HTTP

The objective of this lessonis for you to get comfortable with the main concepts of HTTP. HTTP is the underlying protocol used by the World Wide Web. It's essential knowledge for developers who work with the web. At the end of it, you'll be able to identify common HTTP verbs and status codes, as well as demonstrating how HTTP is used by setting up a simple server.

When you finish, you should be able to

- match the header fields of HTTP with a bank of definitions.
- matching HTTP verbs (GET, PUT, PATCH, POST, DELETE) to their common uses.
- match common HTTP status codes (200, 302, 400, 401, 402, 403, 404, 500) to their meanings.
- send a simple HTTP request to `google.com`
- write a very simple HTTP server using ‘http’ in node with paths that will result in the common HTTP status codes.

Promises Lesson Learning Objectives I

Below is a complete list of the terminal learning objectives for this lesson. When you complete this lesson, you should be able to perform each of the following objectives. These objectives capture how you may be evaluated on the assessment for this lesson.

1. Instantiate a `Promise` object
2. Use `Promises` to write more maintainable asynchronous code
3. Use the `fetch` API to make `Promise`-based API calls

Promises Lesson Learning Objectives II

Below is a complete list of the terminal learning objectives for this lesson. When you complete this lesson, you should be able to perform each of the following objectives. These objectives capture how you may be evaluated on the assessment for this lesson.

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

HTML Learning Objectives

The objective of this lessonis for you to know how to effectively use HTML5 to build semantically and structurally correct Web pages. HTML is the language that renders the cross-platform human-computer interfaces that made the World Wide Web accessible by the world! 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
- Itemized list tags
    - ul
    - ol
    - li
- a
- img
- Tabular-data tags
    - table
    - thead
    - tbody
    - tfoot
    - tr
    - th
    - td

Testing

The objective of this lessonis to ensure that you understand the fundamentals of testing and are capable of reading and solving specs. **This lesson is relevant**to you because good testing is one of the foundations of being a good developer.

When you finish, you should be able to:

- Explain the "red-green-refactor" loop of test-driven development.
- Identify the definitions of SyntaxError, ReferenceError, and TypeError
- Create, modify, and get to pass a suite of Mocha tests
- Use Chai to structure your tests using behavior-driven development principles.
- Use the pre- and post-test hooks provided by Mocha

Well-Tested Full-Stack To-Do Items

In this project, you will test a full-stack JavaScript and HTML application! You will write tests to make sure the code that was written for the project will meet the expectations of the requirements. Your tests will not have to be exhaustive. Instead, there are guidelines for your tests in each test file. Use those guidelines to implement the Web application.

The upcoming video provides you a full walk-through of the system as it is created. Then, once you understand how the application works from watching it be built, you will need to apply your knowledge of writing tests.

It may be hard. However, stick with it. You'll do great. Just take your time, write good tests, and you will be amazed at how much confidence that you will gain in writing code that comes together.

One of the ways that you can make this project more enjoyable is to vary the way that you pair on it. For each step,

- Discuss what the next feature is that you want to write
- One person writes a unit test to test the code
- Both examine the code and determine if there is any duplication to refactor into common functions or classes
- Loop, but swap who writes the unit test and who writes the code

At the end, you will leverage your test by swapping out the mechanism used to generate the HTML. This is the other part of writing good tests: tests give you the confidence to change code. If you do something that is "wrong" in that it breaks current expectations, the tests will tell you!

Tests are such an important part of a developers life. While some developers will complain about having to write them, when you start working on an existing "legacy" code base, making changes can cause a lot of stress unless you have the work of other developers' tests to make sure you don't unintentionally change a method in such a way to break code that you're not working on.

To get started,

- clone the project fromhttps://github.com/appacademy-starters/testing-an-existing-app-project
- change directory into the project
- run `npm install`to install the modules

If you want to run the server, type `node server.js`and go tohttp://localhost:3000/itemsto see what it does. You can add categories, add items, search for items, and complete items.

The code that you will test are the functions used to create the functionality of the data and the creation of the views, not the actual HTTP server. The video after shows a person writing the entire application. You will see what each piece does. Then, you will understand the *intent* of the code that you have to test.

HTTP

The objective of this lessonis for you to get comfortable with the main concepts of HTTP. HTTP is the underlying protocol used by the World Wide Web. It's essential knowledge for developers who work with the web. At the end of it, you'll be able to identify common HTTP verbs and status codes, as well as demonstrating how HTTP is used by setting up a simple server.

When you finish, you should be able to

- match the header fields of HTTP with a bank of definitions.
- matching HTTP verbs (GET, PUT, PATCH, POST, DELETE) to their common uses.
- match common HTTP status codes (200, 302, 400, 401, 402, 403, 404, 500) to their meanings.
- send a simple HTTP request to `google.com`
- write a very simple HTTP server using ‘http’ in node with paths that will result in the common HTTP status codes.

HTTP Basics

In the late 1980s, a computer scientist named Tim Berners-Lee proposed the concept of the "WorldWideWeb", laying the foundation for our modern Internet. A critical part of this concept was *HTTP*, the *Hypertext Transfer Protocol*.

We're going to to dive into what makes HTTP such an important part of Web browsing and learn how to leverage it in our applications.

We'll cover:

- the vocabulary of the Web,
- how stateless connections work,
- and HTTP request & response types.

First, some context

"If you want to build a ship, don't drum up the men and women to gather wood, divide the work, and give orders. Instead, teach them to yearn for the vast and endless sea."

-- Antoine de Saint-Exupéry (paraphrased)

So far, you've written code that runs in isolation on your own system. Now it's time to set sail into the "vast and endless" Internet! Before we can do so, we need to review the fundamentals: what makes the Web a "web"?

We're going to share a lot of vocabulary here, and it may be a little dry at times, but remember that these are the principles upon which the rest of your journey will be built! You'll find these concepts missing from most programming tutorials, so you'll be ahead of the game if you lay a strong foundation now.

Breaking it down...

Like many disciplines, computer science is built around a shared vocabulary. Let's demystify the acronym "HTTP" to understand it better.

HT-: HyperText

*Hypertext*is simply "content with references to other content". This term is used specifically to refer to content in computing, and may include text, images, video, or any other digital content. If "hypertext" sounds familiar, that's because you've heard it before: *HTML*stands for "**H**yper**T**ext**M**arkup **L**anguage".

Hypertext is what makes the Web a "web", and it's the most fundamental part of how we interact online. We refer to references between hypertext resources as*hyperlinks*, though you're probably used to hearing them referred to as*links*. Without links, the Internet would resemble a massive collection of separate books: each blog, news report, and social media site would exist in total isolation from each other. The ability to link these pages is what makes the kind of interactivity you're learning to build possible, and it was a revolutionary concept when it was introduced!

-TP: Transfer Protocol

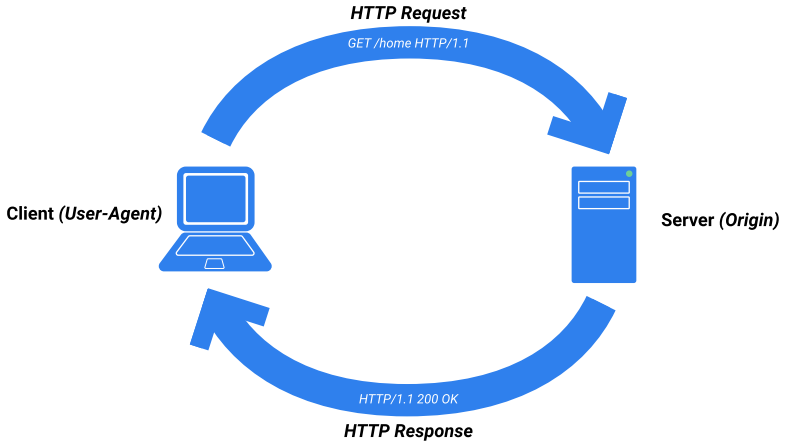
A *protocol*in computer science is a set of guidelines surrounding the transmission of data. Protocols define the process of exchanging data, but don't define exactly what that data must be. Think of it like a multi-course meal: we expect the appetizer, then the entree, then the dessert, but we could have any type of food for each of those courses! As long as the plates arrive in the particular order we expect, protocol is being followed.

HTTP acts as a *transfer protocol*. It defines the expectations for both ends of the transfer, and it defines some ways the transfer might fail. More specifically, HTTP is defined as a *request/response*protocol. An HTTP exchange is more like a series of distinct questions & answers than a conversation between two systems.

...and bringing it back together

HTTP defines the process of exchanging hypertext between systems. Specifically, HTTP works between *clients*and *servers*. A *client*(sometimes called the *user agent*) is the data consumer. This is usually your web browser. A *server*(sometimes referred to as the *origin*) is the data provider, often where an application is running. In a typical HTTP exchange, the client sends a *request*to the server for a particular *resource*: a webpage, image, or application data. The server provides a *response*containing either the resource that the client requested or an explanation of why it can't provide the resource.

Here's a high-level overview of the exchange:



We'll look more closely at the *request*and *response*in separate lessons.

Properties of HTTP

There are a few important properties of HTTP that we need to understand in order to use it effectively.

Reliable connections

Let's consider the example of two friends passing a note. If the note contains important information, the sender will want to make sure that it gets to its destination. They'll likely take a little extra time to deliver it carefully, and they'll expect confirmation once it's been received. In computing, we'd refer to this as a *reliable connection*: messages passed between a client & server sacrifice a little speed for the sake of trust, and we can rest assured that each message will be confirmed.

HTTP doesn't work well if messages aren't received in the correct order, so it's critical that the connection your hypertext is crossing is reliable! Tim Berners-Lee chose *TCP*, another transmission protocol, as HTTP's preferred connection type. We'll discuss TCP in greater detail when we get into network models in a future lesson.

Stateless transfer

HTTP is considered a *stateless*protocol, meaning it doesn't store any information. Each request you send across an HTTP connection should contain

all its own context. This is unlike a *stateful*protocol, that might include specifications for storing data between requests.

This can be nice because we only ever need to read a single HTTP request to understand its intent, but it can cause headaches when it comes to things like maintaining your login status or the contents of your shopping cart!

To help us with this, HTTP supports *cookies*, bits of data that a client sends in with their request. The server can examine this data and look up a *session*for your account, or it can act on the info in the cookie directly. Note that neither the cookie nor the session are part of HTTP. They're just workarounds we've created due to the protocol's stateless nature.

Intermediaries

The Web is a big place, and it's unlikely that your request will go directly to its destination! Instead, it will pass through a series of *intermediaries*: other servers or devices that pass your request along. These intermediaries come in three types:

- *proxies*, which may modify your request so it appears to come from a different source,
- *gateways*, which pretend to be the resource server you requested,
- and *tunnels*, which simply pass your request along.

Here's an idea of how these intermediaries might be laid out:

Notice that these are interchangeable depending on the flow of data. When the response is sent back, "Their Router" is acting as a proxy and "Your Router" is acting as a gateway! This is an important part of HTTP: a single server may act as any of the intermediary types, depending on the needs of the HTTP message it's transmitting.

We'll discuss some of these intermediaries more in later lessons. For now, the takeaway is that HTTP isn't limited to your browser & application server. Lots of devices support HTTP in their own special way.

Digging deeper with the HTTP spec

We're just scratching the surface of how HTTP works. If you're interested in learning more, you can go straight to the source: the HTTP spec. A *spec*(short for *specification*) describes a protocol in great detail. It's the document generated by an idea's founders, and it's reviewed and carefully edited before being adopted by the IETF(*Internet Engineering Task Force*).

Specs are intended to be exhaustive, so they can be overwhelming at first! This is definitely not light reading but any question you have about a particular protocol can likely be answered from its spec.

What we've learned

Whew, that's a lot of jargon! Hopefully the fundamental aspects of HTTP are clearer to you now. Next up, we'll look at an HTTP request & response, and we'll cover how to generate each.

After completing this lesson, you should have a clear understanding of:

- HTTP's origin & purpose,
- special properties of HTTP,
- and how to learn more from the HTTP spec.

HTTP Requests

Without a query, there wouldn't be a need for a response! Let's take a look at the *request*: the client-initiated portion of an HTTP exchange.

We'll cover:

- what an HTTP request looks like,
- fields that make up a request,
- and how to send a request of your own!

Retrieving hypertext

Years ago, daily shopping looked very different. Instead of walking the aisles and picking up what they wanted, customers would approach a counter and ask a clerk to retrieve the items on their list. The clerk was responsible for knowing where those items were located and how best to get them to the customer.

While the retail industry has changed dramatically since that time, the Web follows that old tried-and-true pattern. You tell your browser which website you would like to access, and your browser hands that request off to a server that can get you what you've asked for. At the simplest level, the Web is just made up of computers asking each other for things!

Your browser's part in this transaction is called the *request*. Since the browser is acting on your behalf, we sometimes refer to it as the *user-agent*(you being the *user*). You might also hear this referred to more generically as the *client* in the exchange.

Structure of an HTTP request

Your browser is designed to be compliant with the HTTP specification, so it knows how to translate your instructions into a well-formatted HTTP request. An important part of the HTTP spec is that it's simple to read, so let's take a look at an example.

Here's what the HTTP request looks like for visiting `appacademy.io`:

```
GET / HTTP/1.1
Host: appacademy.io
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_5) AppleWebKit/537.36 (K
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/ap
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
```

Let's break it down!

Request-line & HTTP verbs

The first line of an HTTP request is called the **request-line**, and it sets the stage for everything to come. It's made up of three parts, separated by spaces:

- the *method*, indicated by an *HTTP verb*,
- the *URI (Uniform Resource Indicator)*that identifies what we've requested,
- and the *HTTP version*we expect to use (usually `HTTP/1.1`or `HTTP/2`).

In our `appacademy.io`example, we can see that our version matches the most common HTTP version (`1.1`) and that our URI is `/`, or the *root*resource of our target. That first word, `GET` is the HTTP verb we're using for this request.

HTTP verbs are a simple way of declaring our intention to the server. We do the same thing with English verbs when asking for help: "Can you **get** me

that?", "Should I **remove** this?", etc. HTTP has a small handful of verbs available, but we're going to look at the five most common: `GET`, `POST`, `PUT`, `PATCH`, and `DELETE`.

- `GET` is used for direct requests. A `GET` request is generally how websites are retrieved, and they only require that the server return a resource. These types of requests will never have a body. Any data you need to send in a `GET` request must be shared via the URI.
- `POST` is typically used for creating new resources on the server. Most of the time, when you submit a form a `POST` request is generated. These types of requests can have a *body* containing any data the server might need to complete your request, like your username & password or the contents of your shopping cart.
- `PUT` requests are used to update a resource on the server. These will contain the whole resource you'd like to update. For example: when updating your name on a website, a `PUT` request will be generated containing not just your new name but also your user ID, email, etc.
- `PATCH` requests are very similar to `PUT` requests, but do not require the whole resource to perform the update. Keeping with our example of updating your name: a `PATCH` request would only require your new name, not the rest of your account details, to succeed.
- `DELETE` requests destroy resources on the server. These might be saved database records, like removing a product that's sold out, or more ephemeral resources, like logging a user out of their current session.

Ultimately, how these verbs get acted upon is up to the server. You could write an application that totally ignores these rules and uses a `DELETE` request to log in, but that's only going to confuse your teammates and frustrate you in the future! It's best to use them as the spec intends.

Headers

The *request-line* sets the table, but it's the headers that describe the menu! *Headers* are key/value pairs that come after the *request-line*. They each appear on separate lines and define metadata needed to process the request. Here are some common request headers you'll see:

- `Host`: The root path for our URI. This is typically the *domain* we'd like to request our resource from. As you can see above, our `Host` header for `appacademy.io` is, appropriately, `appacademy.io`.!
- `User-Agent`: This header displays information about which browser the request originated from. It's generally formatted as `name/version`. You can see in the `User-Agent` header above that we're using `Chrome/76.0`

  Our `User-Agent` has much more content, including references to Mozilla, makers of the popular Firefox browser, and Safari, Apple's default browser of choice. What gives?

  There is some interesting history behind those additional references, and you can use www.useragentstring.com for additional details about your current browser's `user-agent`.
- `Referer`: This defines the URL you're coming from. There's none in our example since we navigated directly to the App Academy website, but if we click any link on the page, the resulting HTTP request will have `Referer: https://appacademy.io/` in its headers. Also, you're not reading it wrong - this header is misspelled! It should be "referrer", but it was written incorrectly in the original specification and the typo stuck. Let this be a lesson: your poorly-written code might still be around in 20 years, too!
- `Accept`: "Accept-" headers indicate what the client can receive. When we go to most websites, our `Accept` header will be long to ensure we get all the various types of content that site might include. However, we can modify this

header in our requests to only get back certain types of data. One common use is setting `Accept: application/json` to get a response in JSON format instead of HTML. You may see variations of this header like `Accept-Language` for internationalized websites or `Accept-Encoding` for sites that support alternative compression formats.

- `Content-*`: Content headers define details about the body of the request. The most common content header is `Content-Type`, which lets the server know what format we're sending our body data as. This might be `application/json` from a JavaScript app or `application/x-www-form-urlencoded` for info submitted from a web form. Content headers will only show up on requests that support content in the body, so `GET` requests should never have this!

There are LOTS of other header keys! MDN has an exhaustive reference list with examples.

Body

When we need to send data that doesn't fit in a header & is too complex for the URI, we can place it in the *body* of our HTTP request. The body comes right after the headers and can be formatted a few different ways.

The most common way form data is formatted is *URL encoding*. This is the default for data from web forms and looks a little like this:

```
name=claire&age=29&iceCream=vanilla
```

Alternatively, you might format your request body using JSON or XML or some other standard. What's most important is that you remember to set the appropriate `Content-Type` header so the server knows how to interpret your body.

Sending an HTTP request from the command line

We've discussed HTTP requests mostly in the context of your web browser, but that's not the only way. There are lots of HTTP clients out there you can use to send requests.

Let's stay close to the exchange itself with a lightweight tool that requires us to do most of the work ourselves. We'll use `netcat` (also known as `nc`), a utility that comes as part of Unix-like environments such as Ubuntu and macOS.

`netcat` allows you to open a direct connection with a URL and manually send HTTP requests. Let's see how this works with a quick `GET` request to App Academy's homepage.

From your command line, type `nc -v appacademy.io 80`. This will open a connection to `appacademy.io` on port 80 (the port most-often used for web connections). Once the connection is established, you'll be able to type out a simple HTTP request by hand! Let's copy the *request-line* and `Host:` header from our request above:

```
GET / HTTP/1.1
Host: appacademy.io
```

Now hit "Return" on your keyboard twice. This will send the request and display the server's response. You should see something similar to this:

```
HTTP/1.1 301 Moved Permanently
Date: Thu, 03 Oct 2019 04:17:23 GMT
Transfer-Encoding: chunked
Connection: keep-alive
Cache-Control: max-age=3600
Expires: Thu, 03 Oct 2019 05:17:23 GMT
Location: https://www.appacademy.io/
Server: cloudflare
CF-RAY: 51fc1b0f8b98d304-ATL
```

Congratulations! You've sent your first manual HTTP request. We'll discuss the parts of the HTTP response you received in an upcoming lesson.

Try it one more time, this time typing `nc -v neverssl.com 80` and making the same HTTP request with the command `GET / HTTP/1.1`and the header `Host: neverssl.com`. Don't forget to hit Enter twice. Look! That's the HTML coming back from the server! Neat-o!

You can read much more about `netcat`by invoking the manual: `man nc`. We'll also use it in an upcoming project for extra practice.

What we've learned

HTTP requests are the first step to getting what you want on the web. Having completed this lesson, you should be able to recount:

- what an HTTP request is,
- some common HTTP request verbs,
- a rough outline of the HTTP request format,
- and how to use `netcat`to send HTTP requests from your command line.

HTTP Responses

A web server delivers content via *responses*, the second part of the HTTP's request/response cycle. Let's dive into how a response is structured and what your client can expect from the server.

We'll cover:

- HTTP response structure,
- differentiating errors & successful transfers,
- and how to use a server to generate your own responses.

Hypertext delivered

An HTTP response contains either the content we requested or an explanation of why that content couldn't be delivered. It's just like ordering at a restaurant: you place your order and receive either a plate of delicious food or an apology from the chef. In a good restaurant, the apology will include some extra help: "I'm sorry, we're out of broccoli. Can we get you something else? How can we make this right?".

When designing your own HTTP responses, remember that restaurant example. It's important to note that there's a problem, but it's equally important to provide reliable, helpful details. We'll look at some examples of this when we build our own HTTP server in a later lesson.

Structure of a Response

Responses are formatted similarly to requests: we'll have a *status-line*(instead of a request-line), headers that provide helpful metadata about the response, and the response body: a representation of the requested resource.

