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 securing the app (at least from basic attacks) and making the app much more robust. We will use middleware to handle a lot of issues for us.

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.

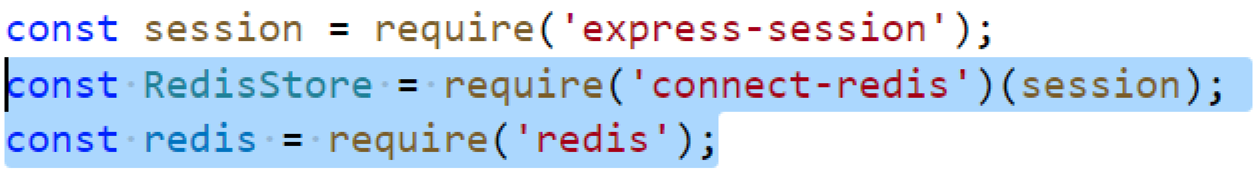
   Additionally, it means we can run multiple node apps that have the same shared variables. The variables are actually in Redis, so any of our apps talking to our Redis database can get/set variables in it. This means several different machines running different node apps (or even the same node apps) can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

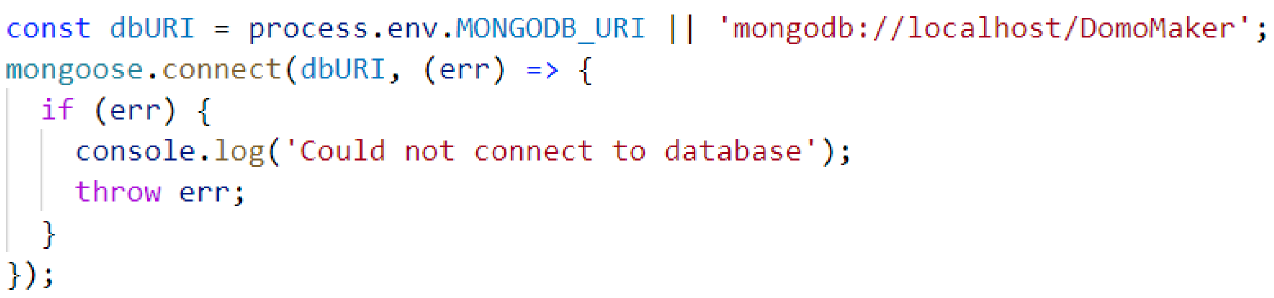
   Inside of your app.js, pull in redis and connect-redis. Pass session into connect-redis.

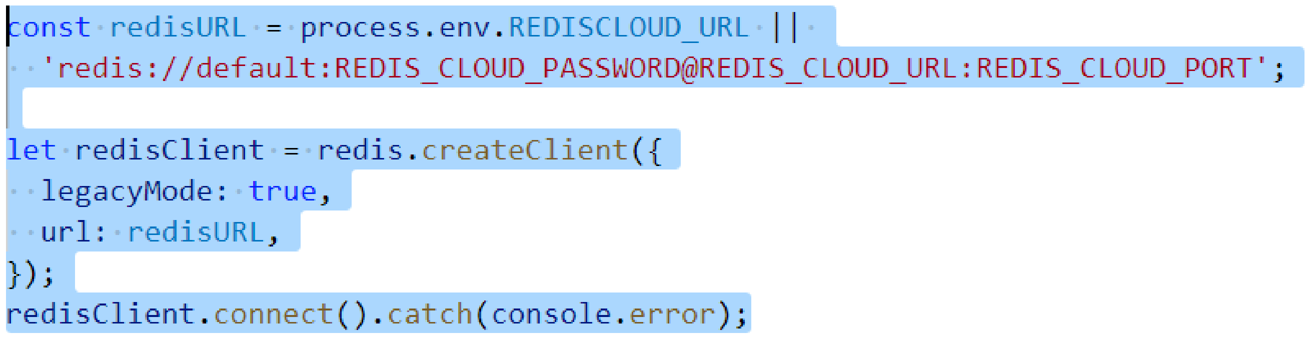

```
const session = require('express-session');
const RedisStore = require('connect-redis')(session);
const redis = require('redis');
```

3. In your app.js, you will need to connect to your Redis instance. We will use the redis server on redislabs you created. Check out the “Setting Up Redis for Local Use” document for more info. For the time being we will hardcode our redis connection into our code. This is not ideal as anyone with access to the source code can see our connection info. Additionally, if we want to change the server connection info, we need to redeploy the app. In a future class demo we will see how to use config files, which is the correct way to configure this information.

   With our login info, we then construct a Redis client from this information.

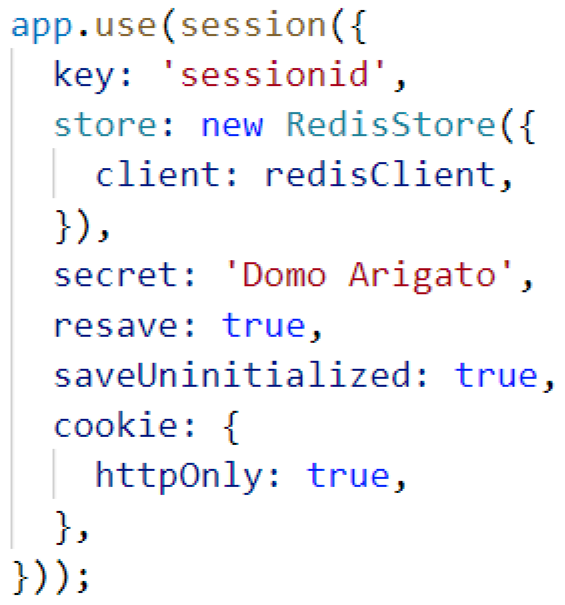
   In the redisURL, replace REDIS_CLOUD_PASSWORD with the one provided by RedisCloud. Replace the REDIS_CLOUD_URL with the url provided by RedisCloud. Replace the REDIS_CLOUD_PORT with the port provided by RedisCloud. Chances are the URL already has the port appended at the end, in which case you can ignore the :PORT at the end.


