devcomp offers **two online courses (http://developer-competence.com/course/index.php?categoryid=2)** based on our book **Building Front-End Web Apps with Plain JS (http://web-engineering.info/JsFrontendApp-Book)**. Learn how to build fully functional apps using HTML, CSS and plain JavaScript. Take interactive quizzes and work on guided practice projects with personal feedback. Ask your coach whenever you have a question.

web-engineering.info

High-quality resources for learning developers and makers

Learn JavaScript/ES6 Promises by Examples

Tweet Like 5 Share G+1 3 Share

Submitted by **mdiaconescu** on Mon, 02/15/2016 - 15:34

Short History

In concurrent programming languages, at least three terms have been used for the same promising synchronization feature: *future*, *promise*, and *delay*. These terms refer to an object that acts as a proxy for a result that will be available only some time later because its computation takes place asychronously. In this tutorial, we show how to use ECMAScript 6 (ES6) *promises* and how they can help with asynchronous computations.

The term *promise* was coined by Daniel P. Friedman and David Wise in 1976, while Peter Hibbard called it *eventual*. The first *future* and/or *promise* constructs were implemented in programming languages such as *MultiLisp* and *Act 1*. Later, promises became popular and some high level programming languages, such as for example C++ and Java, started to provide built-in support for them. JavaScript promises started out in the DOM as *Futures*, before they were renamed to *Promises*, and finally moved into JavaScript, as part of the ECMAScript 6 (ES6) specification. Having them in JavaScript rather than in the DOM is great because this makes them available in other JavaScript contexts, like in NodeJS.

What is a Promise?

A Promise represents an operation that has not completed yet, but is expected in the future. It is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers to an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of the final value, the asynchronous method returns a promise of having a value at some point in the future.

A `Promise` is created by using the `Promise( function (resolve, reject) {...})` constructor. It takes one argument, a function with two parameters, `resolve` and `reject`. Notice that `resolve` and `reject` are also functions. The generic structure of a promise is shown in the example below:

```
var promise = new Promise( function (resolve, reject) {
  // some code (synchronous or asynchronous)
  // ...

  if (/* the expected result/behavior is obtained */) {
    resolve( params);
    // OR
    return resolve( params);
  } else {
    reject( err);
    // OR
    return reject( err);
  }
});
```

Then, we can use the `promise` by invoking its `then` method (from here the *thenable* terminology):

```
promise.then( function (params) {
  //the promise was fulfilled
  console.log(params);
}).catch( function (err) {
  // the promise was rejected
  console.log( err);
});
```

The `then` method takes as parameter a method, invoked when the promise *fulfilled*, i.e., the `resolve` method was invoked. The `catch` method takes as parameter a method, invoked when the promise is *rejected*, i.e., the `reject` method was invoked. For both cases, the callback parameters are the ones used to call the `resolve`, respectively the `reject` methods. Alternatively, is possible to use only the `then` method for both, fulfilled and rejected cases:

```
promise.then( function (params) {
  //the promise was fulfilled
  console.log( params);
},function (err) {
  // the promise was rejected
  console.log( err);
});
```

While `catch` is almost the equivalent of `then( undefined, catchCallback)`, using both, `then` and `catch`, makes the code easier to read and understand.

Promises can also be chained, so we can write code similar to:

```
promise.then( function (params) {
  //the promise was fulfilled
  console.log(params);
}).catch( function (err) {
  // the promise was rejected
  console.log(err);
}).then(function (params) {
  // do something more...
});
```

Notice that we can chain `then` with `then` (i.e., `.then(function (p){...}).then(function (p){...})`) or `catch` with `then` (i.e., `.then(function (p){...}).catch(function (p){...}).then(function (p){...})`) as many times as needed.

States and Fates

Promises may be in one of the following three states:

- **fulfilled:** if `promise.then(f)` will call `f` at some point in time;
- **rejected:** if `promise.catch(e)` will call `e` at some point in time;
- **pending:** if it is neither fulfilled nor rejected.

Another term, i.e., *settled*, is used for something that's not really a promise state, but it expresses the fact that a *promise* has either *fulfilled* or *rejected*.

Note: a promise can be, at any point in time, only in one of the three above described states.

Promises have two possible mutually exclusive fates:

- **resolved:** the promise either follow another promise, or has been fulfilled or rejected;
- **unresolved:** if trying to resolve or reject it will have an impact on the promise.

A promise can be *resolved* to either a promise or *thenable*, in which case it will store the *promise* or *thenable* for later unwrapping, or it can be resolved to a non-promise value, in which case it is fulfilled with that value.

