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.

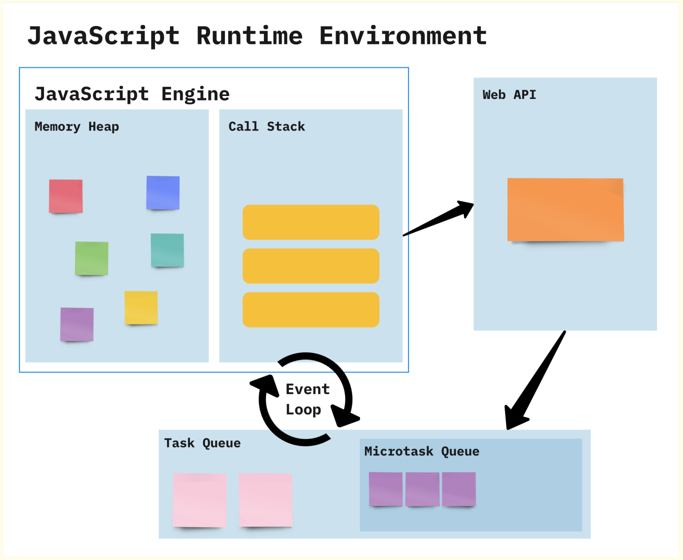

Figure 1: The JS runtime environment contains task queues, the Web API, and the event handler.

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 a method called `then`, which takes two arguments:

1. arg 1: the function that will be called with `PromiseResult` when the Promise is resolved.
2. arg 2: the function that will be called with `PromiseResult` when the Promise is rejected.

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)
);
```

2.4 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.