<http://www.gistia.com/mastering-promises/>

<http://www.gistia.com/mastering-promises/>

# Mastering Promises in Angular 2

All applications have some amount of asynchronous code. Imagine if you had to wait for every application on your computer to complete a task before you could do something else.

What if you had to wait for all the posts to load on Facebook before you could see your Facebook home page.

That’s just not how the internet works. Everything is asynchronous. Google Analytics is asynchronously tracking your every move as you read this article…

## Asynchronous coding

Asynchronous (or async) sounds like a fancy word. But it’s very simple.

Blocking is the opposite of async. If you had blocking code, that would mean that every line of code has to wait for the previous line of code to finish executing.

Now let’s look at what JavaScript is –

JavaScript is a single-threaded, non-blocking, asynchronous, concurrent language.

Let’s take a quick detour to understand how concurrency works in JavaScript. This will help us figure out why JavaScript is a single-thread language and yet async.

## JavaScript Async Mysteries

The JavaScript runtime mainly consists of –

### 1. Heap

The heap is an area of unstructured memory. If you’re familiar with memory allocation, you probably know what the heap does.

If you’re not, don’t worry. That’s all you need to know about the heap for now.

### 2. Call Stack

The call stack is where the methods get stacked as they are called and then executed.

For example, if we had the following code –



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | var writeArticle = function() {    // do article writing stuff  }    var publishArticle = function() {    writeArticle();  }    var gistia = function() {    publishArticle();  } |

The call stack would look like –



|  |  |
| --- | --- |
| 1  2  3  4 | writeArticle // Executed first  publishArticle // Executed second  gistia // Executed last |

The call stack is what we are referring to when we talk about the single thread. This is our code that is executed by the browser.

If we put any code in the call stack that takes too much time, we block the code execution and cause the page to freeze.

Things like console logs and algebra / calculations are fine to push on the stack. But operations like timeouts and API calls would cause our application to freeze till they aren’t complete.

This obviously isn’t good for user experience.

### 3. Web APIs (or other APIs)

Many of the APIs we use when we are writing JavaScript code are made available to use through Web APIs.

The Web API handles async functionality such as setTimeout. That’s why we don’t need to worry about blocking code when we use a timeout.

A perfect place to experiment with and see how these different elements come together is [**Loupe**](http://latentflip.com/loupe/?code=JC5vbignYnV0dG9uJywgJ2NsaWNrJywgZnVuY3Rpb24gb25DbGljaygpIHsKICAgIHNldFRpbWVvdXQoZnVuY3Rpb24gdGltZXIoKSB7CiAgICAgICAgY29uc29sZS5sb2coJ1lvdSBjbGlja2VkIHRoZSBidXR0b24hJyk7ICAgIAogICAgfSwgMjAwMCk7Cn0pOwoKY29uc29sZS5sb2coIkhpISIpOwoKc2V0VGltZW91dChmdW5jdGlvbiB0aW1lb3V0KCkgewogICAgY29uc29sZS5sb2coIkNsaWNrIHRoZSBidXR0b24hIik7Cn0sIDUwMDApOwoKY29uc29sZS5sb2coIldlbGNvbWUgdG8gbG91cGUuIik7!!!PGJ1dHRvbj5DbGljayBtZSE8L2J1dHRvbj4%3D). Loupe allows you to see the runtime elements while the sample code is running.

Try setting timeouts with a 0 delay and see what happens!

### 4. Callback Queue

Let’s understand this through an example –



|  |  |
| --- | --- |
| 1  2  3  4  5 | setTimeout(() => {    console.log('5 seconds are over!');  }, 5000);    console.log('Thank god you did not block me! :)'): |

That should log the second statement first and then the first statement.

Why does that happen?

The call to setTimeout is handled by the WebAPI. The timeout doesn’t occur on the main call stack. But what happens when the 5 seconds get over. The WebAPI cannot push the log function straight to the main call stack. That would cause all kinds of problems with functions being executed in an unpredictable order.

Instead, the WebAPI queues the functions that need to run next in the callback queue. The callback queue waits for the functions on the call stack to finish executing and then pushes it’s next function into the call stack.

In this manner, if you had multiple callbacks waiting to be fired, they’d all wait in the callback queue before being sent to the call stack for execution.

Now that you understand how basic async works in JavaScript let’s see how we can use this to our advantage.