Example 1: Asynchronous JavaScript File Loading

Before ECMAScript6, the JavaScript programmers used to implement promises in ad-hock manners, with the help of callback methods and/or using `setTimeout/setInterval` calls combined with flags and conditional checks.

One common example of asynchronous operation was dynamic JavaScript scripts loading. This is very useful to obtain a modular behavior for our JavaScript applications, and also to avoid the load of unnecessarily JavaScript code if not needed (i.e., save some bandwidth in case of mobile devices).

No-Promise JavaScript implementation

To dynamically load a JavaScript file, we had a code similar with the following one:

```
function loadJsFile( filePath, onLoadFinishedCallback, onLoadErrorCallback) {
  var scriptElem = document.createElement('script');
  scriptElem.onload = onLoadFinishedCallback;
  scriptElem.onerror = onLoadErrorCallback;
  // trigger the file loading
  scriptElem.src = filePath;
  // append element to head
  document.head.appendChild( scriptElem );
};
```

Then, when needed, we may call the `loadJsFile` method:

```
loadJsFile("js/script.js",
  function () {
    // At this point we are sure that the file loaded was completed successfully.
    // Now is possible to invoke methods defined by the newly loaded JS file.
  },
  function (err) {
   // alert the user about the failure
   alert("Failed to load JS file!);
   // write to JS console the details about the error
   console.log(err);
  }
);
```

Run the example (http://web-engineering.info/tech/promise/loadJsFileNoPromise/) **Download the code (http://web-engineering.info/tech/promise/loadJsFileNoPromise.zip)**

Note: the example code works in most of the modern web browsers. Some old web browsers may not be able to execute it. Check the JavaScript console for details in case of errors. It also works locally, by using the *file* protocol (e.g., `file://path/to/app/index.html`).

ES6 promise implementation

Lets see now how we obtain the same behavior, this time with the help of *ECMAScript6 Promises*. First we re-implement the `loadJsFile` method as follows:

```
function loadJsFile( filePath) {
  return new Promise( function (resolve, reject) {
    var scriptElem = document.createElement('script');
    scriptElem.onload = resolve;
    scriptElem.onerror = reject;
    // trigger the file loading
    scriptElem.src = filePath;
    // append element to head
    document.head.appendChild(scriptElem );
  });
};
```

When the file loaded successfully, the promise `resolve` is called. The same, if the file loading fails, e.g., the file does not exists, the promise `reject` is called. The difference is that our `loadJsFile` has now only one parameter, i.e., the `filePath` instead of three, which is what we normally expect from a "file loading" method.

Later, somewhere in our JavaScript code, we like to invoke the `loadJsFile` method:

```
loadJsFile("js/myScript.js").then(
  function () {
    // At this point we are sure that the file loaded was completed successfully.
    // Now is possible to invoke methods defined by the newly loaded JS file.
  })
  .catch(function (err) {
    // alert the user about the failure
    alert("Failed to load JS file!");
    // write the error details to JS console
    console.log(err);
  }
);
```

Instead of using *success* and *error* callback methods as parameters of the `loadJsFile` method, we use `then` and `catch`. If the promise is *fulfilled*, the method given as parameter to `then` method is executed. If the promise is *rejected*, the method given as parameter to `catch` method is invoked, expressing the failure/reject state.

Note: the example code works in most of the modern web browsers. Some old web browsers may not be able to execute it. Check the JavaScript console for details in case of errors. It also works locally, by using the *file* protocol (e.g., `file://path/to/app/index.html`).

Run the example (http://web-engineering.info/tech/promise/loadJsFileWithPromise/) **Download the code (http://web-engineering.info/tech/promise/loadJsFileWithPromise.zip)**

Example 2: XHR (XML HTTP Request)

XHR/AJAX requests are another important use case of JavaScript asynchronous code. Normally we use such code to retrieve/send data from/to a server, as for example in the case of dynamic auto-completion form fields, or when we need to retrieve data which is stored into a server database. More complex applications, such as web file managers, use XHR/AJAX to inform a back-end web application that a file (or folder) needs to be created/deleted, or that we need the index of folder.

No-Promise JavaScript implementation

We start by creating a minimalistic NodeJS based HTTP server which is able to reply to GET requests with a simple message in the format `{timestamp: timestamp_value, message: "some message..."}`. Fortunatelly, we don't have to install any additional module, but you'll have **download and install NodeJS**, in case you don't have it already.

