Advanced NodeJS with HTTP

simplilearn

A Day in the Life of a MERN Stack Developer

It is another day in Joe's life as a MERN Stack Developer in an IT company. Joe has completed all his previous tasks and has now been assigned to an important project.

Joe needs to develop an app using advanced NodeJS where users can list down the daily tasks that must be completed. Users should be able to delete a task as and when they complete it. Users must also add new tasks when they get one.

In this lesson, we will learn how to solve this real-world scenario to help Joe complete his task effectively and quickly.

Learning Objectives

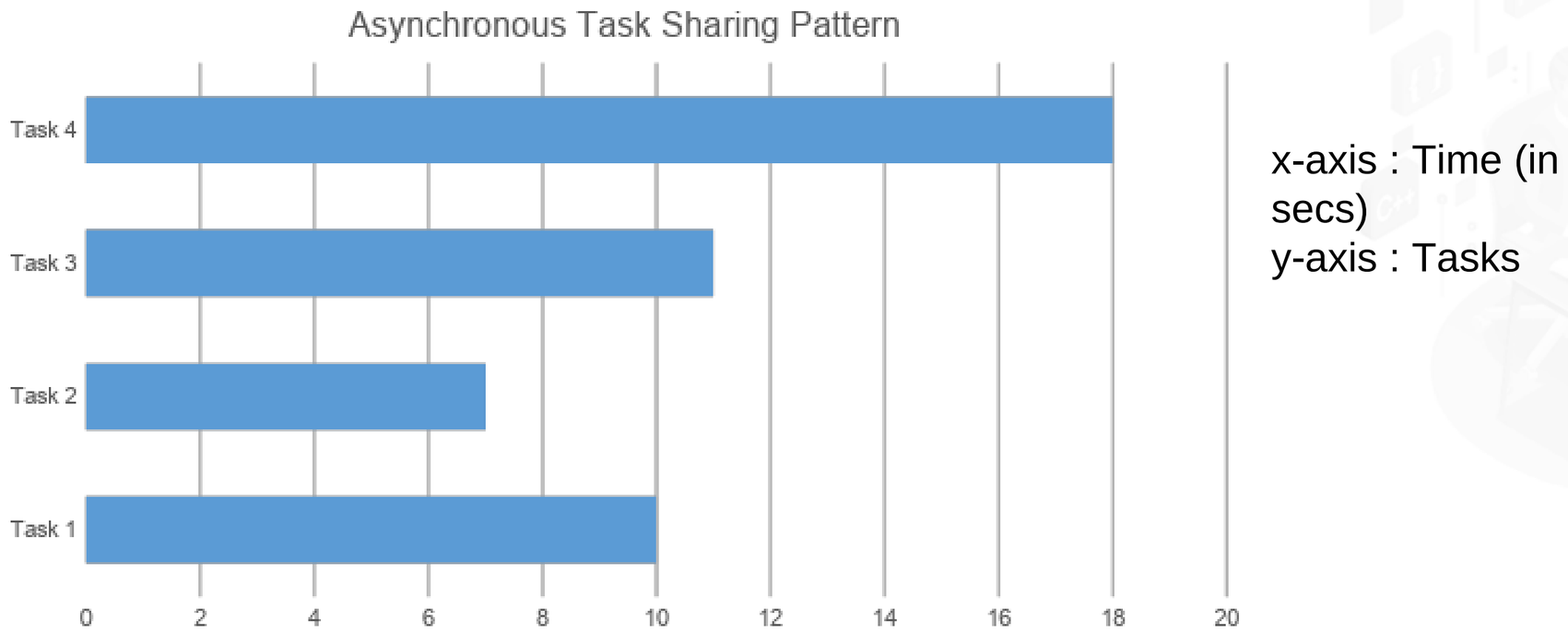
By the end of this lesson, you will be able to:

- Explain call stack, callback queue, and event loop
- Create HTTP requests and handle errors
- Work with GitHub and Heroku
- Explain version control
- Differentiate between synchronous and asynchronous I/O
- Work with child and cluster API

Asynchronous NodeJS

Asynchronous Node.js

- In asynchronous, code gets executed continuously without any dependency and waiting time.
- Asynchronous is a programming pattern which provides the feature of non-blocking code.
- Asynchronous ensures increase in performance, resource utilization, and system throughput.



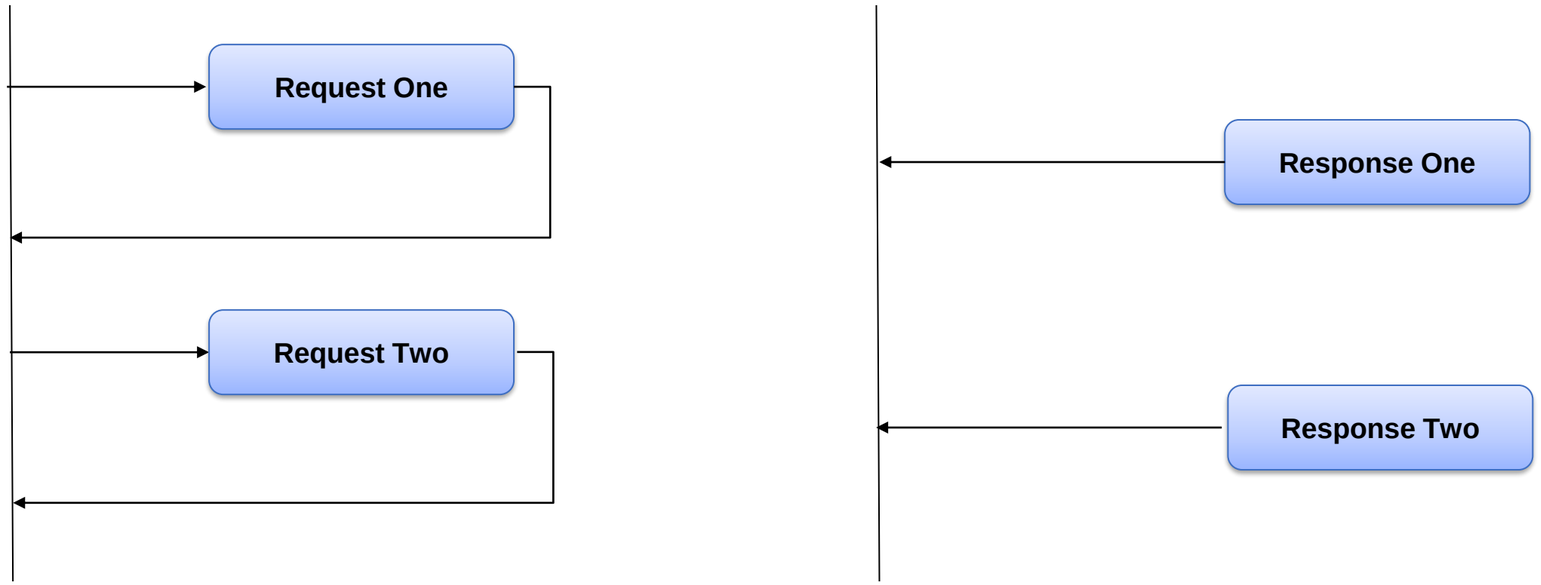
x-axis : Time (in secs)
y-axis : Tasks

Asynchronous Node.js

Real-Life Sit-down Restaurant Scenario:

If two people (P1, P2) go to a sit-down restaurant and P1 ordered food first. P2 need not wait for P1 to receive his food and finish it. P2 can also order food after P1 has ordered and wait to receive his food. There's no interrelation between the way order has been placed and has been served. A quickly prepared food will be served earlier even if the order is placed later.

Similarly, in asynchronous javascript, programming tasks don't depend on other tasks to get executed and the execution of the tasks will depend on the time taken to complete the tasks.



Asynchronous JavaScript Code

script.js

```javascript
function asyncExample()
{
  for(var count=0; count< 10; count++) //loop for counter
    {
    window.setTimeout(function(){    //asynchronous function to prevent the running of code
      console.log(count);            //printing of the count value in the console
      }, 200);                       //200 is the timeout in ms
    }
}
asyncExample();                      //calling of the function
```

Output:

```
console
10
10
10
10
10
10
10
10
10
10
```

In the above code one might perceive that the output should be 0,1,2...9, but the output is 10 ten times. The reason is that, asynchronous javascript does not wait for 200ms and executes the loop. Hence it prints the last value of count.

Call Stack

It is a stack with LIFO(Last In First Out) structure provided by v8 JavaScript engine.

It keeps track of all the functions running in the program.

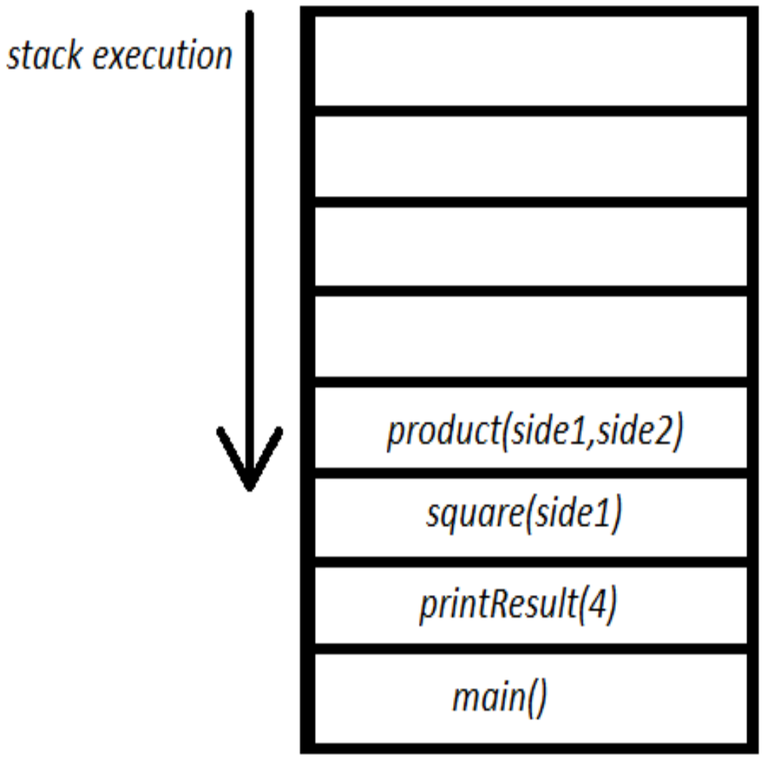
It helps to execute the procedures in an organized manner.

A call stack helps to keep track of the point to which each active subroutine must return control when it finishes its execution.

Call Stack

```
script.js                                                    JavaScript

1  function product(value1,value2)
2  {
3    return value1 * value2;
4  }
5
6  function square(side1)
7  {
8    return product(side1,side1);
9  }
10
11 function printResult(side1)
12 {
13   var result = square(side1);
14   console.log(result);
15 }
16
17 printResult(4);
```



Output:

```
console
16
```

Call Stack

- When printResult() gets executed, an empty stack frame is created. It is the main entry point of the program.
- printResult() then calls square() which is pushed into the stack.
- square() calls product() which is again pushed to the top of the stack.
- product() returns the product of two values to square().
- product() is popped off the stack.
- square() returns the product returned by product() to printResult().
- square() is then popped off the stack.
- printResult() prints the value returned by square().
- printResult() is finally popped off the stack, clearing the memory.

Callback Queue

- A JavaScript runtime contains a callback queue, which is a list of messages and all the callback functions ready to be executed.

For example,

```
setTimeout( () => {
        console.log("0 second!");
 },0);
```

- When the zero timer is completed, it's callback function will get added to the callback queue.

Callback Queue

For example:

```
const bar = () => console.log('bar');
const baz = () => console.log('baz');
const foo = () => {
        console.log('foo');
        setTimeout(bar,0);
        baz();
}
foo();
```


Output:

foo
baz
bar

Callback Queue

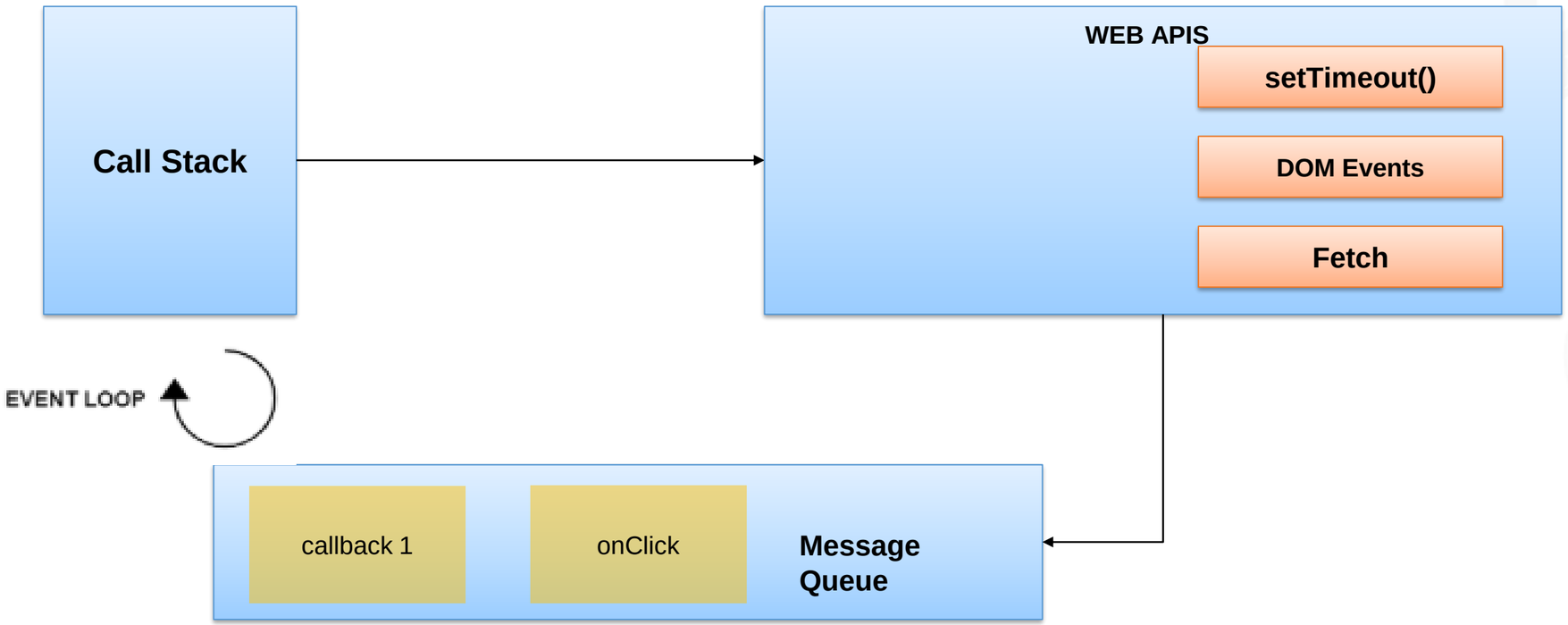
- In the above example, the timer is started when setTimeout() is called.
- Once the timer expires, the callback function is put in the callback queue.
- The queue is where user initiated events like click or keyboard events are queued before they are executed.
- Call stack is given priority and everything in the call stack is first executed.
- After everything in the call stack are executed, callback queue is given priority.

