Chapter 3: Fetching Multiple Resources

Adam Isom, Caleb Smith, Steven Ni

Outline

1 - Background on Promises & Async -> Steven

2 - Promise.all vs. allSettled -> Caleb

3 - Code our own version of the above -> Adam

Q: When should the `Promise` constructor be used?

```
function retrieveUrl() {
   return new Promise(function(resolutionFunc, rejectionFunc) {
     // Async task, retrieve url
     let urlRetrieved = true;
       if (urlRetrieved) {
           resolutionFunc('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js');
       } else {
           rejectionFunc('Failed to retrieve url.');
       }
   });
 }

 let promise = retrieveUrl();
 console.log(promise);

 promise
 .then(function onFulfilled (resolvedValue) {
   console.log(`Success: ${resolvedValue}`);
 }, function onRejected(rejectionReason) {
   console.log(`Error: ${rejectionReason}`);
 });
```

```
Promise {<fulfilled>: 'https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js'}
    [[Prototype]]: Promise
        catch: ƒ catch()
        constructor: ƒ Promise()
            all: ƒ all()
            allSettled: ƒ allSettled()
            any: ƒ any()
            length: 1
            name: "Promise"
            prototype: Promise {Symbol(Symbol.toStringTag): 'Promise', constructor: ƒ, then: ƒ, catch: ƒ, finally: ƒ}
            race: ƒ race()
            reject: ƒ reject()
            resolve: ƒ resolve()
            Symbol(Symbol.species): ƒ Promise()
            Symbol(Symbol.species): ƒ Promise()
            arguments: (...)
            caller: (...)
            [[Prototype]]: ƒ ()
            [[Scopes]]: Scopes[0]
        finally: ƒ finally()
        then: ƒ then()
        Symbol(Symbol.toStringTag): "Promise"
        [[Prototype]]: Object
    [[PromiseState]]: "fulfilled"
    [[PromiseResult]]: "https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js"
```

```
function retrieveUrl() {
   return new Promise(function(resolutionFunc, rejectionFunc) {
     // Async task, retrieve url
     let urlRetrieved = true;
       // if (urlRetrieved) {
       //     resolutionFunc('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js');
       // } else {
       //     rejectionFunc('Failed to retrieve url.');
       // }
   });
 }
 let promise = retrieveUrl();
 console.log(promise);

 promise
 .then(function onFulfilled (resolvedValue) {
   console.log(`Success: ${resolvedValue}`);
 }, function onRejected(rejectionReason) {
   console.log(`Error: ${rejectionReason}`);
 });

// console.log of pending promise object and resulting value
Promise {<pending>}
[[Prototype]]: Promise
[[PromiseState]]: "pending"
[[PromiseResult]]: undefined
```

```
function retrieveUrl() {
   return new Promise(function(resolutionFunc, rejectionFunc) {
     // Async task, retrieve url
     let urlRetrieved = true;
       if (urlRetrieved) {
           resolutionFunc('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js');
       } else {
           rejectionFunc('Failed to retrieve url.');
       }
   });
 }

 let promise = retrieveUrl();
 console.log(promise);

 promise
 .then(function onFulfilled (resolvedValue) {
   console.log(`Success: ${resolvedValue}`);
 }, function onRejected(rejectionReason) {
   console.log(`Error: ${rejectionReason}`);
 });

// console.log of fulfilled promise object and resulting value
Promise {<fulfilled>: 'https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js'}
[[Prototype]]: Promise
[[PromiseState]]: "fulfilled"
[[PromiseResult]]: "https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js"
```

```
function retrieveUrl() {
   return new Promise(function(resolutionFunc, rejectionFunc) {
     // Async task, retrieve url
     let urlRetrieved = false;
       if (urlRetrieved) {
           resolutionFunc('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js');
       } else {
           rejectionFunc('Failed to retrieve url.');
       }
   });
 }

 let promise = retrieveUrl();
 console.log(promise);

 promise
 .then(function onFulfilled (resolvedValue) {
   console.log(`Success: ${resolvedValue}`);
 }, function onRejected(rejectionReason) {
   console.log(`Error: ${rejectionReason}`);
 });

// console.log of rejected promise object and resulting value
Promise {<rejected>: 'Failed to retrieve url.'}
[[Prototype]]: Promise
[[PromiseState]]: "rejected"
[[PromiseResult]]: "Failed to retrieve url."
```