The NodeJS server code is very simple:

```
var port = 3000,
     http = require('http'),
     httpServer = null;

// create HTTP server
httpServer = http.createServer(function (req, resp){
  resp.end(JSON.stringify(
    {timestamp: (new Date()).getTime(),
     message: "Hi, I am the NodeJS HTTP server!"}));
});

// start HTTP server
httpServer.listen(port, function (){
  //Callback triggered when server is successfully listening. Hurray!
  console.log("HTTP Server started on port " + port);
});
```

Create a new file named `server.js` somewhere on your disk (or to a folder of an online server if you have one), and copy/paste the above example code. To start the HTTP server, use a command line tool, navigate to the folder where you have created the `server.js` file, and execute `node server.js`. To try if your HTTP server runs, open your favorite web browser and navigate to `http://domain:3000`. Remember to replace the `domain` with `localhost`, if you run the server on your local machine, or with the real web domain name, if you use an online server. Notice that 3000 is the port number we have been used in our example code, so you may need to change it with something else, depending on your server configuration (e.g., the port is not open in the server firewall, or some other application uses port 3000).

It is now the time to write a method that uses XHR (XML HTTP Request),to make a GET request to our HTTP server:

```
function makeGetRequest(url, successCallback, errCallback) {
  var req = new XMLHttpRequest();
  req.open('GET', url);
  req.onload = function () {
    if (req.status == 200) successCallback(JSON.parse(req.response));
    else errCallback({code: req.status, message: req.statusText});
  };
  // make the request
  req.send();
};
```

For simplicity reasons we create a lightweight version of a GET request implementation, which does not allow request parameters and does not treat more specific error cases, such as the ones specific to network problems, normally being captured with the help of `onerror` (i.e., `req.onerror = function (err) {...}`)).

Calling the `makeGetRequest` method somewhere in our JavaScript code can be done as in the following example:

```
makeGetRequest("/",
  function (resp) {
    // do something with the response (resp parameter)
  },
  function (err) {
    alert("GET request failed!")
    console.log(err);
    }
  );
```

Remember to change the request URL with whatever you need to, specially if the NodeJS HTTP server runs code behind (or together with, e.g., using a proxy) Apache or Nginx. If you are behind a corporate firewall, it is very likely that any other port excepting 80 and maybe 443 are closed. In this case, you may have to use Apache to create a proxy for your HTTP server application. This can be done by editing your `httpd.conf` file, and adding the following code as part of the virtual host of the used domain:

```
ProxyPass /example/promise http://localhost:3000/
ProxyPassReverse /example/promise http://localhost:3000/


  Order Allow,Deny
  Allow from all
```

Remember to restart the `httpd` service (e.g., by using `service httpd restart`) after saving the above modifications. Also, remember to call the `makeGetRequest` with the new URL as parameter, i.e., `makeGetRequest("example/promise").then(...)`.

Note: the example client code works in most of the modern web browsers. Some old web browsers may not be able to execute it. Check the JavaScript console for details in case of errors. It does NOT execute locally, i.e., by using the *file* protocol!

Run the example (http://web-engineering.info/tech/promise/ajaxCallNoPromise/) **Download the code (http://web-engineering.info/tech/promise/ajaxCallNoPromise.zip)**

ES6 promise implementation

Our NodeJS code remains unchanged, but the client `makeGetRequest` method needs to be re-implemented by using a `Promise` this time:

```
function makeGetRequest(url) {
  return new Promise(function (resolve, reject) {
    var req = new XMLHttpRequest();
    req.open('GET', url);
    req.onload = function () {
      if (req.status == 200) resolve(JSON.parse(req.response));
      else reject({code: req.status, message: req.statusText});
    };
    // make the request
    req.send();
  });
};
```

Mainly, we check if the response code is 200 (HTTP code 200 means all went fine, "OK"). In case that we get that code we parse the response by using JSON.parse, since we know that we expect a string serialization of a JSON object from our server. Any other HTTP code than 200 in our example represents an error, so the promise `reject` is called.

In our HTML code, at some point we need to make to invoke the `makeGetRequest` method:

```
makeGetRequest("/example/promise")
.then(function (resp) {
  // do something with the response (resp parameter)
})
.catch(function (err) {
  alert("AJAX GET request failed!")
  console.log(err);
});
```

Note: the example code works in most of the modern web browsers. Some old web browsers may not be able to execute it. Check the JavaScript console for details in case of errors. It does NOT execute locally, i.e., by using the *file* protocol!