**Never Miss an Article**

**Javascript and Engineering Leadership**

Get our best content in a single weekly email,  
Join 34,302 subscribers (as of sept  25, 16).



## How we used to execute async calls

Before promises, callbacks were what we used for async functionality.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | // Goal is to get all the users orders  fetchUser({}, function(err, user) {    if (err) { // do something with error }      // No error. Get orders for this user    fetchOrders({userId: user.\_id}, function(err, orders) {    if (err) { // do something else with this error }      return orders;    });  }); |

The async functions like fetchUser would look something like –



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | // cb is the callback funciton passed as the argument  var fetchUser = function(userObj, cb) {    // Find user with userObj stuff    // ...setting data and error      // Callback after we get the user    return cb(err, user);  } |

The code above executes some code to get the user data and then calls the function passed in the cb argument.

That doesn’t look so bad, does it?

## Callback Hell

Callbacks are fine till the code doesn’t get complex.

But what happens when you have many layers of calls and many errors to handle?

You encounter [**Callback Hell!**](http://callbackhell.com/).

But don’t worry! Callback hell is nothing more than a fancy way of saying code that looks bad. Imagine looking at code that looks something like this –



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | // Code that looks really bad  Order.findById({id: orderId}, function(err, order) {    if (err) // do something with the error    if (!order) // order not found      User.findById({id: order.userId}, function(err, user) {    if (err) // do something with the error    if (!user) // user not found      Company.findById({id: user.companyId}, function(err, company) {    if (err) // do something with the error    if (!company) // company not found      // Do something with company    });    });  }); |

The code above just has three layers and we aren’t even doing any data manipulation yet. You can see how the pyramid structure starts to make the code less readable.

That’s why programmers started developing solutions like Promises. Until now promises were not natively available in JavaScript, and the solution was to either use a promises library or write an implementation of your own.

That is no longer the case! Because ES6 officially support promises.

## What is a promise?

Imagine this…

You are the King of a medieval empire. Everything was great in your Kingdom, till your favorite messenger pigeon returns with a message and a deep wound.

Give us your elixir of power or we will destroy your Kingdom  
– Genghis Khan

Your walls are strong, and your soldiers are brave, but you don’t want beef with Genghis Khan. Your last hope is your loyal friend two kingdoms away who has an army strong enough to put fear in the heart of Genghis Khan!

You call your most trusted messenger, Artemedes, and tell him to ask your friend for aid.

Artemedes **promises** you that he will return with an answer and starts running towards your friend’s kingdom.

Meanwhile, you cannot **wait** for Artemedes to come back before you make any other preparations. You have to be ready when Genghis Khan arrives.

So you give the gate protector of your kingdom the Elixir and the following instructions –

If Artemedes arrives with an army or a positive response from my loyal friend, seal the gates and prepare for battle.

But if Artemedes comes back with a negative message, give the Elixir to Genghis Khan.

You are probably wondering what happens next. But you should not be. Because as the King, you did a great job creating a **async** operation by sending Artemedes to your friend’s Kingdom.

Artemedes gave you a **promise** that he will return with an answer and your gatekeeper knows what to do in a positive or negative scenario.

Congratulations King! You just created your first promise. A promise is simply the **result** of an asynchronous operation.

## States of a promise

At any time, a promise can be in one of these three states –

### 1. Pending

A promise is pending when you start it. It hasn’t been resolved or rejected. It is in a neutral state.

When you sent Artemedes (in the example above) to your friends Kingdom, his promise to come back was pending. He just started off on his journey.

### 2. Fulfilled

When the operation is successful, a promise is considered fulfilled or resolved.

If Artemedes were to come back with an army, he would have fulfilled his promise and resolved the issue.

### 3. Rejected

If an operation fails, the promise is rejected.

If Artemedes came back and told you that your friend refused to help, the promise would have been rejected. Artemedes tried, but the external circumstances (your friend’s response) didn’t work out as planned.

## Write your first promise

Now that you have a better idea of how Promises work let’s write one together!



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | const promise = new Promise((resolve, reject) => {    // Do some async stuff      // If async opp successful    resolve();      // If async opp fails    reject();  }); |

new Promise() is the simplest way to create a promise. The promise constructor takes one function argument. That function, in turn, has two arguments – resolve (we call this if the async operation is successful) and reject (if the async operation fails).

