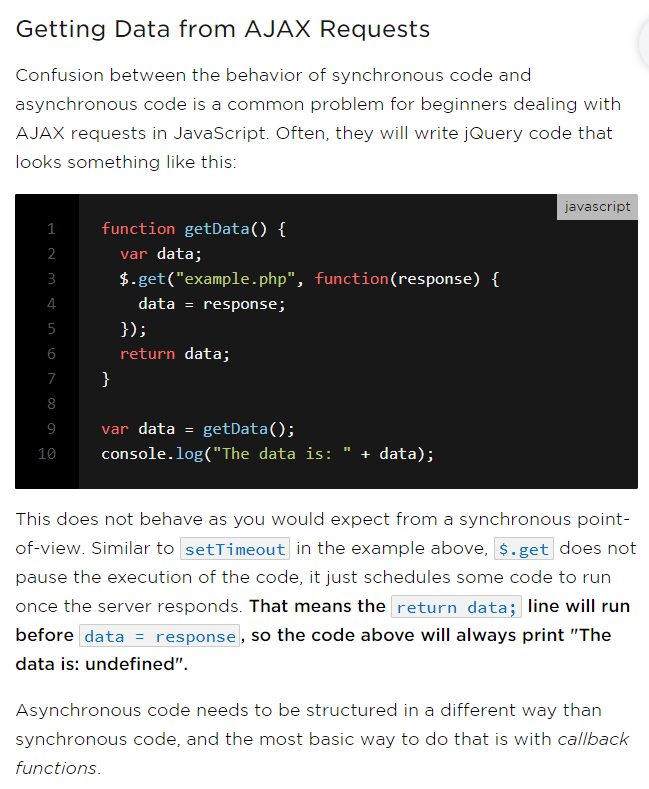
## Promises

What are promises in ES6, why you need them and how do you use them.

When you execute a task *synchronously*, you wait for it to finish before moving on to the next line of code.

When you execute a task *asynchronously*, the program moves to the next line of code before the task finishes.

NOTE: Why do we require asynchronous programming



Think of synchronous programming like waiting in line and asynchronous programming like taking a ticket. When you take a ticket you can go do other things and then be notified when ready.

**Callbacks**

One way to program asynchronously is to use *callbacks(*this is classic mechanism*)*. We pass to an asynchronous function a function which it will call when the task is completed.

console.log("After setTimeout");

function doAsyncTask(cb) {

setTimeout(() => {

console.log("Async Task Calling Callback");

cb();

}, 60000);

}

doAsyncTask(() => console.log("Callback Called"));

