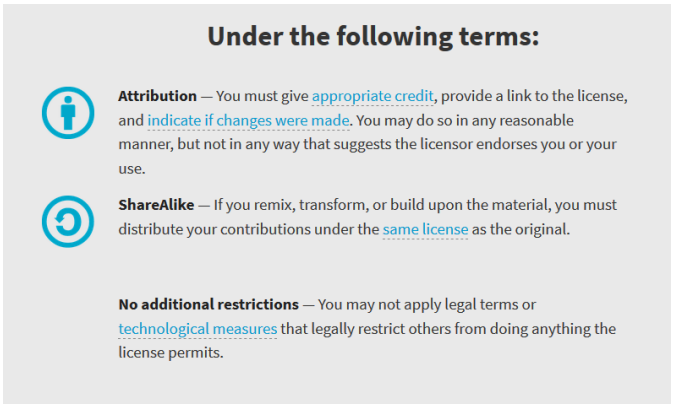
- In progress

Libraries/Framework/Services used

Express

- What does this technology (library/framework/service) accomplish for you?
    - If we did not have this then we would have to parse the incoming request for the route and request type. We would also have to build our own http response but express handles the incoming route and request type by “app.get” for get request, “app.post” for post request. “app.get” takes in a string and a handler where the string will be the route we will accept and the handler will handle what we need. This handler takes in a (request, and response) where the request is the incoming request to the server and response is the response from the server. We can build a response to send back to the client by simply doing response.send(content) to build a response to send content back to the client. If we did not use Express, we would have to build a http response from scratch. In order to get the html pages, we used res.render(html file) which sends the html pages. Express overall helps us not worry about the small things such as building a response,parsing a response and etc so we can write things that will improve the logic and functionality of our app.

- How does this technology accomplish what it does?
    - For our sign in page we used express to:
        - Features we used Express to handle for User Account:
            - Sign up
                - Express will handle routing for the http request/response for “/signup” when a user first creates an account.
                - When the user first creates an account it will take:
                    - Email
                    - Password
                    - Confirm password
                    - First name, Last name
                - We will fetch that information with a POST request and confirm if the email exists already in the MongoDB database.
                - For secure authentication for the passwords first we compare the password and confirm password to see if they match. If the passwords match then we hash the password using **bcrypt** library which will hash passwords for us. After hashing, we compare the hashed password and if it's the same then the user is created.
            - Sign in
                - Express will handle routing for the http/request response for “/login” when a user attempts to sign in
                - When a user attempts to log in it will first find the email in the database and then compare the input password password from the user to the hashed password that we stored in our MongoDB database.
                - If the password is the same then we will remove the token that is already associated with the user and create a new token for that user. This token will be attached to every single request and will be only valid for 24 hrs.
                - Users won’t be able to visit certain pages without a valid token.
- What license(s) or terms of service apply to this technology?
    - As stated by the Creative Commons Attribution-ShareAlike 3.0 United States License. :
        - **Share** — copy and redistribute the material in any medium or format
        - **Adapt** — remix, transform, and build upon the material
        - for any purpose, even commercially.



Socket io

- What does this technology (library/framework/service) accomplish for you?
- How does this technology accomplish what it does?
- What license(s) or terms of service apply to this technology?

Pug

- What does this technology (library/framework/service) accomplish for you?
    - We used Pug as a template for the data that we received from our database to customize specific user's information.
- How does this technology accomplish what it does?
    - We used pug to template our data that we received from requests from Express. We will display the data from from the database on our home screen using pug as a template to replace html tags to display the following:
        - User profile
        - Online user list
        - DMs
    - Link : https://github.com/pugjs/pug
- What license(s) or terms of service apply to this technology?
    - License **:** MIT

Multer

- **What does this technology (library/framework/service) accomplish for you?**
    - Multer is middleware for node.js that handles multipart/form data. We are using multer specifically for image upload and local storage. When a user wants to change their profile picture, the upload and local storage process is taken care of by multer. We chose to use multer so we would not have to code our own image processing code. We do not have to worry about reading the bytes until we get all of the image data (like we had to do for homework 3), multer takes care of that.

