JavaScript Event Loop

Summary: in this tutorial, you'll learn about the event loop in JavaScript and how JavaScript achieves the concurrency model based on the event loop.

JavaScript single-threaded model

JavaScript is a single-threaded programming language. This means that JavaScript can do only one thing at a single point in time.

The JavaScript engine executes a script from the top of the file and works its way down. It creates the execution contexts and pushes and pops functions onto and off the call stack in the execution phase.

If a function takes a long time to execute, you cannot interact with the web browser during the function's execution because the page hangs.

A function that takes a long time to complete is called a *blocking function*. Technically, a blocking function blocks all the interactions on the webpage, such as mouse clicks.

An example of a blocking function is a function that calls an API from a remote server. The following example uses a big loop to simulate a blocking function:

```
function task(message) {
    // emulate time consuming task
    let n = 10000000000;
    while (n > 0){
        n--;
    }
    console.log(message);
}

console.log('Start script...');
```

```
task('Call an API');
console.log('Done!');
```

In this example, we have a big `while` loop inside the `task()` function that emulates a time-consuming task. The `task()` function is a blocking function.

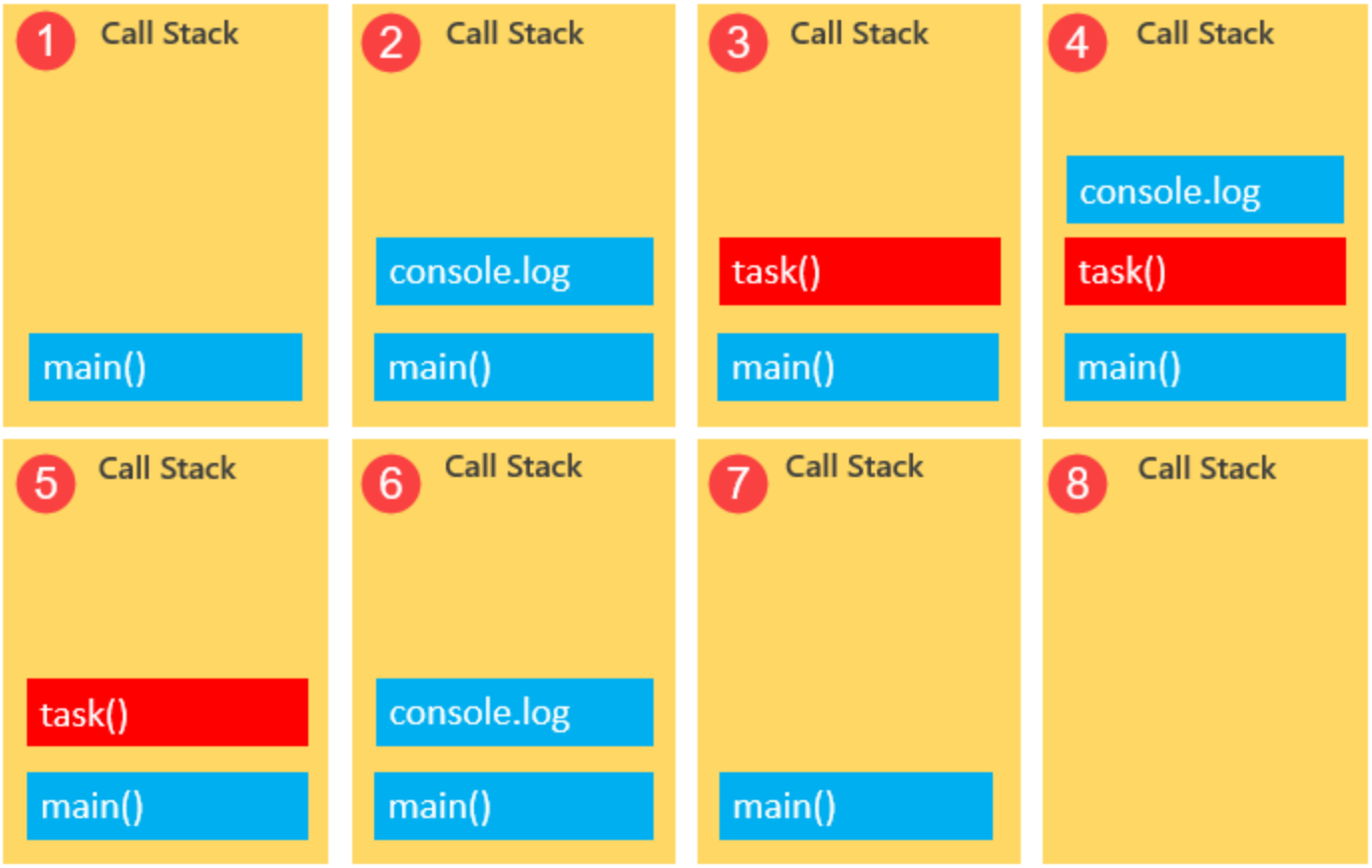
The script hangs for a few seconds (depending on how fast the computer is) and issues the following output:

```
Start script...
Download a file.
Done!
```

To execute the script, the JavaScript engine places the first call `console.log()` on top of the call stack and executes it. Then, it places the `task()` function on top of the call stack and executes the function.