Run the example (http://web-engineering.info/tech/promise/ajaxCallWithPromise/) **Download the code (http://web-engineering.info/tech/promise/ajaxCallWithPromise.zip)**

Example 3: Login Action with NodeJS and MongoDB Database

Latest NodeJS releases provide native support for promises. We've tested the example code fragments shown below with NodeJS v5.4.1. This is awesome, because we can use them with a JavaScript engine which does not run only in a web browser. A relevant use case for asynchronous code in a NodeJS server application is provided by database operations, such as connect and query.

In this section we show how to use promises in combination with a MongoDB database. We'll try to connect to a MongoDB database, then check the credentials (username and password) for a login request. Such scenarios are often seen in applications that provide a user authentication mechanism. For this scenario we'll use the **mongodb** NodeJS module, which can be installed by executing the `npm install mongodb` command, in the root folder of the application. Remember that you need to install **MongoDB Database Engine** before being able to use MongoDB with any application.

No-Promise JavaScript implementation

Our database contains a collection named *users* which stores the registered users for our app. Due to JavaScript's non-blocking nature, however, we can't just use `find({username: val1, password: val2})`. First, we should be sure that we have connected successfully to the database, so we need to use `mongodb.connect` and pass it a callback function where, depending on the success or error case, we look for the user or we have to deal with a connection error.

```
var MongoClient = require('mongodb').MongoClient;

function login(username, password, successCallback, errCallback) {
  MongoClient.connect("mongodb://localhost:27017/myapp", function (err, db) {
    var usersCollection = null, stream = null;
    if (err) errCallback(err);
    else {
      usersCollection = db.collection('users');
      stream = usersCollection.findOne(
        {username: username, password: password},
        function (err, user) {
          if (err) errCallback(err);
          else if (!user) errCallback();
          else successCallback(user);
        }
      );
    }
  });
};
```

Our method takes 4 parameters: the `username`, associated `password`, a `success` callback and an `error` callback. While this code does not look bad, the things can be worst in complex scenarios. Imagine for example that we have to connect to database, then check authentication, then make a modification (update) and then update a log collection, and all these operations have callbacks. In such a case, a "doom pyramid" code is obtained, which is really ugly and hard to read.

Later on, the login method can be invoked to check the credentials of a specific user:

```
login("johnny", "12CqqDt4e*",
  function (user) {
    console.log("Login successful");
    // now provide user access to your app...
  },
  function (err) {
    console.log("Login failed.");
    // user should be informed that the login failed...
  }
);
```

Create a file named `mongotest.js` and copy/paste the above two pieces of code to it. Using a command line tool (e.g., `cmd` for Windows or `terminal` for Linux) navigate to the folder where this file is located and install the `mongodb` NodeJS module by executing `npm install mongodb`.

To run the example, execute `node mongotest.js`. Before being able to have a succesful login, you need to create the Mongo collection and insert the user(s). For this, using your command tool, type `mongo` and press enter. In the interactive Mongo client mode, type the following code:

```
use myapp;
db.createCollection("users");
db.users.insert({username:"johnny", password:"12CqqDt4e"});
```

Download the code (http://web-engineering.info/tech/promise/nodeMongoDbNoPromise.zip)

ES6 promise implementation

Our `login` method is now updated to use ECMAScript6 Promises:

```
var MongoClient = require('mongodb').MongoClient;

function login(username, password) {
  return new Promise(function (resolve, reject) {
    MongoClient.connect("mongodb://localhost:27017/myapp", function (err, db) {
      var usersCollection = null, stream = null;
      if (err) reject(err);
      else {
        usersCollection = db.collection('users');
        stream = usersCollection.findOne(
          {username: username, password: password},
          function (err, user) {
            if (err) reject(err);
            else if (!user) reject();
            else resolve(user);
          }
        );
      }
    });
  });
};
```

Practically, we replaced all the `successCallback` and `errCallback` calls with `resolve` and respctively `reject`. The two callback methods used as parameters before are no longer needed. Using our `login` method is straight forward:

```
login("johnny", "12345")
.then(function (user) {
  console.log("Login successful");
  // now provide user access to the site...
})
.catch(function (err) {
  console.log("Login failed.");
  // user should be informed that the login failed...
});
```

Download the code (http://web-engineering.info/tech/promise/nodeMongoDbWithPromise.zip)

Interesting Facts About Promises

Promises does not only make you code to look better and to be more readable, but it also provide additional advantages, such as a semantic handling of errors by using `catch` instead of our old `errorCallback` buddy. There are a few more important facts to consider when using promises, which are discussed in this section.