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


```
const redisURL = process.env.REDISCLOUD_URL ||
  'redis://default:REDIS_CLOUD_PASSWORD@REDIS_CLOUD_URL:REDIS_CLOUD_PORT';

let redisClient = redis.createClient({
  legacyMode: true,
  url: redisURL,
});
redisClient.connect().catch(console.error);
```

4. Once we have the Redis URL and password, and a Redis client created from it, then we need to tell our session module to use a RedisStore based on this client.

Instead of storing everything in server variables, this will instead store everything in a Redis database, assuming Redis is running.



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

5. Go ahead and start the server. **You will need Redis and Mongo running for this**. Refer to the “Setting up MongoDB Locally” and “Setting up Redis for Local Use” 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.

6. 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. It allows you to store 'states' in a 'stateless' server.

7. Keep your server running, but in code change the node port from 3000 to 3001. Then open a second Git bash (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.

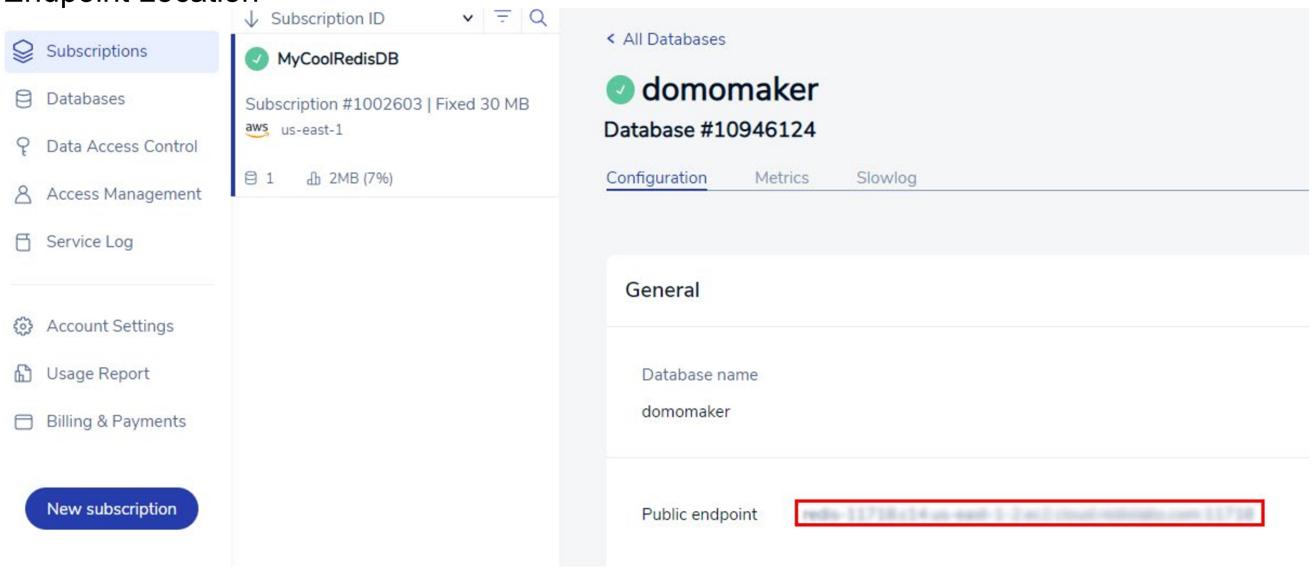
   You can store all sorts of shared variables in Redis that all of your servers can access. This helps with dev (being able to restart node constantly without losing data) and with production (having server redundancy).

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

9. On Heroku, add your `REDISCLOUD_URL` to your config vars. It should follow this format. You can find your redis password and public endpoint on the page for your database on the redislabs website.

   redis://default:REDIS_PASS@PUBLIC_ENDPOINT

Endpoint Location



MyCoolRedisDB
Subscription #1002603 | Fixed 30 MB
aws us-east-1
1 2MB (7%)

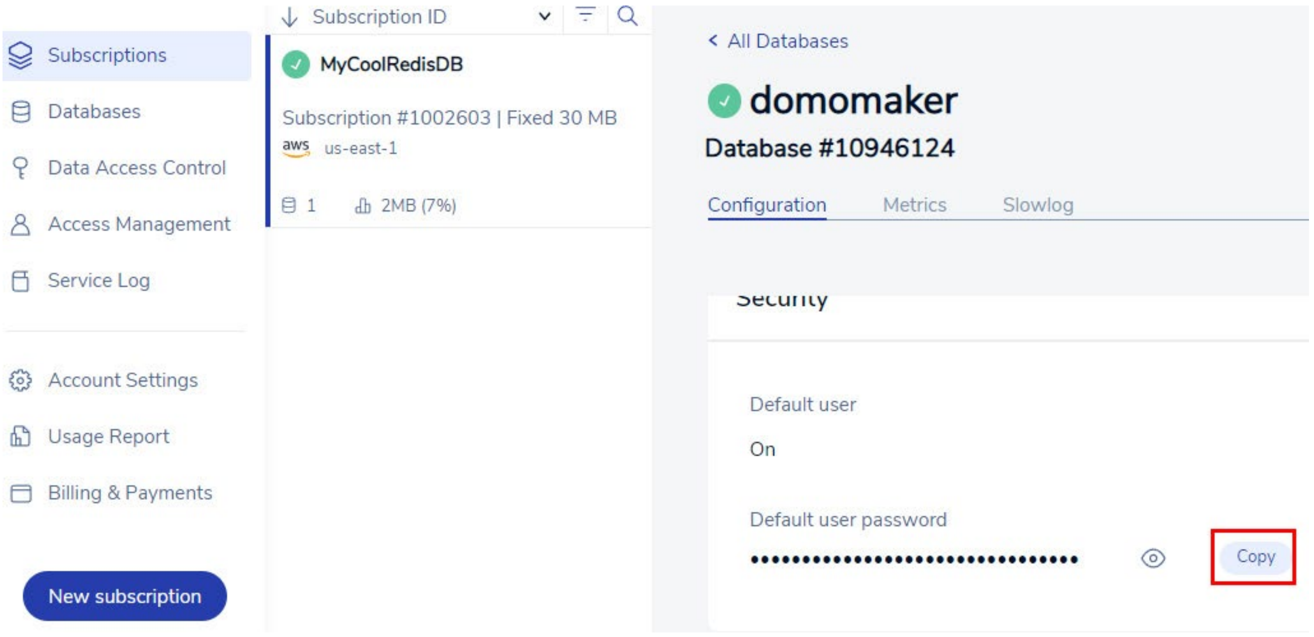
< All Databases
domomaker
Database #10946124

