Develop a Reliable Backend with Node and Express

Getting Started with Node.js

A Day in the Life of a MERN Stack Developer

Joe works as a MERN Stack Developer for a software development company which appreciates his commitment and progress. He has diligently completed all the projects assigned.

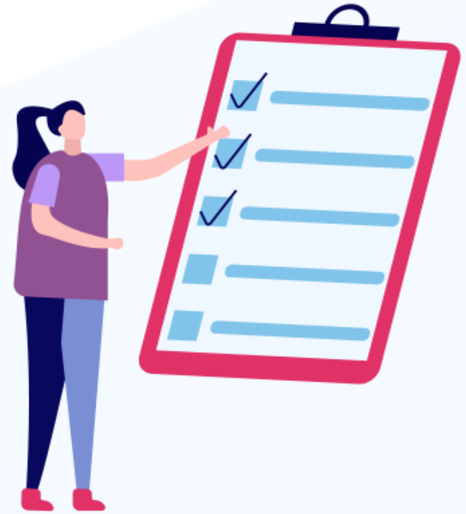
Now, Joe has been assigned to an important project, where he has been asked to build a back-end web application that offers high performance in real-time applications and is cost-effective.

In this lesson, he will learn how to solve this real-world scenario effectively and quickly.

Learning Objectives

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

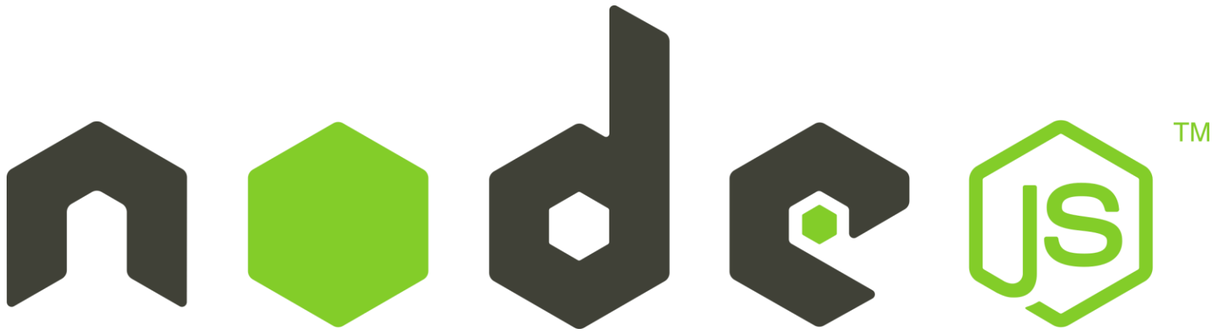
- Implement a node server and obtain different data responses by handling various API endpoints and data sources
- Assess the working of Node.js using the asynchronous and non-blocking I/O concept, allowing for concurrent handling of multiple tasks
- Illustrate how a response is given to the client using HTML, enabling the server to dynamically generate HTML content based on client requests
- Implement a callback queue to organize asynchronous operations in Node.js, ensuring a structured flow of execution for asynchronous tasks



Introduction to Node.js

Overview

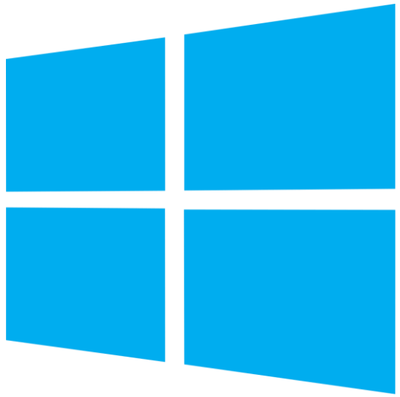
Node.js is a JavaScript framework that serves as a platform for JavaScript based web applications.



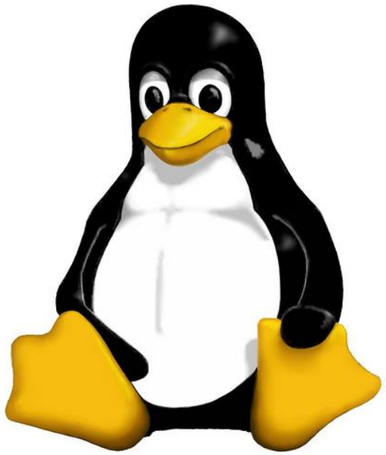
Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.

Node.js Installation

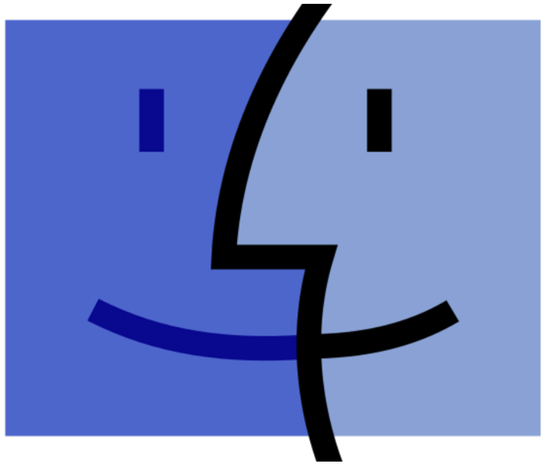
Node.js is supported by all the popular operating systems.



Windows



Linux



Mac OS

Use different package managers for different Linux distributions.

Node.js for Alpine Linux

In the main repository, Node.js, LTS, and npm packages are available.

```
apk add nodejs npm
```

Node.js for Alpine Linux

Install Node.js current from the community repository

```
apk add nodejs-current
```

Node.js for Arch Linux

In the community repository, Node.js and npm packages are available.

```
pacam -S nodejs npm
```

Node.js for Linux Distribution

It is available in the Node.js module in CentOS/RHEL 8 and Fedora.

Install Node.js

```
dnf module install nodejs:<stream>
```

Corresponds to a major version of Node.js

To see a list of available stream:

```
dnf module list nodejs
```

Helps to view a list of available streams

Node.js for Linux Distribution: Example

To install Node.js 18:

Example:

```
dnf module install nodejs:18/common
```

Node.js for Linux Distribution

NodeSource offers binary distributions of Node.js for:

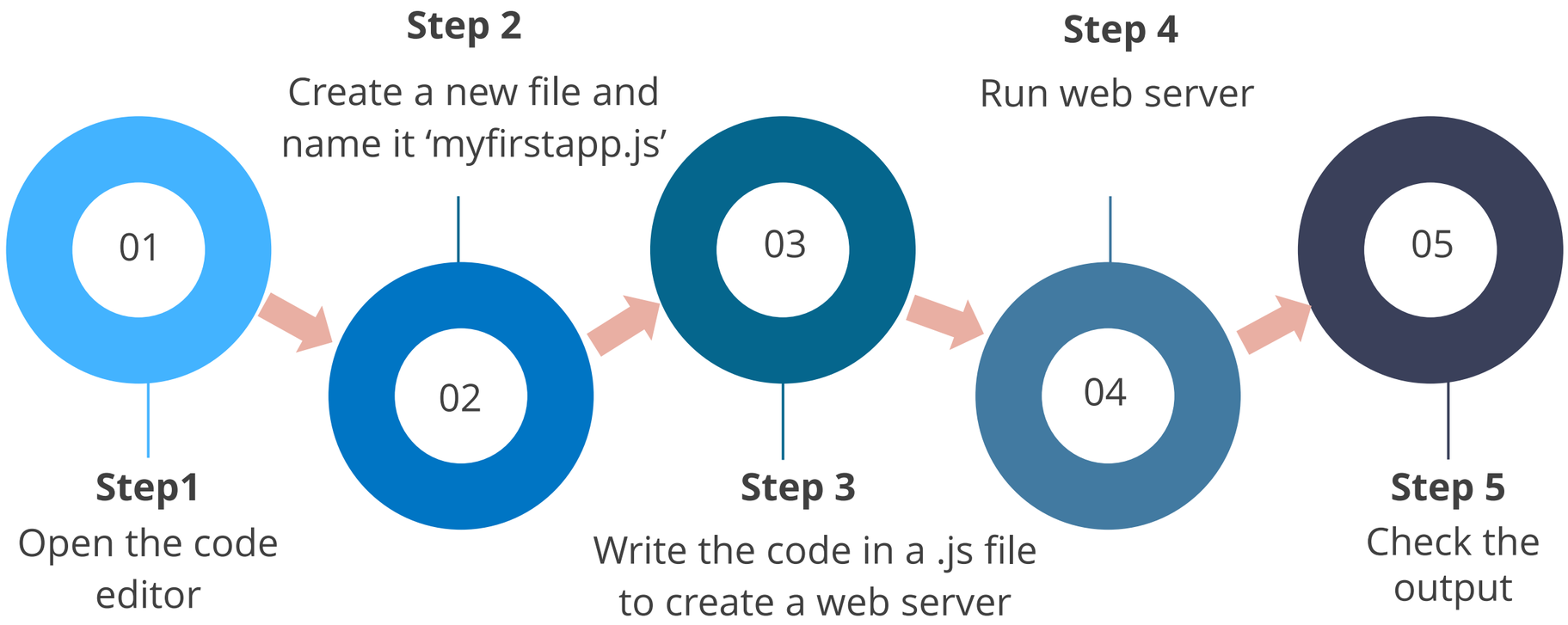
Debian

Ubuntu

Packages compatible with Linux distributions are available via Node.js snaps.

