DOMOMAKER - C

This assignment is by the due date on the dropbox.

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

In this section, you will be setting up Redis to persist your sessions so that they will remain even if the server shuts down. Additionally, if you were to ever scale your server across multiple machines the sessions would continue to work as normal.

You will be starting where you left off in Part-B.

1. One of the first things we are going to do is hook up Redis (**Re**mote **Di**ctionary **S**erver). Redis is a key:value database that persists in memory. We can store variables in Redis, but when we restart Redis everything is cleared out of memory.

   How is that different from just putting the variables in memory in the server code?

   Well the answer is that it's not in the server code. That means our server is still mostly or entirely stateless. It's not holding very many variables since they are stored somewhere else. This means we can restart our server over and over without losing variables.

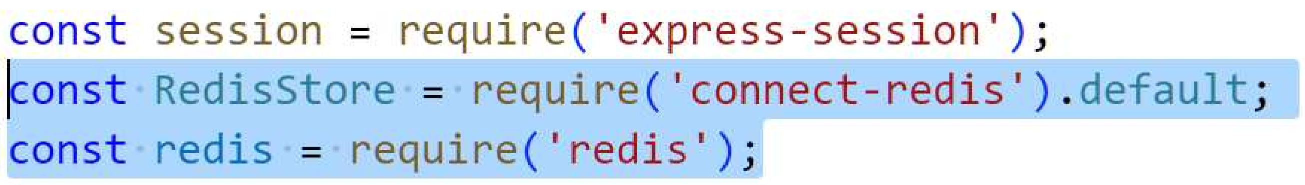
   Additionally, it means we can run multiple node apps that have the same shared variables. The variables are actually in Redis, so any of our apps talking to our Redis database can get/set variables in it. This means several different machines running different node apps (or even the same node apps) can share variables by accessing the same Redis database.

   Redis also clusters so that you can run Redis on many servers and have it automatically sync data across many machines. If one server shuts down, then you can have dozens of others mirroring it (and you don't lose any variables).

   Systems like Redis allow us to seamlessly shut down a server and bring it back up without anyone noticing (users still stay logged in, pages still work, permissions still work, etc). As an added benefit to you, Redis will let you work on your node app (restarting it as many times as you want) without getting logged out or anything.

2. We will use the connect-redis package. The connect-redis package is designed for storing session data & cookies in Redis. If you need Redis for other reasons, you could just get the redis package alone. We will also need the base Redis library.

   Inside of your app.js, pull in redis and connect-redis.



```
const session = require('express-session');
const RedisStore = require('connect-redis').default;
const redis = require('redis');
```

3. In your app.js, you will need to connect to your Redis instance. We will use the redis server on Redis Cloud that you created. Check out the “Setting Up Redis Cloud” document for more info.

   Since we can’t run Redis locally on windows, we need to use the cloud server even for local development. This presents a new problem, because we do not want to put our connection string directly into our code. Doing so would mean that it would become publicly visible on GitHub, which is a massive security problem.

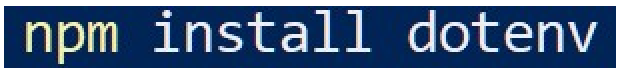
   The common solution to this problem is to use an environment variables (env) file. A .env file is a file containing all of our sensitive connection strings. By design, our .gitignore is setup to not track the .env file (you can see this at the very bottom of the .gitignore). That means it will be useful to us locally, but won’t get sent to GitHub. This also means that you’ll need to recreate the .env file on every machine you work on, or store it somewhere secure. Google Drive is secure enough for our purposes in this class, but companies would likely have an internal file store.

   First, begin by creating a file called .env in the root of the project (where the package.json is). Then in that file, write the following:

   REDISCLOUD_URL=YOUR_REDIS_CLOUD_URL

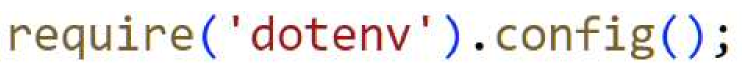
   In place of YOUR_REDIS_CLOUD_URL, put your connection string from Redis Cloud with the username and password filled out.