General

Database name
domomaker

Public endpoint [redacted]

Password Location



MyCoolRedisDB
Subscription #1002603 | Fixed 30 MB
aws us-east-1
1 2MB (7%)

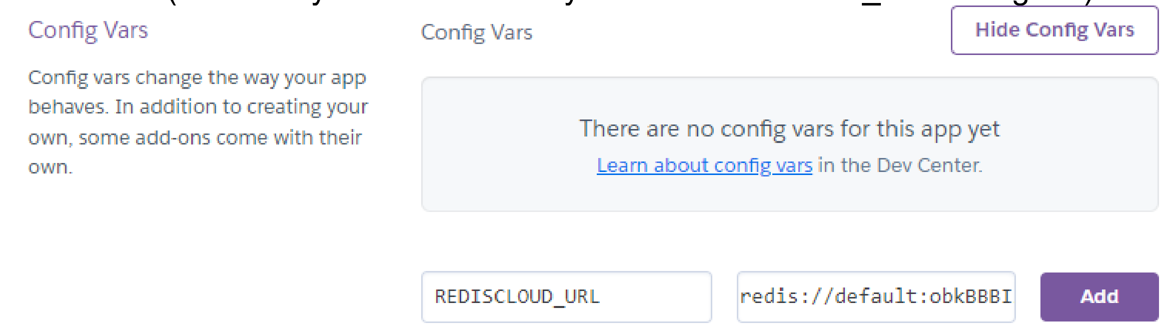
< All Databases
domomaker
Database #10946124

Security

Default user
On

Default user password
•••••••••••••••••••••••••••••••••

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



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

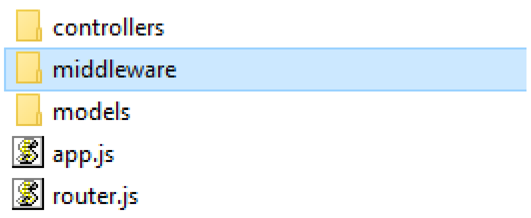
There are no config vars for this app yet
Learn about config vars in the Dev Center.

REDISCLOUD_URL | redis://default:obkBBBI

10. 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.

    Technically 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.



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

12. 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?

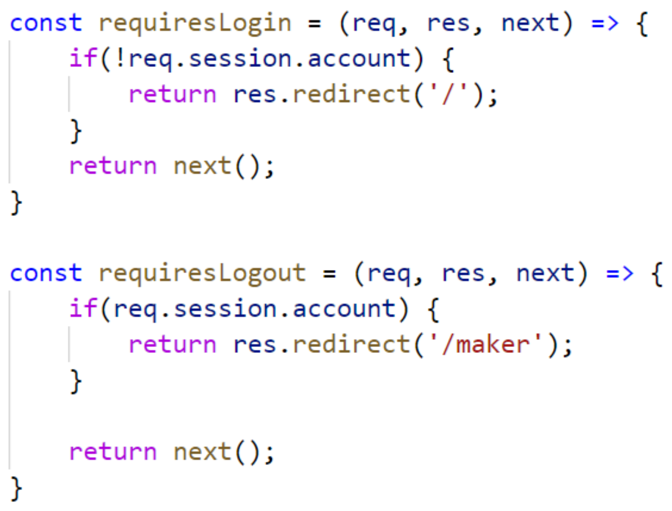
    Middleware functions receive a request, response, and the next middleware function to call. This allows your application to make various decisions and potentially chain into the next middleware call. The request will not continue through the system UNLESS you call the next function at the end.

    In order 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();
}
```

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

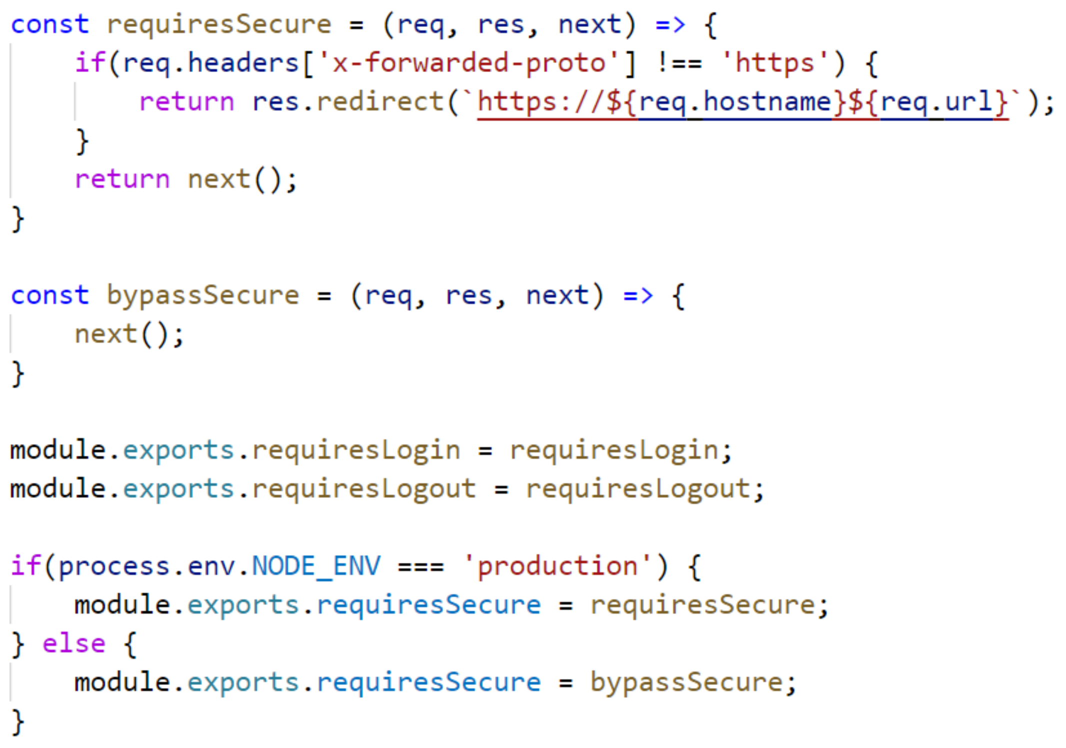
    If the user is trying to do something secure, such as logging in, then we will check to see if they are on HTTPS and redirect them if not.

    ***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.

    Middleware index.js Part II



