Outline:

2 hours

Mixture of multiple choice (15-20) and VSCode (3-5) problems.

Coding problems will have specs to run (npm test) and check your work against

Part of the assessment will include writing tests using either assert or chai.expect (our tests will test your tests... meta)

Standard assessment procedures

You will be in an individual breakout room

Use a single monitor and share your screen

Only have open those resources needed to complete the assessment:

Zoom

VSCode

Browser with AAO and Progress Tracker (to ask questions)

Approved Resources for this assessment:

MDN: https://developer.mozilla.org/en-US/docs/Web/JavaScript

Node.js Documentation: https://nodejs.org/docs/latest-v12.x/api/assert.html

Chai Documentation: https://www.chaijs.com/api/bdd/

2 hours total

-------------Multiple Choice------------------(No extended response)---

---

~ 45 min

Mostly HTTP

2-3 Broad concept promise or testing questions

---

1) Matching HTTP verbs (GET, PUT, PATCH, POST, DELETE) to their common uses.

2) Match the header fields of HTTP with a bank of definitions.

3) Match common HTTP status codes (201, 301, 302, 400, 401, 402, 403, 404, 500) to their meanings.

4) HTTP Status Code Ranges(100 - 199: Informational; 200 - 299: Successful; 300 - 399: Redirection; 400 - 499: Client Error; 500 - 599: Server Error)

5) What does the promise in code above return? / what happens ?

6) Match error type to definition (TypeError, ReferenceError, SyntaxError)

7) Predict if and what type of error certain code will throw

8) Something to do with changing catch onto the end of a series of chained then statements

9) Know beforeEach and afterEach hooks do in a describe block

10) know content type is and how it is used in HTML body.

11) Know send HTTP request to google manually: (nc -v google.com 80   .... GET / HTTP/1.1) capitalize!

12) What kind of HTTP requests have a body? (Any HTTP Request;(when the data doesn't fit into the header &|| is too complex for the URI we can use the body)

()It has a body when HTTP verb is POST, PUT, Patch

13) Know that HTTP stands for Hypertext Transfer Protocol

14) Three Stages of TDD (red, green , refactor) and what they mean

15) What are the states that a promise can be in (pending, fulfilled (rejected), rejected)

16) Parts of an HTTP Request(3 parts: Request line & HTTP Verbs-method, URI, HTTP - Version; Headers-metadata as key value pairs; Body-update something on the backend)

17) What is a promise? describe it in terms of synchronous and async (js as a language is synchronous/blocking).. I will eventually get some response from this promise

--Coding--

~1.25 hrs

---

18) Write a promise (3-4 lines of code) (__ don't need a .then().catch or console.error() __) ...or anything you're not explicitely asked to do

19) Async await (3-4 lines of code)

20) Unit Test (you are given 3 functions any you have to edit unit tests (remove a line) and write your expect/assert

--> don't run tests with mocha! (run npm test).... in your package.json mocha will be the specified test (if you run mocha you will be testing your tests not the js functions)

---->pull down from git repo and run npm install....

------> make sure to call in variables you need for test

--------> don't need chai spy or chai spy on

----------> need to require (whatever was exported in the not-test file) and your assertion library

---

Answers:

1) Matching HTTP verbs (GET, PUT, PATCH, POST, DELETE) to their common uses.

HTTP Verbs are a simply way of declaring our intention to the server.

- `GET` : Used for direct requests.
- `POST`: Used for creating new resources on the server.
- `PUT`: Used to updated a resource on the server.
- `PATCH` : Similar to PUT, but do not require the whole resource to perform the update.
- `DELETE` : Used to destroy resources on the server.

2) Match the header fields of HTTP with a bank of definitions.

Headers

- Key-Value pairs that come after the request line - they appear on sep. lines and define metadata needed to process the request.
- Some common headers:

`Host` :	Root path for our URI.
`User-Agent` :	Displays information about which browser the request originated from.
`Referer` :	Defines the URL you're coming from.
`Accept` :	Indicates what the client will receive.
`Content-` :	Define Details about the body of the request.

3.) Match common HTTP status codes (201, 301, 302, 400, 401, 402, 403, 404, 500) to their meanings.

HTTP Status Code Ranges(100 - 199: Informational; 200 - 299: Successful; 300 - 399: Redirection; 400 - 499: Client Error; 500 - 599: Server Error)

Status

- First line of an HTTP response - gives us a high level overview of the server's intentions. (`status line`)
- `HTTP/1.1 200 OK`
- `HTTP status codes` : numeric way of representing a server's response.
    - Follow the structure: x: xxx - xxx;
    - **`Status codes 100 - 199: Informational`**
        - Allow the clinet to know that a req. was received, and provides extra info from the server.
        -
    - **`Status codes 200 - 299: Successful`**
        - Indicate that the request has succeeded and the server is handling it.
        - Common Examples:

200 OK (req received and fulfilled) 201 Created (received and new record was created)

- **`Status codes 300 - 399: Redirection`**
    - Let the client know if there has been a change.
    - Common Examples:

301 Moved Permanently (resource you requested is in a totally new location) & 302 Found (indicates a temporary move)

- **`Status codes 400 - 499: Client Error`**
    - Indicate problem with client's request.
    - Common Examples:

400 Bad Request (received, but could not understand) & 401 Unauthorized (resource exists but you're not allowed to see w/o authentication)

& 403 Forbidden (resource exists but you're not allowed to see it at all) & 404 Not Found (resource requested does not exist);

- **`Status codes 500 - 599: Server Error`**
    - Indicates request was formatted correctly, but the server couldn't do what you asked due to an internal problem.
    - Common Examples:

500 Internal Server Error (Server had trouble processing) & 504 Gateway Timeout (Server timeout);

5) What does the promise in code above return? / what happens ?

`Promise` : a commitment that sometime in the future, your code will get a value from some operation (like reading a file or getting JSON from a Website) or your code will get an error from that operation (like the file doesn't exist or the Web site is down).

6) Match error type to definition (TypeError, ReferenceError, SyntaxError)

1.

Q: **When is a JavaScript Error Object thrown**?

A: The Error object is how JavaScript deals with runtime errors - so at code runtime!

2.

Q: **How do you halt program execution with an instance of an error object in JavaScript?**

A: Using the keyword throw you can throw your own runtime errors that will stop program execution.

3.

Q: What type of block will allow you to run an erroring function then continue the execution of code after that function is run?

A: We can use try...catch blocks with functions that might throw an error to catch that error and continue code execution after that error was thrown

4.

Q: When kind of error is thrown when the JavaScript engine attempts to parse code that does not conform to the syntax of the JavaScript language?