4. To work with this .env file, we need a new npm package: dotenv



```
npm install dotenv
```

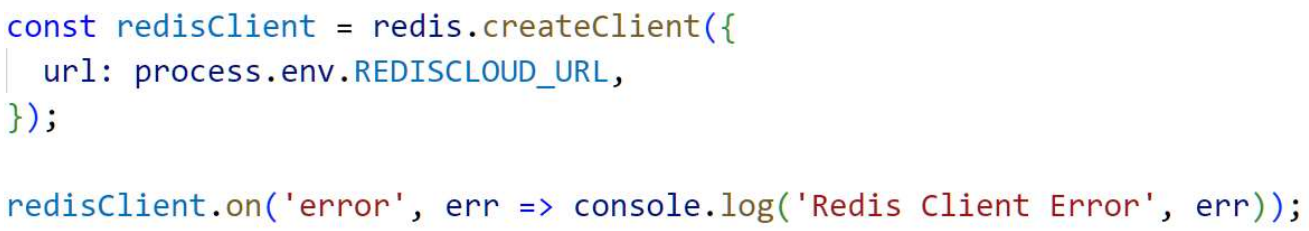
5. Now go to src/app.js, and add the following as the very first line of code in the file before any other require statements.



```
require('dotenv').config();
```

   The dotenv library is a simple tool for working with .env files. Simply put, it will parse any key-value pairs in the .env file and import them into the process.env object. This is the same thing that config vars on Heroku do. We can also add more config variables here if we want. The format is that each line is a different variable, and they are in the KEY=VALUE format.

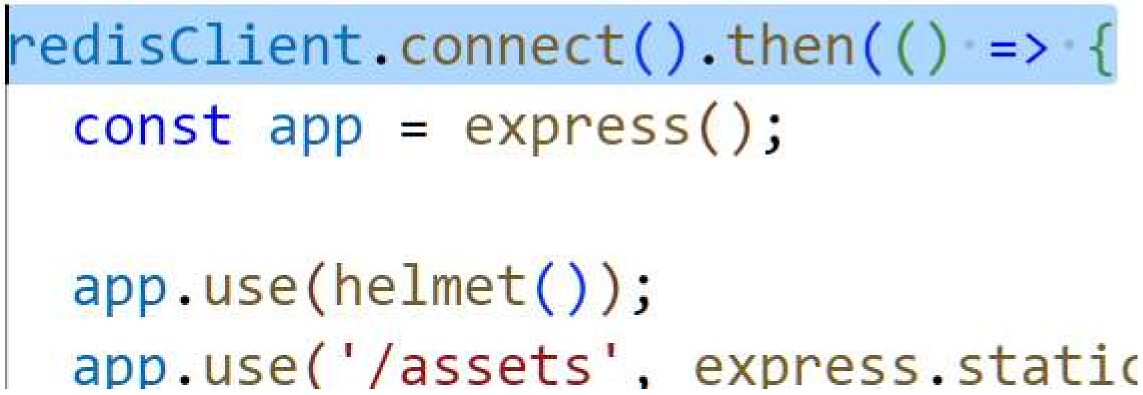
6. Now that your REDISCLOUD_URL has been loaded into process.env.REDISCLOUD_URL, we can use it to create our redis client. Because Heroku will populate this same environment variable, our code will work in both locations. We will use it to instantiate the redis client, and add an error handler to it. Do this just below where you connected mongoose to your mongo database.



```
const redisClient = redis.createClient({
  url: process.env.REDISCLOUD_URL,
});

redisClient.on('error', err => console.log('Redis Client Error', err));
```

7. Now that we have the client setup, we need to tell it to connect to our Redis instance. To do this we will call redisClient.connect(). This process is contacting the database, and the rest of our code needs to wait until we have connected. Since node does not support top-level await in CommonJS, we will need to use a promise to delay the rest of our code.

