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?
    - Express Source Code : https://github.com/expressjs/express
    - res.send([body]) -> This sends a response with body inside of it. Body can be a Buffer, string, object, array, and much more.

```
res.send = function send(body) {
  var chunk = body;
  var encoding;
  var req = this.req;
  var type;

  // settings
  var app = this.app;

  // allow status / body
  if (arguments.length === 2) {
    // res.send(body, status) backwards compat
    if (typeof arguments[0] !== 'number' && typeof arguments[1] === 'number') {
      deprecate('res.send(body, status): Use res.status(status).send(body) instead');
      this.statusCode = arguments[1];
    } else {
      deprecate('res.send(status, body): Use res.status(status).send(body) instead');
      this.statusCode = arguments[0];
      chunk = arguments[1];
    }
  }

  // disambiguate res.send(status) and res.send(status, num)
  if (typeof chunk === 'number' && arguments.length === 1) {
    // res.send(status) will set status message as text string
    if (!this.get('Content-Type')) {
      this.type('txt');
    }

    deprecate('res.send(status): Use res.sendStatus(status) instead');
    this.statusCode = chunk;
    chunk = statuses[chunk]
  }
```

This checks to see if the body is a number, if it is a number then it sends you a deprecation warning. In order to send a number, express has made it so that you need to use the sendStatus method.

```
// write strings in utf-8
if (typeof chunk === 'string') {
  encoding = 'utf8';
  type = this.get('Content-Type');

  // reflect this in content-type
  if (typeof type === 'string') {
    this.set('Content-Type', setCharset(type, 'utf-8'));
  }
}

// determine if ETag should be generated
var etagFn = app.get('etag fn')
var generateETag = !this.get('ETag') && typeof etagFn === 'function'

// populate Content-Length
var len
if (chunk !== undefined) {
  if (Buffer.isBuffer(chunk)) {
    // get length of Buffer
    len = chunk.length
  } else if (!generateETag && chunk.length < 1000) {
    // just calculate length when no ETag + small chunk
    len = Buffer.byteLength(chunk, encoding)
  } else {
    // convert chunk to Buffer and calculate
    chunk = Buffer.from(chunk, encoding)
    encoding = undefined;
    len = chunk.length
  }

  this.set('Content-Length', len);
}

// populate ETag
var etag;
if (generateETag && len !== undefined) {
  if ((etag = etagFn(chunk, encoding))) {
    this.set('ETag', etag);
  }
}
```

This checks if the chunk is a string, if it is a string then you're going to set the content type as utf-8. It then determines if the Etag should be created. Next it populates the content-length. This is basically done writing the HTTP header and how its going to add the body of content length. We get the string and turn it into a buffer so that we can send it along with the HTTP header. We get the content length by getting the buffer size.

```
  switch (typeof chunk) {
    // string defaulting to html
    case 'string':
      if (!this.get('Content-Type')) {
        this.type('html');
      }
      break;
    case 'boolean':
    case 'number':
    case 'object':
      if (chunk === null) {
        chunk = '';
      } else if (Buffer.isBuffer(chunk)) {
        if (!this.get('Content-Type')) {
          this.type('bin');
        }
      } else {
        return this.json(chunk);
      }
      break;
  }
```

This checks the other cases from when it is not a string or if that string file is a HTML file. It does the same thing with the other types where it converts it into a buffer and gets the content length. This is basically getting all the required information to build a HTTP response with the proper headers and data.

```
// strip irrelevant headers
if (204 === this.statusCode || 304 === this.statusCode) {
  this.removeHeader('Content-Type');
  this.removeHeader('Content-Length');
  this.removeHeader('Transfer-Encoding');
  chunk = '';
}

if (req.method === 'HEAD') {
  // skip body for HEAD
  this.end();
} else {
  // respond
  this.end(chunk, encoding);
}

return this;
```

This checks the status code, if it is 204 or 304, we can disregard the response headers so we can remove them. Since

we have all we need we return this since this has all the required information to build the response with the body in it to send back to the user.