```
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;
}
```

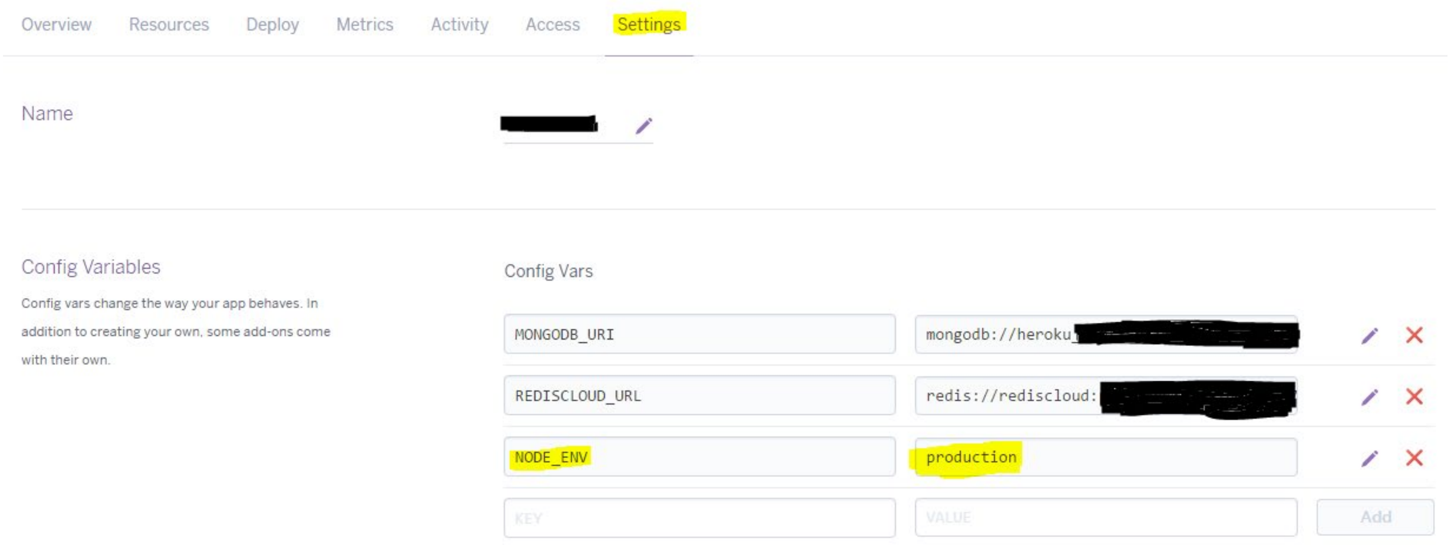
14. For this code to work, we need to add an environment variable on Heroku.

    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".



Name [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

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

15. 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');
```

16. 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.

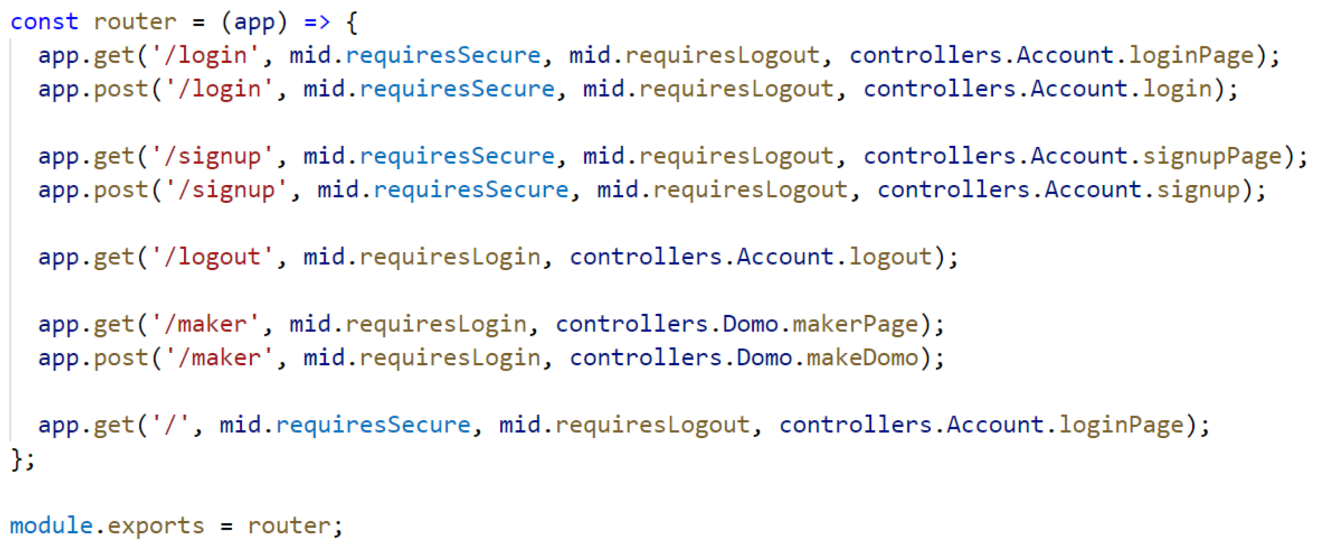
    - For POSTs to /login or /signup we want to make sure it's secure, so we call requiresSecure. We also want to make sure they are logged out (so they can't try to login when logged in), so we call requiresLogout.

    - 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.