Here's what the HTTP response looks like when visiting `appacademy.io`:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Transfer-Encoding: chunked
Connection: close
X-Frame-Options: SAMEORIGIN
X-Xss-Protection: 1; mode=block
X-Content-Type-Options: nosniff
Cache-Control: max-age=0, private, must-revalidate
Set-Cookie: _rails-class-site_session=BAh7CEkiD3Nlc3Npb25faWQGOgZFVEkiJTM5NWM5YTV
X-Request-Id: cf5f30dd-99d0-46d7-86d7-6fe57753b20d
X-Runtime: 0.006894
Strict-Transport-Security: max-age=31536000
Vary: Origin
Via: 1.1 vegur
Expect-CT: max-age=604800, report-uri="https://report-uri.cloudflare.com/cdn-cgi/
Server: cloudflare
CF-RAY: 51d641d1ca7d2d45-TXL

<!DOCTYPE html>
<html>
...
...
</html>
```

Oof! That's a lot of unfamiliar stuff. Let's walk through the important bits together.

Status

Like the request, an HTTP response's first line gives you a high-level overview of the server's intention. For the response, we refer to this as the*status-line*.

Here's the status line from our `appacademy.io` response:

```
HTTP/1.1 200 OK
```

We open with the HTTP version the server is responding with. `1.1` is still the most commonly used, though you may occasionally see `2` or even `1.0`. We follow this with a `Status-Code` and `Reason-Phrase`. These give us a quick way of understanding if our request was successful or not.

HTTP status codes are a numeric way of representing a server's response. Each code is a three-digit number accompanied by a short description. They're grouped by the first digit (so, for example, all "Informational" codes begin with a `1`: `100` - `199`).

Let's take a look at the most common codes in each group.

Status codes 100 - 199: Informational

Informational codes let the client know that a request was received, and provide extra info from the server. There are very few informational codes defined by the HTTP specification and you're unlikely to see them, but it's good to know that they exist!

Status codes 200 - 299: Successful

Successful response codes indicate that the request has succeeded and the server is handling it. Here are a couple common examples:

- **200 OK**: Request received and fulfilled. These usually come with a `body` that contains the resource you requested.
- **201 Created**: Your request was received and a new record was created as a result. You'll often see this response to `POST` requests.

Status codes 300 - 399: Redirection

These responses let the client know that there has been a change. There are a few different ways for a server to note a redirect, but the two most common are also the most important:

- **301 Moved Permanently**: The resource you requested is in a totally new location. This might be used if a webpage has changed domains, or if resources were reorganized on the server. Most clients will automatically process this redirect and send you to the new location, so you may not notice this response at all.
- **302 Found**: Similarly, to *301 Moved Permanently*, this indicates that a resource has moved. However, this code is used to indicate a temporary move. It's not often that you see temporary moves online, but this code may be used to indicate a permanent move where the old domain should still be valid too. Clients will usually follow this redirect automatically as well, but you shouldn't necessarily update your links until the server returns a `301`.

301 Moved Permanently and 302 Found often get confused. When might we want to use a 302 Found `? The most common use case today is for the transition from HTTP to HTTPS. _HTTPS is secure HTTP messaging, where requests & responses are encrypted so they can't be read by prying eyes while en route to their destinations.

This is a much safer way of communicating online, so most websites require access via `https://` before the domain. However, we don't want to ignore folks still trying to access our content from the older `http://` approach!

In this case, we'll return a 302 Foundresponse to the client, letting them know that it's okay to keep navigating to `http://our-website.com`*, but we're going to redirect them to* `https://our-website.com`*for their protection.*

Status codes 400 - 499: Client Error

The status codes from 400 to 499, inclusive, indicate that there is a problem with the client's request. Maybe there was a typo, or maybe the resource we requested is no longer available. You'll see lots of these as you're learning to format HTTP requests. Here are the most common ones:

- **400 Bad Request**: Whoops! The server received your request, but couldn't understand it. You might see a *400 Bad Request*in response to a typo or accidentally truncated request. We often refer to these as *malformed*requests.
- **401 Unauthorized**: The resource you requested may exist, but you're not allowed to see it without authentication. These type of responses might mean one of two things: either you didn't log in yet, or you tried to log in but your credentials aren't being accepted.
- **403 Forbidden**: The resource you requested may exist, but you're not allowed to see it *at all*. This response code means this resource isn't accessible to you, even if you're logged in. You just don't have the correct permission to see it.
- **404 Not Found**: The resource you requested doesn't exist. You may see this response if you have a typo in your request (for example: going to`appaccccademy.io`), or if you're looking for something that has been removed.

403 Forbiddenrequests let the client know that a valid resource was requested. This can be a security risk! For example: if I guess that you havepasswords.htmlon your website because you just want to be hacked, a 403 Forbiddenresponse tells me I'm correct. For this reason, some sites will return a 404 Not Foundfor resources that exist but aren't accessible.

A well-known example is GitHub. If you try to open a repository you don't have permission to access, GitHub will return a 404 Not Foundeven if your URL is correct! This protects you from random users guessing the names of your projects.

Status codes 500 - 599: Server Error

This range of response codes are the Web's way of saying "It's not you, it's me." These indicate that your request was formatted correctly, but that the server couldn't do what you asked due to an internal problem.

There are two common codes in this range you'll see while getting started:

- **500 Internal Server Error**: Your request was received, and the server tried to process it, but something went awry! As you're learning to write your own servers, you'll often see a *500 Internal Server Error*as your code fails unexpectedly.
- **504 Gateway Timeout**: Your request was received but the server didn't respond in a reasonable amount of time. Timeout errors can be tricky: your first instinct may be that your own connection is bad, but this code means the problem is likely on the server's side. You'll often see these when a server is no longer reachable (maybe due to an unexpected outage or power failure).

Headers

Headers on HTTP responses work identically to those on requests. They establish metadata that the receiving client might need to process the response. Here are a few common response headers you'll see:

- **Location**: Used by the client for redirection responses. This contains the URL the client should redirect to.
- **Content-Type**: Lets the client know what format the body is in. Your client will display different types of response content in different ways, to setting the *Content-Type*is important! Notice that this header can be present on responses **and**requests. It's a generic header for any HTTP interaction involving content.
- **Expires**: When the response should be considered *stale*, or no longer valid. The *Expires*header lets your client *cache*responses (that is: save them locally to prevent having to repeatedly re-download them). The client may ignore requests to that same resource until after the date set in the*Expires*header.
- **Content-Disposition**: This header lets the client know how to display the response, and is specifically devoted to whether the response should be visible to the client or delivered as a download. Think about your own experience online: sometimes you click a button and get an immediate download, while in other cases you click a button and get to "preview" the content before you download it. This is controlled by the *Content-Disposition*header.
- **Set-Cookie**: This header sends data back to the client to set on the*cookie*, a set of key/value pairs associated with the server's domain. Remember how HTTP is *stateless*? Cookies are one way to get around that!*Set-Cookie*may send back information like a unique ID for the user you've logged in as or details about other resources you've requested on this domain.

Remember - this isn't an exhaustive list of headers! Sites and intermediate gateways/proxies can define their own custom headers, so you'll see many more than these. If you're unsure what a header does, the MDN HTTP Header documentationis a great place to start searching.

Body

Assuming a successful request, the *body*of the response contains the resource you've requested. For a website, this means the HTML of the page you're accessing.

The format of the body is dictated by the *Content-Type*header. This is an important detail! If you accidentally configure your server to send "Content-Type: application/json" along with a body containing HTML, your HTML won't be rendered properly and your users will see plain text instead of beautifully-rendered elements. In the same way, API responses should be clearly marked so that other applications know how to manage them.

We can see in our `appacademy.io`response above that the body begins with `!<DOCTYPE html>`and ends with `</html>`. If you inspect the source of the page in your browser, you'll see that this is exactly what's being rendered. Headers may change **how** the browser handles the body, but they won't **modify**the body's content.

Using a custom server to generate responses

At its most basic, a web server is just a tool to generate HTTP responses. Therefore, the best way to practice is to build your own webserver!

We'll walk through building our own server from scratch using Node.js in an upcoming video lesson.

What we've learned

Like HTTP requests, HTTP responses involve lots of new lingo and details. Hang in there - we'll start doing practical work with this new vocabulary in the projects & video demos coming up.

After this reading, you should:

- Understand the parts of an HTTP response,
- be able to identify common status codes & their meanings,
- recognize common response headers,
- and be prepared to build your own server!

Promises Lesson Learning Objectives I

Below is a complete list of the terminal learning objectives for this lesson. When you complete this lesson, you should be able to perform each of the following objectives. These objectives capture how you may be evaluated on the assessment for this lesson.

1. Instantiate a `Promise` object
2. Use `Promises` to write more maintainable asynchronous code
3. Use the `fetch` API to make `Promise`-based API calls

A Promise is a Promise: A Mostly Complete Guide to JavaScript Promises I

This article is about a JavaScript feature formally introduced into the language in 2015: the `Promise` object. The technical committee that governs the JavaScript language recognized that programmers had a hard time reasoning about and maintaining asynchronous code. They included `Promise`s as a way to encourage writing asynchronous code in a way that *appeared* synchronous.

When you finish this article, you should be able to:

- Provide examples of why `Promise`-based code is easier to maintain than traditional asynchronous callbacks;
- Recall the three states of a `Promise`, what each state means, and any associated data with that state.

A quick review of function declarations

It's important to remember about how JavaScript handles the declaration of a function. Please look at the following code.

```
function loudLog(message) {
  console.log(message.toUpperCase());
}
```

When JavaScript encounters that code, it does not run the function. You probably know that, but it's important to read again. When JavaScript encounters that code, it does not run the function.

It *does* create a `Function` object and stores that in a variable named `loudLog`. At some time later, you can run the function object in that variable with the syntax `loudLog("error occurred");`. That *runs* the function. Just declaring a function doesn't run it. Look at this following code.

```
function () {
  console.log('How did you call me?');
}
```

JavaScript will, again, create a `Function` object. However, there's no name for the function, so it doesn't get assigned to any variable, and just disappears with no way for us to use it. So, why would you declare functions without names?

The looming problem of asynchronous code with callbacks

Let's look at the documentation for how to read files in Node.js. Don't worry if you haven't used Node.js, yet. It's just like any other JavaScript.

```
readFile(path, encoding, callback)

Arguments:
  path      <string>    path to the file
  encoding  <string>    the encoding of the file
  callback  <function>  two arguments:
                        err      <error object>
                        content  <string>

Asynchronously reads the entire contents of a file.
```

The function named `readFile` accepts two arguments, a string that contains the `path` to the file and a function that `readFile` calls once it's read the content

of the file. If you wanted to write out the content of the file with a header, you could write code like this.

```
function writeWithHeader(err, content) {
  console.log("YOUR FILE CONTAINS:");
  console.log(content);
}

readFile('~/Documents/todos.txt', 'utf8', writeWithHeader);
```

Recall that when JavaScript found the function declaration at the beginning of that code block, it created a `Function` object and stored it in a variable named `writeWithHeader`. Now, that variable contains the actual function that can later be run. That code passes the value of that variable, the `Function` object, into the `readFile` function so the `readFile` function can run it later.

If you're not going to use the `writeWithHeader` function anywhere else in your code, idiomatic JavaScript instructs you to get rid of the name of the function and declare it directly as the second argument of the `readFile` functions. That would turn the above code block into the following.

```
readFile('~/Documents/todos.txt', 'utf8', function (err, content) {
  console.log("YOUR FILE CONTAINS:");
  console.log(content);
});
```

Since 2015, idiomatic JavaScript would instruct you to get rid of the function keyword and just use an arrow function.

```
readFile('~/Documents/todos.txt', 'utf8', (err, content) => {
  console.log("YOUR FILE CONTAINS:");
  console.log(content);
});
```

The key to remember here is that you have only declared that function that `readFile` will call later, `readFile` is in charge of running that function.

Imagine that you have a file that has a list of other file names in it named `manifest.txt`. You want to read the file and read each of the files listed in it. Then, you want to count the characters in each of those files and print those numbers.

You would start out by reading `manifest.txt` and splitting the content on the newline character to get the names of the files. That would look like this:

```
readFile('manifest.txt', 'utf8', (err, manifest) => {
  const fileNames = manifest.split('\n');

  // More to come
});
```

Now that you have the list of file names, you can loop over them to read each of those files. As each of those files are read, you want to count the characters in each file. Imagine that you already have the function named `countCharacters` somewhere. The looping code could look like this:

```
readFile('manifest.txt', 'utf8', (err, manifest) => {
  const fileNames = manifest.split('\n');
  const characterCounts = {};

  // Loop over each file name
  for (let fileName of fileNames) {
    // Read that file's content
    readFile(fileName, 'utf8', (err, content) => {
      // Count the characters and store it in
      // characterCounts
      countCharacters(characterCounts, content);
    });
  }
});
```

At this point, you feel pretty good. There's only one thing left to do: print out the total of all the characters in the files. So, where do you put that `console.log` statement?

This is kind of a trick question because there's no place to put it in the way the code works now.

If you put it here:

```
readFile('manifest.txt', 'utf8', (err, manifest) => {
  const fileNames = manifest.split('\n');
  const characterCounts = {};

  // Loop over each file name
  for (let fileName of fileNames) {
    // Read that file's content
    readFile(fileName, 'utf8', (err, content) => {
      // Count the characters and store it in
      // characterCounts
      countCharacters(characterCounts, content);
    });
  }

  // MY PRINT STATEMENT HERE
  console.log(characterCounts);
});
```

then you will get the output `{}` every time because the code in the inner `readFile`s doesn't run until after the `console.log` because `readFile` doesn't run the function with the arguments `(err, content)` until *after* the file is read and the current function completes.

If you put it here:

```
readFile('manifest.txt', 'utf8', (err, manifest) => {
  const fileNames = manifest.split('\n');
  const characterCounts = {};

  // Loop over each file name
  for (let fileName of fileNames) {
    // Read that file's content
    readFile(fileName, 'utf8', (err, content) => {
      // Count the characters and store it in
      // characterCounts
      countCharacters(characterCounts, content);

      // MY PRINT STATEMENT HERE
      console.log(characterCounts);
    });
  }
});
```

then it will print the number of times that your code reads a file. That's not what you want, either. To get it to work, you have to count the number of files that have been read each time one completes. Then, you only print when that number equals the total number of files to be read. The code could like this:

```
readFile('manifest.txt', 'utf8', (err, manifest) => {
  const fileNames = manifest.split('\n');
  const characterCounts = {};
  let numberOfFilesRead = 0;

  // Loop over each file name
  for (let fileName of fileNames) {
    // Read that file's content
    readFile(fileName, 'utf8', (err, content) => {
      // Count the characters and store it in
      // characterCounts
      countCharacters(characterCounts, content);

      // Increment the number of files read
      numberOfFilesRead += 1;
```

```
      // If the number of files read is equal to the
      // number of files to read, then print because
      // you're done!
      if (numberOfFilesRead === fileNames.length) {
        console.log(characterCounts);
      }
    });
  }
});
```

The asynchronous nature of this code requires you to do a lot of housekeeping just to figure out when everything is done. Imagine writing this code and going back to it in six months to add a new feature. It's not the clearest code in the world, even with code comments. That leads to a maintenance nightmare. The JavaScript community wanted a way to code better and clearer.

Designing a better solution

Look at the following code that has numbers in the order in which the `console.log` statements are run. It will print out "Q", "W", "E", "R", and "T" on separate lines.†

```
console.log('Q'); //---- 1
setTimeout(() => {
  console.log('E'); //-- 3
  setTimeout(() => {
    console.log('T'); // 5
  }, 100);
  console.log('R'); //-- 4
}, 200);
console.log('W'); //---- 2
```

What would really help is if you could get those numbers in order so that what appears in the code at least **appears** to be synchronous even though it might be asynchronous in nature. As humans, we understand things from top-to-bottom much better than in the order 1, 3, 5, 4, 2.

Reordering the code above to reflect how it really runs, you'd get this somewhat more maintainable block.

```
console.log('Q'); //---- 1
console.log('W'); //---- 2
setTimeout(() => {
  console.log('E'); //-- 3
  console.log('R'); //-- 4
  setTimeout(() => {
    console.log('T'); // 5
  }, 100);
}, 200);
```

But, now you're stuck with those human-necessary indents to understand the function calls that occur in the code. And, to know how long the `setTimeout`s run, you have to go way to the bottom of the code blocks. The JavaScript community agreed with you and decided it'd be great if they could somehow just chain a bunch of those things together without the indentation, something like this. (The function names are completely invented for this code block.);

```
log('Q')
  .then(() => log('W'))
  .then(() => pause(200))
  .then(() => log('E'))
  .then(() => log('R'))
  .then(() => pause(100))
  .then(() => log('T'));
```

The JavaScript community realized that they'd have to use functions in the `then` blocks lest the function be immediately invoked. Remember, a

function declaration is not invoked when interpreted. That means each function in each of the `then` calls is passed into the `then` function for it to run at a later time, presumably when the previous thing finishes, a previous `log` or `pause` in this example. They decided to create a new kind of abstraction in JavaScript named the "Promise".

So, what is a "Promise"?

Look at a line of code using the `readFile` method found in Node.js. Don't worry if you don't know the specifics about this function. It's the *form* of the code to which you should draw your attention.

```
readFile('manifest.txt', 'utf8', (err, manifest) => {
```

You could interpret that line of code as "Read the file named "manifest.txt" and, when done, call the method that is declared with `(err, manifest) => {`.

The important part to understand is the "when done, call the method...". That's the part that's potentially asynchronous, the part that is beyond your control. When it calls that function, it will either provide an error in the `err` parameter or a value in the `manifest` parameter. When you change it to the `then` version, you still get the same kind of guarantee: eventually, you will get an error or the value of the operation. So that's what a `Promise` is.

*A `Promise` in JavaScript is 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 Web site) or your code will get **an error** from that operation (like the file doesn't exist or the Web site is down).*

Promises can exist in three states. They are:

- **Pending**: The `Promise` object has not resolved. Once it does, the state of the `Promise` object may transition to either the fulfilled or rejected state.
- **Fulfilled**: Whatever operation the `Promise` represented succeeded and your success handler will get called. Now that it's *fulfilled*, the `Promise`:
  - must not transition to any other state.
  - must have a value, which must not change.
- **Rejected**: Whatever operation the `Promise` represented failed and your error handler will get called. Now that it's *rejected*, the `Promise`:
  - must not transition to any other state.
  - must have a reason, which must not change.

`Promise` objects have the following methods available on them so that you can handle the state change from *pending* to either *fulfilled* or *rejected*.

- `then(successHandler, errorHandler)` is a way to handle a `Promise` when it leaves the *pending* state.
- `catch(errorHandler)`

The handlers mentioned in the previous list are:

- **Success Handler** is a function that has one parameter, the value that a *fulfilled* `Promise` has.
- **Error Handler** is a function that has one parameter, the reason that the `Promise` failed.

We'll elaborate on these methods in part two of this article.

What you've learned

In this reading, you learned some fancy new things that let's you turn asynchronous code into seemingly synchronous-looking code. You did that by

learning that...

- A `Promise` in JavaScript is 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 Web site) or your code will get **an error** from that operation (like the file doesn't exist or the Web site is down).

†: One can argue that the code following this statement is already very bad and shouldn't be written that way. I would agree. Please don't write code like that. It is *only* for demonstration purposes. However, do not be surprised if you find **someone else** wrote code like that. ;-)

A Promise is a Promise: A Mostly Complete Guide to JavaScript Promises II

This is part two of an article about classic JavaScript promises. If you have not read part one, we recommend that you navigate to the previous task to do so.

When you finish this article, you should be able to:

- Create your own `Promise`s
- Use `Promise`objects returned by language and framework libraries

Handling success with `then`

Returning to another file-reading example, consider the following block of code.

```
readFile("manifest.txt", "utf8", (err, manifest) => {
  if (err) {
    console.error("Badness happened", err);
  } else {
    const fileList = manifest.split("\n");
    console.log("Reading", fileList.length, "files");
  }
});
```

If this succeeds, then you would expect a statement like "Reading 12 files" to appear if the file contained a list of 12 files.

Now, to rewrite that using a `Promise`and printing that same statement, you would get a file-reading function that returns a `Promise`object. Later on, you'll see how to create one for yourself. At this moment, just presume that a function named `readFilePromise`exists. When you call it, it would return a promise that, when *fulfilled*, would invoke the success handler registered for the object through the `then`method. Very explicitly, you could write that code like this.

```
/* EXPLICIT CODE: NOT FOR REAL USE */

// Declare a function that will handle the content of
// the file read by readFilePromise.
function readFileSuccessHandler(manifest) {
  const fileList = manifest.split("\n");
  console.log("Reading", fileList.length, "files");
}

// Get a promise that will return the contents of the
// file.
const filePromise = readFilePromise("manifest.txt");

// Register a success handler to process the contents
// of the file. In this case, it is the function
// defined above.
filePromise.then(readFileSuccessHandler);
```

Most `Promise`-based code does **not**look like that, though. Idiomatic JavaScript instructs to not create variables that don't need to be created. You would see the above code in a real-live code base written like this, instead. Spend a moment comparing and contrasting the forms from **very explicit**to**idiomatic**.

