

Class #22 – Challenges 01

Problems using Promises

Objectives:

- How to create a promise.
- How to consume a promise.
- How to chain promises.
- To learn for seeking information, reading documentation and checking out topics about promises, setTimeout(), XMLHttpRequest (XHR) and data-interchange format (JSON).

Theory & Documentation

A Promise in Layman Terms

Check this conversation between two people:

Wendy: Hey Mr. Promise! Can you run to the store down the street and get me sauce for this chilaquiles we are cooking tonight?

Mr. Promise: Sure thing!

Wendy: While you are doing that, I will prepare fried tortillas (asynchronous operation). But make sure you let me know whether you could find sauce (promise return value).

Mr. Promise: What if you are not at home when I am back?

Wendy: In that case, send me a text message saying you are back and have the sauce for me (success callback). If you don't find it, call me immediately (failure callback).

Mr. Promise: Sounds good! See you in a bit.

You can check in simply terms that a ***promise object*** is data returned by asynchronous function. It can be a ***resolve*** if the function returned successfully or a ***reject*** if function returned an error.

Promise Terminology

A Promise is an object representing the eventual completion or failure of an asynchronous operation. Essentially, a promise is a returned object you attach callbacks to, instead of passing callbacks into a function.

At their most basic, promises are a bit like event listeners except:

- A promise can only succeed or fail once. It cannot succeed or fail twice, neither can it switch from success to failure or vice versa.
- If a promise has succeeded or failed and you later add a success/failure callback, the correct callback will be called, even though the event took place earlier.

This is extremely useful for async success/failure, because you're less interested in the exact time something became available, and more interested in reacting to the outcome.

A promise has 3 states. They are:

1. **Pending**: awaiting promise response.
2. **Resolve**: promise has successfully returned.
3. **Reject**: failure occurred.

Creating a Promise

In a typical web-app, you could be doing multiple asynchronous operations, such as fetching images, getting data from JSON endpoint, talking to an API, etc.

Here's how you create a promise:

```
var promise = new Promise(function(resolve, reject) {  
    // do a thing, possibly async, then...  
  
    if (/* everything turned out fine */) {  
        resolve("Stuff worked!");  
    }  
    else {  
        reject(Error("It broke"));  
    }  
});
```

Steps to create a promise:

- 1) The **promise constructor** takes one argument.
- 2) A **callback** with two parameters.
- 3) **Resolve** and **reject**. Do something within the callback, perhaps async, then call resolve if everything worked, otherwise call reject.

Consuming a Promise

The promise from above can then be used like this:

```
promise.then(function(result) {  
    console.log(result); // "Stuff worked!"  
}, function(err) {  
    console.log(err); // Error: "It broke"  
});
```

'**.then()**' takes two arguments, a callback for a success case, and another for the failure case. Both are optional, so you can add a callback for the success or failure case only.

If promise was successful, a resolve will happen and the console will log **Stuff worked**, otherwise **It broke**. That state between **resolve** and **reject** where a response hasn't been received is **pending** state.

Chaining Promises

You can chain then's together to transform values or run additional async actions one after another.

Transforming values

You can transform values simply by returning the new value:

```
var promise = new Promise(function(resolve, reject) {
  resolve("Hello");
});

promise.then(function(str) {
  console.log(str); // Hello
  return str + " " + "World";
}).then(function(str) {
  console.log(str); // Hello World
})
```

Error Handling

As we saw earlier, then() takes two arguments, one for success, one for failure (or fulfill and reject). Rejections happen when a promise is explicitly rejected, but also implicitly if an error is thrown in the constructor callback. We can use '.catch()' to display an error to the user:

```
var jsonPromise = new Promise(function(resolve, reject) {
  // JSON.parse throws an error if you feed it some
  // invalid JSON, so this implicitly rejects:
  resolve(JSON.parse("This ain't JSON"));
});

jsonPromise.then(function(data) {
  // This never happens:
  console.log("It worked!", data);
}).catch(function(err) {
  // Instead, this happens:
  console.log("It failed!", err);
})
```

This means it's useful to do all your promise-related work inside the promise constructor callback, so errors are automatically caught and become rejections.