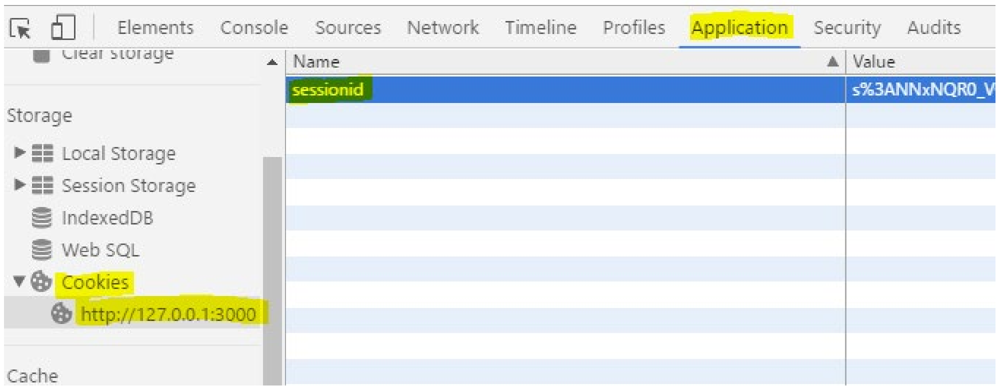
These middleware calls are all functions we made in the middleware file.



```
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;
```

17. 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.



18. 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

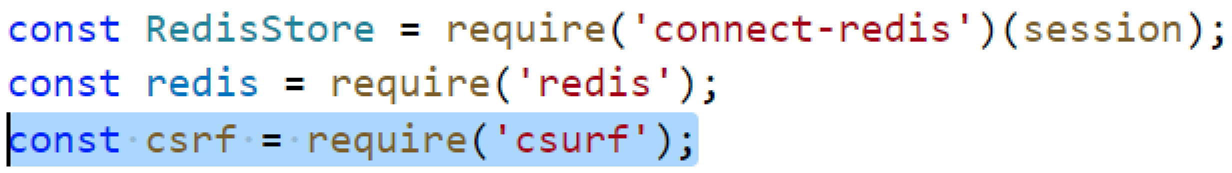
    How does this work? Well, the middleware fires in order on each URL and if it passes the middleware criteria then it calls the next middleware function. If it does not pass the middleware criteria then the flow is broken and a different page is rendered instead of their request. This is what our middleware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

19. Finally, let's prevent people (under most circumstances) from stealing our session key.

    HTTP is stateless, so it waits for a request and with a session the server will check the request for a session cookie. If the cookie exists, then the user has a session. If the cookie does not exist, the page may be restricted or redirected entirely. Each request & response contains the session cookie.

    If someone else gets a hold your session cookie, then they can take your session (known as session hijacking). Sometimes they can do this by doing Cross-Site-Request-Forgery (CSRF). We want to prevent that from happening under basic circumstances.

    In your app.js, pull in the CSurf module. CSurf is a middleware library which automatically generates unique tokens for each user for each page. It then checks for those tokens on requests to prevent forgery from another page. From here on out, we will refer to CSurf as csrf, since that is what we import it as.

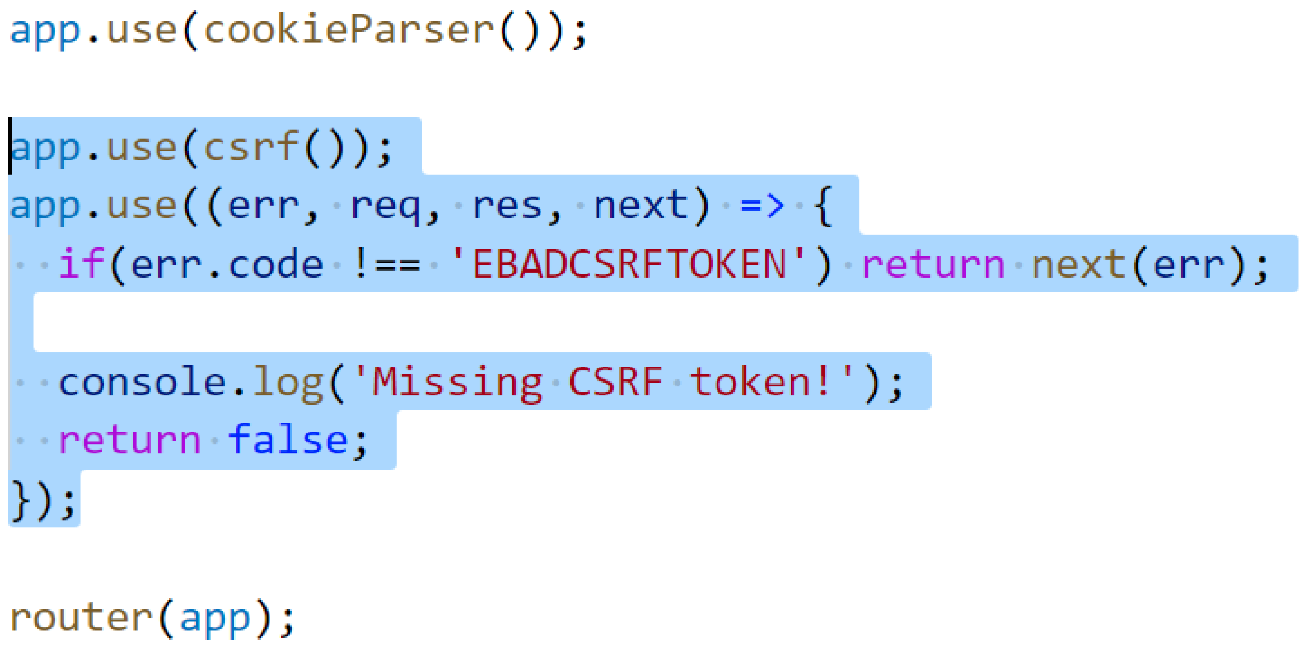


```
const RedisStore = require('connect-redis')(session);
const redis = require('redis');
const csrf = require('csurf');
```

20. Inside of your app.js, after you enable the cookieParser and the sessions but before you setup the router, we need to add csrf. csrf will generate a unique token for each request and requests from the same session must match. If they don't, csrf will create an err called 'EBADCSRFTOKEN' and the requests will not work.

    After your cookieParser call, make a call to csrf and tell the app to use a function to check for anything but the 'EBADCSRFTOKEN' error. If there is a 'EBADCSRFTOKEN' error, we will not do anything. We won't even respond because a bad token here likely means someone is up to non-sense. If we don't respond, they have limited ways of knowing that their request even got to the server. Plus, if someone is up to non-sense, why even bother processing the request.