However, it'll take a while to complete the `task()` function. Therefore, you'll see the message `'Download a file.'` a little time later. After the `task()` function completes, the JavaScript engine pops it off the call stack.

Finally, the JavaScript engine places the last call to the `console.log('Done!')` function and executes it, which will be very fast.



Callbacks to the rescue

To prevent a blocking function from blocking other activities, you can put it in a callback function for execution later. For example:

```
console.log('Start script...');

setTimeout(() => {
    task('Download a file.');
}, 1000);

console.log('Done!');
```

In this example, you'll see the message `'Start script...'` and `'Done!'` immediately. And after that, you'll see the message `'Download a file'`.

Here's the output:

```
Start script...
Done!
Download a file.
```

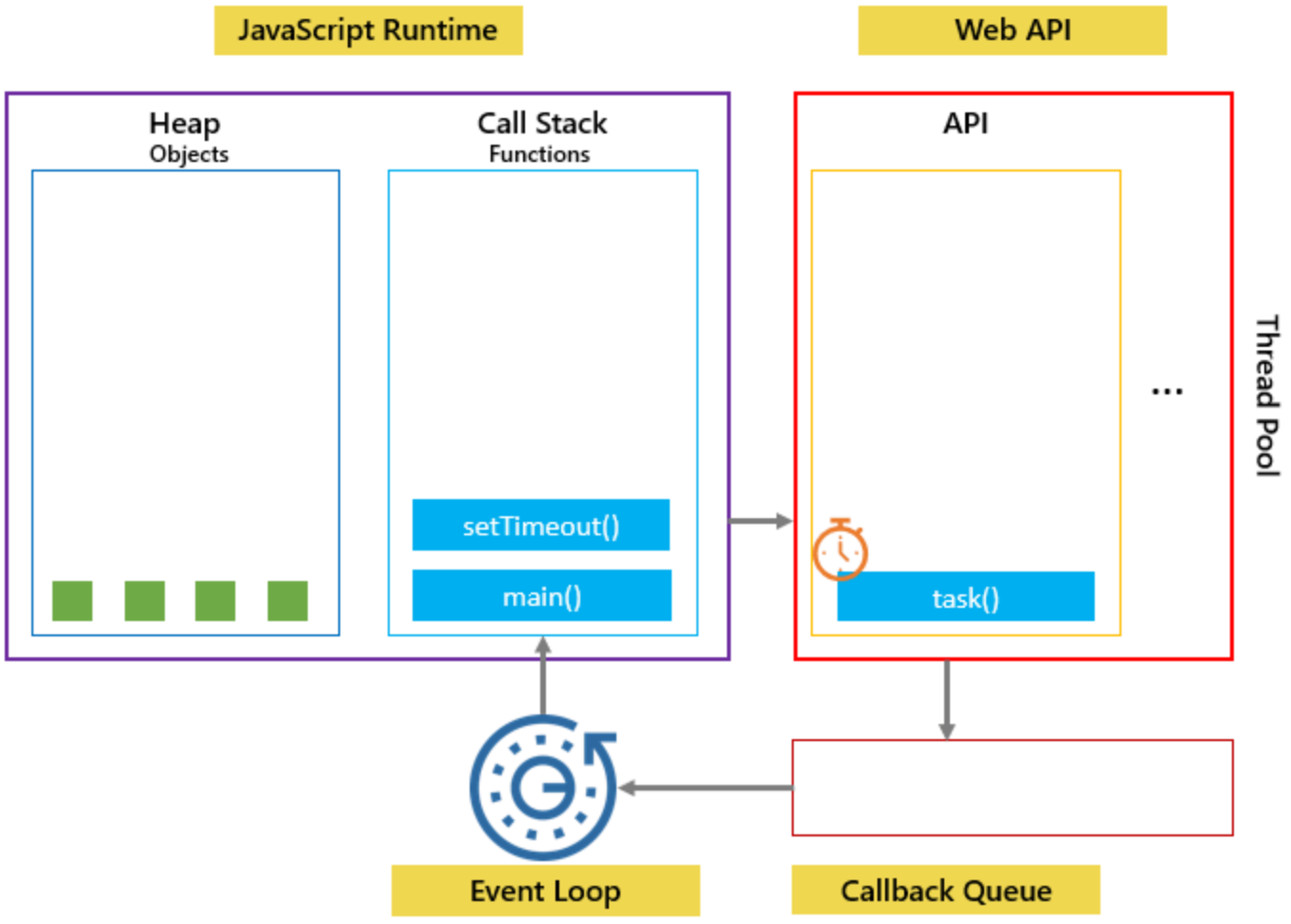
As mentioned earlier, the JavaScript engine can do only one thing at a time. However, it's more precise to say that the JavaScript runtime can do one thing at a time.