- res.render(view [, locals] [, callback]) ->This sends the rendered html string as a response. The view is the html file and the local/callbacks are optional. View is the path to the html file that is sent in the response.

```
app.render = function render(name, options, callback) {
  var cache = this.cache;
  var done = callback;
  var engines = this.engines;
  var opts = options;
  var renderOptions = {};
  var view;

  // support callback function as second arg
  if (typeof options === 'function') {
    done = options;
    opts = {};
  }

  // merge app.locals
  merge(renderOptions, this.locals);

  // merge options._locals
  if (opts._locals) {
    merge(renderOptions, opts._locals);
  }

  // merge options
  merge(renderOptions, opts);

  // set .cache unless explicitly provided
  if (renderOptions.cache == null) {
    renderOptions.cache = this.enabled('view cache');
  }

  // primed cache
  if (renderOptions.cache) {
    view = cache[name];
  }

  // view
  if (!view) {
    var View = this.get('view');

    view = new View(name, {
      defaultEngine: this.get('view engine'),
      root: this.get('views'),
      engines: engines
    });

    if (!view.path) {
      var dirs = Array.isArray(view.root) && view.root.length > 1
        ? 'directories "' + view.root.slice(0, -1).join('", "') + '" or "' + view.root[view.root.length - 1] + '"'
        : 'directory "' + view.root + '"'
      var err = new Error('Failed to lookup view "' + name + '" in views ' + dirs);
      err.view = view;
      return done(err);
    }

    // prime the cache
    if (renderOptions.cache) {
      cache[name] = view;
    }
  }

  // render
  tryRender(view, renderOptions, done);
};
```

It adds all the render options and checks for a view which is just a file path that needs to be rendered. This is also where the html templating is changed via the view engine that you set. You can set a view template by simply requiring it. If no view engine is set then nothing happens it just just sends as it is.

- res.cookie(name, value[options] ->Sets a cookie name to value like how you would set a cookie. You pass in the name and the value you want to set the cookie. When you use a cookie-parser middle-ware, the cookies are simply returned in a key value pair. You can access it by its key such as req.cookie[“name”]

```
res.cookie = function (name, value, options) {
  var opts = merge({}, options);
  var secret = this.req.secret;
  var signed = opts.signed;

  if (signed && !secret) {
    throw new Error('cookieParser("secret") required for signed cookies');
  }

  var val = typeof value === 'object'
    ? 'j:' + JSON.stringify(value)
    : String(value);

  if (signed) {
    val = 's:' + sign(val, secret);
  }

  if ('maxAge' in opts) {
    opts.expires = new Date(Date.now() + opts.maxAge);
    opts.maxAge /= 1000;
  }

  if (opts.path == null) {
    opts.path = '/';
  }

  this.append('Set-Cookie', cookie.serialize(name, String(val), opts));

  return this;
};
```

```
/**
 * Append additional header `field` with value `val`.
 *
 * Example:
 *
 *    res.append('Link', ['<http://localhost/>', '<http://localhost:3000/>']);
 *    res.append('Set-Cookie', 'foo=bar; Path=/; HttpOnly');
 *    res.append('Warning', '199 Miscellaneous warning');
 *
 * @param {String} field
 * @param {String|Array} val
 * @return {ServerResponse} for chaining
 * @public
 */

res.append = function append(field, val) {
  var prev = this.get(field);
  var value = val;

  if (prev) {
    // concat the new and prev vals
    value = Array.isArray(prev) ? prev.concat(val)
      : Array.isArray(val) ? [prev].concat(val)
      : [prev, val];
  }

  return this.set(field, value);
};
```

If you are using a cookie parser, it will add the key value of the cookie by using the append function. Res.cookie is also protected by a secret so if you don't have the secret, the cookie info is encrypted but if you do then it will just return the cookie.

