**Introduction**

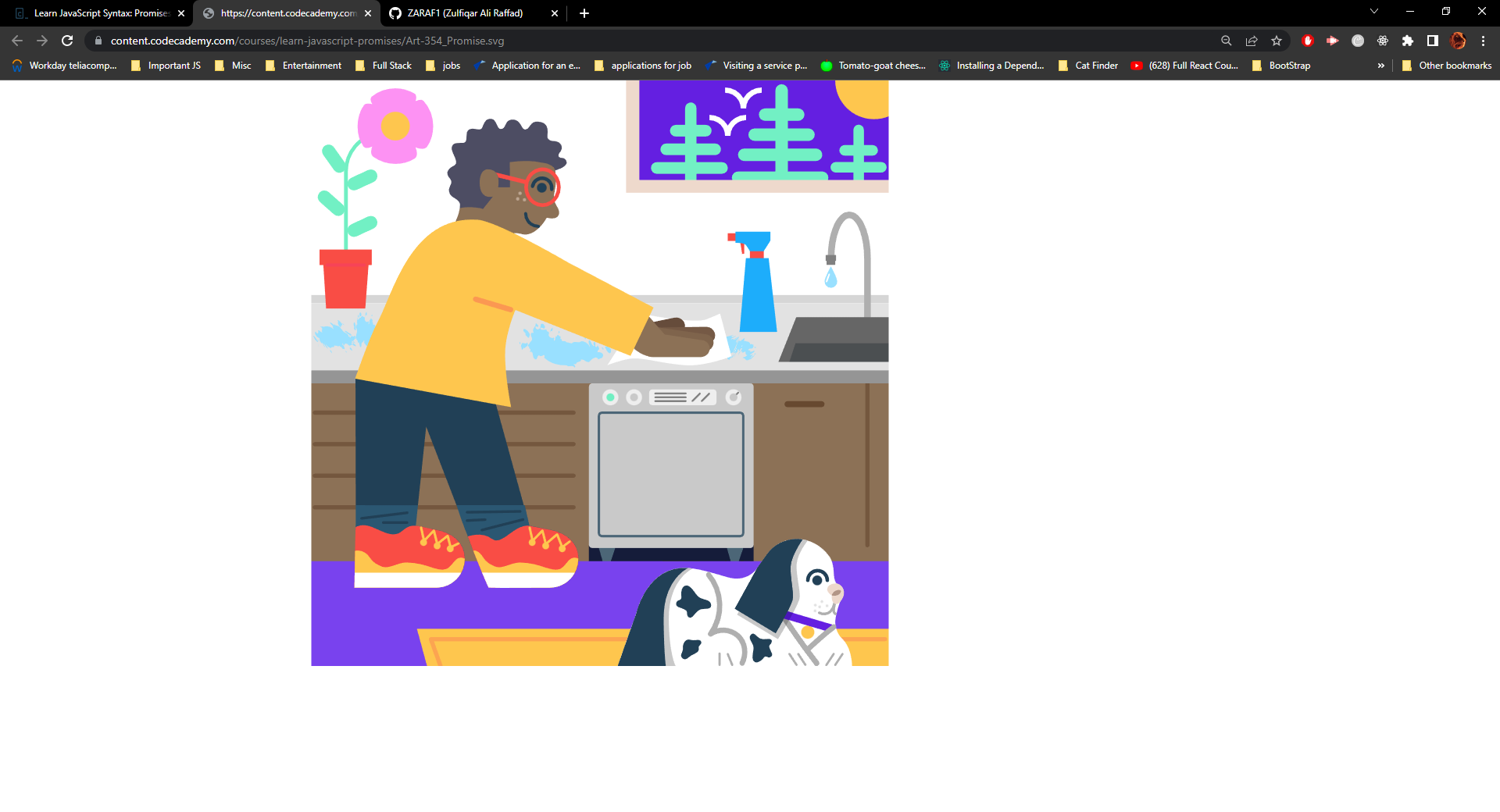
In web development, asynchronous programming is notorious for being a challenging topic.

An *asynchronous operation* is one that allows the computer to “move on” to other tasks while waiting for the asynchronous operation to complete. Asynchronous programming means that time-consuming operations don’t have to bring everything else in our programs to a halt.

There are countless examples of asynchronicity in our everyday lives. Cleaning our house, for example, involves asynchronous operations such as a dishwasher washing our dishes or a washing machine washing our clothes. While we wait on the completion of those operations, we’re free to do other chores.

Similarly, web development makes use of asynchronous operations. Operations like making a network request or querying a database can be time-consuming, but JavaScript allows us to execute other tasks while awaiting their completion.

This lesson will teach you how modern JavaScript handles asynchronicity using the Promise object, introduced with ES6. Let’s get started!