The web browser also has other components, not just the JavaScript engine.

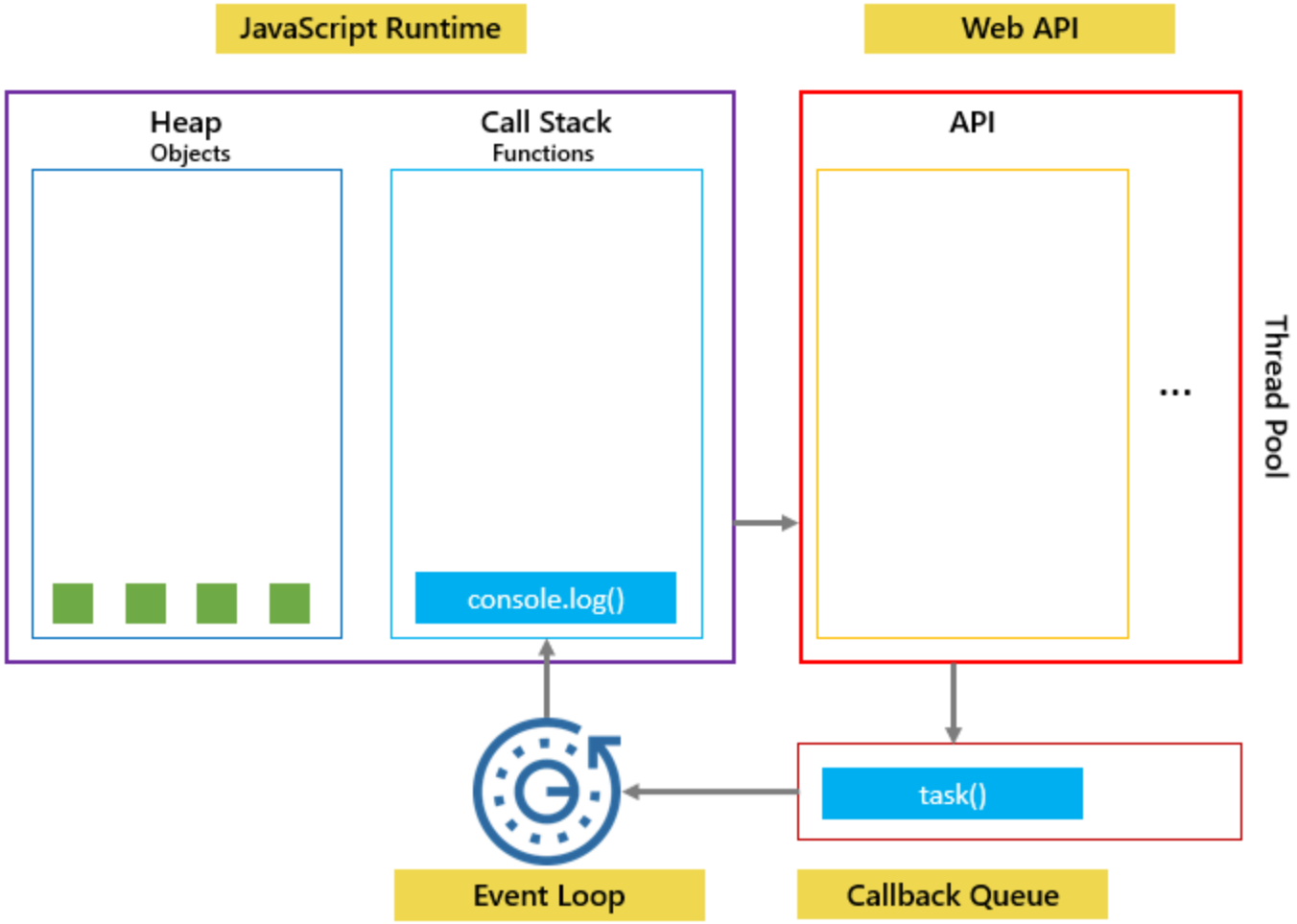
When you call the `setTimeout()` function, make a fetch request, or click a button, the web browser can do these activities concurrently and asynchronously.