- req.body -> This contains key value pairs of the data submitted by the requester. This parameter is passed into the request body so that the information can then be used in the callback function to process the data and send back a response

```
req.param = function param(name, defaultValue) {
  var params = this.params || {};
  var body = this.body || {};
  var query = this.query || {};

  var args = arguments.length === 1
    ? 'name'
    : 'name, default';
  deprecate('req.param(' + args + '): Use req.params, req.body, or req.query instead');

  if (null != params[name] && params.hasOwnProperty(name)) return params[name];
  if (null != body[name]) return body[name];
  if (null != query[name]) return query[name];

  return defaultValue;
};
```

If the request has a body then it will parse the body, if it has a query it will parse the query and if it has params it will parse the params. All these are parsed by the name

- app.post(path, callback [, callback ...]) -> takes in a path and listens to a callback that executes when the path is met. This path is defined for POST http request. This is generally used when an incoming request has a body that the server needs to process. This pretty much uses the same code as the .get function.

```
var method = req.method;
var has_method = route._handles_method(method);
```

```
// get pathname of request
var path = getPathname(req);
```

- app.get(path, callback [, callback ...]) -> takes in a path that it listens to and a callback that executes when the path is met. This path is defined for GET HTTP methods

```
methods.forEach(function(method){
  app[method] = function(path){
    if (method === 'get' && arguments.length === 1) {
      // app.get(setting)
      return this.set(path);
    }

    this.lazyrouter();

    var route = this._router.route(path);
    route[method].apply(route, slice.call(arguments, 1));
    return this;
  };
});
```

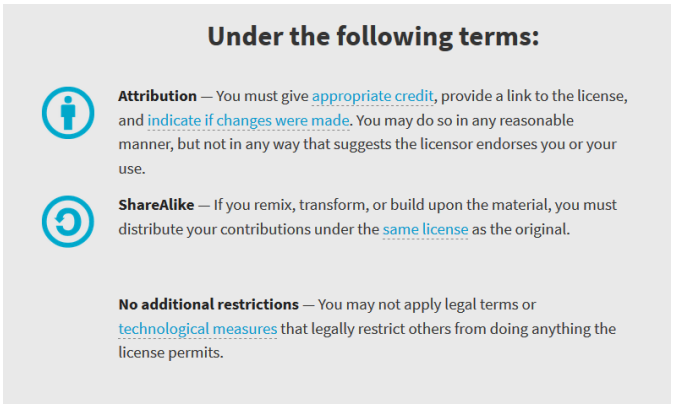
```
function Route(path) {
  this.path = path;
  this.stack = [];

  debug('new %o', path)

  // route handlers for various http methods
  this.methods = {};
}
```

The get method defines a GET path that gets added to the router. When the router gets a response from that path, it will execute the callback function that runs and sends back a response. The default path is / but the path can be defined as any string that comes behind the / such as /home, /dashboard, and etc

- 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?
    - This library allows us to set up a socket connection with our browser. Instead of creating our own handshake, the lib handles that for us as well as frame parsing. Instead of reading the frames sent by the client, we get the straight up json of the message. This is made possible by the socket parsing the web frame before sending it to the server so we can access the data without any parsing. Another thing that the lib helps us handle is sending a certain type of socket and being able to handle that socket. For example, on the client side I send a socket to “newConnect” and this socket can then be handled along with its message in the server that catches the “newConnect” with that message being sent. The cool part about this is that the server can then also send a socket to a “route” that can be handled in the client side of the browser. Without the lib we probably would’ve had to create our own socket routes to handle each socket request as well as decode them. For this project this lib helped up a lot in creating a socket connection and maintaining it. The whole player seeing other player’s movement is entirely controlled by the sockets. I defined a socket route called "gameStart” where it signals all the players in the game that a new player has joined and then renders that player in everyone’s screen. This also grabs all the positions of the other players in the game and tells them to the person who connected to the game. Without this library, I would've had to write a route to handle the gameStart route and parse the userId as well as the initial random positions given to it but with the lib i’m able to use it like I use json. I also defined a route in the server called “movement” where every time the user in the front end moves, the position would then be sent to the “movement” socket route in the server which sends it back to the “newConnection” route in the front end to update everyone’s position of that user on the screen to the newly moved position. This makes it so that other players can see each other move on the screen when using the arrow keys. I also have an initial socket called “init” which simply adds the user’s into a dictionary along with its socket. I do this so that I’m able to differentiate between the user sockets and only send messages to users who it was intended to be sent to. This is also used to send everyone that is connected to an online