Event Loop

Event loop does a simple job of monitoring the call stack and callback queue.

If the call stack is empty, event loop will take the first event from the queue and push it to the stack.

The event loop is like a task scheduler, which decides what should be executed now and what should be executed later.



Event Loop

Let’s execute a small program and see how it works internally:

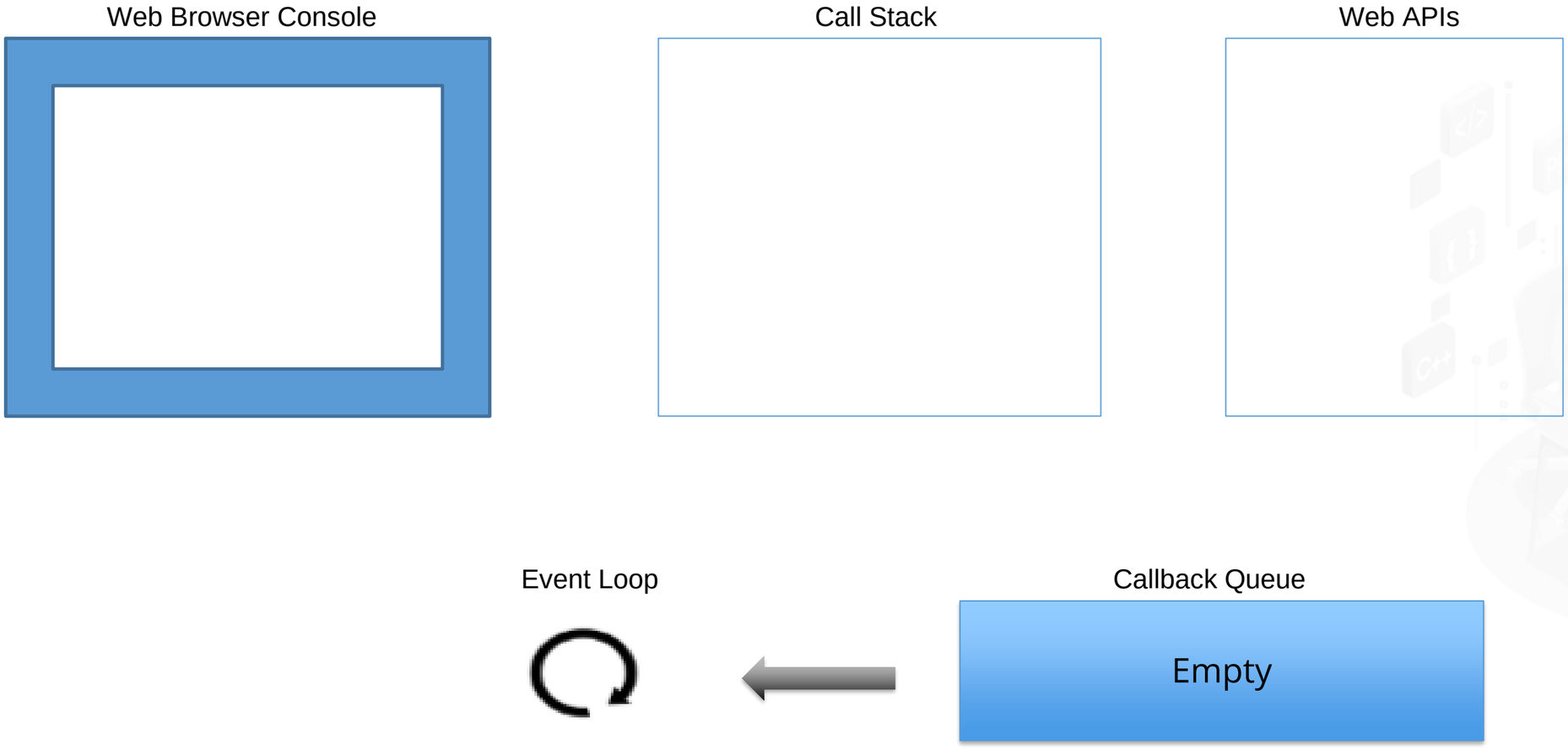
For example:

```
console.log('Start Point’);
setTimeout(function cb( )
{
console.log(‘Now cb got executed');
}
, 5000);
console.log('End Point’);
```


Output:
‘Start Point'
'End Point'
'Now cb got executed’

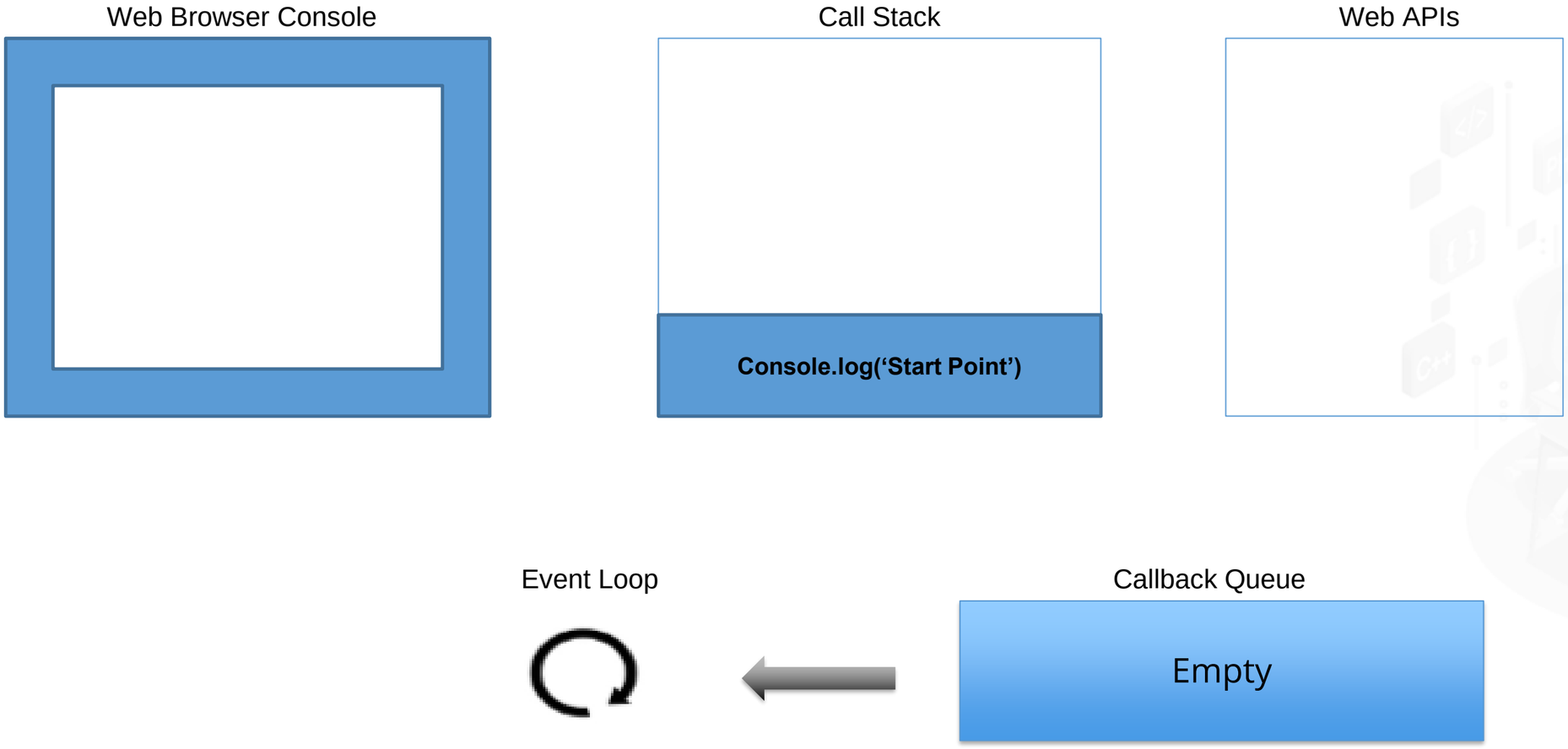
Event Loop

Step 1: When the state is clear, the call stack he and tcallback queue are empty.



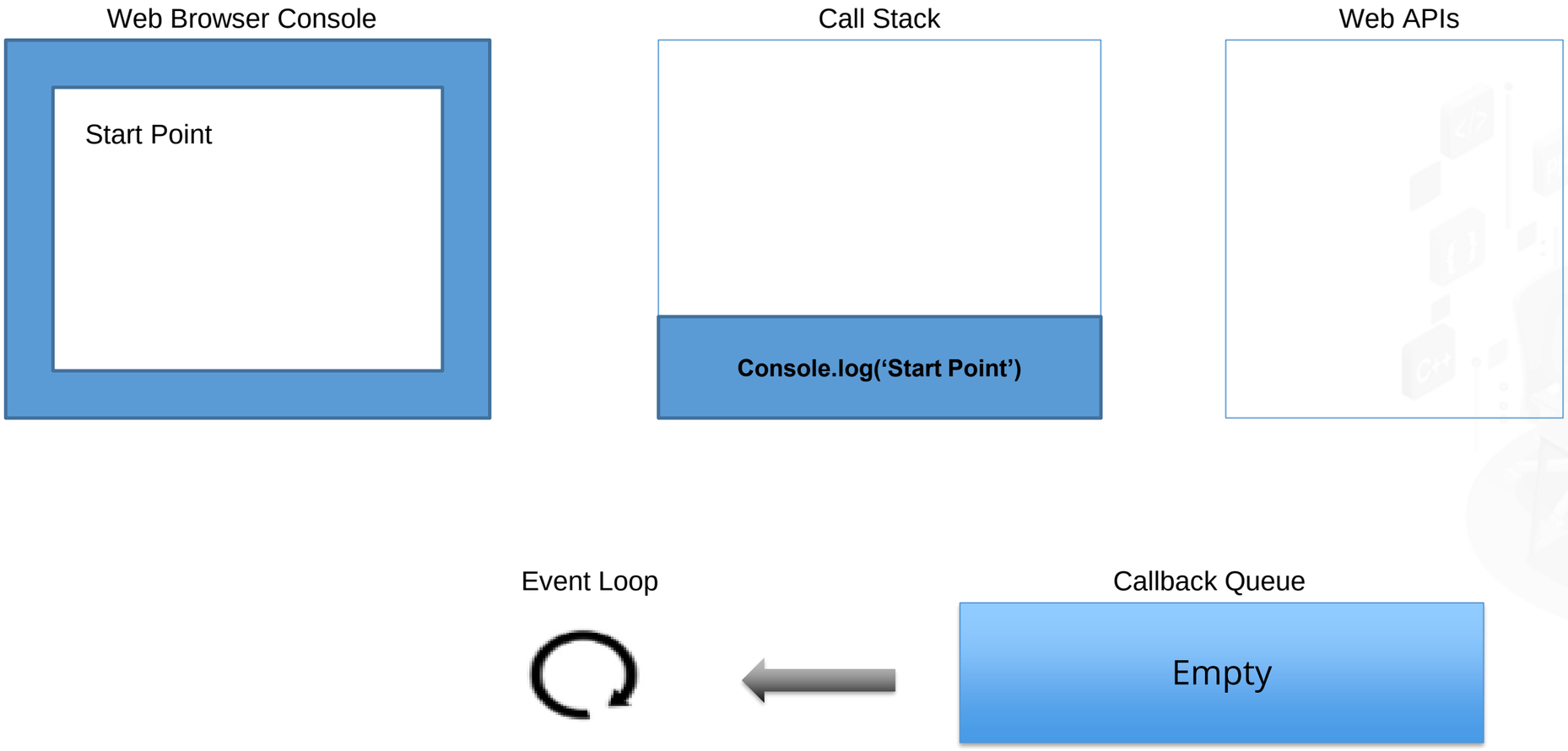
Event Loop

Step 2: console.log (‘Start Point’) is added to the Call Stack.



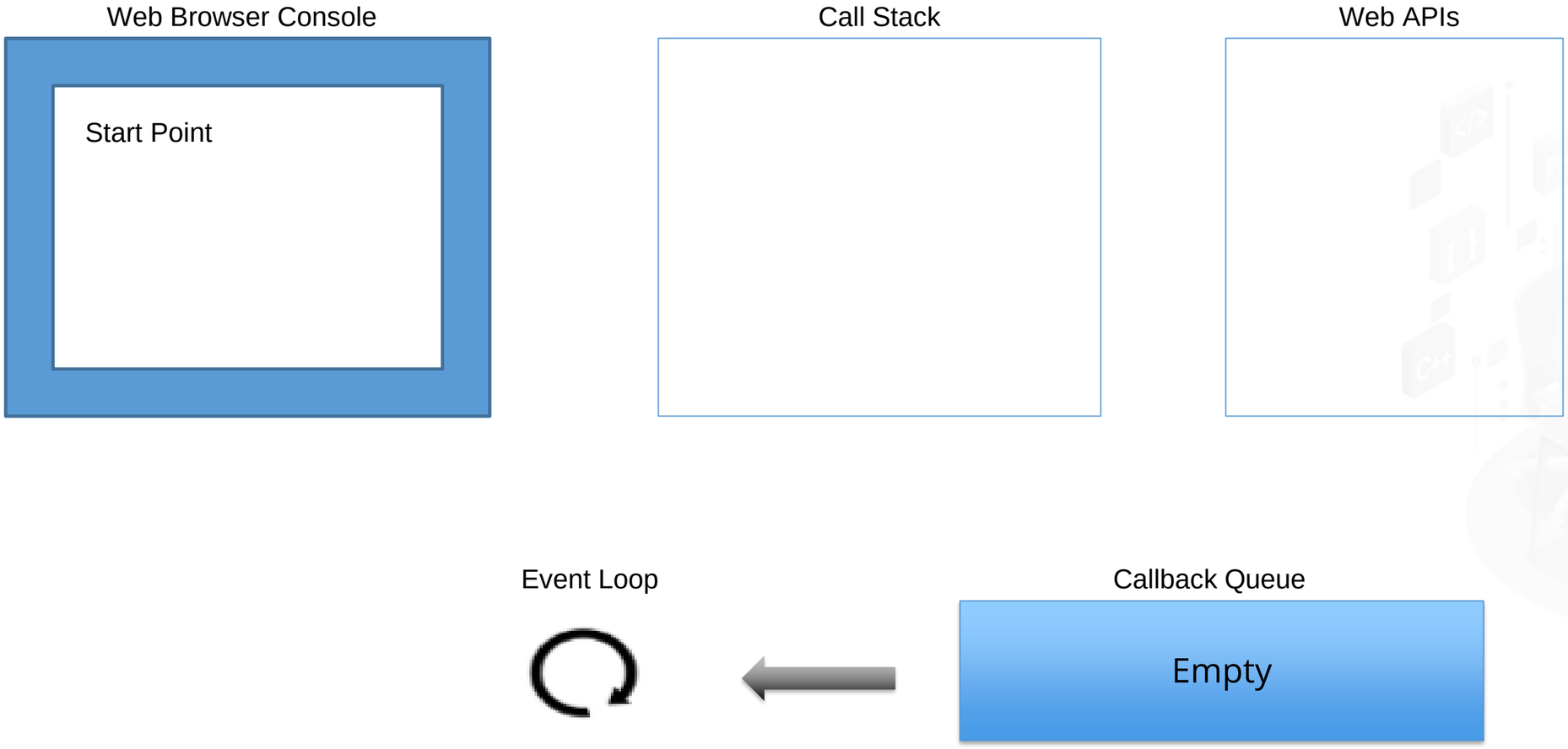
Event Loop

Step 3: console.log (‘Start Point’) is executed.



