JavaScript

Muchang Bahng

Fall 2024

Contents

1 Events

There are many events that we can handle. Say that we have an event `event`. Then, upon this event, we can have it call some JavaScript code of the form.

```
<p id="paragrpah" event="function()"></p>
```

If we think of each tag as an object, these events are really attributes of this object that point to JS functions.

Example 1.1 (Significant Events)

It's worth mentioning a couple events that are important.

1. `onclick`. When a user clicks on something.
2. `onload`. When a tag loads.
3. `onerror`. When a tag fails to load.
4. `onkeydown`. When a key is pressed down.

2 Asynchronous Handling

Let's talk about the architecture of the JavaScript runtime environment, which has a bit more components than that of other languages like C or Python. This allows better handling of asynchronous code and provides additional objects by the browser.

Definition 2.1 (JavaScript Runtime Environment)

It contains the following.

1. The *JavaScript Engine* consists of the *stack* and the *heap* handles function calls and allows access to larger pools of memory, respectively.
2. The *Web/Browser API*, separate fro the JS Engine, can be communicated with using JS, enabling us to do things concurrently outside of the JS interpreter. The language itself is single-threaded, but the browser APIs act as separate threads. Callback functions from the API is sent to the task queue.
3. The *callback/task queue* is a message queue where each message has an associated function to be called. After the call stack is emptied, during the Event Loop, runtime handles the first message in the queue by callings its functions and popping them onto to the call stack.
4. The *Microtask queue* is a higher priority of the task queue and handles Microtasks callbacks.
5. The *event loop* constantly checks whether or not the call stack is empty. When the call stack is empty, all queued up microtasks from this queue are popped onto the call stack. If both the call stack and the microtask queue are empty, the event loops dequeues tasks from the task queue and calls them.

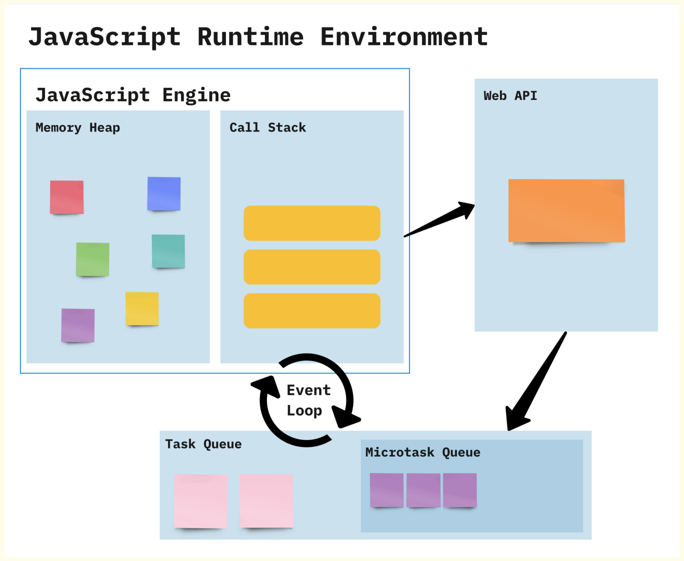

Figure 1: The JS runtime environment contains task queues, the Web API, and the event handler.

Three common functions in the web API are

1. `setTimeout(function, ms)`, which delays the call of a function by `ms` milliseconds.
2. `setInterval(function, ms)`, which repeats the call of a function every `ms` milliseconds.
3. `addEventListener(type, listener)`, which adds a listener that scans for some event, e.g. a mouse click, scroll, hover, etc.

In the regular call stack, we can already see that calling these functions will just stop everything. `setTimeout` will stop everything for some time before its argument function is called, and an event listener will repeatedly call some function to detect an event, overflowing the stack.

2.1 Callback Functions

Definition 2.2 (Callback Function)

A **callback function** is a function that is passed as an argument to another function and is executed at the completion of that function.

This is useful since it guarantees that asynchronous tasks that gets moved to the web API is called.

