JavaScript(https://www.sitepoint.com/javascript/) - December 23, 2013 - By Sandeep Panda (https://www.sitepoint.com/author/spanda/)

An Overview of JavaScript Promises

Well, this has come like a Christmas gift to all JavaScript developers. You will be glad to know that promises are now a part of standard JavaScript. Chrome 32 beta has already implemented the basic promise API. The concept of promises is not new to web development. Many of us have already used promises in the form of several JS libraries such as Q, when, RSVP.js, etc. Even jQuery has something called a Deferred object (https://api.jquery.com/category/deferred-object/) which is similar to a promise. But having native support for promises in JavaScript is really amazing. This tutorial will cover the basics of promises and show how you can leverage them in your JS development.

Note: This is still an experimental feature. Only Chrome 32 beta and the latest Firefox nightly currently support it.

Overview

A `Promise` object represents a value that may not be available yet, but will be resolved at some point in the future. It allows you to write asynchronous code in a more synchronous fashion. For example, if you use the promise API to make an asynchronous call to a remote web service you will create a `Promise` object which represents the data that will be returned by the web service in future. The caveat being that the actual data is not available yet. It will become available when the request completes and a response comes back from the web service. In the meantime the `Promise` object acts like a proxy to the actual data. Furthermore, you can attach callbacks to the `Promise` object which will be called once the actual data is available.

The API

To get started, let's examine the following code which creates a new `Promise` object.

```
if (window.Promise) { // Check if the browser supports Promises
  var promise = new Promise(function(resolve, reject) {
    //asynchronous code goes here
  });
}
```

We start by instantiating a new `Promise` object and passing it a callback function. The callback takes two arguments, `resolve` and `reject`, which are both functions. All your asynchronous code goes inside that callback. If everything is successful, the promise is fulfilled by calling `resolve()`. In case of an error, `reject()` is called with an `Error` object. This indicates that the promise is rejected.

Now let's build something simple which shows how promises are used. The following code makes an asynchronous request to a web service that returns a random joke in JSON format. Let's examine how promises are used here.

```
if (window.Promise) {
  console.log('Promise found');

  var promise = new Promise(function(resolve, reject) {
    var request = new XMLHttpRequest();

    request.open('GET', 'http://api.icndb.com/jokes/random');
    request.onload = function() {
      if (request.status == 200) {
        resolve(request.response); // we got data here, so resolve the Promise
      } else {
        reject(Error(request.statusText)); // status is not 200 OK, so reject
      }
    };

    request.onerror = function() {
      reject(Error('Error fetching data.')); // error occurred, reject the  Promise
    };

    request.send(); //send the request
  });

  console.log('Asynchronous request made.');

  promise.then(function(data) {
    console.log('Got data! Promise fulfilled.');
    document.getElementsByTagName('body')[0].textContent = JSON.parse(data).value.joke;
  }, function(error) {
    console.log('Promise rejected.');
    console.log(error.message);
  });
} else {
  console.log('Promise not available');
}
```


In the previous code, the `Promise` constructor callback contains the asynchronous code used to get data the from remote service. Here, we just create an Ajax request to `http://api.icndb.com/jokes/random` (http://api.icndb.com/jokes/random) which returns a random joke. When a JSON response is received from the remote server, it is passed to `resolve()`. In case of any error, `reject()` is called with an `Error` object.

When we instantiate a `Promise` object we get a proxy to the data that will be available in future. In our case we are expecting some data to be returned from the remote service at some point in future. So, how do we know when the data becomes available? This is where the `Promise.then()` function is used. This function takes two arguments: a success callback and a failure callback. These callbacks are called when the `Promise` is settled (i.e. either fulfilled or rejected). If the promise was fulfilled, the success callback will be fired with the actual data you passed to `resolve()`. If the promise was rejected, the failure callback will be called. Whatever you passed to `reject()` will be passed as an argument to this callback.

Try this Plunkr (http://plnkr.co/edit/ilf9xtDqrimWxZd77yLI?p=preview) example. Simply refresh the page to view a new random joke. Also, open up your browser console so that you can see the order in which the different parts of the code are executed. Note that a promise can have three states:

- pending (not fulfilled or rejected)
- fulfilled
- rejected

The `Promise.status` property, which is code-inaccessible and private, gives information about these states. Once a promise is rejected or fulfilled, this status gets permanently associated with it. This means a promise can succeed or fail only once. If the promise has already been fulfilled and later you attach a `then()` to it with two callbacks, the success callback will be correctly called. So, in the world of promises, we are not interested in knowing when the promise is settled. We are only concerned with the final outcome of the promise.

Chaining Promises

It is sometimes desirable to chain promises together. For instance, you might have multiple asynchronous operations to be performed. When one operation gives you data, you will start doing some other operation on that piece of data and so on. Promises can be chained together as demonstrated in the following example.


```
function getPromise(url) {
  // return a Promise here
  // send an async request to the url as a part of promise
  // after getting the result, resolve the promise with it
}

var promise = getPromise('some url here');

promise.then(function(result) {
  //we have our result here
  return getPromise(result); //return a promise here again
}).then(function(result) {
  //handle the final result
});
```

The tricky part is that when you return a simple value inside `then()`, the next `then()` is called with that return value. But if you return a promise inside `then()`, the next `then()` waits on it and gets called when that promise is settled.

Handling Errors

You already know the `then()` function takes two callbacks as arguments. The second one will be called if the promise was rejected. But, we also have a `catch()` function which can be used to handle promise rejection. Have a look at the following code:

```
promise.then(function(result) {
  console.log('Got data!', result);
}).catch(function(error) {
  console.log('Error occurred!', error);
});
```

This is equivalent to:

```
promise.then(function(result) {
  console.log('Got data!', result);
}).then(undefined, function(error) {
  console.log('Error occurred!', error);
});
```

Note that if the promise was rejected and `then()` does not have a failure callback, the control will move forward to the next `then()` with a failure callback or the next `catch()`. Apart from explicit promise rejection, `catch()` is also called when any exception is thrown from the `Promise` constructor callback. So, you can also use `catch()` for logging purposes. Note that we could use `try...catch` to handle errors, but that is not necessary with promises as any asynchronous or synchronous error is always caught by `catch()`.

Conclusion

This was just a brief introduction to JavaScript's new Promises API. Clearly it lets us write asynchronous code very easily. We can proceed as usual without knowing what value is going to be returned from the asynchronous code in the future. There is more to the API, which has not been covered here. To learn more about Promises, browse the following resources, and stay tuned to SitePoint!

- HTML5Rocks (http://www.html5rocks.com/en/tutorials/es6/promises/)
- Mozilla Developer Network (https://developer.mozilla.org/en-US/docs/Mozilla/JavaScript_code_modules/Promise.jsm/Promise)

Meet the author

Sandeep Panda (https://www.sitepoint.com/author/spanda/) (https://twitter.com/Sandeepg33k) (https://plus.google.com/+SandeepPanda) (https://www.facebook.com/sandeep.panda92) (http://www.linkedin.com/pub/sandeep-panda/45/768/b23)

Sandeep is the Co-Founder of Hashnode (https://hashnode.com). He loves startups and web technologies.