RESULT OF THE ABOVE PROGRAM

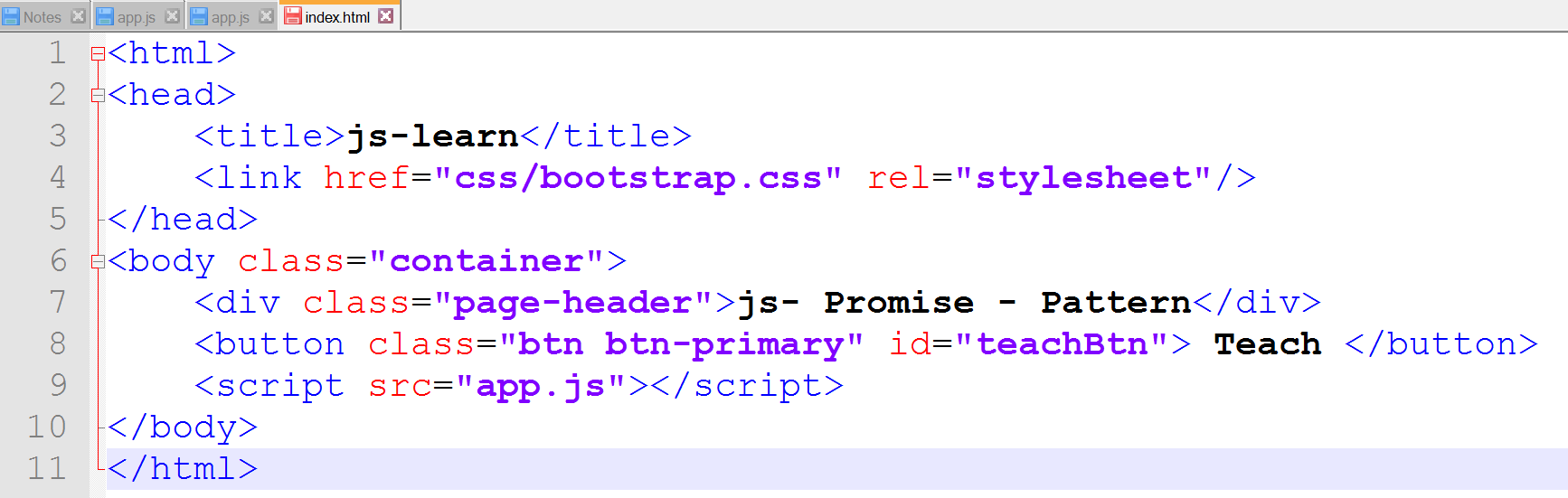
In Console

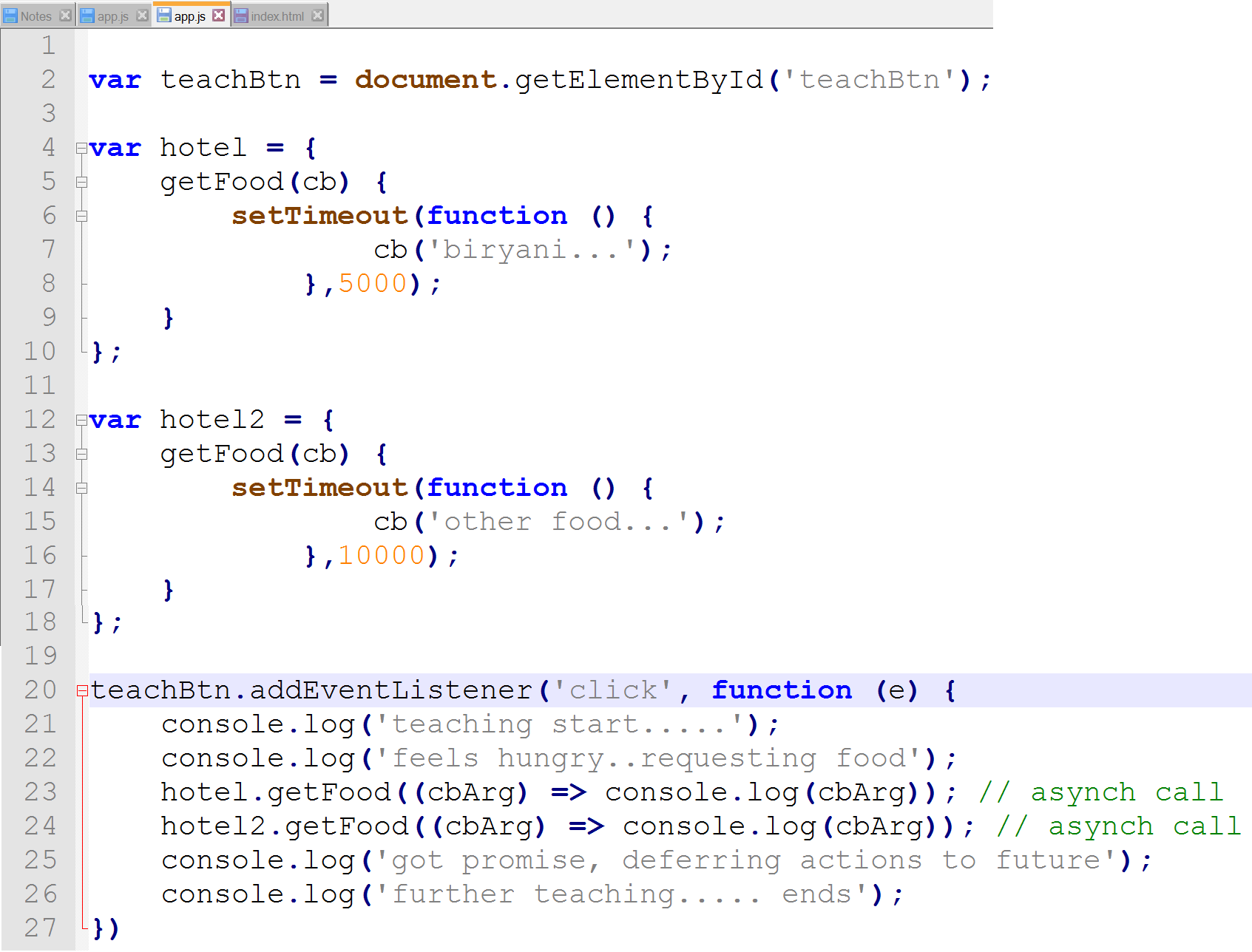
After setTimeout - displayed at 10 HH :00 MM

Async Task Calling Callback- displayed at 10 HH :01 MM

Callback Called- displayed at 10 HH :01 MM

ANOTHER SOME WHAT REASONABLE EXAMPLE WHICH WE CAN TRANSLATE EASILY WITH PROMISES (My Example – From Hexagon training)

