DOMOMAKER - A

This assignment is due by the due date on the dropbox.

This exercise is the first part of an assignment series to build a small MVC-based login system with some basic per-user functionality.

You will be building a site to make new '*characters'* and track some basic stats. The user will make an account, login and then name their new character and give them an age.

In this first section, we will setup our own basic MVC framework that allows users to make accounts and login.

1. Download the starter code from https://github.com/IGM-RichMedia-at-RIT/DomoMaker-A-Start

2. Once you have downloaded the code, make sure you run *npm install* successfully Now let us familiarize ourselves with the code base.

3. Open the package.json file and check out the dependencies we are using. Some of these are new packages that will be introduced by this series of assignments. Note that your version numbers may differ slightly.

```
"devDependencies": {
  "eslint": "^8.11.0",
  "eslint-config-airbnb": "^19.0.4",
  "eslint-plugin-import": "^2.25.4",
  "nodemon": "^2.0.15",
  "webpack": "^5.70.0",
  "webpack-cli": "^4.9.2"
},
"dependencies": {
  "bcrypt": "^5.0.1",
  "body-parser": "^1.19.2",
  "compression": "^1.7.4",
  "connect-redis": "^6.1.3",
  "cookie-parser": "^1.4.6",
  "csurf": "^1.11.0",
  "express": "^4.17.3",
  "express-handlebars": "^6.0.3",
  "express-session": "^1.17.2",
  "helmet": "^5.0.2",
  "mongoose": "^6.2.7",
  "redis": "^4.0.4",
  "serve-favicon": "^2.5.0",
  "underscore": "^1.13.2"
}
```

4. Now go into src/views and open the handlebar files.

    - login.handlebars – Our default homepage, contains our login form that posts to /login. Take note that the form input names are “username” and “pass”. These will become important later!

    - signup.handlebars – Our signup page, directed from the homepage login.handlebars. This contains a form similar to our login page but posts to /signup. Take note that the form input names are “username”, “pass” and “pass2”. These will become important later! We will use “pass” and “pass2” to ensure their password is correct when they make their account.

    - app.handlebars – This will become the main app the user logins to. Here they will be able to make little characters.

5. Take note that our handlebars files pull in assets from “/assets” on the server. The “/assets” URL will be linked to the “hosted” folder in the app using static file hosting.

6. Open the client folder and look at the asset files. The client folder will be all of our client-side code. We will use webpack to bundle it into a single file for the client.

    Open the client.js file and look at the code to get an idea how it works. When the login and signup forms are submitted, the JS will make the POST itself instead of the form itself. Then the JS will wait for some JSON response back from the server.

    The client.js file holds some basic event handlers that send requests for multiple forms across the whole project. All views will be sharing the same bundle.js.

    ***NOTE: Remember that our client-side JS CANNOT talk directly to server code. It must communicate through fetch requests or a similar tool. ***

7. Open the package.json file and look at the ‘webpack’ script. We will use this script using ‘npm run webpack to take our client-side files and create bundle.js in the hosted folder. That bundle is the JS file that views will actually download and run. Webpack allows us to easily break code into separate files, import libraries, etc.

8. Open the Account.js file in Models. This file is already complete (mostly due to its more complicated nature). Be sure to read the comments to understand what each function does.

    Similar to our previous examples, this file has a Schema. Remember that a Schema is a definition of what data looks like. This is what allows us to know our data is

formatted correctly going into the database and what the variable names/types will be coming out of the database.

Notice that the schema also has a few methods attached as AccountSchema.statics. Anything attached to a schema in .statics is going to be a method you can call through the Model.

For example, the authenticate function in the Account.js Model is a static function defined by the Schema. This means that you can call authenticate by calling Account.authenticate from any file that pulls the account model in. Since it has been attached to the Model through the Schema, the function has access to the database but not a particular object.

9. Take note, that as a whole, the Account.js Model holds user accounts.

   Each account has

   - Username – The user's username. The match field ensures users Regex to ensure that the username is alphanumeric and 1-16 characters.