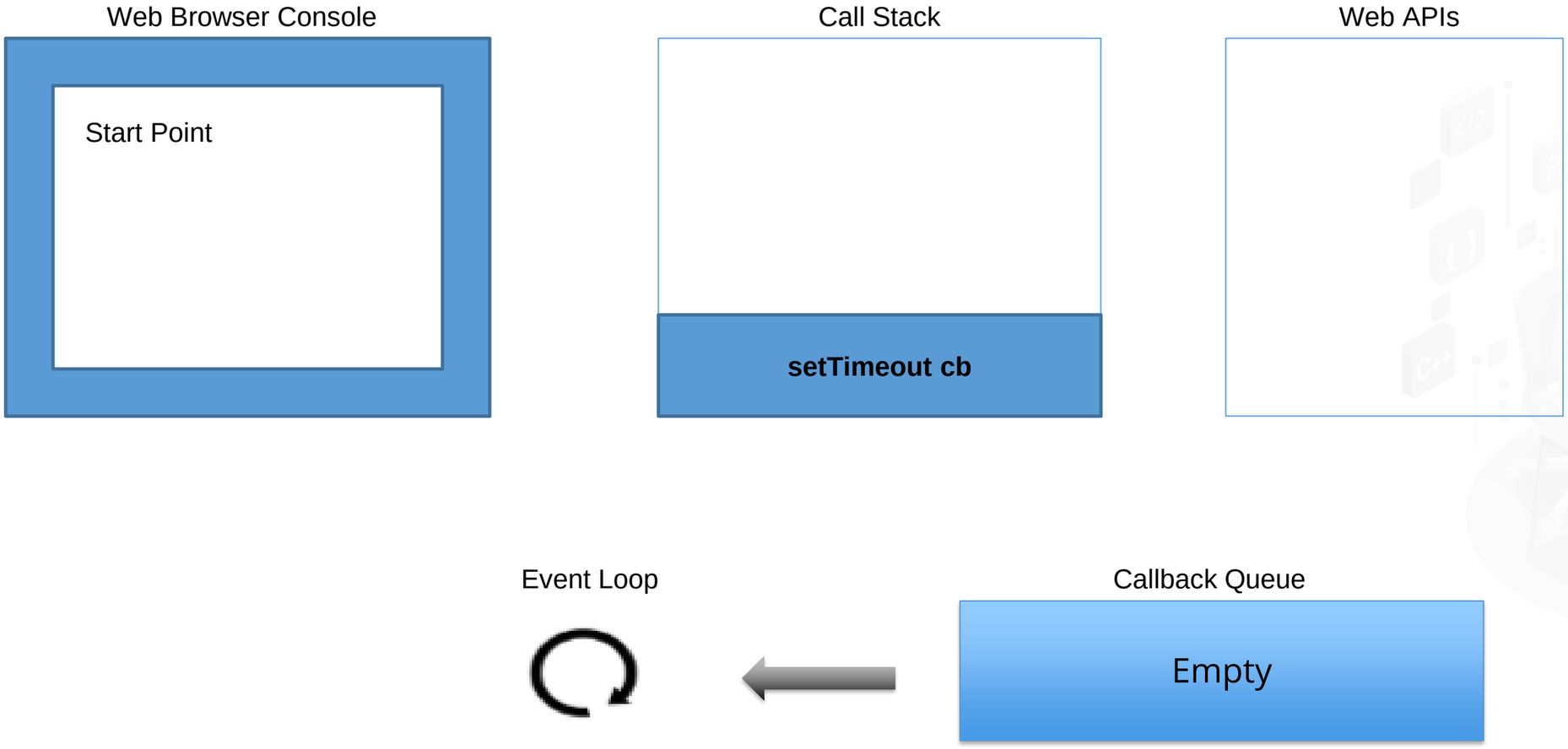
Event Loop

Step 4: console.log (‘Start Point’) is removed from the stack.



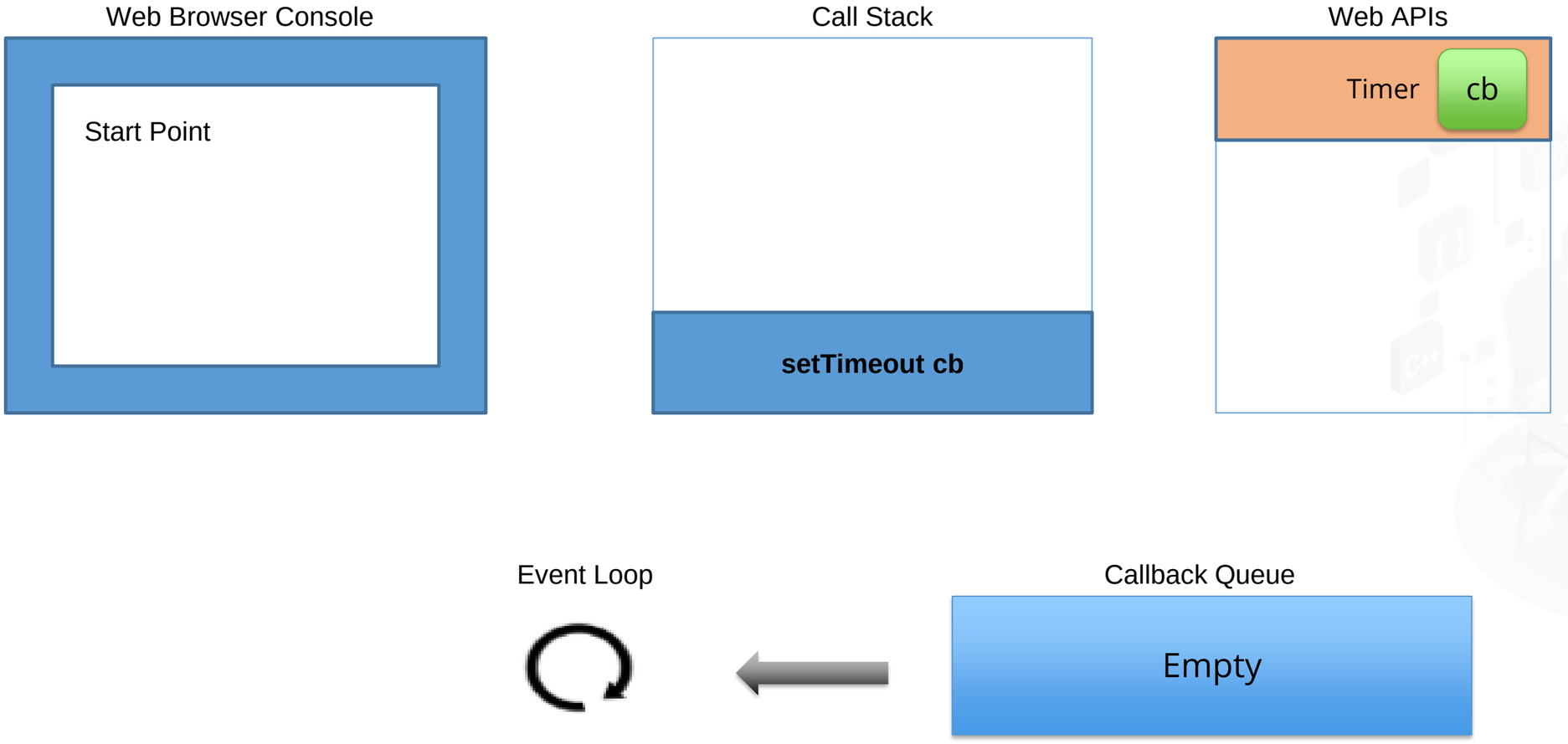
Event Loop

Step 5: setTimeout(function cb ( ){console.log (‘Now cb got executed’); }, 5000) is added to the Call Stack.



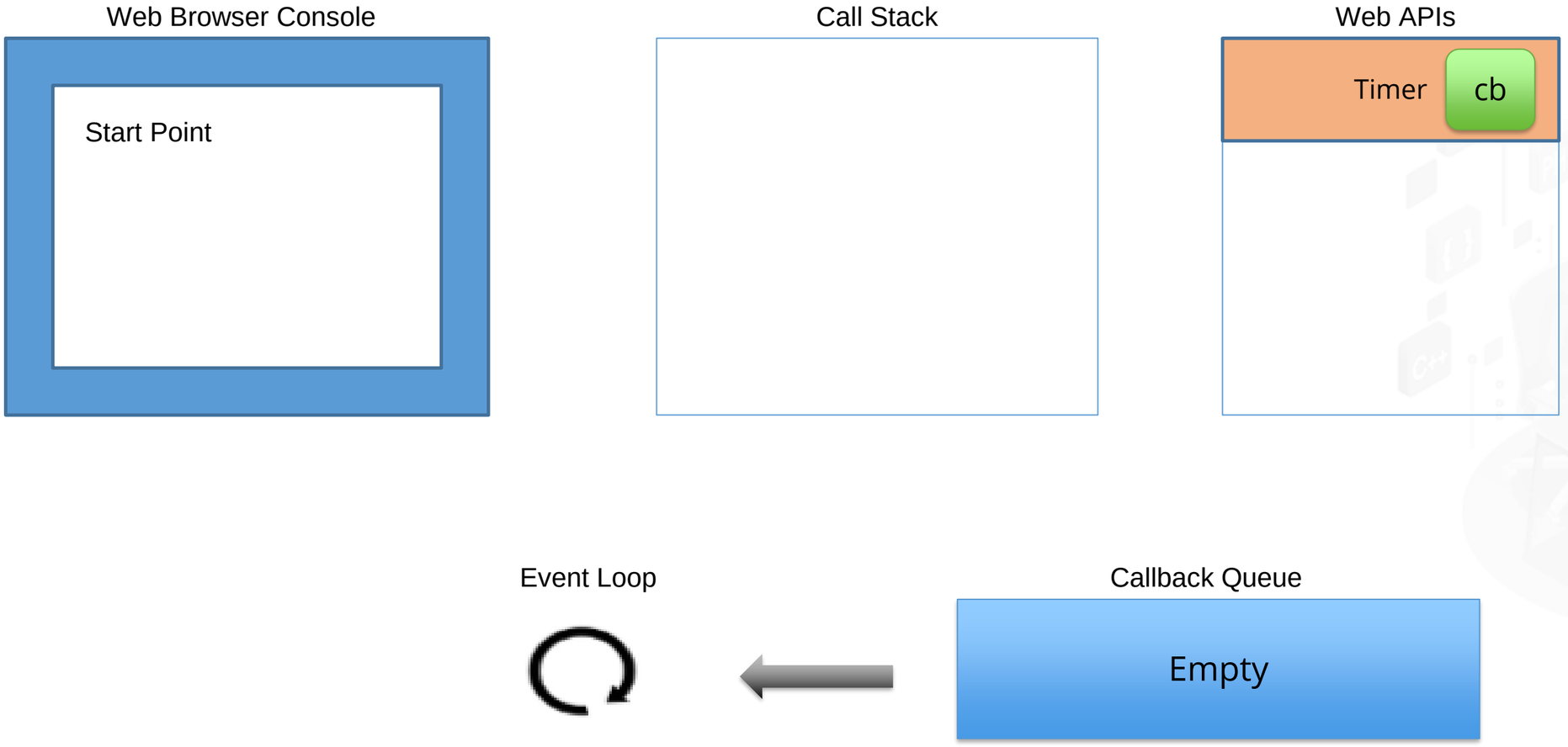
Event Loop

Step 6: setTimeout(function cb ( ){console.log (‘Now cb got executed’); }, 5000) got executed. Now the browser creates a timer as part of the Web API. It is going to handle the counter on its own.



