

Extracted from:

Modern Asynchronous JavaScript

Tackle Complex Async Tasks with Less Code

This PDF file contains pages extracted from *Modern Asynchronous JavaScript*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit <http://www.pragprog.com>.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2021 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf

Raleigh, North Carolina

The
Pragmatic
Programmers

Modern Asynchronous JavaScript

*Tackle Complex
Async Tasks
with Less Code*

Faraz K. Kelhini
edited by Margaret Eldridge

Modern Asynchronous JavaScript

Tackle Complex Async Tasks with Less Code

Faraz K. Kelhini

The Pragmatic Bookshelf

Raleigh, North Carolina



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit <https://pragprog.com>.

The team that produced this book includes:

CEO: Dave Rankin

COO: Janet Furlow

Managing Editor: Tammy Coron

Development Editor: Margaret Eldridge

Copy Editor: L. Sakhi MacMillan

Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact support@pragprog.com.

For international rights, please contact rights@pragprog.com.

Copyright © 2021 The Pragmatic Programmers, LLC.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

ISBN-13: 978-1-68050-904-5

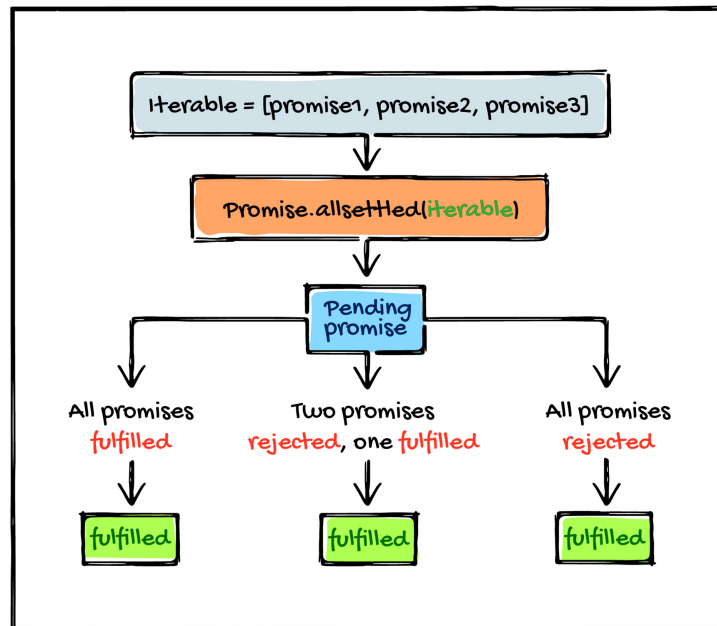
Encoded using the finest acid-free high-entropy binary digits.

Book version: P1.0—December 2, 2021

Using Promise.allSettled() to Fetch Multiple Resources

The `Promise.allSettled()` method returns a pending promise that resolves when all of the given promises have either successfully fulfilled or rejected (“settled,” in other words). This behavior is very useful to track multiple asynchronous tasks that are not dependent on one another to complete.

The following image shows how the `Promise.allSettled()` method resolves a pending promise:



In the following example, we attempt to fetch three resources, one of which doesn't exist. Notice how `Promise.allSettled()` reports the result of every promise:

`promise.allSettled/tracking_promises_ex04.js`

```
const promises = [
  fetch('https://picsum.photos/200', {mode: "no-cors"}),
  fetch('https://does-not-exist', {mode: "no-cors"}),
  fetch('https://picsum.photos/100/200', {mode: "no-cors"})
];

Promise.allSettled(promises)
  .then((results) => results.forEach((result) => console.log(result)));

// logs:
// => { status: "fulfilled", value: Response }
// => { status: "rejected", reason: TypeError }
// => { status: "fulfilled", value: Response }
```

Rather than immediately rejecting when one of the promises fails, `Promise.allSettled()` waits until they all have completed.

