HTTP Learning Objectives

HTTP stands for **HyperText Transfer Protocol.** Is a protocol for transmitting hypermedia documents, such as HTML. HTTP requests can have up to three parts: a request line, headers, and a body.

1. Match the header fields of HTTP with a bank of definitions

HTTP headers let the client and the server pass additional information with an HTTP request or response. Here are some common request headers you'll see:

- Host: specifies the domain name of the server.
- User-Agent: a string that identifies the operating system, software vendor or version of the requester.
- Referer: the address of the previous web page from which a link to the currently requested page was followed.
- Accept: informs the server about the types of data that can be sent back.
- Content-Type: Indicates the media type found in the body of the HTTP message.

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

HTTP defines a set of request methods to indicate the desired action to be performed for a given resource.

- GET: a request to retrieve data. It will never have a body.
- POST: sends data to the server creating a new resource.
- PUT: updates a resource on the server.
- PATCH: similar to PUT, but it applies partial modifications to a resource.
- DELETE: deletes the specified resource.

3. Match common HTTP status codes (200, 302, 400, 401, 402, 403, 404, 500) to their meanings

HTTP response status codes indicate whether a specific HTTP request has been successfully completed.

- 200: OK. The request has succeeded.
- 302: Found. The URI of requested resource has been changed temporarily.
- 400: Bad Request. The server could not understand the request due to invalid syntax.
- 401: Unathorized. The client must authenticate itself to get the requested response.
- 402: Payment Required.
- 403: Forbidden. The client does not have access rights to the content.
- 404: Not Found. The server can not find the requested resource.
- 500: Internal Server Error. The range from 500-599 indicate server errors.

4. Send a simple HTTP request to google.com

netcat (nc) allows you to open a direct connection with a URL and manually send HTTP requests.

Request

```
nc -v google.com 80
GET / HTTP/1.1
```

Response

```
HTTP/1.1 200 OK
Date: Thu, 28 May 2020 20:50:17 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
<!doctype html>
<html>
</html>
```

5. Write a very simple HTTP server using ‘http’ in node with paths that will result in the common HTTP status codes

```
const http = require('http');

http.createServer(function(request, response) {
    if (request.url === '/') {
        response.writeHead(
            200,
            { 'Content-Type': 'text/html' }
        );
        response.write('<h1>OK</h1>');
        response.end();
    } else {
        response.writeHead(404);
        response.end();
    }
}).listen(8080, function() {
    console.log(
        'listening for requests on port 8080...'
    );
});
```

Promises Lesson Learning Objectives I

1. Instantiate a Promise object

```
const myPromise = new Promise((resolve, reject) => {
    try {
        // try some code, if it works, then we can call `resolve()`
        someAsynchronousFunctionThatMightFail(result => {
            // If the async function works, it'll call this callback
            // and pass us the result of whatever it did.
            resolve(result); // Then we can call resolve with the result.
        });
    }
    catch (error) {
        // if we get an error we can call `reject()` with the error
        reject(error);
    }
});
```

2. Use Promises to write more maintainable asynchronous code

Let's assume we want to wait 10 seconds, do a thing, then wait 10 more then do a different thing, then wait 30 seconds and do a final thing. We can use setTimeout but with callbacks it looks like this:

```
setTimeout(() => { // Look at all the deep nesting!
    doAThing();
    setTimeout(() => {
        doADifferentThing();
        setTimeout(() => {
            doAFinalThing();
        }, 30000)
    }, 10000)
}, 10000)
```

If we wrap setTimeout in a promise like this:

```
const sleep = (milliseconds) => {
    return new Promise((resolve) => {
        setTimeout(resolve, milliseconds);
    });
}
```

Then we can write code that looks like this using our `sleep` function.

```
sleep(1000)
    .then(() => {
        doAThing();  // We don't need to return a promise here `.then()`
                     // automatically returns one anyway.
    })
    .then(() => {
        return sleep(1000); // We have to return the promise we
                            // got from the sleep function
                            // then() is smart enough to know when
                            // we return a promise vs just returning a value
    })
    .then(() => {
        doADifferentThing();
    })
    .then(() => {
        return sleep(3000);
    })
    .then(() => {
        doAFinalThing();
    });
```