Creating an Application Using Node.js

Steps to build a Node.js application:



Creating an Application Using Node.js

Step 1: Create a web server

```
const http =require('http');
const hostname = '127.0.0.1';
const port = 8000;
const server = http.createServer((req,res)=>{
  res.statusCode=400;
  res.setHeader('Content-Type','text/plain');
  res.end('Welcome to my First Node Application');
  });

  server.listen(port,hostname, () => {
      console.log ('Server is up and running at host '+hostname+" with port "+port);
      });
```

Creating an Application Using Node.js

Step 2: Execute the source code

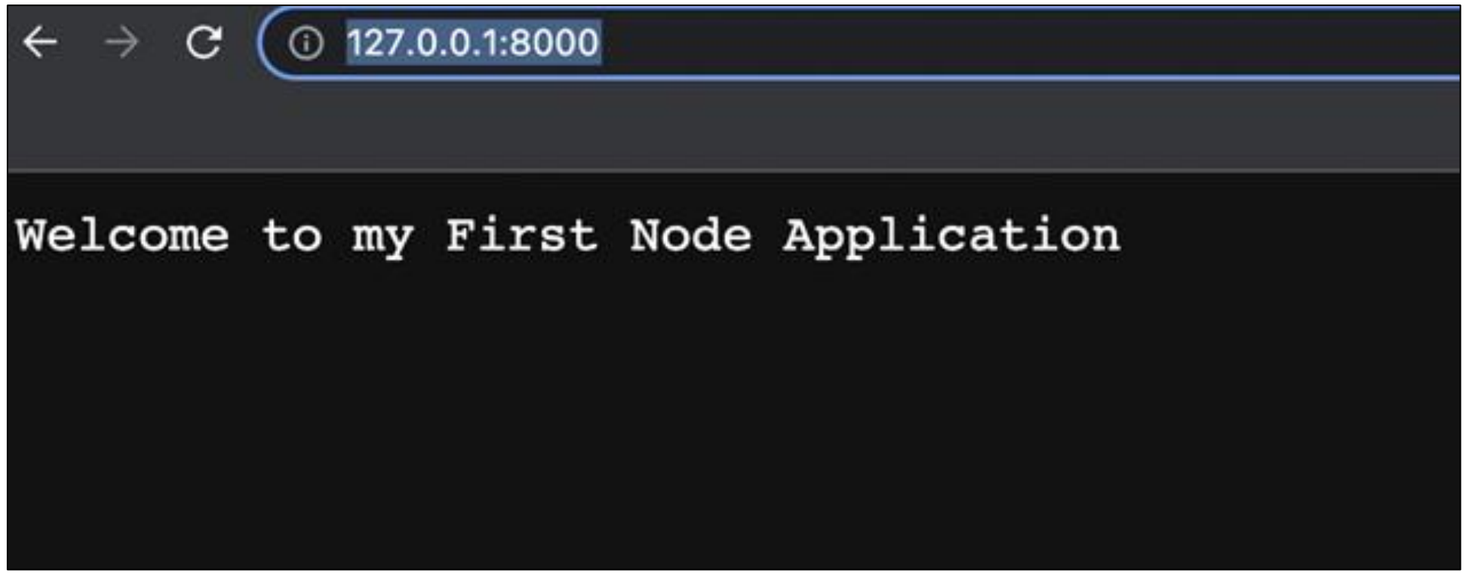
```
const http =require('http');
const hostname = '127.0.0.1';
const port = 8000;
const server = http.createServer((req,res)=>{
  res.statusCode=400;
  res.setHeader('Content-Type','text/plain');
  res.end('Welcome to my First Node Application');
  });

  server.listen(port,hostname, () => {
      console.log ('Server is up and running at host '+hostname+" with port "+port);
      });
```

Output:

```
Server is up and running at host 127.0.0.1 with port 8000.
```

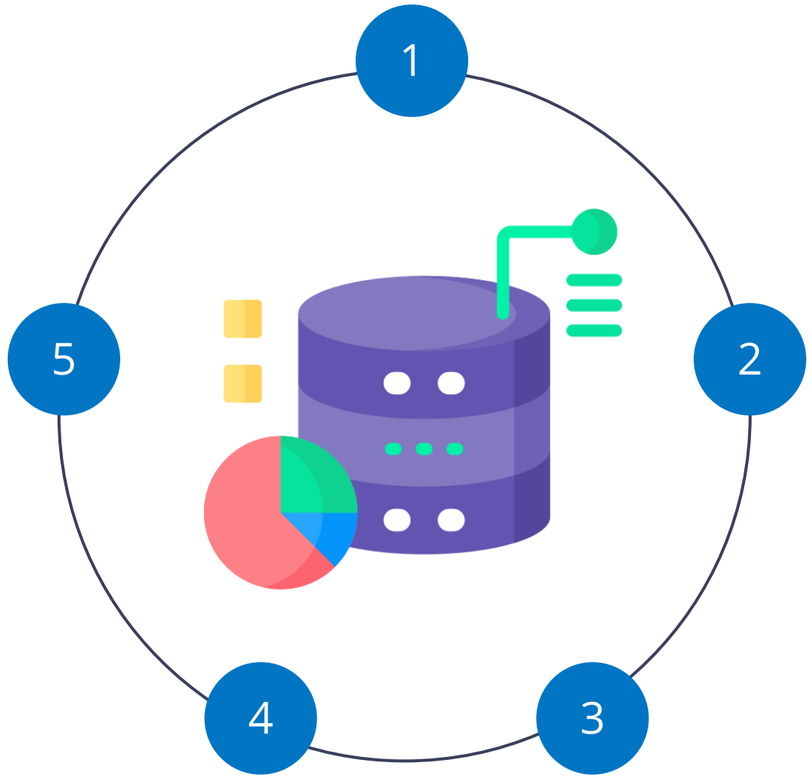
Creating an Application Using Node.js

To check the output, navigate to the browser, and type the IP address and port number:



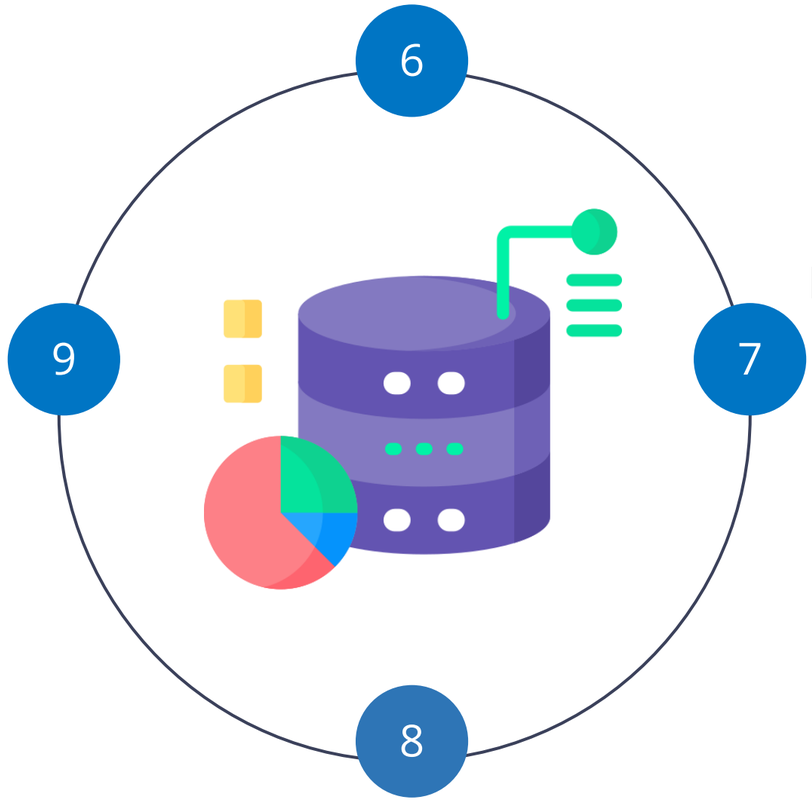
Roles and Usages of Node.js

1. Helps to write clean, efficient, and reusable code
2. Develops and manages server-side components
3. Deploys high-performance applications
4. Provides cybersecurity measures
5. Integrates server-side logic into the user-facing components