    If there is not an error, then we will just call the next middleware (next) function.



```
app.use(cookieParser());

app.use(csrf());
app.use((err, req, res, next) => {
  if(err.code !== 'EBADCSRFTOKEN') return next(err);

  console.log('Missing CSRF token!');
  return false;
});

router(app);
```

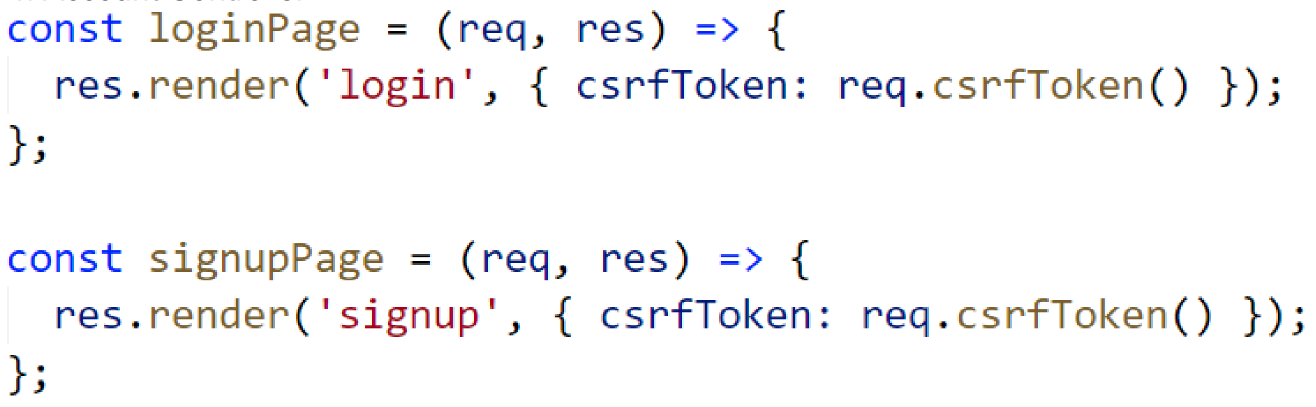
21. Now we have to add the tokens in our views and controllers.

    We only have to do this on functions that render data that will be submitted. In our case, that's the loginPage, signupPage and makerPage functions.

    Since the CSRF module is setup, every request will have a csrfToken function that generates a new token. This will be a new token on every request, which means it only works once. This one-time token combined with the session key makes it hard to forge a request.

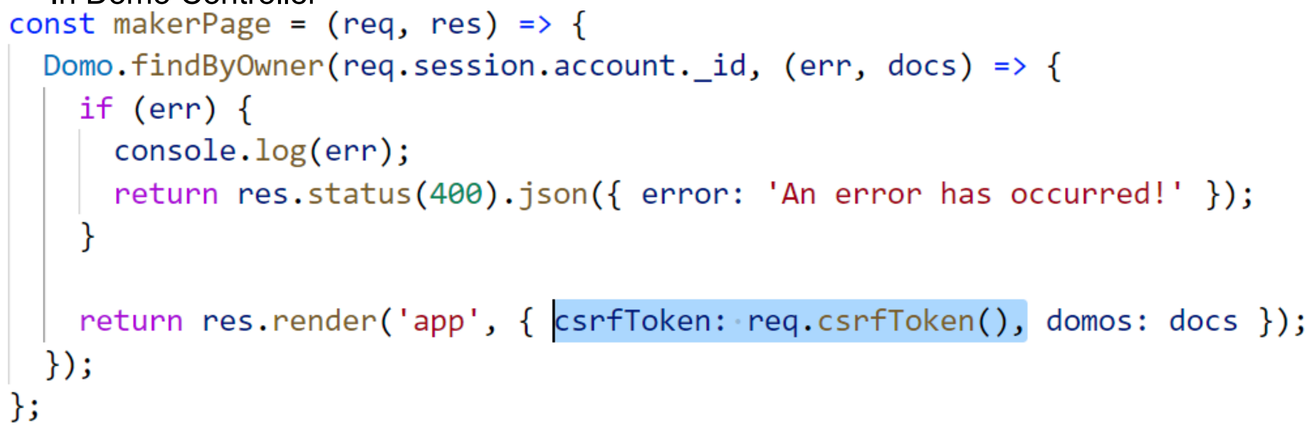
    On each request we will generate a new token and pass it into the view for that request.

    In Account Controller



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

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

In Domo Controller

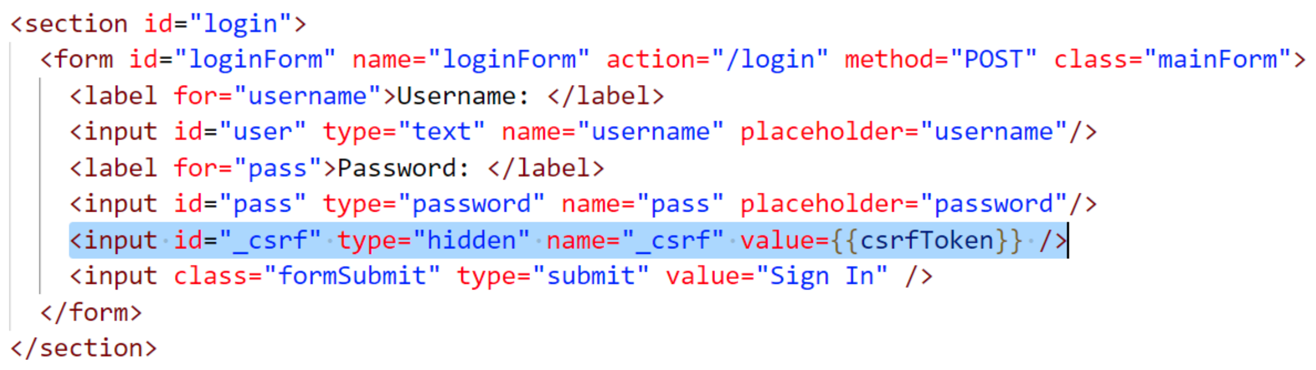


```
const makerPage = (req, res) => {
  Domo.findByOwner(req.session.account._id, (err, docs) => {
    if (err) {
      console.log(err);
      return res.status(400).json({ error: 'An error has occurred!' });
    }

    return res.render('app', { csrfToken: req.csrfToken(), domos: docs });
  });
};
```

22. Next, we need to embed the token into the view in a way that gets submitted. Csrf expects the data to come as a parameter named "_csrf", so we will make a hidden input with the name "_csrf" and the value of our token. This value will change on each page reload. The csrf module tracks all of the tokens that have been assigned to each user on any given page and will expire them when needed.