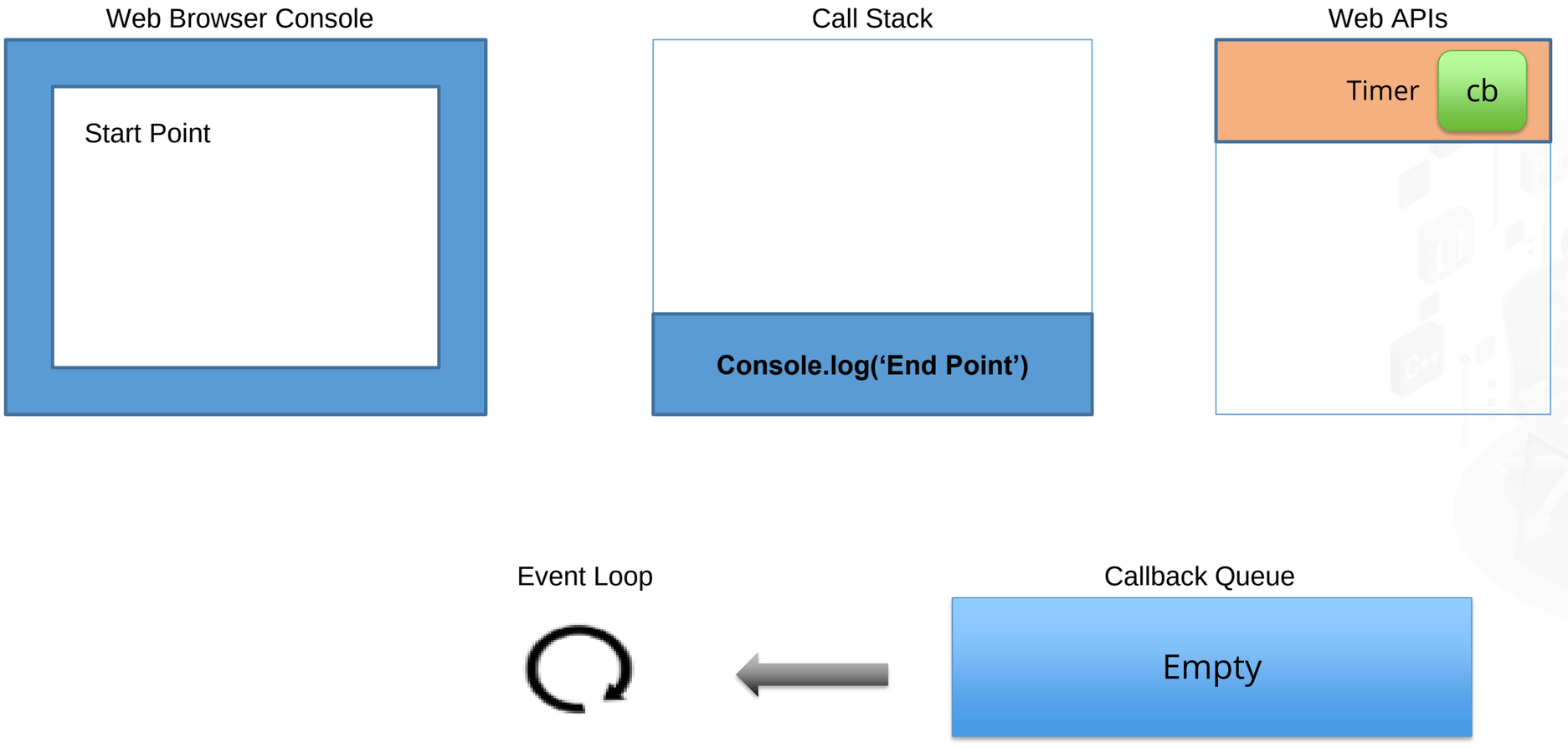
Event Loop

Step 7: The setTimeout(function cb ( ){console.log (‘Now cb got executed’); }, 5000) has completed its execution and is removed from the Call Stack.



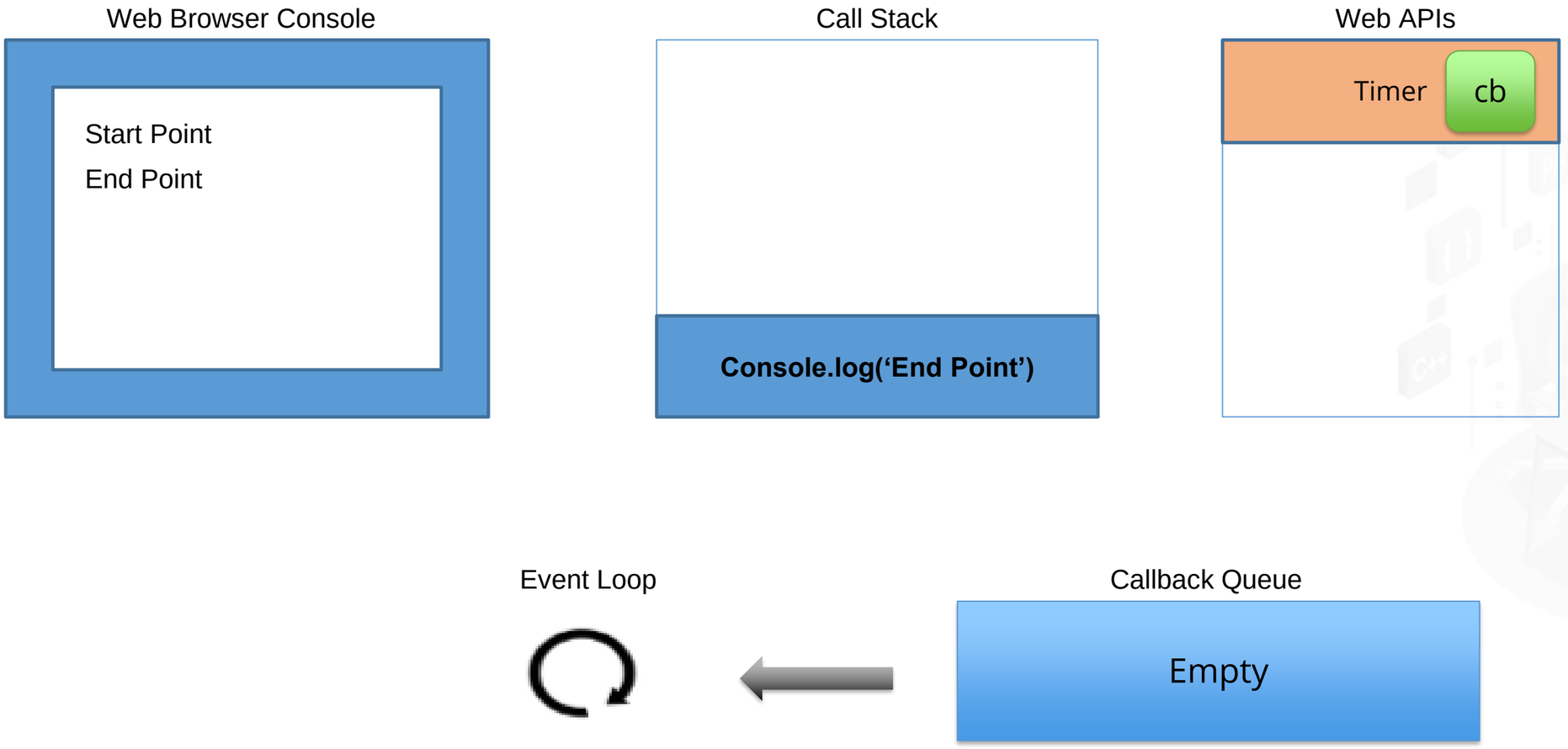
Event Loop

Step 8: console.log('End Point') is added to the Call Stack.



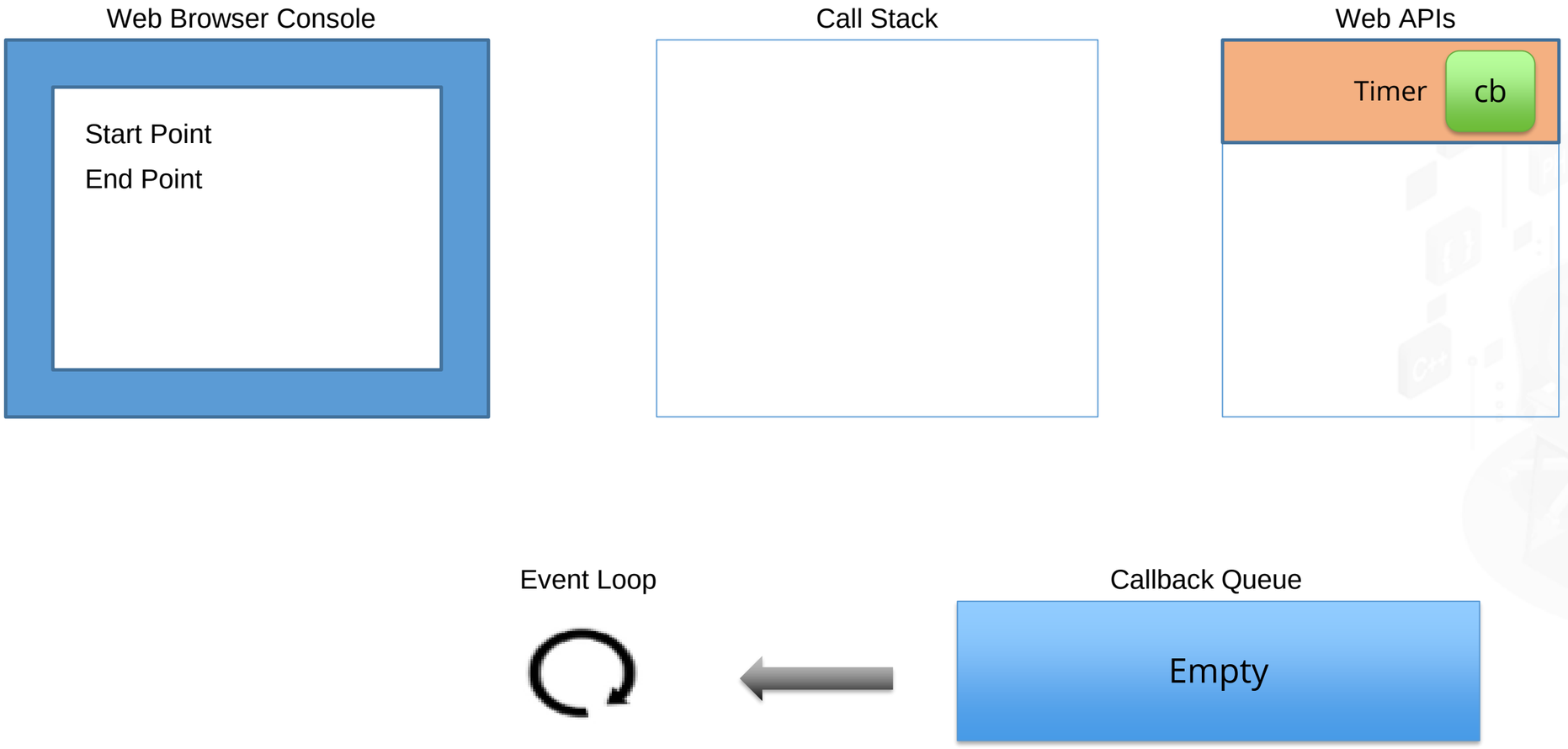
Event Loop

Step 9: console.log('End Point') is executed.



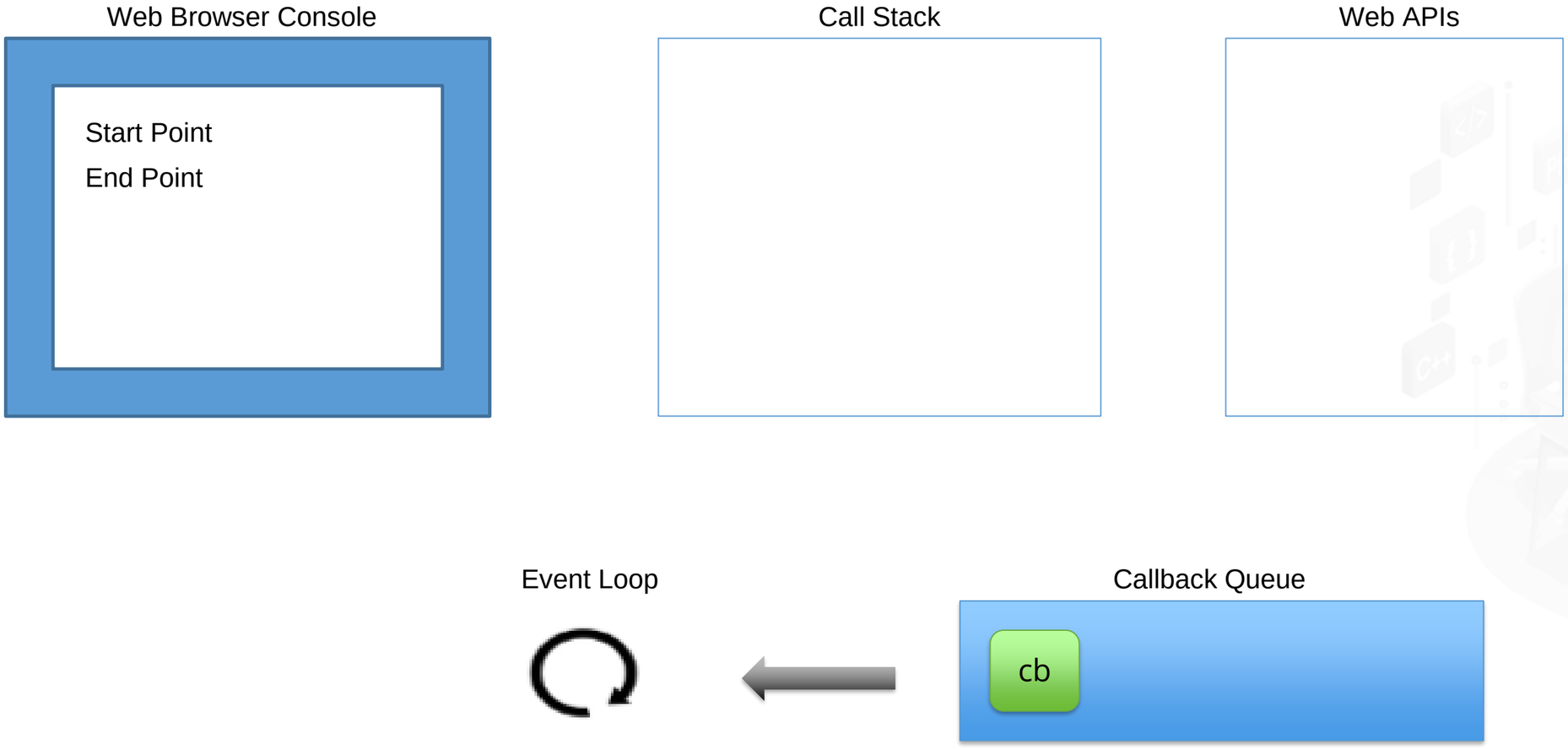
Event Loop

Step 10: console.log('End Point') is removed from the Call Stack.



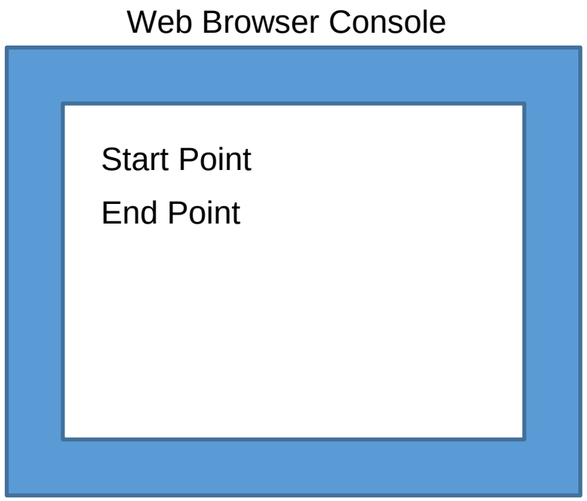
Event Loop

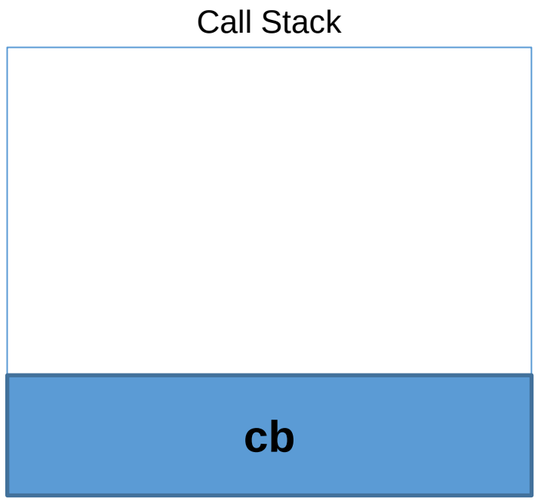
Step 11: After 5000ms, the timer completes and pushes the callback (cb) to the Callback Queue.



