Information Security with HelmetJS

1. Install and Require Helmet

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. Helmet helps you secure your Express apps by setting various HTTP headers. Install the package, then require it.

Instructions

Challenge Seed

Solution

```
// solution required
```

2. Hide Potentially Dangerous Information Using helmet.hidePoweredBy()

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. Hackers can exploit known vulnerabilities in Express/Node if they see that your site is powered by Express. X-Powered-By: Express is sent in every request coming from Express by default. The helmet.hidePoweredBy() middleware will remove the X-Powered-By header. You can also explicitly set the header to something else, to throw people off. e.g. app.use(helmet.hidePoweredBy({ setTo: 'PHP 4.2.0' }))

Instructions

Challenge Seed

Solution

```
// solution required
```

3. Mitigate the Risk of Clickjacking with helmet.frameguard()

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. Your page could be put in a `<frame>` or `<iframe>` without your consent. This can result in clickjacking attacks, among other things. Clickjacking is a technique of tricking a user into interacting with a page different from what the user thinks it

is. This can be obtained executing your page in a malicious context, by mean of iframing. In that context a hacker can put a hidden layer over your page. Hidden buttons can be used to run bad scripts. This middleware sets the X-Frame-Options header. It restricts who can put your site in a frame. It has three modes: DENY, SAMEORIGIN, and ALLOW-FROM. We don't need our app to be framed. You should use `helmet.frameguard()` passing with the configuration object `{action: 'deny'}`.

Instructions

Challenge Seed

Solution

```
// solution required
```

4. Mitigate the Risk of Cross Site Scripting (XSS) Attacks with helmet.xssFilter()

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. Cross-site scripting (XSS) is a frequent type of attack where malicious scripts are injected into vulnerable pages, with the purpose of stealing sensitive data like session cookies, or passwords. The basic rule to lower the risk of an XSS attack is simple: "Never trust user's input". As a developer you should always sanitize all the input coming from the outside. This includes data coming from forms, GET query urls, and even from POST bodies. Sanitizing means that you should find and encode the characters that may be dangerous e.g. <, >. Modern browsers can help mitigating the risk by adopting better software strategies. Often these are configurable via http headers. The X-XSS-Protection HTTP header is a basic protection. The browser detects a potential injected script using a heuristic filter. If the header is enabled, the browser changes the script code, neutralizing it. It still has limited support.

Instructions

Challenge Seed

Solution

```
// solution required
```

5. Avoid Inferring the Response MIME Type with helmet.noSniff()

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. Browsers can use content or MIME sniffing to adapt to different datatypes coming from a response. They override the Content-Type headers to guess and process the data. While this can be convenient in some scenarios, it can also lead to some dangerous attacks. This middleware sets the X-Content-Type-Options header to nosniff. This instructs the browser to not bypass the provided Content-Type.

Instructions

Challenge Seed

Solution

```
// solution required
```

6. Prevent IE from Opening Untrusted HTML with helmet.ieNoOpen()

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. Some web applications will serve untrusted HTML for download. Some versions of Internet Explorer by default open those HTML files in the context of your site. This means that an untrusted HTML page could start doing bad things in the context of your pages. This middleware sets the X-Download-Options header to noopen. This will prevent IE users from executing downloads in the trusted site's context.

Instructions

Challenge Seed

Solution

```
// solution required
```

7. Ask Browsers to Access Your Site via HTTPS Only with helmet.hsts()

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. HTTP Strict Transport Security (HSTS) is a web security policy which helps to protect websites against protocol downgrade attacks and cookie hijacking. If your website can be accessed via HTTPS you can ask user's browsers to avoid using insecure HTTP. By setting the header Strict-Transport-Security, you tell the browsers to use HTTPS for the future requests in a specified amount of time. This will work for the requests coming after the initial request. Configure helmet.hsts() to use HTTPS for the next 90 days. Pass the config object {maxAge: timeInMilliseconds, force: true}. Glitch already has hsts enabled. To override its settings you need to set the field "force" to true in the config object. We will intercept and restore the Glitch header, after inspecting it for testing. Note: Configuring HTTPS on a custom website requires the acquisition of a domain, and a SSL/TSL Certificate.

Instructions

Challenge Seed

Solution

```
// solution required
```

8. Disable DNS Prefetching with helmet.dnsPrefetchControl()

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. To improve performance, most browsers prefetch DNS records for the links in a page. In that way the destination ip is already known when the user clicks on a link. This may lead to over-use of the DNS service (if you own a big website, visited by millions people...), privacy issues (one eavesdropper could infer that you are on a certain page), or page statistics alteration (some links may appear visited even if they are not). If you have high security needs you can disable DNS prefetching, at the cost of a performance penalty.

Instructions

Challenge Seed

Solution

```
// solution required
```

9. Disable Client-Side Caching with helmet.noCache()

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. If you are releasing an update for your website, and you want the users to always download the newer version, you can (try to) disable caching on client's browser. It can be useful in development too. Caching has performance benefits, which you will lose, so only use this option when there is a real need.

Instructions

Challenge Seed

Solution

```
// solution required
```

10. Set a Content Security Policy with helmet.contentSecurityPolicy()

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. This challenge highlights one promising new defense that can significantly reduce the risk and impact of many type of attacks in modern browsers. By setting and configuring a Content Security Policy you can prevent the injection of anything unintended into your page. This will protect your app from XSS vulnerabilities, undesired tracking, malicious frames, and much more. CSP works by defining a whitelist of content sources which are trusted. You can configure

them for each kind of resource a web page may need (scripts, stylesheets, fonts, frames, media, and so on...). There are multiple directives available, so a website owner can have a granular control. See HTML 5 Rocks, KeyCDN for more details. Unfortunately CSP is unsupported by older browser. By default, directives are wide open, so it's important to set the defaultSrc directive as a fallback. Helmet supports both defaultSrc and default-src naming styles. The fallback applies for most of the unspecified directives. In this exercise, use helmet.contentSecurityPolicy(), and configure it setting the defaultSrc directive to ["self"] (the list of allowed sources must be in an array), in order to trust only your website address by default. Set also the scriptSrc directive so that you will allow scripts to be downloaded from your website, and from the domain 'trusted-cdn.com'. Hint: in the "'self'" keyword, the single quotes are part of the keyword itself, so it needs to be enclosed in double quotes to be working.