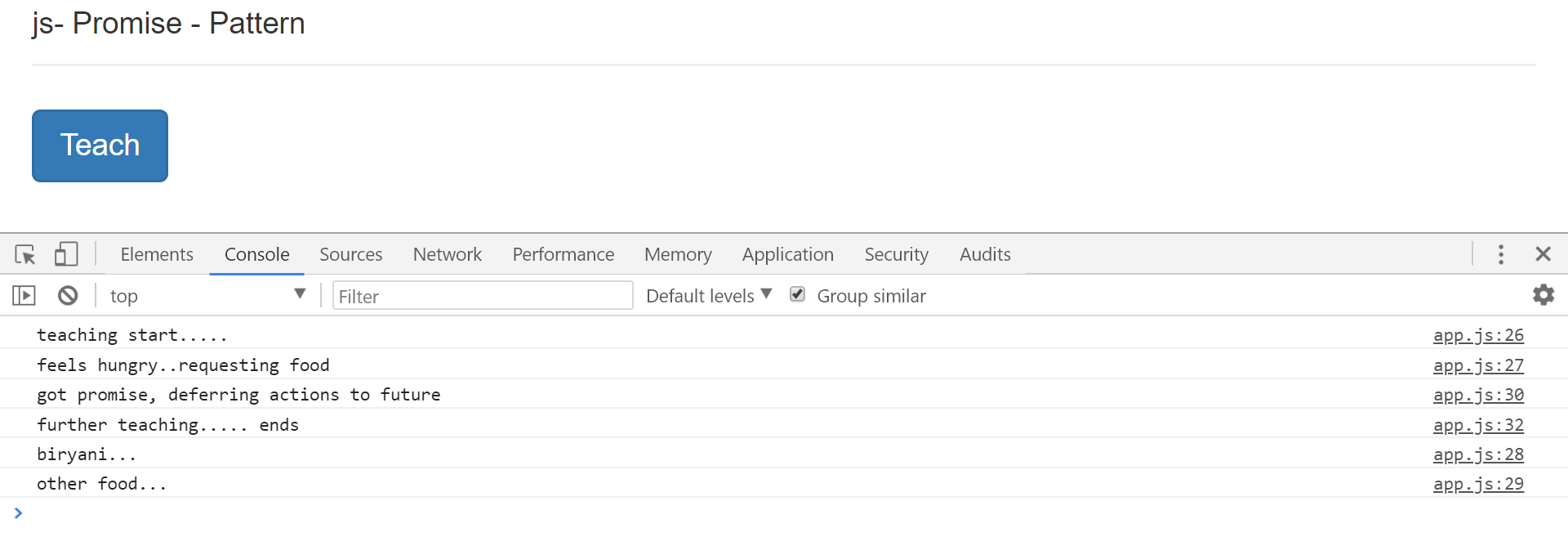




RESULT

Biryani appears after 5 secs

Otherfood appears after 10 secs



The doAsyncTask function when called kicks of an asynchronous task and *returns immediately*. So doAsyncTask function is called Asynchronous function.

To get notified when the async task completes we pass to doAsyncTask a function(called as callback function) which it will call when the task completes.

It’s called a *callback* function, cb for short, because it *calls-you-back*.

The above approach is traditional way of handling asynchronous work.

**Promise API (REAL TIME EXAMPLE AT THE END)**

In ES6 we have an alternative mechanism built into the language called a *promise*.

A *promise* is a *placeholder* for a future value.

**It serves the same function as callbacks but has a nicer syntax and makes it easier to handle errors**.

NOTE: Promises are not a feature of only ES6. With capable of doing promises and using promises in ES5 javascript doing that we may have to use third party libraries. With ES6 PROMISE API has become the core part of the javascript language.

**Creating a Promise**

We create an instance of a promise by calling new on the Promise class, like so:

var promise = new Promise((resolve, reject) => {

})

;

Note: Now we are using fat arrow syntax for anonymous functions.

We pass to Promise an inner function that takes two arguments (resolve, reject).

Since we are defining the function we can call these arguments whatever we want but the convention is to call them resolve and reject.

resolve and reject are in fact functions themselves.

Inside this inner function we perform our asynchronous processing and then when we are ready we call resolve(), like so:

i.e. resolve is like callback function.

var promise = new Promise((resolve, reject) => {

setTimeout(() => {

console.log("Async Work Complete");

resolve();

}, 1000);

})

;

function doAsyncTask() {

var promise = new Promise((resolve, reject) => {

setTimeout(() => {

console.log("Async Work Complete");

resolve();

}, 1000);

});

return promise;

}

If there was an error in the async task then we call the reject() function like so:f

Note: error = can be a Boolean value which is set to true in the asynchronous task (or) work

function doAsyncTask() {

var promise = new Promise((resolve, reject) => {

setTimeout(() => {

console.log("Async Work Complete");

if (error) {

reject();

} else {

resolve();

}

}, 1000);

});

return promise;