    Add the tokens into the Handlebars forms for the functions we wanted. This will make sure nothing gets POST to the server without a valid token. **Ensure that there is a space after the {{csrfToken}} and before />**. If you don't, you will accidentally escape part of your csrfToken and it will not work!

    In login.handlebars



```
<section id="login">
  <form id="loginForm" name="loginForm" action="/login" method="POST" class="mainForm">
    <label for="username">Username: </label>
    <input id="user" type="text" name="username" placeholder="username"/>
    <label for="pass">Password: </label>
    <input id="pass" type="password" name="pass" placeholder="password"/>
    <input id="_csrf" type="hidden" name="_csrf" value={{csrfToken}} />
    <input class="formSubmit" type="submit" value="Sign In" />
  </form>
</section>
```

In signup.handlebars

```
<section id="signup">
  <form id="signupForm" name="signupForm" action="/signup" method="POST" class="mainForm">
    <label for="username">Username: </label>
    <input id="user" type="text" name="username" placeholder="username"/>
    <label for="pass">Password: </label>
    <input id="pass" type="password" name="pass" placeholder="password"/>
    <label for="pass2">Password: </label>
    <input id="pass2" type="password" name="pass2" placeholder="retype password"/>
    <input id="_csrf" type="hidden" name="_csrf" value={{csrfToken}} />
    <input class="formSubmit" type="submit" value="Sign Up" />
  </form>
</section>
```

In app.handlebars

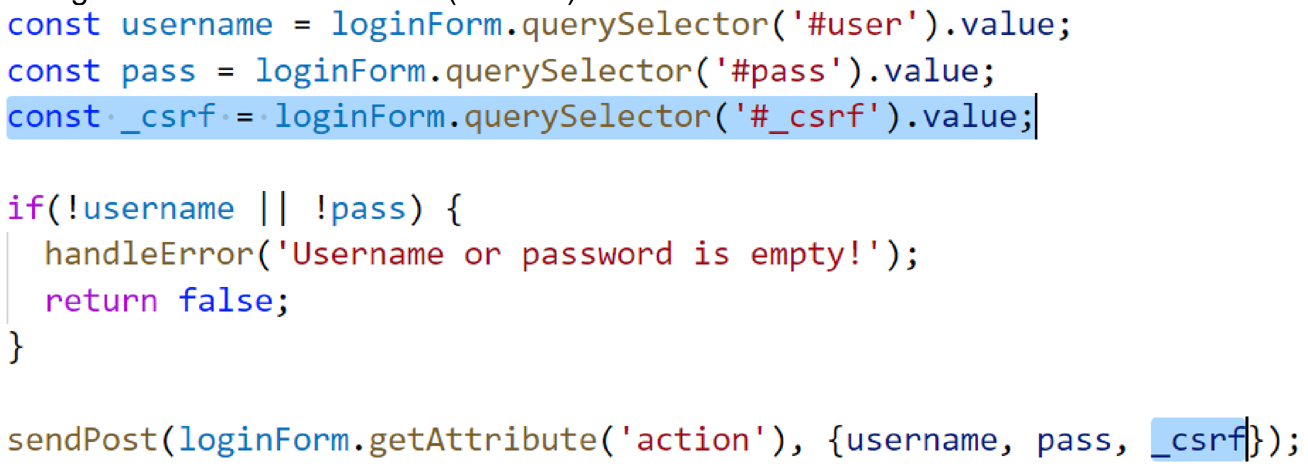
```
<section id="makeDomo">
  <form id="domoForm" name="domoForm" action="/maker" method="POST" class="domoForm">
    <label for="name">Name: </label>
    <input id="domoName" type="text" name="name" placeholder="Domo Name"/>
    <label for="age">Age: </label>
    <input id="domoAge" type="text" name="age" placeholder="Domo Age"/>
    <input id="_csrf" type="hidden" name="_csrf" value={{csrfToken}} />
    <input class="makeDomoSubmit" type="submit" value="Make Domo" />
  </form>
</section>
```

23. We now need to update our client-side code, so that it properly sends the csrf token with our requests. Go to ./client/client.js and add the following:

To signupForm.addEventListener('submit')

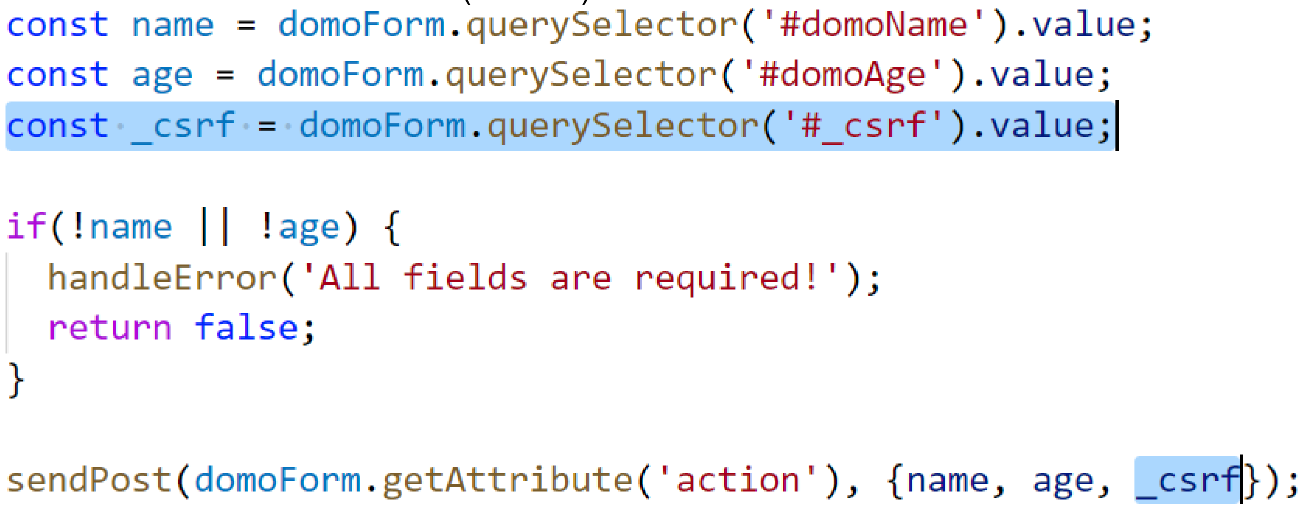
```
const username = signupForm.querySelector('#user').value;
const pass = signupForm.querySelector('#pass').value;
const pass2 = signupForm.querySelector('#pass2').value;
const _csrf = signupForm.querySelector('#_csrf').value;

if(!username || !pass || !pass2) {
  handleError('All fields are required!');
  return false;
}

if(pass !== pass2) {
  handleError('Passwords do not match!');
  return false;
}

sendPost(signupForm.getAttribute('action'), {username, pass, pass2, _csrf});
```