Event Loop

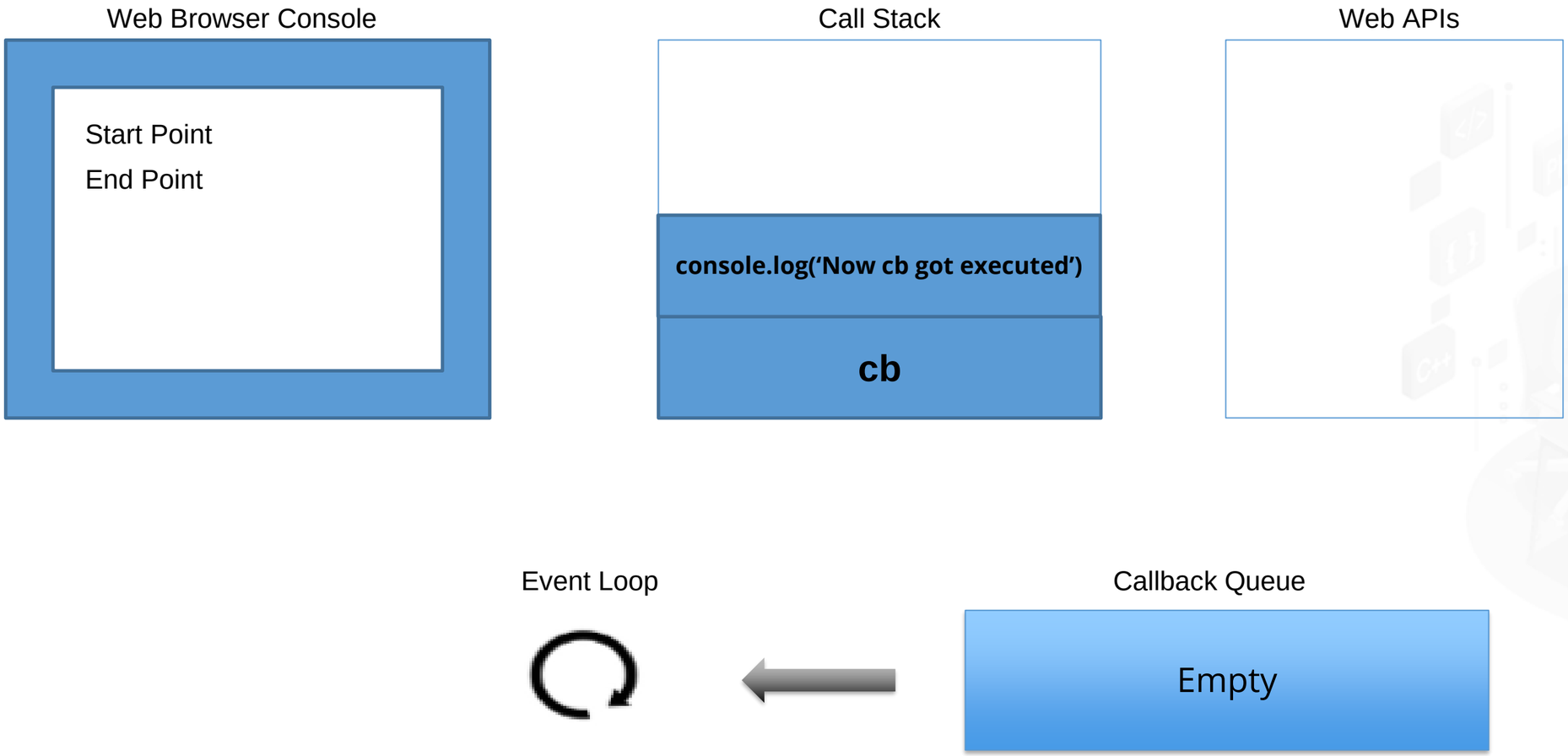
Step 12: The Event Loop takes cb from Callback Queue and pushes it to the Call Stack.





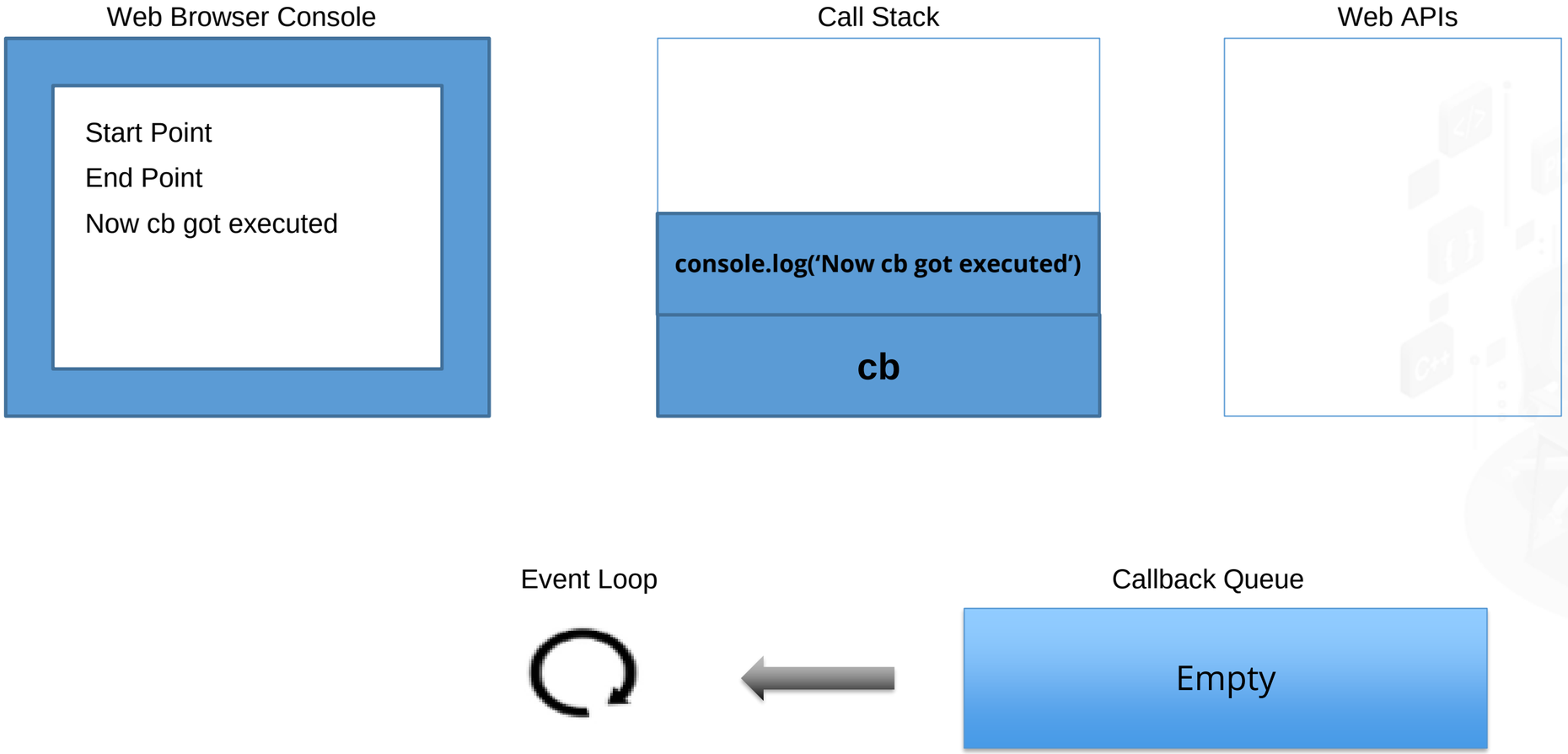
Event Loop

Step 13: cb is executed and adds console.log('Now cb got executed') to the Call Stack.



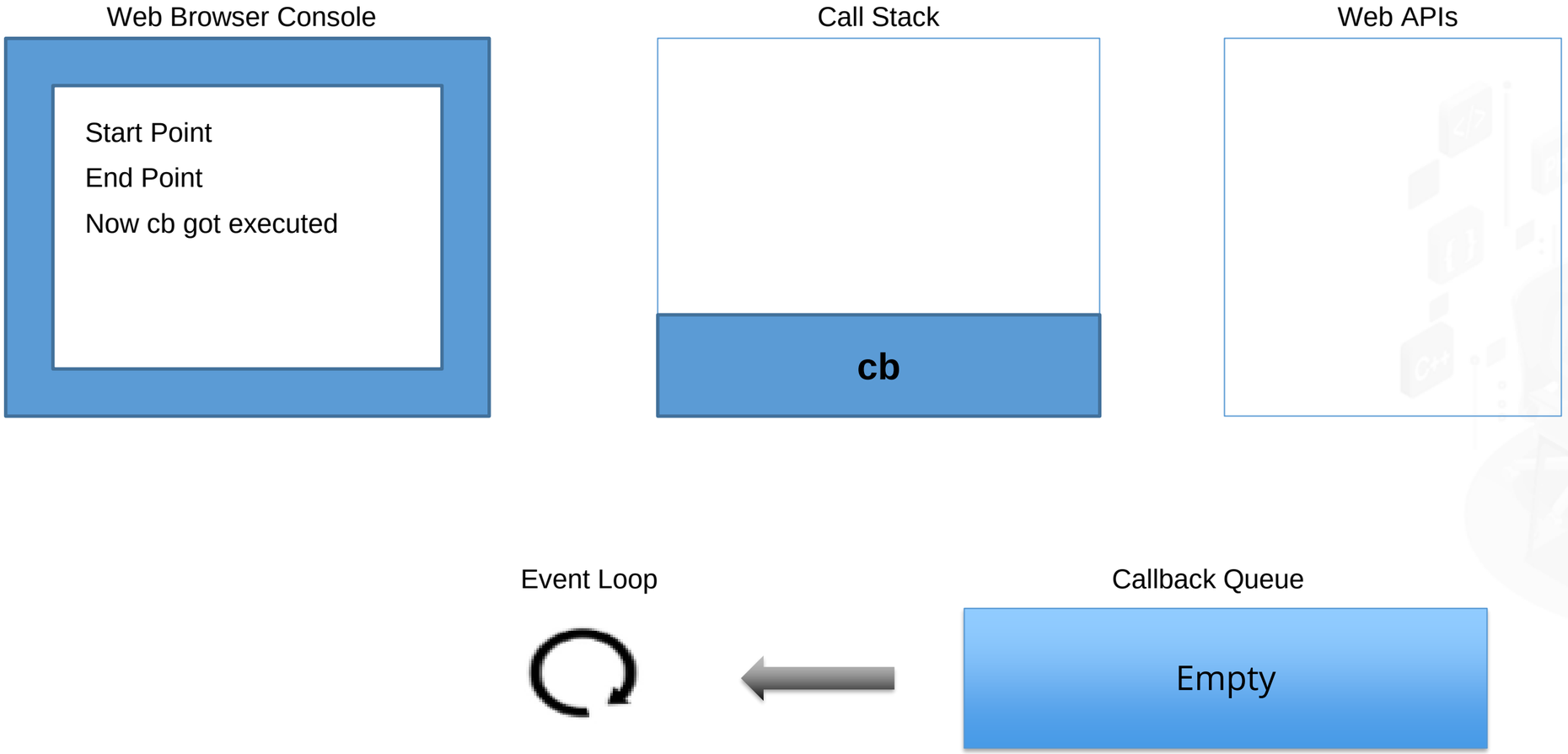
Event Loop

Step 14: console.log('Now cb got executed') is executed.



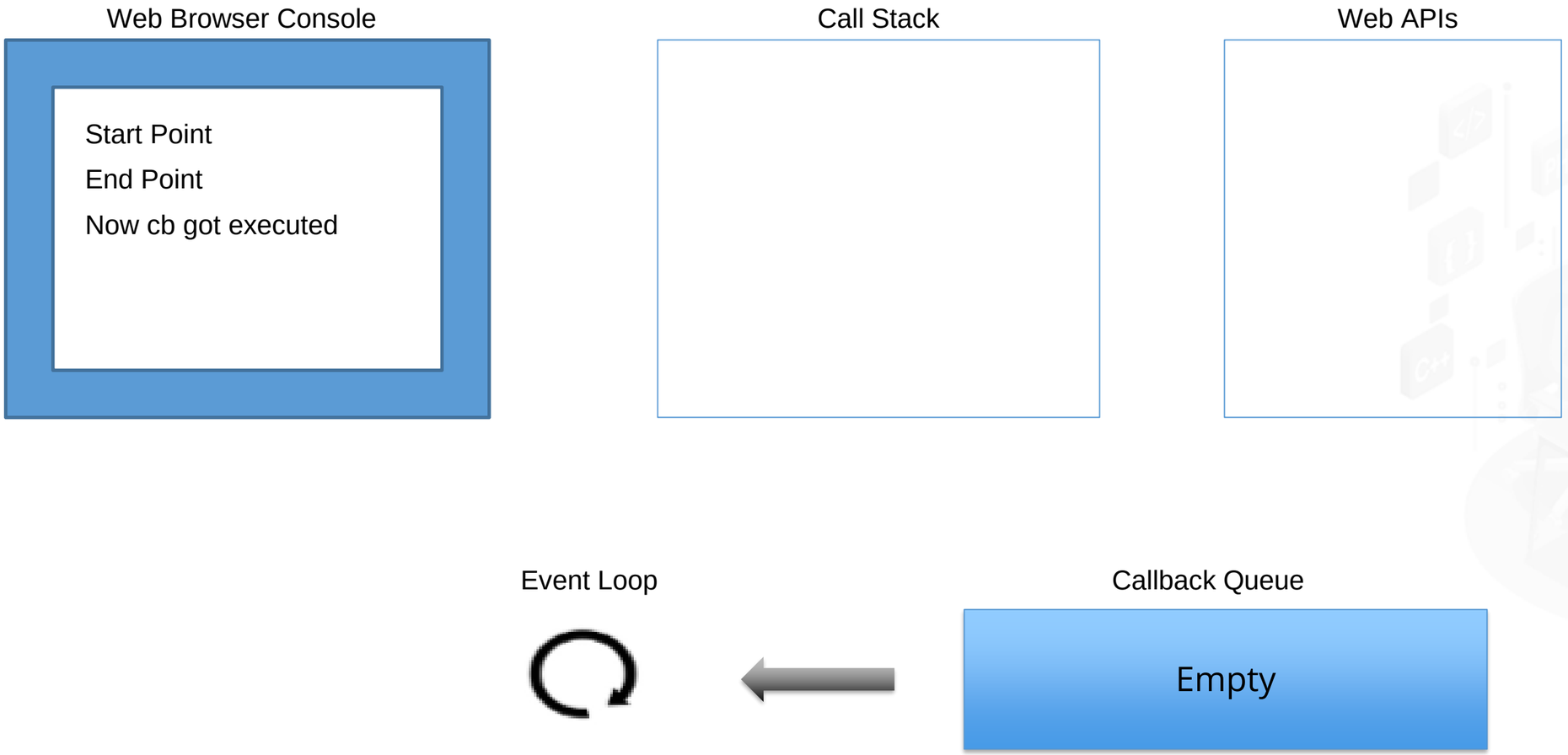
Event Loop

Step 15: console.log('Now cb got executed') is removed from the Call Stack.