```
readFilePromise("manifest.txt").then(manifest => {
  const fileList = manifest.split("\n");
  console.log("Reading", fileList.length, "files");
});
```

That's slightly easier to read than the weird callback thing you had above. But, you still have that nasty double indentation. The designers of the `Promise` didn't want that for you, so they allow you to chain `then`s.

Chaining `then`s.

In the above code that uses `readFilePromise`, it does not look like the ideal code that JavaScript could give us because of the success-handling function being on multiple lines that require another indent. It may be a little thing, but it still prevents you from the most readable code. Again, the Technical Committee 39 had your back. They designed "chainable thens" for you. The rules are a little complex to read.

- Each `Promise` has a `then` method that handles what happens when the `Promise` transitions out of the **pending** state.
- Each `then` method returns a `Promise` that transitions out of its **pending** state when the `then` that created it completes.
- (One more condition described below.)

That chaining property gives you the ability to break apart the two lines of the success handler in the previous example to two one-line functions that do the same thing with less code! If you write that form explicitly, you'd have the following.

```
/* EXPLICIT CODE: NOT FOR REAL USE */

// Get a Promise that fulfills when the file is read
// with the value of the content of the file.
const filePromise = readFilePromise("manifest.txt");

// Register a success handler that takes the fulfilled
// value of the filePromise in the parameter named "manifest",
// which is the content of the file, split it on newline
// characters, and return a Promise whose fulfilled value is
// list of lines.
const fileListPromise = filePromise.then(manifest => manifest.split("\n"));

// Register a success handler to the fileListPromise that
// receives the fulfilled value in the "fileList" parameter
// and returns a Promise whose fulfilled value is the length
// of the fileList array.
const lengthPromise = fileListPromise.then(fileList => fileList.length);

// Register a success handler to the lengthPromise that
// receives the fulfilled value in the "numberOfFiles" parameter
// and uses it to print the number of files to be read.
lengthPromise.then(numberOfFiles =>
  console.log("Reading", numberOfFiles, "files")
);
```

That code block has a lot of words to describe what happens at each step of the process of using "chainable thens". In the real world, were you to find that code in a real application, it would likely look like the following.

```
readFilePromise("manifest.txt")
  .then(manifest => manifest.split("\n"))
  .then(fileList => fileList.length)
  .then(numberOfFiles => console.log("Reading", numberOfFiles, "files"));
```

Here's a diagram of what happens in the above code.

You can see that each call to `then` creates a new `Promise` object that resolves to the value of the output of the previous success handler. That's what happens when everything works out. What happens when it doesn't?

Handling failure with `then`

As you may recall from the section So, what is a "Promise"?, you learned that the `then` method can also accept a second argument that is an error handler that takes care of things should something go wrong. Back to the file reading example from above, you add a second method to the `then` which accepts a **reason** that the error happened. For reading a file, that could be that the file doesn't exist, the current user doesn't have permissions to read it, or it ran out of memory trying to read a *huge* file.

```
readFilePromise("manifest.txt").then(
  manifest => {
    const fileList = manifest.split("\n");
    console.log("Reading", fileList.length, "files");
  },
  reason => {
    console.error("Badness happened", reason);
  }
);
```

That works, but has taken you back to the original bad multiline form of the success handler. What happens if you did it like this? How does this work?

```
readFilePromise("manifest.txt")
  .then(
    manifest => manifest.split("\n"),
    reason => console.err("Badness happened", reason)
  )
  .then(fileList => fileList.length)
  .then(numberOfFiles => console.log("Reading", numberOfFiles, "files"));
```

Here's what happens with regard to the `Promise`s in this chain of `then`s.

As you can see, the first `Promise` object from the `readFilePromise` function goes into the **rejected** state because, according to the error message, the file didn't exist at the time the system tried to read it. That reason is represented as an object that has a code of "ENOENT" which a Unix error code and a message that provides a human-readable explanation of the error. That error reason object gets passed to the error handler of the first `then`. It's what happens after that that is crazy neat.

The second `Promise` object is **fulfilled**! Because the first `then` doesn't have any errors, because the error handler in the first then completes without any problem (printing out the error reason), the `Promise` returned by that `then` *does not* get **rejected**. Because of that, the `Promise` resolves with the value returned by the `console.error('Badness happened', err)` call. The `console.error` method returns `undefined`, so that becomes the value passed into the next `then` handler.

Because the second `then` success handler relies on an object with a `length` property, when it runs, an exception gets raised because the `undefined` value has no `length` property. This causes the `Promise` returned by the second `then` to become **rejected** because the code threw an exception.

Because that `Promise` is in the **rejected** state, it attempts to run the error handler of the next (third) `then`. There is no error handler. In the browser, it just looks like nothing happened. In Node.js, an `UnhandledPromiseRejectionWarning` is emitted to the console. In a future version of Node.js, it will cause the process to terminate with an exit code indicating an error bringing your service to a halt.

To correctly handle the exception of no file to read and still have all of the other lines of code run properly, you should write the following code.

```
readFilePromise("manifest.txt")
  .then(manifest => manifest.split("\n"))
  .then(fileList => fileList.length)
  .then(
    numberOfFiles => console.log("Reading", numberOfFiles, "files"),
    reason => console.err("Badness happened", reason)
  );
```

Now, if an error occurs, the chain of `then`s evaluates like this:

1. First `then`: I do not have an error handler. I will pass the error on and not run the success handler.
2. Second `then`: I do not have an error handler. I will pass the error on and not run the success handler.
3. Third `then`: I have an error handler and will run it.

Now, the code looks almost like you'd imagined back in the Designing a better solution section. There's still that annoying last double handler code that makes us use indentation and passing in two function objects to a `then` which looks kind of yucky. The Technical Committee gave you a solution for that, too.

*`then` can handle both success and failures. The success handler is called with the value of the operation of the `Promise` when the `Promise` object transitions to the **fulfilled** state. If an error condition occurs, them the error handler of the `then` is called.*

*If a `Promise` object transitions to the **rejected** state and no error handler exists for the `then`, then that `then` is skipped altogether.*

*If an error handler is called and does not raise an exception, then the next `Promise` object transitions to the **fulfilled** state and the next success handler is called.*

Handling failure with `catch`

Rather than using a `then` with a success and error handler, you can use the similar `catch` method that takes just an error handler. By doing that, the code from the last section ends up looking like this.

```
readFilePromise("manifest.txt")
  .then(manifest => manifest.split("\n"))
  .then(fileList => fileList.length)
  .then(numberOfFiles => console.log("Reading", numberOfFiles, "files"))
  .catch(reason => console.err("Badness happened", reason));
```

That is exactly what the design expressed. The `catch` acts just like an error handler in the last `then`. If the `catch` doesn't throw an exception, then it returns a `Promise` in a fulfilled state with whatever the return value is, just like the error handler of a `then`.

`catch` is a convenient way to do error handling in a `then` chain that looks kind of like part of a try/catch block.

Using `Promise.all` for many future values

You're almost to the place where you can read the manifest file, get the list, and then count the characters in each of the files, and print out the result. You need to learn about two more features of JavaScript `Promise`s.

Imagine that you have three files that you want to read with the `readFilePromise` method. You want to wait until all three are done, but let them read files simultaneously. How do you manage all three `Promise`s as one `Promise`? That's what the `Promise.all` method allows you to do.

For example, imagine you have the following array.

```
const values = [
  readFilePromise("file-boop.txt"), // this is a Promise object: pending
  readFilePromise("file-doop.txt"), // this is a Promise object: pending
  readFilePromise("file-goop.txt"), // this is a Promise object: pending
];
```

When you pass that array into `Promise.all`, it returns a `Promise` object that manages all of the `Promise`s in the array!

```
const superPromise = Promise.all(values);
// superPromise is a Promise object in the pending state.
//
// Inside superPromise is an array of Promise objects
// that look like this:
//
// 1. file reading promise in pending state, same as the one passed in
// 2. file reading promise in pending state, same as the one passed in
// 3. file reading promise in pending state, same as the one passed in
```

When all of the `Promise` objects in the super `Promise` transition out of the pending state, then the super `Promise` will also transition out of the pending state. If any one of the `Promise` objects in the array transition to the **rejected** state, then the super `Promise` will immediately transition to the **rejected** state with the same reason as the inner `Promise` failed with. If *all* of the internal `Promise` objects transition to the **fulfilled** state, then the super `Promise` will transition to the **fulfilled** state and its value will be an array of *all* of the resolved values of the original array.

With that in mind, you could continue the above code with a `then` and a `catch` that would demonstrate what happens.

```
superPromise
  .then(values => console.log(values))
  .catch(reason => console.error(reason));
```

```
// If the function successfully reads the file, the values passed
// to the then come from the values that were in the superPromise
//
// 1. the content of file-boop.txt
// 2. the content of file-doop.txt
// 3. the content of file-goop.txt

// If something goes wrong with reading the file, then the `catch`
// gets called with the error reason from the Promise object that
// first failed.
```

*`Promise.all` accepts an array of values and returns a new `Promise` object in the **pending** state colloquially called a "super promise". It converts all non-`Promise` values into `Promise` objects that are immediately in the **fulfilled** state. Then,*

- *If any one of the `Promise`s in the array transitions to the **rejected** state, then the "super promise" transitions to the **rejected** state with the same reason that the inner `Promise` object failed.*
- *If all of the inner `Promise` objects in the array transition to the **fulfilled** state, then the "super promise" transitions to the **fulfilled** state with a value of an array populated, in order, of the resolved values of the original array.*

Flattening `Promises`

The last thing you need to learn about `Promise`s is the coolest feature of them all. If you return a `Promise` object from either a success or error handler, the next step doesn't get run until that `Promise` object resolves! Here's what happens when you type the following code. It's step 4 that is the amazing part.

```
readFilePromise("manifest.txt")
  .then(manifestContent => manifestContent.split("\n"))
  .then(manifestList => manifestList[0])
  .then(fileName => readFilePromise(fileName))
  .then(otherFileContent => console.log(otherFileContent));

// Interpreted as:
// 1. Read the file of the manifest.txt file and pass the
//    content to the first then.
// 2. Split the content from manifest.txt on newline chars
//    to get the full list of files.
// 3. Return just the first entry in the list of files.
// 4. RETURN A PROMISE THAT WILL READ THE FILE NAMED ON THE
//    FIRST LINE OF THE manifest.txt! The next then method
//    doesn't get called until this Promise object completes!
// 5. Get the content of the file just read and print it.
```

Again, here's the rule.

If you return a `Promise` from a success or error handler, the next handler isn't called until that `Promise` completes.

Putting it all together

You can now use all of this knowledge to use `Promise`s to read a manifest file, read each of the files in the manifest files, and count all of the characters in those files with code that reads much better than this.

```
readFile("manifest.txt", "utf8", (err, manifest) => {
  const fileNames = manifest.split("\n");
  const characterCounts = {};
  let numberOfFilesRead = 0;

  // Loop over each file name
  for (let fileName of fileNames) {
    // Read that file's content
    readFile(fileName, "utf8", (err, content) => {
      // Count the characters and store it in
      // characterCounts
      countCharacters(characterCounts, content);

      // Increment the number of files read
      numberOfFilesRead += 1;

      // If the number of files read is equal to the
      // number of files to read, then print because
      // we're done!
      if (numberOfFilesRead === fileNames.length) {
        console.log(characterCounts);
      }
    });
  }
});
```

Remember that you've created a `countCharacters` methods elsewhere that does the grunt work of counting characters. So, now, if you were to list out the steps that you'd like to have the code perform, you should be able to write a `Promise`-based chain of `then`s that does that work.

1. Read `manifest.txt`.
2. Split the content into a list of files.
3. Read the contents of each file.
4. If all of them succeed, then
    - count the characters in each file and
    - print the character counts.
5. If anything fails, print the error.

So, in code, that you would translate that to the following.

```
const characterCounts = {};
readFilePromise('manifest.txt')
  .then(fileContent => fileContent.split('\n'))
  .then(fileList => fileList.map(fileName => readFilePromise(fileName)))
  .then(lotsOfReadFilePromises => Promise.all(lotsOfReadFilePromises))
  .then(contentsArray => contentsArray.forEach(c => countCharacters(characterCour
  .then(() => console.log(characterCounts))
  .catch(reason => console.error(reason));
```

Through the magic of `Promise`s, you have now been able to do lots of asynchronous work but make it look synchronous!

Creating your own `Promise`s

Early on, you designed the way `Promise`s should work to look something like this.

```
log("Q")
  .then(() => log("W"))
  .then(() => pause(2))
  .then(() => log("E"))
  .then(() => log("R"))
  .then(() => pause(1))
  .then(() => log("T"));
```

That code uses two functions that you can define:

- a `log` function that takes a value to print and returns a `Promise` object that is in the **fulfilled** state; and,
- a `pause` function that takes a number and returns a `Promise` object that, after the indicated number of seconds, transitions to the **fulfilled** state.

Here is a way that you could create those functions.

```
function log(message) {
  console.log(message);
  return Promise.resolve();
}
```

The above function logs the message passed to it and, then creates a `Promise` object already transitioned to the **fulfilled** state. If you provide a value to the resolve method, then that becomes the value of the `Promise` object.†

The `pause` method is a little more difficult. You have to create a new `Promise` object from scratch to pause and then continue. To do that, you will use the `Promise` constructor.

The `Promise` constructor accepts a function that has two parameters. Each of those parameters will be functions, themselves. The first parameter is the so-called **resolve** parameter which, when called, transitions the `Promise` object to the **fulfilled** state. The second parameter is the so-called **reject** parameter which, when called, transitions the `Promise` object to the **rejected** state.

```
function pause(numberOfSeconds) {
  return new Promise((resolve, reject) => {
    setTimeout(() => resolve(), numberOfSeconds * 1000);
  });
}
```

As you can see from the above code, the `new Promise` gets a single argument, a two-parameter function that does some asynchronous thing. The two parameters are the **resolve** and the **reject** functions that you can use to transition the state of the `Promise` object being constructed. In this case, after a certain amount of time, the `resolve()` method is invoked which transitions

the `Promise` object to the **fulfilled** state. The value is `undefined` because you've passed no value into the `resolve()` function invocation. If you wanted the `Promise` to have the value of 6.28, then you would invoke it like this `resolve(6.28)`. You can pass any one value into the `resolve` function, be it a number, a boolean, an array, an object, or whatever.

With that knowledge, think about how you would write a function using the `readFile` function that would return a `Promise` object that would resolve to the contents of the file on success and reject the `Promise` if an error occurred. Take a moment to scratch that out into an editor or something.

If you wrote something similar to the following, then you did a great job! If you didn't, work through the following in a Node.js JavaScript environment to figure out how it works. You can use it like in any of the above examples.

```
const { readFile } = require("fs"); // This is just the way to get
// the readFile method into the
// current file. If you don't
// understand it, that's ok.

function readFilePromise(path) {
  return new Promise((resolve, reject) => {
    readFile(path, "utf8", (err, content) => {
      if (err) {
        reject(err);
      } else {
        resolve(content);
      }
    });
  });
}
```

What you've learned

In this reading, you learned some fancy new things that let's you turn asynchronous code into seemingly synchronous-looking code. You did that by learning that...

- `then` can handle both success and failures. The success handler is called with the value of the operation of the `Promise` when the `Promise` object transitions to the **fulfilled** state. If an error condition occurs, them the error handler of the `then` is called.
- If a `Promise` object transitions to the **rejected** state and no error handler exists for the `then`, then that `then` is skipped altogether.
- If an error handler is called and does not raise an exception, then the next `Promise` object transitions to the **fulfilled** state and the next success handler is called.
- `catch` is a convenient way to do error handling in a `then` chain that looks kind of like part of a try/catch block.
- `Promise.all` accepts an array of values and returns a new `Promise` object in the **pending** state colloquially called a "super promise". It converts all non-`Promise` values into `Promise` objects that are immediately in the **fulfilled** state. Then,
  - If any one of the `Promise`s in the array transitions to the **rejected** state, then the "super promise" transitions to the **rejected** state with the same reason that the inner `Promise` object failed.
  - If all of the inner `Promise` objects in the array transition to the **fulfilled** state, then the "super promise" transitions to the **fulfilled** state with a value of an array populated, in order, of the resolved values of the original array.
- If you return a `Promise` from a success or error handler, the next handler isn't called until that `Promise` completes.
- You can create a **fulfilled** `Promise` object by using the `Promise.resolve(value)` method.
- You can create your own `Promise` objects from scratch by using the `Promise` constructor with the form

```
new Promise((resolve, reject) => {
  // do some async stuff
  // call resolve(value) to make the Promise succeed
```

```
  // call reject(reason) to make the Promise fail
});
```

See also

- Section: Promises, ECMAScript® 2015 Language Specification is the minimum standard for how JavaScript Promises should act in **all** JavaScript environments. Language standards are dense and hard to read. You may want to give it a shot. The more you grow in your knowledge of how JavaScript works, the clearer it should become.
- The Promises/A+ Specification has a very nice terse description of how Promises work. It is mostly the standard that was adopted by the Technical Committee 39 when including the `Promise` object into JavaScript.

†: There's a corresponding `Promise.reject(reason)` method that creates a `Promise` object immediately in the **rejected** state.

Promises Lesson Learning Objectives II

Below is a complete list of the terminal learning objectives for this lesson. When you complete this lesson, you should be able to perform each of the following objectives. These objectives capture how you may be evaluated on the assessment for this lesson.

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

HTML Learning Objectives

The objective of this lessonis for you to know how to effectively use HTML5 to build semantically and structurally correct Web pages. HTML is the language that renders the cross-platform human-computer interfaces that made the World Wide Web accessible by the world! 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
- Itemized list tags
    - ul
    - ol
    - li
- a
- img
- Tabular-data tags
    - table
    - thead
    - tbody
    - tfoot
    - tr
    - th
    - td

Modern `Promise`s With `async` And `await`

While `Promise`s helped revolutionize the way that JavaScript programmers could structure asynchronous code, the technical committee that governs JavaScript realized it could take this feature one step further. It could design a language feature that allowed programmers to write true synchronous code based on `Promise`s. Thus, the `async` and `await` keywords came into being in 2017.

When you finish this article, you should be able to:

- Properly implement a function declaration with the `async function` expression;
- Explain how the JavaScript runtime creates an implicit `Promise` for `async function` declarations; and,
- Execute functions marked with the `async` keyword using the `await` keyword.

Classic promise example

Let's review with an example of classic promise handling using two functions. `walkTheDog` will return a promise that resolves with a `'happy dog'` after 1 second. `doChores` will act as our main function and will handle that promise with a `then`:

```
function walkTheDog() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('happy dog');
    }, 1000);
  });
}

function doChores() {
  console.log('before walking the dog');
  walkTheDog()
    .then(res => {
        console.log(res);
        console.log('after walking the dog');
    });
  return 'done';
}

console.log(doChores());

// prints:
//
// before walking the dog
// done
// happy dog
// after walking the dog
```

Notice that `'done'` will be returned by `doChores` before the promise resolves, because it is asynchronous.

If we want to take any actions after the promise resolves, we can do so by chaining `then`. This code works, but we may have one complaint regarding aesthetics: there is some added bulk because the `then` accepts a callback containing the code we want to execute after the promise resolves. This bulk compounds further if we want to chain multiple `then`s. Let's refactor this. Enter `async` and `await`.

`async` function declarations

Declaring a function with `async` will create the function so it returns an implicit promise containing it's result. Let's declare our `doChores` function as `async` and check it's return value. For now we'll leave out the explicit `walkTheDog` promise:

```
async function doChores() {
  // ...
  return 'done';
}

console.log(doChores());
// prints:
// Promise { 'done' }
```

This function now returns a promise automatically! Notice that the promise returned contains the immediately resolved value of `'done'`.
An `async` declaration isn't super useful by itself. However, it allows us to utilize the `await` keyword inside the function.

`await`ing a promise

The await operator can be used to wait for promise to be fulfilled. We are only allowed to use `await` in an `async` function. Using `await` outside of an `async` will result in a SyntaxError. When a promise is `await`ed, execution of the containing `async` function will pause until the promise is fulfilled.

Let's use `await` in our `doChores` function:

```
function walkTheDog() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('happy dog');
    }, 1000);
  });
}

async function doChores() {
  console.log('before walking the dog');
  const res = await walkTheDog();
  console.log(res);
  console.log('after walking the dog');
}

doChores();
// prints:
// before walking the dog
// happy dog
// after walking the dog
```

Whoa! This code looks synchronous. Instead of using `then`, we can `await` the `walkTheDog()` promise, pausing execution until the promise is fulfilled. Once fulfilled, the `await` expression will evaluate to the `resolve`d value, `'happy dog'` in this case.

Remember that the `async doChores` function will implicitly return a promise. Now that promise will fulfill once the entire function is finished executing. The function's return value will be the resolved value of the implicit promise. Let's handle it with `then`:

```
function walkTheDog() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('happy dog');
    }, 1000);
  });
}

async function doChores() {
  console.log('before walking the dog');
  const res = await walkTheDog();
  console.log('after walking the dog');
  return res.toUpperCase();
}

doChores().then(result => console.log(result));
```

```
// prints:
// before walking the dog
// after walking the dog
// HAPPY DOG
```

You're probably wondering why we chain `then` and not simply use `await doChores()`, that's because we can only use `await` *inside* of an `async` function. Currently our call to `doChores` is not within any function.

For fun, let's use a surrounding `async` function to `await doChores()`. We'll also add some numbered print statements to show the order of execution:

```
function walkTheDog() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log('2');
      resolve('happy dog');
    }, 1000);
  });
}

async function doChores() {
  console.log('1');
  const res = await walkTheDog();
  console.log('3');
  return res.toUpperCase();
}

async function wrapper() {
  console.log('0');
  const finalResult = await doChores();
  console.log('4');
  console.log(finalResult + '!!!');
}

wrapper();
// prints:
// 0
// 1
// 2
// 3
// 4
// HAPPY DOG!!!
```

Refactoring a promise chain

Refactoring a promise chain is straightforward with `async`/`await`. Let's say wanted to print the resolved values for 3 promises in order:

```
function wrapper() {
  promise1
    .then(res1 => {
      console.log(res1);
      return promise2;
    })
    .then(res2 => {
      console.log(res2);
      return promise3;
    })
    .then(res3 => {
      console.log(res3);
    });
}
```

We can refactor it into this:

```
async function wrapper() {
  console.log(await promise1);
  console.log(await promise2);
  console.log(await promise3);
  console.log(await promise4);
}
```

Error handling

Since `async` / `await` allows for seemingly synchronous execution, we can use a normal `try...catch` pattern to handle errors when the promise is rejected:

```
function action() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      reject('uh-oh'); // rejected
    }, 3000);
  });
}

async function handlePromise() {
  try {
    const res = await action();
    console.log('resolved with', res);
  } catch (err) {
    console.log('rejected because of', err);
  }
}

handlePromise();
// prints:
// rejected because of: uh-oh
```

What you learned

In this article, you learned how to use the `async` and `await` keywords in modern JavaScript to truly turn asynchronous code into synchronous style code. You will want to do this to make the code more readable and maintainable. The steps to do this are to

- mark a function with the `async` keyword to make JavaScript have it return a `Promise` if your code doesn't, and
- use the `await` keyword to turn the invocation of a function marked `async` into a blocking call that returns the resolved value of the underlying `Promise` or throws an exception.

The Basics of HTML

The *HyperText Markup Language*(HTML) revolutionized the way that we use computers. Before HTML and Web browsers, all computer applications were so-called "desktop applications" that had to be created by software developers using Win32 C and other such specialized languages. With the advent of HTML, anyone with a text editor and a way to host HTML files could create content that people all over the world could see!

In this reading, you will either get to know HTML or brush up on the basics. At the end, you should know

- The three components that are the building blocks of HTML
- How to add a title to a Web page
- Add some headings to your HTML document
- Add blocks of paragraph text
- Use lists to itemize or define content
- Use hyperlinks to instruct the browser to go from one page (or *resource*) to another
- Put an image in your Web page
- Properly display tabular data
- Put comments in the source of your HTML code

The three components of HTML

HTML has three components that form its basic building blocks: **tags**,**elements**, and **attributes**. Once you've learned the rules for how each of these components function, you should have no trouble writing and editing HTML.

These things called "tags"

Any text that you write inside the angle brackets "<" and ">" will not be displayed in the browser. The text inside the angle brackets is just used to tell the browser how to display or transform regular text located between the opening tag (also called the start tag) and the closing tag (also called the end tag).

Tags usually come in pairs, and the difference between an opening tag and a closing tag is that the first symbol inside the brackets of a closing tag is a slash "/" symbol.

For example, here's a pair of h1 tags (used to identify heading text), with some content in-between:

```
<h1>This is some content.</h1>
```

In that example, the <h1>is the opening tag and the </h1> is the closing tag.

There are a whole mess of tags in HTML for you to use. The ones that you should know because you'll put them to use are in the following list. Go read *each of the following documentation pages*.

- html
- head
- title
- link
- script
- Header tagsThere are six of these. This link is to just "h1".
- p
- article
- section
- main
- nav
- header

- `footer`
- Itemized list tags
  - `ul`
  - `ol`
  - `li`
- `a`
- `img`
- Table tags
  - `table`
  - `thead`
  - `tbody`
  - `tfoot`
  - `tr`
  - `th`
  - `td`

There are two main rules that you need to follow when using tags. So, don't forget them.

1. *You must always use angle brackets for tags.*Square brackets, curly braces, parentheses, none of those are for tags in HTML. Just the "<" and ">".
2. Tags almost *always come in pairs*. This means that, except for a few number of tags, you *must always close a tag after opening it*. If you forget to add a closing tag, sometimes the browser will kindly figure it out and insert one for you when it renders your content. However, *don't rely on that behavior*! Different browsers will do different things. Put the closing tag when you are supposed to put it.

There's also a general guideline for writing tags: *please write them in lower case letters*. Sure, you can write them with upper case letters. The browsers will accept "img" and "IMG" and "ImG" and "iMg" as the image tag. However, for consistency, convention, and clarity, please choose lower case.

These things called "elements"

You now know that most tags come in pairs, and some tags don't have a closing tag. An **HTML element**is defined as

- If a tag is supposed to have both an opening tag and a closing tag, then when you refer to it as an *HTML element*, you actually mean:
  - The opening tag
  - The closing tag
  - All of the content between the opening and closing tags
- If a tag is *not*supposed to have a closing tag, then when you refer to it as an *HTML element*, you mean just the tag itself.

Here's an example of how to specify a title in an HTML document.

```
<title>Pictures of Barry's Beautiful Baby</title>
```

The *HTML element*is the opening tag (`<title>`), the closing tag (`</title>`), and the content inside the tags ("Pictures of Barry's Beautiful Baby").

Here's an example of how you can show an image in an HTML document.

```
<img src="./images/baby-bess-bouncing-backwards.jpg">
```

Because images don't have closing tags, the *HTML element*is everything from`<img` to the `>`. Tags that don't have closing tags are called **empty tags**.

In some examples you find on the Internet, you are going to see empty tags with a weird slash at the end like this.

```
<!-- This is bad code with the slash -->
<img src="./images/baby-bess-bouncing-backwards.jpg" />
```

That is *OLD SYNTAX* and should not be used unless you are working on an old Web site where *all* of the empty elements have that syntax. (It is from an old standard called "XHTML", an abomination if ever there was one on the face of the World Wide Web.)

These things called "attributes"

Attributes are used to define additional information about an element. They are located inside the opening tag, and usually come in name/value pairs (name= “value”).

All HTML elements can have attributes, but for most elements, we only use them when we need to. Attributes common to *all* HTML elements are the *class* and *id* attributes that you can use to categorize and identify HTML elements in your HTML document. Of course, the most common reason to use those is so that you can write CSS to style those elements or write JavaScript to manipulate the elements through the document object model.

You may have noticed that the previous example had a name-value pair in it. That is an attribute. Here's the example, again, for convenience.

```
<img src="./images/baby-bess-bouncing-backwards.jpg">
```

The attribute's *name* is "src" and the attribute's *value* is "./images/baby-bess-bouncing-backwards.jpg". The "src" attribute provides the additional information to the browser that this specific image's source file can be found at that path.

There are three main guidelines for using attributes. They're not really rules. However, you should follow them irrespective of the examples you see on the Internet, especially some of the sludge found on Stack Overflow.

1. *Write attributes in lower case only*.You can write them in upper case. HTML tags and attributes are "case insensitive". So, `<img src="...">` is the same to a browser as `<IMG SRC="...">` is the same as `<Img Src="...">`. Just use lower case. Everybody else does it, too. Be cool. Stay in the lower-case HTML school.
2. *Put quotation marks around the value*.This makes it easy to identify. In some cases, you will find it necessary because you'll need to put a space in the value of the attribute.
3. *Those quotations marks, make them double quotes*.HTML will take single quotes as delimiters for attributes. Just don't do it. Double quotes are the convention, even though you'll see plenty of people railing against the convention, proclaiming "It doesn't matter! I like single quotes better!" Let them do what they want. If you don't have any real compelling reason, please use double quotes.

When you put together all of these guidelines, the general way an HTML element should look is

```
<closeabletag attribute="value">Some content</closeabletag>
<noclosetag attribute="value">
```

Whitespace, tags, attributes, and content

When a browser is parsing an HTML document, it ignores whitespace, including line breaks, between the tag name and the attributes. So, the three element declarations are considered the same to the browser. The whitespace that is ignored is called **negligible whitespace**.

```
<tag attr1="value1" attr2="value2" attr3="really-long-attribute-value-that-is-rea

<tag attr1="value1"
     attr2="value2"
     attr3="really-long-attribute-value-that-is-really-long">content</tag>

<tag
```

```
  attr1="value1"
  attr2="value2"
  attr3="really-long-attribute-value-that-is-really-long"
>content</tag>
```

You *cannot*put space between the opening angle bracket and the tag name. This is wrong HTML.

```
<!-- This is NOT HTML. -->
< tag attr1="value">content</tag>
```

Whitespace between the opening tag and the closing tag is _part of the content of the tag. So, the two elements in the following HTML snippet are not the same because the second one has a line break and two spaces before the words and a line break after the words. This kind of whitespace is called **non-negligible whitespace**.

```
<tag attr1="value">Some content</tag>

<tag attr1="value">
  Some content
</tag>
```

What you've learned

You've learned about the three components of HTML documents: tags, elements, and attributes. You've learned about how to write them in the document. You've also gone and read about some of the most commonly used elements. You also know about how to add line breaks in-between attributes to make your document easier to read.

Testing

The objective of this lessonis to ensure that you understand the fundamentals of testing and are capable of reading and solving specs. **This lesson is relevant**to you because good testing is one of the foundations of being a good developer.

When you finish, you should be able to:

- Explain the "red-green-refactor" loop of test-driven development.
- Identify the definitions of SyntaxError, ReferenceError, and TypeError
- Create, modify, and get to pass a suite of Mocha tests
- Use Chai to structure your tests using behavior-driven development principles.
- Use the pre- and post-test hooks provided by Mocha

All About Testing!

In your daily life you have encountered tests before - though school, work, or even through trivia, a test is a way to ensure something is correct. In your programming careers so far you've tested most of your work by hand. Testing one function at a time can be tedious, repetitive, and worst of all, it is a method vulnerable to both false positives and false negatives.

Let's talk about *automated testing* - the how, the what, and most importantly the **why**. The general idea across all testing frameworks is to allow developers to write code that would specify the behavior of a function or module or class. We've reached a point in software development where developers can now run test code against their application code and have confidence that their code will work as intended.

When you finish this reading you should be able to paraphrase the how and why we test as well as how to read automated tests without necessarily knowing the syntax.

Why do we test?

Yes, making sure the dang thing actually works is important. But beyond the obvious, why take the time to write tests?

- *To make sure the dang thing works*

  Yes, that's obvious, but dagnabit, it's important!

- *Increase flexibility & reduce fear (of code)*

  You've written a whole bunch of functionality, multiple other developers have worked on the code, you're deep into the project... And then you realize you have to refactor big chunks of it. Without automated tests, you'll be walking on eggshells, frightened of the codebase and the various landmines that are surely lying in wait.

  With tests, you can aggressively refactor with confidence. If anything breaks, you'll know. And you'll know exactly what the expectations are for the module you're refactoring, so as long as it meets the specs, you're good.

When you are writing automated tests for an application you are writing thespecficationsof how that application should behave. In the software industry automated tests are often called "specs", which is short for the word "specification".

- *Make collaboration easier*

  Complex applications are built by teams of developers. It may be that not all those developers will actually get the chance to talk to one another (they're busy, they may live in different places, some of them may have left the company, new people just joined, it's a huge project, etc.).

  Specs allow teams to have confidence that each module performs a specific task and reduces the need for expensive coordination. The specs themselves become an effective form of communication.

- *Produce documentation*

  If the tests are written well, the tests can serve as documentation for the codebase. Need to know what such and such module does? Check out the specs. This is related to easing collaboration.

How we Test

Testing frameworks vs Assertion libraries

An important distinction to understand is the difference between a *testing framework* and an *assertion library*. The job of a testing framework is to **run** tests and present them to a user. An assertion library is the backbone of any written test - it is the code that we use to **write** our tests. Assertion libraries will do the heavy lifting of comparing and verifying our code. Some testing frameworks will have built in assertion libraries, others will need you to import an assertion library to use.

Mocha

Mocha is a JavaScript *testing framework* that specializes in *running* tests and presenting them in an organized user friendly way. The `Mocha` testing framework is widely used because of it's flexibility. `Mocha` supports a whole variety of different assertion libraries and DSL interfaces for writing tests in the way the best suites the developer.

When writing tests with Mocha we will be using `Mocha`'s DSL (Domain Specific Language). A Domain Specific Language refers to a computer language specialized for a particular purpose - in `Mocha`'s case the DSL has been engineered for providing structure for writing tests. A DSL is it's own language that will usually be familiar but syntactically a little different from the languages you know. That being said you don't have to worry about memorizing every single piece of syntax for writing tests - just get a good grasp of the basics of testing and use the documentation to fill in any knowledge gaps.

You've seen what `Mocha` looks like already because all the specs for your assessments and projects so far have been written utilizing `Mocha` as the testing framework.

We'll be talking more about different assertion libraries a little later when we talk about *writing* tests.

What do we test?

So now that we talked about why we test and what we use to test...what exactly do we test?

Test the public interface

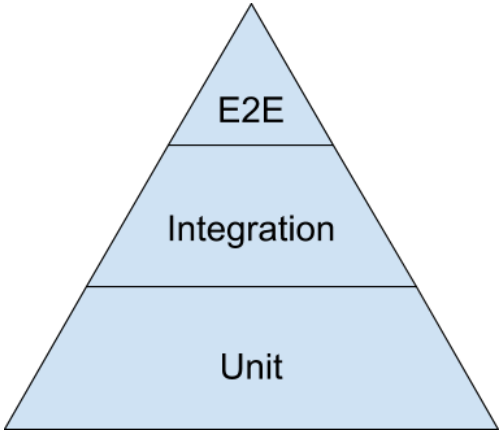
When you're trying to figure out what you should be testing, ask yourself, "What is (or will be) the public interface of the module or class I'm writing?" That is, what are the functions that the outside world will have access to and rely on?

Ideally, you'd have thorough test coverage on the entire public interface. When that's not possible, ensure that your tests cover the most important and/or complex parts of that interface - that is, the pieces that you need to make sure work as intended (and expected).

Kent Dodds has a great article on how to identify what you should be testing.

The testing pyramid

A common metaphor used to group software tests into separate levels of testing is the testing pyramid.



Let's quickly go over each level before talking about the pyramid as a whole:

- **Unit Tests**: The smallest unit of testing - used to test the smallest pieces of your application in isolation to ensure each piece works before you attempt to put those pieces together. Each unit test should focus on testing**one**thing. These are generally the fastest tests to write and run.
- **Integration Tests**: Once you have your unit tests in place you know each piece works in isolation - but what about when those pieces interact with each other? Integration tests are the next level up, they will test the interactions between two pieces of your application. Integration tests will ensure the units you've written work coherently together.
- **End-to-End (E2E) Tests**: End-to-end tests are the highest level of testing - these will test the whole of your application. End-to-end tests are the closest automated tests come to testing the an actual user experience of your application. These are generally the slowest tests to write and run.

For a real life example of how you'd utilize each of these tests imagine coding a Chess game and wanting to test it. Unit tests would be best for testing each class your wrote in isolation - like ensuring each piece's instance methods work as you expect them to before involving them with any other pieces. Next you'd write integration tests - so you'd want to ensure that each piece instance interacted correctly with the `Board` class. The final level would be End-to-End tests which would be like testing a round of chess - testing the `Board`, `Game`, and `Piece` classes all working together.

According to the testing pyramid - you want to have a solid base of a lot of Unit tests, then a medium amount of integration tests built upon that base, then finally a smaller amount of End-to-End tests. Writing tests in this way is practical for a couple of reasons. As we said before, unit tests ensure each piece of your application works in isolation - if you know each piece works then you can more easily find errors. Unit tests are also *fast*. The bigger your application gets the longer your testing suite will take to run - if all your tests are end-to-end tests your tests could be running for **hours**.

Here is a great blog from google about why they use the testing pyramid.

Reading Tests

No matter what kind of test you are encountering the most important thing about a test is that it is **readable** and **understandable**. Good tests use descriptive strings to enumerate what they are testing as well as how they are testing it.

We'll be diving more into the actual syntax of writing tests soon but for right now let's see what you can glean without knowing the syntax:

```
describe("avgValue()", function() {
  it("should return the average of an array of numbers", function() {
    assert.equal(avgValue([10, 20]), 15);
  });
});
```

So without knowing the specfic syntax we can tell a few things from the outset - the outer function has a string with the name of a function `avgValue()` which is most likely the function we will be testing. Next we see a description string `should return the average of an array of numbers`.

So even without understanding the syntax for the test above we can tell *what* we are testing - the `avgValue` function, and how we are testing it - `should return the average of an array of numbers`.

Being able to read tests is an important skill. You'll sometimes find yourself working with unfamiliar testing libraries, but if the test is well written you should be able to determine what the test is doing regardless of the syntax it uses.

Below we've re-written the above example using the Ruby language testing library RSpec:

```
describe "avg_value" do
  it "should return the average of an array of numbers" do
    expect(avg_value([10, 20])).to eq(15)
  end
end
```

Now you probably don't know Ruby - but using the same methods of deduction as we used above we can figure out what is being testing in the above snippet. The outer block mentions `avg_value` which is probably the method or function being tested and the inner block says how things are being tested - `"should return the average of an array of numbers"`. Without knowing the language, or the testing library, we can still figure out generally what what is being tested. That is the important thing about reading tests - having the patience to parse the information before you.

What you learned

We covered a high level overview of testing - the *why*, the *what* and the *how* of testing as well as the basics of how to read a test regardless of the syntax used in writing that test.

Test-Driven Development

At this point of the course you have all encountered an automated test also known as a "spec" - short for specification. In software engineering a collection of automated tests, also known as test suites, are a common way to ensure that when a piece of code is run it will perform the minimum of a specified set of behaviors. We've used the JavaScript testing framework, Mocha, up to this point to test the behavior of functions of all kinds from `myForEach` to `avgValue` to ensure each function runs as intended.

The main question we should be able to answer when writing any piece of code is: what does this code do? How should this code behave? One of the popular ways to answer this question is through a software development process called Test-driven development or TDD. TDD is a quick repetitive cycle that revolves around *first* determining what a piece of code should do and writing tests for that behavior *before actually writing any code*.

Test-driven development dictates that tests, not application code, should be written first, and then application code should only be written to pass the already written tests. When you finish this reading you should be able to identify the three steps of Test Driven Development as well as identify the advantages of using TDD to write code.

Motivations for TDD

Imagine being handed a file of 10 functions that all invoke each other and being told to add a new function to the mix and ensure all the previous functions work properly. First you'd have to figure out what each function actually did, then determine if they did what they were supposed to do. Sounds like a total pain right? A modern web application is thousands of lines of code that are worked on and maintained by teams of developers. Using TDD is one way for developers to ensure that the code written by every member of their team is testable and modular.

Here are some of the biggest motivations for why developers use test-driven development:

1. Writing tests before code ensures that the code written *works*.
    - Code written to pass specs is guaranteed to be testable.
    - Code with pre-written tests easily allows other developers to add and test new code while ensuring nothing else breaks along the way.

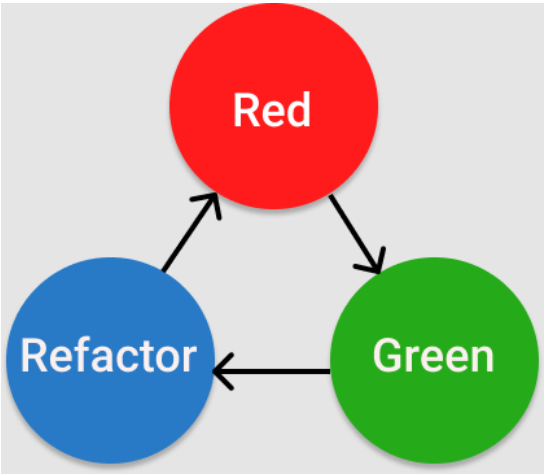
2. Only required code is written.
    - In the face of having to write tests for every piece of added functionality TDD can help reduce bloated un-needed functionality.
    - TDD and YAGNI ("you ain't gonna need it") go hand in hand!

3. TDD helps enforce code modularity.
    - A TDD developer is forced to think about their application in small, testable chunks - this ensures the developer will write each chunk to be modular and capable of individual testing.

4. Better understanding of *what* the code should be doing.
    - Writing tests for a piece of code ensures that the developer writing that code knows what the piece of code is trying to achieve.

Now that we've covered why developers would want to use TDD let's go into *how* to do TDD.

The three steps of TDD: red, green, refactor!



The Test-driven development workflow can be broken down intro three simple steps. **Red, Green, Refactor**:

1. **Red**: Write the tests and watch them fail (a failing test is red). It's important to ensure the tests initially fail so that you don't have false positives.
2. **Green**: Write the minimum amount of code to ensure the tests pass (a passing test will be green).
3. **Refactor**: Refactor the code you just wrote. Your job is not over when the tests pass! One of the most important things you do as a software developer is to ensure the code you write is easy to maintain and read.

Generally, the TDD workflow loop of Red, Green, Refactor is quick. TDD developers will write small tests ensuring each individual part of their application works properly and their code looks good before moving on - making for a short development cycle.

What you learned

TDD stands for test-driven development. TDD is a repetitive process that revolves around three steps: Red, Green, Refactor.

A Comedy of Errors in JavaScript

You know that feeling when you've just finished your perfect function then you go to run your code and: BAM! A big error is thrown? We all have felt that pain from the starting student to the experienced engineer. Runtime errors are a part of daily life when writing code. It is now time to dive into what each type of error you encounter means in order to more quickly and efficiently fix the problem that created that error.

When you finish this reading you should be able to: identify the difference between `SyntaxError`, `ReferenceError`, and `TypeError`s as well as create and `throw` new errors.

JavaScript Errors

In JavaScript the `Error` constructor function is responsible for creating different instances of `Error` objects. The `Error` object is how JavaScript deals with runtime errors and the *type* of error created and thrown will attempt to communicate *why* that error occurred.

Creating your own errors

Since the Error constructor is just a constructor function we can use it to create new `Error` object instances with the following syntax:

```
new Error([message[, fileName[, lineNumber]]])
```

As seen above you can optionally supply a `message`, `fileName` and `lineNumber` where the error occurred.

The `Error` constructor is also somewhat unique in that you can call it with or without the `new` keyword and it will return a new `Error` object:

```
const first = Error("I am an error object!");
const second = new Error("I am too an error object!");

console.log(first); // Error: I am an error object!
console.log(second); // Error: I am too an error object!
```

Let's take a look at what we can do with our newly created `Error` objects.

Throwing your own errors

Tired of JavaScript being the only one to throw errors? Well you can too! Using the keyword `throw` you can throw your own runtime errors that will stop program execution.

Let's take a look at the syntax for `throw`:

```
function giveMeNumber(num) {
  if (typeof num !== "number") {
    throw new Error("Give me a number!");
  } else {
    return "yay number!";
  }
}

console.log(giveMeNumber(1)); // prints "yay number!";
console.log(giveMeNumber("apple")); // Uncaught Error: Give me a number!
console.log(giveMeNumber(1)); // doesn't get run
```

Now as we can see in the above example throwing an error is a powerful tool that stops program execution. If we wanted to throw an error *without* stopping

program execution we can use a `try...catch` block.

Let's look at the syntax for using the `try...catch` block syntax:

```
try {
  // statements that will be attempted to here
} catch (error) {
  // if an error is thrown it will be "caught"
  // allowing the program to continue execution
  // these statements will be run and the program will continue!
}
```

We normally use `try...catch` blocks with functions that might throw an error. Let's look at an example where an error *will not* be thrown:

```
function safeDivide(a, b) {
  if (b === 0) {
    throw new Error("cannot divide by zero");
  } else {
    return a / b;
  }
}

try {
  console.log(safeDivide(30, 5)); // prints 6
} catch (error) {
  console.error(error.name + ": " + error.message);
}

console.log("hello"); // prints hello
```

Note: We can use `console.error` instead of `console.log` to make logged errors more noticeable.

Above you can see our `safeDivide` function ran as expected. Now let's see what happens when an error will be thrown and **caught** inside a `try...catch` block:

```
function safeDivide(a, b) {
  if (b === 0) {
    throw new Error("cannot divide by zero");
  } else {
    return a / b;
  }
}

try {
  console.log(safeDivide(30, 0));
} catch (error) {
  console.error(error.name + ": " + error.message); // Error: cannot divide by ze
}
// the above error will be caught allowing our program to continue!
console.log("hello"); // prints  "hello"
```

Those are the basics of creating and throwing your own errors. You can `throw` your newly created `Error` to stop program execution or use a `try...catch` block to catch your error and continue running your code. Now that we've learned how to create new errors let's go over the core errors built into JavaScript and what they signify.

Types of JavaScript errors

There are seven core errors you'll encounter in JavaScript and each type of error will try to communicate why that error occurred:

1. `SyntaxError` - represents an error in the syntax of the code.
2. `ReferenceError` - represents an error thrown when an invalid reference is made.
3. `TypeError` - represents an error when a variable or parameter is not of a valid type.

4. `RangeError` - representing an error for when a numeric variable or parameter is outside of its valid range.
5. `InternalError` - represents an error in the internal JavaScript engine.
6. `EvalError` - represents an error with the global `eval` function.
7. `URIError` - represents an error that occurs when `encodeURI()` or `decodeURI()` are passed invalid parameters.

For this reading we'll be going in depth of the three most common errors you have encountered so far: `SyntaxError`, `ReferenceError`, and `TypeError`.

SyntaxError

A SyntaxError is thrown when the JavaScript engine attempts to parse code that does not conform to the syntax of the JavaScript language. When learning the JavaScript language this error is a constant companion for any missing `}` or misspelled `function` keywords.

Let's look at a piece of code that would throw a syntax error:

```
funtion broken () { // Uncaught SyntaxError: Unexpected identifier
  console.log("I'm broke")
}
```

Another example with an extra curly brace `}`:

```
function broken () { // Uncaught SyntaxError: Unexpected identifier
  console.log("I'm broke")
}} // Uncaught SyntaxError: Unexpected token '}'
```

The examples go on and on - you can count on a `SyntaxError` to be thrown whenever you attempt to run code that is not syntactically correct JavaScript.

Important! One thing to note about Syntax Errors is that many of them can't be caught using `try` `catch` blocks.

For instance, the following code will throw a `SyntaxError` and no matter how hard you try, you can't catch it.

```
try {
    if (true { // throws "SyntaxError: Unexpected token '{'"
        console.log("SyntaxErrors are the worst!");
    }
} catch (e) {
    console.log(e);
}
```

The missing parenthesis after `true` will throw a `SyntaxError` but can't be caught by the `catch` block.

This is because this kind of `SyntaxError` happens at *compile time* not *run time*. Any errors that happen at *compile time* can't be caught using `try` `catch` blocks.

ReferenceError

Straight from the MDN docs: "The `ReferenceError` object represents an error when a non-existent variable is referenced." This is the error that you'll encounter when attempting to reference a variable that does not exist (either within your current scope or at all).

Let's take a took at some examples for the causes of this error. One common cause for this error is misspelling a variable name:

```
function callPuppy() {
  const puppy = "puppy";
```

```
  console.log(pupy);
}

callPuppy(); // ReferenceError: pupy is not defined
```

Another common cause for a thrown `ReferenceError` is attempting to access a variable that is not in scope:

```
function callPuppy() {
  const puppy = "puppy";
}

console.log(puppy); // ReferenceError: puppy is not defined
```

The aptly named `ReferenceError` will be thrown whenever you attempt to *reference* a variable that doesn't exist.

TypeError

A `TypeError` is commonly thrown for a couple of reasons:

1. When an operation cannot be performed because the operand is a value of the wrong type.
2. When you are attempting to modify a value that cannot be changed.

Let's look at a couple of examples that will each throw a `TypeError` for a different reason. Below we are attempting an operation (in this case a function call) on a value of the wrong type:

```
let dog; // Remember unassigned variables are undefined!

dog(); // TypeError: dog is not a function
```

In the above example we attempt to invoke a declared but not assigned variable (which will evaluate to `undefined`). This will cause a `TypeError` because `undefined` cannot be invoked - it is the wrong type.

Next let's look at a example of attempting to change a value that cannot be changed:

```
const puppy = "puppy";

puppy = "apple"; // TypeError: Assignment to constant variable.
```

Attempting te reassign a `const` declared variable will result in a `TypeError`. You've probably run into many other examples of `TypeError` yourself but, the most important thing to know is that a `TypeError` is thrown when you attempting to perform an operation on the wrong type of value.

Catching known errors

Now that we've covered the the names of common JavaScript errors as well as how to use a `try...catch` block we can combine these two ideas to catch specific types of errors using `instanceof`:

```
function callThatArg(arg) {
  arg(); // this will cause a TypeError because callThatArg is being passed a num
}

try {
  callThatArg(42);
  console.log("call successful"); // this line never executes
} catch (error) {
  if (error instanceof TypeError) {
    console.error(`Wrong Type: ${error.message}`); // prints: Wrong Type: arg is
  } else {
    console.error(error.message); // prints out any errors that aren't TypeErrors
```

```
  }
}

console.log("done"); // prints: done
```

What you learned

If you read an error and know *why* that error is being thrown it'll be much easier to find the cause of the problem! In this reading we went over how to create and throw new `Error`objects as well as the definitions for some of the most common types of errors: `SyntaxError`, `ReferenceError`, and `TypeError`s.

Writing Tests

For weeks you have been using one of JavaScript's most popular test frameworks, `Mocha`, to run tests that ensure a function you've written works as expected. It's time to dive deeper into **how** to write our own tests using `Mocha` as our test framework coupled with Assertion libraries such as the built-in `Assert` module of Node or the `Chai` library.

For the rest of the readings in this section we will be covering how to write tests. These readings will be done in the style of a code-along demo so make sure you follow these in order. When you have finished the next series of reading you should know how to:

- properly format and denote your mocha tests using `describe`, `context` and `it` blocks
- write tests for individual functions as well as writing tests for class instance and static methods
- test that functions will throw certain errors
- use `chai-spies` to test how many times a function has been called
- recognize and utilize the Mocha hooks: `before`, `beforeEach`, `after`, and `afterEach`

In this reading we'll be covering:

- the file structure of testing with `Mocha`
- testing with `Mocha` and Node's built-in `Assert` module

Part Zero: Testing file structure

We find that reading about testing is best understood when you can play around within the functions being tested so for that reason this reading will be in the style of a code along demo. We started this reading by created a directory called `testing-demo` where all the code within this reading will be written.

Let's start off with how to write tests for a basic function. Say we've been handed a directory with a function to test `problems/reverse-string.js`. Below is the named function we'll be testing, `reverseString`, which will intake a string argument and then reverse it:

```
// in testing-demo/problems/reverse-string.js

const reverseString = str => {
  // throws a specific error unless the the incoming arg is a string
  if (typeof str !== "string") {
    throw new TypeError("this function only accepts string args");
  }

  return str
    .split("")
    .reverse()
    .join("");
};

// note this function is being exported!
module.exports = reverseString;
```

How would you go about testing the above function? Let's start by setting up our file system correctly. Whenever you are running tests with `Mocha` the important thing to know is that the `Mocha` CLI will automatically be looking for a directory named `test`.

The created `test` directory's file structure should mirror that of the files you intend to test - with each test file appending the word "spec" to the end of the file name. So for the above example we would create `test/reverse-string-spec.js` which should be on the same level as the `problems` directory.

Our file structure should look like this:

```
testing-demo
  └──
  problems
    └── reverse-string.js
  test
    └── reverse-string-spec.js
```

Take a moment to ensure your file structure looks like the one above and that you've copied and pasted the `reverseString` function into the `reverse-string.js` file. Now that we've ensured our file structure is correct let's write some tests!

Part One: Writing tests with Mocha and Assert

The first step in any testing workflow is initializing our test file. Now let's make a clear distinction before moving forward - `Mocha` is a test framework that specializes in *running* tests and presenting them in an organized user friendly way. The code responsible for actually verifying things for us will come from using an *Assertion Library*. Assertion Libraries will do the heavy lifting of comparing and verifying code while `Mocha` will run those tests and then present them to us.

The tests we'll be writing for this next section will use Node's built-in `Assert` module as our Assertion Library.

So inside of `test/reverse-string-spec.js` at the top of the file we will require the `assert` module and the function we intend to test:

```
const assert = require("assert");
// this is a relative path to the function's location
const reverseString = require("../problems/reverse-string.js");
```

Take a moment to open up the Mocha documentation - it will come in handy as a reference for the syntax we'll be using. The `Mocha` DSL (Domain Specific Language) comes with a few different interfaces or "flavors" of their DSL for our purposes we'll be structuring our tests using the BDD interface.

The `describe` function is an organizational function that accepts a descriptive string and a callback. We'll use the `describe` function to **describe** what we will be testing - in this case the `reverseString` function:

```
// test/reverse-string-spec.js

const assert = require("assert");
const reverseString = require("../problems/reverse-string.js");

describe("reverseString()", function() {});
```

The callback handed to the `describe` function will be where we insert our actual tests. We can now use the `it` function - the `it` function is an organizational function we will use to wrap around each test we write. The `it` function accepts a descriptive string and callback to set up our test:

```
describe ('reverseString()', function () {
  it('should reverse the input string', function () {
    // a test will go here!
  })
}
```

The code written above will serve as a great template for future tests we wish to write. Finally, we can insert the actual test we intend to write within the callback handed to the `it` function. We'll use the assert.strictEqual function which allows you to compare one value with another value. We'll use `assert.strictEqual` to compare two strings - one from our function's result and our expected result which we will we define ourselves:

```
// remember we required the assert module at the top of this file
describe("reverseString()", function() {
  it("should reverse the input string", function() {
    let test = reverseString("hello");
    let result = "olleh";
    // the line below is where the actual test is!
    assert.strictEqual(test, result);
  });
});
```

Now if we run `mocha` in the upper most `testing-demo` directory we will see:

```
reverseString()
  ✓ should reverse the input string


1 passing (5ms)
```

We now have a working spec! Take notice of how `Mocha` structures its response in exactly the way we nested our test. The outer `describe` function's message of `reverseString()` is on the upper level and the inner `it` function's message of `should reverse the input string` is nested within.

Strictly speaking we aren't required to nest our `it` functions within `describe` functions but it is best practice to do so. As you can see yourself - it will make your tests a lot easier to read!

Let's add one more spec for `reverseString`, we'll do this by adding another `it` function within the `describe` callback:

```
describe("reverseString()", function() {
  it("should reverse the input string", function() {
    let test = reverseString("hello");
    let result = "olleh";

    assert.strictEqual(test, result);
  });

  it("should reverse the input string and output the same capitalization", functi
    let test = reverseString("Apple");
    let result = "elppA";

    assert.strictEqual(test, result);
  });
});
```

Running the `mocha` command again will return:

```
reverseString()
  ✓ should reverse the input string
  ✓ should reverse the input string and output the same capitalization


2 passing (11ms)
```

Looking good so far - head to the next reading to learn how to test errors.

Writing Tests

In this reading we'll be covering:

- how to test errors using `Mocha` and `Assert`
- test organization using `context` functions

Part Two: Testing errors

Let's jump right in where we left off! We've written a couple of nice *unit tests* - ensuring that this function works in isolation by testing the input we provided matches the expected output. One aspect of this function is not yet being tested - the error thrown when the argument is not of type `String`:

```
// str is the passed in parameter
if (typeof str !== "string") {
  throw new TypeError("this function only accepts string args");
}
```

Organizing tests

Now the above error actually sets up two *different* scenarios - one where the incoming argument is a string and the second where the incoming argument isn't a string and an error is thrown. We can denote these two different states by adding an additional level of organizational nesting to our tests. You can nest `describe` function callbacks arbitrarily deep - but this quickly becomes unreadable. When nesting, we make use of the `context` function, which is an alias for the `describe` function - the `context` function denotes that we are setting up the **context** for a particular set of tests.

Let's refactor our tests from before with some `context` functions before moving on:

```
describe("reverseString()", function() {
  context("given a string argument", function() {
    it("should reverse the given string", function() {
      let test = reverseString("hello");
      let result = "olleh";

      assert.strictEqual(test, result);
    });

    it("should reverse the given string and output the same capitalization", func
      let test = reverseString("Apple");
      let result = "elppA";
      assert.strictEqual(test, result);
    });
  });

  context("given an argument that is not a string", function() {});
});
```

Running the above test will give us this readable output:

```
reverseString()
  given a string argument
    ✓ should reverse the given string
    ✓ should reverse the given string and output the same capitalization
  given an argument that is not a string

2 passing (11ms)
```

Testing errors

Nice now that we have our `context` functions in place we can work on our second scenario where the incoming argument is *not* a string. When using an assertion library like Node's built in Assert we will have access to many functions that will allow us the flexibility to test all kinds of things. For testing errors using Node's built in `Assert` module we can use the assert.throws function.

Now we'll setup up our `it` function within the `context` function we setup above:

```
context("given an argument that is not a string", function() {
  it("should throw a TypeError when given an argument that is not a string", func
    assert.throws();
  });
});
```

The `assert.throws` function works different from the `assert.strictEqual` function in that it does not compare the return value of a function, but it attempts to invoke a function in order to verify that it will throw a particular error. The `assert.throws` function accepts a function as the first argument, then the error that should be thrown as the second argument with an optional error message as our third argument.

So following that logic, we can test the `TypeError` error thrown by `reverseString` with something like this:

```
context("given an argument that is not a string", function() {
  it("should throw a TypeError when given an argument that is not a string", func
    assert.throws(reverseString(3), TypeError);
  });
});
```

However, when we run the `mocha` command we will get:

```
reverseString()
#  etc.
  given an argument that is not a string
    1) should throw a TypeError when given an argument that is not a string


2 passing (11ms)
1 failing

1) reverseString()
    given an argument that is not a string should throw a TypeError when given ar
    argument that is not a string:
    TypeError: this function only accepts string args
```

We are failing the above spec because we passed the invoked version of the `reverseString` function with a number argument - which as we know will throw a `TypeError` and halt program execution. This is a common mistake made everyday by developers when writing tests. We can get around this by wrapping our error expecting function within another function. This will ensure we can still invoke the `reverseString` function with an argument but not throw the error until `assert.throws` is ready to catch it.

We can also add the explicit error message that `reverseString` throws to make our spec as specific as possible:

```
context("given an argument that is not a string", function() {
  it("should throw a TypeError when given an argument that is not a string", func
    assert.throws(
      function() {
        reverseString(3);
      },
      TypeError,
      "this function only accepts string args"
    );
  });
});
```

Now when we run `mocha` we will see:

```
reverseString()
  given a string argument
    ✓ should reverse the given string
    ✓ should reverse the given string and output the same capitalization
  given an argument that is not a string
    ✓ should throw a TypeError when given an argument that is not a string


3 passing (13ms)
```

Awesome! So we've covered writing unit tests using `describe`, `context`, and `it` functions for organization. We have also covered how to test for equality and thrown errors using Node's built-in assertion library, `Assert`.

Head to the next reading to learn about how to test classes using `Mocha` and another assertion library named `Chai`.

Writing Tests

In this reading we'll be covering:

- how to test classes using `Mocha` and `Chai`

Part Three: Testing classes using Mocha and Chai

Let's expand our knowledge of testing syntax by testing some classes! In order to fully test a class, we'll be looking to test that class's instance and static methods. Create a new file in the `problems` folder - `dog.js`. We'll use the following code for the rest of our tests so make sure to copy it over:

```
// testing-demo/problems/dog.js

class Dog {
  constructor(name) {
    this.name = name;
  }

  bark() {
    return `${this.name} is barking`;
  }

  chainChaseTail(num) {
    if (typeof num !== "number") {
      throw new TypeError("please only use numbers for this function");
    }
    for (let i = 0; i < num; i++) {
      this.chaseTail();
    }
  }

  chaseTail() {
    console.log(`${this.name} is chasing their tail`);
  }

  static cleanDogs(dogs) {
    let cleanDogs = [];
    dogs.forEach(dog => {
      let dogStr = `I cleaned ${dog.name}'s paws.`;
      cleanDogs.push(dogStr);
    });
    return cleanDogs;
  }
}

// ensure to export our class!
module.exports = Dog;
```

To test this class we'll create a new file in our `test` directory - `dog-spec.js` so your file structure should now look like this:

```
testing-demo
  └──
  problems
    └── reverse-string.js
    └── dog.js
  test
    └── reverse-string-spec.js
    └── dog-spec.js
```

Let's now set up our `dog-spec.js` file. For this example we'll get experience using another assertion library named Chai. As you'll soon see, the Chai library comes with a lot more built-in more functionality than Node's `Assert` module.

Now since `Chai` is another external library we'll need to import it in order to use it. We need to run a few commands to first create a `package.json` and then we can import the `chai` library. Start off by running `npm init --y` in the top

level directory (`testing-demo`) to create a `package.json` file. After that is finished you can import the `Chai` library by running `npm install chai`.

Here is what that will look like in your terminal:

```
~ testing-demo $ npm init --y
Wrote to /testing-demo/problems/package.json:

{
  "name": "testing-demo",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "directories": {
    "test": "test"
  },
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}