The `setTimeout()`, fetch requests and DOM events are parts of the Web APIs of the web browser.

In our example, when calling the `setTimeout()` function, the JavaScript engine places it on the call stack, and the Web API creates a timer that expires in 1 second.

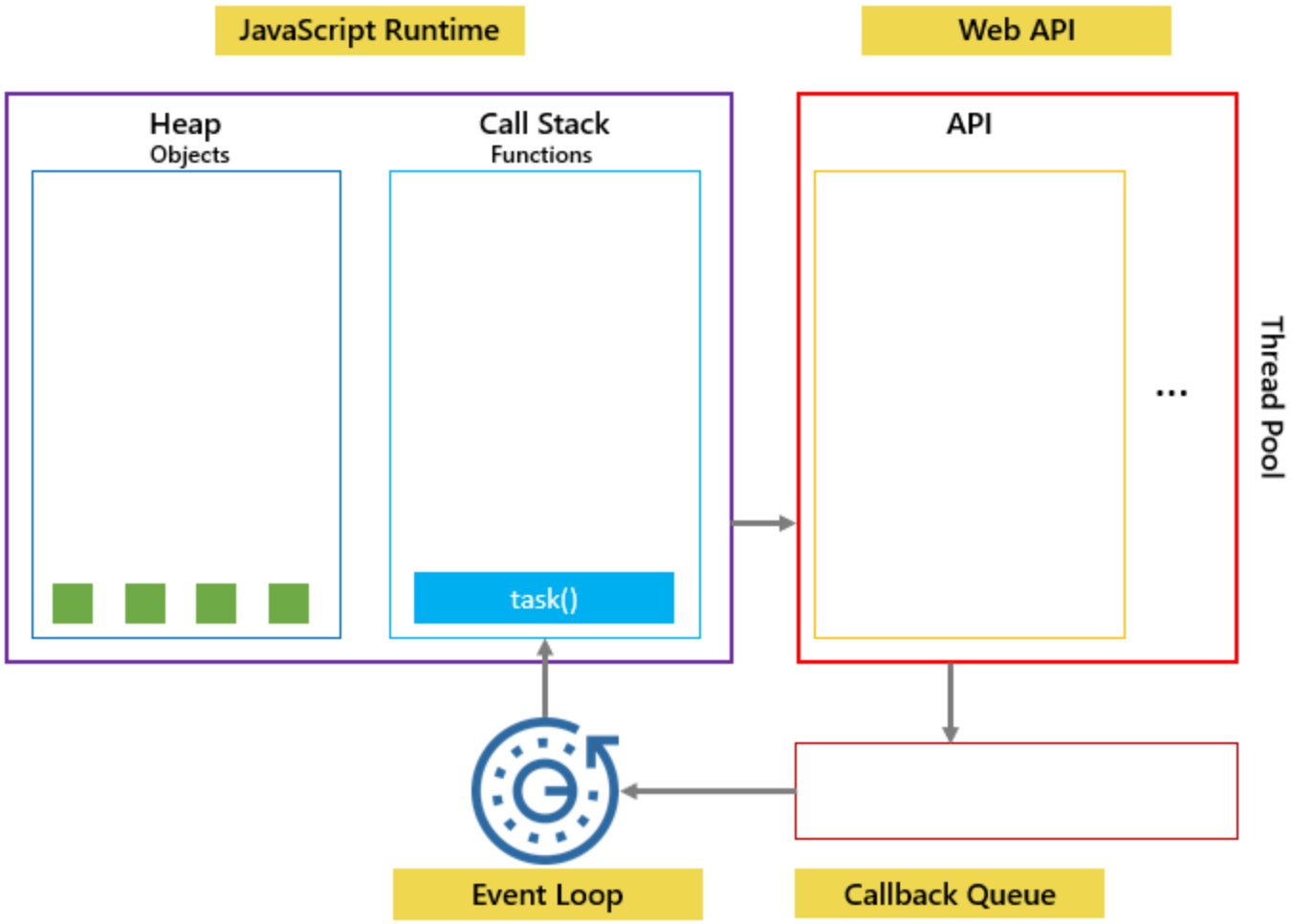


Then JavaScript engine places the `task()` function into a queue called a callback queue or a task queue:



The event loop is a constantly running process that monitors both the callback queue and the call stack.