Using shortened arrow function syntax we can make this even shorter and make it look really synchronous, even though it's asynchronous.

```
sleep(1000)
.then(() => doAThing())
.then(() => sleep(1000))
.then(() => doADifferentThing())
.then(() => sleep(3000))
.then(() => doAFinalThing());
```

You can also use `Promise.all()` when you don't care about the order.

```
const fs = require('fs').promises // requires the promises version of fs

// Read in three files and concatenate them together.
Promise.all([
    fs.readFile("d1.md", "utf-8"),
    fs.readFile("d2.md", "utf-8"),
    fs.readFile("d3.md", "utf-8"),
])
.then((contents1, contents2, contents3) => { // Even though they run
                                             // asynchronously
                                             // in an indeterminate order,
                                             // Promise.all keeps them in the
                                             // right order in the arguments
                                             // list
    return contents1 + contents2 + contents3;
})
.then((concatted) => {
    console.log(concatted);
});
```

Imagine if we tried to do this without Promises:

```
const fs = require('fs');

fs.readFile("d1.md", "utf-8", contents1 => {
    fs.readFile("d1.md", "utf-8", contents2 => {
        fs.readFile("d3.md", "utf-8", contents3 => {
            console.log(contents1 + contents2 + contents3);
        });
    });
});
```

In fact this isn't doing the same thing at all! This is actually only reading the next file when the first one is completed. Promise.all() is probably faster since all the readFiles are kicked off at the same time.

3. Use the fetch API to make Promise-based API calls

```
const fetch = require('node-fetch'); // we need this to use fetch in node
                                     // It's built into the browser
fetch("https://ifconfig.me/all.json")
  .then((response) => {
    return response.json(); // the json() method returns a promise and is async
  })
  .then((data) => {
    console.log(data);
  });
```

Async and Await Learning Objectives

1. Use async/await with promise-based functions to write asynchronous code that behaves synchronously.

```
function slow() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve('That was slow.');
        }, 2000);
    });
}

function fast() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve('That was fast.');
        }, 1000);
    });
}

async function syncLike() {
    const slowVal = await slow();
    console.log(slowVal);
    const fastVal = await fast();
    console.log(fastVal);
}

syncLike(); // Prints 'that was slow.' after 2000ms, then 'that was fast.' after 1000ms
```

The real power comes in leveraging libraries that support promises like `fs`
of `fetch`

Instead of:

```
const fs = require("fs");

function concatFiles() {
    fs.readFile("d1.md", "utf-8", (err, contents1) => {
        fs.readFile("d1.md", "utf-8", (err, contents2) => {
            fs.readFile("d3.md", "utf-8", (err, contents3) => {
            console.log(contents1 + contents2 + contents3);
            });
        });
    });
}

concatFiles();
```

We can do this:

```
const fs = require("fs").promises;

async function concatFiles() {
    const contents1 = await fs.readFile("d1.md", "utf-8");
    const contents2 = await fs.readFile("d2.md", "utf-8");
    const contents3 = await fs.readFile("d3.md", "utf-8");
    console.log(contents1 + contents2 + contents3);
}

concatFiles();
```

Or when using fetch, instead of this:

```
const fetch = require("node-fetch");

function getIpAddress(callback) {
    return new Promise((resolve) => {
        fetch("https://ifconfig.me/all.json")
        .then((response) => {
            return response.json();
        })
        .then((ipInfo) => {
            resolve(ipInfo);
        });
    }
}

getIpAddress().then(ipInfo => {
    console.log(ipInfo);
})
```

We can just write this:

```
const fetch = require('node-fetch');

async function getIpAddress() {
    const response = await fetch("https://ifconfig.me/all.json");
    const ipInfo = await response.json(); // Remember .json() returns a promise
    return ipInfo;
}

// Rememeber to use await, we must be inside an async function, so
// I just made an async IIFE.
(async() => {
    const ipInfo = await getIpAddress();
})()
```

Much better!