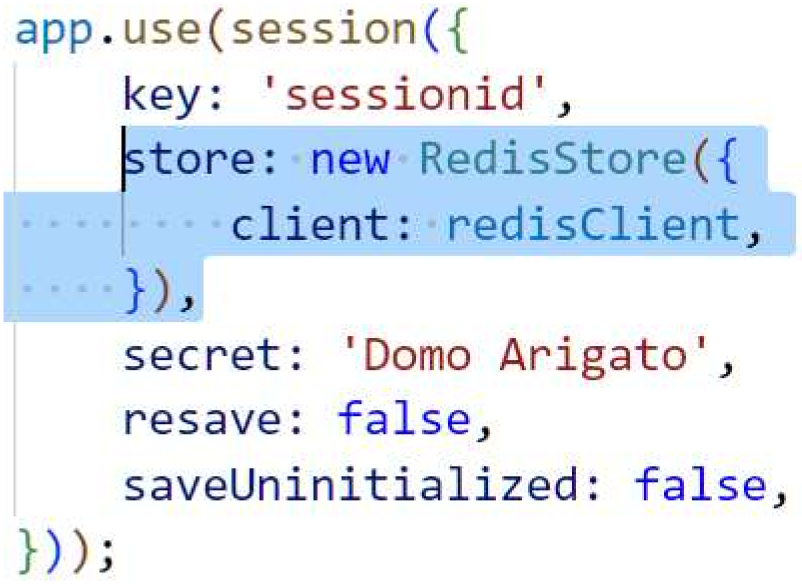
   Add the following code immediately after the code above, and then move all the code from "const app = express();" and down into the .then() handler function. This should be everything, including app.listen() at the very bottom.



```
redisClient.connect().then(() => {
  const app = express();

  app.use(helmet());
  app.use('/assets', express.static
```

8. We now have a Redis client connected to our database. With that, we can set up our session system in express and tell it to use this client. Below, inside of the .then() handler, update the session setup to look like the code below. This will tell the

express-session library to utilize our redis database to store the sessionid for each user.



```
app.use(session({
    key: 'sessionid',
    store: new RedisStore({
        client: redisClient,
    }),
    secret: 'Domo Arigato',
    resave: false,
    saveUninitialized: false,
}));
```

9. Go ahead and start the server. **You will need Redis Cloud setup and Mongo running locally for this**. Refer to the “Setting up MongoDB Locally” and “Setting up Redis Cloud” documents if needed.

   ***Note:** If Redis is not connected, your app will crash on login.

   Try to login or signup. It should be successful and you should go to the app page.

10. Once you are at the main app page. Shutdown and restart the server. Reload the app page in the same browser.

    You should stay signed in!

    In the previous assignment, when you restarted the server and went to a page it broke because the sessions were no longer there. This is because the sessions were stored in server variables. This means when the server restarts and memory is cleared, the data is no longer there.

    With Redis, our sessions are stored outside of the server. This means during dev you can restart the server as many times as you want without losing variable data.

11. Keep your server running, but in code change the node port from 3000 to 3001. Then open a second Powershell (or Terminal) window and start a second node server.

In the same browser you logged in on, open a tab and go to 127.0.0.1:3001/maker

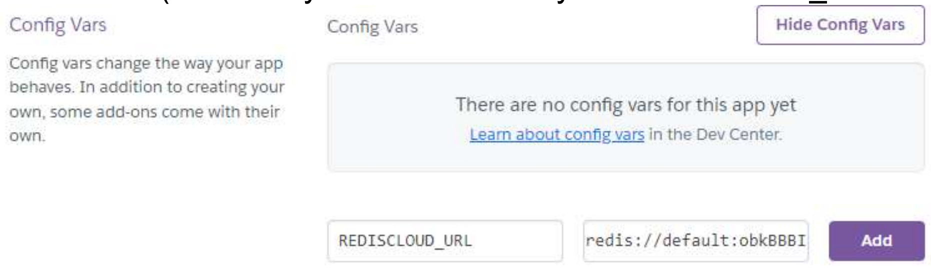
It should take you directly to the app. Since the sessions are stored in Redis and both servers (port 3000 and port 3001) are using Redis, they both have access to the same variables.