Roles and Usages of Node.js

6. Provides technical support to users by performing diagnostic tests and bug fixing



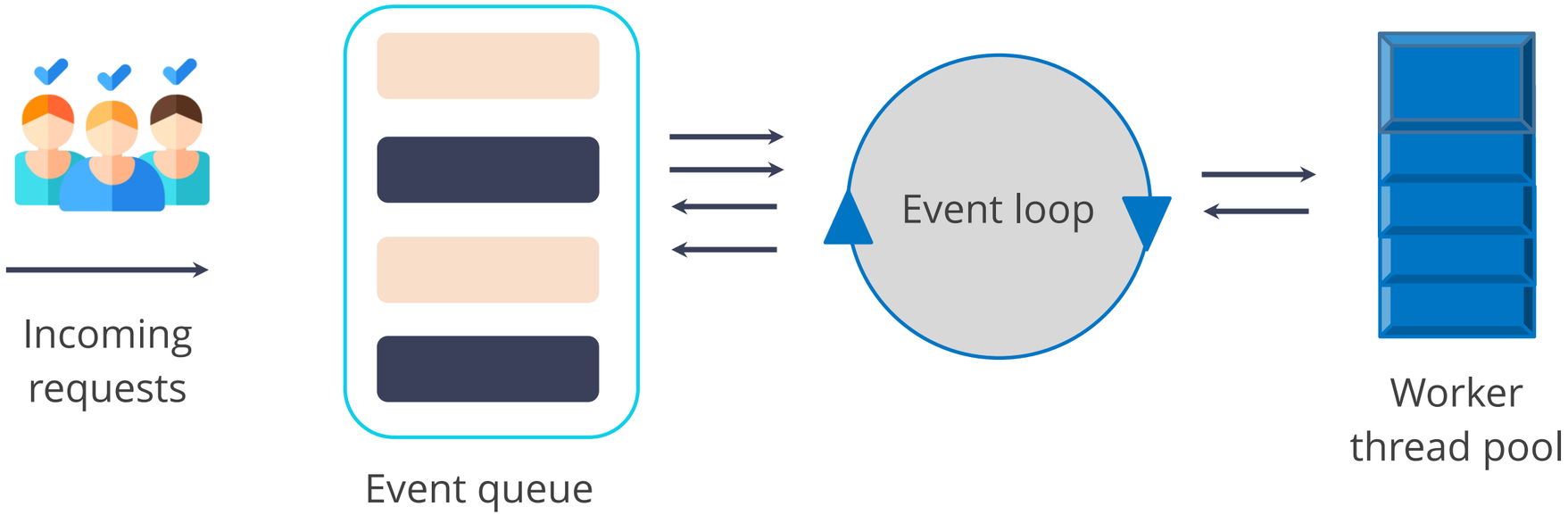
7. Prepares project reports and documents for software development processes

8. Implements data storage solutions

9. Ensures optimal performance of the main database and front-end requests

Working of Node.js

Node.js provides information to multiple clients by running parallel events on the server. It receives the request from the client and sends the response.

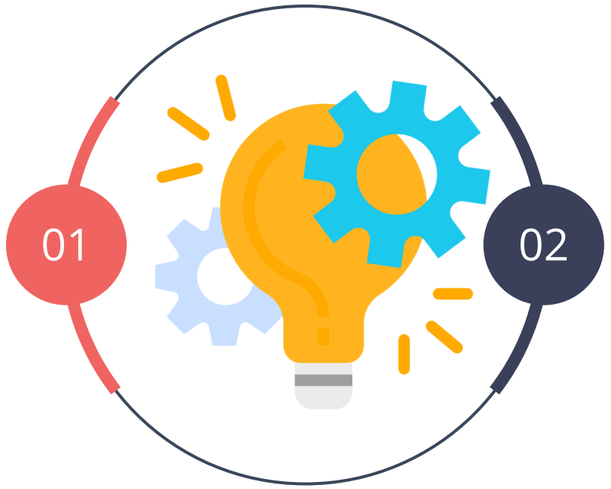


It manages multiple requests simultaneously in a single-thread without blocking it for a single request.

Working of Node.js: Concept

It works on the concept of:

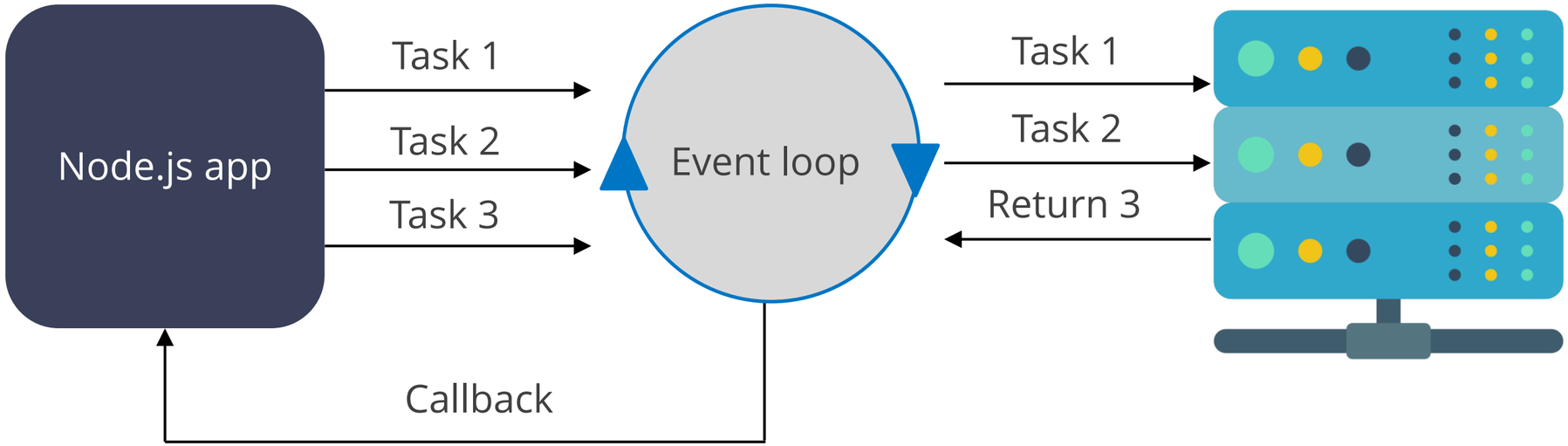
01 Asynchronous



02 Non-blocking I/O

Asynchronous

The event-driven architecture of Node.js helps to invoke callback functions as soon as the task is completed.



Non-Blocking I/O

Non-blocking I/O is the ability to handle several requests without causing the thread to be blocked for a single request.

Node.js is not used for CPU-intensive tasks like:

Calculations

Video processing

Installing and Creating a Node.js App

Problem Statement:

Duration:15 min.

You are given a task to install and create an application in Node.js.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Download Node.js
2. Install Node.js
3. Install npm
4. Install Visual Studio Code
5. Create a Node app

Node Server

Node Server: Overview

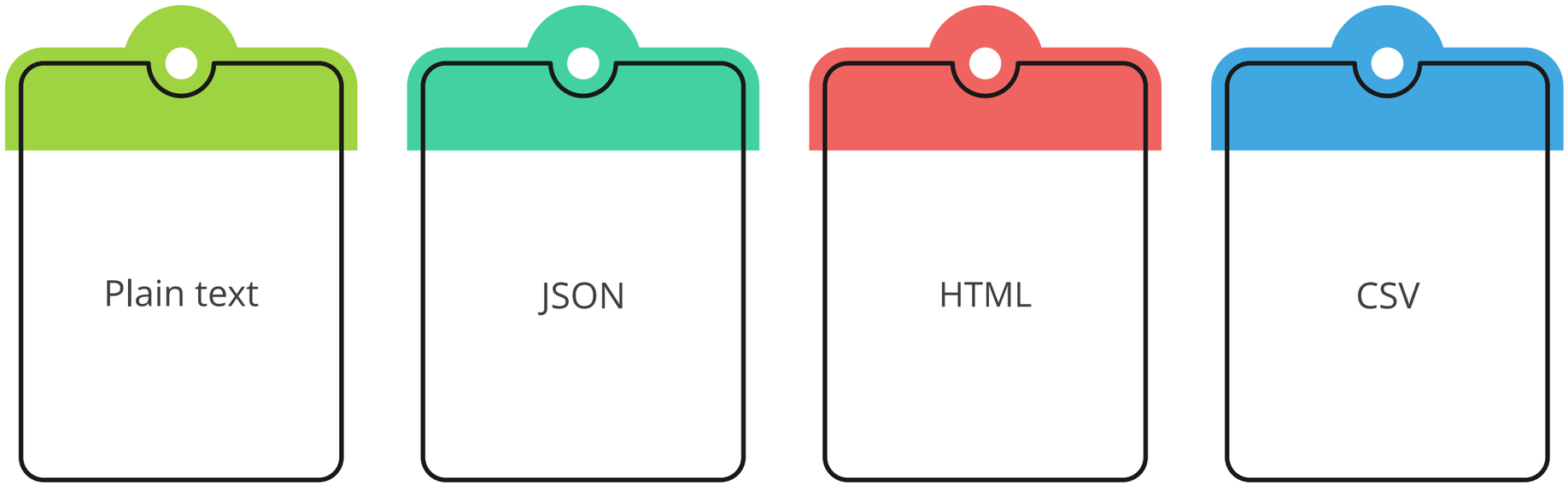
Node server is nothing but a Node.js application in execution. It runs, by default, on port 3000.

It helps to write web server codes.

Types of Data as a Response

Node.js has a built-in module called HTTP, which allows Node.js to transfer data over the Hypertext Transfer Protocol (HTTP).