message when a new socket is added and update their status on the database. I have a function in the client side that triggers when a user sends a message to another user. I used the lib to define a “msg” route in the server which simply calls a function to add a chat to the database for the 2 sender and receiver of the message. With this lib I don’t need to parse the webframe so I can simply access the message contents like a json object to create a chat message and save it to the database. Another thing this was used for is the disconnect where I remove the user from the user socket list and notify all the users that have their sockets that the user has disconnected by sending a message using the emit to the “leftGame” route in the client side. To send those messages to the user I simply used the .emit() function which takes in a string route that the socket would listen to as well as the content that I want to send to that route and the lib would simply handle create the webframe for me and sending it so I can access the same information in the client/server side of the server. One of the most useful parts of this socket lib is that it's able to decode a webframe for the information as well as be able to construct a webframe to send information.

- How does this technology accomplish what it does?
- Source: https://github.com/socketio/socket.io/blob/master/client-dist/socket.io.js

```
}, {
  key: "connect",
  value: function connect(fn) {
    return this.open(fn);
  }
  /**
   * Called upon transport open.
   *
   * @private
   */
```

```
}, {
  key: "ondata",
  value: function ondata(data) {
    this.decoder.add(data);
  }
  /**
   * Called when parser fully decodes a packet.
   *
   * @private
   */
```

Decode data to make frame

```
}, {
  key: "ondecoded",
  value: function ondecoded(packet) {
    this.emitReserved("packet", packet);
  }
  /**
   * Called upon socket error.
   *
   * @private
```

On decoded create the packet

```
}, {
  key: "disconnect",
  value: function disconnect() {
    return this._close();
  }
  /**
   * Called upon engine close.
   *
   * @private
   */
```

- Close the socket

```
 */
function Socket(io, nsp, opts) {
  var _this;

  _classCallCheck(this, Socket);

  _this = _super.call(this);
  _this.receiveBuffer = [];
  _this.sendBuffer = [];
  _this.ids = 0;
  _this.acks = {};
  _this.flags = {};
  _this.io = io;
  _this.nsp = nsp;
  _this.ids = 0;
  _this.acks = {};
  _this.receiveBuffer = [];
  _this.sendBuffer = [];
  _this.connected = false;
  _this.disconnected = true;
  _this.flags = {};

  if (opts && opts.auth) {
    _this.auth = opts.auth;
  }

  if (_this.io._autoConnect) _this.open();
  return _this;
}
/**
```

- Socket object, used to create frame to send

```
/**
 * Connects a client to a namespace.
 *
 * @param {String} name - the namespace
 * @param {Object} auth - the auth parameters
 * @private
 */
private connect(name: string, auth: object = {}): void {
  if (this.server._nsps.has(name)) {
    debug("connecting to namespace %s", name);
    return this.doConnect(name, auth);
  }

  this.server._checkNamespace(
    name,
    auth,
    (dynamicNspName: Namespace<ListenEvents, EmitEvents> | false) => {
      if (dynamicNspName) {
        debug("dynamic namespace %s was created", dynamicNspName);
        this.doConnect(name, auth);
      } else {
        debug("creation of namespace %s was denied", name);
        this._packet({
          type: PacketType.CONNECT_ERROR,
          nsp: name,
          data: {
            message: "Invalid namespace",
          },
        });
      }
    }
  );
}
```

-

```
  /**
   * Writes a packet to the transport.
   *
   * @param {Object} packet object
   * @param {Object} opts
   * @private
   */
  _packet(packet: Packet, opts?: any): void {
    opts = opts || {};
    const self = this;

    // this writes to the actual connection
    function writeToEngine(encodedPackets: any) {
      // TODO clarify this.
      if (opts.volatile && !self.conn.transport.writable) return;
      for (let i = 0; i < encodedPackets.length; i++) {
        self.conn.write(encodedPackets[i], { compress: opts.compress });
      }
    }

    if ("open" === this.conn.readyState) {
      debug("writing packet %j", packet);
      if (!opts.preEncoded) {
        // not broadcasting, need to encode
        writeToEngine(this.encoder.encode(packet)); // encode, then write results to engine
      } else {
        // a broadcast pre-encodes a packet
        writeToEngine(packet);
      }
    } else {
      debug("ignoring packet write %j", packet);
    }
  }

  /**
```