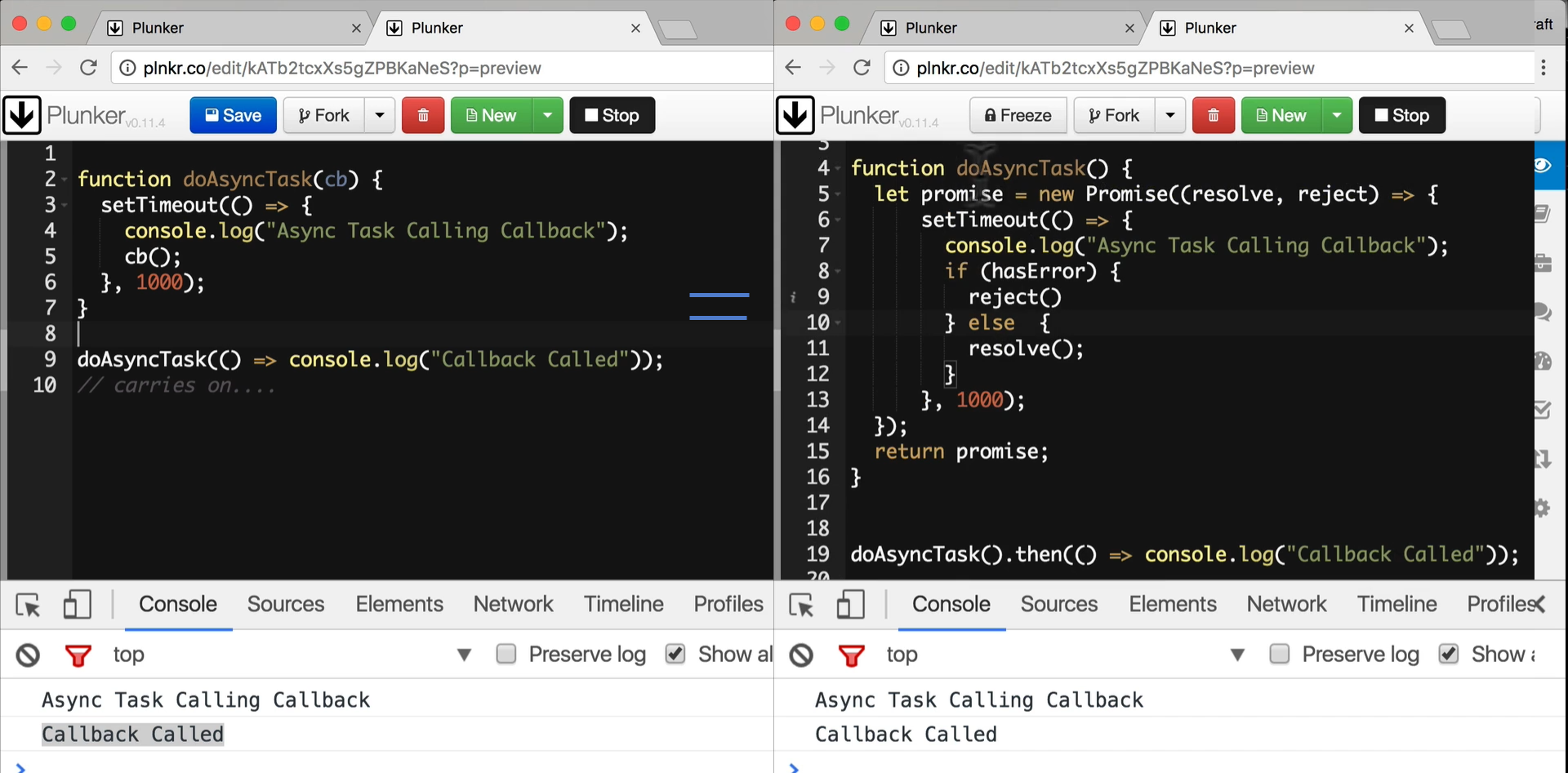
}

**Promise Notifications**

How do we get notified when a Promise resolves. We can get notified when a promise resolves by attaching a *success* handler to its then function, like so:

doAsyncTask().then(() => console.log("Task Complete!"));

We have the same functionality as below (both of the below programs results in same output)



then can take two arguments, the second argument is a *error* handler that gets called if the promise is rejected, like so:

doAsyncTask().then(

() => console.log("Task Complete!"),

() => console.log("Task Errored!"),

)

;

i.e. 1st argument – success handler and the 2nd argument – error handler.

Any values we pass to the resolve and reject functions are passed along to the *error* and *success* handlers, like so:

let error = true;

function doAsyncTask() {

return new Promise((resolve, reject) => {

setTimeout(() => {

if (error) {

reject('error'); // pass values

} else {

resolve('done'); // pass values

}

}, 1000);

});

}

doAsyncTask().then(

(val) => console.log(val),

(err) => console.error(err)

)

;

**Immediate Resolution or Rejection**

We can create an immediately *resolved* Promise by using the Promise.resolve() method, like so:

let promise = Promise.resolve('done');

And an immediately *rejected* Promise by using the Promise.reject() method, like so:

let promise = Promise.reject('fail');

One of the nice things about Promises is that if we add a then handler **after** the promise resolves or rejects the handler **still** gets called.

let promise = Promise.resolve('done');

promise.then((val) => console.log(val)); // 'done'

In the above example, even though the Promise has resolved *before* we added the success handler, the promise framework still calls the success handler.

**Chaining**

We can also connect a series of then handlers together in a chain, like so:

Promise.resolve("done")

.then(

(val) => {

console.log(val);

return 'done2';

},

(err) => console.error(err)

)

.then(

(val) => console.log(val),

(err) => console.error(err)

);

// 'done'

// 'done2'

Promises pass an error along the chain till it finds an error handler. So we don’t need to define an error handler for each then function, we can just add one at the end like so:

Promise.reject('fail')

.then((val) => console.log(val))

.then(

(val) => console.log(val),

(err) => console.error(err)

);

If we throw an exception from our promise function or one of the success handlers, the promise gets rejected and the error handler is called, like so:

Promise.resolve('done')

.then((val) => {

throw new Error("fail")

})