HTML Learning Objectives

1. You'll be able to create structurally and semantically valid HTML5 pages using the following elements: html, head, title, link, script, The six header tags, p, article, section, main, nav, header, footer, ul, ol, li, a, img, table, thead, tbody, tfoot, tr, th, td.

```
<!DOCTYPE html>
<html>
<head>
    <title>HTML Example</title>
    <link rel="stylesheet" href="style.css">
    <script async type="module" src="index.js"></script>
</head>
<body>
    <main>
        <h1>An HTML page example</h1>
        <p>
            This is a very basic HTML page. For more examples click
            <a href="https://open.appacademy.io/learn/js-py---apr-2020-online/week-6-apr-2020-online/brushing-up-on-your-ht
        </p>
    </main>
</body>
</html>
```

Testing Learning Objectives

1. Explain the "red-green-refactor" loop of test-driven development.

Red, Green, Refactor refers to the development loop at the heart of TDD.

- **RED:** We begin by writing a test (or tests) that speicify what we expect our code to do. We run the test, to see it fail, and in doing so we ensure that our test won't be a false positive.
- **GREEN:** We write the minimum amount of code to get our test to pass. This step may take just a few moments, or a longer time depending on the complexity of the task.
- **REFACTOR:** The big advantage of test driven development is that it means we always have a set of tests that cover our codebase. This means that we can safely make changes to our code, and as long as our tests still pass, we can be confident that the changes did not break any functionality. This gives us the confidence to be able to constantly refactor and simplify our code - we include a refactor step after each time we add a passing test, but it isn't always necessary to make changes.

2. Identify the definitions of SyntaxError, ReferenceError, and TypeError

- **SyntaxError:** These errors refer to problems with the *syntax* of our code, they usually refer to either missing or rogue characters that cause the compiler to be able to understand the code we are feeding it.
- **ReferenceError:** These errors refer to times in our code where we reference a variable that is *not* available in the current scope.
- **TypeError:** These errors refer to times in our code where we reference a variable of the *wrong type*. Modifying a value that cannot be changed, using a value in an inappropriate way, or an argument of an unexpected type being passed to a function, are all causes of TypeErrors.

3. Create, modify, and get to pass a suite of Mocha tests

A minimal example:

`test/reverse-string.spec.js`

```
const assert = require("assert");
const reverseString = require('../lib/reverse-string').reverseString;


describe("reverseString", () => {
    it("should reverse simple strings", () => {
        assert.equal(reverseString("fun"), "nuf");
    });
    it("should throw a TypeError if it doesn't receive a string", () => {
        assert.throws(() => reverseString(0));
    });
});
```