A: A SyntaxError is thrown when there is an error in the syntax of the executed code.

5.

Q: What kind of error is thrown when a variable or parameter is not of a valid type?

A: A TypeError is thrown when an operation cannot be performed because the operand is a value of the wrong type.

6.

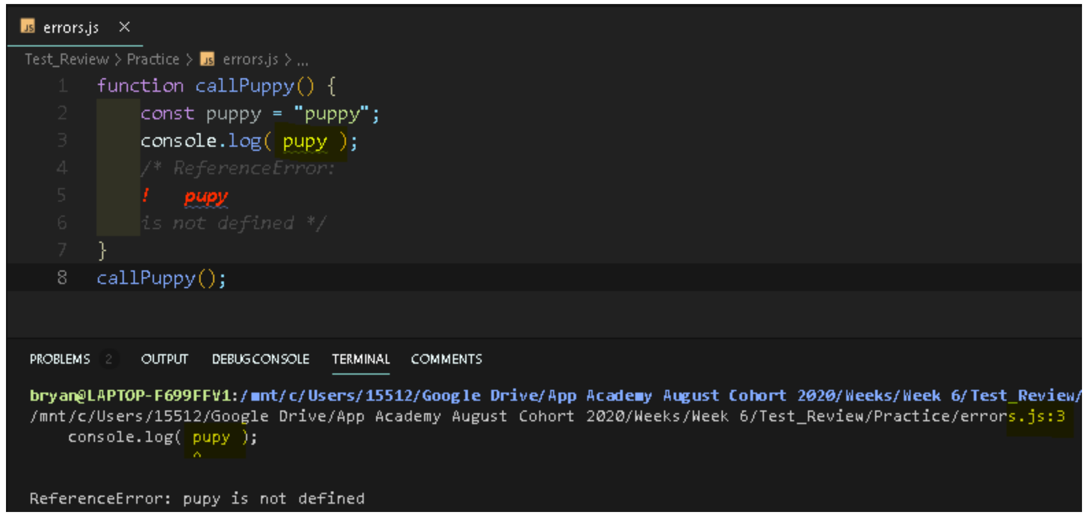
Q: What type of error is thrown when a non-existent variable is referenced?

A: The ReferenceError object represents an error when a non-existent variable is referenced.

7) Predict if and what type of error certain code will throw

7.

Q: What kind of error will be thrown when the below code is executed?



```
1  function callPuppy() {
2      const puppy = "puppy";
3      console.log( pupy );
4      /* ReferenceError:
5      !   pupy
6      is not defined */
7  }
8  callPuppy();
```

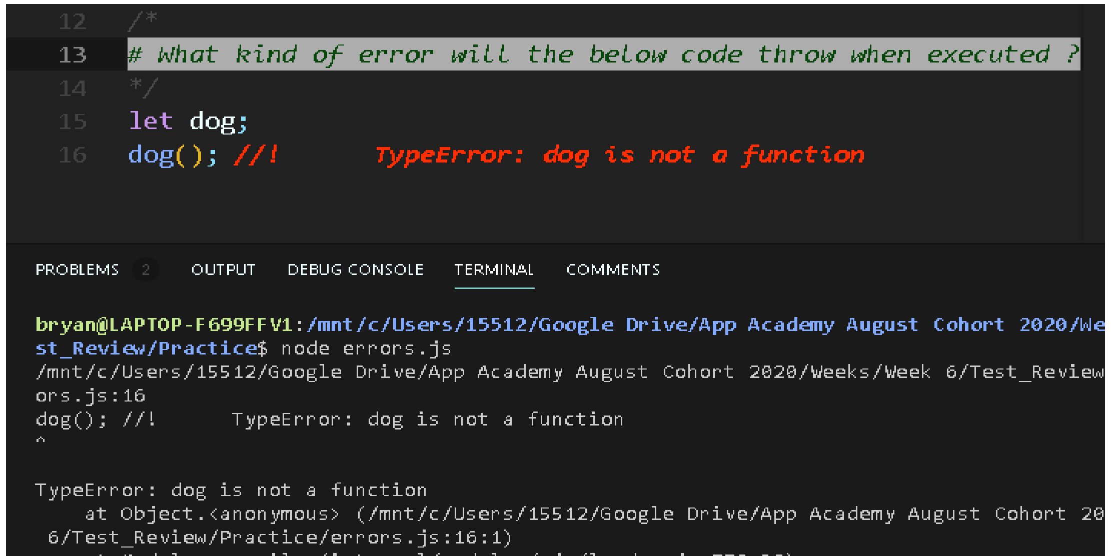
```
bryan@LAPTOP-F699FFV1:/mnt/c/Users/15512/Google Drive/App Academy August Cohort 2020/Weeks/Week 6/Test_Review/
/mnt/c/Users/15512/Google Drive/App Academy August Cohort 2020/Weeks/Week 6/Test_Review/Practice/errors.js:3
    console.log( pupy );
                 ^

ReferenceError: pupy is not defined
```

A: ReferenceError: pupy is not defined

8.

Q: What kind of error will the below code throw when executed?