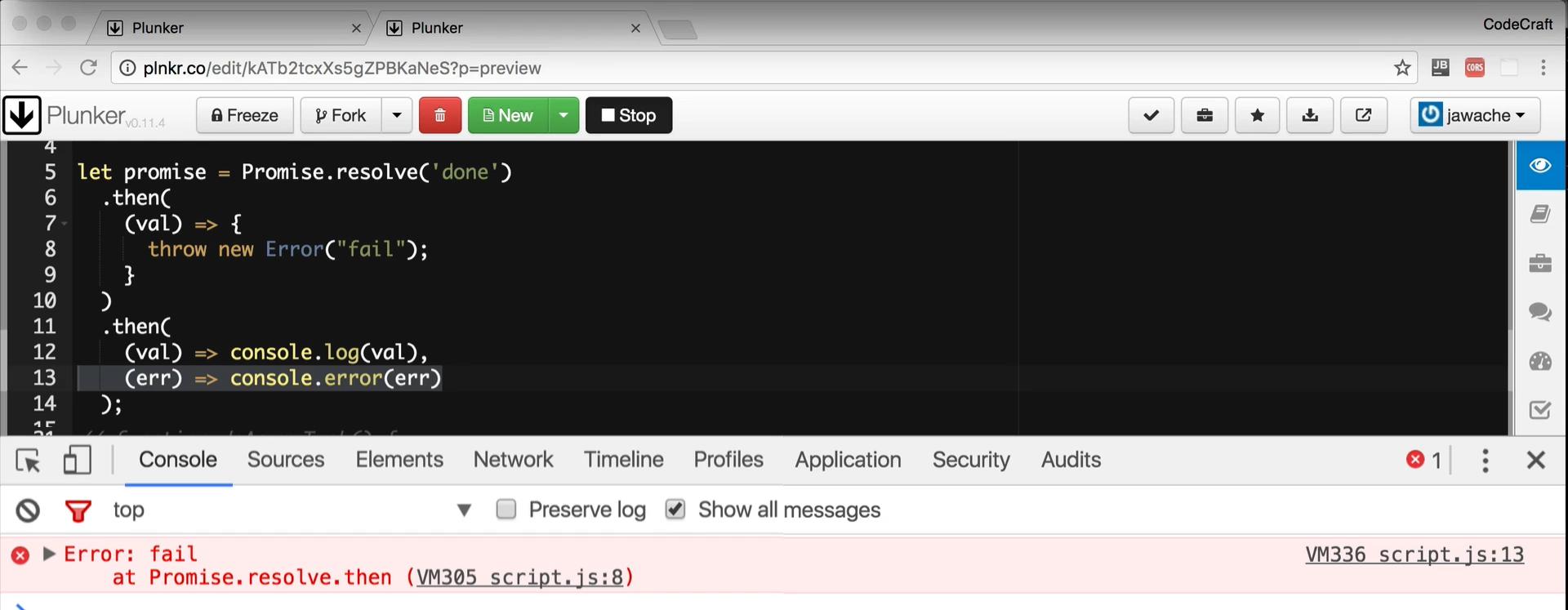
.then(

(val) => console.log(val),

(err) => console.error(err)

);

// [Error: fail]



**Catch**

The catch function works exactly the same way as the then error handler, it’s just clearer and more explicitly describes our intent to handle errors.

Promise.resolve('done')

.then((val) => {throw new Error("fail")})

.then((val) => console.log(val))

.catch((err) => console.error(err));

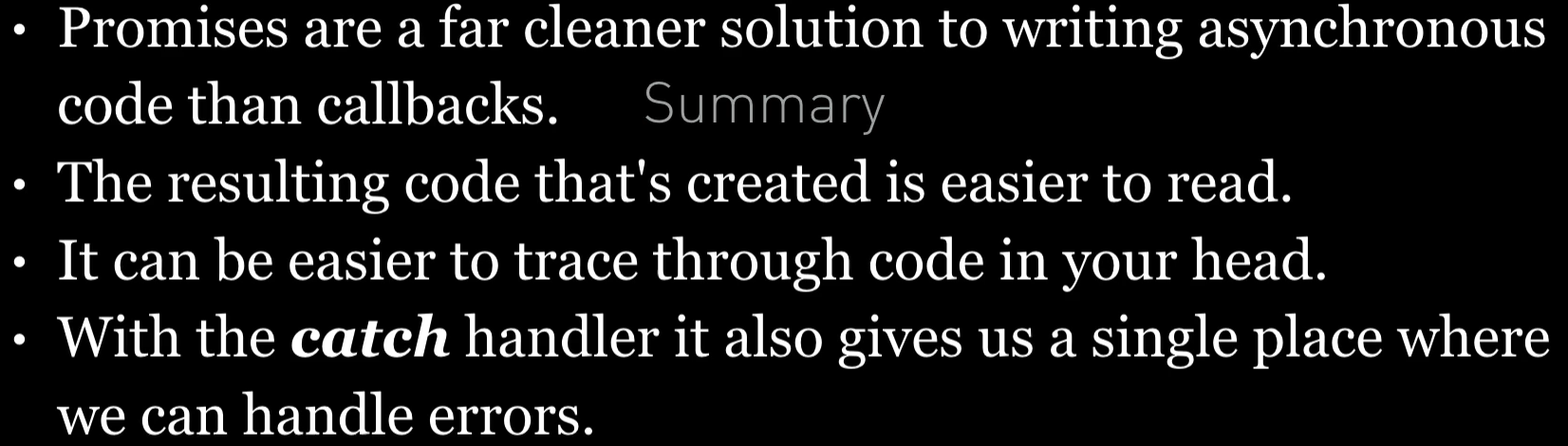
**Summary**

Promises are a far cleaner solution to writing asynchronous code than callbacks.

The resulting code that’s created is easier to read and is often written the order the application will execute.

So it can be easier to trace through code in your head.