The same goes for errors thrown in then() callbacks:

```
get('/').then(JSON.parse).then(function() {  
  // This never happens, '/' is an HTML page, not JSON  
  // so JSON.parse throws  
  console.log("It worked!", data);  
}).catch(function(err) {  
  // Instead, this happens:  
  console.log("It failed!", err);  
})
```

More information about promises: [Javascript Promises for Dummies](#).

Challenge 01 - 1 (Instructions) – setTimeout

Problem using Chaining

Steps

- Download the 'challenge-01--files.zip' file in Trello.
- You must run your app in the server, use node js:

1) Install http-server by running:

```
$ npm install http-server -g
```

2) Use cd to find your directory app.

3) Start http-server by running:

```
$ http-server -c-1 -p 8000
```

Deriverables

- Upload the github repository url to trello.

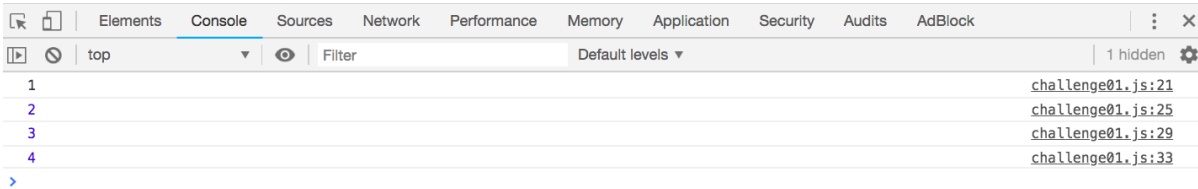
Final result

Our client has the next problem using setTimeout in Javascript:

```
console.log("1");  
setTimeout(function(){console.log("2");},3000);  
console.log("3");  
setTimeout(function(){console.log("4");},1000);
```

The output will be 1 3 4 2 . Client wonders why 4 is logged before 2. The reason for that is that even though line 2 started executing before line 4 , it did not start executing for 3000ms and hence 4 is logged before 2 .

IMPORTANT. Client has decided to use promises to resolve this problem. Check final result:



Final result

Challenge 01 - 2 (Instructions) – GET Request

Problem using a Promise

Steps

- Download the 'challenge-01--files.zip' file in Trello.
- You must run your app in the server, use node js:

1) Install http-server by running:

```
$ npm install http-server -g
```

2) Use cd to find your directory app.

3) Start http-server by running:

```
$ http-server -c-1 -p 8000
```

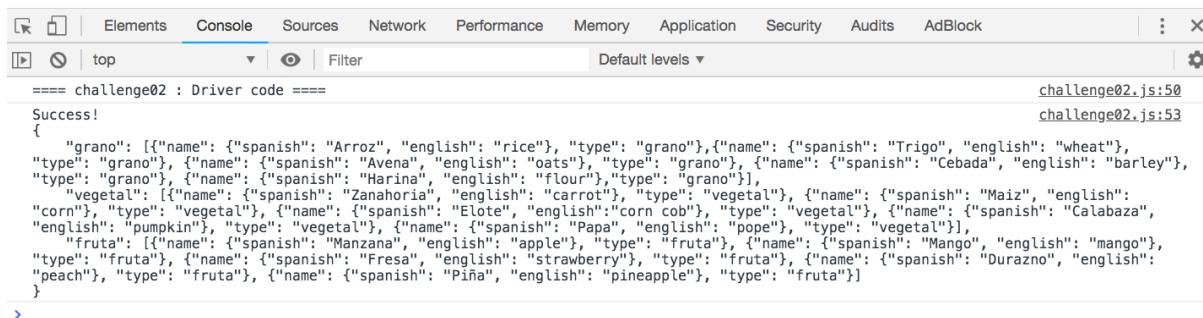
Deriverables

- Upload the github repository url to trello.

