

Promises provide an elegant way to handle asynchronous operations in JavaScript. Once, it was the breakthrough solution to avoid callback hell. However, not many developers understand what’s going inside them. As a result, many tend to make mistakes in using Promises in practice.

In this article, I will discuss five common mistakes using Promises in JavaScript so that you can avoid them in the first place.

**1. Avoid Promise Hell**

Usually, Promises are used to avoid callback hell. But misusing them can cause Promise Hell.

Graphical user interface, text, application

Description automatically generated

In the above example, we have nested three Promises to userLogin, getArticle, and showArticle. As you can see, the complexity will grow proportionally to the lines of code and, it could become unreadable.

To avoid this we need to un-nest our code by returning getArticle from the first then and handle it in the second then .

Graphical user interface, text, application, email

Description automatically generated

**2. Using try/catch block inside the Promise Definition**

Usually, we use the try/catch block for error handling. However, using the try/catch block within the Promise object is not recommended.

That’s because if there are any errors, the Promise object will automatically handle them in the catch scope.

Graphical user interface, text, application, email

Description automatically generated

In the above example, we have used the try/catch block within the Promise scope.

But, Promise itself catches all errors (even typos) within its scope without the try/catch block. It ensures that all exceptions thrown during the execution are acquired and converted to a rejected Promise.

Graphical user interface, text, application

Description automatically generated

***Note:****It is crucial to use the .catch() block in the Promise block. Else, your test cases can fail and also the application might crash during the production phase.*

This will always work except for the next mistake I’m going to discuss.

**3. Using the Asynchronous function inside the Promise block**

Async/Await is a more advanced syntax for dealing with multiple Promises in synchronous code. When we use the async keyword before a function declaration, it returns a Promise, and we can use the await keyword to stop the code until the Promise we’re waiting for resolves or rejects.

*But there are some side effects of an Async function when you put it inside a Promise block.*

Let’s imagine you want to do an async operation in your Promise block, and you add the async keyword, and your code throws an error.

Even if you use a catch() block or wait for your Promise inside a try/catch block, you won’t be able to handle this error immediately. Check the following example:

Graphical user interface, text, application, email

Description automatically generated

When I encounter async functions inside a Promise block, I attempt to keep the async logic outside the Promise block to keep it synchronous. And it works 9 out of 10 times.

However, in some instances, an async function may be required. In that situation, you’ll have no choice but to manage it with a try/catch block manually.

Graphical user interface, text, application

Description automatically generated

**4. Executing a Promise block immediately after creating the Promise**

As for the below code snippet, if we place the code snippet to invoke an HTTP request, it gets executed immediately.

Graphical user interface, text, application, email

Description automatically generated

The reason is that the code snippet is wrapped in a Promise constructor. However, some of you might it’s only triggered after a then method when myPromise gets executed.

That, however, is not the case. Instead, when a Promise is created, the callback is immediately executed.

It means that by the time you get to the following line after establishing *myPromise,*your HTTP request is most likely already running, or at the very least scheduled state.

**Promises are always eager to the execution process.**

What should you do, though, if you desire to carry out the Promise later? What if you don’t want to make that HTTP request right now? Is there any magical mechanism built into the Promises that would allow you to do that?

The answer is often more apparent than what developers expect. Functions are a time-consuming mechanism. Only when the developer expressly invokes them with the (), they executed. Simply defining a function isn’t going to get you anywhere just yet. So, the most effective approach to make a Promise lazy is to wrap it in a function!

Graphical user interface, text, application

Description automatically generated

Promise constructor inside a function and nothing got called yet in our model. So we change the variable name since it is no longer a Promise but creates and returns a Promise object.

With an HTTP request, the Promise constructor and the callback function will only be called when the function is executed. So now we have a *lazy Promise* that is only performed when we need it.

**5. Not using Promise.all() method necessarily**

If you’re a professional developer, you already know what I’m talking about. If you have numerous Promises that are unrelated to one another, you can resolve them all simultaneously.

The Promises are concurrent, but it will take too long if you wait for them one at a time. You’ll save much time by using Promise.all().

***Remember, Promise.all() is your friend!!!***

Graphical user interface, text, application, chat or text message

Description automatically generated

The above code will take around 6 seconds to execute. but if we replace this with Promise.all() it will reduce the execution time.

Graphical user interface, text, application

Description automatically generated

The above code will take around 6 seconds to execute. but if we replace this with Promise.all() it will reduce the execution time.