- **How does this technology accomplish what it does?**
    - **Local Image Storage**
        - Multer is used to create a local storage engine for the profile picture uploads. We create a variable called storage and set it equal to the **diskStorage** property of multer. We then pass in two objects, destination and filename. The destination object is the destination in which we want the images to be locally stored. In our case, we set the destination to ./views/uploads. Filename is a function and it takes in three parameters; the request, the file itself, and a callback. The callback will be used to name the image when we store it locally (so we are not taking the original name of the file).The file naming convention we are using at the moment is “ProPic-<current timestamp>.<file type>”. We get the current timestamp by using Date.now() and we get the file type by calling .extname() on the file. When the image has been processed by the server, this storage variable will use the diskStorage property of multer to rename the file, and save it to the given destination.
        - **Disk Storage Source Code**
            - https://github.com/expressjs/multer/blob/master/storage/disk.js
            - The diskStorage property of multer exposes the _handleFile function. This function takes req, file, cb as parameters in that order specifically and takes the responsibility of storing the file at a specific destination as well as naming the file so that the file can be accessed in the future. For storing the file at the specific destination, the _handleFile function calls the getDestination function. This function takes req, file, and cb as parameters and returns the destination specified by the programmer (/views/uploads in our case). Now, _handleFile has the destination. For the naming, _handleFile calls the getFilename function which takes req, file, cb as parameters. This function takes the naming convention (“ProPic-<current timestamp>.<file type>” in our case) and converts it to hex. It needs to be in hex because that is how _handleFIle handles the naming. Now, that _handleFile has the destination and how the file will be named, it joins them together and sends it out via a file stream. This stream then initiates the local storage with the naming convention.

- **_handleFile**

```
DiskStorage.prototype._handleFile = function _handleFile (req, file, cb) {
  var that = this

  that.getDestination(req, file, function (err, destination) {
    if (err) return cb(err)

    that.getFilename(req, file, function (err, filename) {
      if (err) return cb(err)

      var finalPath = path.join(destination, filename)
      var outStream = fs.createWriteStream(finalPath)

      file.stream.pipe(outStream)
      outStream.on('error', cb)
      outStream.on('finish', function () {
        cb(null, {
          destination: destination,
          filename: filename,
          path: finalPath,
          size: outStream.bytesWritten
        })
      })
    })
  })
}
```

- **getDestination and getFilename**

```
function getFilename (req, file, cb) {
  crypto.randomBytes(16, function (err, raw) {
    cb(err, err ? undefined : raw.toString('hex'))
  })
}

function getDestination (req, file, cb) {
  cb(null, os.tmpdir())
}
```

- **Image Uploads/File Processing**
  - Multer is used for the image upload and processing. We initialize the upload variable called upload. The **multer** object itself contains all of functionality for gathering all of the image data and storing. Thus, we set

the upload variable equal to the multer object (multer()). The multer object requires the data and a storage object so we pass it the storage engine mentioned above. Multer has the ability to upload single or multiple images at once. We are only allowing the user to upload one profile picture at a time so end of the initialization with .single() to indicate that this is a single image upload. When the user uploads the image, it makes a post request on the route /profilepictureupload. At that route, the data received from the request is fed into the upload function (which is basically multer()). All of the image data is processed and then fed into the storage engine, which stores it locally.

- **Multer Object Source Code**
  - https://github.com/expressjs/multer/blob/master/lib/make-middleware.js
  - The multer object generates the middleware needed for multipart/form-data. This object requires a storage object that is of type StorageEngine (diskStorage is used in our case). When the object processes the data, it will save the data in this storage object.
  - The setup function that is generated by the multer object is makeMiddleware. This is the function that sets up all of the parts needed for the file processing and storage. One of the variables it initializes is storage. This variable is set to the StorageEngine object passed in by the programmer. Once the setup is complete, it is time to process the data. This is done by the function busboy. This function takes all of the variables set by makeMiddleware and the multipart req as parameters. This function processes the multipart data via a filestream. This filestream is then parsed to get various information such as the fieldname, original name, encoding, and mime type. This data is then saved into an object called file. Now that we have all of the information, it's time to store it. Busboy does this by calling the _handleFile property of the storage variable initialized by makeMiddleware (the StorageEnginer object passed in by the programmer). This function takes the file objects as a parameter and stores the data in its storage object.