**What is a Promise?**

Promises are objects that represent the eventual outcome of an asynchronous operation. A Promise object can be in one of three states:

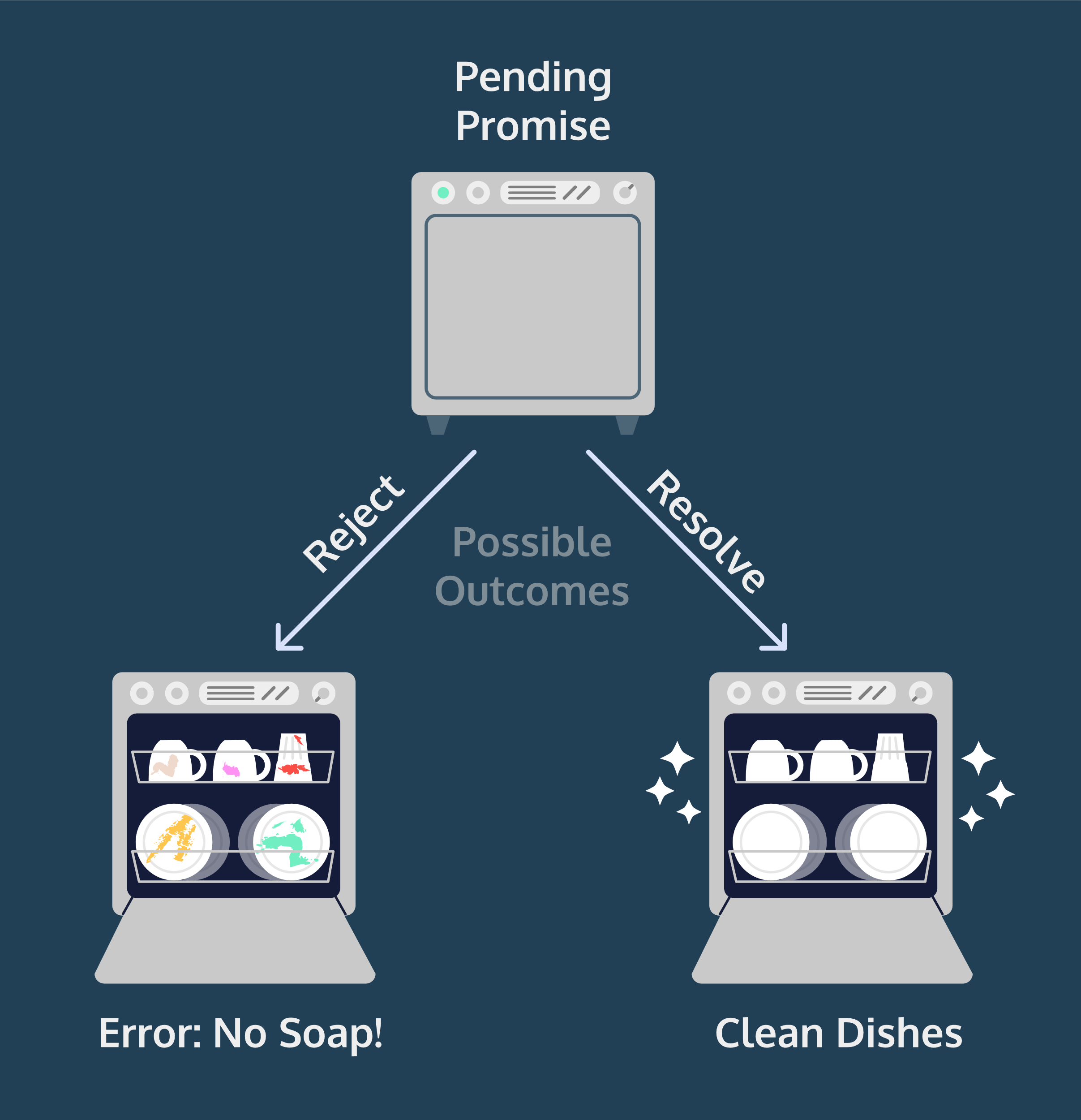
* **Pending**: The initial state— the operation has not completed yet.
* **Fulfilled**: The operation has completed successfully and the promise now has a *resolved value*. For example, a request’s promise might resolve with a JSON object as its value.
* **Rejected**: The operation has failed and the promise has a reason for the failure. This reason is usually an Error of some kind.

We refer to a promise as *settled* if it is no longer pending— it is either fulfilled or rejected. Let’s think of a dishwasher as having the states of a promise:

* **Pending**: The dishwasher is running but has not completed the washing cycle.
* **Fulfilled**: The dishwasher has completed the washing cycle and is full of clean dishes.
* **Rejected**: The dishwasher encountered a problem (it didn’t receive soap!) and returns unclean dishes.