You can store all sorts of shared variables in Redis that all of your servers can access. This helps with dev (being able to restart node constantly without losing data) and with production (having server redundancy).

12. **Change your server port back to 3000** and shutdown your Node servers.

13. On Heroku, add your `REDISCLOUD_URL` to your config vars. You can get here by opening your Heroku app's dashboard, clicking "Settings" and scrolling down to reveal config vars.

    On Heroku (note that you should already have a `MONGODB_URI` config var)



Config Vars

Config vars change the way your app behaves. In addition to creating your own, some add-ons come with their own.

Config Vars

Hide Config Vars

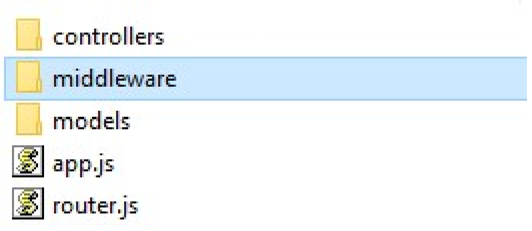
There are no config vars for this app yet
Learn about config vars in the Dev Center.

REDISCLOUD_URL	redis://default:obkBBBI	Add

14. Next we will setup Middleware. Middleware in Express is a powerful system for injecting code calls in between other calls. Often you see it used in routing and error handling. Most all of our express plugins could be considered middleware.

    For example, our Express-session module is middleware. On every request, it is injecting checks for cookies, getting variables from Redis and creating a variable in the request called req.session.

    Create a "middleware" folder inside of your "server" folder.



```
controllers
middleware
models
app.js
router.js
```

15. Inside of your new middleware folder, create an index.js file.

16. Now we will add some useful redirection for requests. For example, if a user is not logged in (i.e., does not have a session), then why even let them go to the app page and show an error?

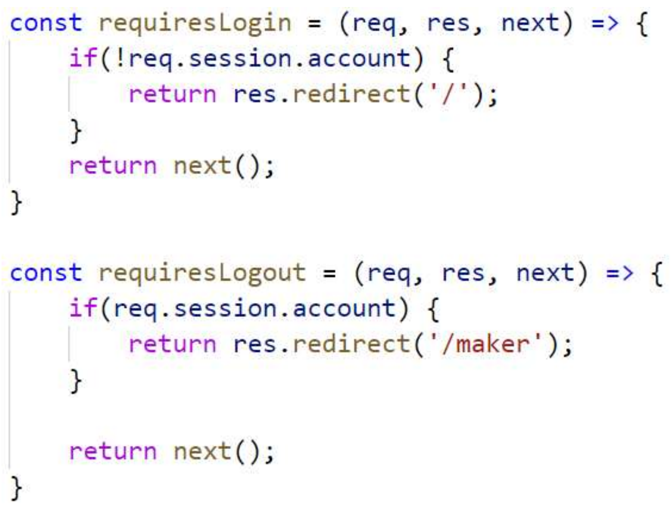
    Middleware functions receive a request, response, and the next middleware function to call. This allows your application to make various decisions and potentially chain into the next middleware call. The request will not continue through the system UNLESS you call the next function at the end.

    For middleware functions to continue to the controllers, you MUST call the next function. However, you may not want your request to get the controllers if the request is not valid. You may want to redirect them to a different page or stop the request entirely.

    Add a requiresLogin function that checks if we attached an account to their session and redirects to the homepage if not.

    Similarly, add a requiresLogout function that checks if the user is already logged in (we attached an account to their session) and redirects them to the app if so.