To loginForm.addEventListener('submit')



```
const username = loginForm.querySelector('#user').value;
const pass = loginForm.querySelector('#pass').value;
const _csrf = loginForm.querySelector('#_csrf').value;

if(!username || !pass) {
  handleError('Username or password is empty!');
  return false;
}

sendPost(loginForm.getAttribute('action'), {username, pass, _csrf});
```

To domoForm.addEventListener('submit')

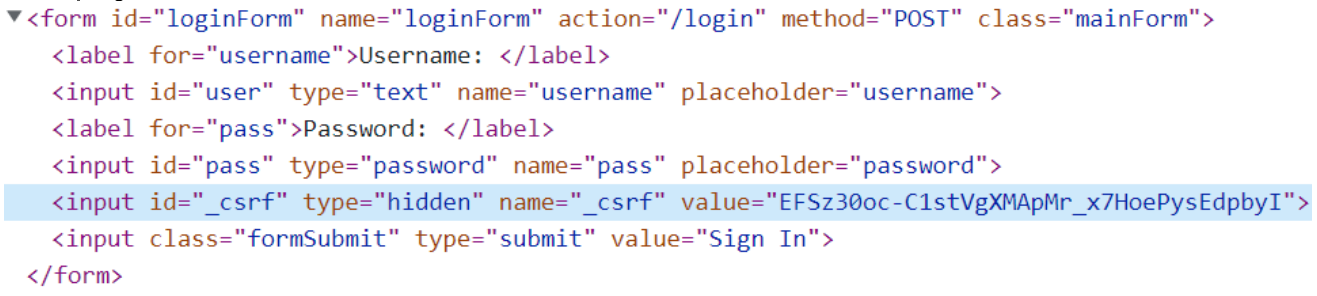


```
const name = domoForm.querySelector('#domoName').value;
const age = domoForm.querySelector('#domoAge').value;
const _csrf = domoForm.querySelector('#_csrf').value;

if(!name || !age) {
  handleError('All fields are required!');
  return false;
}

sendPost(domoForm.getAttribute('action'), {name, age, _csrf});
```

24. After making edits to your client code, remember to run "npm run webpack" to recompile your bundle.js

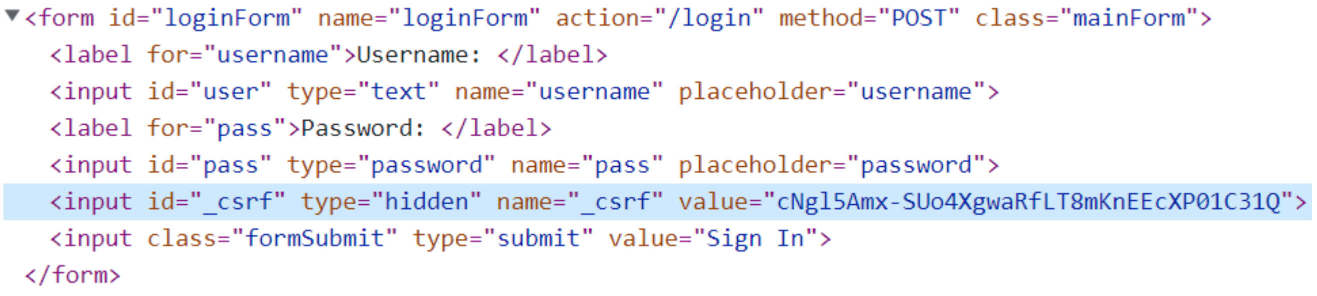
25. Restart your server and load any page. Open the inspector and look at the _csrf input. On every page reload this will change the token, making it difficult for people to steal or guess your _csrf token and session cookie.

First page load



```
▼<form id="loginForm" name="loginForm" action="/login" method="POST" class="mainForm">
   <label for="username">Username: </label>
   <input id="user" type="text" name="username" placeholder="username">
   <label for="pass">Password: </label>
   <input id="pass" type="password" name="pass" placeholder="password">
   <input id="_csrf" type="hidden" name="_csrf" value="EFSz30oc-C1stVgXMApMr_x7HoePysEdpbyI">
   <input class="formSubmit" type="submit" value="Sign In">
 </form>
```

Second page load (notice the different _csrf value)



```
▼<form id="loginForm" name="loginForm" action="/login" method="POST" class="mainForm">
   <label for="username">Username: </label>
   <input id="user" type="text" name="username" placeholder="username">
   <label for="pass">Password: </label>
   <input id="pass" type="password" name="pass" placeholder="password">
   <input id="_csrf" type="hidden" name="_csrf" value="cNgl5Amx-SUo4XgwaRfLT8mKnEEcXP01C31Q">
   <input class="formSubmit" type="submit" value="Sign In">
 </form>
```

If you were to change that value in the inspector and try to submit the form, the server would never respond (which we said not to) because the token does not match that user and page. If the value is right though, then our server would work correctly.

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

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

28. 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.

29. 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.		**15**
Middleware Working & Connected in Routes – The middleware is connected and checked for in the routes. Routes still continue to direct to the correct controllers.		**15**
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.		**15**
HTTPS Redirection – The app successfully redirects to HTTPS from HTTP on Heroku.		**15**
CSRF and Security added – CSRF tokens are working and connected successfully. Forms with valid tokens can be submitted. Forms without valid tokens fail to submit.		**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.*