Definition 2.3 (Callback Hell)

At some points,

2.2 Promises

Definition 2.4 (Promise)

A `Promise` object is a wrapper around a (HTTP) request. It has the following attributes.

1. `PromiseState` starts off as `pending` as we request the packet, and then depending on if it successfully retrieved the data (called *resolved*) or not, it updates to `OK` or `ERR`.
2. `PromiseResult`

3. `PromiseFulfillReactions`
4. `PromiseRejectReactions`
5. `PromiseIsHandled`

It is constructed with a function that takes in two arguments.

1. A `resolve` function that is called when the promise is successful. When `resolve(val)` is called, `val` is stored in `PromiseResult`.
2. A `reject` function that is called when the promise is unsuccessful. When `reject(val)` is called, `val` is stored in `PromiseResult`.

Example 2.1 (Resolve without Asynchronous Functions)

A trivial promise object is constructed that calls `resolve` or `reject`.

```
let p = new Promise(
  function(resolve, reject) {
    resolve(1);
  }
)
console.log(p);

Promise {
  1,
  [Symbol(async_id_symbol)]: 28,
  [Symbol(trigger_async_id_symbol)]: 6
}
.
.
```

```
let p = new Promise(
  function(resolve, reject) {
    reject("bruh");
  }
)

console.log(p);
Promise {
  <rejected> 'bruh',
  [Symbol(async_id_symbol)]: 29,
  [Symbol(trigger_async_id_symbol)]: 6
}
undefined
Uncaught 'bruh'
```

2.3 Then, Catch, Finally

A `Promise` object has three methods that tells us how to *react* depending on the value of `PromiseResult`, and it ends up returning a *new* Promise object with its own `PromiseResult` attribute being what was returned by the method.

1. `then` is used to react mainly to successful resolves and sometimes to errors. `then(f, g)` will call `f(PromiseResult)` when the promise is resolved and `g(PromiseResult)` if not. It takes two arguments:

2. `catch` is used to resolve errors, which takes in one argument that will be called with `PromiseResult` when it is rejected. `catch(g)` will call `g(PromiseResult)` upon error. It can be used multiple times and in between `then` handlers, so when a promise rejects, the control jumpt to the closest rejection handler.

3. `finally` is used as a cleanup method and is always called. `finally(f)` is equivalent to `then(f, f)`.

Example 2.2 ()

Here is an example of how we use then statements.

```
let promise = new Promise(
  function(resolve, reject) {
    setTimeout(
      () => resolve("done!"), 1000
    );
  }
);
// resolve runs 1st function in .then
promise.then(
  // shows "done!" after 1 second
  result => alert(result),
  // doesn't run
  error => alert(error)
);
```

```
let promise = new Promise(
  function(resolve, reject) {
    setTimeout(() => reject(
      new Error("Whoops!")), 1000
    );
  }
);
// reject runs 2nd function in .then
promise.then(
  // doesn't run
  result => alert(result),
  // "Error: Whoops!" after 1 sec
  error => alert(error)
);
```

Since calling a bunch of `then`'s gives us a new Promise, we can chain them together to get a bunch of promises. Check this out.

```
new Promise(function(resolve, reject) {

  setTimeout(() => resolve(1), 1000); // (*)

}).then(function(result) { // (**)

  alert(result); // 1
  return result * 2;

}).then(function(result) { // (***)

  alert(result); // 2
  return result * 2;

}).then(function(result) {

  alert(result); // 4
  return result * 2;

});
```

1. The initial promise is resolved with a value of 1.
2. The `then` handler creates a new promise resolved with value 2.
3. the next `then` handler creates a new promise with value 4.

Since we have a bunch of alert calls, we will be alerted with values of 1, 2, and 4. Rather than returning a bunch of numbers we can also return the promise object itself. The following code is equivalent, but with a 1 second delay in between.

```
new Promise(function(resolve, reject) {

  setTimeout(() => resolve(1), 1000);

}).then(function(result) {

  alert(result); // 1

  return new Promise((resolve, reject) => { // (*)
    setTimeout(() => resolve(result * 2), 1000);
  });

}).then(function(result) { // (**)

  alert(result); // 2

  return new Promise((resolve, reject) => {
    setTimeout(() => resolve(result * 2), 1000);
  });

}).then(function(result) {

  alert(result); // 4

});
```

'

Note that a classic newbie error is that the following is not promise chaining, as it produces alerts 1, 1, and 1.