- Takes the packet and write it

```
 */
private ondata(data): void {
  // try/catch is needed for protocol violations (GH-1880)
  try {
    this.decoder.add(data);
  } catch (e) {
    this.onerror(e);
  }
}

/**
 * Called when parser fully decodes a packet.
 *
 * @private
 */
private ondecoded(packet: Packet): void {
  if (PacketType.CONNECT === packet.type) {
    if (this.conn.protocol === 3) {
      const parsed = url.parse(packet.nsp, true);
      this.connect(parsed.pathname!, parsed.query);
    } else {
      this.connect(packet.nsp, packet.data);
    }
  } else {
    const socket = this.nsps.get(packet.nsp);
    if (socket) {
      process.nextTick(function () {
        socket._onpacket(packet);
      });
    } else {
      debug("no socket for namespace %s", packet.nsp);
    }
  }
}
```

- ```
  /**
  ```
  On receiving data, add it to the decoder and when it is decoded send it to the pathname
- What I think these snippets of code does is it creates a handshake with our browser and defines some basic things like “connection” for the initial connection. There is also an encode and decode that takes in data and decodes/encoded it to send the webframe. The decode is used by both the client and the server to decode a webframe for information and the encode is used to encode json into a webframe to send to both the client or server.
- When a connection is established, socket io will create a Socket class for the incoming connection. The Socket class will contain important information about the connection like the acknowledgement numbers, flags, receive buffer, and nsp (ip address) etc. This information will be used to form and receive incoming packets from the client. The class also has subevents mapped to. The connect event will keep the connection open by upgrading the handshake and update the class variables. The emit event will create a packet object which contains properties of a socket frame and the data being sent. The emit event takes an event type so the client can parse the event. Once the packet is created, it will send the frame to the client. The event function will essentially close the connections between the client and free the class.
- The socket.io api is inspired from the Node.js EventEmitter (source : https://nodejs.org/docs/latest/api/events.html#events_events , https://github.com/nodejs/node/blob/v16.1.0/lib/events.js), which allows you to register with .on and then emit the events to users with .emit.
  ```

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

Http

- What does this technology (library/framework/service) accomplish for you?
    - It creates a server that can listen to a port and handle incoming request
- How does this technology accomplish what it does?

```
function createServer(opts, requestListener) {
  return new Server(opts, requestListener);
}
```

-
- Starts a http connection that listens to the given port. Whenever the user access that port with a specific route, it will use the requestListeners to handle them

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

The lexer file is in charge of formatting the pug file into a list of objects. The object contains information of the html tag like attributes, class, and type of tag. Each object is associated with the line number.

Source: pug/packages/pug-lexer/index.js

The lexer file starts by calling getTokens which forms a while loop till the end of the file and calls the advance function. The advance calls multiple functions in the file which build an object with properties that are used to parse the template. Advance is ordered so it can build the object from top down so the first call is to blank which checks if the line is blank so the other functions do not get called.

One way that pug used to template code into html is by indentation. Html tags are nested by indentation. The indentation is found by lexer.scanIndentation. This function goes over the current line and checks if the line starts with indentation, in which it will set a property to an object stating the current object is nested.

When there is a conditional in the pug file it will call lexer's conditional function. The function checks if the current line is conditional using switches to check if the first word is if, else if, and else. The function adds the conditional to the tokens and then returns it so it can go to the next line. There are also an each, eachOf, and while function that is similar to conditional but checks if there is a loop. (lines 984-1131)

To get the id pug finds a # in the line and to get the class pug finds all . in the line and added a property to the corresponding object (functions id and className). Attributes start and end with parentheses. Anything inside the parentheses are treated exactly like html attributes. The attribute function separates everything by space then parses each value by =. If there is text in a tag, the text function is called which uses regex to validate the text and add it to the object.

pug/packages/pug-linker/index.js

After the lexer creates the list of objects, the linker file is called to flatten the objects down and level out the inheritance. The link function iterates through the list of objects and checks the object's type property. There are two types which are NamedBlock and Block. NamedBlock are the objects that are well defined and they do not need to be parsed whereas Block are not well defined. The link function also calls the extend function which checks for inheritance by checking the object's mode which can be append, prepend, and replace. The mode tells the function where inside the parent the tag should be.

pug/packages/pug-code-gen/index.js

After all the previous files are executed, the code-gen file will take the list of objects and turn them into a Html string. The compile function is the first call back function to get called. The compile function creates a buffer that loops through all the objects and their child objects to build the buffer up. A lot of the code in the file is conditionals and string manipulation.

- 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

bodyParser

- **What does this technology (library/framework/service) accomplish for you?**
    - The bodyParser.json library parses json and only looks at the request body where “Content-Type” matches its type option and then it returns a body object of the parsed data from the request.
- **How does this technology accomplish what it does?**
    - https://github.com/expressjs/body-parser/blob/master/lib/types/json.js