~ testing-demo $ npm install chai
```

Now that we've installed `Chai` we can set up our test file. Create a new file in the `test` folder named `dog-spec.js` We'll require the `expect` module from Chai for our assertions, import our `Dog` class, and set up our outer `describe` function for testing the `Dog` class:

```
// testing-demo/test/dog-spec.js

// set up chai
const chai = require("chai");
const expect = chai.expect;

// don't forget to import the class you are testing!
const Dog = require("../problems/dog.js");

// our outer describe for the whole Dog class
describe("Dog", function() {});
```

So the first thing we'll generally want to test on classes is their `constructor` functions - we need to make sure new instances have the correct properties and that those properties are being set properly before we can test anything else. For the `Dog` class it looks like a name is accepted on instantiation, so let's test that!

We'll start with a nested describe function within our outer `Dog` describe function:

```
describe("Dog", function() {
  describe("Dog Constructor Function", function() {
    it('should have a "name" property', function() {});
  });
});
```

Now we are using a different assertion library so we'll be working with some different syntax. Open up the Chai Expect documentation for reference, we won't be going into tons of detail into every function we use with Chai because Chai allows for a lot of smaller chainable functions and we know you have lives outide this reading.

The nice thing about Chai is that the chainable functions available will often read like English. Check out the right column of this handy Chai cheatsheet for a quick and easy reference on chainable functions.

We'll start our first spec off by using the property matcher to ensure that a newly instantiated object has a specified property:

```
describe("Dog", function() {
  describe("Dog Constructor Function", function() {
    it('should have a "name" property', function() {
      const layla = new Dog("Layla");
      // all our of chai tests will begin with the expect function
      // .to and .have are Chai chainable functions
      // .property is the matcher we are using
      expect(layla).to.have.property("name");
    });
  });
});
```

Now to test our new spec we can run just the Dog class specs by running `mocha test/dog-spec.js` from our top level directory. Running that command we'll see:

```
Dog
  Dog Constructor Function
    ✓ should have a "name" property


1 passing (8ms)
```

Nice! We tested that the name property exists on a new dog instance. Next, we can make sure our name is set properly with another test:

```
describe("Dog Constructor Function", function() {
  it('should have a "name" property', function() {
    const layla = new Dog("Layla");
    expect(layla).to.have.property("name");
  });

  it('should set the "name" property when a new dog is created', function() {
    const layla = new Dog("Layla");
    // we are using the eql function to compare the value of layla.name
    // with the provided string
    expect(layla.name).to.eql("Layla");
  });
});
```

Running the above using `mocha` we'll see both of our specs passing! Now take extra note of the fact that we are defining the same variable twice using `const` within the above `it` callbacks. This is important to note because it underlines the fact that each of the unit tests you write will have their own scope - meaning that they are each independent of the specs that came before or after them.

Head to the next reading to refactor some of the code we just wrote using `Mocha` hooks!

Writing Tests

This will be the final demo in our writing tests series! In this reading we'll be covering:

- Using `Mocha` Hooks to DRY up testing
- Using `Chai Spies` to "spy" on functions to see how many times they've been invoked

Part Four: Mocha Hooks and Chai Spies

Let's jump right back in. We've written some nice unit tests up to this point:

```
describe("Dog Constructor Function", function() {
  it('should have a "name" property', function() {
    const layla = new Dog("Layla");
    expect(layla).to.have.property("name");
  });

  it('should set the "name" property when a new dog is created', function() {
    const layla = new Dog("Layla");
    // we are using the eql function to compare the value of layla.name
    // with the provided string
    expect(layla.name).to.eql("Layla");
  });
});
```

This is how unit tests are supposed to work, buuuut it will be annoying over time if we have to define a new `Dog` instance in *every single spec*. `Mocha` has some built in functionality to help us with this problem though: Mocha Hooks!

Introducing Mocha Hooks

Mocha Hooks give you a convenient way to do set up prior to running a related group of specs or to do some clean up after running those specs. Using hooks helps to keep your testing code DRY so you don't unnecessarily repeat set up and clean up code within each test.

Mocha Hooks have very descriptive function names and two levels of granularity - before/after each block of tests *or* before/after each test:

1. the hooks `before` and `after` will be invoked either before or after the block of tests is run (depending on which function is used)
2. the hooks `beforeEach` and `afterEach` will be invoked either before or after **each** test (depending on which function is used)

Let's look at a simple example that uses each of the available hooks to log a message to the console. Two placeholder tests are also defined to help demonstrate the differences between the available hooks:

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

describe('Hooks demo', () => {
  before(() => {
    console.log('Before hook...');
  });

  beforeEach(() => {
    console.log('Before each hook...');
  });

  afterEach(() => {
    console.log('After each hook...');
  });

  after(() => {
    console.log('After hook...');
  });
```

```
  it('Placeholder one', () => {
    assert.equal(true, true);
  });

  it('Placeholder two', () => {
    assert.equal(true, true);
  });
});
```

Running the above spec produces the following output:

```
  Hooks demo
Before hook...
Before each hook...
    ✓ Placeholder one
After each hook...
Before each hook...
    ✓ Placeholder two
After each hook...
After hook...


  2 passing (5ms)
```

Notice that the `before` and `after` hooks only ran once while the `beforeEach` and `afterEach` hooks each ran once per test.

Hooks are defined within a `describe` or `context` function. While hooks can be defined before, after, or interspersed with your tests, keeping all of your hooks together (before or after your tests) will help others to read and understand your code.

Defining hooks out of their logical order has no effect on when they're ran. Consider the following example that defines an `afterHook` before a `beforeEach` hook:

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

describe('Hooks demo', () => {
  afterEach(() => {
    console.log('After each hook...');
  });

  beforeEach(() => {
    console.log('Before each hook...');
  });

  it('Placeholder one', () => {
    assert.equal(true, true);
  });

  it('Placeholder two', () => {
    assert.equal(true, true);
  });
});
```

Running the above spec produces the following output:

```
  Hooks demo
Before each hook...
    ✓ Placeholder one
After each hook...
Before each hook...
    ✓ Placeholder two
After each hook...


  2 passing (6ms)
```

The order of your hooks only matters when you define multiple hooks of the same type. When a hook type is defined more than once, they'll be ran in the order that they're defined in:

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

describe('Hooks demo', () => {
  beforeEach(() => {
    console.log('Before each hook #1...');
  });

  beforeEach(() => {
    console.log('Before each hook #2...');
  });

  it('Placeholder one', () => {
    assert.equal(true, true);
  });

  it('Placeholder two', () => {
    assert.equal(true, true);
  });
});
```

Running the above spec produces the following output:

```
  Hooks demo
Before each hook #1...
Before each hook #2...
    ✓ Placeholder one
Before each hook #1...
Before each hook #2...
    ✓ Placeholder two


  2 passing (5ms)
```

You can also define hooks within nested `describe` or `context` functions:

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

describe('Hooks demo', () => {
  before(() => {
    console.log('Before hook...');
  });

  beforeEach(() => {
    console.log('Before each hook...');
  });

  afterEach(() => {
    console.log('After each hook...');
  });

  after(() => {
    console.log('After hook...');
  });

  it('Placeholder one', () => {
    assert.equal(true, true);
  });

  it('Placeholder two', () => {
    assert.equal(true, true);
  });

  describe('nested tests', () => {
    before(() => {
      console.log('Nested before hook...');
    });

    beforeEach(() => {
      console.log('Nested before each hook...');
    });

    afterEach(() => {
      console.log('Nested after each hook...');
    });

    after(() => {
      console.log('Nested after hook...');
```

```
    });

    it('Placeholder one', () => {
      assert.equal(true, true);
    });

    it('Placeholder two', () => {
      assert.equal(true, true);
    });
  });
});
```

Running the above spec produces the following output:

```
  Hooks demo
Before hook...
Before each hook...
    ✓ Placeholder one
After each hook...
Before each hook...
    ✓ Placeholder two
After each hook...
    nested tests
Nested before hook...
Before each hook...
Nested before each hook...
      ✓ Placeholder one
Nested after each hook...
After each hook...
Before each hook...
Nested before each hook...
      ✓ Placeholder two
Nested after each hook...
After each hook...
Nested after hook...
After hook...


  4 passing (7ms)
```

Notice that the `before` and `after` hooks defined in the top-level `describe` function run only once while the `beforeEach` and `afterEach` hooks run before and after (respectively) for each of the tests defined in the top-level `describe` function *and* for each of the tests defined in the nested `describe` function.

While the need to define nested hooks won't come up very often (especially when you're just starting out with unit testing), it is very helpful to be able to define a `beforeEach` hook in a top-level `describe` function that will run before every test in that block and before every test within nested `describe` or `context` functions (you'll do exactly that in just a bit).

You can also optionally pass a description for a hook or a named function:

```
beforeEach('My hook description', () => {
  console.log('Before each hook...');
});

beforeEach(function myHookName() {
  console.log('Before each hook...');
});
```

If an error occurs with executing the hook, the hook description or function name will display in the console along with the error information to assist with debugging.

Using the `beforeEach` Mocha Hook

Let's go back to our spec and see how we can use hooks to DRY up our code. Here's where we left off:

```
describe("Dog Constructor Function", function() {
  it('should have a "name" property', function() {
    const layla = new Dog("Layla");
    expect(layla).to.have.property("name");
  });

  it('should set the "name" property when a new dog is created', function() {
    const layla = new Dog("Layla");
    // we are using the eql function to compare the value of layla.name
    // with the provided string
    expect(layla.name).to.eql("Layla");
  });
});
```

Let's refactor our code to use a `beforeEach` hook to assign the value of our new dog instance:

```
describe("Dog", function() {
  // we'll declare our variable here to ensure it's available within the scope
  // of all the specs below
  let layla;

  // now for each test below we'll create a new instance to ensure each of our
  // dog instances is exactly the same
  beforeEach("set up a dog instance", function() {
    layla = new Dog("Layla");
  });

  describe("Dog Constructor Function", function() {
    it('should have a "name" property', function() {
      expect(layla).to.have.property("name");
    });

    it('should set the "name" property when a new dog is created', function() {
      expect(layla.name).to.eql("Layla");
    });
  });
});
```

Now let's write a test from the next method on the `Dog` class: `Dog.prototype.bark()`. For testing classes we'll create a new `describe` function to test each individual method. We'll now write our unit test inside taking advantage of our `beforeEach` hook:

```
describe("Dog", function() {
  let layla;

  beforeEach("set up a dog instance", function() {
    layla = new Dog("Layla");
  });

  // etc, etc.

  describe("prototype.bark()", function() {
    it("should return a string with the name of the dog barking", function() {
      expect(layla.bark()).to.eql("Layla is barking");
    });
  });
});
```

Not only are we avoiding repeating our setup code within each test but we've improved the readability of our code by making it more self-descriptive. The code that runs before each test is literally contained with a hook named `beforeEach`!

The `after` and `afterEach` hooks are generally used less often than the `before` and `beforeEach` hooks. Most of the time, it's preferable to avoid using the `after` and `afterEach` hooks to perform clean up tasks after your tests. Instead, simply use the `before` and `beforeEach` hooks to create a clean starting point for each of your tests. Doing this will ensure that your tests run in a consistent, predictable manner.

Using Chai Spies

Sweet - let's now look to the next method on the `Dog.prototype`- `Dog.prototype.chainChaseTail`. This instance method intakes a number (num) and will then invoke the `Dog.prototype.chaseTail` function `num` number of times. The `chaseTail` function will just `console.log` a string - meaning that we have no function output to test. The `Dog.prototype.chainChaseTail` function will additionally throw a `TypeError` if the incoming argument is not a number.

We'll start by setting up our outer `describe` block for the `prototype.chainChaseTail` method. Next we'll add two `context` functions for our two contexts - valid **or** invalid parameters:

`context` is just an alias for `describe` it's just another way to make your tests more understandable and readable, in this case we are testing our method with different parameters, and thus in different contexts. (not to be confused with "context" in the javascript sense of the value of `this`)

```
describe("prototype.chainChaseTail()", function() {
  context("with an invalid parameter", function() {});
  context("with a valid number parameter", function() {});
});
```

We'll start by writing our test for when the method is invoked with invalid parameters. To do this we'll use Chai's `throw` method ensuring to wrap our error throwing function in another function:

```
context("with an invalid parameter", function() {
  it("should throw a TypeError when given an argument that is not a number", func
    expect(() => layla.chainChaseTail("3")).to.throw(TypeError);
  });
});
```

Note here we are passing the literal string `"3"` not the number 3.

Nice, now we can concentrate on our other context with a valid parameter - and how to go about testing this function. In order to test `chainChaseTail` properly we'll need to see *how many times* the `chaseTail` method is invoked. Which means we'll need to import another library that will add extra functionality to `Chai`. We'll import the Chai Spies library using `npm install chai-spies` in our top level directory.

Now we'll insert a few lines of code to the top of file to set up our shiny new Chai Spies:

```
// top of dog-spec.js
const chai = require("chai");
const expect = chai.expect;
const spies = require("chai-spies");
chai.use(spies);
```

We now have access to the `chai-spies` module in our tests. The `Chai Spies` library provides a lot of added functionality including the ability to determine if a function has been called and how many times that function has been called.

So let's get started spying! We'll setup our `it` function with an appropriate string:

```
context("with a valid number parameter", function() {
  it("should call the chaseTail method n times", function() {});
});
```

Now in order to spy on a function we first need to tell Chai which function we'd like to spy on using the `chai.spy.on` method. In this case we'd like to spy on the instance of a Dog that will be invoking the `chainChaseTail` method to determine how many times the `chaseTail` method is then invoked.

So we will set up our spy on the dog instance in question, as well as tell our chai spy which method to keep track of:

```
context("with a valid number parameter", function() {
  it("should call the chaseTail method n times", function() {
    // the first argument will be the instance we are spying on
    // the second argument will be the method we want to keep track of
    const chaseTailSpy = chai.spy.on(layla, "chaseTail");
  });
});
```

Now that our spy is set up we now need make sure our dog instance will actually call the `chainChaseTail` function! Otherwise our spy won't have anything to spy on:

```
context("with a valid number parameter", function() {
  it("should call the chaseTail method n times", function() {
    const chaseTailSpy = chai.spy.on(layla, "chaseTail");
    // we need to invoke chainChaseTail because that is the method that
    // will invoke chaseTail which is the method we are spying on
    layla.chainChaseTail(3);
  });
});
```

Finally, we need to add our actual test - otherwise this is all for naught! Chai has some really nice chaining methods when it comes to checking how many times a function has been invoked. Here we'll use the method chain of `expect(func).to.have.been.called.exactly(n)` to test that the method we are spying on - `chaseTail` was invoked a certain number of times:

```
context("with a valid number parameter", function() {
  it("should call the chaseTail method n times", function() {
    const chaseTailSpy = chai.spy.on(layla, "chaseTail");
    layla.chainChaseTail(3);
    // below is our actual test to see how many times our spy was invoked
    expect(chaseTailSpy).to.have.been.called.exactly(3);
  });
});
```

Testing static methods on classes

Sweet! We are almost done testing this class - just one more method to go. We'll now work on testing the class method `Dog.cleanDogs`. To denote that this is a class method, not an instance method, our `describe` string will not use the word `prototype`:

```
describe("cleanDogs()", function() {
  it("should return an array of each cleaned dog string", function() {});
});
```

Now the `Dog.cleanDogs` class method will intake an array of dogs and output an array where each element is a string noting that the passed in dog instance's paws are now clean. In order to properly test this function we'll probably want an array of more than one dog instance. Let's create a new dog and pass an array of two dog instances to the `Dog.cleanDogs` method:

```
describe("cleanDogs()", function() {
  it("should return an array of each cleaned dog string", function() {
    const zoey = new Dog("Zoey");
    let cleanDogsArray = Dog.cleanDogs([layla, zoey]);
  });
});
```

Then we'll create a variable for our expected output and compare the output we received from `Dog.cleanDogs`:

```
describe("cleanDogs()", function() {
  it("should return an array of each cleaned dog string", function() {
    const zoey = new Dog("Zoey");
    let cleanDogsArray = Dog.cleanDogs([layla, zoey]);
    let result = ["I cleaned Layla's paws.", "I cleaned Zoey's paws."];
    expect(cleanDogsArray).to.eql(result);
  });
});
```

Awesome! We have fully testing the `Dog`class's methods and learned a lot about testing along the way.

Here is our full testing file so you can ensure you got everything:

```
const chai = require("chai");
const expect = chai.expect;
const spies = require("chai-spies");
chai.use(spies);

// this is a relative path to the function location
const Dog = require("../problems/dog.js");

describe("Dog", function() {
  let layla;

  beforeEach("set up a dog instance", function() {
    layla = new Dog("Layla");
  });

  describe("Dog Constructor Function", function() {
    it('should have a "name" property', () => {
      expect(layla).to.have.property("name");
    });

    it('should set the "name" property when a new dog is created', () => {
      expect(layla.name).to.eql("Layla");
    });
  });

  describe("prototype.bark()", function() {
    it("should return a string with the name of the dog barking", () => {
      expect(layla.bark()).to.eql("Layla is barking");
    });
  });

  describe("prototype.chainChaseTail()", function() {
    context("with a valid number parameter", function() {
      it("should call the chaseTail method n times", function() {
        const chaseTailSpy = chai.spy.on(layla, "chaseTail");
        layla.chainChaseTail(3);

        expect(chaseTailSpy).to.have.been.called.exactly(3);
      });
    });

    context("with an invalid parameter", function() {
      it("should throw a TypeError when given an argument that is not a number",
        expect(() => layla.chainChaseTail("3")).to.throw(TypeError);
      });
    });
  });

  describe("cleanDogs()", function() {
    it("should return an array of each cleaned dog string", function() {
      const zoey = new Dog("Zoey");
      let cleanDogsArray = Dog.cleanDogs([layla, zoey]);

      let result = ["I cleaned Layla's paws.", "I cleaned Zoey's paws."];
      expect(cleanDogsArray).to.eql(result);
    });
  });
});
```

What you learned

In the upcoming project we'll be covering a lot more Chai syntax - but don't worry about memorizing this syntax! The point we are trying to make is that in the future you'll be using a variety of software testing frameworks and assertion libraries - the most important things are to know the basics of how to structure tests as well as being able to read and parse documentation to write tests.

In this series of readings we covered the basics of how to:

- properly format and denote your mocha tests using `describe`, `context` and `it` blocks
- write tests for individual functions as well as writing tests for classes
- use `chai-spies` to test how many times a function has been called
- test that functions will throw certain errors
- recognize and utilize the Mocha hooks: `before`, `beforeEach`, `after`, and `afterEach`

Well-Tested Full-Stack To-Do Items

In this project, you will test a full-stack JavaScript and HTML application! You will write tests to make sure the code that was written for the project will meet the expectations of the requirements. Your tests will not have to be exhaustive. Instead, there are guidelines for your tests in each test file. Use those guidelines to implement the Web application.

The upcoming video provides you a full walk-through of the system as it is created. Then, once you understand how the application works from watching it be built, you will need to apply your knowledge of writing tests.

It may be hard. However, stick with it. You'll do great. Just take your time, write good tests, and you will be amazed at how much confidence that you will gain in writing code that comes together.

One of the ways that you can make this project more enjoyable is to vary the way that you pair on it. For each step,

- Discuss what the next feature is that you want to write
- One person writes a unit test to test the code
- Both examine the code and determine if there is any duplication to refactor into common functions or classes
- Loop, but swap who writes the unit test and who writes the code

At the end, you will leverage your test by swapping out the mechanism used to generate the HTML. This is the other part of writing good tests: tests give you the confidence to change code. If you do something that is "wrong" in that it breaks current expectations, the tests will tell you!

Tests are such an important part of a developers life. While some developers will complain about having to write them, when you start working on an existing "legacy" code base, making changes can cause a lot of stress unless you have the work of other developers' tests to make sure you don't unintentionally change a method in such a way to break code that you're not working on.

To get started,

- clone the project fromhttps://github.com/appacademy-starters/testing-an-existing-app-project
- change directory into the project
- run `npm install`to install the modules

If you want to run the server, type `node server.js`and go tohttp://localhost:3000/itemsto see what it does. You can add categories, add items, search for items, and complete items.

The code that you will test are the functions used to create the functionality of the data and the creation of the views, not the actual HTTP server. The video after shows a person writing the entire application. You will see what each piece does. Then, you will understand the *intent* of the code that you have to test.

Create And Serve the Category Screen

You'll now write the tests for the part of the application that shows the list of categories. That code, in **server.js**looks like this.