This data can be:



JSON as a Response: Example

Code to obtain JSON as a response:

```
const http = require('http');
const hostname = '127.0.0.1’;
const port = 8000;
Const server = http.createServer((req, res) => {
res.setHeader("Content-Type", "application/json");
res.writeHead (200);
res.send('{"quote": "Search the Candle rather than cusring the Darkness"}’);
});
server.listen(port, hostname, () => {
console.log('Server is up and running at host '+hostname+" with port "+port);
});
```

HTML as a Response: Example

Code to obtain HTML as a response:

```
const http = require('http');
const hostname = '127.0.0.1';
Const port = 8000;
const server = http.createServer((req, res) · => {
res.setHeader("Content-Type", "text/html");
res.writeHead (200);
res.end('<html><body><h1>Search the Candle rather than cursing the
Darkness</h1></body></html>');
});
server.listen(port, hostname, () => {
console.log('Server is up and running at host '+hostname+" with port "+port);
});
```

CSV as a Response: Example

Code to obtain CSV as a response:

```
const http = require('http');
const hostname = '127.0.0.1';
const port = 8000;
Const server = http.createServer((req, res) => {
res.setHeader("Content-Type", "text/csv");
res.setHeader("Content-Disposition", "attachment; filename=data.csv");
res.writeHead (200);
res.end('id,name, email\n1, John Watson, john@example.com');
});
server.listen(port, hostname, () => {
console.log('Server is up and running at host '+hostname+" with port "+port);
});
```

Event Loop

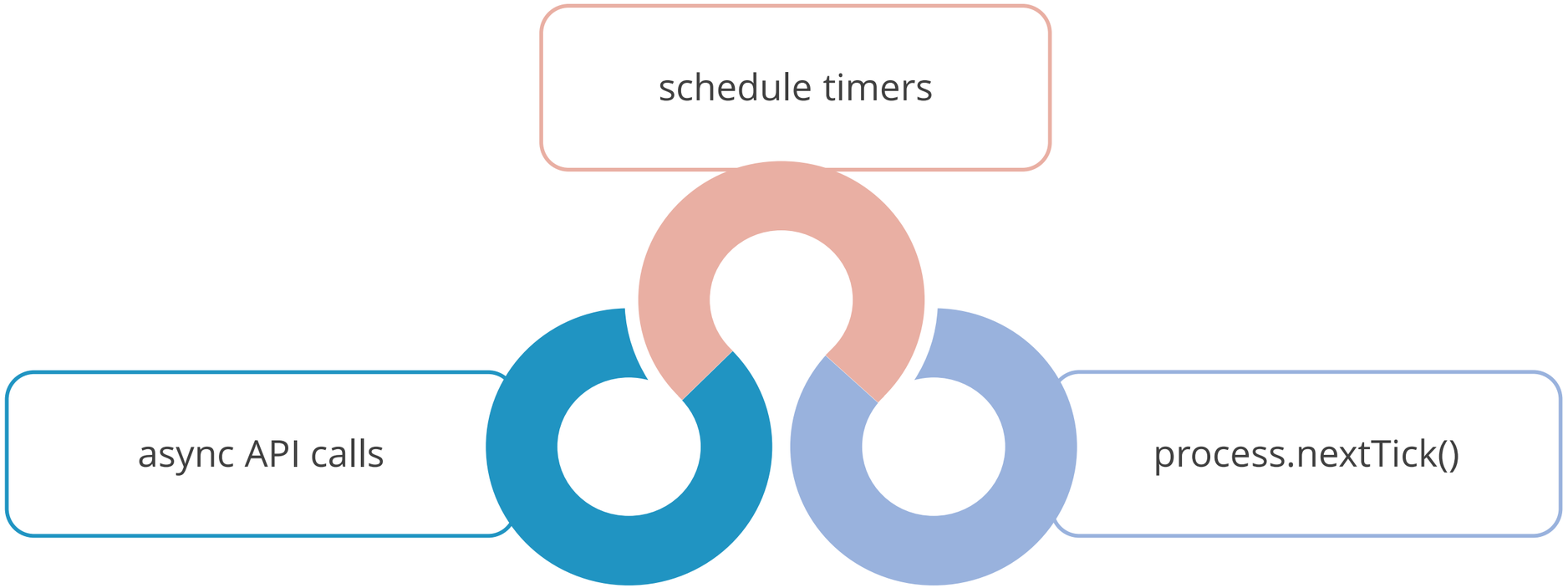
Event loop allows Node.js to perform blocking I/O operations by offloading operations to the system kernel.

Kernels are multi-threaded and handle multiple operations in the background.

Kernels inform Node.js to add callbacks to the poll queue to execute operations eventually.

How Does Event Loop Work?

The event loop is set up by Node.js, which also processes the input script, containing:

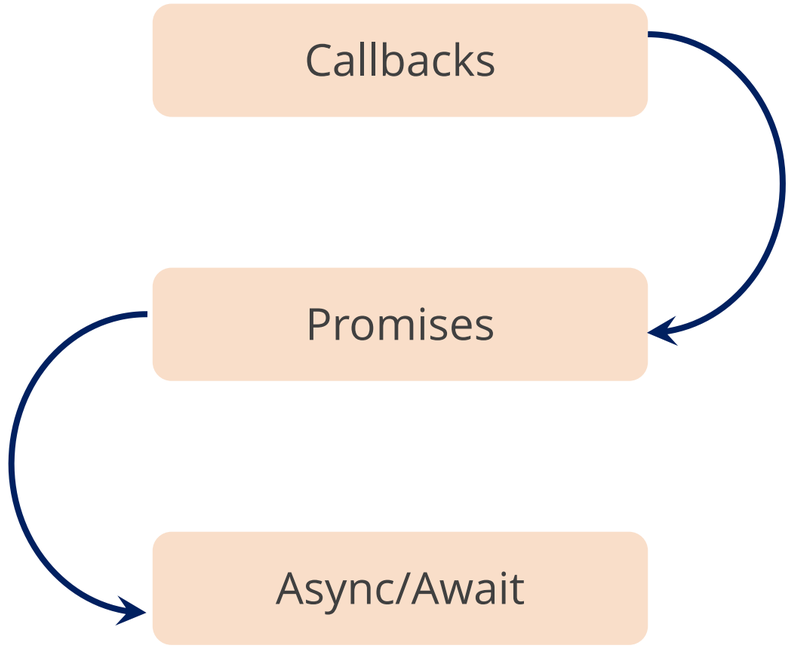


Async API Calls

The users can create asynchronous code using these three different approaches:

Syntax:

```
const request = require('request');

request('API URL', (error, response, body) => {
    if (error) {
    // Handle the error
    }

    if (response.statusCode != 200) {
       // Handle the status code
    }

    // Parse the Response
    result = JSON.parse(body);

});
```



Schedule Timers

Schedule timers are used to call a function after a set period.
There are three types of schedule timers, namely:

setImmediate()

It is used to execute the function in the next iteration of the event loop.

setInterval()

It is used to set a delay for functions that are executed again and again.

setTimeout()

It is used to execute a function after a designated amount of time.

process.nextTick()

The entire trip of an event loop is referred to as a tick.

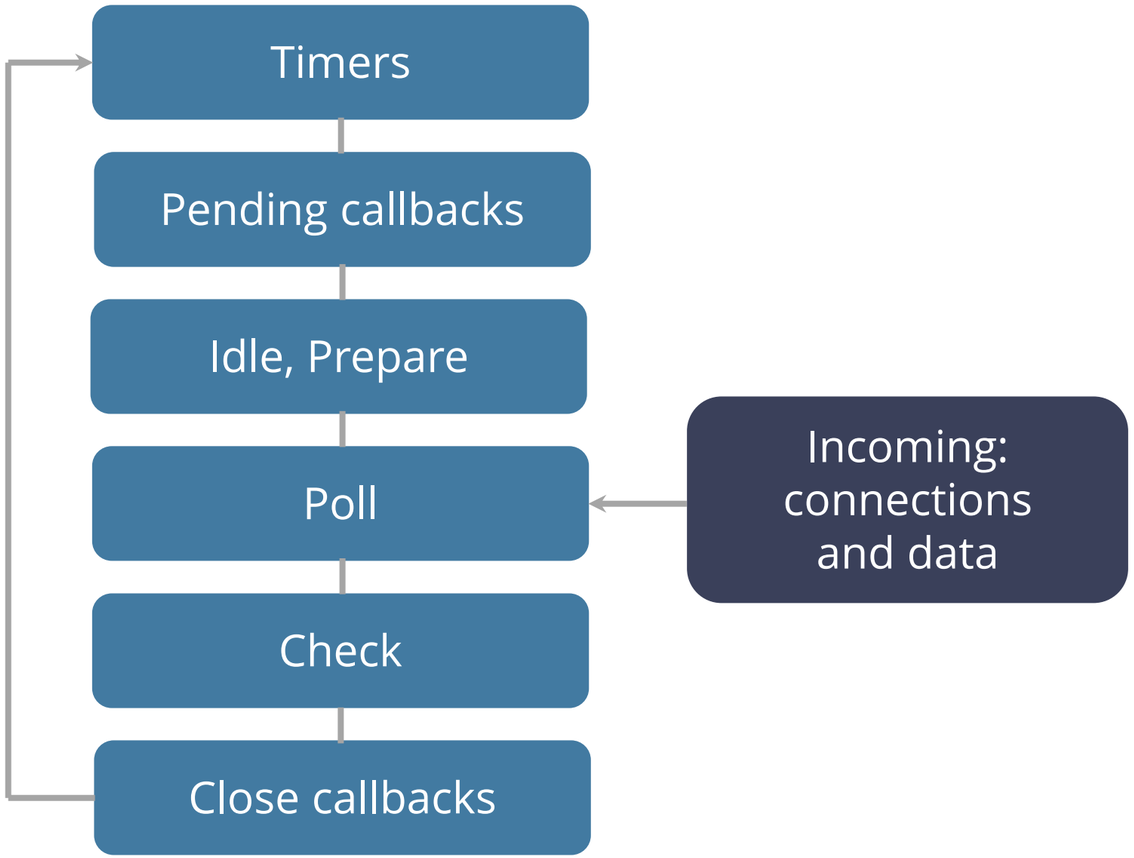
Example:

```
process.nextTick(() => {
  // Your logic goes here
});
```

A function, when passed to the **process.nextTick()** engine, gets instructed to invoke this function at the end of the current operation.