A: TypeError: dog is not a function

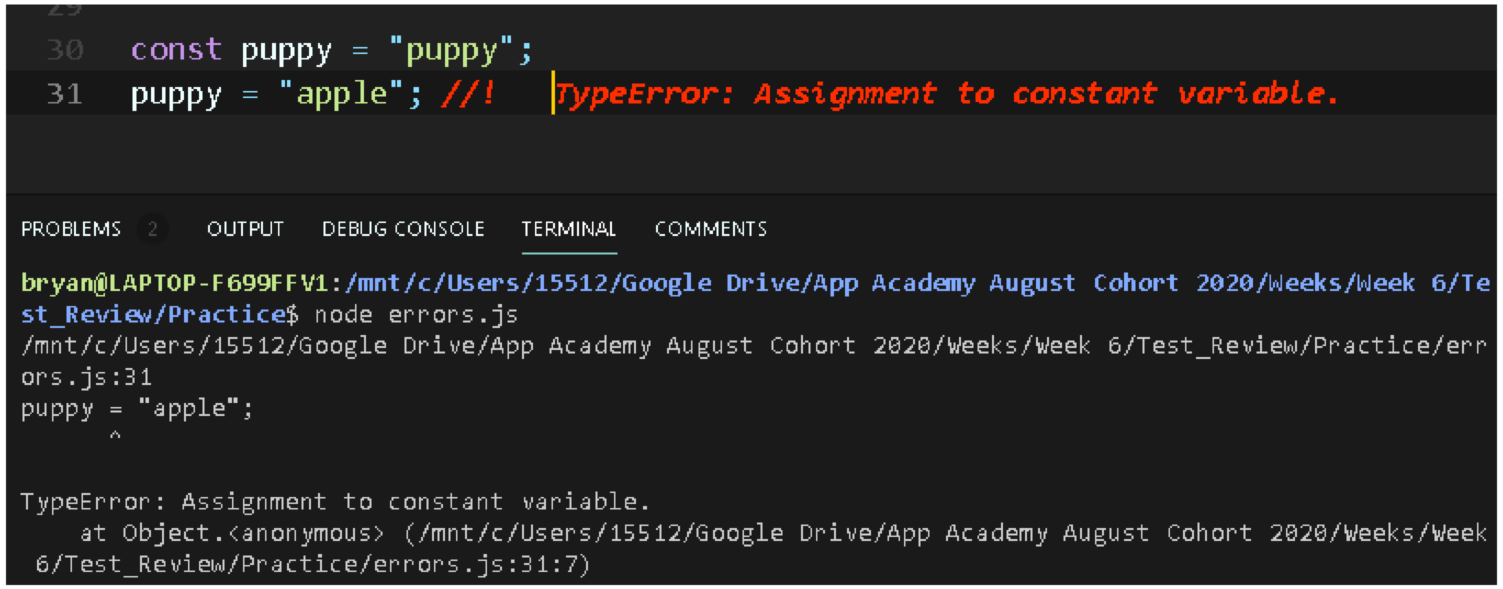


```
12  /*
13  # What kind of error will the below code throw when executed ?
14  */
15  let dog;
16  dog(); //!       TypeError: dog is not a function
```

```
bryan@LAPTOP-F699FFV1:/mnt/c/Users/15512/Google Drive/App Academy August Cohort 2020/We
st_Review/Practice$ node errors.js
/mnt/c/Users/15512/Google Drive/App Academy August Cohort 2020/Weeks/Week 6/Test_Review
ors.js:16
dog(); //!       TypeError: dog is not a function
^

TypeError: dog is not a function
    at Object.<anonymous> (/mnt/c/Users/15512/Google Drive/App Academy August Cohort 20
 6/Test_Review/Practice/errors.js:16:1)
```

9.

Q: What kind of error will the below code throw when executed?



```
30  const puppy = "puppy";
31  puppy = "apple"; //!    TypeError: Assignment to constant variable.
```

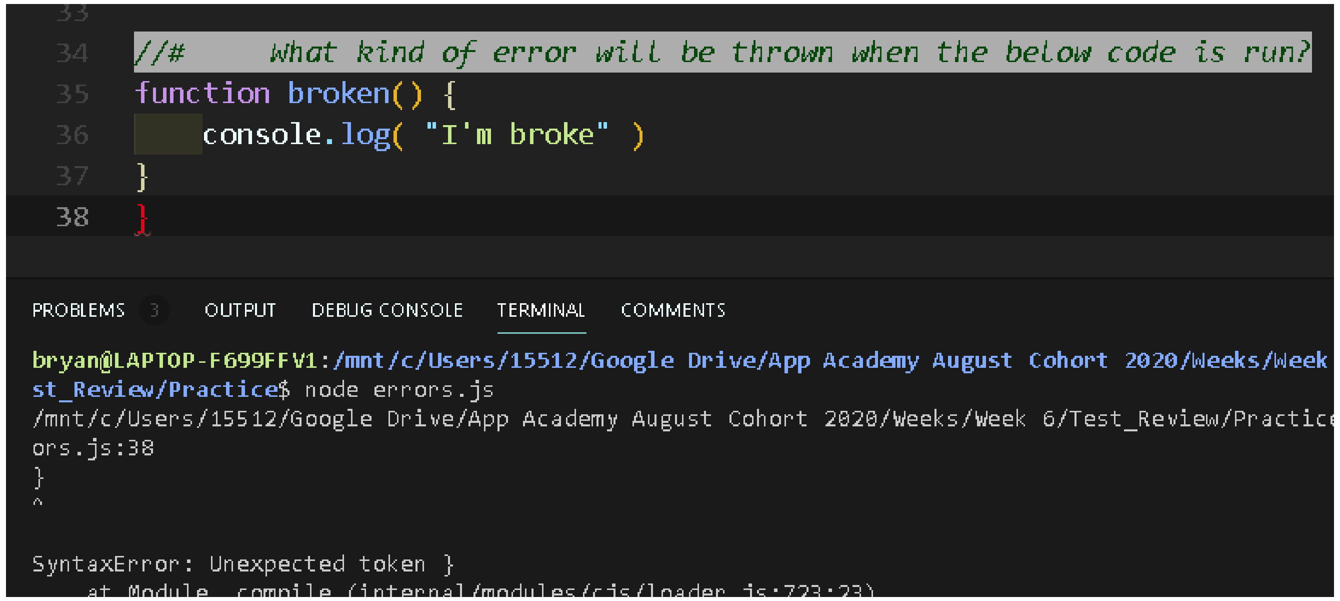
PROBLEMS 2   OUTPUT   DEBUG CONSOLE   TERMINAL   COMMENTS

```
bryan@LAPTOP-F699FFV1:/mnt/c/Users/15512/Google Drive/App Academy August Cohort 2020/Weeks/Week 6/Te
st_Review/Practice$ node errors.js
/mnt/c/Users/15512/Google Drive/App Academy August Cohort 2020/Weeks/Week 6/Test_Review/Practice/err
ors.js:31
puppy = "apple";
      ^

TypeError: Assignment to constant variable.
    at Object.<anonymous> (/mnt/c/Users/15512/Google Drive/App Academy August Cohort 2020/Weeks/Week
 6/Test_Review/Practice/errors.js:31:7)
```

A: TypeError: Assignment to constant variable.

10.

Q: What kind of error will be thrown when the below code is run?