```
function json (options) {
  var opts = options || {}

  var limit = typeof opts.limit !== 'number'
    ? bytes.parse(opts.limit || '100kb')
    : opts.limit
  var inflate = opts.inflate !== false
  var reviver = opts.reviver
  var strict = opts.strict !== false
  var type = opts.type || 'application/json'
  var verify = opts.verify || false

  if (verify !== false && typeof verify !== 'function') {
    throw new TypeError('option verify must be function')
  }

  // create the appropriate type checking function
  var shouldParse = typeof type !== 'function'
    ? typeChecker(type)
    : type

  function parse (body) {
    if (body.length === 0) {
      // special-case empty json body, as it's a common client-side mistake
      // TODO: maybe make this configurable or part of "strict" option
      return {}
    }

    if (strict) {
      var first = firstchar(body)

      if (first !== '{' && first !== '[') {
        debug('strict violation')
        throw createStrictSyntaxError(body, first)
      }
    }

    try {
      debug('parse json')
      return JSON.parse(body, reviver)
    } catch (e) {
      throw normalizeJsonSyntaxError(e, {
        message: e.message,
        stack: e.stack
      })
```

    - bodyParser starts by creating a middleware to parse the json bodies. The function starts by building options that the user can input. The options are:
        - **inflate** which compresses bodies when set to true and deflates when set to false.
        - **Limit** which controls the max request body size the default is set to ‘100kb’
        - **Reviver** which is passed into the JSON.parse function

        - **Strict** when set to true will only accept data types arrays and objects and false it will accept anything.
        - **Type** used to determine what media type is going to be parsed such as application/json
        - **Verify** used to determine whether or not the supplied request,response,buffer of the raw body and encoding is appropriate for parsing.
    - Then there is a function parse(body) which takes in a body as an argument and first checks if the body is empty or not. Then it checks if the regex variable FIRST_CHAR_REGEXP to see if it contains “{“ or “[“ if it does its invalid.
    - After checking for a valid body to be parsed it tries to return a JSON object using the library **JSON.parse** (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/parse) which takes in the function argument body, and reviver(an option that passes in the object through the reviver function that transforms or changes the object before being returned). If it fails it will throw a Json syntax error message.
    - Next, it has a return function jsonParser which takes in a request,response, next as arguments. It checks if the request has a body or not if it doesn’t it will skip that request and return.
    - Then, it checks if the request is the appropriate type to be parsed by calling shouldParse(req) which checks whether or not it is the appropriate type to be parsed.