The Node Lifecycle

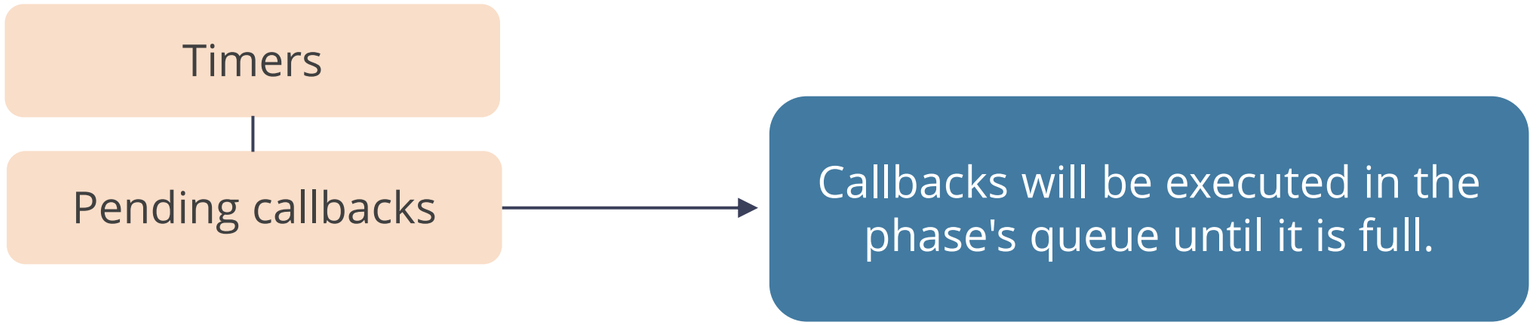
Simplified overview of the event loop:



The Node Lifecycle

There is a FIFO queue of callbacks to be executed for each phase of the node lifecycle.

The event loop enters a phase during which it carries out actions unique to that phase.



The event loop moves to the subsequent stage when the queue becomes full.

The Node Lifecycle: Components

Timers

Callbacks scheduled by setTimeout() and setInterval are carried out during this phase ().

Pending callback

I/O callbacks that were deferred to the following loop iteration are carried out during this phase.

Idle, Prepare

There is no external use for this phase; it is used only internally.

The Node Lifecycle: Components

Poll

I/O related callbacks node are blocked during this phase.

Check

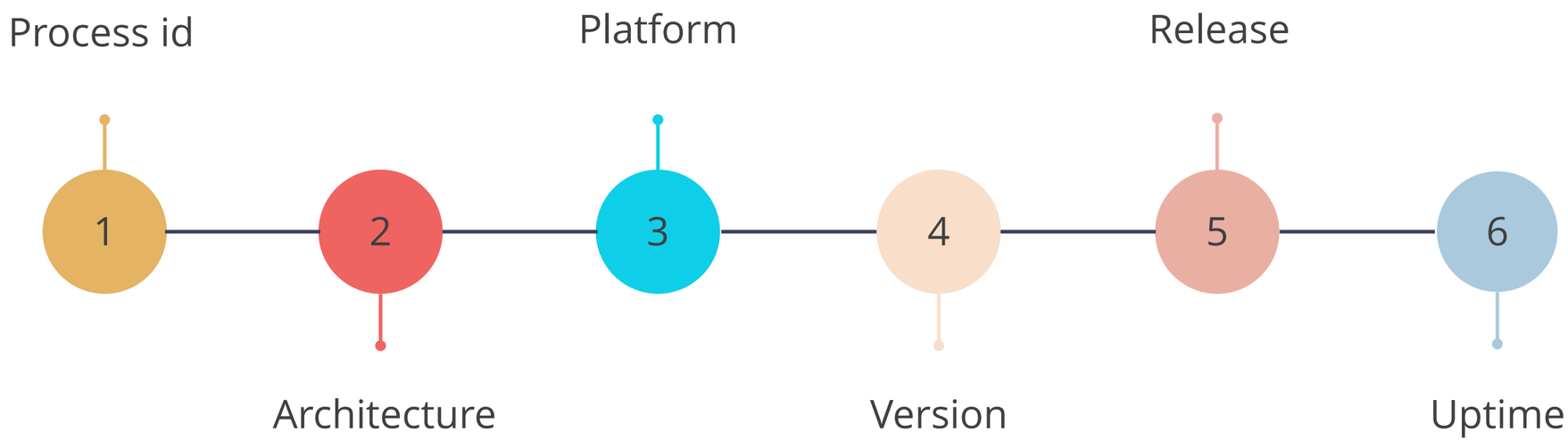
setImmediate() callbacks are invoked during this phase.

Close callbacks

Close callbacks are closed during this phase.

Node.js Process: Usage

Node.js process provides information about current objects.
It provides the facility to get the process information, such as:



It can be used to kill processes and set uids.

Controlling the Node.js Process

Node.js process is an instance of EventEmitter. It emits the following events:

1. beforeExit
2. uncaughtException
3. disconnect
4. uncaughtExceptionMonitor
5. exit
6. unhandledRejection
7. message
8. warning
9. rejectionHandled

Event: beforeExit

beforeExit is emitted when Node.js closes the event loop.

Syntax:

```
import process from 'node:process';

process.on('exit', (code) => {

  console.log(`About to exit with code: ${code}`);

});
```

The listener callback function is invoked with the value of **process.exitCode.**

Event: disconnect

Event **disconnect** is emitted when the IPC channel is closed and the Node.js process is started with an IPC channel.

Syntax:

```
addEventListener('disconnect', (event) => { });

ondisconnect = (event) => { };
```

Event: exit

In Node.js process, event exit is emitted when:

01

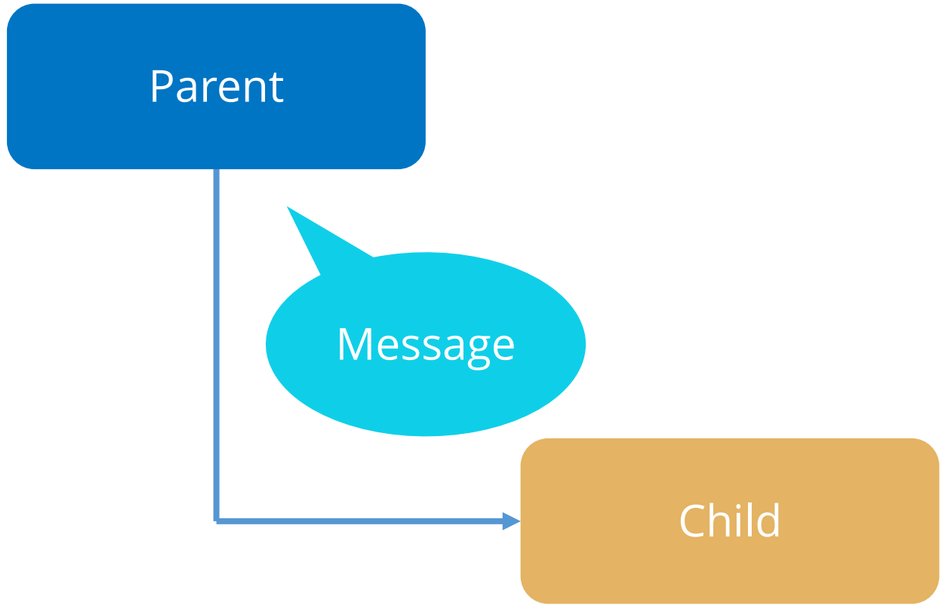
The process.exit() method is called explicitly

02

The Node.js event loop has no work to perform.

Event: message

An event message is emitted when the Node.js process is spawned with an IPC channel.



Syntax:

```
Event: 'message'
```

The child process receives messages sent by the parent process using childprocess.send().

Event: rejectionHandled

This event is emitted whenever a **promise** is rejected and an error handler is attached.