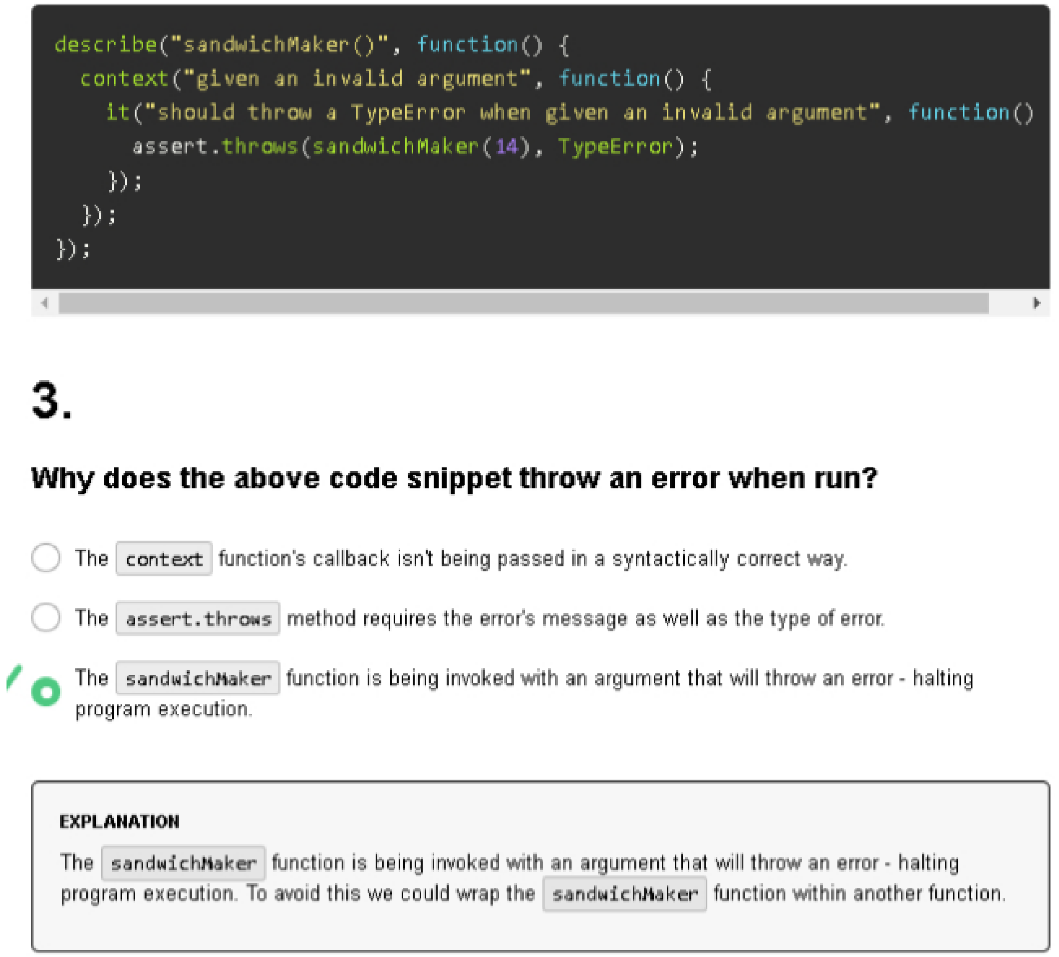


```
34  //#     What kind of error will be thrown when the below code is run?
35  function broken() {
36      console.log( "I'm broke" )
37  }
38  }
```

PROBLEMS 3   OUTPUT   DEBUG CONSOLE   TERMINAL   COMMENTS

```
bryan@LAPTOP-F699FFV1:/mnt/c/Users/15512/Google Drive/App Academy August Cohort 2020/Weeks/Week
st_Review/Practice$ node errors.js
/mnt/c/Users/15512/Google Drive/App Academy August Cohort 2020/Weeks/Week 6/Test_Review/Practice
ors.js:38
}
^

SyntaxError: Unexpected token }
    at Module._compile (internal/modules/cjs/loader.js:723:23)
```

A: SyntaxError: Unexpected token }



```
describe("sandwichMaker()", function() {
  context("given an invalid argument", function() {
    it("should throw a TypeError when given an invalid argument", function()
      assert.throws(sandwichMaker(14), TypeError);
    });
  });
});
```

3.

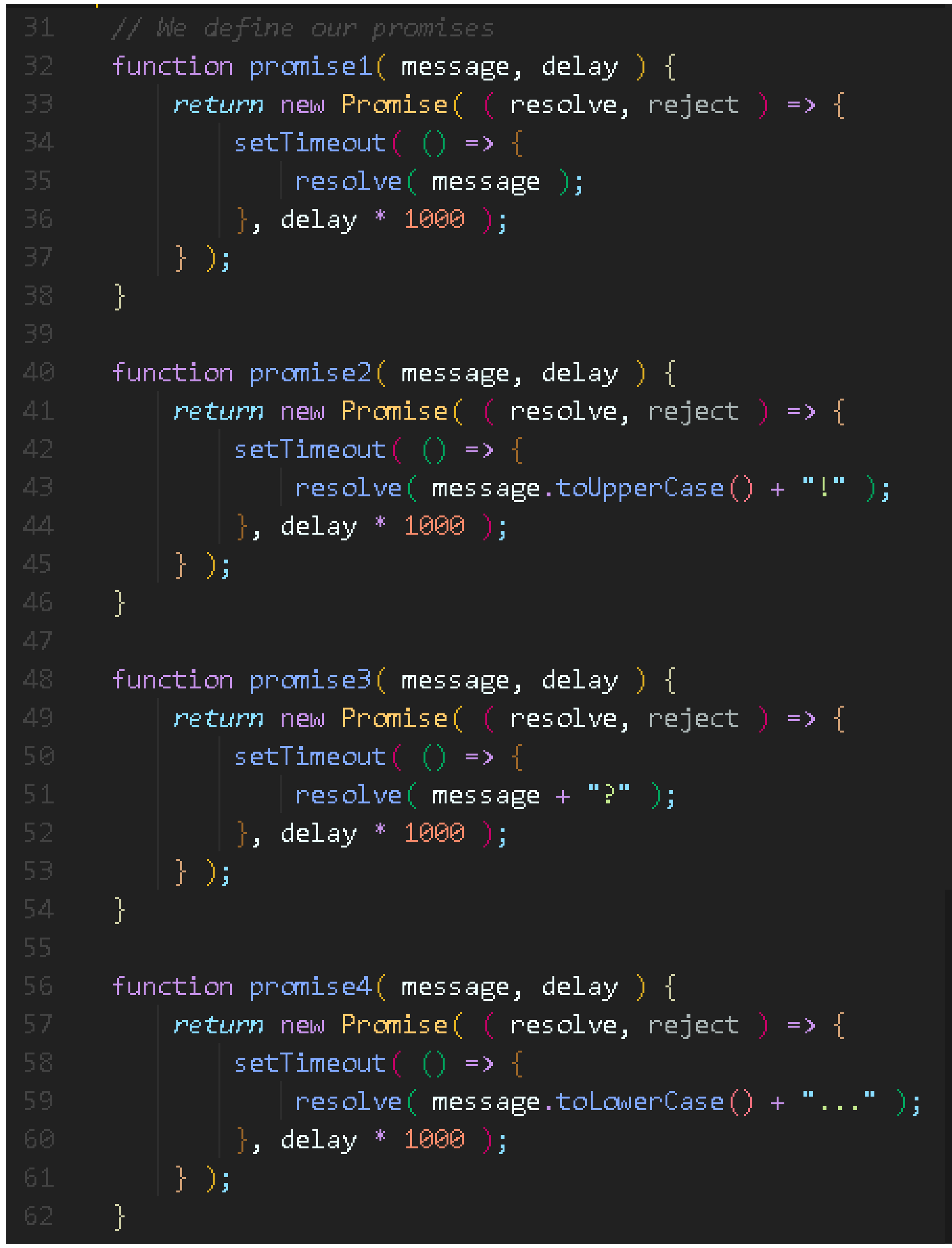
Why does the above code snippet throw an error when run?