Final result

Our client has decided to make a GET [request](#) from data [json](#) file using a promise and [XMLHttpRequest](#) (XHR).

IMPORTANT. Client has decided to use its code structure for this algorithm.



```
==== challenge02 : Driver code ====
Success!
{
  "grano": [{ "name": { "spanish": "Arroz", "english": "rice" }, "type": "grano" }, { "name": { "spanish": "Trigo", "english": "wheat" }, "type": "grano" }, { "name": { "spanish": "Avena", "english": "oats" }, "type": "grano" }, { "name": { "spanish": "Cebada", "english": "barley" }, "type": "grano" }, { "name": { "spanish": "Harina", "english": "flour" }, "type": "grano" } ],
  "vegetal": [{ "name": { "spanish": "Zanahoria", "english": "carrot" }, "type": "vegetal" }, { "name": { "spanish": "Maiz", "english": "corn" }, "type": "vegetal" }, { "name": { "spanish": "Elote", "english": "corn cob" }, "type": "vegetal" }, { "name": { "spanish": "Calabaza", "english": "pumpkin" }, "type": "vegetal" }, { "name": { "spanish": "Papa", "english": "potato" }, "type": "vegetal" }, { "name": { "spanish": "Pepino", "english": "cucumber" }, "type": "vegetal" } ],
  "fruta": [{ "name": { "spanish": "Manzana", "english": "apple" }, "type": "fruta" }, { "name": { "spanish": "Mango", "english": "mango" }, "type": "fruta" }, { "name": { "spanish": "Fresa", "english": "strawberry" }, "type": "fruta" }, { "name": { "spanish": "Durazno", "english": "peach" }, "type": "fruta" }, { "name": { "spanish": "Piña", "english": "pineapple" }, "type": "fruta" } ]
}
```

Final result

Challenge 01 - 3 (Instructions) – Sum Problem using a Promise and Chaining

Steps

- Download the 'challenge-01--files.zip' file in Trello.
- You must run your app in the server, use node js:

1) Install http-server by running:

```
$ npm install http-server -g
```

2) Use cd to find your directory app.

3) Start http-server by running:

```
$ http-server -c-1 -p 8000
```

Deriverables

- Upload the github repository url to trello.

Final result

Our client has decided to develop a sum algorithm using a promise and chaining for solving a problem of transforming values. It is necessary catch error and display in browser for user.

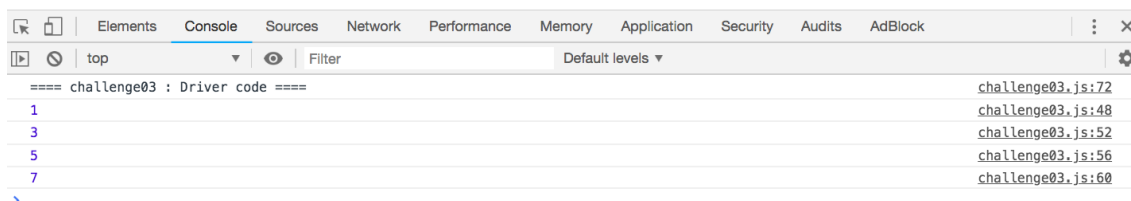
IMPORTANT. Client has decided to use its code structure for this algorithm.

Check below some output examples:

Output 1 --> when argument is a number: 1

Welcome

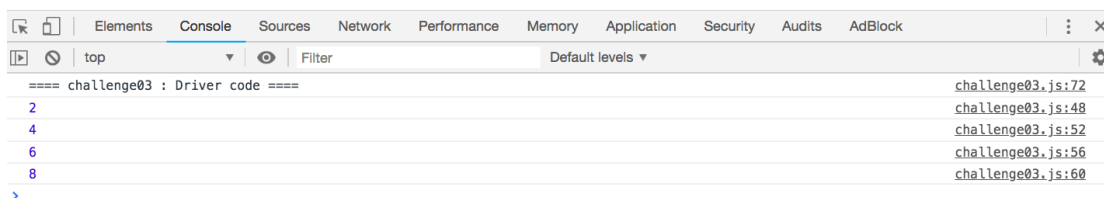
- 1
- 3
- 5
- 7



Output 2 --> when argument is a number: 2

Welcome

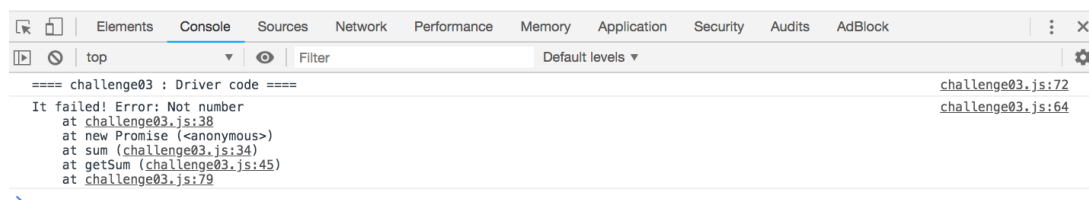
- 2
- 4
- 6
- 8



Output 3 --> when argument is not a number

Welcome

It failed! Error: Not number



Challenge 01 - 4 (Instructions) – GET Request

Problem using a Promise and Chaining

Steps

- Download the 'challenge-01--files.zip' file in Trello.
- You must run your app in the server, use node js:

1) Install http-server by running:

```
$ npm install http-server -g
```

2) Use cd to find your directory app.

3) Start http-server by running:

```
$ http-server -c-1 -p 8000
```

Deriverables

- Upload the github repository url to trello.

Final result

Our client has decided to make a GET [request](#) using a promise and chaining to parse data [json](#). It is necessary to use [XMLHttpRequest](#) (XHR).

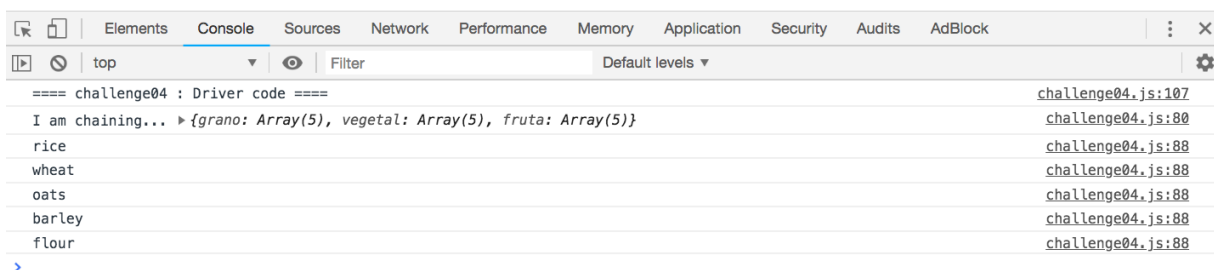
IMPORTANT. Client has decided to use its code structure and some restrictions for this algorithm.

Check below some output examples:

Output 1 --> when it exists a food

Welcome

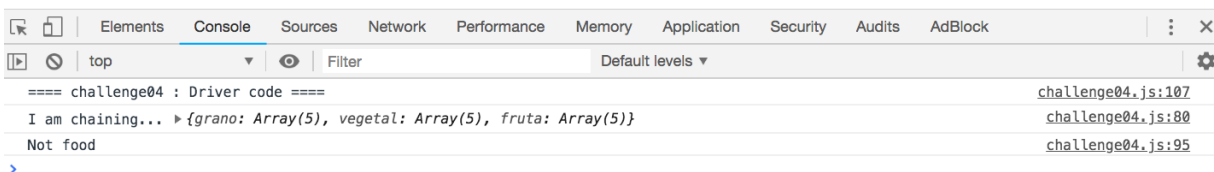
- rice
- wheat
- oats
- barley
- flour



Output 2 --> when it does not exist

Welcome

Not Food



Output 3 --> when not found data file

Welcome