Example:

```
promise.catch()
```

Event: uncaughtException

When an uncaught JavaScript exception returns to the event loop, this event is released.

Syntax:

```
Event: 'uncaughtException'
```

By default, Node.js handles these errors by leaving with code 1 and sending the stack trace to stderr, overriding any previously configured **processes.exitCode**.

Event: uncaughtExceptionMonitor

This event is sent out before an **uncaughtException** event or the installation of a hook.

Syntax:

```
process.setUncaughtExceptionCaptureCallback()
```

Event: unhandledRejection

When a **Promise** is denied and no error handler is attached to the **Promise**, this event is generated.

Syntax:

```
new PromiseRejectionEvent(type, options)
```

Exceptions are contained as rejected promises in programming with promises.

Event: warning

When Node.js emits a process warning, this event is emitted.

Syntax:

```
Event: 'warning'
```

Similar to an error, a process warning alerts the user to exceptional circumstances.

Creating a Node Server

Problem Statement:

Duration: 15 min.

You are given a task to create a node server to obtain different types of data as a response.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

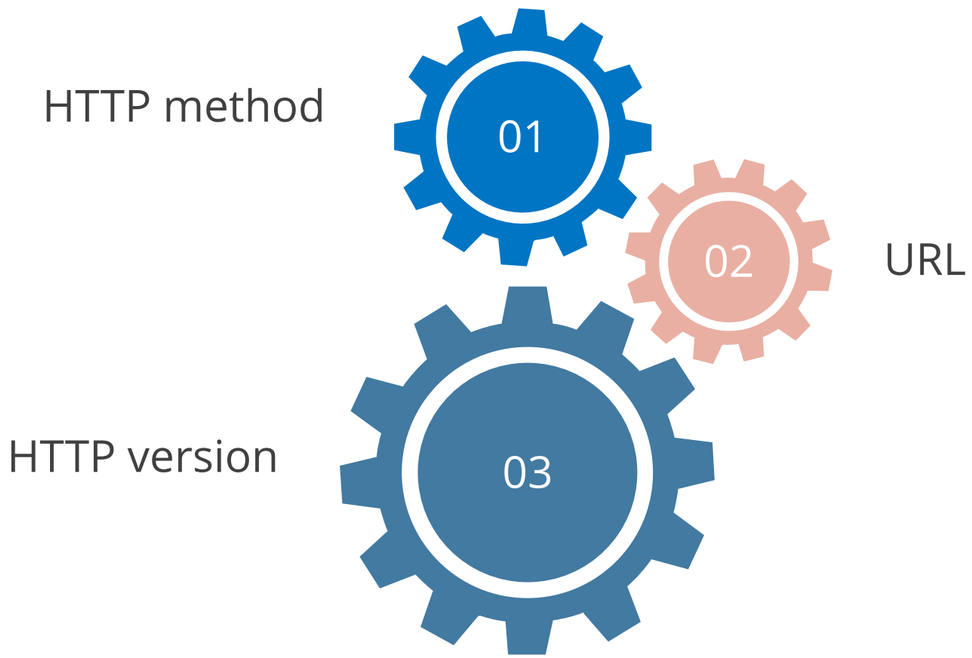
1. Create a new directory
2. Create a package.json file
3. Create an index.js file
4. Install package
5. Update package.json
6. Insert code using the HTTP Package
7. Start the server
8. Test the server on Chrome

Requests and Responses

Requests

The client sends a message to initiate an action on the server.

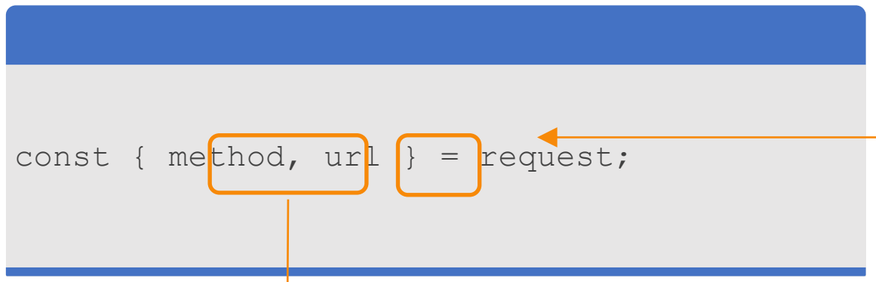
It contains three elements:



Request Object

Request object is an instance of the IncomingMessage.

The request object (req) represents the HTTP request.



The URL is complete without the server, protocol, or port.

The method is always a standard HTTP method or verb.

Request in Node

An example of how to use a request object in Node JS:

Example:

```
const http = require('http');
const server = http.createServer((request, response) => {
    // Use the request object to fetch data and perfrom the suitable
action
});
```

Request Body

Request body helps to receive a POST or PUT request.

Steps to receive a POST or PUT request:

- Pass the request object into handler to implement ReadableStream interface
- Route ReadableStream interface to another location or listen to it in the same location
- Obtain request from the ReadableStream by listening to its data and end events

Request Handler

For each HTTP request made to that server, the function supplied to createServer is called once.

```
const http = require('http');
const server = http.createServer((request, response) => {
        //...
});
```

It is called with a few objects to handle requests and responses.

Response Object

Request object belongs to WritableStream, a standard abstraction for writing streaming data to a destination, known as a sink.

Stream methods are used to write the client's response body.

Response is sent using:

```
response.send()
```

```
response.end()
```

Sending Response

res.send() is used to send HTTP response:

```
response.send([body])
```

res.end() is used to end the response process:

```
response.end();
```


This method comes from http.ServerResponse.

Sending Response as HTML

The below example shows how to use the end() to send HTML response.

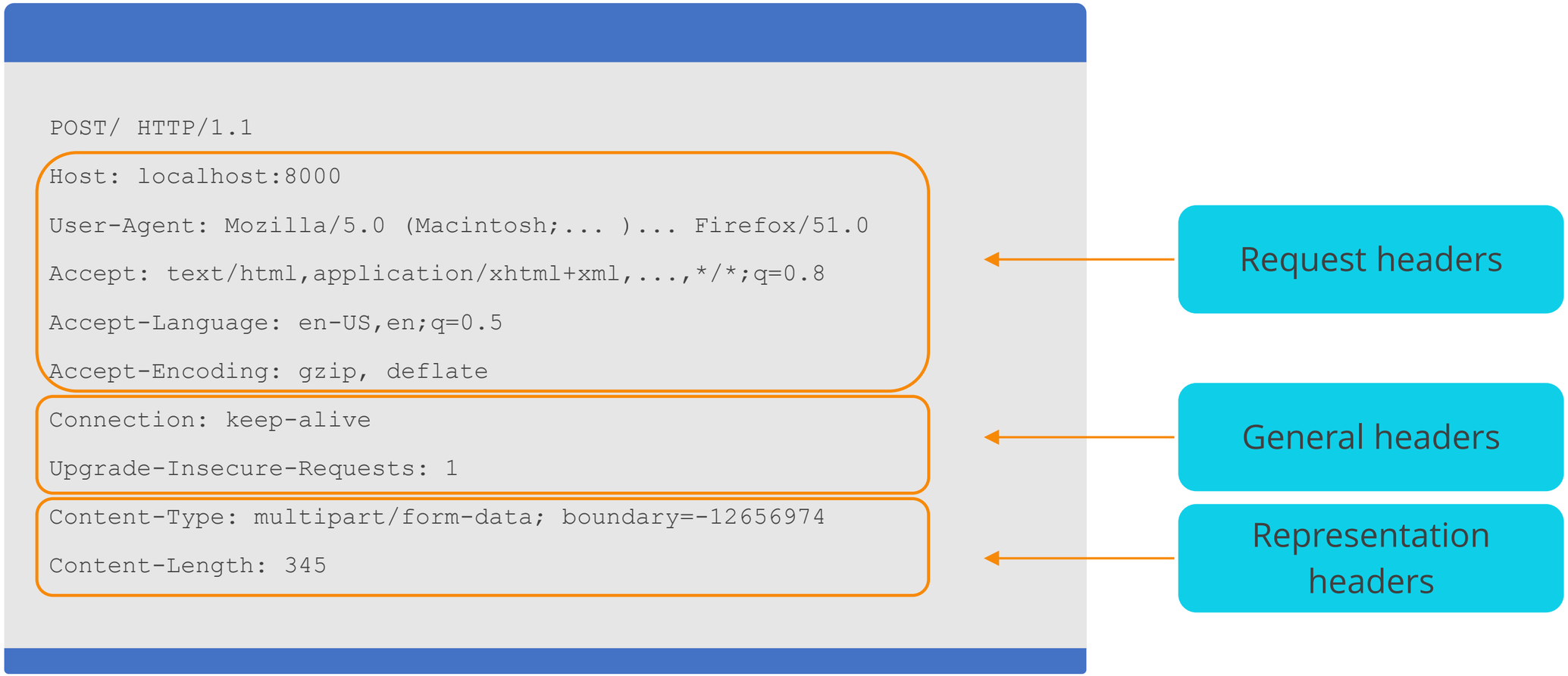
```
response.write('<html>');
response.write('<body>');
response.write('<h1>This is HTML Response :) </h1>');
response.write('</body>');
response.write('</html>');
response.end();
```

Users can simplify the usage of end() on streams as below:

```
response.end('<html><body><h1>This is HTML Response :) </h1></body></html>');
```

Request Headers

Request headers are HTTP headers used in HTTP requests to provide information to the server about the request context.