With the catch handler it also gives us a single place where we can handle errors.



**Listing**

<http://plnkr.co/edit/OMGgN6qpED6wTEEP2XYN?p=preview>

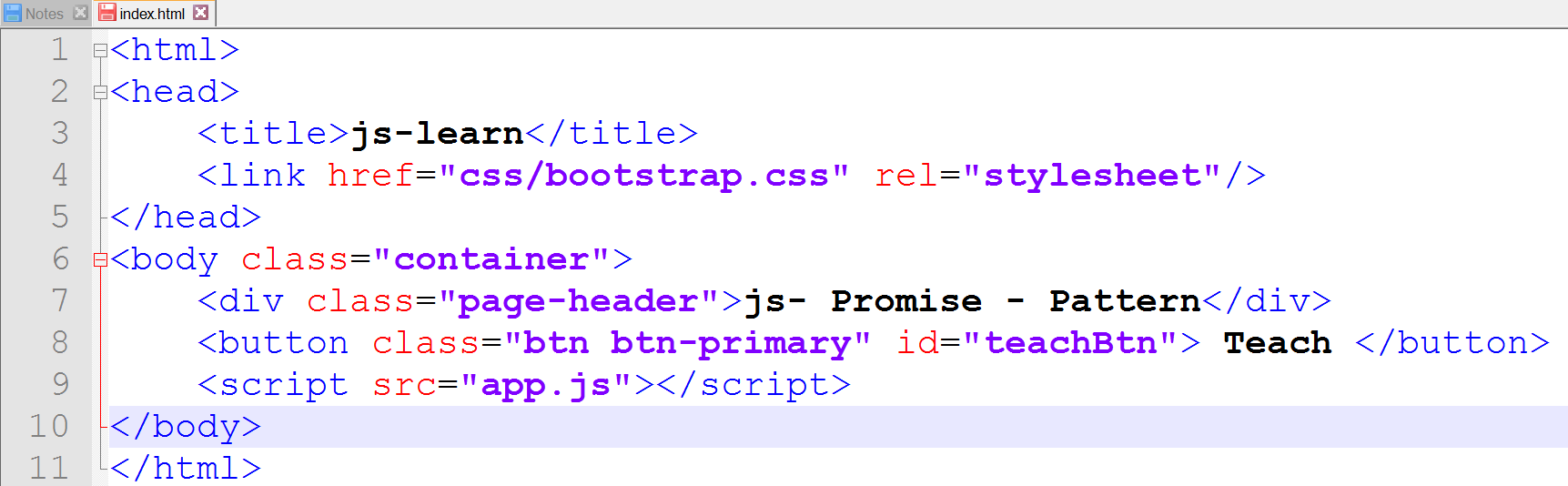
VERY GOOD EXAMPLE FOR PROMISE API