If our dishwashing promise is fulfilled, we’ll be able to perform related tasks, such as unloading the clean dishes from the dishwasher. If it’s rejected, we can take alternate steps, such as running it again with soap or washing the dishes by hand.

All promises eventually settle, enabling us to write logic for what to do if the promise fulfills or if it rejects.



**Constructing a Promise Object**

Let’s construct a promise! To create a new Promise object, we use the new keyword and the Promise constructor method:

const executorFunction = (resolve, reject) => { };  
const myFirstPromise = new Promise(executorFunction);

The Promise constructor method takes a function parameter called the *executor function* which runs automatically when the constructor is called. The executor function generally starts an asynchronous operation and dictates how the promise should be settled.

The executor function has two function parameters, usually referred to as the resolve() and reject() functions. The resolve() and reject() functions aren’t defined by the programmer. When the Promise constructor runs, JavaScript will pass **its own** resolve() and reject() functions into the executor function.

* resolve is a function with one argument. Under the hood, if invoked, resolve() will change the promise’s status from pending to fulfilled, and the promise’s resolved value will be set to the argument passed into resolve().
* reject is a function that takes a reason or error as an argument. Under the hood, if invoked, reject() will change the promise’s status from pending to rejected, and the promise’s rejection reason will be set to the argument passed into reject().

Let’s look at an example executor function in a Promise constructor:

const executorFunction = (resolve, reject) => {  
  if (someCondition) {  
      resolve('I resolved!');  
  } else {  
      reject('I rejected!');   
  }  
}  
const myFirstPromise = new Promise(executorFunction);

Let’s break down what’s happening above:

* We declare a variable myFirstPromise
* myFirstPromise is constructed using new Promise() which is the Promise constructor method.
* executorFunction() is passed to the constructor and has two functions as parameters: resolve and reject.
* If someCondition evaluates to true, we invoke resolve() with the string 'I resolved!'
* If not, we invoke reject() with the string 'I rejected!'

In our example, myFirstPromise resolves or rejects based on a simple condition, but, in practice, promises settle based on the results of asynchronous operations. For example, a database request may fulfill with the data from a query or reject with an error thrown. In this exercise, we’ll construct promises which resolve synchronously to more easily understand how they work.

**The Node setTimeout() Function**

Knowing how to construct a promise is useful, but most of the time, knowing how to *consume*, or use, promises will be key. Rather than constructing promises, you’ll be handling Promise objects returned to you as the result of an asynchronous operation. These promises will start off pending but settle eventually.

Moving forward, we’ll be simulating this by providing you with functions that return promises which settle after some time. To accomplish this, we’ll be using setTimeout(). setTimeout() is a Node API (a comparable API is provided by web browsers) that uses callback functions to schedule tasks to be performed after a delay. setTimeout() has two parameters: a callback function and a delay in milliseconds.

const delayedHello = () => {  
  console.log('Hi! This is an asynchronous greeting!');  
};  
   
setTimeout(delayedHello, 2000);

Here, we invoke setTimeout() with the callback function delayedHello() and 2000. In at least two seconds delayedHello() will be invoked. But why is it “at least” two seconds and not exactly two seconds?

This delay is performed asynchronously—the rest of our program won’t stop executing during the delay. Asynchronous JavaScript uses something called *the event-loop*. After two seconds, delayedHello() is added to a line of code waiting to be run. Before it can run, any synchronous code from the program will run. Next, any code in front of it in the line will run. This means it might be more than two seconds before delayedHello() is actually executed.

Let’s look at how we’ll be using setTimeout() to construct asynchronous promises:

const returnPromiseFunction = () => {  
  return new Promise((resolve, reject) => {  
    setTimeout(( ) => {resolve('I resolved!')}, 1000);  
  });  
};  
   
const prom = returnPromiseFunction();

In the example code, we invoked returnPromiseFunction() which returned a promise. We assigned that promise to the variable prom. Similar to the asynchronous promises you may encounter in production, prom will initially have a status of pending.

Let’s explore setTimeout() a bit more.

**Instructions**

**1.**

Create a function, usingSTO(). Your usingSTO() function should have no parameters. Inside the function body, it should print a string to the console. This can be any string you want (as long as it’s not either “This is the first line of synchronous code.” or “This is the last line of synchronous code.”).

Check your work to move on to the next step.

Stuck? Get a hint

**2.**

Now, let’s invoke the setTimeout() function. Remember, setTimeout() has two parameters. Invoke setTimeout() with your usingSTO() function as the first argument and a number between 0 and 3000 as the second argument.

Stuck? Get a hint

**3.**

Take a moment to predict the output of this program.

Whenever you’re ready, type node app.js in the terminal and hit enter.

See if the program’s output in the terminal lines up with what you expected.