Event Loop

Step 16: cb is removed from the Call Stack.



This is how internal process executes the code.

Asynchronous NodeJS

Duration: 75 min.

Problem Statement:

You are given a project to demonstrate how NodeJS runs asynchronous programs.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to demonstrate asynchronous programming:

1. Create a NodeJS project in your IDE

2. Write an asynchronous program in NodeJS to get the weather details of a city

3. Initialize the .git file

4. Add and commit the program files

5. Push the code to your GitHub repository

NodeJS: Util Module

Introduction to Modules

- Modules in NodeJS are same as the libraries in JavaScript.
- Module is a set of functions that you want to include in your application.
- NodeJS has a set of built-in modules which can be used in your application without any installation
- Example: assert, buffer, cluster, fs, http, https, path, stream, url, and util

Util Module: Overview

The util module is designed to support the needs of NodeJs internal APIs. It provides access to the same utility function.

The different util methods are:

Methods	Description
util.callbackify(original)	Accepts an async function and returns a function that follows the error-first callback style
util.debuglog(section)	Writes the debug messages to the error object
util.deprecate(fn, msg[, code])	Marks the specified function as deprecated
util.format(format, [, ...args])	Formats the specified string using the specified arguments

NodeJS: HTTP

Introduction to Server-Side Programming

Server-side programming is the name given to the set of programs that run on the web server.

The operations performed by server-side programs are:
- Processing the user input
- Querying the database
- Reading or writing files on server
- Building web application structure

NodeJS and JavaScript on Server

The components that make **NodeJs** faster on server are:

- V8 engine support
- Non-blocking input-output
- Asynchronous event handling

NodeJS has scalable technology for microservices, rich ecosystem, and seamless JSON support.

JavaScript is the commonly used programming language for web development. It is a client-side programming language.

JavaScript is easy to learn and faster to master.

HTTP Requests

- An application can communicate with the outside world using HTTP requests.
- For example, if you want to get real time weather data in your application, you will need to make an HTTP request.
- You need to specify the URL you want to make a request to. You will fire the request with some data and you will get back the expected response.
- For example, to get weather information, you will send the location for which you want the weather information. In response, you will get back the expected weather information.

HTTP Requests

The seven most popular NodeJS libraries that support HTTP requests:

1. **Using standard HTTP library:**

- There is a default HTTP module in teh NodeJS library.
- You will not need to install any external dependencies.
- However, this is a low-level module and not very user-friendly.
- The following code will send GET request to NASA's API and print out the explanation of the astronomy picture of the day:

Code:

app.js ▸ ...

```
1   const https = require('https');
2   https.get('https://api.nasa.gov/planetary/apod?api_key=DEMO_KEY', (resp) => {
3     let data = '';
4     resp.on('data', (chunk) => {
5       data += chunk;
6     });
7     resp.on('end', () => {
8       console.log(JSON.parse(data).explanation);
9     });
10  }).on("error", (err) => {
11    console.log("Error: " + err.message);
12  });
```

Output:

```
C:\Users\shalini.basu\temp1>node app.js
You are a spaceship soaring through the universe. So is
 your dog. We all carry with us trillions of microorgan
isms as we go through life. These multitudes of bacteri
a, fungi, and archaea have different DNA than you. Coll
ectively called your microbiome, your shipmates outnumb
er your own cells. Your crew members form communities,
help digest food, engage in battles against intruders,
and sometimes commute on a liquid superhighway from one
 end of your body to the other.  Much of what your micr
obiome does, however, remains unknown. You are the capt
ain, but being nice to your crew may allow you to explo
re more of your local cosmos.
```

HTTP Requests

2. **Using Request library:**
   - Request is a simplified HTTP client.
   - This is more user-friendly than the HTTP module.
   - You can install it as a dependency from the Node Package Manager using the following command:
     npm install request --save

Code:

app.js ▸ request('https://api.nasa.gov/planetary/apod?api_key=DEMO_KEY') callback

```
1  const request = require('request');
2
3  request('https://api.nasa.gov/planetary/apod?api_key=DEMO_KEY',
4      { json: true }, (err, res, body) => {
5      if (err) { return console.log(err); }
6      console.log(body.url);
7      console.log(body.explanation);
8  });
```

Output:

```
C:\Users\shalini.basu\temp1>node app.js
You are a spaceship soaring through the universe. So is
 your dog. We all carry with us trillions of microorgan
isms as we go through life. These multitudes of bacteri
a, fungi, and archaea have different DNA than you. Coll
ectively called your microbiome, your shipmates outnumb
er your own cells. Your crew members form communities,
help digest food, engage in battles against intruders,
and sometimes commute on a liquid superhighway from one
 end of your body to the other.  Much of what your micr
obiome does, however, remains unknown. You are the capt
ain, but being nice to your crew may allow you to explo
re more of your local cosmos.
```

HTTP Requests

3. **Using Axios library:**
   - Axios is a Promise based HTTP client for NodeJS.
   - It automatically transforms the response data to JSON object.
   - You can install it as a dependency from the Node Package Manager using the following command:
     npm install axios

Code:

app.js

```
1   const axios = require('axios');
2
3   axios.get('https://api.nasa.gov/planetary/apod?api_key=DEMO_KEY')
4     .then(response => {
5       console.log(response.data.url);
6       console.log(response.data.explanation);
7     })
8     .catch(error => {
9       console.log(error);
10    });
```

Output:

```
C:\Users\shalini.basu\temp1>node app.js
You are a spaceship soaring through the universe. So is
 your dog. We all carry with us trillions of microorgan
isms as we go through life. These multitudes of bacteri
a, fungi, and archaea have different DNA than you. Coll
ectively called your microbiome, your shipmates outnumb
er your own cells. Your crew members form communities,
help digest food, engage in battles against intruders,
and sometimes commute on a liquid superhighway from one
 end of your body to the other.  Much of what your micr
obiome does, however, remains unknown. You are the capt
ain, but being nice to your crew may allow you to explo
re more of your local cosmos.
```

HTTP Requests

4. **Using SuperAgent library:**

- SuperAgent is another HTTP library which is used for making AJAX requests in NodeJS.
- You can chain other useful functions onto requests such as *query()*.
- It is also highly expandable via plugins to perform different tasks like URLs prefixes and suffixes.
- You can install it as a dependency from the Node Package Manager using the following command:
  npm install superagent

Code:

app.js ▸ ...

```
1   const superagent = require('superagent');
2
3   superagent.get('https://api.nasa.gov/planetary/apod')
4   .query({ api_key: 'DEMO_KEY', date: '2017-08-02' })
5   .end((err, res) => {
6     if (err) { return console.log(err); }
7     console.log(res.body.url);
8     console.log(res.body.explanation);
9   });
```

Output:

```
C:\Users\shalini.basu\temp1>node app.js
https://apod.nasa.gov/apod/image/1708/Trunk_Abdul_960
.jpg
Is there a monster in IC 1396?  Known to some as the
Elephant's Trunk Nebula, parts of gas and dust clouds
 of this star formation region may appear to take on
foreboding forms, some nearly human.  The only real m
onster here, however, is a bright young star too far
from Earth to hurt us.  Energetic light from this sta
r is eating away the dust of the dark cometary globul
e near the top of the featured image.  Jets and winds
 of particles emitted from this star are also pushing
 away ambient gas and dust.  Nearly 3,000 light-years
 distant, the relatively faint IC 1396 complex covers
 a much larger region on the sky than shown here, wit
h an apparent width of more than 10 full moons.  APO
D Retrospective: August 2
```

HTTP Requests

5. **Using Got library:**
   - Got is a lightweight and user-friendly HTTP request library.
   - It also supports promises.
   - You can install it as a dependency from the Node Package Manager using the following command:
     npm install got

Code:

```
JS app.js ▸ ...
1   const got = require('got');
2   got('https://api.nasa.gov/planetary/apod?api_key=DEMO_KEY',
3     { json: true }).then(response => {
4     console.log(response.body.url);
5     console.log(response.body.explanation);
6   }).catch(error => {
7     console.log(error.response.body);
8   });
```

Output:

```
C:\Users\shalini.basu\temp1>node app.js
https://apod.nasa.gov/apod/image/1708/Trunk_Abdul_960
.jpg
Is there a monster in IC 1396?  Known to some as the
Elephant's Trunk Nebula, parts of gas and dust clouds
 of this star formation region may appear to take on
foreboding forms, some nearly human.  The only real m
onster here, however, is a bright young star too far
from Earth to hurt us.  Energetic light from this sta
r is eating away the dust of the dark cometary globul
e near the top of the featured image.  Jets and winds
 of particles emitted from this star are also pushing
 away ambient gas and dust.  Nearly 3,000 light-years
 distant, the relatively faint IC 1396 complex covers
 a much larger region on the sky than shown here, wit
h an apparent width of more than 10 full moons.  APO
D Retrospective: August 2
```

HTTP Requests

6. **Using Node-fetch library:**
   - NodeJS is a lightweight HTTP library.
   - It brings browser's Fetch API functionality to NodeJ.
   - You can install it as a dependency from the Node Package Manager using the following command:
     npm install node-fetch

Code:

app.js ▸ then() callback

```
1   const fetch = require('node-fetch');
2
3   fetch('https://api.nasa.gov/planetary/apod?api_key=DEMO_KEY')
4       .then(res => res.json()) // expecting a json response
5       .then(json => {
6           console.log(json.url);
7           console.log(json.explanation);
8       })
9       .catch(err => {
10          console.log(err);
11      });
```

Output:

```
C:\Users\shalini.basu\temp1>node app.js
https://apod.nasa.gov/apod/image/1908/HumanSpaceship2
_TsevisHubbleRJN_960.jpg
You are a spaceship soaring through the universe. So
is your dog. We all carry with us trillions of microo
rganisms as we go through life. These multitudes of b
acteria, fungi, and archaea have different DNA than y
ou. Collectively called your microbiome, your shipmat
es outnumber your own cells. Your crew members form c
ommunities, help digest food, engage in battles again
st intruders, and sometimes commute on a liquid super
highway from one end of your body to the other.  Much
 of what your microbiome does, however, remains unkno
wn. You are the captain, but being nice to your crew
may allow you to explore more of your local cosmos.
```

HTTP Requests

7. **Using Needle library:**
   - NodeJS is a streamable HTTP client for NodeJS.
   - It supports proxy, iconv, cookie, deflate, and multipart requests.
   - It supports promises.
   - You can install it as a dependency from the Node Package Manager using the following command:
     npm install node-fetch

Code:

app.js ▸ ...

```
1   const needle = require('needle');
2   needle.get('https://api.nasa.gov/planetary/apod?api_key=DEMO_KEY',
3      {json: true}, (err, res) => {
4       if (err) {
5            return console.log(err);
6         }
7       console.log(res.body.url);
8       console.log(res.body.explanation);
9   });
```

Output:

```
C:\Users\shalini.basu\temp1>node app.js
https://apod.nasa.gov/apod/image/1908/HumanSpaceship2
_TsevisHubbleRJN_960.jpg
You are a spaceship soaring through the universe. So
is your dog. We all carry with us trillions of microo
rganisms as we go through life. These multitudes of b
acteria, fungi, and archaea have different DNA than y
ou. Collectively called your microbiome, your shipmat
es outnumber your own cells. Your crew members form c
ommunities, help digest food, engage in battles again
st intruders, and sometimes commute on a liquid super
highway from one end of your body to the other.  Much
 of what your microbiome does, however, remains unkno
wn. You are the captain, but being nice to your crew
may allow you to explore more of your local cosmos.
```

Handling Errors

- You need to create an error handling for your HTTP requests to make sure nothing goes wrong.
- One of the most common error while making an HTTP network request is the unavailability of network.
- In such a case, you need to add some conditional logic to make sure you are checking that there is no error before you try to interact with the response.
- You can console an error message informing the user that you currently do not have access to the network.

Error Object

- Error object is an implementation of a constructor function that uses a set of instructions to create an object.
- The first argument of an error object is its description which is a human-readable string.
- They also have a name property which is the computer-readable part of the object.
- You can extend the native error object using the following code:

app.js ...

```
1  class FancyError extends Error {
2      constructor(args){
3          super(args);
4          this.name = "FancyError"
5      }
6  }
7  console.log(new Error('A standard error'))
8  console.log(new FancyError('An augmented error'))
```

Try Catch

- Throw stops the program and finds a catch to execute.
- It diminishes the risk of any further error occurring by preventing any more functions running.
- With the program halted, JavaScript will find the nearest catch to execute.
- If no try/catch is found, the exception throws and the NodeJS process will exit, causing the server to restart.
- You can implement try and catch using the following code:

app.js ...

```
1   function doAthing() {
2       byDoingSomethingElse();
3   }
4   function byDoingSomethingElse() {
5       throw new Error('Error!');
6   }
7   function init() {
8       try {
9           doAthing();
10      } catch(e) {
11          console.log(e);
12      }
13  }
14  init();
```

Call Stack

- Call stack is a stack with LIFO structure that keeps track of all the functions running.
- Call stack can help us to trace the origin of a failure.
- A stack trace is a set of messages you get, when there is something wrong with the program.
- An example of a stack trace is given below:

```
C:\Users\shalini.basu\temp1>node app.js
Error: Error!
    at byDoingSomethingElse (C:\Users\shalini.basu\temp1\app.js:6:11)
    at doAthing (C:\Users\shalini.basu\temp1\app.js:2:5)
    at init (C:\Users\shalini.basu\temp1\app.js:11:9)
    at Object.<anonymous> (C:\Users\shalini.basu\temp1\app.js:18:1)
    at Module._compile (internal/modules/cjs/loader.js:701:30)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:712:10)
    at Module.load (internal/modules/cjs/loader.js:600:32)
    at tryModuleLoad (internal/modules/cjs/loader.js:539:12)
    at Function.Module._load (internal/modules/cjs/loader.js:531:3)
    at Function.Module.runMain (internal/modules/cjs/loader.js:754:12)
```

Callback function

- A callback is a function passed as an argument to another function which will call it back later.

```
fs.readFile(FileName, function(err, file) {
   if(err)
      handleError(err);
   console.log("file: ", file)
})
```

- When fs.readFile has fetched FileName, it executes the callback function.
- The callback function handles the error if an error is thrown and displays the retrieved file.
- It takes two functions: err and file. If no error is thrown, then the first argument is taken as null.