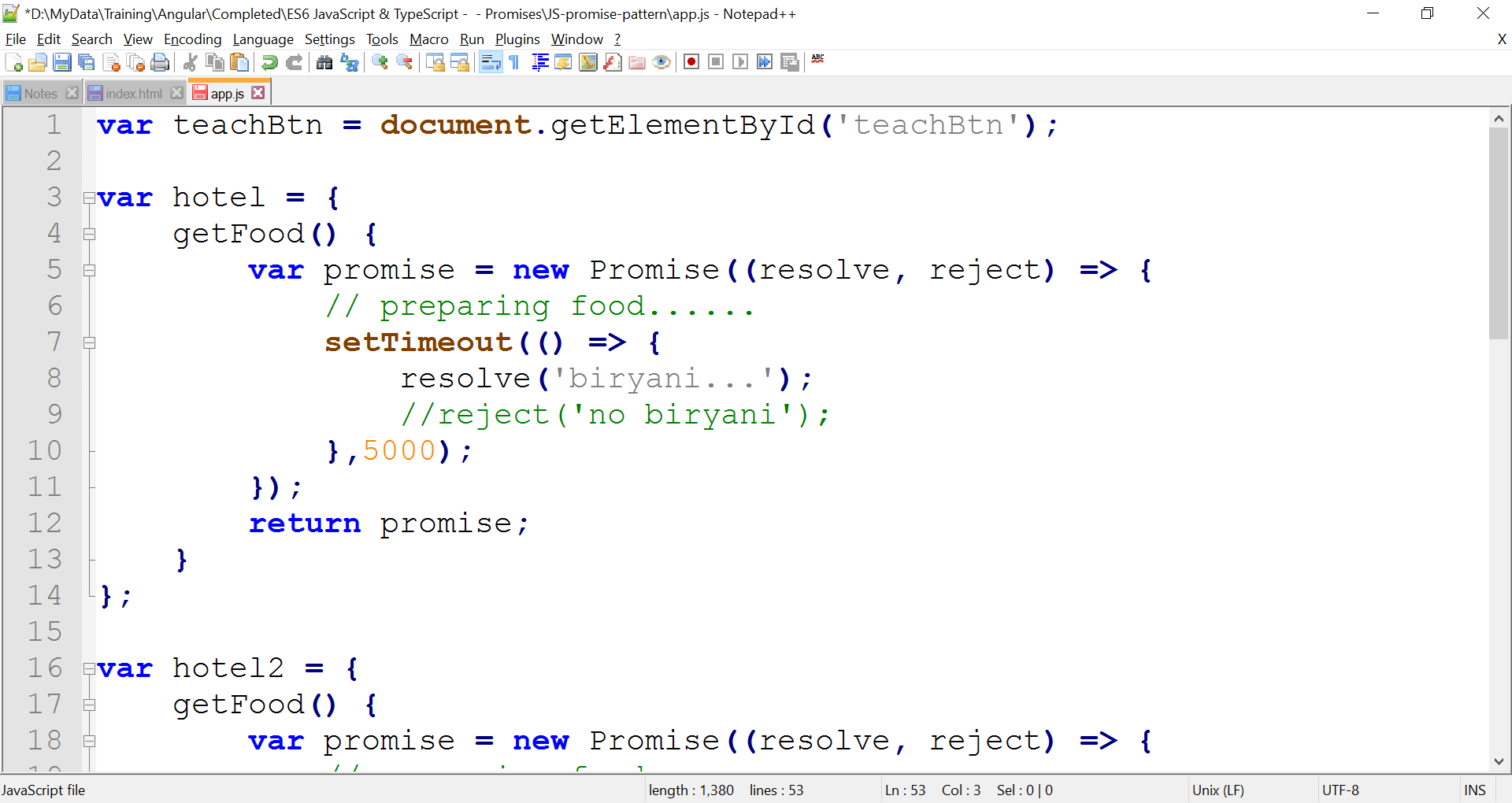
Promise Pattern

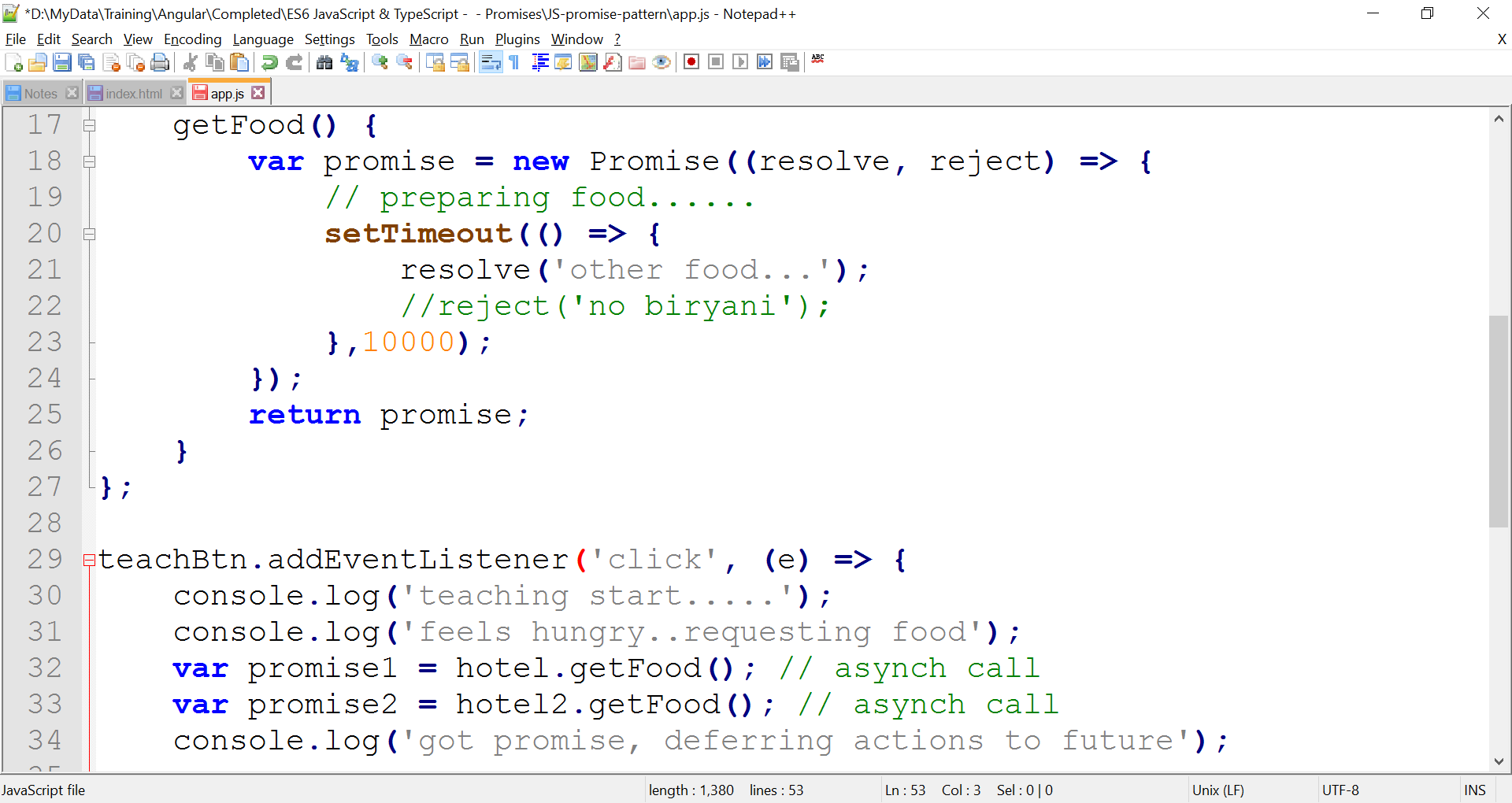
-----------------

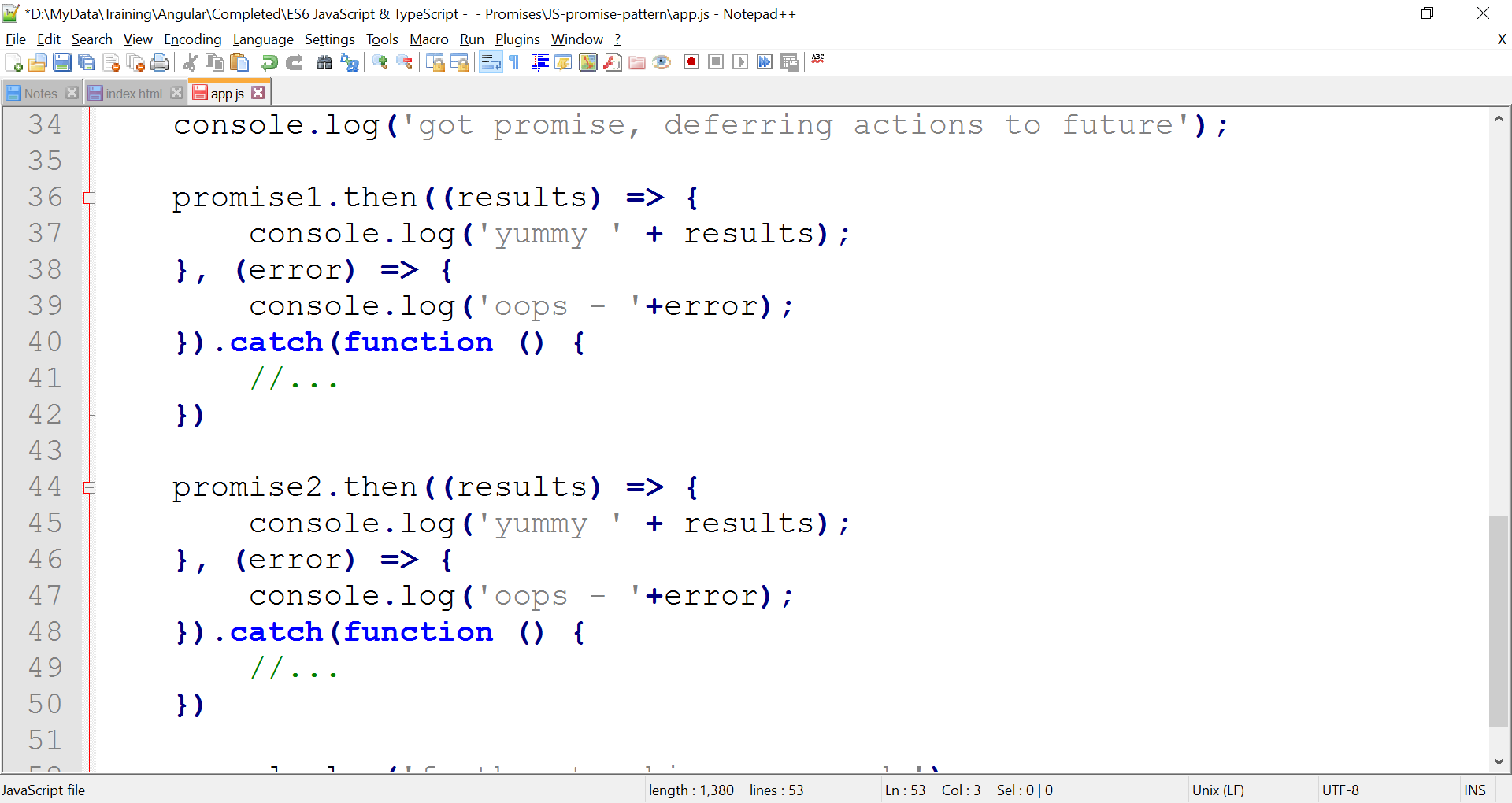
helps you run functions asynchronously,

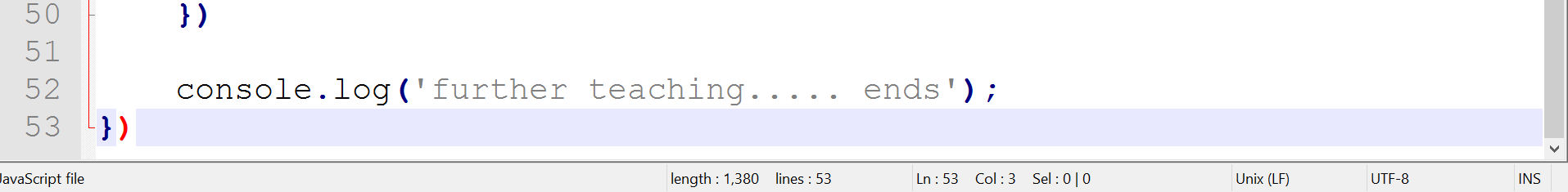
and use their return values (or exceptions) when they are done processing.











RESULT

Yummy Biryani appears after 5 secs

Yummy Otherfood appears after 10 secs