That’s great. We can now do some async stuff without blocking the main call stack.

When the promise is no longer in a pending state then…

Exactly… then is the keyword we will use to tell the promise what to do when it succeeds / fails.



|  |  |
| --- | --- |
| 1  2  3  4  5 | const promise =    new Promise((resolve, reject) => {    // Do some promise stuff    })    .then((whenResolved, whenRejected)); |

The then method has two arguments (functions) which get called based on whether the promise is resolved or rejected. We can pass arguments to these methods through the resolve and reject methods in the Promise constructor.

Here’s what a more holistic promise looks like.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | // To reject promise change value of CONDITION to false  const CONDITION = true;    // Data passed when the promise is resolved  const DATA = 'hello world';    // Error passed when the promise is rejected  const ERROR = new Error('Ooops!');    const promise =    new Promise((resolve, reject) => {    // do some async stuff      if (CONDITION) resolve(DATA);    else reject(ERROR);    })    .then(    (data) => { console.log(data); },    (err) => { console.log(err); }    ); |

This code looks much better and cleaner. We don’t need to deal with messy callbacks and unreadable code.

There is another way we could deal with the error



|  |  |
| --- | --- |
| 1  2  3 | // Other promise stuff    .then((data) => { console.log(data); })    .then(undefined, (error) => { console.log(error); }); |

This code does the same thing as the code above. The only difference is that the data handling and error handling has been separated to make the code more readable.

ES6 adds some more syntactic sugar by allowing you to write the same code like this.



|  |  |
| --- | --- |
| 1  2  3 | // Other promise stuff    .then((data) => { console.log(data); })    .catch((error) => { console.log(error); }); |

.catch(whenRejected) is nothing but a prettier version of .then(undefined, whenResolved).

Now you have everything you need to create your promises!

Many time when you’re using an external library to do API calls, you don’t need to create the promise yourself. Many libraries like **[axios](https://github.com/mzabriskie/axios)** are promise based libraries.

This means that all you need to do is handle the data with .then(whenResolved) and handle the errors with `.then(whenRejected).’

## But All Browsers Don’t Support ES6

You don’t need to worry about that. If you’re working with Angular 2, React, or any other modern framework or library, you’ll be using a transpiler (such as babel with ES6 and tsc with Typescript).

That means that the code will be converted to a readable and usable version for older browsers that don’t support ES6.

## Other functions with Promises

### 1. Instantly resolve or reject promises

Using the Promise.resolve(data) or Promise.reject(error) methods you can resolve or reject a promise immediately.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | let firstPromise = Promise.resolve(10);  let secondPromise = Promise.reject(Error('Ooops!'));    firstPromise    .then(console.log)    .catch(console.log);    secondPromise    .then(console.log)    .catch(console.log); |

The output for the code above would be 10 and then the Error object.

### 2. Executing all promises

Using the syntax Promise.all(iterable), you can execute an array of Promises. This method resolves when all promises have resolved and fails if any of those promises fail.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | let firstPromise = Promise.resolve(10);  let secondPromise = Promise.resolve(5);  let thirdPromise = Promise.resolve(20);    Promise    .all([firstPromise, secondPromise, thirdPromise])    .then(values => {    console.log(values);    }); |

The output for the code above would be [10, 5, 20];

Promise.all() has fail fast behavior. This means that as soon as any one promise fails, Promise.all will reject immediately.

### 3. Promise race function

You can use the Promise.race(iterable) function when you want the value of the first promise that resolves or rejects.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | let promise1 = new Promise((resolve, reject) => {    setTimeout(resolve, 2000, 'promise 1 resolved');  });    let promise2 = new Promise((resolve, reject) => {    setTimeout(reject, 3000, 'promise 2 rejected');  });    Promise    .race([promise1, promise2])    .then(console.log)    .catch(console.log); |

In the example above, the .then() function would be invoked because promise1resolves before promise 2 – even though promise 2 fails. The output would be "promise one resolved";

If you want to know more about promises, don’t forget to check out the [**API docs**](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Promise)!

## Are Promises the only solution?

Of course not! Programmers never have ONLY ONE best solution. Let’s not get dogmatic.

You could write great code with less abstraction using callbacks. You need to plan out the structure of your application better and make the code more modular.