Request Headers

Request headers are available as objects known as headers within the request.

Extract user-agent from headers:

```
const { headers } = request;

const userAgent = headers['user-agent'];
```

Response Headers

Response headers are HTTP headers used in HTTP responses.

They do not relate to the content of the message.

They use age, location, or server to provide the detailed context of the response.

All headers appearing in the response are not categorized as response headers by the specification.

Response Headers: Example

The successful response to a GET request is represented by 200 OK.

```
200 OK
Access-Control-Allow-Origin: *
Connection: Keep-Alive
Content-Encoding: gzip
Content-Type: text/html; charset=utf-8
Date: Mon, 18 Jul 2016 16:06:00 GMT
Etag: "c561c68d0ba92bbeb8b0f612a9199f722e3a621a"
Keep-Alive: timeout=5, max=997
Last-Modified: Mon, 18 Jul 2016 02:36:04 GMT
Server: Apache
Set-Cookie: mykey=myvalue; expires=Mon, 17-Jul-2017 16:06:00 GMT; Max-Age=31449600;
Path=/; secure
Transfer-Encoding: chunked
Vary: Cookie, Accept-Encoding
X-Backend-Server: developer2.webapp.scl3.mozilla.com
X-Cache-Info: not cacheable; meta data too large
X-kuma-revision: 1085259
x-frame-options: DENY
```

SetHeader

setHeader method is used to set headers in the node.

```
response.setHeader('Content-Type', 'application/json’);
response.setHeader('X-Powered-By', 'bacon');
```

The names are case-insensitive.

The sent value is always the last value set.

Response Stream

Response stream is used for writing headers explicitly.

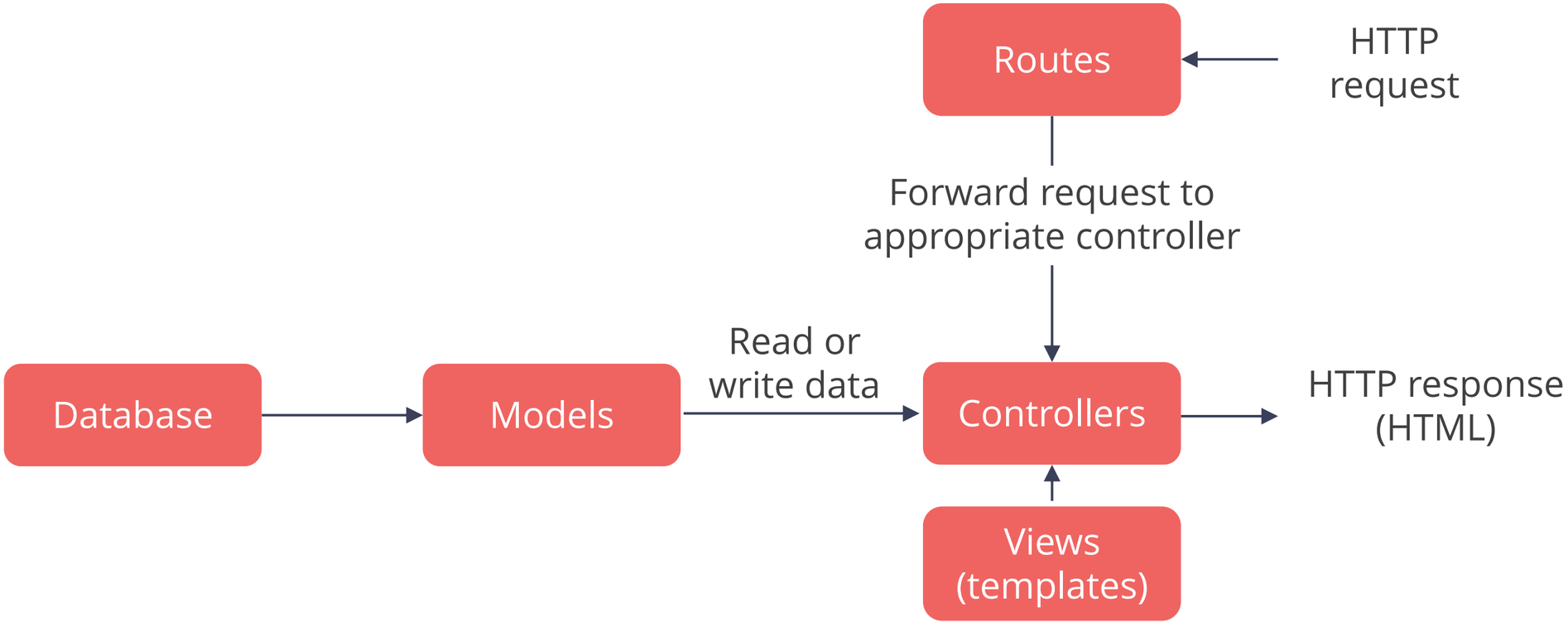
writeHead method is used to write the status code and the headers to the stream.

Example:

```
response.writeHead (200, {
          'Content-Type': 'application/json’,
          'X-Powered-By': 'bacon’
});
```

Routing

Routing means forwarding the supported requests and information encoded in request URLs to the appropriate controller functions.



Routing Request

The HTTP request is routed to execute different business rules based on the request URL, and the responses differ for each route.

Example:

```
const http = require('http');
const hostname = '127.0.0.1';
const port = 8000;
const server = http.createServer((req, res) => {
const url =req.url;
if( url === '/') {
     res.setHeader("Content-Type", "text/html");
     res.writeHead (200);
     res.end('<html><body><h1>Welcome to My App</h1></body></html>');
}
```

Routing Request: Example

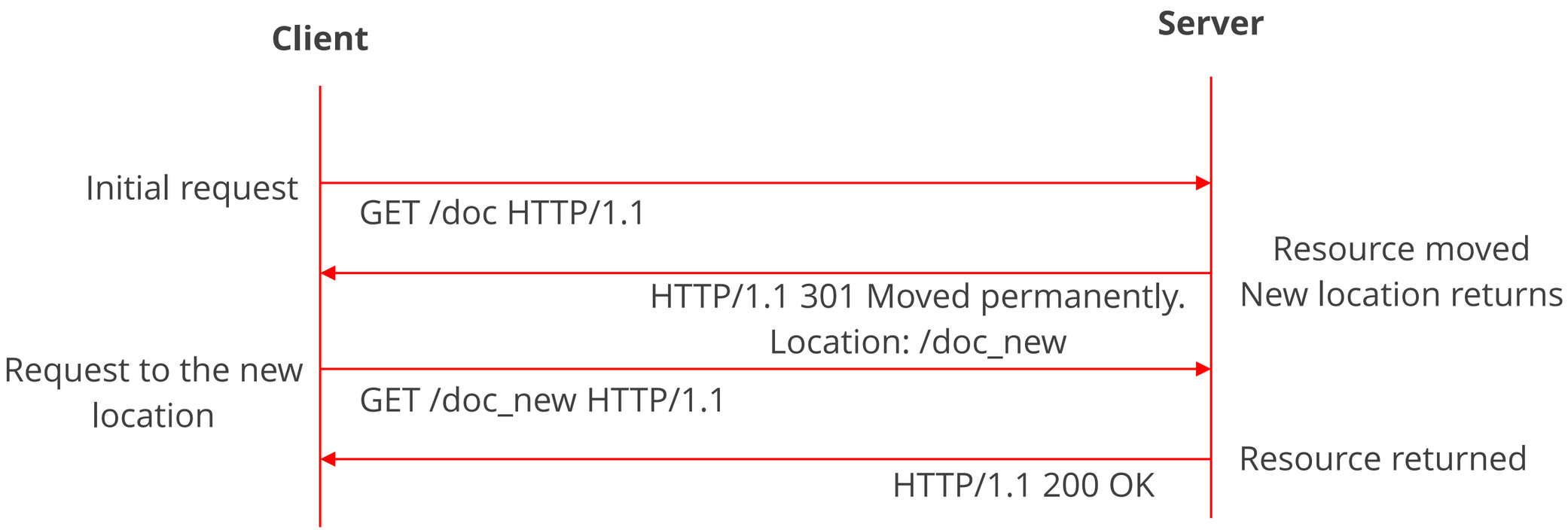
Example continued:

```
if (url === '/quote’){
    res.setHeader("Content-Type", "text/html");
    res.writeHead (200);
    res.end('<html><body><h1>Search the Candle rather than cursing the Darkness</h1></body></html>’);
    }
});
server.listen(port, hostname, () => {
console.log('Server is up and running at host '+hostname+" with port "+port);
});
```

Redirecting Requests

Redirecting requests are carried by a server that sends another redirect response to a particular request.