Instructions

Challenge Seed

Solution

```
// solution required
```

11. Configure Helmet Using the 'parent' helmet() Middleware

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. app.use(helmet()) will automatically include all the middleware introduced above, except noCache(), and contentSecurityPolicy(), but these can be enabled if necessary. You can also disable or configure any other middleware individually, using a configuration object. // Example `app.use(helmet({ frameguard: { // configure action: 'deny' }, contentSecurityPolicy: { // enable and configure directives: { defaultSrc: ["self"], styleSrc: ['style.com'], } }, dnsPrefetchControl: false // disable }))` We introduced each middleware separately for teaching purpose, and for ease of testing. Using the 'parent' helmet() middleware is easiest, and cleaner, for a real project.

Instructions

Challenge Seed

Solution

```
// solution required
```

12. Understand BCrypt Hashes

Description

For the following challenges, you will be working with a new starter project that is different from earlier challenges. This project is being built upon the following starter project on Glitch, or cloned from GitHub. BCrypt hashes are very secure. A hash is basically a fingerprint of the original data- always unique. This is accomplished by feeding the original data into a algorithm and having returned a fixed length result. To further complicate this process and make it more secure, you can also *salt* your hash. Salting your hash involves adding random data to the original data before the hashing process which makes it even harder to crack the hash. BCrypt hashes will always looks like

`$2a$13$ZyprE5MRw2Q3WpNOGZWGbeG7ADUre1Q8QO.uUUtcbqloUOyvzavOm` which does have a structure. The first small bit of data `$2a` is defining what kind of hash algorithm was used. The next portion `$13` defines the *cost*. Cost is about how much power it takes to compute the hash. It is on a logarithmic scale of 2^cost and determines how many times the data is put through the hashing algorithm. For example, at a cost of 10 you are able to hash 10 passwords a second on an average computer, however at a cost of 15 it takes 3 seconds per hash... and to take it further, at a cost of 31 it would takes multiple days to complete a hash. A cost of 12 is considered very secure at this time. The last portion of your hash `$ZyprE5MRw2Q3WpNOGZWGbeG7ADUre1Q8QO.uUUtcbqloUOyvzavOm`, looks like 1 large string of numbers, periods, and letters but it is actually 2 separate pieces of information. The first 22 characters is the salt in plain text, and the rest is the hashed password!

To begin using BCrypt, add it as a dependency in your project and require it as 'bcrypt' in your server. Submit your page when you think you've got it right.

Instructions

Challenge Seed

Solution

```
// solution required
```

13. Hash and Compare Passwords Asynchronously

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. As hashing is designed to be computationally intensive, it is recommended to do so asyncronously on your server as to avoid blocking incoming connections while you hash. All you have to do to hash a password asynchronous is call `bcrypt.hash(myPlaintextPassword, saltRounds, (err, hash) => { /*Store hash in your db*/ });`

Add this hashing function to your server(we've already defined the variables used in the function for you to use) and log it to the console for you to see! At this point you would normally save the hash to your database. Now when you need to figure out if a new input is the same data as the hash you would just use the compare function `bcrypt.compare(myPlaintextPassword, hash, (err, res) => { /*res == true or false*/ });`. Add this into your existing hash function(since you need to wait for the hash to complete before calling the compare function) after you log the completed hash and log 'res' to the console within the compare. You should see in the console a hash then 'true' is printed! If you change 'myPlaintextPassword' in the compare function to 'someOtherPlaintextPassword' then it should say false.

```
bcrypt.hash('passwOrd!', 13, (err, hash) => {
  console.log(hash); //$2a$12$Y.PHPE15wR25qrrtgGkiYe2sXo98cjuMCG1YwSI5rJW1DSJpOgEYS
  bcrypt.compare('passwOrd!', hash, (err, res) => {
      console.log(res); //true
  });
});
```

Submit your page when you think you've got it right.

Instructions

Challenge Seed

Solution

```
// solution required
```

14. Hash and Compare Passwords Synchronously

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. Hashing synchronously is just as easy to do but can cause lag if using it server side with a high cost or with hashing done very often. Hashing with this method is as easy as calling `var hash = bcrypt.hashSync(myPlaintextPassword, saltRounds);`

Add this method of hashing to your code and then log the result to the console. Again, the variables used are already defined in the server so you wont need to adjust them. You may notice even though you are hashing the same password as in the async function, the result in the console is different- this is due to the salt being randomly generated each time as seen by the first 22 characters in the third string of the hash. Now to compare a password input with the new sync hash, you would use the compareSync method: `var result = bcrypt.compareSync(myPlaintextPassword, hash);` with the result being a boolean true or false. Add this function in and log to the console the result to see it working. Submit your page when you think you've got it right. If you ran into errors during these challenges you can take a look at the example completed code here.

Instructions

Challenge Seed

Solution

```
// solution required
```

Quality Assurance and Testing with Chai

1. Learn How JavaScript Assertions Work

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. Use assert.isNull() or assert.isNotNull() to make the tests pass.

Instructions

Challenge Seed

Solution

```
// solution required
```

2. Test if a Variable or Function is Defined

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. Use assert.isDefined() or assert.isUndefined() to make the tests pass

Instructions

Challenge Seed

Solution

```
// solution required
```

3. Use Assert.isOK and Assert.isNotOK

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. Use assert.isOk() or assert.isNotOk() to make the tests pass. .isOk(truthy) and .isNotOk(falsey) will pass.