    Code samples on the next page.



```
const requiresLogin = (req, res, next) => {
    if(!req.session.account) {
        return res.redirect('/');
    }
    return next();
}

const requiresLogout = (req, res, next) => {
    if(req.session.account) {
        return res.redirect('/maker');
    }

    return next();
}
```

17. Next, we'll add some other useful functions.

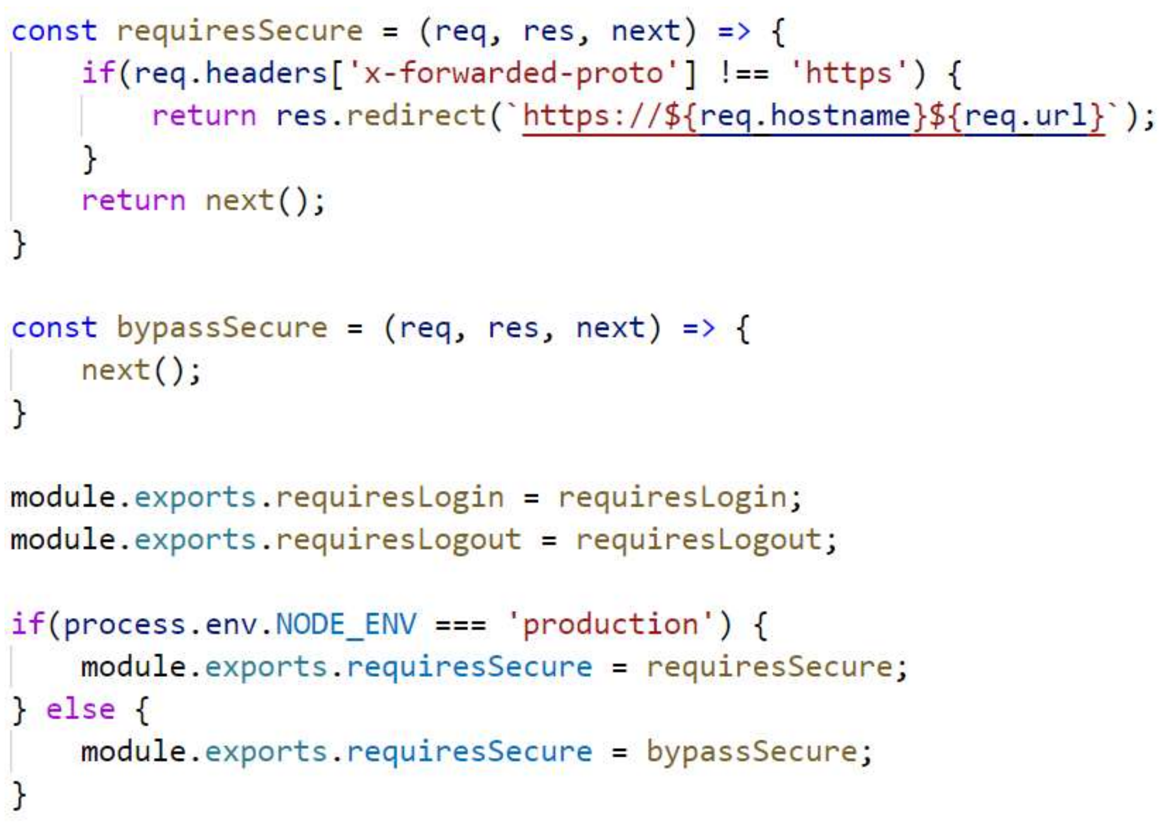
If the user is trying to do something secure, such as logging in, then we will check to see if they are on HTTPS and redirect them if not.

***Note:** Normally you check for security by checking if *req.secure* is true, but the Heroku environment is all encrypted internally so it would always be true. Instead, we will check to see if the forwarded request (through Heroku) was secure by checking the request's "x-forwarded-proto" header.

For local development/testing, we can't easily run HTTPS, so instead we will just bypass the check.

How do we know if we are on Heroku or not? Environment variables! This is where the power of custom environment variables comes in. We will be creating an environment variable called NODE_ENV that we will set to "production" on Heroku. Then when the server starts, we can just check which to export based on the environment.