Notice how the result of all promises is passed as an array to `then()` and how they are in the same order as the iterable that was given even though they settled out of order. The outcome of each promise has a `status` property, indicating whether the promise has fulfilled. When a promise is rejected, the result won't have a `value` property. Instead, it has a `reason` property containing the rejection reason.

Keep in mind that the promise returned by `Promise.allSettled()` will almost always be fulfilled. The promise will reject if and only if we pass a value that's not iterable, such as a plain object.

Let's look at the rewritten version of this code, this time with the `Promise.all()` method:

`promise.allSettled/tracking_promises_ex05.js`

```
const promises = [
  fetch('https://picsum.photos/200', {mode: "no-cors"}),
  fetch('https://does-not-exist', {mode: "no-cors"}),
  fetch('https://picsum.photos/100/200', {mode: "no-cors"})
];

Promise.all(promises).
  then((results) => results.forEach((result) => console.log(result)));

// logs:
// => Uncaught (in promise) TypeError: Failed to fetch
```

This time, the promise rejects immediately upon the second input promise rejecting. One important difference between these two methods is that `Promise.allSettled()` has an extra property that `Promise.all()` doesn't: `status`. In fact, `Promise.all()` returns the raw value that `Promise.allSettled()` tucks into its resulting object. Compare:

`promise.allSettled/tracking_promises_ex06.js`

```
const promises = [
  Promise.resolve(1),
  Promise.resolve(2)
];

Promise.allSettled(promises).
  then((results) => results.forEach((result) => console.log(result)));

// logs:
// => { status: "fulfilled", value: 1 }
// => { status: "fulfilled", value: 2 }

Promise.all(promises).
  then((results) => results.forEach((result) => console.log(result)));
```

```
// logs:
// => 1
// => 2
```

Notice how `Promise.all()` directly returns the response. If you're in an old JavaScript environment that doesn't support `Promise.allSettled()` or if you'd like to directly return the promises, there's a simple workaround for you. Consider the following code:

```
promise.allSettled/tracking_promises_ex07.js
const promises = [
  fetch('https://picsum.photos/200', {mode: "no-cors"}),
  fetch('https://does-not-exist', {mode: "no-cors"}),
  fetch('https://picsum.photos/100/200', {mode: "no-cors"})
].map(p => p.catch(e => e));

Promise.all(promises)
  .then((results) => results.forEach((result) => console.log(result)));
```

Here, we've applied the `map()` method to an iterable of promises. Within the method, we use `catch()` to return promises that resolve with an error value. This way, we can simulate the behavior of `Promise.allSettled()` while being able to directly access the result of promises.

Often, we use `Promise.all()` and `Promise.allSettled()` with similar types of requests, but there's no written rule that we should. You may find yourself in a situation where you need to read a local file, retrieve a JSON document from a web API, and load an XML document from another API. Once you obtain data from all three async requests, you want to process them. `Promise.all()` and `Promise.allSettled()` are ideal for such scenarios.

Keep in mind that you will want to use these methods only when you need to process the result of multiple async requests together. If it's possible to process the result of each async request individually, then handle each promise with its own `then()` handler. This way, you can execute your code as soon as each promise is resolved.

Wrapping Up

In this chapter, we looked at potential pitfalls when executing multiple promises at the same time. We learned why looping over asynchronous tasks could be a bad idea because it will cause the promises to run sequentially. Then we learned about the `Promise.allSettled()` method and compared it to `Promise.all()`.

While `Promise.all()` is very strict in its execution policy, `Promise.allSettled()` is forgiving. That doesn't mean `Promise.allSettled()` is superior to `Promise.all()`: they comple-

ment each other. Using `Promise.all()` is more appropriate when you have essential async tasks that are dependent on each other. On the other hand, `Promise.allSettled()` is more suitable for async tasks that might fail but are not essential for your program to function.

As of ES2021, the ECMAScript standard includes one more method for the promise object: `Promise.any()`. This method is the opposite of `Promise.all()`. In the next chapter, we're going to learn how `Promise.any()` can help you when you need to focus on the promise that resolves first.