Instructions

Challenge Seed

Solution

```
// solution required
```

4. Test for Truthiness

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. Use assert.isTrue() or assert.isNotTrue() to make the tests pass. .isTrue(true) and .isNotTrue(everything else) will pass. .isFalse() and .isNotFalse() also exist.

Instructions

Challenge Seed

Solution

```
// solution required
```

5. Use the Double Equals to Assert Equality

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. .equal(), .notEqual() .equal() compares objects using '=='

Instructions

Challenge Seed

Solution

```
// solution required
```

6. Use the Triple Equals to Assert Strict Equality

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. .strictEqual(), .notStrictEqual() .strictEqual() compares objects using '==='

Instructions

Challenge Seed

Solution

```
// solution required
```

7. Assert Deep Equality with .deepEqual and .notDeepEqual

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. .deepEqual(), .notDeepEqual() .deepEqual() asserts that two object are deep equal

Instructions

Challenge Seed

Solution

```
// solution required
```

8. Compare the Properties of Two Elements

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. .isAbove() => a > b , .isAtMost() => a <= b

Instructions

Challenge Seed

Solution

```
// solution required
```

9. Test if One Value is Below or At Least as Large as Another

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. .isBelow() => a < b , .isAtLeast => a >= b

Instructions

Challenge Seed

Solution

```
// solution required
```

10. Test if a Value Falls within a Specific Range

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. .approximately .approximately(actual, expected, range, [message]) actual = expected +/- range Choose the minimum range (3rd parameter) to make the test always pass it should be less than 1

Instructions

Challenge Seed

Solution

```
// solution required
```

11. Test if a Value is an Array

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub.

Instructions

Challenge Seed

Solution

```
// solution required
```

12. Test if an Array Contains an Item

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub.

Instructions

Challenge Seed

Solution

```
// solution required
```

13. Test if a Value is a String

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. #isString asserts that the actual value is a string.

Instructions

Challenge Seed

Solution

```
// solution required
```

14. Test if a String Contains a Substring

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. #include (on #notInclude) works for strings too !! It asserts that the actual string contains the expected substring

Instructions

Challenge Seed

Solution

```
// solution required
```

15. Use Regular Expressions to Test a String

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. #match Asserts that the actual value matches the second argument regular expression.

Instructions

Challenge Seed

Solution

```
// solution required
```

16. Test if an Object has a Property

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. #property asserts that the actual object has a given property. Use #property or #notProperty where appropriate

Instructions

Challenge Seed

Solution

```
// solution required
```

17. Test if a Value is of a Specific Data Structure Type

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. #typeOf asserts that value's type is the given string, as determined by Object.prototype.toString. Use #typeOf or #notTypeOf where appropriate

Instructions

Challenge Seed

Solution

```
// solution required
```

18. Test if an Object is an Instance of a Constructor

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. #instanceOf asserts that an object is an instance of a constructor. Use #instanceOf or #notInstanceOf where appropriate

Instructions

Challenge Seed

Solution

```
// solution required
```

19. Run Functional Tests on API Endpoints using Chai-HTTP

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. Replace assert.fail(). Test the status and the text.response. Make the test pass. Don't send a name in the query, the endpoint with responds with 'hello Guest'.

Instructions

Challenge Seed

Solution

```
// solution required
```

20. Run Functional Tests on API Endpoints using Chai-HTTP II

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. Replace assert.fail(). Test the status and the text.response. Make the test pass. Send you name in the query appending ?name=, the endpoint with responds with 'hello '.

Instructions

Challenge Seed

Solution

```
// solution required
```

21. Run Functional Tests on an API Response using Chai-HTTP III - PUT method

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. In the next example we'll see how to send data in a request payload (body). We are going to test a PUT request. The '/travellers' endpoint accepts a JSON object taking the structure : {surname: [last name of a traveller of the past]} , The route responds with : {name: [first name], surname:[last name], dates: [birth - death years]} see the server code for more details. Send {surname: 'Colombo'}. Replace assert.fail() and make the test pass. Check for 1) status, 2) type, 3) body.name, 4) body.surname Follow the assertion order above, We rely on it.

Instructions

Challenge Seed

Solution

```
// solution required
```

22. Run Functional Tests on an API Response using Chai-HTTP IV - PUT method

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. This exercise is similar to the preceding. Look at it for the details. Send {surname: 'da Verrazzano'}. Replace assert.fail() and make the test pass. Check for 1) status, 2) type, 3) body.name, 4) body.surname Follow the assertion order above, We rely on it.

Instructions

Challenge Seed

Solution

```
// solution required
```

23. Run Functional Tests using a Headless Browser

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. In the next challenges we are going to simulate the human interaction with a page using a device called 'Headless Browser'. A headless browser is a web browser without a graphical user interface. These kind of tools are particularly useful for testing web pages as they are able to render and understand HTML, CSS, and JavaScript the same way a browser would. For these challenges we are using Zombie.JS. It's a lightweight browser which is totally based on JS, without relying on additional binaries to be installed. This feature makes it usable in an environment such as Glitch. There are many other (more powerful) options.
Look at the examples in the code for the exercise directions Follow the assertions order, We rely on it.

Instructions

Challenge Seed

Solution

```
// solution required
```

24. Run Functional Tests using a Headless Browser II

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. This exercise is similar to the preceding. Look at the code for directions. Follow the assertions order, We rely on it.

Instructions

Challenge Seed

Solution

```
// solution required
```

Advanced Node and Express

1. Set up a Template Engine

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. A template engine enables you to use static template files (such as those written in *Pug*) in your app. At runtime, the template engine replaces variables in a template file with actual values which can be supplied by your server, and transforms the template into a static HTML file that is then sent to the client. This approach makes it easier to design an HTML page and allows for displaying of variables on the page without needing to make an API call from the client. To set up *Pug* for use in your project, you will need to add it as a dependency first in your package.json. `"pug": "^0.1.0"` Now to tell Node/Express to use the templating engine you will have to tell your express **app** to **set** 'pug' as the 'view-engine'. `app.set('view engine', 'pug')` Lastly, you should change your response to the request for the index route to `res.render` with the path to the view *views/pug/index.pug*. If all went as planned, you should refresh your apps home page and see a small message saying you're successfully rending the Pug from our Pug file! Submit your page when you think you've got it right.