You could also achieve great results using ES6 generators. These are especially useful when you need to execute multiple async events that depend on data returned by the previous async call.

**Practical code :**

ngOnInit() {

console.log('Start')

this.getUser().then(function(data){

console.log('The casual data is ')

console.log(data)

}).catch(function(err){

console.log('The error is ')

console.log(err)

});

}

getUser(){

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

setTimeout(()=>{

resolve('Got users data')

},20000);

})

}

**Some more notes :**

[**http://blog.ninja-squad.com/2015/10/13/es6-part-2/**](http://blog.ninja-squad.com/2015/10/13/es6-part-2/) This post is the second part of a series on Angular 2, showing some chosen parts of our [ebook](https://books.ninja-squad.com/angular). You may want to read the [first part](http://blog.ninja-squad.com/2015/10/06/es6-part-1/) if you missed it!

We stopped when we were talking about classes, a new concept in JavaScript. Let’s see what we have in our second part!

# Promises

Promises are not so new, and you might know them or use them already, as they were a big part of AngularJS 1.x. But since you will use them a lot in Angular 2, and even if you’re just using JS, I think it’s important to make a stop.

Promises aim to simplify asynchronous programming. Our JS code is full of async stuff, like AJAX requests, and usually we use callbacks to handle the result and the error. But it can get messy, with callbacks inside callbacks, and it makes the code hard to read and to maintain. Promises are much nicer than callbacks, as they flatten the code, and thus make it easier to understand. Let’s consider a simple use case, where we need to fetch a user, then its rights, then update a menu when we have these.

with callbacks:

getUser(login, function(user) {

getRights(user, function(rights) {

updateMenu(rights);

});

});

now, let’s compare it with promises:

getUser()

.then(function(user) {

return getRights(user);

})

.then(function(rights) {

updateMenu(rights);

})

I like this version, because it executes as you read it: I want to fetch a user, then get its rights, then update the menu.

As you can see, a promise is a ‘thenable’ object, which simply means it has a then method. This method takes two arguments: one success callback and one reject callback. The promise has three states:

* pending: while the promise is not done, for example, our server call is not completed yet.
* fulfilled: when the promise is completed with success, for example, the server call returns an OK HTTP status.
* rejected: when the promise has failed, for example, the server returns a 404 status.

When the promise is fulfilled, then the success callback is called, with the result as an argument. If the promise is rejected, then the reject callback is called, with a rejected value or an error as the argument.

So, how do you create a promise? Pretty simple, there is a new class called Promise, whose constructor expects a function with two parameters, resolveand reject.

let getUser = function() {

return new Promise(function(resolve, reject) {

// async stuff, like fetching users from server, returning a response

if (response.status === 200) {

resolve(response.data);

} else {

reject('No user');

}

});

};

Once you have created the promise, you can register callbacks, using the thenmethod. This method can receive two parameters, the two callbacks you want to call in case of success or in case of failure. Here we only pass a success callback, ignoring the potential error:

getUser()

.then(function(user) {

console.log(user);

})

Once the promise is resolved, the success callback (here simply logging the user on the console) will be called.

The cool part is that it flattens the code. For example, if your resolve callback is also returning a promise, you can write:

getUser()

.then(function(user) {

return getRights(user) // getRights is returning a promise

.then(function(rights) {

return updateMenu(rights);

});

})

but more beautifully:

getUser()

.then(function(user) {

return getRights(user); // getRights is returning a promise

})

.then(function(rights) {

return updateMenu(rights);

})

Another interesting thing is the error handling, as you can use one handler per promise, or one for all the chain.

One per promise:

getUser()

.then(function(user) {

return getRights(user);

}, function(error) {

console.log(error); // will be called if getUser fails

return Promise.reject(error);

})

.then(function(rights) {

return updateMenu(rights);

}, function(error) {

console.log(error); // will be called if getRights fails

return Promise.reject(error);

})

One for the chain:

getUser()

.then(function(user) {

return getRights(user);

})

.then(function(rights) {

return updateMenu(rights);

})

.catch(function(error) {

console.log(error); // will be called if getUser or getRights fails

})

You should seriously look into Promises, because they are going to be the new way to write APIs, and every library will use them. Even the standard ones: the new [Fetch API](https://fetch.spec.whatwg.org/) does for example.