If the call stack is not empty, the event loop waits until it is empty and places the next function from the callback queue to the call stack. If the callback queue is empty, nothing will happen:



See another example:

```
console.log('Hi!');

setTimeout(() => {
    console.log('Execute immediately.');
}, 0);

console.log('Bye!');
```

In this example, the timeout is 0 seconds, so the message `'Execute immediately.'` should appear before the message `'Bye!'`. However, it doesn't work like that.

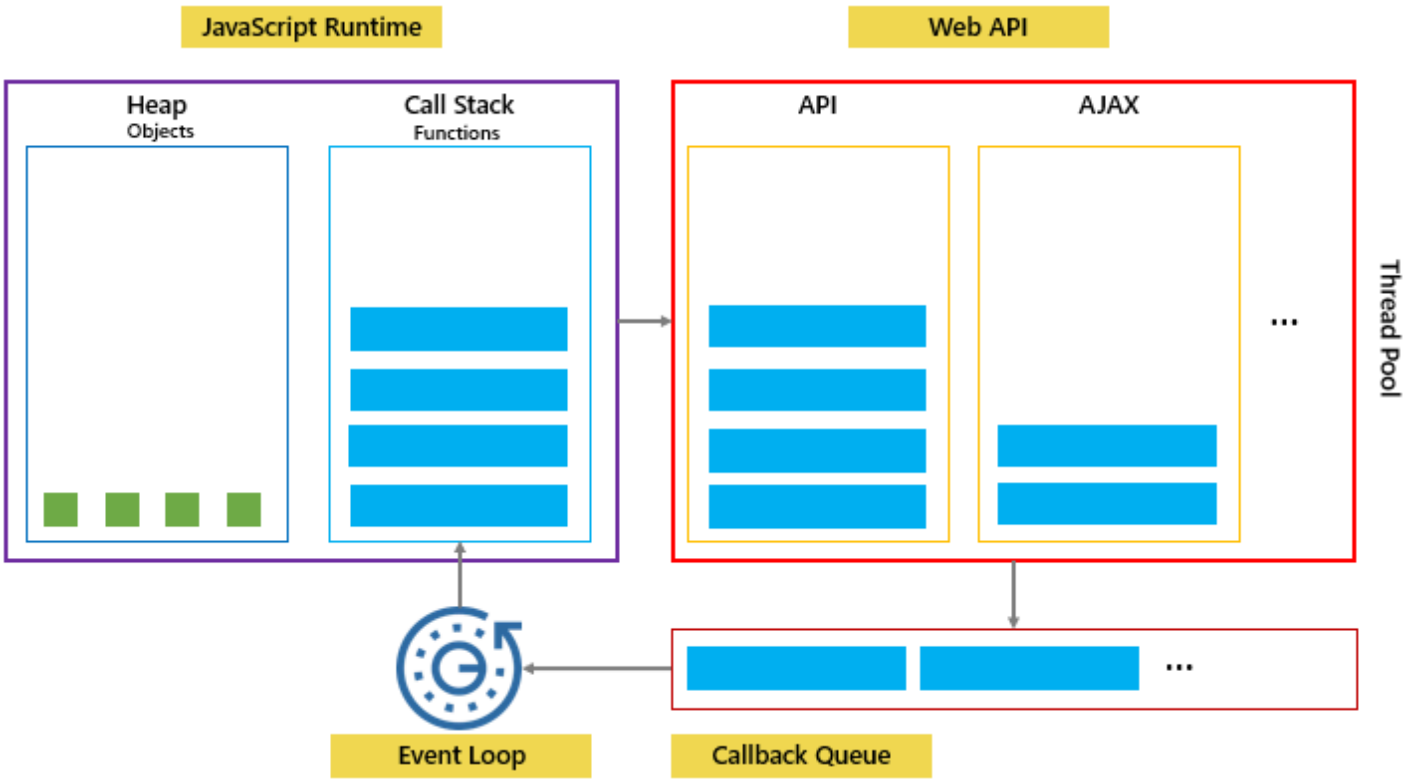
The JavaScript engine places the following function call on the callback queue and executes it when the call stack is empty. In other words, the JavaScript engine executes it after the `console.log('Bye!')`.

```
console.log('Execute immediately.');
```

Here's the output:

```
Hi!
Bye!
Execute immediately.
```

The following picture illustrates JavaScript runtime, Web API, Call stack, and Event loop:



In this tutorial, you have learned about the JavaScript event loop, a constantly running process that coordinates the tasks between the call stack and callback queue to achieve concurrency.