If later on your code isn't working on Heroku but works locally, chances are it has to do with you either not setting up your NODE_ENV config var correctly, or something being messed up in the requiresSecure function.



```
const requiresSecure = (req, res, next) => {
    if(req.headers['x-forwarded-proto'] !== 'https') {
        return res.redirect(`https://${req.hostname}${req.url}`);
    }
    return next();
}

const bypassSecure = (req, res, next) => {
    next();
}

module.exports.requiresLogin = requiresLogin;
module.exports.requiresLogout = requiresLogout;

if(process.env.NODE_ENV === 'production') {
    module.exports.requiresSecure = requiresSecure;
} else {
    module.exports.requiresSecure = bypassSecure;
}
```

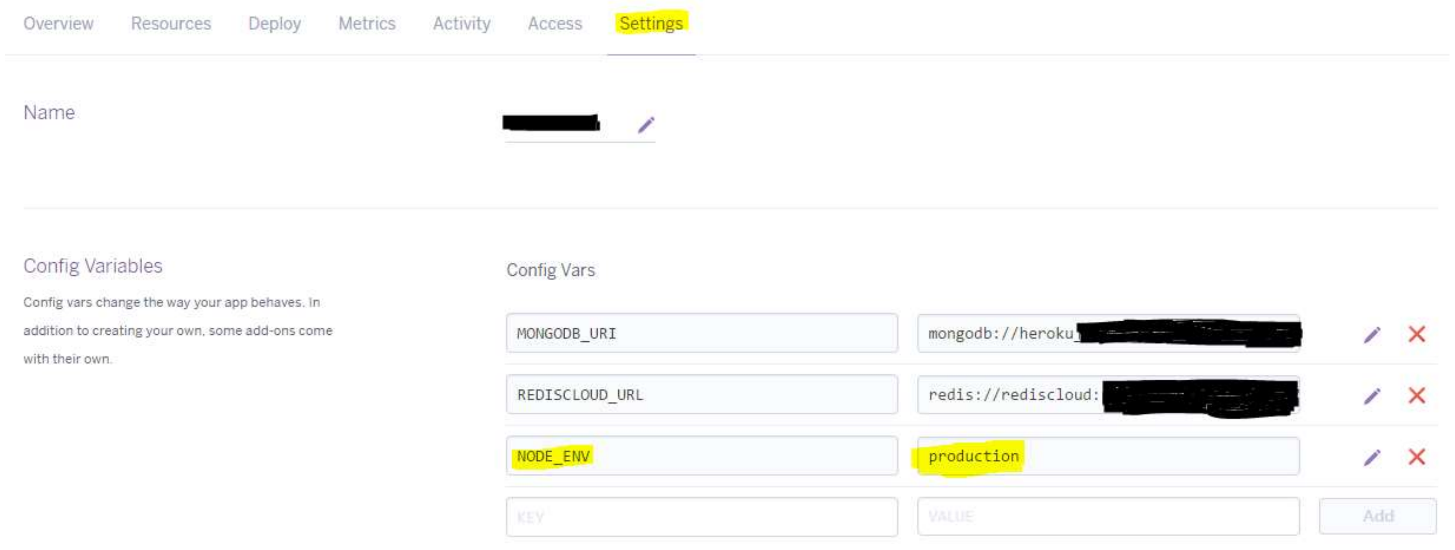
18. For this code to work, we need to add an environment variable on Heroku.

In your Heroku app, go to "settings" and hit "reveal config vars"

Any of these environment variables can be accessed by name in code with `process.env.VAR_NAME`

`MONGODB_URI` was manually added by you as a part of the instructions from the "MongoDB Cloud Setup" document. We configured `REDISCLOUD_URL` earlier in this document.

Add a new one called `NODE_ENV` and set it to "production".