Q: What happens after the promise is settled?

```
function retrieveUrl() {
   return new Promise(function(resolutionFunc, rejectionFunc) {
     // Async task, retrieve url
     let urlRetrieved = true;
       if (urlRetrieved) {
           resolutionFunc('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js');
       } else {
           rejectionFunc('Failed to retrieve url.');
       }
   });
 }

 let promise = retrieveUrl();
 console.log(promise);

 promise
 .then(function onFulfilled (resolvedValue) {
   console.log(`Success: ${resolvedValue}`);
 }, function onRejected(rejectionReason) {
   console.log(`Error: ${rejectionReason}`);
 });
// console.log of fulfilled promise object and resulting value
Promise {<fulfilled>: 'https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js'}

// console.log of the 'retrieved url' from the onFulfilled handler
Success: https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js
```

```
function retrieveUrl() {
   return new Promise(function(resolutionFunc, rejectionFunc) {
     // Async task, retrieve url
     let urlRetrieved = false;
       if (urlRetrieved) {
           resolutionFunc('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js');
       } else {
           rejectionFunc('Failed to retrieve url.');
       }
   });
 }

 let promise = retrieveUrl();
 console.log(promise);

 promise
 .then(function onFulfilled (resolvedValue) {
   console.log(`Success: ${resolvedValue}`);
 }, function onRejected(rejectionReason) {
   console.log(`Error: ${rejectionReason}`);
 });

// console.log of rejected promise object and resulting value
Promise { <rejected> 'Failed to fetch url.' }

// console.log of the rejection reason from the onRejected handler
Error: Failed to fetch url.
```

Q: What is the `async...await` syntax and why use it?

```
let myPromise = () => {
   return new Promise((resolve, reject) => {
     setTimeout(() => {
       resolve('I am the resolved value.')
     }, 1000);
   });
 }

  async function noAwait() {
  let value = myPromise();
  console.log(value);
 }

 async function yesAwait() {
  let value = await myPromise();
  console.log(value);
 }

 noAwait(); // Prints: Promise { <pending> }
 yesAwait(); // Prints: I am the resolved value.
```

Loading Multiple Resources

Single Resource

promise.allSettled/tracking_promises_ex01.js

```
async function getPost(id = 1) {
  try {
    return await Utility.loadPost(id);
  } catch (error) {
    // handle error
  }
}
```

Problem: Only fetches one resource

Loop to retrieve multiple resources

promise.allSettled/tracking_promises_ex02.js

```
const postIds = ['1', '2', '3', '4'];

postIds.forEach(async id => {
  const post = await getPost(id);

  // process the post
})
```

Problem: loads posts sequentially

Q: What are the advantages and disadvantages of fetching multiple resources versus an individual resource?

Promise.All()

The **Promise.all()** method takes an iterable of promises as an input, and returns a single Promise that resolves to an array of the results of the input promises.

```
1  const promise1 = Promise.resolve(3);
2  const promise2 = 42;
3  const promise3 = new Promise((resolve, reject) => {
4    setTimeout(resolve, 100, 'foo');
5  });
6
7  Promise.all([promise1, promise2, promise3]).then((values) => {
8    console.log(values);
9  });
10 // expected output: Array [3, 42, "foo"]
11
```

promise.allSettled/tracking_promises_ex03.js

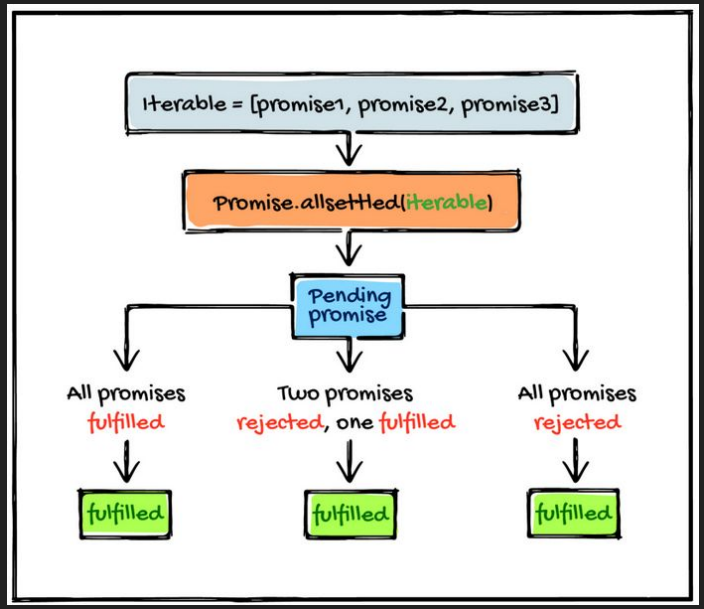
```
const postIds = ['1', '2', '3', '4'];

const promises = postIds.map(async (id) => {
  return await getPost(id);
});

const arr = Promise.all(promises);
```