- **MakeMiddleware**

```
function makeMiddleware (setup) {
  return function multerMiddleware (req, res, next) {
    if (!is(req, ['multipart'])) return next()

    var options = setup()

    var limits = options.limits
    var storage = options.storage
    var fileFilter = options.fileFilter
    var fileStrategy = options.fileStrategy
    var preservePath = options.preservePath

    req.body = Object.create(null)

    var busboy

    try {
      busboy = new Busboy({ headers: req.headers, limits: limits, preservePath: preservePath })
    } catch (err) {
      return next(err)
    }

    var appender = new FileAppender(fileStrategy, req)
    var isDone = false
    var readFinished = false
    var errorOccured = false
    var pendingWrites = new Counter()
    var uploadedFiles = []

    function done (err) {
      if (isDone) return
      isDone = true

      req.unpipe(busboy)
      drainStream(req)
      busboy.removeAllListeners()

      onFinished(req, function () { next(err) })
    }
```

- **Filestream and file object of busboy**

```
busboy.on('file', function (fieldname, fileStream, filename, encoding, mimetype) {
  // don't attach to the files object, if there is no file
  if (!filename) return fileStream.resume()

  // Work around bug in Busboy (https://github.com/mscdex/busboy/issues/6)
  if (limits && Object.prototype.hasOwnProperty.call(limits, 'fieldNameSize')) {
    if (fieldname.length > limits.fieldNameSize) return abortWithCode('LIMIT_FIELD_KEY')
  }

  var file = {
    fieldname: fieldname,
    originalname: filename,
    encoding: encoding,
    mimetype: mimetype
  }
```

- **_handleFile of storage**

```
        storage._handleFile(req, file, function (err, info) {
          if (aborting) {
            appender.removePlaceholder(placeholder)
            uploadedFiles.push(extend(file, info))
            return pendingWrites.decrement()
          }

          if (err) {
            appender.removePlaceholder(placeholder)
            pendingWrites.decrement()
            return abortWithError(err)
          }

          var fileInfo = extend(file, info)

          appender.replacePlaceholder(placeholder, fileInfo)
          uploadedFiles.push(fileInfo)
          pendingWrites.decrement()
          indicateDone()
        })
      })
    })
```

- **Other Documentation (via https://www.npmjs.com/package/multer):**

`storage`

`DiskStorage`

The disk storage engine gives you full control on storing files to disk.

```
var storage = multer.diskStorage({
  destination: function (req, file, cb) {
    cb(null, '/tmp/my-uploads')
  },
  filename: function (req, file, cb) {
    cb(null, file.fieldname + '-' + Date.now())
  }
})

var upload = multer({ storage: storage })
```

There are two options available, `destination` and `filename`. They are both functions that determine where the file should be stored.

`destination` is used to determine within which folder the uploaded files should be stored. This can also be given as a `string` (e.g. `'/tmp/uploads'`). If no `destination` is given, the operating system's default directory for temporary files is used.

Note: You are responsible for creating the directory when providing `destination` as a function. When passing a string, multer will make sure that the directory is created for you.

`filename` is used to determine what the file should be named inside the folder. If no `filename` is given, each file will be given a random name that doesn't include any file extension.

Note: Multer will not append any file extension for you, your function should return a filename complete with an file extension.

Each function gets passed both the request (`req`) and some information about the file (`file`) to aid with the decision.

`.single(fieldname)`

Accept a single file with the name `fieldname`. The single file will be stored in `req.file`.

`multer(opts)`

Multer accepts an options object, the most basic of which is the `dest` property, which tells Multer where to upload the files. In case you omit the options object, the files will be kept in memory and never written to disk.

By default, Multer will rename the files so as to avoid naming conflicts. The renaming function can be customized according to your needs.

The following are the options that can be passed to Multer.

Key	Description
`dest` or `storage`	Where to store the files
`fileFilter`	Function to control which files are accepted
`limits`	Limits of the uploaded data
`preservePath`	Keep the full path of files instead of just the base name

In an average web app, only `dest` might be required, and configured as shown in the following example.

```
var upload = multer({ dest: 'uploads/' })
```

- **What license(s) or terms of service apply to this technology?**
  - MIT