JavaScript for App Development

By Mark Lassoff, Founder Framework Television

ES6 Promises

After completing this section, you will be able to:

- ☐ Understand asynchronicity
- ☐ Understand the form of a JavaScript Promise
- ☐ Apply a Promise to an asynchcronous event
- ☐ Work with the successful or unsuccessful resolution of an event within a Promise

Introduction

There are many situations in coding where events take place asynchronously. While JavaScript code is executed line-by-line from top to bottom, procedures like getting data from an external server or loading external media can vary in the amount of time they take. In almost all cases, the next line of code will be executed before the procedure is complete.

That's where promises come in.

Promises react to events that take an unpredictable amount of time. Promises can also respond to an error condition in those events allowing the program to handle unexpected problems gracefully.

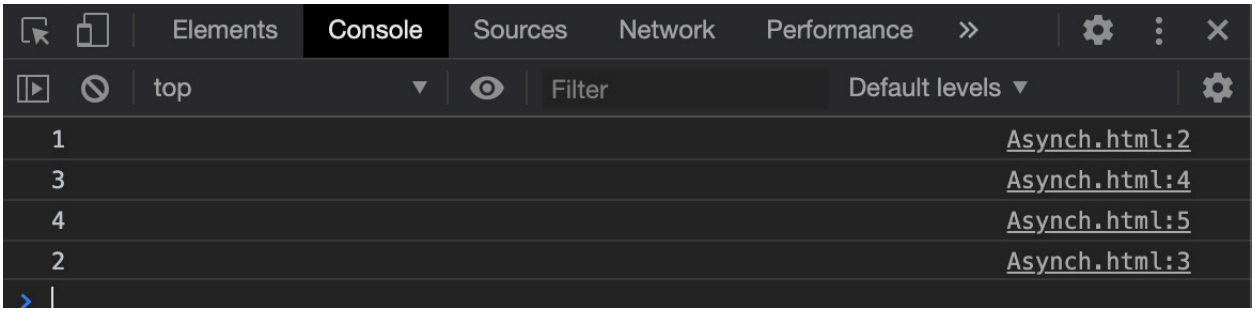
An Asynchronous Example

Consider the following bit of code:

```
<script>
  console.log("1");
  setTimeout(function(){console.log("2");}, 3000);
  console.log("3");
  setTimeout(function(){console.log("4");},1000);
</script>
```

setTimeout() creates a delay before a function is run. The first argument of the setTimeout() function is the function to be run. The second is the amount of time to wait before executing the function expressed in milliseconds.

In which order do you think the numbers will appear in the console?



In this case, the `setTimeout()` method allows the statements to run asynchronously. However, as a JavaScript coder, it is our responsibility to respond to these asynchronous events, and they are less predictable than in the previous example.

Generic Promise

Let's look at a promise in a generic form.

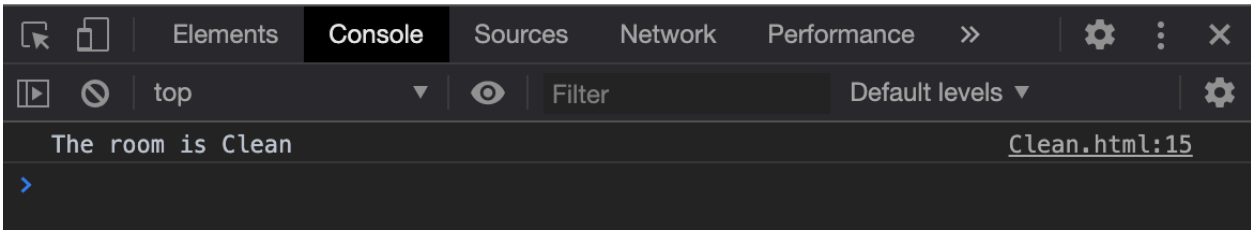
```
<script>
   let promiseToClean = new Promise(function(resolve, reject){
   //cleaning
   let isClean = true;

   if(isClean){
      resolve("Clean");
   } else
   {
      reject("Not Clean");
   }
   });
</script>
```

So in this generic promise, the asynchronous event is cleaning. We don't know how long it will take to clean and we've represented that procedure by the comment `//cleaning`. Once cleaning is complete, we're setting the value of the `isClean` variable to `true`. We then test the `isClean` variable with an if statement and run the `resolve` or `reject`method based on its value.

We can chain additional statements on to the Promise the `then()` method.

```
promiseToClean.then(function(fromResolve){
   console.log("The room is " + fromResolve);
}).catch(function(fromReject){
   console.log(froReject);
});
```

When running the above code the result is as follows:



Since `isClean` is evaluated as true, the `then()` method is run. The string from the `resolve()` call in the initial code block is passed to the function which serves as the argument of the `then()` method.

If the cleaning had not resolved successfully the `catch()` method would have been called, and the value would have been passed by the `reject()` method in the code.