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve(1), 1000);
});

promise.then(function(result) {
  alert(result); // 1
  return result * 2;
});

promise.then(function(result) {
  alert(result); // 1
  return result * 2;
});

promise.then(function(result) {
  alert(result); // 1
  return result * 2;
});
```

This is because we are just adding several handlers to one promise, and so we are not taking the new promise returned from the previous handler into the next handler.

Example 2.3 (Quotes API)

Let's use this to process some quotes. We call fetch on some API url, which returns a promise object. I use the `then` handler to call a function that first preprocesses the promise result into a json format. Once this is done, I call another function to print this json data. If any of these don't work, i.e. the Promise calls the `reject` function, then there is a catch statement to log the error.

```
let p = fetch("https://zenquotes.io/api/random")
  .then((resolve) => resolve.json())
  .then((data) => console.log(data))
  .catch((err) => console.log(err));

...
[
  {
    q: 'Happiness is a gift and the trick is not...',
    a: 'Charles Dickens',
    h: '<blockquote>&ldquo;Happiness is a gift and...'
  }
]
```

We have just added one catch statement so far to handle all errors. However, the errors could come in any link of the Promise chain. For example,

1. We may not be able to retrieve the data from the network.
2. The data format may not be in json.
3. Some downstream postprocessing could be corrupted.

2.4 Promise API

There are some static methods in the `Promise` class that are useful to know.

Definition 2.5 (`Promise.all`)

If we want many promises to execute in parallel and wait until all of them are ready, we can use `Promise.all`, which wraps all promises into a single `Promise` object. It takes an iterable (usually an array of promises) and returns a new promise.

```
let promise = Promise.all(iterable);
```

If any of the promises are rejected, the promise returned by `Promise.all` immediately rejects with that error and forgets about all the resolved ones.

Example 2.4 ()

The wrapper is handled with the alert function, which will return $1, 2, 3$, in the order of the elements of the array, regardless of the time it takes to complete.

```
Promise.all([
  new Promise(resolve => setTimeout(() => resolve(1), 3000)), // 1
  new Promise(resolve => setTimeout(() => resolve(2), 2000)), // 2
  new Promise(resolve => setTimeout(() => resolve(3), 1000))  // 3
]).then(alert); // 1,2,3
```

A slightly more practical example is if we want to fetch an array of URLs.

```
let urls = [
  'https://api.github.com/users/iliakan',
  'https://api.github.com/users/remy',
  'https://api.github.com/users/jeresig'
];

// map every url to the promise of the fetch
```

```
let requests = urls.map(url => fetch(url));

// Promise.all waits until all jobs are resolved
Promise.all(requests)
  .then(responses => responses.forEach(
    response => alert(`${response.url}: ${response.status}`)
  ));
```

Notice that `Promise.all` can be wasteful since if we resolve in all Promises except one, all Promises are lost. This is good for all or nothing cases, but sometimes, we just want a simple wrapper. This is where allSettled comes in.

Definition 2.6 (`Promises.allSettled`)

`Promises.allSettled` waits for all promises to settle, regardless of the result and gives an array of form

1. `{status:"fulfilled", value:result}` for successful responses,
2. `{status:"rejected", reason:error}` for errors

Its syntax is:

```
let promise = Promise.allSettled(iterable);
```

Definition 2.7 (`Promise.race`)

`Promise.race` is similar to `Promise.all`, but waits only for the first settled promise and gets its result (or error).

```
let promise = Promise.race(iterable);
```

Definition 2.8 (`Promise.any`)

`Promise.any` is similar to `Promise.race`, but waits only for the first *successful* settled promise and gets its result.