```
const filePath = path.join(__dirname, 'category-list-screen.html');
const template = await fs.promises.readFile(filePath, 'utf-8');
const html = mergeCategories(template, categories, 'li');
res.setHeader('Content-Type', 'text/html');
res.writeHead(200);
res.write(html);
```

You need to write tests for the function `mergeCategories()`for the portion that outputs the HTML for list items. Open the file **test/merge-categories-spec.js**. You will see

```
describe("mergeCategories()", () => {
  context("Using <li> tags", () => {
    const template = `
      <div>
        <ul>
          <!-- Content here -->
        </ul>
      </div>
    `;

    it("should return no <li>s for no categories", () => {
      expect.fail('please write this test');
    });

    it("should return a single <li> for one category", () => {
      expect.fail('please write this test');
    });

    it("should return an <li> for each category", () => {
      expect.fail('please write this test');
    });
  });

  // more code ...
```

The `context`block is for writing tests for when we use `mergeCategories()`and pass it `<li>`tags.

You will need to write tests in all the `it`blocks. Just replace the `expect.fail`calls with your own tests. (`expect.fail`is a chai assertion to force a spec to fail so we are using it for all the unwritten tests so that when you run `npm test`you will see all the tests you haven't written failing)

Open **merge-categories.js**to review the code before writing the tests.

The `mergeCategories`function takes a string through its `template`parameter, a list of strings through its `categories`parameter and an HTML tag through it's `tagName`parameter.

It then replaces the HTML comment `<!-- Content here ..>`with the newly created `<li>`tags (one for each catagory) and returns a new string of HTML.

Use the `template`variable that is available to you for these tests.

The first test

The first test reads

```
it("should return no <li>s for no categories", () => {
  expect.fail('please write this test');
});
```

Replace the `expect.fail` line with a test that properly follows the *Three As* of unit testing.

In the *arrange* section, you will need to create an empty array for the `categories` and store it in a variable. You will use the variable in the action.

In the *act* section, you will invoke the `mergeCategories` function with the `template` as the first argument, the variable that contains an empty array as the second argument, and the string 'li' for the tag name as the third argument. Store the return value in a variable.

In the *assert* section, assert that each of the following are true using the include assertion provided by Chai:

- To make sure that the method doesn't *remove* the wrong things
  - Assert that it contains the string " "
  - Assert that it contains the string " "
  - Assert that it contains the string "<ul>"
  - Assert that it contains the string "
"
- To make sure that the method doesn't *add* the wrong things
  - Assert that it does not contain the string "<li>"
  - Assert that it does not contain the string "
"
- To make sure it replaces what you expect it to replace
  - Assert that it does not contain the string "<!-- Content here -->"

Run the test to make sure it passes.

Here's what the test could look like.

```
it("should return no LIs for no categories", () => {
  const categories = [];
  const result = mergeCategories(template, categories, 'li');
  expect(result).to.contain('<div>');
  expect(result).to.contain('</div>');
  expect(result).to.contain('<ul>');
  expect(result).to.contain('</ul>');
  expect(result).to.not.contain('<li>');
  expect(result).to.not.contain('</li>');
});
```

Notice we are using `contain` here instead of `include`. `contain` is an alias to `include` that chai provides, and it reads better here than `include`.

The second test

The second test reads

```
it("should return a single <li> for one categories", () => {
  expect.fail('please write this test');
});
```

Replace the `expect.fail` line with a test that properly follows the *Three As* of unit testing.

In the *arrange* section, you will need to create an array for the `categories` argument that contains a single string and store it in a variable. You will use the variable in the action and the value that you typed in the assertion.

In the *act* section, you will invoke the `mergeCategories` function with the `template` as the first argument, the variable that contains the array with the single value as the second argument, and the string 'li' for the tag name as the third argument. Store the return value in a variable.

In the *assert*section, assert that each of the following are true using theincludeassertion provided by Chai:

- To make sure that the method doesn't *remove*the wrong things
  - Assert that it contains the string " "
  - Assert that it contains the string " "
  - Assert that it contains the string "<ul>"
  - Assert that it contains the string "
"
- To make sure that the method *adds*the right things
  - Assert that it does contain the string "
your string here
" where "your string here" is the single value that you placed in the array
- To make sure it replaces what you expect it to replace
  - Assert that it does not contain the string "<!-- Content here -->"

Run the test to make sure it passes.

Here's what the test could look like.

```
it("should return a single LI for one categories", () => {
  const categories = ['Cat 1'];
  const result = mergeCategories(template, categories, 'li');
  expect(result).to.contain('<div>');
  expect(result).to.contain('</div>');
  expect(result).to.contain('<ul>');
  expect(result).to.contain('</ul>');
  expect(result).to.contain('<li>Cat 1</li>');
});
```

The third test

The third test reads

```
it("should return an <li> for each category", () => {
  expect.fail('please write this test');
});
```

Replace the `expect.fail`line with a test that properly follows the *Three A*sof unit testing.

In the *arrange*section, you will need to create an array for the `categories`argument that contains multiple strings and store it in a variable. You will use the variable in the action and the values that you typed in the assertion.

In the *act*section, you will invoke the `mergeCategories`function with the`template`as the first argument, the variable that contains the array with the multiple values as the second argument, and the string 'li' for the tag name as the third argument. Store the return value in a variable.

In the *assert*section, assert that each of the following are true using theincludeassertion provided by Chai:

- To make sure that the method doesn't *remove*the wrong things
  - Assert that it contains the string " "
  - Assert that it contains the string " "
  - Assert that it contains the string "<ul>"
  - Assert that it contains the string "
"
- To make sure that the method *adds*the right things, for *each*of the values that you put in your categories array:
  - Assert that it does contain the string "
value n
" where "value n" is one of the values in your array
- To make sure it replaces what you expect it to replace

- Assert that it does not contain the string "<!-- Content here -->"

Run the test to make sure it passes.

Here's what that code could look like.

```
it("should return an LI for each category", () => {
  const categories = ['Cat 1', 'Cat 2', 'Cat 3'];
  const result = mergeCategories(template, categories, 'li');
  expect(result).to.contain('<div>');
  expect(result).to.contain('</div>');
  expect(result).to.contain('<ul>');
  expect(result).to.contain('</ul>');
  expect(result).to.contain('<li>Cat 1</li>');
  expect(result).to.contain('<li>Cat 2</li>');
  expect(result).to.contain('<li>Cat 3</li>');
});
```

You have won this round!

Save Submitted Category Information

You'll now write the tests for the part of the application that saves a category when it is submitted. That code, in **server.js**looks like this and is what happens when a new category is sent to the server in an HTTP POST.

```
else if (req.url === "/categories" && req.method === 'POST') {
    const body = await getBodyFromRequest(req);
    const newCategory = getValueFromBody(body, 'categoryName')
    categories = saveCategories(categories, newCategory);
    res.setHeader('Location', '/categories');
    res.writeHead(302);
}
```

There are three main functions we need to test in this block of code.

Here's what they all three do in a nutshell:

`getBodyFromRequest`- Gets the raw POST body string from the HTTP POST request`getValueFromBody`- Parses this raw string into individual values representing the new categories`saveCategories`- Saves the new categories into the existing list of categories;

You need to write tests for all three functions `getBodyFromRequest`,`getValueFromBody`, and `saveCategories`.

Testing getting the body from the request (`getBodyFromRequest()`)

Open up **get-body-from-request.js**and review it.

The `getBodyFromRequest()`function takes one argument `req`which is an IncomingMessageobject. It returns a Promise which means it is an *asynchronous*function.

The `IncomingMessage`stored in `req`contains properties like `url`and `method`, but also a stream of data that was sent to us by the browser. We call this stream of data the POST "body".

"GET" requests have *no*data in the body, ever. "POST" submissions almost *always*contain data. Because this is a POST that the code is handling, the code needs to read *all*of the data from the stream. To do that, it listens for two events, the "data" event and the "end" event.

When data shows up for the server to read, it has to do it in chunks because we can't predict how much data the browser will be sending the server and the data could be huge!

The code to do that is

```
req.on('data', chunk => {
  data += chunk;
});
```

The callback will be called everytime the server receives a chunk of data from the browser. Then we just append the incoming chunk of data to the existing data variable with `+=`.

We will continue to do this as long as the server is still receiving chunks.

When the data finishes arriving at the server, the "end" event occurs. That signals the code that it has finished arriving and the Promise in the method can finish with a call to `resolve`passing it the `data`. That is this piece of code from `getBodyFromRequest`:

```
req.on('end', () => {
  resolve(data);
});
```

This is a hard one to test because you need to test those events. The stream of data inherits from a class `EventEmitter`. You can use an instance of the `EventEmitter` class to test this code. This is called *a stub* or *a fake* because it's not a real `IncomingMessage`. You can trigger an event using the `emit` method which takes the name of the event as the first parameter and, as an optional second parameter, any data.

For Example:

```
const fakeReq = new EventEmitter();
fakeReq.emit('end');
```

would emit the "end" event.

Another thing that makes this hard is that it is an *asynchronous* test which means that you **must** use the `done` method that mocha provides as part of the test callback. If everything is ok, then you call `done` without any arguments. If something bad happens, you call `done` with the error message.

You can see an example in this `it` block from the **get-body-from-request-spec.js** file. The `done` function is the first argument to the `it` callback.

```
  it('returns an empty string for no body', done => {
    expect.fail('please write this test');
  });
```

This should remind you of the `resolve` function in Promises, it's a similar pattern.

The first body request test

For the first test, *returns an empty string for no body*, the following code uses the `EventEmitter` stored in `fakeReq` (which is created in the `beforeEach` block) as the fake request to test the `getBodyFromRequest` function.

Write your assertion in the `then` handler of the promise returned by `getBodyFromRequest`. Check to see if the value in `body` is an empty string. If it is, the function works as you expect and you should call `done()`. If not, you should call `done` with an error message. The comments in the `then` function are there to guide you to do that.

```
it('returns an empty string for no body', done => {
  // Arrange
  const bodyPromise = getBodyFromRequest(fakeReq);

  // Act
  // This next line emits an event using
  // emit(event name, optional data)
  fakeReq.emit('end');

  // Assert
  bodyPromise
    .then(body => {
      // Write the following code:
      // Determine if body is equal to ""
      // If it is, call done()
      // If it is not, call
      //   done(`Failed. Got "${body}"`)
    });
});
```

The second body request test

For the second test, *returns the data read from the stream*, use the `EventEmitter` stored in `fakeReq` as the fake request to test the `getBodyFromRequest` function. This time, though, you need to emit some "data" events before you emit the "end" event to test the data-gathering functionality of the method.

From the last section, you know that the signature for the `emit` method is

```
eventEmitter.emit('event name', 'optional data');
```

In the cases below, the event name is "data" and the optional data is stored in `data1` and `data2`. So, you should have *two* calls to `emit` before the `fakeReq.emit('end');`. You can see space for you to write those calls.

Then, in the `then` handler of the Promise, you should check to see if the value in `body` is the same as `data1 + data2`. If it is, the function works as you expect and you should call `done()`. If not, you should call `done` with an error message. The comments in the `then` function are there to guide you to do that.

```
it('returns the data read from the stream', done => {
  // Arrange
  const bodyPromise = getBodyFromRequest(fakeReq);
  const data1 = "This is some";
  const data2 = " data from the browser";

  // Act
  // Write code to emit a "data" event with
  // the data stored in data1

  // Write code to emit a "data" event with
  // the data stored in data2

  fakeReq.emit('end');

  // Assert
  bodyPromise
    .then(body => {
      // Write the following code:
      // Determine if body is equal to data1 + data2
      // If it is, call done()
      // If it is not, call
      //   done(`Failed. Got "${body}"`)
    });
});
```

Testing getting the value from the body (`getValueFromBody`)

It's not enough to just get the stream of raw data from the `POST` body, we also need to parse that data into the categories the user is saving.

When someone POSTs a form from the browser to the server, it comes to the server in a format called "x-www-form-urlencoded". This is also sometimes called a "Query String"

"x-www-form-urlencoded" is just a format for data just like JSON is also a format for data. This specific format is made up of key/value pairs. The key/value pairs are in the form "*key=value*". Those pairs are joined together in a single string by using the ampersand character. The following are valid strings contained in the "x-www-form-urlencoded" format.

- **""**: The empty string is a valid x-www-form-urlencoded string.
- **"name=Morgan"**: The key in this case is "name" and the value is "Morgan"
- **"name=Petra&age=31"**: There are two key/value pairs in this, "name" and "Petra", and "age" and "31"
- **"name=Bess&age=&job=Boss"**: There are three key/value pairs in this
  - "name" and "Bess"
  - "age" and "" (I guess they didn't answer?)
  - "job" and "Boss"

If one of the values contains a character that's not a letter or number, it's replaced by a weird number that begins with a percent sign. That's known as URL encoding. The most common replacement is "%20" for the space character. Here are some valid strings with that replacement.

- **"name=Chandra%20K&age=13%20years%20old"**: This string has two key/value pairs
  - "name" and "Chandra K"
  - "age" and "13 years old"

- **"title=Boop%20Lord"**: This string has one key/value pair, "title" and "Boop Lord"

In the tests that you write, you will not have to write these "x-www-form-urlencoded" strings. They will be provided to you in the test. However, you should be able to read them so that you become familiar with how they > work.

Open up the **get-value-from-body.js**file to see the two lines of code in the `getValueFromBody`function that implement this behavior. Notice we are using the built in `querystring`module in Node.js to `parse()`our `x-www-form-urlencoded`string.

`getValueFromBody`takes in two arguments, `body`and `key`. It then parses the `x-www-form-urlencoded`body and returns the value that cooresponds to `key`.

To make sure that it behaves, though, there are multiple tests in **get-value-from-body-spec.js**in the **test**directory. Let's look at those.

There are five tests. The first three have the body *and*key defined for you. The last two have the body defined and you should figure out the key to test.

The first test

The first test is *returns an empty string for an empty body*. So, if the body is empty, regardless of the key, the `getValueFromBody`method returns an empty string.

```
it('returns an empty string for an empty body', () => {
  // Arrange
  const body = "";
  const key = "notThere";

  // Act
  // Write code to invoke getValueFromBody and collect
  // the result


  // Assert
  // Replace the fail line with an assertion for the
  // expected value of ""
  expect.fail('please write this test');
});
```

In this test, you need to write the code that invokes the `getValueFromBody`method with the `body`and `key`arguments. The result that comes back is what you should assert instead of just having it fail.

Take a moment and try to complete that on your own. The following code snippet will show you the solution, so give it a shot figuring out the two lines of code that you need to complete the previous one.

Here's the solution:

```
it('returns an empty string for an empty body', () => {
  // Arrange
  const body = "";
  const key = "notThere";
```

```
  // Act
  // Write code to invoke getValueFromBody and collect
  // the result
  const result = getValueFromBody(body, key);

  // Assert
  // Replace the fail line with an assertion for the
  // expected value of ""
  expect(result).to.equal('');
});
```

The second test

The second test is *returns an empty string for a body without the key*. So, if you ask for the value of a key that is not in the body, the `getValueFromBody` method returns an empty string.

```
it('returns an empty string for a body without the key', () => {
  // Arrange
  const body = "name=Bess&age=29&job=Boss";
  const key = "notThere";

  // Act
  // Write code to invoke getValueFromBody and collect
  // the result


  // Assert
  // Replace the fail line with an assertion for the
  // expected value of ""
  expect.fail('please write this test');
});
```

This code will look very, very similar to the last test. Complete it to make it pass.

The third test

The third test, *returns the value of the key in a simple body*, is also very similar to the past two tests. In this case, you have to compare it to the expected value "Bess".

```
it('returns the value of the key in a simple body', () => {
  const body = "name=Bess";
  const key = "name";

  // Act
  // Write code to invoke getValueFromBody and collect
  // the result


  // Assert
  // Replace the fail line with an assertion for the
  // expected value of "Bess"
  expect.fail('please write this test');
});
```

The fourth test

The fourth test, *returns the value of the key in a complex body*, is also very similar to the past three tests. In this case, you have to choose a key that you want to test from the existing keys in the body and, then, the value that it has so that you can make the assertion at the end.

```
it('returns the value of the key in a complex body', () => {
  const body = "name=Bess&age=29&job=Boss";
  // Select one of the keys in the body

  // Act
  // Write code to invoke getValueFromBody and collect
  // the result
```

```
  // Assert
  // Replace the fail line with an assertion for the
  // expected value for the key that you selected
  expect.fail('please write this test');
});
```

The fifth test

The fifth test, *decodes the return value of URL encoding*, is also very similar to the past three tests. In this case, you will test the value of the "level" key. Complete the code with the correct assertion. Remember that `%20` should be decoded and be turned into the space character.

```
it('decodes the return value of URL encoding', () => {
  const body = "name=Bess&age=29&job=Boss&level=Level%20Thirty-One";
  const key = "level";

  // Act
  // Write code to invoke getValueFromBody and collect
  // the result


  // Assert
  // Replace the fail line with an assertion for the
  // expected value for the "level" key
  expect.fail('please write this test');
});
```

Testing saving the categories

Open up **save-categories.js** and review it. This contains a method that pushes a new category in the `newCategory` parameter onto the argument provided in `categories` (hopefully an array!). Then, it sorts the `categories` array. Finally, it returns a "clone" of the array by just creating a new array with all of the old entries. This is done to keep modifications to the old array from messing with the new array. It is an implementation detail that you just need to test for.

Open **save-categories-spec,js**. It has three tests in it for you to complete.

The first test

In the first test, you must provide the "Act" stage by calling the `saveCategories` method with the provided `categories` and `newCategory` values and store its return value in a variable named "result".

Of note with the first test is that the assertion (that you do not have to write) uses the "include" method to test if a value is in an array.

The second test

In the second test, you must provide the "Assert" stage by writing the assertion to test using a new method named "eql" rather than "equal". Everything else remains the same.

The reason that you use eql instead of `equal` is the "type" of equality each one provides. The `equal` function, which you've used until now, compares objects and arrays only by their instance. That means equality between arrays and objects using `equal` will only pass if they're *the same object in memory*.

```
// Different arrays with the same content
expect(['a', 'b']).to.equal(['a', 'b']); // => FAIL

// Same arrays
const array = ['a', 'b'];
expect(array).to.equal(array); // => PASS
```

The `eql` method performs "member-wise equality". It will compare the values *inside* the array as opposed to the instance of the array. Because of that, both of the previous examples pass with the `eql` method.

```
// Different arrays with the same content
expect(['a', 'b']).to.eql(['a', 'b']); // => PASS

// Same arrays
const array = ['a', 'b'];
expect(array).to.eql(array); // => PASS
```

The third test

In the third test, you must provide the "Arrange" portion. Interestingly, you can really provide any array and string value. That's an easy one.

Et, voila!

It seems that you have fully tested all of the code that it takes to test the "save category" code. Well done!

You win this round, too!

Create And Serve A To-Do Item Form

To display the form that lets you enter new items, it has to create a dropdown that contains the categories that are in the application. This is identical in *intent* to the code that creates a list of categories to display on the category screen. Because of that, here's the code that handles displaying the "new item" screen.

```
else if (req.url === "/items/new" && req.method === 'GET') {
  const filePath = path.join(__dirname, 'todo-form-screen.html');
  const template = await fs.promises.readFile(filePath, 'utf-8');
  const html = mergeCategories(template, categories, 'option');
  res.setHeader('Content-Type', 'text/html');
  res.writeHead(200);
  res.write(html);
}
```

In this case, the `mergeCategories` method is now called with the third argument of "option" rather than "li" as it was before. This is what the last three tests in the **merge-categories-spec.js** file address. You will write tests that generate "option" tags rather than "li" tags. You'll also test that the replacement correctly occurred.

In this case, you'll modify the tests in the second sub-"describe" section, the one that reads "For selects".

The first test

The first test reads

```
it("should return no <option>s for no categories", () => {
  expect.fail('please write this test');
});
```

Replace the `expect.fail` line with a test that properly follows the *Three As* of unit testing.

In the *arrange* section, you will need to create an empty array for the `categories` and store it in a variable. You will use the variable in the action.

In the *act* section, you will invoke the `mergeCategories` function with the `template` as the first argument, the variable that contains an empty array as the second argument, and the string 'option' for the tag name as the third argument. Store the return value in a variable.

In the *assert* section, assert that each of the following are true in the return value that you saved in the *act* section using the include assertion provided by Chai:

- To make sure that the method doesn't *remove* the wrong things
  - Assert that it contains the string " "
  - Assert that it contains the string " "
  - Assert that it contains the string "<select>"
  - Assert that it contains the string "</select>"

- To make sure that the method doesn't *add* the wrong things
  - Assert that it does not contain the string "<option>"
  - Assert that it does not contain the string "</option>"

- To make sure it replaces what you expect it to replace
  - Assert that it does not contain the string "<!-- Content here -->"

Run the test to make sure it passes.

Except for some string differences, this test will look nearly identical to the first test that you did for the `<li>` tags earlier in this project.

The second test

The second test reads

```
it("should return a single <option> for one category", () => {
  expect.fail('please write this test');
});
```

Replace the `expect.fail` line with a test that properly follows the *Three A*s of unit testing.

In the *arrange* section, you will need to create an array for the `categories` argument that contains a single string and store it in a variable. You will use the variable in the action and the value that you typed in the assertion.

In the *act* section, you will invoke the `mergeCategories` function with the `template` as the first argument, the variable that contains the array with the single value as the second argument, and the string 'option' for the tag name as the third argument. Store the return value in a variable.

In the *assert* section, assert that each of the following are true using the include assertion provided by Chai:

- To make sure that the method doesn't *remove* the wrong things
    - Assert that it contains the string " "
    - Assert that it contains the string " "
    - Assert that it contains the string "<select>"
    - Assert that it contains the string "</select>"
- To make sure that the method *adds* the right things
    - Assert that it does contain the string "<option>your string here</option>" where "your string here" is the single value that you placed in the array
- To make sure it replaces what you expect it to replace
    - Assert that it does not contain the string "<!-- Content here -->"

Run the test to make sure it passes.

Except for some string differences, this test will look nearly identical to the second test that you did for the `<li>` tags earlier in this project.

The third test

The third test reads

```
it("should return an <option> for each category", () => {
  expect.fail('please write this test');
});
```

Replace the `expect.fail` line with a test that properly follows the *Three A*s of unit testing.

In the *arrange* section, you will need to create an array for the `categories` argument that contains multiple strings and store it in a variable. You will use the variable in the action and the values that you typed in the assertion.

In the *act* section, you will invoke the `mergeCategories` function with the `template` as the first argument, the variable that contains the array with the

many values as the second argument, and the string 'option' for the tag name as the third argument. Store the return value in a variable.

In the *assert*section, assert that each of the following are true using theincludeassertion provided by Chai:

- To make sure that the method doesn't *remove*the wrong things
  - Assert that it contains the string " "
  - Assert that it contains the string " "
  - Assert that it contains the string "<select>"
  - Assert that it contains the string "</select>"

- To make sure that the method *adds*the right things, for *each*of the values that you put in your categories array:
  - Assert that it does contain the string "<option>value n</option>" where "value n" is one of the values in your array

- To make sure it replaces what you expect it to replace
  - Assert that it does not contain the string "<!-- Content here -->"

Run the test to make sure it passes.

Except for some string differences, this test will look nearly identical to the third test that you did for the `<li>`tags earlier in this project.

You have won this round!

Save And Show To-Do Items

In this step, you'll test the code for two different handlers, the one that shows the screen that has the list of items on it and the one that handles the creation of a new item. Here are those two parts of the `if-else` block in **server.js**.