   - Password – The encrypted password. We are making use of the bcrypt npm package to hash our passwords in the database. We never want to store unencrypted/plaintext passwords, as it would be very simple for someone to steal them. Hashing a password can take fractions of a second, but days/weeks/months to unhash.

   - CreatedDate – Just basic date data for when an account was made. Very useful to have in a production system.

10. Now open the app.js file. We need to setup our basic MVC functionality. Setup the server, express MVC and our database connection.

    Use "`mongodb://127.0.0.1/DomoMaker`" and the `process.env.MONGODB_URI` variable for the database connection. Node will first attempt to connect to your MongoDB Cloud instance, and will fallback to your local Mongo instance.

Part 1

```
const path = require('path');
const express = require('express');
const compression = require('compression');
const favicon = require('serve-favicon');
const cookieParser = require('cookie-parser');
const bodyParser = require('body-parser');
const mongoose = require('mongoose');
const expressHandlebars = require('express-handlebars');
const helmet = require('helmet');

const router = require('./router.js');

const port = process.env.PORT || process.env.NODE_PORT || 3000;

const dbURI = process.env.MONGODB_URI || 'mongodb://127.0.0.1/DomoMaker';
mongoose.connect(dbURI, (err) => {
  if (err) {
    console.log('Could not connect to database');
    throw err;
  }
});
```

--Part 2 Continued on Next Page –

Part 2 of app.js

```
const app = express();

app.use(helmet());
app.use('/assets', express.static(path.resolve(`${__dirname}/../hosted/`)));
app.use(favicon(`${__dirname}/../hosted/img/favicon.png`));
app.use(compression());
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());
app.engine('handlebars', expressHandlebars.engine({ defaultLayout: '' }));
app.set('view engine', 'handlebars');
app.set('views', `${__dirname}/../views`);
app.use(cookieParser());

router(app);

app.listen(port, (err) => {
  if (err) { throw err; }
  console.log(`Listening on port ${port}`);
});
```

Some of the libraries used above are not ones we have seen before. Here are some brief descriptions.

- Helmet: a security library for express. Sets a bunch of default options for us to obscure information from malicious attacks. https://www.npmjs.com/package/helmet
- CookieParser: a tool for parsing http cookies. Cookies are simply key-value pairs stored on the client's machine that we can use to track data. We will begin using cookies later in DomoMaker.

11. Next open router.js and setup the following.

```
const controllers = require('./controllers');

const router = (app) => {

};

module.exports = router;
```

12. Startup your server on your computer and make sure it starts on port 3000 successfully.

    If there are any database connection errors or anything, make sure that Mongo is running (reference the *Setting up MongoDB Locally* guide if you forgot how to get Mongo running)

    Otherwise, correct any errors before moving on.

13. Now we need to connect our routes. Inside of the router.js file, fill out the following routes. We have not yet made these controllers, but we will shortly.

```
const router = (app) => {
    app.get('/login', controllers.Account.loginPage);
    app.post('/login', controllers.Account.login);

    app.get('/signup', controllers.Account.signupPage);
    app.post('/signup', controllers.Account.signup);

    app.get('/logout', controllers.Account.logout);
    app.get('/maker', controllers.Domo.makerPage);
    app.get('/', controllers.Account.loginPage);
};
```

14. Now we need to setup the controller. Go into the controllers folder and open index.js.

    Remember that this file gets automatically imported when the folder is imported. Since we know this, we will make this file pull in other files we want in public scope. We will pull in Account and Domo.

```
module.exports.Account = require('./Account.js');
module.exports.Domo = require('./Domo.js');
```

