Jesus Ortega

JavaScript ASYNC Notes Part 2

PROMISES

Promises are the foundation of asynchronous programming in modern JS. A *promise* is an object that is returned by an async function, it represents the current state of the operation. When it is returned the operation is not finished, but the promise object has methods to handle the eventual success or failure of the operation.

With a promised-based API, asynchronous functions start the operation and return a promise object. Then you can attach handlers to the promise object, and the handlers will be executed when the operation has succeeded or failed.

USING THE fetch() API

Example 1: Download JSON file and log some information about it

To do this:

Make an HTTP request to the server. In an HTTP request, we send a message to a remote server, and it sends us back a response. In this case, we'll send a request to get a JSON file from the server. We will use the fetch() API (A modern, promise-based replacement for XMLHttpRequest).

```
const fetchPromise =
fetch('https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json');

console.log(fetchPromise);
```

```
fetchPromise.then((response) => {
  console.log(`Received response: ${response.status}`);
});

console.log("Started request…");
```

In the above code we are:

1. Calling the **fetch()** API, and assigning the return value to the fetchPromise variable.
2. Immediately after, we are logging the *fetchPromise* variable. This should output something like: **Promise {<state>: “pending”}**. This means we have a promise object and it has a state with a value of “pending”. “**pending**” means that the fetch operation is still going on.
3. Then we are passing a handler function into the Promise’s **then()** method. When (and if) the fetch operation succeeds, the promise will call our handler, passing in a **Response** object. This object contains the server’s response.
4. Finally we log a message that we have started the request.

The complete output would be something like this:

```
Promise {<pending>}

Started request…

Received response: 200
```

Note that Started request… is logged before we receive the response. Unlike synchronous functions, fetch() returns while the request is still going on. This enables our program to stay responsive. The response shows the 200 (OK) status code. This means our request succeeded.

CHAINING PROMISES

Once you get a Response Object from the fetch() API, you need to call another function to get the response data. In this case, we want to get the response data as a JSON, so we would call the json() method of the Response Object. It so happens that json() is also asynchronous. So this example is a case where we have to call two successive asynchronous functions.

```
const fetchPromise =
fetch('https://mdn.github.io/learning-area/javascript/apis/fetching-data/c
an-store/products.json');

fetchPromise.then((response) => {
  const jsonPromise = response.json();
  jsonPromise.then((data) => {
    console.log(data[0].name);
  });
});
```

In this example, as before, we add the **then()** handler to the promise returned by **fetch()**. But this time our handler calls **response.json()** and then passes a new **then()** handler into the promise returned by **response.json()**.

This logs: *baked beans* The name of the first product listed in the products.json file we downloaded.

To keep code clean and less nested, we need to remember that **then()** itself returns a promise, which will be completed with the result of the function passed to it. We can rewrite the above code like this:

```
const fetchPromise =
fetch('https://mdn.github.io/learning-area/javascript/apis/fetching-data/c
an-store/products.json');

fetchPromise
  .then((response) => response.json())
  .then((data) => {
    console.log(data[0].name);
  });
```

In the above code instead of calling the second then() inside the handler for the first then(), we can return the promise returned by json(), and call the second then() on that return value. This is called **promise chaining**. It means we can avoid ever-increasing levels of indentation when we need to make consecutive async function calls.

To check if the server accepted and was able to handle the request we will check the status code in the response and throw an error if it wasn't OK.

The above code should be changed to this:

```
const fetchPromise =
fetch('https://mdn.github.io/learning-area/javascript/apis/fetching-data/c
an-store/products.json');

fetchPromise
  .then((response) => {
    if (!response.ok) {
      throw new Error(`HTTP error: ${response.status}`);
    }
    return response.json();
  })
  .then((data) => {
    console.log(data[0].name);
  });
```

CATCHING ERRORS

To support error handling, Promise objects provide a **catch()** method. This is a lot like **then():** you call it and pass in a handler function. However, while the handler passed to then() is called when the asynchronous operation succeeds, the handler passed to **catch()** is called when the asynchronous operation fails.

If you add **catch()** to the end of a promise chain, then it will be called when any of the asynchronous function calls fails. You can implement an operation as several consecutive asynchronous function calls, and have one place to handle all errors.

Lets add an error handler to our fetch() code using catch():

```
const fetchPromise =
fetch('bad-scheme://mdn.github.io/learning-area/javascript/apis/fetching-d
ata/can-store/products.json');

fetchPromise
  .then((response) => {
    if (!response.ok) {
      throw new Error(`HTTP error: ${response.status}`);
    }
    return response.json();
  })
  .then((data) => {
    console.log(data[0].name);
  })
  .catch((error) => {
    console.error(`Could not get products: ${error}`);
  });
```

Running this code should log an error by our catch() handler. NOTE: we modified the url so that the request would fail.

PROMISE TERMINOLOGY

A promise can be in one of three states:

1. **pending**: the promise has been created, and the asynchronous function it's associated with has not succeeded or failed yet. This is the state your promise is in when it's returned from a call to fetch(), and the request is still being made.
2. **fulfilled**: the asynchronous function has succeeded. When a promise is fulfilled, its then() handler is called.
3. **rejected**: the asynchronous function has failed. When a promise is rejected, its catch() handler is called.

Sometimes we can use the term **settled** to cover both fulfilled and rejected. A promise is **resolved** if it is settled, or if it has been "locked in" to follow the state of another promise.

NOTE: **fetch()** will consider a request successful if the server returns an error like 404 NOT FOUND, but not if a network error prevented the request from being sent.