```
97
98    return function jsonParser (req, res, next) {
99      if (req._body) {
100       debug('body already parsed')
101       next()
102       return
103     }
104
105     req.body = req.body || {}
106
107     // skip requests without bodies
108     if (!typeis.hasBody(req)) {
109       debug('skip empty body')
110       next()
111       return
112     }
113
114     debug('content-type %j', req.headers['content-type'])
115
116     // determine if request should be parsed
117     if (!shouldParse(req)) {
118       debug('skip parsing')
119       next()
120       return
121     }
122
123     // assert charset per RFC 7159 sec 8.1
124     var charset = getCharset(req) || 'utf-8'
125     if (charset.substr(0, 4) !== 'utf-') {
126       debug('invalid charset')
127       next(createError(415, 'unsupported charset "' + charset.toUpperCase() + '"', {
128         charset: charset,
129         type: 'charset.unsupported'
130       }))
131       return
132     }
133
134     // read
135     read(req, res, next, parse, debug, {
136       encoding: charset,
137       inflate: inflate,
138       limit: limit,
139       verify: verify
```

- After confirming that it is appropriate to parse it gets the charset by calling getCharset(req)(this function is built in their code which parses the contentType for the encoding by parsing the request and after parsing it they will convert it to lowercase.) which gets the charset ‘utf-8’ in lower case. It makes sure that it’s exactly ‘utf-8’ otherwise an error is thrown.
- Finally, it calls read(req,res,next,parse,debug{} which includes the encoding,inflate,limit, and verify.
- The read function is built here : https://github.com/expressjs/body-parser/blob/master/lib/read.js

```
 * Read a request into a buffer and parse.
 *
 * @param {object} req
 * @param {object} res
 * @param {function} next
 * @param {function} parse
 * @param {function} debug
 * @param {object} options
 * @private
 */

function read (req, res, next, parse, debug, options) {
  var length
  var opts = options
  var stream

  // flag as parsed
  req._body = true

  // read options
  var encoding = opts.encoding !== null
    ? opts.encoding
    : null
  var verify = opts.verify

  try {
    // get the content stream
    stream = contentstream(req, debug, opts.inflate)
    length = stream.length
    stream.length = undefined
  } catch (err) {
    return next(err)
  }

  // set raw-body options
  opts.length = length
  opts.encoding = verify
    ? null
    : encoding
```

- read() takes in arguments the request,response,next,parse,debug, and options that are provided to it in its arguments. It first reads the options and verifies that .verify determines that it is appropriately parsed. Then it gets the content stream from the request, with its built in function **contentstream**, it holds the content length and then sets the stream content length to undefined.
- It again checks if the supplied argument encoding from options is valid.

```
      }

      // parse
      var str = body
      try {
        debug('parse body')
        str = typeof body !== 'string' && encoding !== null
          ? iconv.decode(body, encoding)
          : body
        req.body = parse(str)
      } catch (err) {
        next(createError(400, err, {
          body: str,
          type: err.type || 'entity.parse.failed'
        }))
        return
      }

      next()
    })
  }
```

- It will parse the body here by checking that the body is first not empty and the encoding is valid too. It will decode the body given ‘utf-8’ encoding that is deemed valid. And then it will parse the body with the variable str.
- **contentstream** function is the built in function which reads the content stream of the request and builds it and returns it.

```
142   */
143
144  function contentstream (req, debug, inflate) {
145    var encoding = (req.headers['content-encoding'] || 'identity').toLowerCase()
146    var length = req.headers['content-length']
147    var stream
148
149    debug('content-encoding "%s"', encoding)
150
151    if (inflate === false && encoding !== 'identity') {
152      throw createError(415, 'content encoding unsupported', {
153        encoding: encoding,
154        type: 'encoding.unsupported'
155      })
156    }
157
158    switch (encoding) {
159      case 'deflate':
160        stream = zlib.createInflate()
161        debug('inflate body')
162        req.pipe(stream)
163        break
164      case 'gzip':
165        stream = zlib.createGunzip()
166        debug('gunzip body')
167        req.pipe(stream)
168        break
169      case 'identity':
170        stream = req
171        stream.length = length
172        break
173      default:
174        throw createError(415, 'unsupported content encoding "' + encoding + '"', {
175          encoding: encoding,
176          type: 'encoding.unsupported'
177        })
178    }
179
180    return stream
181  }
```

- It uses the .headers function from https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers
- It gets the encoding and length from the request with .header
- Again, it checks if the encoding is valid.
- It then uses the switch statement to determine which type of encoding it is handling, ‘deflate’, ‘gzip’ or ‘identity’ and it will build the stream depending on which case it is.
- It will return the content stream after properly building it.

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

cookieParser

- **What does this technology (library/framework/service) accomplish for you?**
  - By using cookieParser() we don’t have to parse the cookie from the header ourselves. It will parse the cookie header and populate req.cookies with an object keyed by cookie names.

- **How does this technology accomplish what it does?**