Instructions

Challenge Seed

Solution

```
// solution required
```

2. Use a Template Engine's Powers

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. One of the greatest features of using a template engine is being able to pass variables from the server to the template file before rendering it to HTML. In your Pug file, you're about to use a variable by referencing the variable name as `#{variable_name}` inline with other text on an element or by using an equal side on the element without a space such as `p= variable_name` which sets that p elements text to equal the variable. We strongly recommend looking at the syntax and structure of Pug here on their Githubs README. Pug is all about using whitespace and tabs to show nested elements and cutting down on the amount of code needed to make a beautiful site. Looking at our pug file 'index.pug' included in your project, we used the variables *title* and *message* To pass those alone from our server, you will need to add an object as a second argument to your *res.render* with the variables and their value. For example, pass this object along setting the variables for your index view: `{title: 'Hello', message: 'Please login'` It should look like: `res.render(process.cwd() + '/views/pug/index', {title: 'Hello', message: 'Please login'});` Now refresh your page and you should see those values rendered in your view in the correct spot as laid out in your index.pug file! Submit your page when you think you've got it right.

Instructions

Challenge Seed

Solution

```
// solution required
```

3. Set up Passport

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. It's time to set up *Passport* so we can finally start allowing a user to register or login to an account! In addition to Passport, we will use Express-session to handle sessions. Using this middleware saves the session id as a cookie in the client and allows us to access the session data using that id on the server. This way we keep personal account information out of the cookie used by the client to verify to our server they are authenticated and just keep the *key* to access the data stored on the server. To set up Passport for use in your project, you will need to add it as a dependency first in your package.json. `"passport": "^0.3.2"` In addition, add Express-session as a dependency now as well. Express-session has a ton of advanced features you can use but for now we're just going to use the basics! `"express-session": "^1.15.0"` You will need to set up the session settings now and initialize Passport. Be sure to first create the variables 'session' and 'passport' to require 'express-session' and 'passport' respectively. To set up your express app to use use the session we'll define just a few basic options. Be sure to add 'SESSION_SECRET' to your .env file and give it a random value. This is used to compute the hash used to encrypt your cookie!

```
app.use(session({
  secret: process.env.SESSION_SECRET,
  resave: true,
  saveUninitialized: true,
}));
```

As well you can go ahead and tell your express app to **use** 'passport.initialize()' and 'passport.session()'. (For example, `app.use(passport.initialize());`) Submit your page when you think you've got it right. If you're running into errors, you can check out the project completed up to this point here.

Instructions

Challenge Seed

Solution

```
// solution required
```

4. Serialization of a User Object

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. Serialization and deserialization are important concepts in regards to authentication. To serialize an object means to convert its contents into a small *key* essentially that can then be deserialized into the original object. This is what allows us to know whos communicated with the server without having to send the authentication data like username and password at each request for a new page. To set this up properly, we need to have a serialize function and a deserialize function. In passport we create these with `passport.serializeUser( OURFUNCTION )` and `passport.deserializeUser( OURFUNCTION )` The serializeUser is called with 2 arguments, the full user object and a callback used by passport. Returned in the callback should be a unique key to identify that user- the easiest one to use being the users _id in the object as it should be unique as it generated by MongoDb. Similarly deserializeUser is called with that key and a callback function for passport as well, but this time we have to take that key and return the users full object to the callback. To make a query search for a Mongo _id you will have to create `const ObjectID = require('mongodb').ObjectID;` , and then to use it you call `new ObjectID(THE_ID)` . Be sure to add MongoDB as a dependency. You can see this in the examples below:

```
passport.serializeUser((user, done) => {
   done(null, user._id);
 });
```

```
passport.deserializeUser((id, done) => {
        db.collection('users').findOne(
            {_id: new ObjectID(id)},
            (err, doc) => {
                done(null, doc);
            }
        );
    });
```

NOTE: This deserializeUser will throw an error until we set up the DB in the next step so comment out the whole block and just call `done(null, null)` in the function deserializeUser. Submit your page when you think you've got it right.

Instructions

Challenge Seed

Solution

```
// solution required
```

5. Implement the Serialization of a Passport User

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. Right now we're not loading an actual user object since we haven't set up our database. This can be done many different ways, but for our project we will connect to the database once when we start the server and keep a persistent connection for the full life-cycle of the app. To do this, add MongoDB as a dependency and require it in your server. (`const mongo = require('mongodb').MongoClient;`) Now we want to the connect to our database then start listening for requests. The purpose of this is to not allow requests before our database is connected or if there is a database error. To accomplish you will want to encompass your serialization and your app listener in the following:

```
mongo.connect(process.env.DATABASE, (err, db) => {
    if(err) {
        console.log('Database error: ' + err);
    } else {
        console.log('Successful database connection');

      //serialization and app.listen


}});
```

You can now uncomment the block in deserializeUser and remove your `done(null, null)` . Be sure to set *DATABASE* in your .env file to your database's connection string (for example: `DATABASE=mongodb://admin:pass@mlab.com:12345/my-project`). You can set up a free database on mLab. Congratulations- you've finished setting up serialization! Submit your page when you think you've got it right. If you're running into errors, you can check out the project completed up to this point here.

