Student Guide to Coding

Information Security and Quality Assurance Certification (300 hours)

Introduction to Information Security with HelmetJS

HelmetJS is a type of middleware for Express-based applications that automatically sets HTTP headers to prevent sensitive information from unintentionally being passed between the server and client. While HelmetJS does not account for all situations, it does include support for common ones like Content Security Policy, XSS Filtering, and HTTP Strict Transport Security, among others. HelmetJS can be installed on an Express project from npm, after which each layer of protection can be configured to best fit the project.

Install and Require Helmet

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.

Hide Potentially Dangerous Information Using helmet.hidePowerBy( )

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' }))*

Mitigate the Risk of Clickjacking with helmet.frameguard( )

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 by executing your page in a malicious context, by means 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.

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

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.

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

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.

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

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.

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

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: timeInSeconds, 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.

Disable DNS Prefetching with helmet.dnsPrefetchControl( )

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.

Disable Client-Side Caching with helmet.noCache( )

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.

Set a Content Security Policy with helmet.contentSecurityPolicy( )

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 types 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 browsers.

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.

Configure Helmet Using the 'parent' helmet( ) Middleware

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 purposes and for ease of testing. Using the 'parent' helmet() middleware is easy to implement in a real project.

Understand BCrypt Hashes

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 an algorithm and returning 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.uUUtcbqloU0yvzavOm 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 take multiple days to complete a hash. A cost of 12 is considered very secure at this time. The last portion of your hash $ZyprE5MRw2Q3WpNOGZWGbeG7ADUre1Q8QO.uUUtcbqloU0yvzavOm, looks like one large string of numbers, periods, and letters but it is actually two separate pieces of information. The first 22 characters are 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.

Hash and Compare Passwords Asynchronously

As hashing is designed to be computationally intensive, it is recommended to do so asynchronously on your server 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 function. You should see in the console a hash ' 'true' is printed! If you change 'myPlaintextPassword' within the compare function to 'someOtherPlaintextPassword' then it should say false.

```
bcrypt.hash('passw0rd!', 13, (err, hash) => {
  console.log(hash);
  //$2a$12$Y.PHPE15wR25qrrtgGkiYe2sXo98cjuMCG1YwSI5rJW1DSJp0gEYS
  bcrypt.compare('passw0rd!', hash, (err, res) => {
    console.log(res); //true
  });
});
```

Hash and Compare Passwords Synchronously

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 won't 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);
```

Basic Information Security with HelmetJS Review Questions

1. What is HelmetJS?

2. What method allows you to hide the X-Powered-By header?

3. After hiding it how can you set it to something else to throw people off? Set it to PHP 4.2.0.

4. What will prevent IE users from executing downloads in the trusted site's context?

5. What does HSTS stand for?

6. What does HTTP stand for?

7. What method would you use to mitigate the risk of clickjacking?

8. This middleware sets the X-Frame-Options header. It restricts who can put your site in a frame. What are the three modes of this?

9. How can you prevent the injection of anything unintended into your page?

10. What middleware sets the X-Content-Type-Options header to nosniff?

Introduction to Quality Assurance with Chai

As your programs become more complex, you need to test them often to make sure any new code you add doesn't break the program's original functionality. Chai is a JavaScript testing library that helps you check that your program still behaves the way you expect it to after you make changes. Using Chai, you can write tests that describe your program's requirements and see if your program meets them.

Learn How JavaScript Assertions Work

Use `assert.isNull()` or `assert.isNotNull()` to make the tests pass.

Test if a Variable of Function is Defined

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.

Use Assert.isOK and Assert.isNotOK

`isOk()` will test for a truthy value and `isNotOk()` will test for a falsy value. Truthy reference Falsy reference

Test for Truthiness

`isTrue()` will test for the boolean value true and `isNotTrue()` will pass when given anything but the boolean value of true.