- The `context` function's callback isn't being passed in a syntactically correct way.
- The `assert.throws` method requires the error's message as well as the type of error.
- [✓ The `sandwichMaker` function is being invoked with an argument that will throw an error - halting program execution.]

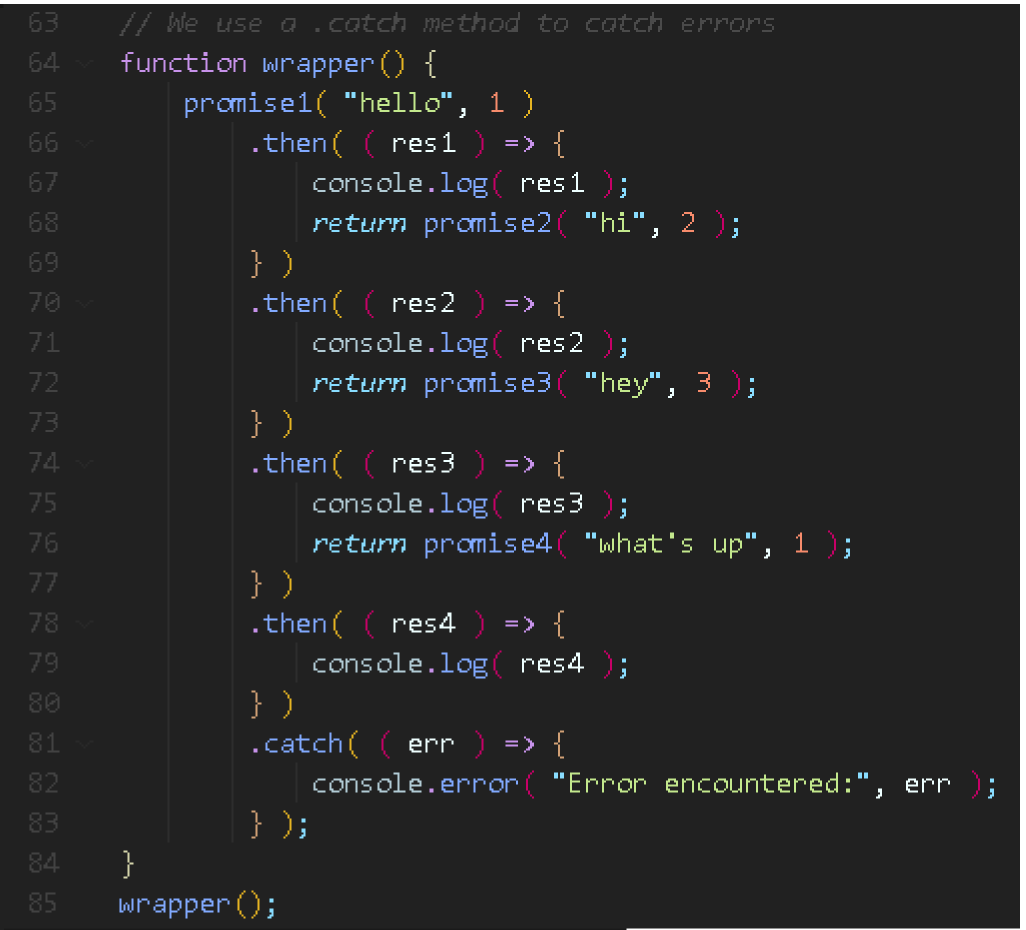
EXPLANATION

The `sandwichMaker` function is being invoked with an argument that will throw an error - halting program execution. To avoid this we could wrap the `sandwichMaker` function within another function.

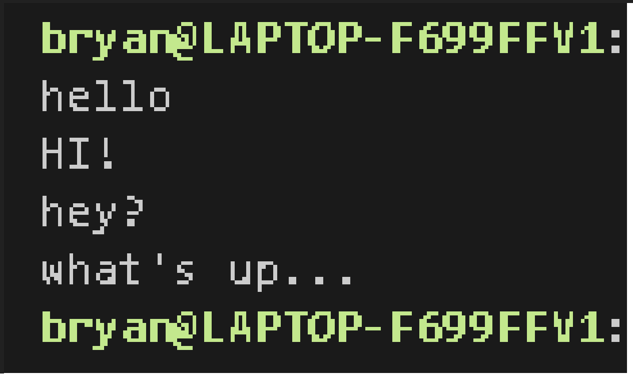
8) Something to do with changing catch onto the end of a series of chained then statements



```
// We define our promises
function promise1( message, delay ) {
    return new Promise( ( resolve, reject ) => {
        setTimeout( () => {
            resolve( message );
        }, delay * 1000 );
    } );
}

function promise2( message, delay ) {
    return new Promise( ( resolve, reject ) => {
        setTimeout( () => {
            resolve( message.toUpperCase() + "!" );
        }, delay * 1000 );
    } );
}

function promise3( message, delay ) {
    return new Promise( ( resolve, reject ) => {
        setTimeout( () => {
            resolve( message + "?" );
        }, delay * 1000 );
    } );
}

function promise4( message, delay ) {
    return new Promise( ( resolve, reject ) => {
        setTimeout( () => {
            resolve( message.toLowerCase() + "..." );
        }, delay * 1000 );
    } );
}
```