[redacted]

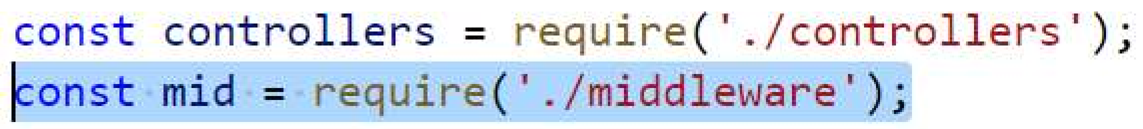
Config Variables

Config vars change the way your app behaves. In addition to creating your own, some add-ons come with their own.

Config Vars

MONGODB_URI	mongodb://heroku[redacted]
REDISCLOUD_URL	redis://rediscloud:[redacted]
NODE_ENV	production

19. Now we need to connect the middleware to your routes. In your router.js, pull in the middleware module.



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

20. We need to connect each request with the series of relevant middleware.

    The way router level middleware in Express works is you connect as many middleware calls as you want in the order you want the middleware to run. The first parameter is always the URL. The last parameter is always the controller. Everything in between is any of the middleware operations you want to call.

- For GETs to / or /login we want to make sure it's secure, so we call requiresSecure. Then we also want to make sure they are logged out (so they don't see a login page if they are logged in), so we call requiresLogout.

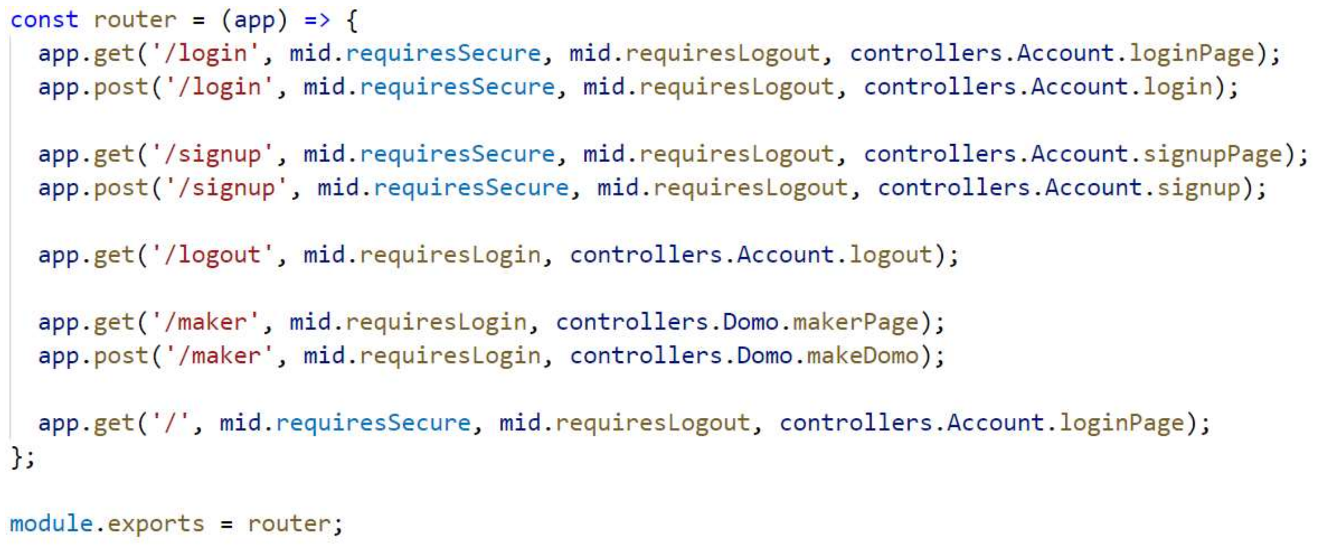
- For POSTs to /login or /signup we want to make sure it's secure, so we call requiresSecure. We also want to make sure they are logged out (so they can't try to login when logged in), so we call requiresLogout.

- For GETs to /logout we want to make sure they are logged in, so we call requiresLogin. They can't logout if they aren't logged in.

- For GETs to /maker we want to make sure they are logged in. They can't get to their app page if they aren't logged in.

- For POSTs to /maker we want to make sure they are logged in. They can't try to add characters to their account if they aren't logged in.