```
function cookieParser (secret, options) {
  var secrets = !secret || Array.isArray(secret)
    ? (secret || [])
    : [secret]

  return function cookieParser (req, res, next) {
    if (req.cookies) {
      return next()
    }

    var cookies = req.headers.cookie

    req.secret = secrets[0]
    req.cookies = Object.create(null)
    req.signedCookies = Object.create(null)

    // no cookies
    if (!cookies) {
      return next()
    }

    req.cookies = cookie.parse(cookies, options)

    // parse signed cookies
    if (secrets.length !== 0) {
      req.signedCookies = signedCookies(req.cookies, secrets)
      req.signedCookies = JSONCookies(req.signedCookies)
    }

    // parse JSON cookies
    req.cookies = JSONCookies(req.cookies)

    next()
  }
}

/**
```

  - It first tries to parse a cookie header and then populate “req.cookies’ with an object keyed by cookie names. The function cookieParser takes two arguments secret which is an array of strings representing the secret arguments given to the function and options which is an Object.
  - It checks if there are any secrets provided in the array by the user.
  - It will set the request secrets to the argument provided by the user, cookies to a null object and signed cookies to a null object.

- It will then try to parse the cookies from the request headers with the function .parse from : https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/parse, this will base parse the json with the given arguments in this case it was the cookie var from the request headers.
- It will then set req.cookies = JSONCookies(req.cookies) which is another built in function from the library.
- The JSONCookie library will parse the JSON cookie string.

```
function JSONCookie (str) {
  if (typeof str !== 'string' || str.substr(0, 2) !== 'j:') {
    return undefined
  }

  try {
    return JSON.parse(str.slice(2))
  } catch (err) {
    return undefined
  }
}

/**
 * Parse JSON cookies.
 *
 * @param {Object} obj
 * @return {Object}
 * @public
 */

function JSONCookies (obj) {
  var cookies = Object.keys(obj)
  var key
  var val

  for (var i = 0; i < cookies.length; i++) {
    key = cookies[i]
    val = JSONCookie(obj[key])

    if (val) {
      obj[key] = val
    }
  }

  return obj
}
```

- It will check that the cookie is a valid cookie and then it will parse the JSON cookie object in JSONCookies() which will return a map object that has key value pairs of the JSON cookie
- The function uses the library **cookie-signature** which will sign and unsign cookies(https://github.com/tj/node-cookie-signature).
- It will sign the cookie and then set the cookie of the page to the signed value and make sure it throws a 200 OK request.

23 lines (19 sloc) | 662 Bytes

```
/**
 * Module dependencies.
 */

var cookie = require('..');

describe('.sign(val, secret)', function(){
  it('should sign the cookie', function(){
    var val = cookie.sign('hello', 'tobiiscool');
    val.should.equal('hello.DGDUkGlIkCzPz+C0B064FNgHdEjox7ch8tOBGslZ5QI');

    var val = cookie.sign('hello', 'luna');
    val.should.not.equal('hello.DGDUkGlIkCzPz+C0B064FNgHdEjox7ch8tOBGslZ5QI');
  })
})

describe('.unsign(val, secret)', function(){
  it('should unsign the cookie', function(){
    var val = cookie.sign('hello', 'tobiiscool');
    cookie.unsign(val, 'tobiiscool').should.equal('hello');
    cookie.unsign(val, 'luna').should.be.false;
  })
})
```

    - These lines of code are test cases to test if the signed cookie is correct with the given string and also checks for unsigned strings to make sure it is equivalent with the given string that was unsigned.
- **What license(s) or terms of service apply to this technology?**
    - MIT