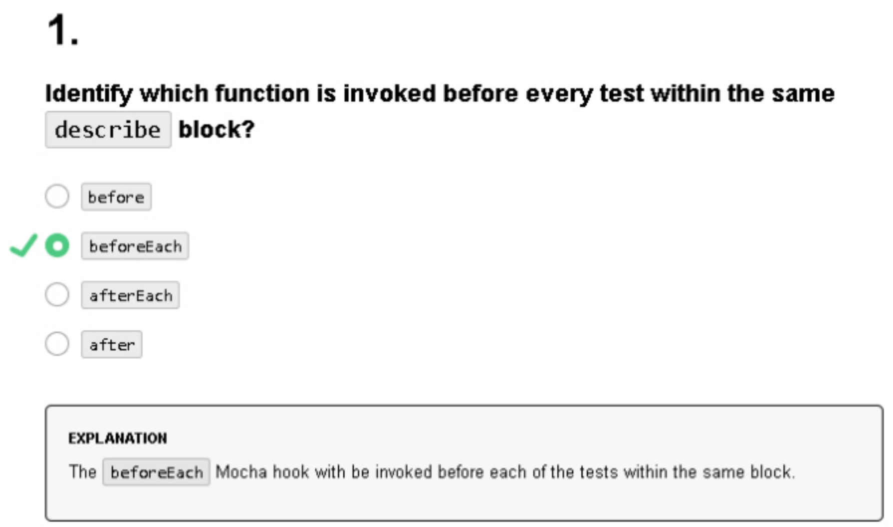


```
// We use a .catch method to catch errors
function wrapper() {
    promise1( "hello", 1 )
        .then( ( res1 ) => {
            console.log( res1 );
            return promise2( "hi", 2 );
        } )
        .then( ( res2 ) => {
            console.log( res2 );
            return promise3( "hey", 3 );
        } )
        .then( ( res3 ) => {
            console.log( res3 );
            return promise4( "what's up", 1 );
        } )
        .then( ( res4 ) => {
            console.log( res4 );
        } )
        .catch( ( err ) => {
            console.error( "Error encountered:", err );
        } );
}
wrapper();
```



```
bryan@LAPTOP-F699FFV1:
hello
HI!
hey?
what's up...
bryan@LAPTOP-F699FFV1:
```

9) Know beforeEach and afterEach hooks do in a describe block



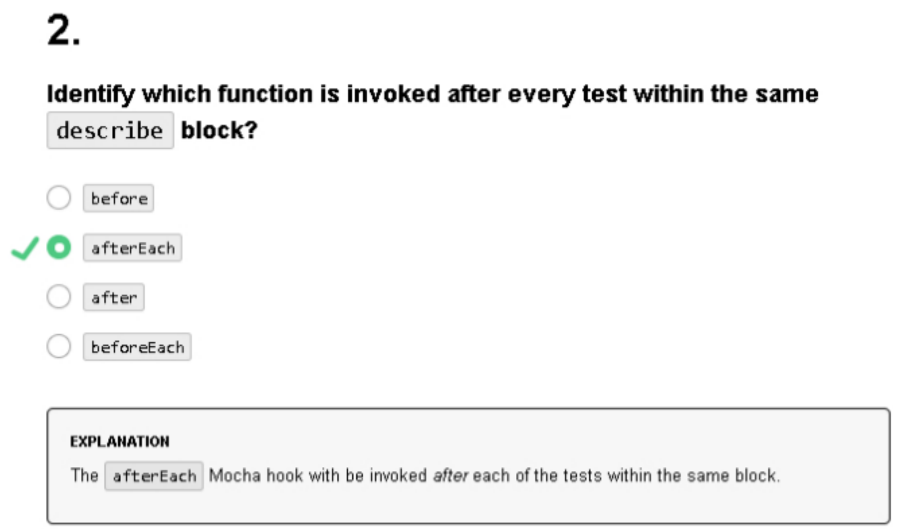
1.

Identify which function is invoked before every test within the same `describe` block?

- `before`
- [✓ `beforeEach`]
- `afterEach`
- `after`

EXPLANATION

The `beforeEach` Mocha hook with be invoked before each of the tests within the same block.



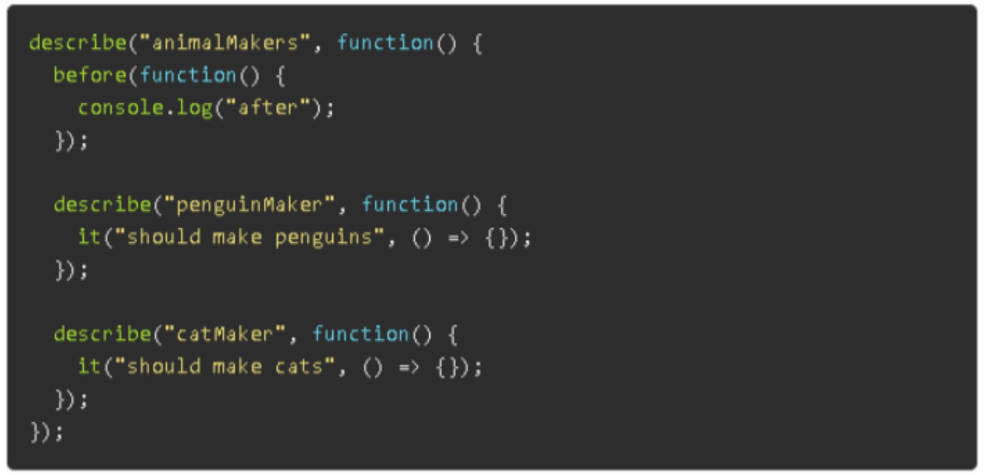
2.

Identify which function is invoked after every test within the same `describe` block?

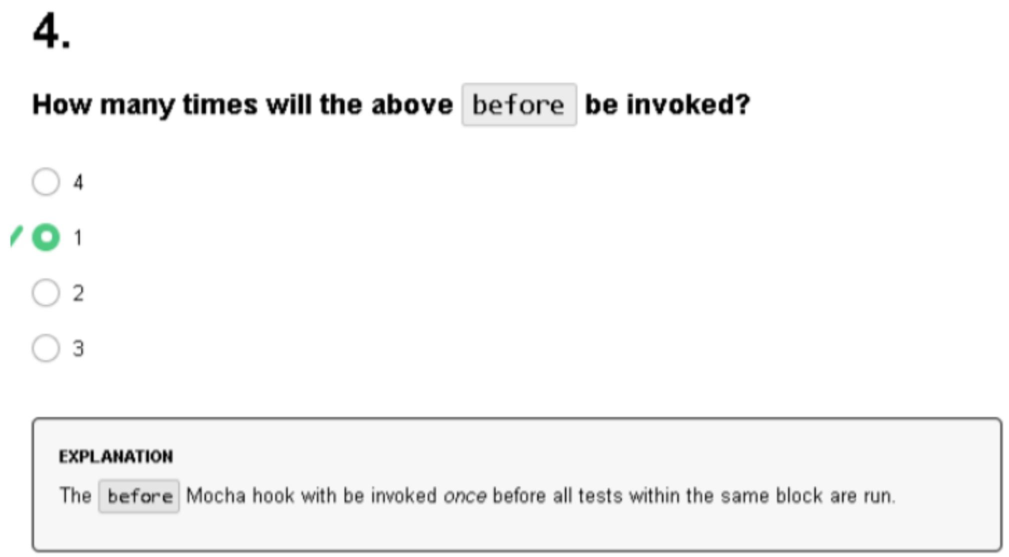
- `before`
- [✓ `afterEach`]
- `after`
- `beforeEach`

EXPLANATION

The `afterEach` Mocha hook with be invoked *after* each of the tests within the same block.



```
describe("animalMakers", function() {
  before(function() {
    console.log("after");
  });

  describe("penguinMaker", function() {
    it("should make penguins", () => {});
  });

  describe("catMaker", function() {
    it("should make cats", () => {});
  });
});
```



4.