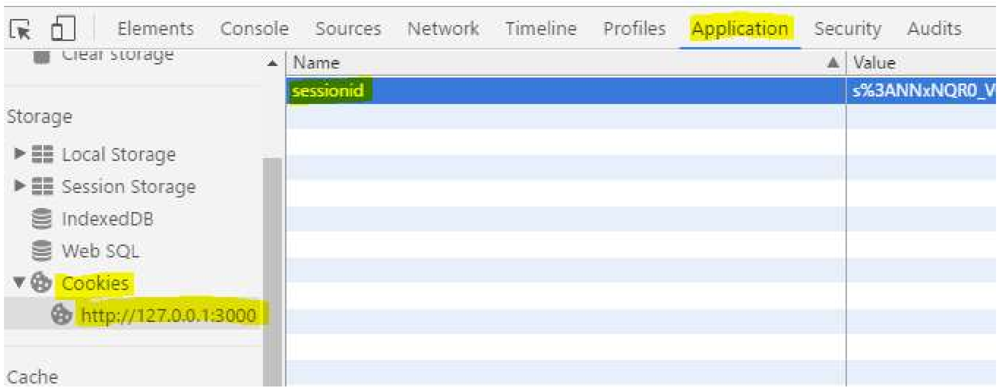
These middleware calls are all functions we made in the middleware file.



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

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

  app.get('/logout', mid.requiresLogin, controllers.Account.logout);

  app.get('/maker', mid.requiresLogin, controllers.Domo.makerPage);
  app.post('/maker', mid.requiresLogin, controllers.Domo.makeDomo);

  app.get('/', mid.requiresSecure, mid.requiresLogout, controllers.Account.loginPage);
};

module.exports = router;
```

21. Let's clear the client cookie to make you are logged out for testing. In Chrome (or equivalent), delete the sessionid cookie for this user. You can right click the sessionid cookie and hit delete.

    Also, make sure Mongo is still running.



Name	Value
sessionid	s%3ANNxNQR0_V

22. Now restart your server and

    - At the Login page, change the URL on the URL bar from / to /login to /maker. It should automatically redirect you to the login page.

    - At the signup page, change the URL on the URL bar from /signup to /maker. It should automatically redirect you to the login page.

    - At the login page, try to go to /logout. It should automatically redirect you to the login page.

    - Login to the app. At the main app page, try to go to /login. It should direct you to the /maker page automatically.

    - At the main app page, try to go to / . It should direct you to the /maker page automatically.

    - At the main app page, try to go to /signup. It should direct you to the /maker page automatically

    How does this work? Well, the middleware fires in order on each URL and if it passes the middleware criteria then it calls the next middleware function. If it does not pass the middleware criteria then the flow is broken and a different page is rendered instead of their request. This is what our middleware index file does.

23. Test the app and make sure everything works correctly before moving on.

24. Upload your app to Heroku. Make sure you have configured the following config vars: `MONGODB_URI`, `REDISCLOUD_URL`, and `NODE_ENV`

25. Open your app on Heroku. At the Domo login page, change the HTTPS protocol in the URL bar to HTTP and try to go there. Due to the `NODE_ENV` variable and our middleware, this should automatically force the page back to HTTPS.

26. Test your app on Heroku and ensure that everything 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.

DOMOMAKER - C

Rubric

DESCRIPTION	SCORE	VALUE %
App works correctly – all pages, forms and functions work correctly. App.js starts and runs correctly.		**25**
Redis Connected & Working – Redis connects successfully and stores sessions. App can be restarted without losing sessions or a user getting logged out.		**20**
Middleware Working & Connected in Routes – The middleware is connected and checked for in the routes. Routes still continue to direct to the correct controllers.		**20**
Redirection to login and app page appropriately – The app successfully redirects to the login page or app depending on whether or not the user is logged in.		**20**
HTTPS Redirection – The app successfully redirects to HTTPS from HTTP on Heroku.		**15**
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%**

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.*