Instructions

Challenge Seed

Solution

```
// solution required
```

6. Authentication Strategies

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. A strategy is a way of authenticating a user. You can use a strategy for allowing users to authenticate based on locally saved information (if you have them register first) or from a variety of providers such as Google or GitHub. For this project we will set up a local strategy. To see a list of the 100's of strategies, visit Passports site here. Add *passport-local* as a dependency and add it to your server as follows: `const LocalStrategy = require('passport-local');` Now you will have to tell passport to **use** an instantiated LocalStartegy object with a few settings defined. Make sure this as well as everything from this point on is encapsulated in the database connection since it relies on it!

```
passport.use(new LocalStrategy(
  function(username, password, done) {
    db.collection('users').findOne({ username: username }, function (err, user) {
      console.log('User '+ username +' attempted to log in.');
      if (err) { return done(err); }
      if (!user) { return done(null, false); }
      if (password !== user.password) { return done(null, false); }
      return done(null, user);
    });
  }
));
```

This is defining the process to take when we try to authenticate someone locally. First it tries to find a user in our database with the username entered, then it checks for the password to match, then finally if no errors have popped up that we checked for, like an incorrect password, the users object is returned and they are authenticated. Many strategies are set up using different settings, general it is easy to set it up based on the README in that strategies repository though. A good example of this is the GitHub strategy where we don't need to worry about a username or password because the user will be sent to GitHub's auth page to authenticate and as long as they are logged in and agree then GitHub returns their profile for us to use. In the next step we will set up how to actually call the authentication strategy to validate a user based on form data! Submit your page when you think you've got it right up to this point.

Instructions

Challenge Seed

Solution

```
// solution required
```

7. How to Use Passport Strategies

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. In the index.pug file supplied there is actually a login form. It has previously been hidden because of the inline JavaScript `if showLogin` with the form indented after it. Before `showLogin` as a variable was never defined, it never rendered the code block containing the form. Go ahead and on the `res.render` for that page add a new variable to the object `showLogin: true` . When you refresh your page, you should then see the form! This form is set up to **POST** on */login* so this is where we should set up to accept the POST and authenticate the user. For this challenge you should add the route /login to accept a POST request. To authenticate on this route you need to add a middleware to do so before

then sending a response. This is done by just passing another argument with the middleware before your `function(req,res)` with your response! The middleware to use is `passport.authenticate('local')`. *passport.authenticate* can also take some options as an argument such as: `{ failureRedirect: '/' }` which is incredibly useful so be sure to add that in as well. As a response after using the middleware (which will only be called if the authentication middleware passes) should be to redirect the user to */profile* and that route should render the view 'profile.pug'. If the authentication was successful, the user object will be saved in *req.user*. Now at this point if you enter a username and password in the form, it should redirect to the home page */* and in the console of your server should be 'User {USERNAME} attempted to log in.' since we currently cannot login a user who isn't registered. Submit your page when you think you've got it right. If you're running into errors, you can check out the project completed up to this point here.

Instructions

Challenge Seed

Solution

```
// solution required
```

8. Create New Middleware

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. As in, any user can just go to /profile whether they authenticated or not by typing in the url. We want to prevent this by checking if the user is authenticated first before rendering the profile page. This is the perfect example of when to create a middleware. The challenge here is creating the middleware function `ensureAuthenticated(req, res, next)`, which will check if a user is authenticated by calling passports isAuthenticated on the *request* which in turn checks for *req.user* is to be defined. If it is then *next()* should be called, otherwise we can just respond to the request with a redirect to our homepage to login. An implementation of this middleware is:

```
function ensureAuthenticated(req, res, next) {
  if (req.isAuthenticated()) {
      return next();
  }
  res.redirect('/');
};
```

Now add *ensureAuthenticated* as a middleware to the request for the profile page before the argument to the get request containing the function that renders the page.

```
app.route('/profile')
  .get(ensureAuthenticated, (req,res) => {
       res.render(process.cwd() + '/views/pug/profile');
  });
```

Submit your page when you think you've got it right.

Instructions

Challenge Seed

Solution

```
// solution required
```

9. How to Put a Profile Together

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. Now that we can ensure the user accessing the */profile* is authenticated, we can use the information contained in 'req.user' on our page! Go ahead and pass the object containing the variable *username* equaling 'req.user.username' into the render method of the profile view. Then go to your 'profile.pug' view and add the line `h2.center#welcome Welcome, #{username}!` creating the h2 element with the class 'center' and id 'welcome' containing the text 'Welcome, ' and the username! Also in the profile, add a link to */logout*. That route will host the logic to unauthenticate a user. `a(href='/logout') Logout` Submit your page when you think you've got it right.

Instructions

Challenge Seed

Solution

```
// solution required
```

10. Logging a User Out

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. Creating the logout logic is easy. The route should just unauthenticate the user and redirect to the home page instead of rendering any view. In passport, unauthenticating a user is as easy as just calling `req.logout();` before redirecting.

```
app.route('/logout')
  .get((req, res) => {
      req.logout();
      res.redirect('/');
  });
```

You may have noticed that we're not handling missing pages (404), the common way to handle this in Node is with the following middleware. Go ahead and add this in after all your other routes:

```
app.use((req, res, next) => {
  res.status(404)
    .type('text')
    .send('Not Found');
});
```

Submit your page when you think you've got it right.

Instructions

Challenge Seed

Solution

```
// solution required
```

11. Registration of New Users

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. Now we need to allow a new user on our site to register an account. On the res.render for the home page add a new variable to the object passed along- `showRegistration: true` . When you refresh your page, you should then see the registration form that was already created in your index.pug file! This form is set up to **POST** on */register* so this is where we should set up to accept the POST and create the user object in the database. The logic of the registration route should be as follows: Register the new user > Authenticate the new user > Redirect to /profile The logic of step 1, registering the new user, should be as follows: Query database with a findOne command > if user is returned then it exists and redirect back to home *OR* if user is undefined and no error occurs then 'insertOne' into the database with the username and password and as long as no errors occur then call *next* to go to step 2, authenticating the new user, which we've already written the logic for in our POST /login route.

```
app.route('/register')
  .post((req, res, next) => {
      db.collection('users').findOne({ username: req.body.username }, function (err, user) {
          if(err) {
              next(err);
          } else if (user) {
              res.redirect('/');
          } else {
              db.collection('users').insertOne(
                {username: req.body.username,
                 password: req.body.password},
                (err, doc) => {
                    if(err) {
                        res.redirect('/');
                    } else {
                        next(null, user);
                    }
                }
              )
          }
      })},
    passport.authenticate('local', { failureRedirect: '/' }),
    (req, res, next) => {
        res.redirect('/profile');
    }
);
```

Submit your page when you think you've got it right. If you're running into errors, you can check out the project completed up to this point here.