Do promises swallow your errors?

Note: this fact is inspired by a **blog post (http://jamesknelson.com/are-es6-promises-swallowing-your-errors/)** of James K. Nelson. You may also want to check this **youtube video** related to this matter.

Considering the following promises:

```
Promise.resolve('promised value').then(function () {
  throw new Error('error');
});

Promise.reject('error value').catch(function () {
  throw new Error('error');
});

new Promise(function (resolve, reject) {
  throw new Error('error');
});
```

You may think: "what? clearly, all three of them throw the error!". Well, that's not really the case, and actually you won't get the error in any of the cases. Solving this issue is possible by using `catch` after each of these promises, such as:

```
Promise.resolve('promised value').then(function () {
  throw new Error('error');
}).catch(function (error) {
  console.error(error.stack);
});
```

However, if this does not looks like your style to work with promises, then try the **bluebirdjs (http://bluebirdjs.com/)** library, which allows you to solve the problem as shown below:

```
// handle any discharged errors
Promise.onPossiblyUnhandledRejection(function (error){
  throw error;
});

// in case you like to discharge errors
Promise.reject('error value').catch(function () {});
```

Promises can be looped

Supposing that you have more than one promise which should be handled in the same way, then you can use `Promise.all` to loop them and execute your code:

```
var p1 = Promise.resolve("I am a solved promise!"),
    p2 = 2.7889,
    p3 = new Promise( function (resolve, reject) {
         setTimeout(resolve, 1000, "My name is 'The Promise'!");
    });

Promise.all( [p1, p2, p3]).then( function (values) {
  console.log(values);
});
```

The result of executing the code is the array: `["I am a solved promise!", 2.7889, "My name is 'The Promise'!"]`.

Equivalent promises

For specific cases, we can use `Promise.resolve`, or we can create a new Promise, or we can use the `then` method, or ... is all getting confusing. In fact, there are many ways to obtain the same promise behavior. For example, all the following promises are functionally equivalent:

```
Promise.resolve(v);
new Promise(function (resolve, reject) {resolve(v);});
Promise.resolve().then( function () {return v; });
Promise.all([v]).then( function (val){ return val[0]; });
```

The choices `catch()` and `then(undefined,function(v){})` are not "exactly" the same

In the first part of the article, we discussed that using `then` and `catch` is "almost" equivalent with using `then` with two parameters. Lets see why "almost", and for this we consider the following "equivalent" promises:

```
somePromise()
.then(function () {  // do something})
.catch(function (err) {  // handle error});

somePromise().then(function () {
  // do something
}, function (err) {
  // handle error
});
```

At first, they look to give the same results, and with an exception, they actually do so. The difference is made in the case when an exception/error is thrown in the `then` handled function:

```
somePromise()
.then(function () {
  throw new Error('Bad, things are not good here!');
}).catch(function (err) {
  // here the thrown error is now catched!
});

somePromise().then(function () {
  throw new Error('Bad, things are not good here!');
}, function (err) {
  // ups, the error is not cached, and this function is not called!
});
```

When using `then`, to handle both, the *resolve* and the *reject* cases, an error (exception) thrown in the resolve case is not passed to the reject handler.

Support for ES6 Promises

Web Browsers: The current versions of all major web browsers (Mozilla Firefox, Google Chrome, Microsoft Edge, Opera and Safari) support the new `promise` method. For a complete list of supportive browsers, check **caniuse.com (http://caniuse.com/#feat=promises)** .

NodeJS: provides builtin support for promises, and in addition, a set of modules, such as **bluebirdjs (http://bluebirdjs.com/)** , make the use of promises even easier and appealing.

Tweet Like 5 Share G+1 3 Share

0 Comments web-engineering.info

1 Login

Recommend Share

Sort by Best

Start the discussion…

Be the first to comment.

ALSO ON **WEB-ENGINEERING.INFO**

Dealing with Enumeration Attributes

1 comment • 2 years ago•

kaffeecko — Very helpful! Mark!

Storing database tables in JavaScript's Local Storage

1 comment • 2 years ago•

Guest — Nice article!

Optimize Arduino Memory Usage

1 comment • a year ago•

wolfv — Thank you for writing a nice clear article on Arduino memory usage.I ran into a mystery with the getFreeSram() function.The following example

Build Your First WebRTC based Video-Conference App

19 comments • a year ago•

f khan — i don't see any download link on github

Subscribe Add Disqus to your site Add Disqus Add Privacy