`lib/reverse-string.js`

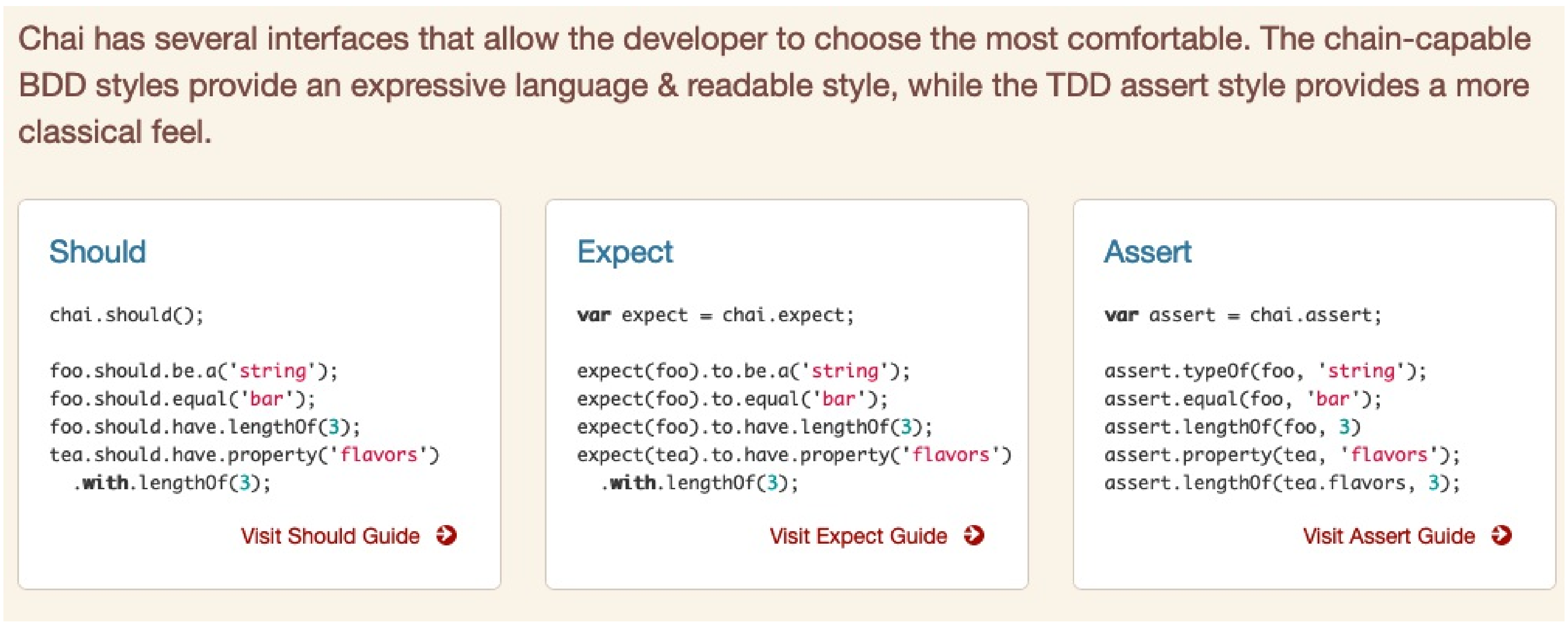
```
const reverseString = (str) => {
    if (typeof str !== "string") {
        throw new TypeError("expecting a string arg");
    }
    return "nuf";
}

module.exports = {
    reverseString
}
```

This minimal example also shows the importance of thorough testing. Though all tests here pass, it would be trivial to come up with a case where our reverseString implementation doesn't do what we want. `reverseString('foo')` for example.

4. Use Chai to structure your tests using behavior-driven development principles

Chai gives us the human-readable verbage that have become popular in the BDD (Behavior Driven Developement) world:



Chai has several interfaces that allow the developer to choose the most comfortable. The chain-capable BDD styles provide an expressive language & readable style, while the TDD assert style provides a more classical feel.

Should

```
chai.should();

foo.should.be.a('string');
foo.should.equal('bar');
foo.should.have.lengthOf(3);
tea.should.have.property('flavors')
  .with.lengthOf(3);
```

Visit Should Guide

Expect

```
var expect = chai.expect;

expect(foo).to.be.a('string');
expect(foo).to.equal('bar');
expect(foo).to.have.lengthOf(3);
expect(tea).to.have.property('flavors')
  .with.lengthOf(3);
```

Visit Expect Guide

Assert

```
var assert = chai.assert;

assert.typeOf(foo, 'string');
assert.equal(foo, 'bar');
assert.lengthOf(foo, 3)
assert.property(tea, 'flavors');
assert.lengthOf(tea.flavors, 3);
```

Visit Assert Guide

Both the `should` and `expect` style assertion terminology are considered BDD style, while the `assert` style expressions are a throwback to TDD style.

5. Use the pre- and post-test hooks provided by Mocha

Mocha provides four pre and post test hooks. These should be used to set up preconditions and clean up after your tests.

```
describe('hooks', function() {
  before(function() {
    // runs once before the first test in this block
  });

  after(function() {
    // runs once after the last test in this block
  });

  beforeEach(function() {
    // runs before each test in this block
  });

  afterEach(function() {
    // runs after each test in this block
  });

  // test cases
});
```

Hooks *run in the order they are defined*, as appropriate; all `before()` hooks run (once), then any `beforeEach()` hooks, tests, any `afterEach()` hooks, and finally `after()` hooks (once).