Instructions

Challenge Seed

Solution

```
// solution required
```

12. Hashing Your Passwords

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. Going back to the information security section you may remember that storing plaintext passwords is *never* okay. Now it is time to implement BCrypt to solve this issue.

Add BCrypt as a dependency and require it in your server. You will need to handle hashing in 2 key areas: where you handle registering/saving a new account and when you check to see that a password is correct on login. Currently on our registeration route, you insert a user's password into the database like the following: `password: req.body.password` . An easy way to implement saving a hash instead is to add the following before your database logic `var hash = bcrypt.hashSync(req.body.password, 12);` and replacing the `req.body.password` in the database saving with just `password: hash` . Finally on our authentication strategy we check for the following in our code before completing the process: `if (password !== user.password) { return done(null, false); }` . After making the previous changes, now `user.password` is a hash. Before making a change to the existing code, notice how the statement is checking if the password is NOT equal then return non-authenticated. With this in mind your code could look as follows to properly check the password entered against the hash: `if (!bcrypt.compareSync(password, user.password)) { return done(null, false); }` That is all it takes to implement one of the most important security features when you have to store passwords! Submit your page when you think you've got it right.

Instructions

Challenge Seed

Solution

```
// solution required
```

13. Clean Up Your Project with Modules

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. Right now everything you have is in your server.js file. This can lead to hard to manage code that isn't very expandable. Create 2 new files: Routes.js and Auth.js Both should start with the following code:

```
module.exports = function (app, db) {
}
```

Now in the top of your server file, require these files like such: `const routes = require('./routes.js');` Right after you establish a successful connect with the database instantiate each of them like such: `routes(app, db)` Finally, take all of the routes in your server and paste them into your new files and remove them from your server file. Also take the ensureAuthenticated since we created that middleware function for routing specifically. You will have to now correctly add the dependencies in that are used, such as `const passport = require('passport');` , at the very top above the export line in your routes.js file. Keep adding them until no more errors exist, and your server file no longer has any routing! Now do the same thing in your auth.js file with all of the things related to authentication such as the serialization and the setting up of the local strategy and erase them from your server file. Be sure to add the dependencies in and call `auth(app,db)` in the server in the same spot. Be sure to have `auth(app, db)` before `routes(app, db)` since our registration route depends on passport being initiated! Congratulations- you're at the end of this section of Advanced Node and Express and have some beautiful code to show for it! Submit your page when you think you've got it right. If you're running into errors, you can check out an example of the completed project here.

Instructions

Challenge Seed

Solution

```
// solution required
```

14. Implementation of Social Authentication

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. The basic path this kind of authentication will follow in your app is:

1. User clicks a button or link sending them to our route to authenticate using a specific strategy (EG. GitHub)
2. Your route calls `passport.authenticate('github')` which redirects them to GitHub.
3. The page the user lands on, on GitHub, allows them to login if they aren't already. It then asks them to approve access to their profile from our app.
4. The user is then returned to our app at a specific callback url with their profile if they are approved.
5. They are now authenticated and your app should check if it is a returning profile, or save it in your database if it is not.

Strategies with OAuth require you to have at least a *Client ID* and a *Client Secret* which is a way for them to verify who the authentication request is coming from and if it is valid. These are obtained from the site you are trying to implement authentication with, such as GitHub, and are unique to your app- **THEY ARE NOT TO BE SHARED** and should never be uploaded to a public repository or written directly in your code. A common practice is to put them in your *.env* file and reference them like: `process.env.GITHUB_CLIENT_ID` . For this challenge we're going to use the GitHub strategy. Obtaining your *Client ID and Secret from GitHub is done in your account profile settings under 'developer settings', then 'OAuth applications'. Click 'Register a new application', name your app, paste in the url to your glitch homepage (**Not the project code's url**), and lastly for the callback url, paste in the same url as the homepage but with '/auth/github/callback' added on. This is where users will be redirected to for us to handle after authenticating on GitHub. Save the returned information as 'GITHUB_CLIENT_ID' and 'GITHUB_CLIENT_SECRET' in your .env file. On your remixed project, create 2 routes accepting GET requests: /auth/github and /auth/github/callback. The first should only call passport to authenticate 'github' and the second should call passport to authenticate 'github' with a failure redirect to '/' and then if that is successful redirect to '/profile' (similar to our last project). An example of how '/auth/github/callback' should look is similar to how we handled a normal login in our last project:*

```
app.route('/login')
  .post(passport.authenticate('local', { failureRedirect: '/' }), (req,res) => {
    res.redirect('/profile');
  });
```

Submit your page when you think you've got it right. If you're running into errors, you can check out the project up to this point here.

Instructions

Challenge Seed

Solution

```
// solution required
```

15. Implementation of Social Authentication II

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. The last part of setting up your GitHub authentication is to create the strategy itself. For this, you will need to add the dependency of 'passport-github' to your project and require it as GithubStrategy like `const GitHubStrategy = require('passport-`

*`github').Strategy;` . To set up the GitHub strategy, you have to tell **passport** to **use** an instantiated **GitHubStrategy**, which accepts 2 arguments: An object (containing clientID, clientSecret, and callbackURL) and a function to be called when a user is successfully authenticated which we will determine if the user is new and what fields to save initially in the user's database object. This is common across many strategies but some may require more information as outlined in that specific strategy's github README; for example, Google requires a scope as well which determines what kind of information your request is asking returned and asks the user to approve such access. The current strategy we are implementing has its usage outlined here, but we're going through it all right here on freeCodeCamp! Here's how your new strategy should look at this point:*

```
passport.use(new GitHubStrategy({
    clientID: process.env.GITHUB_CLIENT_ID,
    clientSecret: process.env.GITHUB_CLIENT_SECRET,
    callbackURL: /*INSERT CALLBACK URL ENTERED INTO GITHUB HERE*/
  },
  function(accessToken, refreshToken, profile, cb) {
      console.log(profile);
      //Database logic here with callback containing our user object
  }
));
```

Your authentication won't be successful yet, and actually throw an error, without the database logic and callback, but it should log to your console your GitHub profile if you try it! Submit your page when you think you've got it right.