How many times will the above `before` be invoked?

- 4
- [✓ 1]
- 2
- 3

EXPLANATION

The `before` Mocha hook with be invoked *once* before all tests within the same block are run.

10) know what content type is and how it is used in HTML body.

- **Headers** : Work just like HTTP requests.
  - Common Examples:
    - **`Location`** : Used by client for redirection responses.
    - **`Content-Type`** : Let's client know what format the body is in.
    - **`Expires`** : When response is no longer valid
    - **`Content-Disposition`** : Let's client know how to display the response.
    - **`Set-Cookie`** : Sends data back to the client to set on the cookie.
- **`Data`** : If the request is successful, the body of the response will contain the resource you have requested.
  Send a simple HTTP request to google.com - We can use netcat in the terminal to make a connection to a URL and send an HTTP request - `nc google.com 80` opens our connection to google.com - After we make our connection, we specify the three parts of an HTTP request: - Request line - Headers - Body - `GET / HTTP/1.1` creates the request-line, indicating our verb (GET), URI (/), and version (HTTP/1.1) - any other headers we would like (optional), such as `Accept: application/json` - body of the request (optional), such as `myKey=myValue`

11) Know how to send HTTP request to google manually:

(nc -v google.com 80 .... GET / HTTP/1.1) capitalize!

12) What kind of HTTP requests have a body?

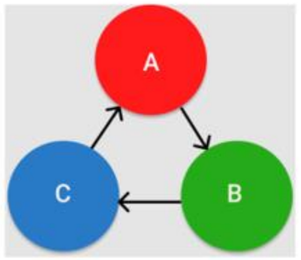
(Any HTTP Request;(when the data doesn't fit into the header &|| is too complex for the URI we can use the body)

It has a body when HTTP verb is POST, PUT, Patch

13) Know that HTTP stands for :

Hypertext Transfer Protocol

14) Three Stages of TDD (red, green , refactor) and what they mean



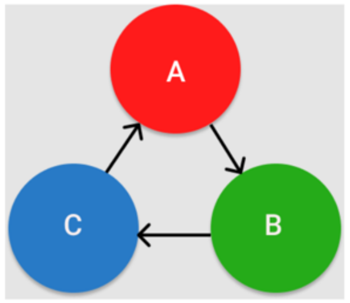
3.

The correct name of step A is:

- ○ Refactor
- ○ Green
- ✓ ● Red

EXPLANATION

The first step of the TDD workflow is **Red** where you write a test and watch it fail.



2.

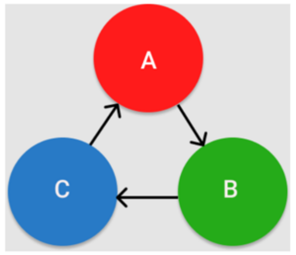
The correct name of step C is:

- ○ Green
- ✓ ● Refactor
- ○ Red

EXPLANATION

The third step of the TDD workflow is Refactor where you refactor the code you wrote to ensure it is of the highest quality.

TDD Workflow Quiz



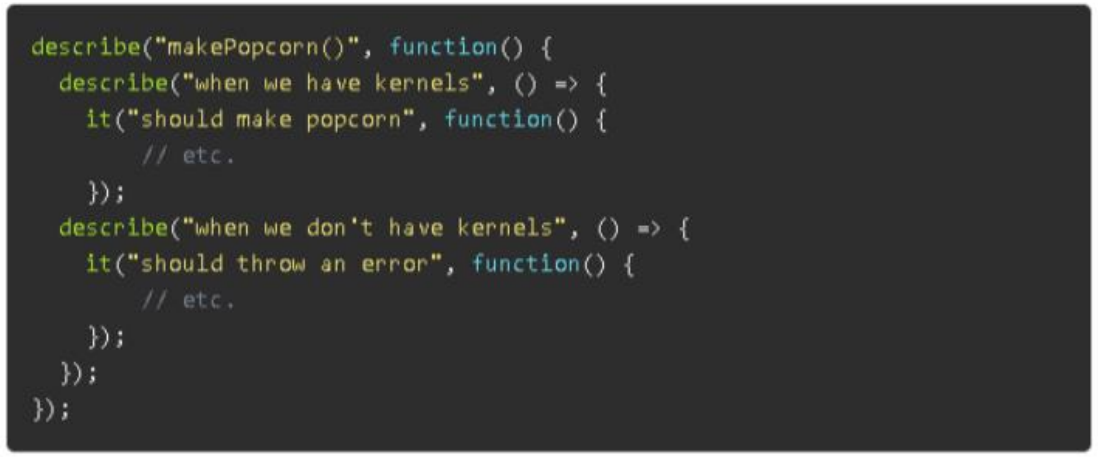
1.

The correct name of step B is:

- ○ Red
- ✓ ● Green
- ○ Refactor

EXPLANATION

The second step of the TDD workflow is **Green** where you write the minimum amount of code required to pass the test.



```
describe("makePopcorn()", function() {
  describe("when we have kernels", () => {
    it("should make popcorn", function() {
      // etc.
    });
  describe("when we don't have kernels", () => {
    it("should throw an error", function() {
      // etc.
    });
  });
});
```

5.

What about the above code snippet is not according to Mocha writing best practices?

- ○ The above code snippet improperly nests `it` blocks within `describe` blocks.
- ○ There should be an additional `describe` block wrapped around the entire test dictating how we are testing.
- ✓ ● We are using `describe` blocks instead of `context` blocks when signifying the state of the function.

EXPLANATION

In the above code snippet we are setting two states (`have kernels` vs. `don't have kernels`) using two `describe` blocks. We should instead use the alias for `describe`, `context`, to signify we are testing two different contexts.