Custom Callback

- You can make your own custom callback function. For example:

```
const checkFileType = function (arg,callback) {
    if(typeof arg !== 'number'){
        return callback("not a number");
    }
    callback(null,"yes it is a number");
}

checkFileType(15,function(err, data){
    if(err){
        console.log(err);
    }else{
        console.log(data);
    }
});
```

- We define a function with two arguments: arg and callback. Arg is any argument you pass while callback is a function which is assigned to a constant variable checkFileType.

Callback Chaining

- Callback chaining is the way of chaining together multiple callbacks to do multiple things in a specific order.

- This is also called *callback hell*. For example:

```
doSomething(param1, param2, function(err, paramx){
        doMore(paramx, function(err, result){
                insertRow(result, function(err){
                        yetAnotherOperation(someparameter, function(s){
                        somethingElse(function(x){
                        });
                        });
                });
        });
});
```

- This leads to confusing and code that is difficult to read.

Callback Chaining

You can avoid callback hell in the following ways:

- The best way to reduce complexity is to modularize a program. For example, modules like *pluralize*, *csv*, and *clone* will help you handle specific tasks that would otherwise clutter the code.
- Name your functions as it gives you a syntax reference point.
- Maintaining better separation of code can reduce clustering. You can avoid deeply nested structures by declaring a callback beforehand and calling it later.
- *Async.js* also helps in avoiding callback hell. It's helper methods like *series* and *parallel can be used.*

NodeJS: HTTP

Duration: 80 min.

Problem Statement:

You are given a project to demonstrate how to create HTTP requests in NodeJS.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to demonstrate HTTP requests:

1. Write a program in NodeJS to take the address, say, Kolkata, and convert it into a latitude-longitude pair

2. Set up an error handler for the above program

3. Modify the geocoding program using callbacks

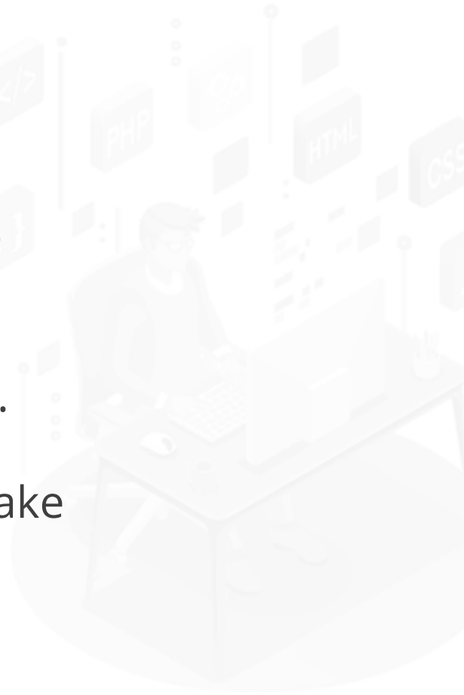
4. Modify the geocoding program to implement callback chaining and accept location via command line argument

5. Push the code to your GitHub repositories

NodeJS App Deployment

Version control

- Version control is a system that records all the changes made to a file or a set of files.
- The most popular and widely used version control system is Git.
- It is fast and easy to work with regardless of the programming language you are working with.
- Suppose you had an application which is used by thousand users by paying you.
- You want to add a new feature to give your users something new. Hence you make some changes to your project and you deploy it.



Version control

- Now, let’s say, you discover a bug in your application.
- You want to now revert to the previous working state.
- You need to spend some more time working on the new feature and you will deploy it back to the users once it is ready.
- You will need version control to easily revert back to the previous working state of the application.
- Version control gives you flexibility and confidence as you can experiment with your code but always get back to the working state you had before.

GitHub and Heroku

GitHub

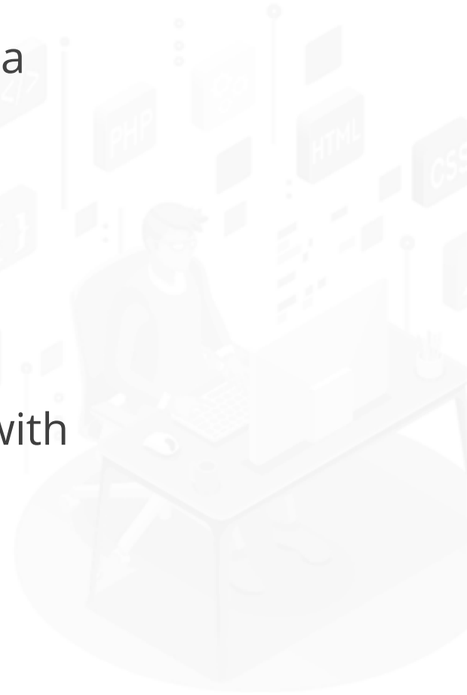
GitHub is a very popular software development platform which provides all the necessary tools to manage software development projects.

Heroku

Heroku provides all the tools and infrastructure required to deploy a NodeJS application to a production ready server.

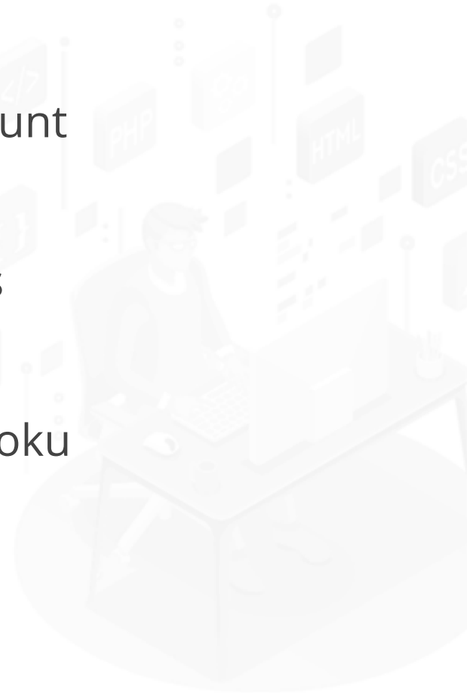
GitHub

- The code of all the NPM modules is hosted on GitHub.
- You can go to https://github.com and either log in to your account or sign up for a new one.
- You will all your code projects on the left hand side.
- You can find your feed in the middle and on the right.
- GitHub also needs access to your project code so as to allow you to collaborate with others, track code changes over time, or manage issues in your application.



Heroku

- Heroku is an application deployment platform.
- It gives you all the tools and infrastructure necessary for your application.
- You can go to https://dashboard.heroku.com/login and either log in to your account or sign up for a new one.
- You also need to install Heroku's command line tools that gives access to various commands that you can execute from your terminal.
- You can go to https://devcenter.heroku.com/articles/heroku-cli to download Heroku CLI.
- You need to restart your terminal once installed.



NodeJS App Deployment

Duration: 70 min.

Problem Statement:

You are given a project to demonstrate how to deploy a NodeJS application.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to demonstrate deployment of a node.js app:

1. Create a NodeJS project in your IDE

2. Sign up on GitHub and Heroku

3. Integrate Git with your NodeJS application

4. Set up SSH keys

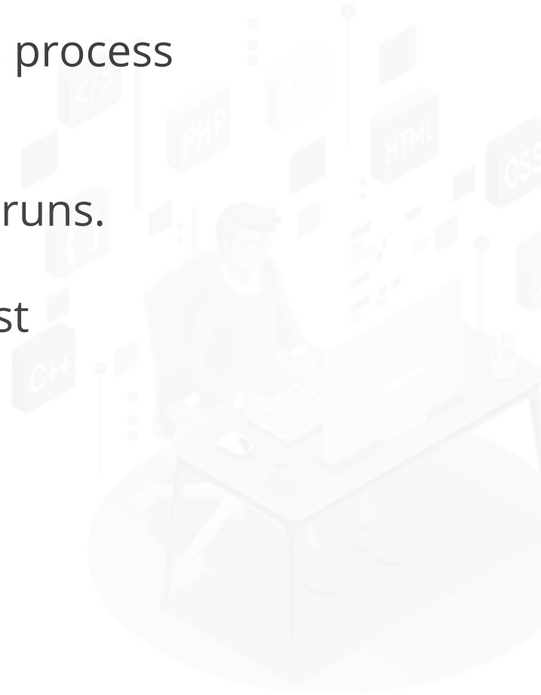
5. Push your code to GitHub

6. Deploy your NodeJS application to Heroku

File System

Synchronous I/O

- Synchronous or blocking is when the execution of JavaScript code in the NodeJS process is stopped until a non-JavaScript operation completes.
- This is because the event loop cannot run JavaScript while a blocking operation runs.
- Synchronous functions in the NodeJS standard library that use *libuv* are the most commonly used synchronous operations.
- Native modules also have blocking methods.

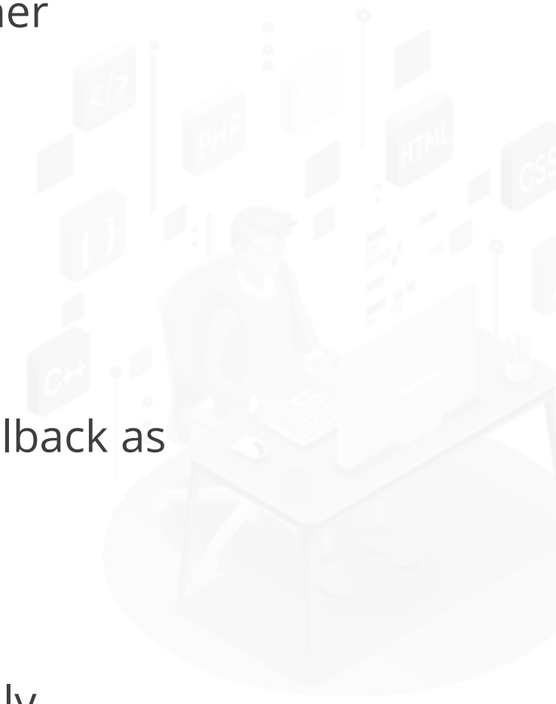


Asynchronous I/O

- Asynchronous programing is a design pattern that ensures non-blocking code execution.
- Asynchronous code executes without having any dependency on others.
- It improves system efficiency and throughput.

File System

- A file system is a mechanism that controls how data is accessed and managed.
- The *fs* module in NodeJS allows you to interact with the file system in a manner closely modeled around standard POSIX functions.
- The file system module can be imported using the following syntax:
  const fs = require('fs');
- All file system operations have synchronous and asynchronous forms.
- Unlike synchronous functions, asynchronous functions take a completion callback as its last argument.
- The first argument of the callback is reserved for an exception.
- The argument is null or undefined if the operation was completed successfully.



Synchronous vs. Asynchronous I/O

Asynchronous

```
var filesystem = require('fs');
filesystem.readFile("myfirstexample.txt", "utf8",
function(error, data) {
  console.log(data);
});
morework()
```

- *filesystem.readFile()* is non-blocking, so JavaScript execution can continue and *morework()* will be called first.
- This allows for higher throughput.

Synchronous

```
var filesystem = require('fs');
var data =
filesystem.readFileSync("myfirstexample.txt",
"utf8");
  console.log(data);
morework()
```

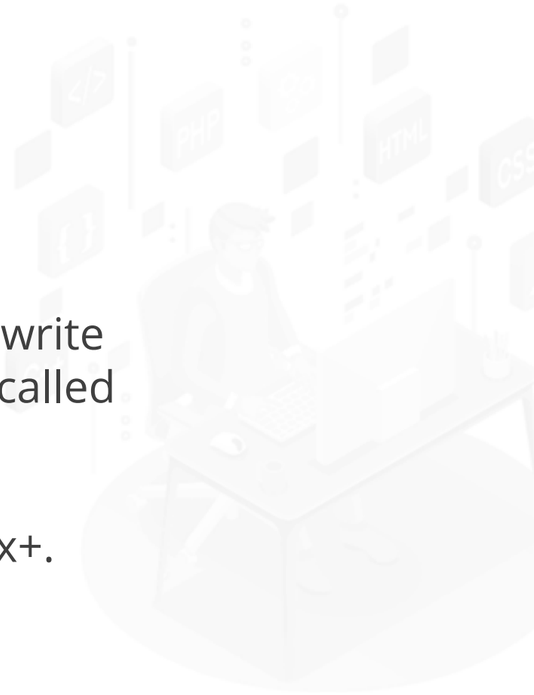
- The second line blocks the execution of any additional JavaScript code till the entire file is read.
- *morework()* will be executed after console.log.

Open a File

- You can open a file asynchronously using the following syntax:
  fs.open(path, flags[, mode], callback)

- The description of the parameters is given below:
  - *path*: It is a string having the full path with file name
  - *flag*: It indicates the behavior of the file to be opened
  - mode: It sets the file permission if the file is created; defaults to 0666 readwrite
  - *callback*: It is a callback function with two parameters, *err* and *fd*, which is called after the open operation completes

- Flags for read/write operations are r, r+, rs, rs+, w, wx, w+, wx+, a, ax, a+, and ax+.