If one of the promises in the iterable rejects, Promise.All() immediately rejects, causing every other post not to load.

Q: When would you want to use Promise.All() instead of Promise.allSettled()?

Promise.allSettled()

The **Promise.allSettled()** method returns a promise that resolves after all of the given promises have either fulfilled or rejected, with an array of objects that each describes the outcome of each promise.




```
1  const promise1 = Promise.resolve(3);
2  const promise2 = new Promise((resolve, reject) => setTimeout(reject, 100, 'foo'));
3  const promises = [promise1, promise2];
4
5  Promise.allSettled(promises).
6    then((results) => results.forEach((result) => console.log(result.status)));
7
8  // expected output:
9  // "fulfilled"
10 // "rejected"
11
```

Demo: Plan

Let’s write our own Promise.all

What does it do?

- input: takes many promises - let’s say, an array
- output: returns a promise
- that resolves after (a) any error or rejection, or (b) all successful fulfillments
- and that resolves to either (a) the error, or (b) an array of the fulfillments

Edge cases?

- if input array empty: return a promise already resolved (to empty array)
- if input array contains non-promises: **we won’t handle this**
    - if we did: non-promise values wrapped in a promise that is already resolved to that value

Demo: Plan

```
IF input empty
 RETURN empty array
SET fulfillment values to empty array
SET number of promises fulfilled to zero
RETURN new promise, and in that promise
 FOR EACH input promise 'p'
   IF p rejects
     Reject
   ELSE
     Store fulfillment value in correct index/location
     Increment number of promises fulfilled
     IF all promises fulfilled
       Resolve
```

```
// should return empty array if given empty array input
all([]).then(result => console.log(result)); // []

// should return array of fulfillments if given array of promises that fulfill
all([Promise.resolve(2), Promise.resolve(3)])
  .then(result => console.log(result)); // [2, 3]

all([Promise.resolve(2), new Promise((resolve, reject) => {
  setTimeout(() => resolve(3), 1000);
})]).then(result => console.log(result)); // [2, 3]

// should return reject value if given array of promises, one of which rejects
const a = new Promise((resolve, reject) => {
  setTimeout(() => resolve('a'), 2000);
});
const b = new Promise((resolve, reject) => {
  setTimeout(() => reject('rejected'), 1000);
});
const c = new Promise((resolve, reject) => {
  setTimeout(() => resolve('c'), 3000);
});

all([a, b, c])
  .then(result => console.log(result),
    error => console.log(error)); // 'rejected'
```

```
function all(promises) {
  if (promises.length === 0) return Promise.resolve([]);

  return new Promise((resolve, reject) => {
    const fulfillmentValues = new Array(promises.length).fill(null);
    let numberOfPromisesFulfilled = 0;

    promises.forEach((promise, index) => {
      promise.then(result => {
        fulfillmentValues[index] = result;
        if (++numberOfPromisesFulfilled === promises.length) {
          resolve(fulfillmentValues);
        }
      }, error => reject(error));
    });
  });
}
```

One-Question Quiz:

What would we change to write `allSettled`?

Answer

1. If a child promise rejects, don’t reject; instead, store the rejection value.

(So maybe we would rename `fulfillmentValues` to `resolutionValues`.)

Implication: allSettled never rejects.

2. Add metadata: resolve the promise with array of objects of form

{ status: “fulfilled”, value: <fulfillment value> } ***or***

{ status: “rejected”, reason: <rejection value> }

Thanks for attending