Instructions

Challenge Seed

Solution

```
// solution required
```

16. Implementation of Social Authentication III

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. The final part of the strategy is handling the profile returned from GitHub. We need to load the user's database object if it exists, or create one if it doesn't, and populate the fields from the profile, then return the user's object. GitHub supplies us a unique id within each profile which we can use to search with to serialize the user with (already implemented). Below is an example implementation you can use in your project- it goes within the function that is the second argument for the new strategy, right below the `console.log(profile);` currently is:

```
db.collection('socialusers').findAndModify(
    {id: profile.id},
    {},
    {$setOnInsert:{
        id: profile.id,
        name: profile.displayName || 'John Doe',
        photo: profile.photos[0].value || '',
        email: profile.emails[0].value || 'No public email',
        created_on: new Date(),
        provider: profile.provider || ''
    },$set:{
        last_login: new Date()
    },$inc:{
        login_count: 1
    }},
    {upsert:true, new: true},
    (err, doc) => {
        return cb(null, doc.value);
    }
);
```

With a findAndModify, it allows you to search for an object and update it, as well as insert the object if it doesn't exist and receive the new object back each time in our callback function. In this example, we always set the last_login as now, we always increment the login_count by 1, and only when we insert a new object(new user) do we populate the majority of the fields. Something to notice also is the use of default values. Sometimes a profile returned won't have all the information filled out or it will have been chosen by the user to remain private; so in this case we have to handle it to prevent an error. You should be able to login to your app now- try it! Submit your page when you think you've got it right. If you're running into errors, you can check out an example of this mini-project's finished code here.

Instructions

Challenge Seed

Solution

```
// solution required
```

17. Set up the Environment

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. Add Socket.IO as a dependency and require/instantiate it in your server defined as 'io' with the http server as an argument. `const io = require('socket.io')(http);` *The first thing needing to be handled is listening for a new connection from the client. The on keyword does just that- listen for a specific event. It requires 2 arguments: a string containing the title of the event thats emitted, and a function with which the data is passed though. In the case of our connection listener, we use socket to define the data in the second argument. A socket is an individual client who is connected. For listening for connections on our server, add the following between the comments in your project:*

```
io.on('connection', socket => {
  console.log('A user has connected');
});
```

Now for the client to connect, you just need to add the following to your client.js which is loaded by the page after you've authenticated:

```
/*global io*/
var socket = io();
```

The comment suppresses the error you would normally see since 'io' is not defined in the file. We've already added a reliable CDN to the Socket.IO library on the page in chat.pug. Now try loading up your app and authenticate and you should see in your server console 'A user has connected'! ***Note***
`io()` *works only when connecting to a socket hosted on the same url/server. For connecting to an external socket hosted elsewhere, you would use* `io.connect('URL');`*. Submit your page when you think you've got it right.*

Instructions

Challenge Seed

Solution

```
// solution required
```

18. Communicate by Emitting

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. Emit is the most common way of communicating you will use. When you emit something from the server to 'io', you send an event's name and data to all the connected sockets. A good example of this concept would be emitting the current count of connected users each time a new user connects!

Start by adding a variable to keep track of the users just before where you are currently listening for connections. `var currentUsers = 0;` *Now when someone connects you should increment the count before emitting the count so you will want to add the incrementer within the connection listener.* `++currentUsers;` *Finally after incrementing the count, you should emit the event(still within the connection listener). The event should be named 'user count' and the data should just be the 'currentUsers'.* `io.emit('user count', currentUsers);`

Now you can implement a way for your client to listen for this event! Similarly to listening for a connection on the server you will use the on keyword.

```
socket.on('user count', function(data){
  console.log(data);
});
```

Now try loading up your app and authenticate and you should see in your client console '1' representing the current user count! Try loading more clients up and authenticating to see the number go up. Submit your page when you think you've got it right.

Instructions

Challenge Seed

Solution

```
// solution required
```

19. Handle a Disconnect

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. You may notice that up to now you have only been increasing the user count. Handling a user disconnecting is just as easy as handling the initial connect except the difference is you have to listen for it on each socket versus on the whole server.

To do this, add in to your existing connect listener a listener that listens for 'disconnect' on the socket with no data passed through. You can test this functionality by just logging to the console a user has disconnected. `socket.on('disconnect', () => { /*anything you want to do on disconnect*/ });` *To make sure clients continuously have the updated count of current users, you should decrease the currentUsers by 1 when the disconnect happens then emit the 'user count' event with the updated count!* **Note**
Just like 'disconnect', all other events that a socket can emit to the server should be handled within the connecting listener where we have 'socket' defined. Submit your page when you think you've got it right.

Instructions

Challenge Seed

Solution

```
// solution required
```

20. Authentication with Socket.IO

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. Currently, you cannot determine who is connected to your web socket. While 'req.user' containers the user object, thats only when your user interacts with the web server and with web sockets you have no req (request) and therefor no user data. One way to solve the problem of knowing who is connected to your web socket is by parsing and decoding the cookie that contains the passport session then deserializing it to obtain the user object. Luckily, there is a package on NPM just for this that turns a once complex task into something simple!