```
let promise = Promise.any(iterable);
```

If all promises are rejected, then the returned promise is rejected with `AggregateError`.

Definition 2.9 (`Promise.resolve`/`Promise.reject`)

The following two class methods are mostly obsolete due to the `async/await` syntax, but we'll cover it here.

1. `Promise.resolve(value)` creates a resolved promise with the result `value`. It is the same as

   ```
   let promise = new Promise(resolve => resolve(value));
   ```

   It is used when a function is expected to return a promise.
2. `Promise.reject(error)` creates a rejected promise with error `error`. It is the same as

   ```
   let promise = new Promise((resolve, reject) => reject(error));
   ```

2.5 Async and Await

We already have the tools and knowledge required to learn about async and await, which is just simpler syntax for handling Promises.

Definition 2.10 (async)

The `async` keyword, which can be put before a function, indicates two things:

1. The function will run asynchronously via the JavaScript's event loop. This means that its simply running on another thread, independent of the main call stack, or more specifically, it is run asynchronously through the web API which is then pushed into the task queue.
2. Its returned object will be a `Promise` object with a result of whatever it returns. So it ensures that the function returns a promise and wraps non-promises in it.

Example 2.5 (Simple Async Function)

The two functions are equivalent.

```
async function f() {
  return 1;
}
f().then(res => console.log(res)) // 1
```

```
async function f() {
  return Promise.resolve(1);
}
f().then(res => console.log(res)) // 1
```

Definition 2.11 (await)

The `await` keyword is used when you want to wait for an asynchronous function to finish before you move on in the call stack, as if you are "keeping" the function in the call stack rather than moving it into the API. It can only be used in async functions and is mainly used for all asynchronous functions (e.g. `fetch .json`, etc.). It simply makes JavaScript wait until that promise settles and returns its result. It has the syntax

```
let value = await promise;
```

`await` suspends the function execution until the promise settles, and then resumes it with the promise result. It doesn't cost any CPU resources, because the JavaScript engine can do other jobs in the meantime.

Note the difference between the two.

1. This binds the `promise` variable to the `Promise` object immediately and does not wait for it to be resolved.

   ```
   let promise = fetch("url");
   ```

2. This binds the `promise` variable to the `Promise` object only after the Promise has been resolved. This way we are guaranteed that it is not in the pending mode.

   ```
   let promise = await fetch("url");
   ```

Now we can build a simple API.

Example 2.6 (Fetching Ethereum Transactions)

An API for `Etherscan.io` exists to get transaction histories for accounts. Let's walk through this code.

```javascript
async function getTransactions() {
  let response = await fetch(transaction_history);
  let data = await response.json();
  return data;
}

// Usage
async function main() {
  try {
    let transactions = await getTransactions();
    console.log(transactions);
  } catch (error) {
    console.error("Error fetching transactions:", error);
  }
}

main();
```

1. We define an async function to retrieve the response and extract json data from it.
2. The `fetch` function returns a Promise that resolves with a Response object. Therefore, we must pause the execution of the function to wait for this line to finish, since network requests are asynchronous and can take time to complete. We don't want to proceed until we have a response.
3. The conversion into JSON is also asynchronous and returns a new Promise. It resolves with the result of parsing the response body text as JSON. We need to `await` this operation because parsing JSON can take time, especially for large responses.
4. The `getTransactions()` function itself is declared `async`, which means that when we call this function inside `main()`, this gets moved into the web API, where it is then resolved and pushed into the task queue. If we hadn't used `async`, then by the time `getTransactions` is moved to the web API, the next line `console.log(transactions)` will execute, causing a pending status. Obviously, we want to pause the execution of `main()` until `getTransactions()` is finished, so we must call `await`, as if we are "keeping" it inside the call stack.
5. Since we can't just call `await getTransactions()` in the global scope, we just make another `async main()` wrapper function as a temporary solution for now.

2.6 API Handling

We can make HTTPS requests by using the `fetch` function, where the argument could be a URL or a requests object, and it always returns a `Promise` object.