```
else if (req.url === "/items" && req.method === 'GET') {
  const filePath = path.join(__dirname, 'list-of-items-screen.html');
  const template = await fs.promises.readFile(filePath, 'utf-8');
  const html = mergeItems(template, items);
  res.setHeader('Content-Type', 'text/html');
  res.writeHead(200);
  res.write(html);
}

else if (req.url === "/items" && req.method === 'POST') {
  const body = await getBodyFromRequest(req);
  const category = getValueFromBody(body, 'category')
  const title = getValueFromBody(body, 'title')
  items = saveItems(items, { title, category });
  res.setHeader('Location', '/items');
  res.writeHead(302);
}
```

What's really great to note here is that you have already tested `getBodyFromRequest` and `getValueFromBody`! That means, out of all that code, there are only two methods for which you must write tests! Those are `mergeItems` and `saveItems`.

Testing the merge items method

This is *really* similar to the `mergeCategories` method that you've now written tests for twice. But, instead of creating an `<li>` or an `<option>`, it creates a row for a table for the items that are passed in and a form that shows a button to complete the item.

Open **merge-items.js** and review that code, please. You can see the loop on lines 5 - 23 that builds the rows of the table. Then, the form is created only if the item is *not* complete. Then, the `<tr>` and its `<td>`s are created. This just means that you will want to test instead of for `<li>`s and `<option>`s, you'll test for many `<td>`s that contain the expected values.

Open **merge-items-spec.js** and see that you have essentially the same tests that you had for the `mergeCategories` function. It may not surprise you to learn that *many* tests look the same, especially if they handle similar functionality. This can get monotonous, at times. It is better, though, to have the protection of tests by investing a little bit of time in writing them as opposed to spending days trying to find a bug that inadvertently got into the code base when someone was writing other code.

The first test

The first test reads

```
it("should return no <tr>s and no <td>s for no items", () => {
  expect.fail('please write this test');
});
```

Replace the `expect.fail` line with a test that properly follows the *Three A*s of unit testing.

In the *arrange* section, you will need to create an empty array for the `items` and store it in a variable. You will use the variable in the action.

In the *act* section, you will invoke the `mergeItems` function with the `template` as the first argument and the variable that contains an empty array as the second argument. Store the return value in a variable.

In the *assert* section, assert that each of the following are true using the include assertion provided by Chai on the result of the *act*:

- To make sure that the method doesn't *remove* the wrong things
  - Assert that it contains the string "<table>"
  - Assert that it contains the string "</table>"
  - Assert that it contains the string "<tbody>"
  - Assert that it contains the string "</tbody>"
- To make sure that the method doesn't *add* the wrong things
  - Assert that it does not contain the string "<tr>"
  - Assert that it does not contain the string "</tr>"
  - Assert that it does not contain the string "<td>"
  - Assert that it does not contain the string "</td>"
- To make sure it replaces what you expect it to replace
  - Assert that it does not contain the string "<!-- Content here -->"

Run the test to make sure it passes.

Because you already have examples of what this looks like in `mergeCategories`, please refer to that.

The second test

The second test reads

```
it("should return a single <tr>, four <td>s, and a <form> for one uncompleted ite
  expect.fail('please write this test');
});
```

Replace the `expect.fail` line with a test that properly follows the *Three A*s of unit testing.

If you look at the code in the `mergeItems` method, you can see that it relies on the item to have the following properties:

- `title`
- `category`
- `isComplete`

In the *arrange* section, you will need to create an array for the `items` argument that contains a single item and store it in a variable. You will use the variable in the action and the value that you typed in the assertion. Something like the following would suffice.

```
const items = [
  { title: 'Title 1', category: 'Category 1' },
];
```

In the *act* section, you will invoke the `mergeItems` function with the `template` as the first argument and the variable that contains an empty array as the second argument. Store the return value in a variable.

In the *assert*section, assert that each of the following are true using theincludeassertion provided by Chai:

- To make sure that the method doesn't *remove*the wrong things
  - Assert that it contains the string "<table>"
  - Assert that it contains the string "</table>"
  - Assert that it contains the string "<tbody>"
  - Assert that it contains the string "</tbody>"
- To make sure that the method *adds*the right things
  - Assert that it contains the string "<tr>"
  - Assert that it contains the string "</tr>"
  - Assert that it contains the string "<td>Title 1</td>"
  - Assert that it contains the string "<td>Category 1</td>"
  - Assert that it contains the string "<form method="POST" action="/items/1">"
- To make sure it replaces what you expect it to replace
  - Assert that it does not contain the string "<!-- Content here -->"

Run the test to make sure it passes.

The third test

Now, you will test that *no*form is created when an item is complete. This will be nearly identical to what you just wrote *except*that your item should have an "isComplete" property set to `true`, and you will assert that it does *not*contain the "<form method="POST" action="/items/1">" string.

Replace the `expect.fail`line with a test that properly follows the *Three A*sof unit testing.

In the *arrange*section, you will need to create an array for the `items`argument that contains a single item that is completed and store it in a variable. Something like the following would suffice.

```
const items = [
  { title: 'Title 1', category: 'Category 1', isComplete: true },
];
```

In the *act*section, you will invoke the `mergeItems`function with the`template`as the first argument and the variable that contains an empty array as the second argument. Store the return value in a variable.

In the *assert*section, assert that each of the following are true using theincludeassertion provided by Chai:

- To make sure that the method doesn't *remove*the wrong things
  - Assert that it contains the string "<table>"
  - Assert that it contains the string "</table>"
  - Assert that it contains the string "<tbody>"
  - Assert that it contains the string "</tbody>"
- To make sure that the method *adds*the right things

    - Assert that it contains the string "<tr>"
    - Assert that it contains the string "</tr>"
    - Assert that it contains the string "<td>Title 1</td>"
    - Assert that it contains the string "<td>Category 1</td>"

- To make sure that the method does not add the wrong things
    - Assert that it does not contain the string "<form method="POST" action="/items/1">"

- To make sure it replaces what you expect it to replace
    - Assert that it does not contain the string "<!-- Content here -->"

Run the test to make sure it passes.

The fourth test

Now, try writing the last test `it('should return three <tr>s for three items')` as a combination or extension of the previous two. Check to make sure that you get all of the indexes for the items that you have in your array. Make sure that the "form" elements appear for those items that are not complete.

Testing the save items method

Open the **save-items.js**file and review the function. It merely adds a new item to the array passed in using the `push`method. Then, it creates a clone of the old array using the "spread operator". If you're not familiar with that syntax, don't worry. All it does is make a copy of the array.

Open the **save-items-spec.js**. You will see two methods in there. These are nearly identical to the first and last tests for the `saveCategories`method. Use the same pattern to complete those tests.

Just as a reminder, a solution for the **save-categories-spec.js**file (with comments removed) could look like this.

```
describe("The saveCategories function", () => {
  it('adds the new category to the list', () => {
    const categories = ['Cat 3', 'Cat 2'];
    const newCategory = 'Cat 1';
    const result = saveCategories(categories, newCategory);
    expect(result).to.contain(newCategory);
  });

  it('makes sure the result and the original are different', () => {
    const categories = ['Cat 3', 'Cat 2'];
    const result = saveCategories(categories, 'Cat 1');
    expect(result).to.not.equal(categories);
  });
});
```

Your code will look a lot like this *except*you should have arrays of items and new items, not strings. Remember from the last section, an array of items might look like this:

```
const items = [
  { title: 'Title 1', category: 'Category 1' },
];
```

Run your tests to make sure they pass.

Show And Complete A To-Do Item

You're almost done with testing this application! Have a look at the method that completes a to-do item.

```
else if (req.url.startsWith('/items/') && req.method === 'POST') {
  const index = Number.parseInt(req.url.substring(7)) - 1;
  items[index].isComplete = true;
  res.setHeader('Location', '/items');
  res.writeHead(302);
}
```

That's interesting. There's nothing to test there, no methods. That's some kind of wonderful! On to the next item!

Search For To-Do Items

What may be the most complex set of tests to write (except that weird event emitter thing), search makes you think though what it should do in a variety of cases.

Here's the relevant part of the server that handles a search query.

```
else if (req.url.startsWith('/search') && req.method === 'GET') {
  const [_, query] = req.url.split('?', 2);
  const { term } = querystring.parse(query);
  const filePath = path.join(__dirname, 'search-items-screen.html');
  const template = await fs.promises.readFile(filePath, 'utf-8');
  let foundItems = [];
  if (term) {
    foundItems = searchItems(items, term);
  }
  const html = mergeItems(template, foundItems);
  res.setHeader('Content-Type', 'text/html');
  res.writeHead(200);
  res.write(html);
}
```

You've already tested `mergeItems`, so that's not needed, again. The only method that you will need to test is `searchItems`.

Open **search-items.js** and review how that code is working. It takes a list of `items` and a search `term`. The first thing it does is force the term to lower case.

```
term = term.toLowerCase();
```

Then, it uses the `filter` function on the array to create a new array of items that meet the comparison in the function. The comparison function makes the title lower case and checks to see if the term is contained in that string.

```
return items.filter(x => {
  const title = x.title.toLowerCase();
  return title.indexOf(term) >= 0;
});
```

If the term *is* in the title, then the comparison returns `true` and the `filter` function will add it to the new array. If the term is *not* in the title, the comparison returns `false` and it is not added to the new array.

Here is an example. Supposed you have the following items in your array.

```
[
  { title: 'Go grocery shopping', category: 'Home' },
  { title: 'Play with my puppy', category: 'Pet' },
  { title: 'Shop for a puppy bed', category: 'Pet' },
]
```

Now, say the search term someone entered is "SHOP". This is what happens in the function.

```
Convert "SHOP" to "shop"
Filter the array of items based on the term "shop":
  Item 1:
    Convert "Go grocery shopping" to "go grocery shopping"
    Does it contain the term "shop"? YES
    Add it to the new array
  Item 2:
    Convert "Play with my puppy" to "play with my puppy"
    Does it contain the term "shop"? NO
  Item 3:
    Convert "Shop for a puppy bed" to "shop for a puppy bed"
    Does it contain the term "shop"? YES
    Add it to the new array
Return the new array that contains items 1 and 3
```

So, that's what you want to test for.

Open **search-items-spec.js**. You'll see three tests.

In the first test, you are asked to fix the *arrange*step to declare `items`and `term`given the directions. This is not a trick. It's just declaring those two variables that it's asking you to create.

In the second test, fix the *assert*step to assert the proper length of the result by completely replacing the `expect.fail`line.

In the third test, you are asked to fix the *arrange*step by choosing a string value for `term`that makes the rest of the test pass.

What have you done?

Now that you've done that, you've won the entire game! All of the meaty logic of the game is now well tested. If someone were to come along and try to change the code, when the tests ran, it would check to make sure they didn't accidentally break something in their earnest to add new functionality!

Here's what you did:

- You've looked at, read, and understood other people's code
- You've seen and used a variety of assertions
- You've seen how to do real (not fake) asynchronous testing using the `done`method
- You've invested time in hardening the maintainability of an application

Here's a link to a solution.https://appacademy-open-assets.s3-us-west-1.amazonaws.com/Module-JavaScript/testing/projects/testing-an-existing-project-solution.zip

In the next step, you're going to use the fact that you have tests to radically change the code.

Refactor To Use A Template Engine

Note: the solution project does not have this step included in it because it changes earlier tests.

The hard work you've done with `mergeCategories` and `mergeItems` to make sure that the important parts of the HTML are generated, those are some really good tests that you're now going to use to change the way the entire HTML is generated.

This is the other side of testing. When you can feel confident that what you are doing will not break the code because you have tests that tell you what to do.

You're going to follow some steps to replace the HTML-generating portion of the application. The steps will be explicit, because this is less about learning a library as it is proving to yourself that tests are a good thing.

Hello, handlebars

There's very little chance that you would create a Web application, anymore, and generate your own HTML the way it was done in the `mergeCategories` and `mergeItems` functions. Instead, you would use a "templating engine" which is a library that takes some template (with some fancy instructions) and some data and generates HTML *for you*.

You'll change your tests to use a handlebars style template which looks nearly identical to HTML. This will break your tests. Then, you will change the functions to use the handlebars engine. Then, you will know that they properly handle handlebars templates, so you'll change the HTML files to use that syntax rather than using the "<!-- Content here -->" placeholder.

All you need to know about handlebars

This is just an informative section so you know what will actually be going on in the tests. You're not going to be asked to come up with any of this yourself. This is showing you how you would, in the real world, update existing code and tests in a real application.

When you use the handlebars engine, you pass it two things, a string that contains the template and an object that contains the data that you want to show.

Assume that this is your data object.

```
const data = {
  name: 'Remhai',
  nicknames: [ 'R', 'Rem', 'Remrem' ],
  addresses: [
    { street: '123 Main St', city: 'Memphis', state: 'TN' },
    { street: '2000 9th Ave', city: 'New York', state: 'NY' },
  ],
};
```

In your template, if you want to output the value in the `name` property, you just put the name of the property in double curly brackets.

```
<div>
  Name: {{ name }}
</div>
```

In your template, if you want to output all of the nicknames of the person, you loop over that property using the `#each` helper like this. Then, inside the `#each` "block", you refer to the value of the string itself as `this`.

```
<ul>
  {{#each nicknames}}
    <li>{{ this }}</li>
  {{/each}}
</ul>
```

In your template, if you want to output all of the addresses of the person, you loop over the property using the `#each`helper in which you will use the property names of the objects inside the array. You can use `@index`to give you the current index.

```
<tbody>
  {{#each addresses}}
    <tr>
      <td>{{ @index }}</td>
      <td>{{ street }}</td>
      <td>{{ city }}</td>
      <td>{{ state }}</td>
    </tr>
  {{/each}}
</tbody>
```

If you want to do a conditional, you can just do something like this.

```
{{#if isVisible}}
  <div>You can see me!</div>
{{else}}
  <div></div>
{{/if}}
```

So, that's *handlebars*. Again, it's just so that you can understand the syntax of the tests and HTML that you'll be changing.

Install handlebars

You just need to use `npm` to do this. `npm install handlebars`. Yay!

To do some math, you'll need to install some handlebars helpers. You just need to use `npm` to do this. `npm install handlebars-helpers`. Yay!

Now, change your merge items test

Inside **merge-items-spec.js**, you will change the `template`string. And, that is all you'll change. Instead of having the "<!-- Content here -->", you'll replace that with handlebars code. Then, your tests will fail. Then, you'll make then pass. Once they pass, you'll know you're safe to change the *real*HTML file.

Update the template from what it is now to the following code.

```
const template = `
  <table>
    <tbody>
      {{#each items}}
        <tr>
          <td>{{ add @index 1 }}</td>
          <td>{{ title }}</td>
          <td>{{ category }}</td>
          <td>
            {{#if isComplete}}
            {{else}}
              <form method="POST" action="/items/{{ add @index 1 }}">
                <button class="pure-button">Complete</button>
              </form>
            {{/if}}
          </td>
        </tr>
      {{/each}}
    </tbody>
```

```
  </table>
`;
```

This seems like a lot when compared to the other template we had. However, this moves all of the HTML-generation to the template. There will be no looping and string manipulation in the `mergeItems` function after you're done with it.

Run your tests and make sure they fail. Without a failing test, you don't know what to fix. And, in this case, all three tests fail.

Fix the merge items function

Now that you have this, it's time to update the `mergeItems` function. You'll know you're done when the tests all pass.

Inside `mergeItems`, import the *handlebars* library at the top of your file, the helpers library, and then register the 'math' helpers with the handlebars library according to the helpers documentation. We need this so we can use the `add` helper in the template to add `1` to the `@index`.

```
const handlebars = require('handlebars');
const helpers = require('handlebars-helpers');
helpers.math({ handlebars });
```

Now, delete *everything* inside the function. Replace it with the following lines.

```
const render = handlebars.compile(template);
return render({ items });
```

You've moved the complexity of the HTML generation from the source code to the HTML code. HTML code is easier to change because it's only about the display and generally won't crash your entire application.

Now, change your merge categories tests

Open **merge-categories-spec.js**. Change the first template to this code.

```
const template = `
  <div>
    <ul>
      {{#each categories}}
        <li>{{ this }}</li>
      {{/each}}
    </ul>
  </div>
`;
```

Change the second template to this code.

```
const template = `
  <div>
    <select>
      {{#each categories}}
        <option>{{ this }}</option>
      {{/each}}
    </select>
  </div>
`;
```

Now, your merge categories tests should not work. Run them to make sure.

Fix the merge categories function

Open **merge-categories.js**and import just *handlebars*. There's no math in the templates, so there's no need for the helpers.

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

Again, delete *everything*inside the function and replace it with the following to make the tests pass, again.

```
const render = handlebars.compile(template);
return render({ categories });
```

AMAZING!

What just happened?

You just performed a major refactor of the application and you knew you did it because you had tests to guide you during the refactor!

Here's an even more amazing thing. You did it without actually running the code! You did it because you had tests that told you if the inputs and outputs matched your expectations!

Now, you can change the content of the HTML pages to use the new *handlebars*syntax. There are only four of them, and you can use the stuff from your tests to update the source code.

Open **category-list-screen.html**and replace the "<!-- Content here -->" with this handlebars syntax lifted straight from the tests.

```
{{#each categories}}
  <li>{{ this }}</li>
{{/each}}
```

Open **list-of-items-screen.html**and replace the "<!-- Content here -->" with this handlebars syntax lifted straight from the tests.

```
{{#each items}}
  <tr>
    <td>{{ add @index 1 }}</td>
    <td>{{ title }}</td>
    <td>{{ category }}</td>
    <td>
      {{#if isComplete}}

      {{else}}
        <form method="POST" action="/items/{{ add @index 1 }}">
          <button class="pure-button">Complete</button>
        </form>
      {{/if}}
    </td>
  </tr>
{{/each}}
```

Open **search-items-screen.html**and replace the "<!-- Content here -->" with this handlebars syntax lifted straight from the tests.

```
{{#each items}}
  <tr>
    <td>{{ add @index 1 }}</td>
    <td>{{ title }}</td>
    <td>{{ category }}</td>
    <td>
      {{#if isComplete}}

      {{else}}
        <form method="POST" action="/items/{{ add @index 1 }}">
```

```
          <button class="pure-button">Complete</button>
        </form>
      {{/if}}
    </td>
  </tr>
{{/each}}
```

Open **todo-form-screen.html**and replace the "<!-- Content here -->" with this handlebars syntax lifted straight from the tests.

```
{{#each categories}}
  <option>{{ this }}</option>
{{/each}}
```

That completes the upgrade of the HTML files. You can run your server using`node server.js`and checkout that everything just works by going tohttp://localhost:3000/items.

Just another note

This may not seem very amazing to you. This unit testing *revolutionized*the way that programmers write software! The fact that you could go into a code base and run tests to see how code works, change code and know that you haven't broken anything, that enabled people to work more confidently that they were not introducing bugs into the software as they were going along.

This is the stuff dreams are made of.

So, good work. Good work