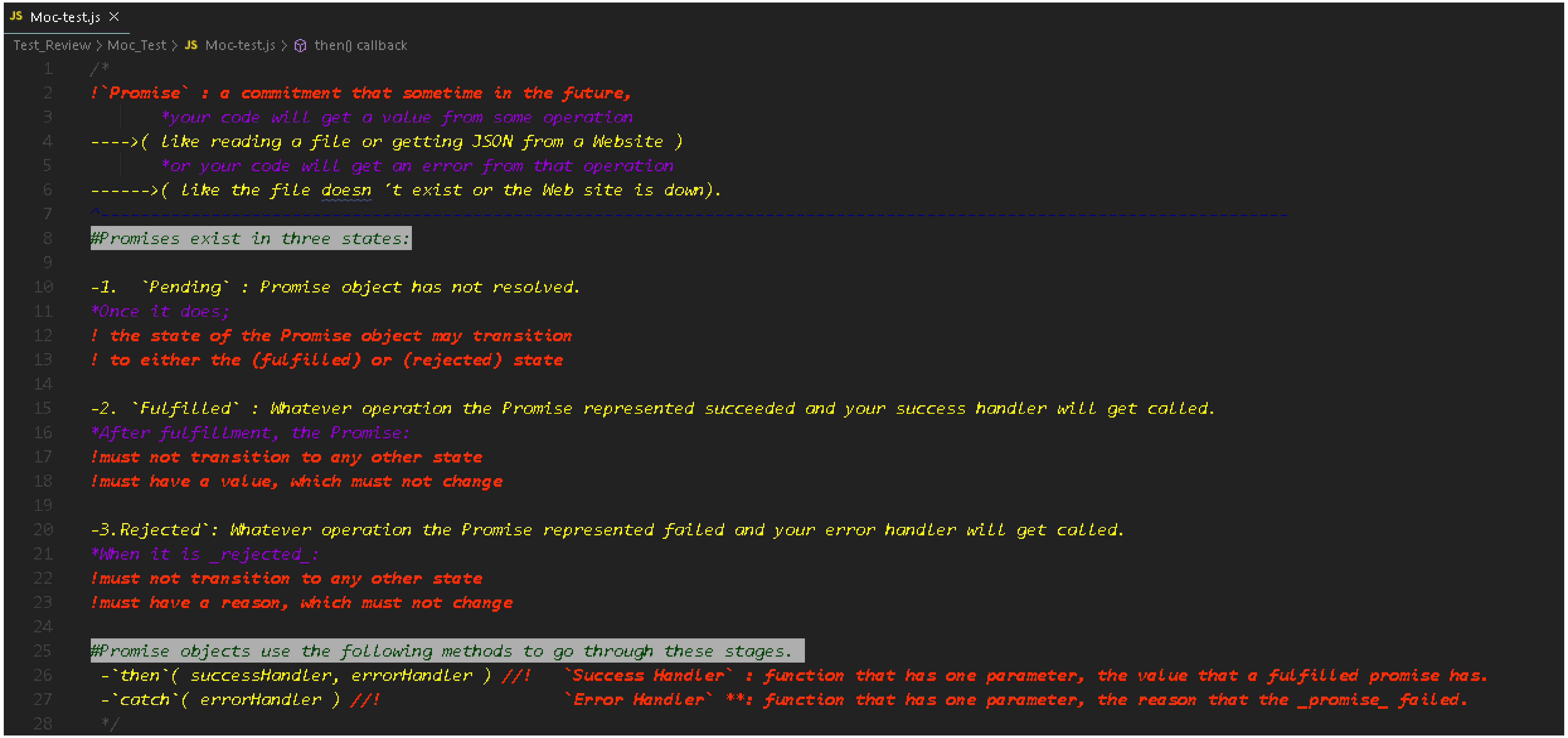
16) Parts of an HTTP Request

(3 parts: Request line & HTTP Verbs-method, URI, HTTP - Version; Headers-metadata as key value pairs; Body-update something on the backend)

Example of how an HTTP request looks like for visiting appacademy.io

- **Request-line & HTTP verbs**
  - The first line of an HTTP Request made up of three parts:
    1. The `Method` : Indicated by an HTTP Verb.
    2. The `URI` : Uniform Resource Indicator that ID's our request.
    3. THe `HTTP` **Version** : Version we expect to use.
  - HTTP Verbs are a simply way of declaring our intention to the server.
    - `GET` : Used for direct requests.
    - `POST`: Used for creating new resources on the server.
    - `PUT`: Used to updated a resource on the server.
    - `PATCH` : Similar to PUT, but do not require the whole resource to perform the update.
    - `DELETE` : Used to destroy resources on the server.
- **Headers**
  - Key-Value pairs that come after the request line - they appear on sep. lines and define metadata needed to process the request.
  - Some common headers:
    - `Host` : Root path for our URI.
    - `User-Agent` : Displays information about which browser the request originated from.
    - `Referer` : Defines the URL you're coming from.
    - `Accept` : Indicates what the client will receive.
    - `Content-` : Define Details about the body of the request.
- **Body**
  - For when we need to send data that doesn't fit into the header & is too complex for the URI we can use the *body*.
  - `URL encoding` : Most common way form data is formatted.
  - `name=claire&age=29&iceCream=vanilla`
  - We can also format using JSON or XML.

17) What is a promise? describe it in terms of synchronous and async & What are the states that a promise can be in (pending, fulfilled (resolved), rejected) (js as a language is synchronous/blocking).. I will eventually get some response from this promise



```
/*
!`Promise` : a commitment that sometime in the future,
        *your code will get a value from some operation
---->( like reading a file or getting JSON from a Website )
        *or your code will get an error from that operation
------>( like the file doesn 't exist or the Web site is down).
^---------------------------------------------------------------------------------------------------------------
#Promises exist in three states:

-1.  `Pending` : Promise object has not resolved.
*Once it does;
! the state of the Promise object may transition
! to either the (fulfilled) or (rejected) state

-2. `Fulfilled` : Whatever operation the Promise represented succeeded and your success handler will get called.
*After fulfillment, the Promise:
!must not transition to any other state
!must have a value, which must not change

-3.Rejected`: Whatever operation the Promise represented failed and your error handler will get called.
*When it is _rejected_:
!must not transition to any other state
!must have a reason, which must not change

#Promise objects use the following methods to go through these stages.
 -`then`( successHandler, errorHandler ) //!   `Success Handler` : function that has one parameter, the value that a fulfilled promise has.
 -`catch`( errorHandler ) //!                  `Error Handler` **: function that has one parameter, the reason that the _promise_ failed.
 */
```