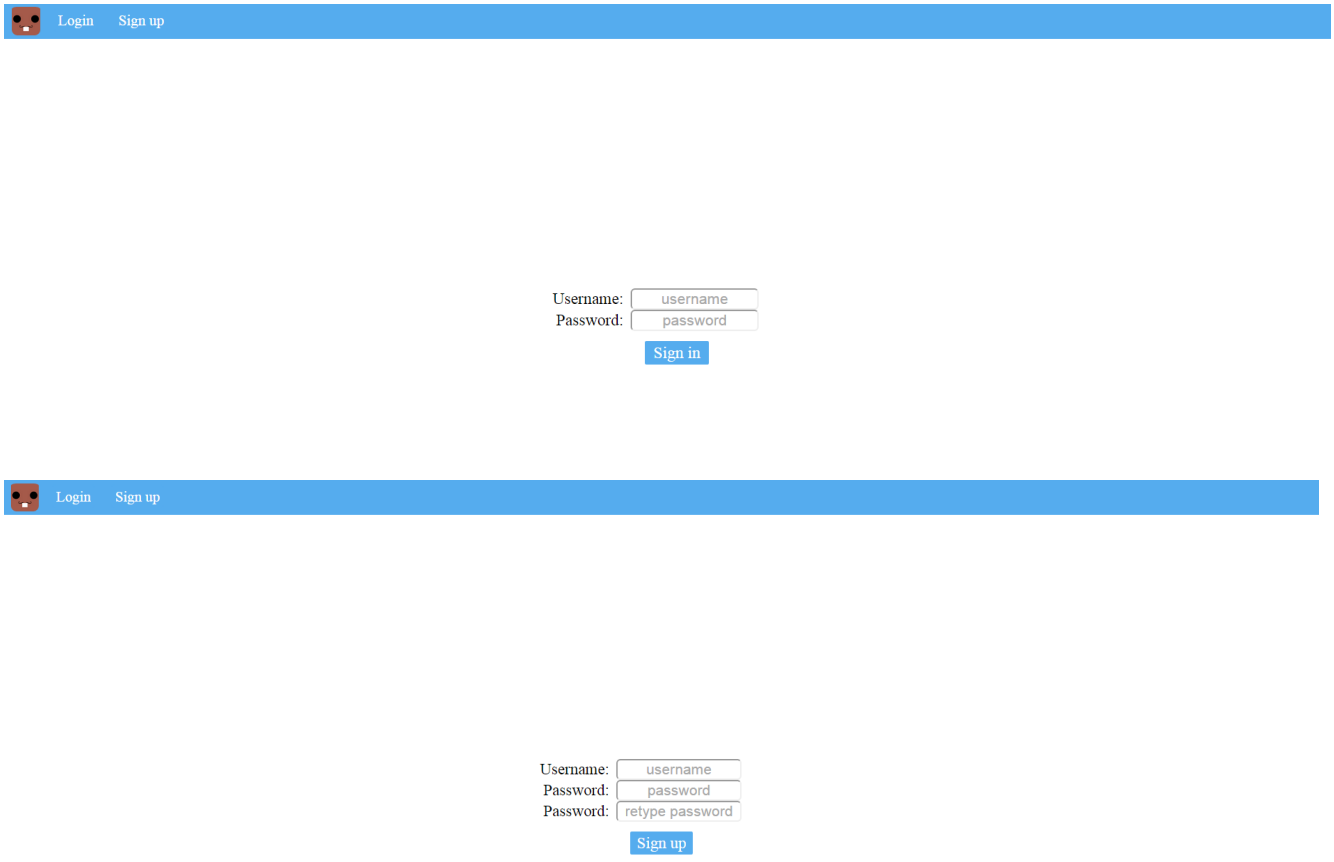
15. Open the Account.js Controller file and add the following.

```
const models = require('../models');
const Account = models.Account;

const loginPage = (req, res) => {
    return res.render('login');
};

const signupPage = (req, res) => {
    return res.render('signup');
};

const logout = (req, res) => {
    return res.redirect('/');
};

const login = (req, res) => {

};

const signup = (req, res) => {

};

module.exports = {
    loginPage,
    signupPage,
    login,
    logout,
    signup,
};
```

16. Next open the Domo.js controller file and add the following. We will do more with this in the next exercise. For now, we will just have it render out the main app page.

```
const makerPage = (req, res) => {
    res.render('app');
}

module.exports = {
    makerPage,
};
```

17. Start the server again and ensure that you can go to the main page and signup page successfully.



18. Now we need to make sure we can create accounts and login successfully. Reopen the client.js file in the assets folder. This is what the JS browser side will be doing.