Cats an a Real-World Promise Example

Now, let's take a look at a Promise implemented in some real-world code.

```
<div id="image-holder"></div>
<script>
  var img= null;
  let imgPromise = new Promise(function(resolve, reject){
    img = new Image();
    img.addEventListener('load', resolve(img));
    img.addEventListener('error', reject('Could not load image'));
    img.src = "http://thecatapi.com/api/images/
get?format=src&type=jpg&size=small";
});
```

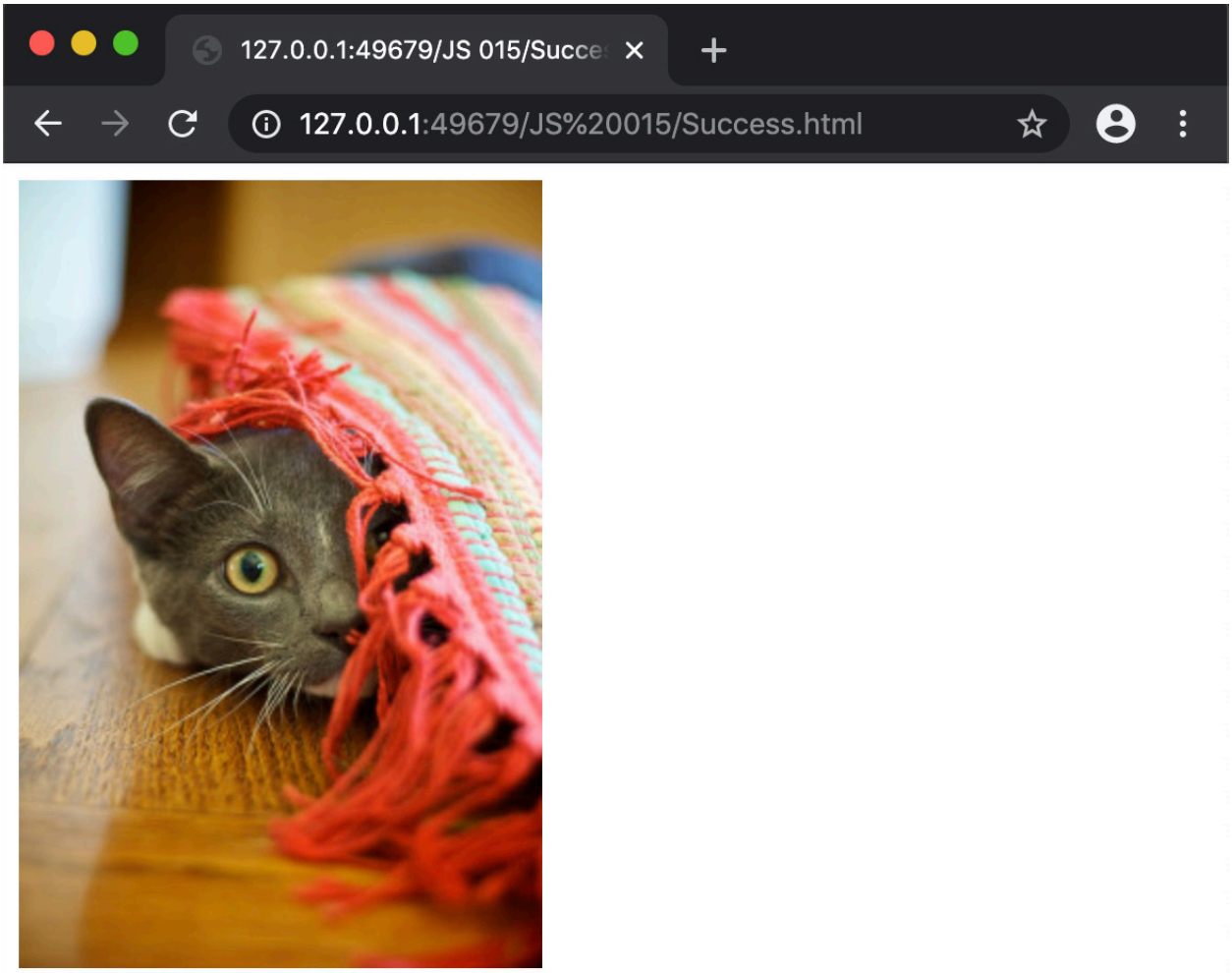
```
    imgPromise.then(function(fromResolve){
        console.log('The image has loaded. Yay!');
        let node = document.getElementById('image-holder');
        node.appendChild(img);
        }).catch(function(fromReject){
            console.log(fromReject);
        })

</script>
```

This code loads pictures of cats from the cat API. There are a few methods here you may not have seen before, but it should be reasonably easy to pick up their meaning through context.

In this case, we don't know how long it's going to take the API to serve up the image and for our client browser to download it. A promise is an appropriate way to deal with this asynchronous event.

Inside the promise, we instantiate an `Image()` object with the name `img`. To this object, we attach two listeners. The first listens for the `load` event. A successful load will resolve the promise. We also listen for an `error` event which will resolve the promise with a rejection.

Once the asynchronous download (or error event) occurs, we pass control to the second block of code.



Inside the `then()` block, our function outputs a message to the console and then displays the image. To display the image, the `Image()` object is simply appended to the HTML logical division identified as `image-holder`.

The `catch()` block deals with any error condition and, in this case, would output whatever string passed by `reject()`.

Submit This: Final Project

Your Imaginary Friends

The “Lorem Picsum” API sends random images, similar to the cat API above. It does have a few more features. You can get a random picture from the API using the following URL: https://picsum.photos/200/300/?random.

In the URL 200 represents the width of the image requested and 300 the height.

For your final project, implement this API using a promise, however, take the width and height from data entered by the user in a form. Your interface should have a field for the user to enter a width, a field to enter the height and a button that retrieves and displays a photo of the indicated size.

Once you have this working view the API instructions at https://picsum.photos/ and add to your form a checkbox that will request a grayscale image when checked.

Once you are able to request grayscale or color images of a specific size and display them, turn in your code according to the directions below.

Congratulations on completing your certification.

Please save your file in the following format to ensure proper credit:

`LastName_ES6_Promises.html.`