```
assert.isTrue(true, 'this will pass with the boolean value true');
assert.isTrue('true', 'this will NOT pass with the string value 'true');
assert.isTrue(1, 'this will NOT pass with the number value 1');
```

`isFalse()` and `isNotFalse()` also exist and behave similarly to their true counterparts except they look for the boolean value of false.

Use the Double Equals to Assert Equality

`equal()` compares objects using `==`.

Use the Triple Equals to Assert Strict Equality

`strictEqual()` compares objects using `===`.

Assert Deep Equality with .deepEqual and .notDeepEqual

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

`deepEqual()` asserts that two objects are deep equal.

Compare the Properties of Two Elements

Use `assert.isAbove()`(i.e. greater) or `assert.isAtMost()` (i.e. less than or equal) to make the tests pass.

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

Use assert.isBelow() (i.e. less than) or assert.isAtLeast() (i.e. greater than or equal) to make the tests pass.

Test if a Value Falls within a Specific Range

.approximately(actual, expected, delta, [message]) Asserts that the actual is equal expected, to within a +/- delta range.

Use assert.approximately() to make the tests pass.
Choose the minimum range (3rd parameter) to make the test always pass. It should be less than 1.

Test if a Value is an Array

Use assert.isArray() or assert.isNotArray() to make the tests pass.

Test if an Array Contains an Item

Use assert.include() or assert.notInclude() to make the tests pass.

Test if a Value is a String

isString or isNotString asserts that the actual value is a string.

Test if a String Contains a Substring

include() and notInclude() work for strings too! include() asserts that the actual string contains the expected substring.

Use Regular Expressions to Test a String

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.

Test if an Object has a Property

#property asserts that the actual object has a given property.

Test if a Value is of a Specific Data Structure Type

#typeOf asserts that value's type is the given string, as determined by Object.prototype.toString.

Test if an Object is an Instance of a Constructor

#instanceOf asserts that an object is an instance of a constructor.

Use assert.instanceOf() or assert.notInstanceOf() to make the tests pass.

Run Functional Test on API Endpoints using Chai-HTTP

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 responds with 'hello Guest'.

Run Functional Test on API Endpoints using Chai-HTTP II

Replace assert.fail(). Test the status and the text.response. Make the test pass.

Send your name in the query appending ?name=<your_name>, the endpoint, with, responds with 'hello <your_name>'.

Run Functional Test on API Response using Chai-HTTP III - PUT method

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.

Run Functional Test on API Response using Chai-HTTP IV - PUT method

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.

Run Functional Test using a Headless Browser

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 kinds 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.

Basic Quality Assurance and Testing with Chai

1. What is Chai?

2. Using assertions, how can you test if a value is an array?

3. Using assertions, how can you test if an array contains an item?

4. _____ asserts that the actual value matches the second argument regular expression.

5. _____asserts that an object is an instance of a constructor.

6. What is a headless browser?

7. Using assertions, how can you test for truthiness?

8. How can you test if a Value Falls within a Specific Range?

9. How do you compare the Properties of Two Elements?

10. What is Zombie.JS?

Introduction to Advanced Node and Express

Authentication is the process or action of verifying the identity of a user or process. Up to this point you have not been able to create an app utilizing this key concept.

The most common and easiest to use authentication middleware for Node.js is Passport. It is easy to learn, light-weight, and extremely flexible allowing for many strategies, which we will talk about in later challenges. In addition to authentication, we will also look at template engines which allow for use of Pug and web sockets which, in turn, allow for real time communication between all your clients and your server.

Set up a Template Engine

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.

Use a Template Engine's Powers

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 able to use a variable by referencing the variable name as #{variable_name} inline with other text on an element or by using an equal sign on the element without a space such as p=variable_name which assigns the variable's value to the p element's text.

We strongly recommend looking at the syntax and structure of Pug here on GitHub's 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 along from our server, you will need to add an object as a second argument to your res.render with the variables and their values. 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.

Set up Passport

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 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,
}));
```

Additionally, 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.

Serialization of a user Object

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 regard 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 who 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.

Basic Advanced Node and Express Review Questions

1. To set up Pug for use in your project, you will need to add it as a _____.

2. 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'. How do you do that?

3. One of the greatest features of using a template engine is being able to?

4. The most common and easiest to use authentication middleware for Node.js is?

5. To set up Passport for use in your project, you will need to add it as a dependency first in your package.json. How?

6. Add Express-session as a dependency now as well.

7. What is used to compute the hash used to encrypt your cookie?

8. What file does SESSION_SECRET go into?

9. To _____ means to convert its contents into a small key essentially that can then be deserialized into the original object.

10. To set this up properly, we need to have a serialize function and a deserialize function. In passport we create these with?