A fair bit of functionality already exists here, mostly for the handling of POST requests made by the various forms in the different views. Note that our sendPost function is being used to send a fetch() request with a JSON body to the server (which is parsed by body parser). When a response comes back, we currently look for a redirect message or an error message.

Additionally, we have a hidden domoMessage div that appears when a message needs to be shown to the user.

19. To build your client side code into the bundle.js, run the following command in Powershell or Terminal: “npm run webpack”. Note that /hosted/bundle.js is now a built version of our client code that is automatically hosted as /assets/bundle.js because of our static file hosting line in app.js.

20. The signup form’s ACTION attribute is set to /signup as a POST. This is where our fetch() request will go. If you look at our router.js, we send all POST requests to /signup to controllers.Account.signup.

    We need to add the functionality though. Currently out Account.signup function is empty.

21. In the Account.js Controller file, replace the signup function with the following. This function has a little bit of complexity to it due to the encryption process. If you want to see how it works, look at the Account Model’s generateHash static function.

    First, we validate the data. We cast to strings to guarantee valid types and then check if it is all there and the passwords match.

```
const signup = async (req, res) => {
  const username = `${req.body.username}`;
  const pass = `${req.body.pass}`;
  const pass2 = `${req.body.pass2}`;

  if (!username || !pass || !pass2) {
    return res.status(400).json({ error: 'All fields are required!' });
  }

  if (pass !== pass2) {
    return res.status(400).json({ error: 'Passwords do not match!' });
  }

};
```

22. Next, add the following after the password checks.

    The first step is for us to attempt to hash the password. This is done by bcrypt inside of our Account.generateHash static function. This will return to us an encrypted version of the password the user gave us.

    Then, we will attempt to create a new user in the database using the username the user gave us, as well as the hashed password. Remember, we **never** want to store plaintext/unencrypted passwords in our database.

    Provided the save goes well, we will send the user a redirect message that sends them to the /maker page. Note that because of how sendPost() in client.js handles requests we can do this.

    If something goes wrong in any part of the process, we will end up in the catch statement. If we get error code 11000 (mongo's "duplicate entry" error), we will tell the user that the username is already taken. Otherwise we will send back a generic error message to the client.

```
  if (pass !== pass2) {
    return res.status(400).json({ error: 'Passwords do not match!' });
  }

  try {
    const hash = await Account.generateHash(pass);
    const newAccount = new Account({username, password: hash});
    await newAccount.save();
    return res.json({ redirect: '/maker' });
  } catch (err) {
    console.log(err);
    if (err.code === 11000) {
      return res.status(400).json({ error: 'Username already in use.' });
    }
    return res.status(400).json({ error: 'An error occurred' });
  }
};
```

23. Let's fill out the login function in the Account Controller. Replace the login function with the following.

    Similar to signup, we make sure we have all the data then we call the Model to handle the database entry for us. We are calling the Model's static authenticate function. Again, to see how this works, just look at the function in the Account Model.

Controllers/account.js Login function

```
const login = (req, res) => {
    const username = `${req.body.username}`;
    const pass = `${req.body.pass}`;

    if(!username || !pass) {
        return res.status(400).json({ error: 'All fields are required!' });
    }

    return Account.authenticate(username, pass, (err, account) => {
        if(err || !account) {
            return res.status(401).json({ error: 'Wrong username or password!' });
        }

        return res.json({ redirect: '/maker' });
    });
};
```

24. Restart the server and try to signup. See what happens when you don't put in a field into the signup form or if your passwords don't match.

    Signup correctly with real data and it should redirect to the main page.

    Fix any problems that occur before moving on.

25. Try to logout once you are logged in. Then login in at the main login page. It should log you in and take you to the main page. Then log back out again.

    Fix any problems that occur before moving on.

26. Finally, shut down your server and restart it. Now try to log back in with the account you made.

You should be able to log back in because all of the data is stored in the database. You can shutdown and restart the server without losing any data.