Add 'passport.socketio' as a dependency and require it as 'passportSocketIo'. Now we just have to tell Socket.IO to use it and set the options. Be sure this is added before the existing socket code and not in the existing connection listener. For your server it should look as follows:

```
io.use(passportSocketIo.authorize({
  cookieParser: cookieParser,
  key:          'express.sid',
  secret:       process.env.SESSION_SECRET,
  store:        sessionStore
}));
```

You can also optionally pass 'success' and 'fail' with a function that will be called after the authentication process completes when a client trys to connect. The user object is now accessible on your socket object as `socket.request.user`. For example, now you can add the following: `console.log('user ' + socket.request.user.name + ' connected');` and it will log to the server console who has connected! Submit your page when you think you've got it right. If you're running into errors, you can check out the project up to this point here.

Instructions

Challenge Seed

Solution

```
// solution required
```

21. Announce New Users

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. Many chat rooms are able to annouce when a user connects or disconnects and then display that to all of the connected users in the chat. Seeing as though you already are emitting an event on connect and disconnect, you will just have to modify this event to support such feature. The most logical way of doing so is sending 3 pieces of data with the event: name of the user connected/disconnected, the current user count, and if that name connected or disconnected.

Change the event name to 'user' and as the data pass an object along containing fields 'name', 'currentUsers', and boolean 'connected' (to be true if connection, or false for disconnection of the user sent). Be sure to make the change to both points we had the 'user count' event and set the disconnect one to sent false for field 'connected' instead of true like the event emitted on connect. `io.emit('user', {name: socket.request.user.name, currentUsers, connected: true});` *Now your client will have all the necessary information to correctly display the current user count and annouce when a user connects or disconnects! To handle this event on the client side we should listen for 'user' and then update the current user count by using jQuery to change the text of* `#num-users` *to '{NUMBER} users online', as well as append a* `<li>` *to the unordered list with id 'messages' with '{NAME} has {joined/left} the chat.'. An implementation of this could look like the following:*

```
socket.on('user', function(data){
  $('#num-users').text(data.currentUsers+' users online');
  var message = data.name;
  if(data.connected) {
    message += ' has joined the chat.';
  } else {
    message += ' has left the chat.';
  }
  $('#messages').append($('<li>').html('<b>'+ message +'<\/b>'));
});
```

Submit your page when you think you've got it right.

Instructions

Challenge Seed

Solution

```
// solution required
```

22. Send and Display Chat Messages

Description

As a reminder, this project is being built upon the following starter project on Glitch, or cloned from GitHub. It's time you start allowing clients to send a chat message to the server to emit to all the clients! Already in your client.js file you should see there is already a block of code handling when the messgae form is submitted! `( $('form').submit(function(){ /*logic*/ });)`

Within the code you're handling the form submit you should emit an event after you define 'messageToSend' but before you clear the text box `#m` *. The event should be named 'chat message' and the data should just be 'messageToSend'.* `socket.emit('chat message', messageToSend);` *Now on your server you should be listening to the socket for the event 'chat message' with the data being named 'message'. Once the event is received it should then emit the event 'chat message' to all sockets* `io.emit` *with the data being an object containing 'name' and 'message'. On your client now again, you should now listen for event 'chat message' and when received, append a list item to* `#messages` *with the name a colon and the message! At this point the chat should be fully functional and sending messages across all clients! Submit your page when you think you've got it right. If you're running into errors, you can check out the project up to this point here for the server and here for the client.*

Instructions

Challenge Seed

Solution

```
// solution required
```

Information Security and Quality Assurance Projects

1. Metric-Imperial Converter

Description

Build a full stack JavaScript app that is functionally similar to this: https://hard-twilight.glitch.me/. Working on this project will involve you writing your code on Glitch on our starter project. After completing this project you can copy your public glitch url (to the homepage of your app) into this screen to test it! Optionally you may choose to write your project on another platform but it must be publicly visible for our testing. Start this project on Glitch using this link or clone this repository on GitHub! If you use Glitch, remember to save the link to your project somewhere safe!

Instructions

Challenge Seed

Solution

```
// solution required
```

2. Issue Tracker

Description

Build a full stack JavaScript app that is functionally similar to this: https://protective-garage.glitch.me/. Working on this project will involve you writing your code on Glitch on our starter project. After completing this project you can copy your public glitch url (to the homepage of your app) into this screen to test it! Optionally you may choose to write your project on another platform but it must be publicly visible for our testing. Start this project on Glitch using this link or clone this repository on GitHub! If you use Glitch, remember to save the link to your project somewhere safe!

Instructions

Challenge Seed

Solution

```
// solution required
```

3. Personal Library

Description

Build a full stack JavaScript app that is functionally similar to this: https://spark-cathedral.glitch.me/. Working on this project will involve you writing your code on Glitch on our starter project. After completing this project you can copy your public

glitch url (to the homepage of your app) into this screen to test it! Optionally you may choose to write your project on another platform but must be publicly visible for our testing. Start this project on Glitch using this link or clone this repository on GitHub! If you use Glitch, remember to save the link to your project somewhere safe!

Instructions

Challenge Seed

Solution

```
// solution required
```

4. Stock Price Checker

Description

Build a full stack JavaScript app that is functionally similar to this: https://giant-chronometer.glitch.me/. Working on this project will involve you writing your code on Glitch on our starter project. After completing this project you can copy your public glitch url (to the homepage of your app) into this screen to test it! Optionally you may choose to write your project on another platform but must be publicly visible for our testing. Start this project on Glitch using this link or clone this repository on GitHub! If you use Glitch, remember to save the link to your project somewhere safe!

Instructions

Challenge Seed

Solution

```
// solution required
```

5. Anonymous Message Board

Description

Build a full stack JavaScript app that is functionally similar to this: https://horn-celery.glitch.me/. Working on this project will involve you writing your code on Glitch on our starter project. After completing this project you can copy your public glitch url (to the homepage of your app) into this screen to test it! Optionally you may choose to write your project on another platform but it must be publicly visible for our testing. Start this project on Glitch using this link or clone this repository on GitHub! If you use Glitch, remember to save the link to your project somewhere safe!

Instructions

Challenge Seed

Solution

```
// solution required
```