Open a File

For example:

```
JS app.js > ...
1   const fs = require("fs");
2   console.log("Going to open file!");
3   fs.open('input.txt', 'w', function(err, fd) {
4       if (err) {
5           return console.error(err);
6       }
7       console.log("File opened successfully!");
8   });
```

Output:

```
C:\Users\shalini.basu\temp1>node app.js
Going to open file!
File opened successfully!
```

Get File Information

- You can get the information about a file using the following syntax:
  fs.stat(path, callback)

- The description of the parameters is given below:
  - *path*: It is a string having the full path with file name
  - *callback*: It is a callback function with two parameters, *err* and *stats*, where *stats* is an object of *fs*

- *fs.Stats* has several useful methods to check file type like *stat.isFile(), stat.isDirectory(), stat.isCharacterDevice(), stats.isBlockDevice(), stats.isSymbolicLink(), stats.isFIFO(), and stats.isSocket().*

Get File Information

For example:

```
JS app.js > fs.stat('input.txt') callback
1   var fs = require("fs");
2   console.log("Going to get file info!");
3   fs.stat('input.txt', function (err, stats) {
4       if (err) {
5           return console.error(err);
6       }
7       console.log(stats);
8       console.log("Got file info successfully!");
9       console.log("isFile ? " + stats.isFile());
10      console.log("isDirectory ? " + stats.isDirectory());
11  });
```

Output:

```
C:\Users\shalini.basu\temp1>node app.js
Going to get file info!
Stats {
  dev: 2256307645,
  mode: 33206,
  nlink: 1,
  uid: 0,
  gid: 0,
  rdev: 0,
  blksize: undefined,
  ino: 6755399441241048,
  size: 0,
  blocks: undefined,
  atimeMs: 1566470625517.424,
  mtimeMs: 1566471016075.1892,
  ctimeMs: 1566471016075.1892,
  birthtimeMs: 1566470625517.424,
  atime: 2019-08-22T10:43:45.517Z,
  mtime: 2019-08-22T10:50:16.075Z,
  ctime: 2019-08-22T10:50:16.075Z,
  birthtime: 2019-08-22T10:43:45.517Z }
Got file info successfully!
isFile ? true
isDirectory ? false
```

Read a File

- You can read a file asynchronously using the following syntax:
  fs.readFile(fileName [options], callback)

- The description of the parameters is given below:
  - *filename*: It is a string having the full path with file name
  - *options*: It can be an object or a string that can include encoding and flag; default encoding is ‘utf8’ and default flag is ‘r’
  - *callback*: It is a callback function with two parameters, *err* and *fd*, which is called after the readFile operation completes

Read a File

For example:

```
JS app.js > ...
1   var fs = require('fs');
2   fs.readFile('input.txt', 'utf8', function (err, data) {
3       if (err) throw err;
4       console.log(data);
5   });
```

Output:

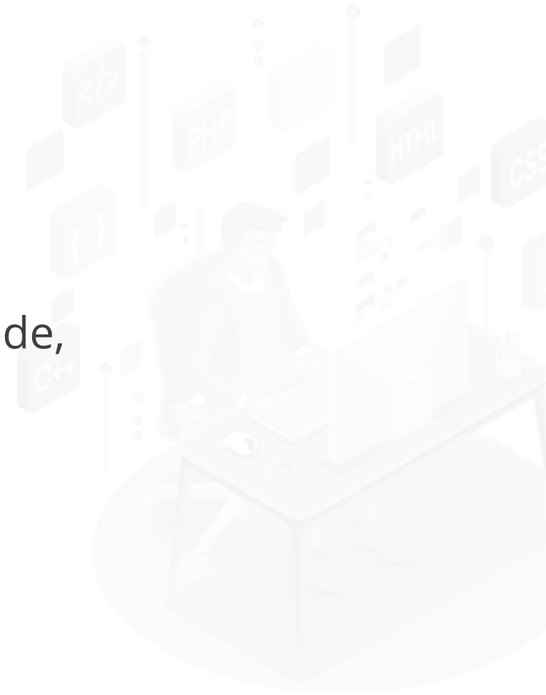
```
C:\Users\shalini.basu\temp1>node app.js
This is an input text file!
```

Write to a File

- You can read a file asynchronously using the following syntax:
  fs.writeFile(fileName, data[options], callback)

- The description of the parameters is given below:
  - *filename*: It is a string having the full path with file name
  - *data*: It is the content to be written in a file
  - *options*: It can be an object or a string that can include encoding, mode, and flag; default encoding is ‘utf8’ and default flag is ‘r’
  - *callback*: It is a callback function with two parameters, *err* and *fd*, which is called after the writeFile operation completes

- A new file will be created if there is no existing file with the given name.



Write to a File

For example:

app.js > fs.writeFile('input.txt', 'Hello World!') callback

```
1   var fs = require('fs');
2   fs.writeFile('input.txt', 'Hello World!', function (err) {
3       if (err)
4           console.log(err);
5       else
6           console.log('Write operation complete.');
7   });
```

Output:

```
C:\Users\shalini.basu\temp1>node app.js
Write operation complete.
```

Append File

fs.appendFile works in the same way as *fs.readFile()*. The content specified will be appended to the file.

For example:

app.js > fs.appendFile('input.txt', 'Good Morning!') callback

```
1   var fs = require('fs');
2   fs.appendFile('input.txt', 'Good Morning!', function (err) {
3       if (err)
4           console.log(err);
5       else
6           console.log('Append operation complete.');
7   });
```

Output:

```
C:\Users\shalini.basu\temp1>node app.js
Append operation complete.
```

Close a File

- You can read a file asynchronously using the following syntax:
  fs.close(fd, callback)

- The description of the parameters is given below:
  - *fd*: It is the file descriptor returned by fs.open() method
  - *callback*: It is a callback function with two parameters, *err* and *fd*, which is called after the writeFile operation completes

Close a File

For example:

```
JS app.js > ...
1   var fs = require("fs");
2   console.log("Going to open an existing file");
3   fs.open('input.txt', 'r+', function(err, fd) {
4      if (err) {
5         return console.error(err);
6      }
7      console.log("File opened successfully!");
8      console.log("Going to read the file");
9      fs.readFile('input.txt', 'utf8', function(err, data) {
10        if (err) {
11           console.log(err);
12        }
13        console.log(data);
14        fs.close(fd, function(err) {
15           if (err) {
16              console.log(err);
17           }
18           console.log("File closed successfully.");
19        });
20     });
21  });
```

Output:

```
C:\Users\shalini.basu\temp1>node app.js
Going to open an existing file
File opened successfully!
Going to read the file
Hello World!Good Morning!
File closed successfully.
```

Delete a File

- You can read a file asynchronously using the following syntax:
  fs.unlink(path, callback)

- The description of the parameters is given below:
  - *path*: It is a string having the full path with file name
  - *callback*: It is a callback function with one parameter which is a possible exception

Delete a File

For example:

```
JS app.js > ...
1   var fs = require("fs");
2   console.log("Going to delete an existing file");
3   fs.unlink('input.txt', function(err) {
4       if (err) {
5           return console.error(err);
6       }
7       console.log("File deleted successfully!");
8   });
```

Output:

```
C:\Users\shalini.basu\temp1>node app.js
Going to delete an existing file
File deleted successfully!
```

__dirname and __filename

- __dirname and __filename are NodeJS global objects.
- NodeJS __dirname is a string which specifies the name of the directory that contains the current code.
- NodeJS __filename is a sting which specifies the filename of the code being executed.
- __filename is the resolved absolute path of the code file.

File System

Duration: 60 min.

Problem Statement:

You are given a project to demonstrate file system in NodeJS.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to demonstrate file system:

1. Create a NodeJS project in your IDE

2. Write a program to read a text file synchronously and asynchronously

3. Write a program to write to a text file synchronously and asynchronously

4. Write a program to locate and read a file using __dirname

5. Write a program to find the current directory and filename

6. Initialize the .git file

7. Add and commit the program files

8. Push the code to your GitHub repository

Multiprocessing in NodeJS

Child Process

- The child_process module in NodeJS provides the ability to spawn child processes. This is primarily provided by *child_process.spawn()* function.

```
const {spawn} = require('child_process');
const ls = spawn('ls', ['-lh', '/usr']);
ls.stdout.on('data', (data) => {
        console.log(`stdout: ${data}`);
});
ls.stderr.on('data', (data) => {
        console.log(`stderr: ${data}`);
});
ls.on('close', (code) => {
        console.log(`child process exited with code
${code}`);
});
```


- The child_process.spawn() method spawns the child process asynchronously, without blocking the Node.js event loop. The child_process.spawnSync() function provides equivalent functionality in a synchronous manner that blocks the event loop until the spawned process either exits or is terminated.

Child Process

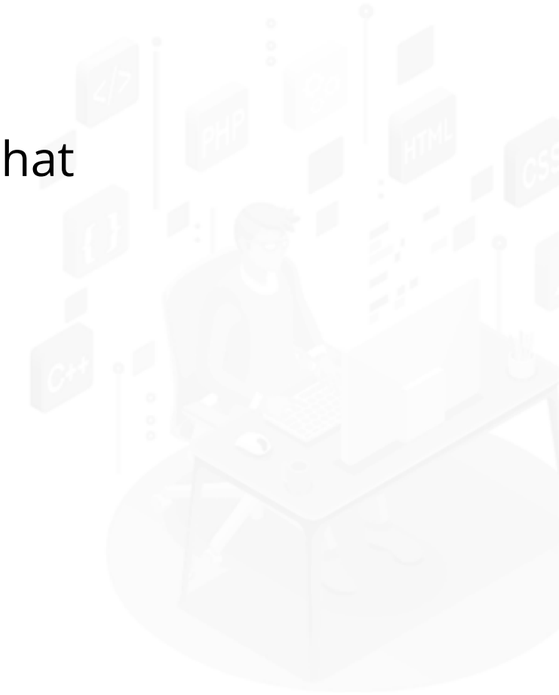
The child_process module provides a handful of synchronous and asynchronous alternatives
to child_process.spawn() andchild_process.spawnSync(). Each of these alternatives are implemented on top
of child_process.spawn() or child_process.spawnSync().

- child_process.exec(): spawns a shell and runs a command within that shell, passing the stdout and stderr to a callback function when complete.

- child_process.execFile(): similar to child_process.exec() except that it spawns the command directly without first spawning a shell by default.

Child Process

- command directly without first spawning a shell by default.
- child_process.fork(): spawns a new Node.js process and invokes a specified module with an IPC communication channel established that allows sending messages between parent and child.
- child_process.execSync(): a synchronous version of child_process.exec() that will block the Node.js event loop.
- child_process.execFileSync(): a synchronous version of child_process.execFile() that will block the Node.js event loop.

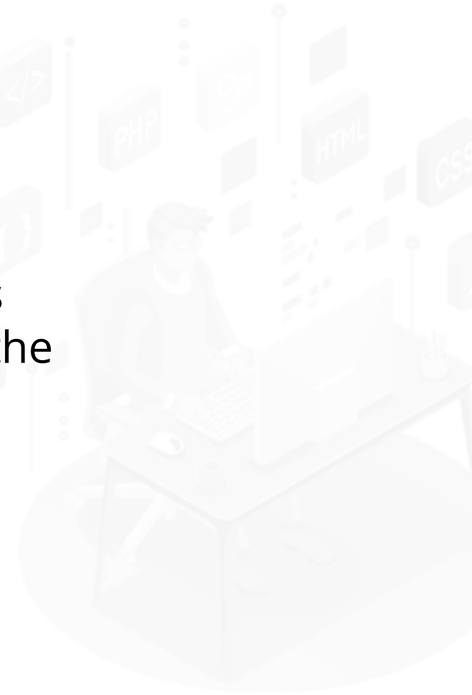


Cluster

- To take advantage of multi-core systems, the user can use a cluster of NodeJS processes to handle the load.
- A cluster scales the execution of an application on multiple processor cores by creating worker processes.
- Worker processes share a single port.
- A *cluster module* is a NodeJS module that contains a set of functions and properties.
- The cluster module allows easy creation of child processes that all share server ports.

Cluster

- A *master* process can be forked into any number of *worker* processes.
- A worker process is spawned using the *child_process.fork()* method, so they can communicate with the parent process via inter-process communication.
- The returned *childProcess* object has a built-in communication channel. It allows messages to be passed back and forth between the parent and the child using the send() method.
- Each process has its own memory with their own v8 instances.
- You should not spawn a large number of child NodeJS processes as additional resource allocations are required.



Cluster

The following code shows how a cluster module is used:

```
const cluster = require('cluster’);
const http = require('http’);
const numCPUs = require('os').cpus().length;
if (cluster.isMaster) {
  console.log(`Master ${process.pid} is running`);
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }
  cluster.on('exit', (worker, code, signal) => {
    console.log(`worker ${worker.process.pid} died`); });
} else
  http.createServer((req, res) => {
    res.writeHead(200);
    res.end('hello world\n'); }).listen(8000);
  console.log(`Worker ${process.pid} started`);
}
```

Cluster

- The worker processes are spawned using the child_process.fork() method, so as to communicate with the parent.
- The cluster module supports two methods of distributing incoming connections.
- The first one is the round-robin approach, where the master process listens on a port, accepts new connections and distributes them across the workers.
- The second approach is where the master process creates the listen socket and sends it to interested workers. The workers then accept incoming connections directly.

Multiprocessing in NodeJS

Duration: 90 min.

Problem Statement:

You are given a project to demonstrate multiprocessing in NodeJS.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to demonstrate multiprocessing:

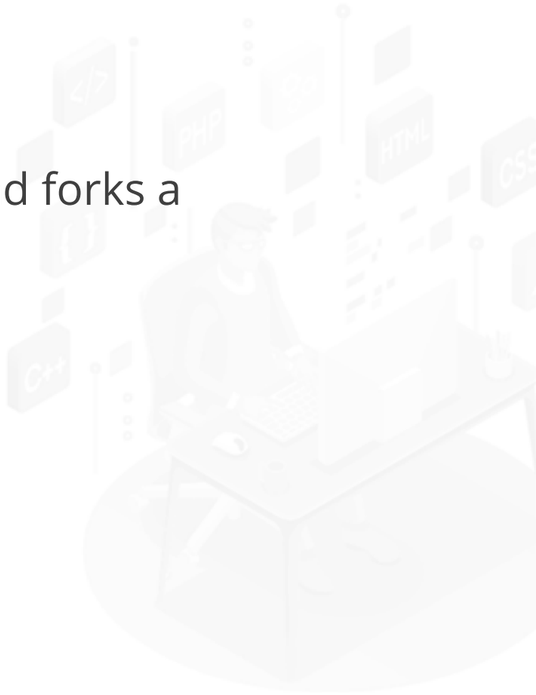
1. Create a NodeJS project in your IDE

2. Write a program to create a master process that retrieves the number of CPUs and forks a worker process for each CPU

3. Initialize the .git file

4.Add and commit the program files

5. Push the code to your GitHub repository



Key Takeaways

- Call stack is a stack with LIFO(Last In First Out) structure provided by v8 JavaScript engine which keeps tracks of all the functions running in the program.
- Callback queue is a list of messages and all the callback functions ready to be executed.
- HTTP requests are used by applications to communicate with the outside world.
- Git is the most popular and widely used version control system that records all the changes made to a file or a set of files.
- GitHub and Heroku are the most widely used application deployment platforms.

Task Planner App

Problem Statement: **Duration: 60 min.**

Write a NodeJS program to create a Task Planner application wherein you can add, delete, and list your daily tasks.

You need to use the following:

- Visual Studio
- NodeJS
- Git
- GitHub

You need to make sure that the versions are tracked in GitHub and Task Planner application should work properly.

LESSON-END PROJECT

Before the Next Class

Course: Restful API Design with Node, Express, and MongoDB

You should be able to:

- Get introduced to RESTful API
- Create API using CRUD operations
- Enable authentication and security
- Add real-world features to API
- Explain caching and rate limiting
- Deploy the application