27. Create a new app on Heroku to push your code to.

28. Now we need to add our MongoDB Cloud Instance to our application on Heroku. Follow the instructions in Part 3 of the “MongoDB Cloud Setup” guide in MyCourses>Content>Guides. **Be sure to replace myFirstDatabase in your connection string to something like DomoMaker**.

29. Upload your code to your new Heroku app and test your app on Heroku to make sure it works correctly.

ESLint & Code Quality

You are required to use ESLint with the AirBnB configuration for code quality checks. The eslintrc config file was given to you in the starter files. Your **server** code must pass ESLint to get credit for this part. Warnings are okay, but errors must be fixed. I won't require ESLint on the client side code in this assignment. ***If you cannot fix an error or need help, please ask us!***

You are also required to use the "pretest" script in order to run ESLint. I should be able to run "*npm test*" and have ESLint scan your code for me.

Continuous Integration

You are required to use CircleCI for continuous integration. Your build must succeed and be handed in with the assignment. The circle.yml config file was given to you in the starter files. CircleCI will build based on your *npm test* and ESLint configuration.

For instructions on how to do this, look at the ***CircleCI Setup*** instructions on mycourses.

Build & Bundling Scripts

Your app should have custom scripts to automatically restart node upon server changes (I suggest using nodemon) and create a client-side JS bundle (Webpack is installed and configured to do this with the "npm run webpack" command).

You should write your JS in separate files and create a bundled JS file from them. The only JS file that should go to the client is the bundled file.

-- Rubric on Next Page –

DOMOMAKER - A

Rubric

DESCRIPTION	SCORE	VALUE %
App works correctly – all pages, forms and functions work correctly. App.js starts and runs correctly.		**15**
App Connects to Database Successfully on Startup – When the server is started, the connection to the database is made. If the database fails to connect, it should show an error that the database did not connect successfully.		**10**
Router Works Correctly – All routes in the router work correctly. All methods (get, post, etc) work correctly and the controllers receive the correct arguments.		**10**
Models Work Correctly – New accounts can be made, saved and loaded correctly. Server should be able to restart without losing the data in the database.		**10**
Signup Controller Works as Instructed – The signup controller should work as instructed and allow users to signup successfully. User is prevented from making an account if data submitted is invalid.		**15**
Login Controller Works as Instructed – The login controller should work as instructed and allow users to login successfully. User is prevented from logging in if data submitted is invalid.		**15**
App Page Loads Successfully on Login – When a user logs in or signs up successfully, they should be redirected to the app page automatically.		**15**
Users can Login and Logout – Users should be able to logout successfully and log back in. This cycle should continue to work repeatedly.		**10**
Git Penality – If your code is not in a Git repo or the link is not handed in, there will be a penalty.		**-30% (this time)**
Heroku Penalty – If your app is not functional on Heroku, there will be a penalty. This means you need to correctly hookup the addons on Heroku.		**-30% (this time)**
CircleCI Penalty – If your build is not successful on CircleCI, there will be a penalty.		**-30% (this time)**
Build and Bundling Penalty - If your package.json does not include properly working automatic build scripts for Node (nodemon or other) and client-side bundling scripts (babel, webpack, browserify, rollup or other), there will be a penalty.		**-20% (this time)**
ESLint Penalties – I should be able to run 'npm test' and have ESLint scan your code without errors. Warnings are acceptable. **You may not alter the rules in the .eslintrc file provided.** **Check your code with ESLint often during development!** **If you cannot fix an error, please ask us!**	1 Error 2 Errors 3 Errors 4 Errors 5+ Errors	**-10%** **-15%** **-20%** **-25%** **-35%**
Additional Penalties – These are point deductions for run time errors, poorly written code or improper code. There are no set values for penalties. The more penalties you make, the more points you will lose.		
TOTAL		**100%**

DOMOMAKER - A

Submission:

By the due date please submit a zip of your project to the dropbox. **Do not** include the node_modules folder in the zip or in your git repo.

In the submission comments include the following:

- *a link to your Heroku App (not the dashboard but the working web link from 'open app'),*
- *a link to your Github repo (if private, please add the TA and myself).*
- a link to your circleCI build *in the submission comments.*