The status codes start with three and a location header holding the URL to redirect.



setHeader

setHeader is used to perform a redirection in Node.js.

```
if (url === '/api'){
   //redirect
   res.statusCode=302;
   res.setHeader('Location','/');
   return res.end();
}
```

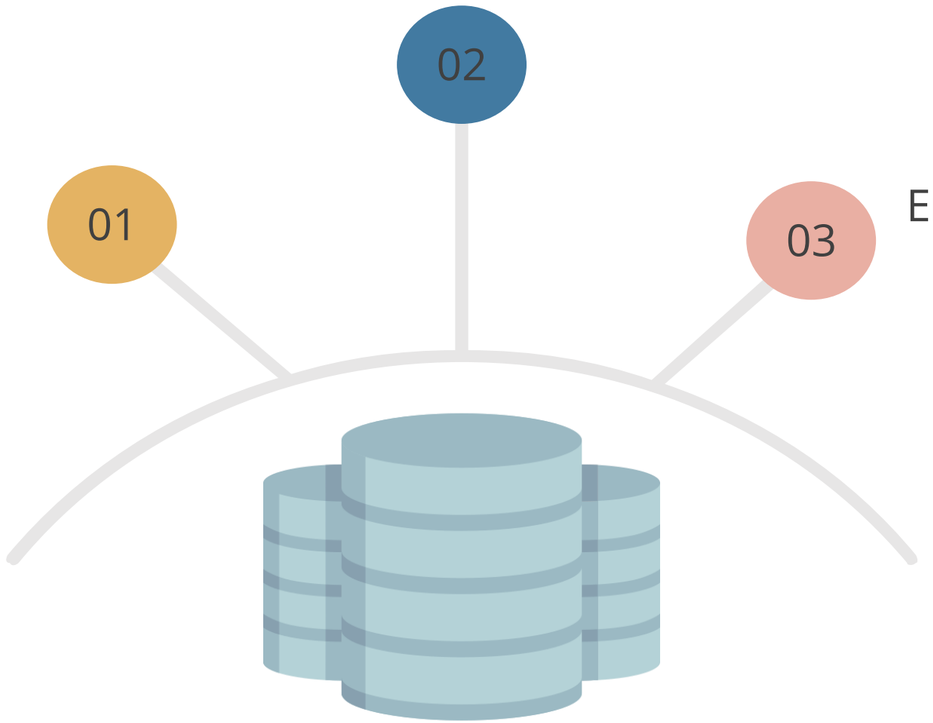
HTTP redirect status code set to 302

The location header points to the URL / where the response will be redirected to.

HTTP Request Object

HTTP request object is a readable stream in Node.js from which data can be consumed.

Data event is emitted.



Data arrives in parts or chunks in the readable stream.

End event is emitted when all parts arrive.

Readable Stream

To parse the request body in Node.js, listen to events emitted by request.

Example:

```
http.createServer((request, response) => {
const chunks = [];
request.on("data", (chunk) => {
chunks.push(chunk);
});
request.on("end", () => {
console.log("Data Received...");
const data = Buffer.concat(chunks);
console.log("Data is: ", data);
});
});
```

Responses Using Request and Response

Problem Statement:

Duration: 20 min.

You need to collect tailored responses from the server using request and response headers.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Understand requests
2. Send response
3. Request and response headers
4. Route requests
5. Redirect requests
6. Parse request bodies

Asynchronous Node.js: Part 1

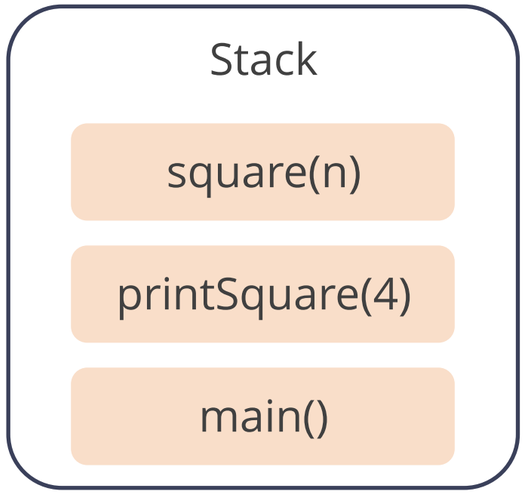
Call Stack

The interpreter uses it to keep track of multiple function calls.

For example, it tracks a function being run and called from within.

Example:

```
function multiply(a, b) {
     return a * b;
}
function square(n) {
     return multiply(n, n);
}
function printSquare(n) {
     var squared = square(n);
     console.log(squared);
}
printSquare(4);
```



How Call Stack Works?

01 The function is called by the script.

02 An interpreter carries out the function declaration.

03 All the functions are added to the call stack.

04 The called functions are then executed.

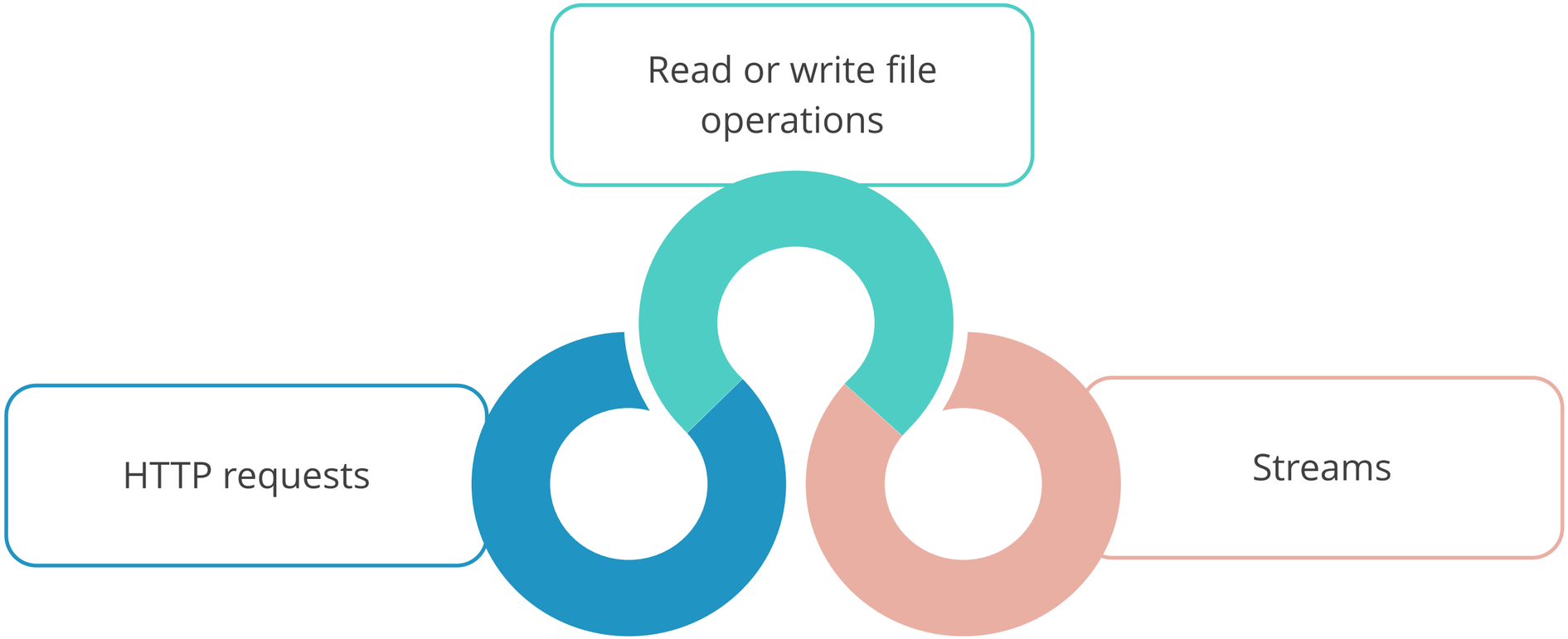
05 Once finished, the interpreter removes the current function from the stack.

If the stack occupies more space than it is allotted, it results in stack overflow errors.

Callback Queue

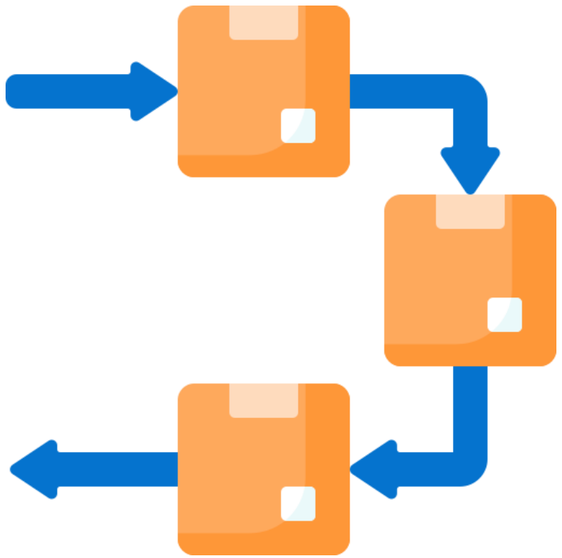
In Node.js, the queue is used to organize asynchronous operations appropriately.

Callback queue exists in different forms:



Callback Queue: Properties

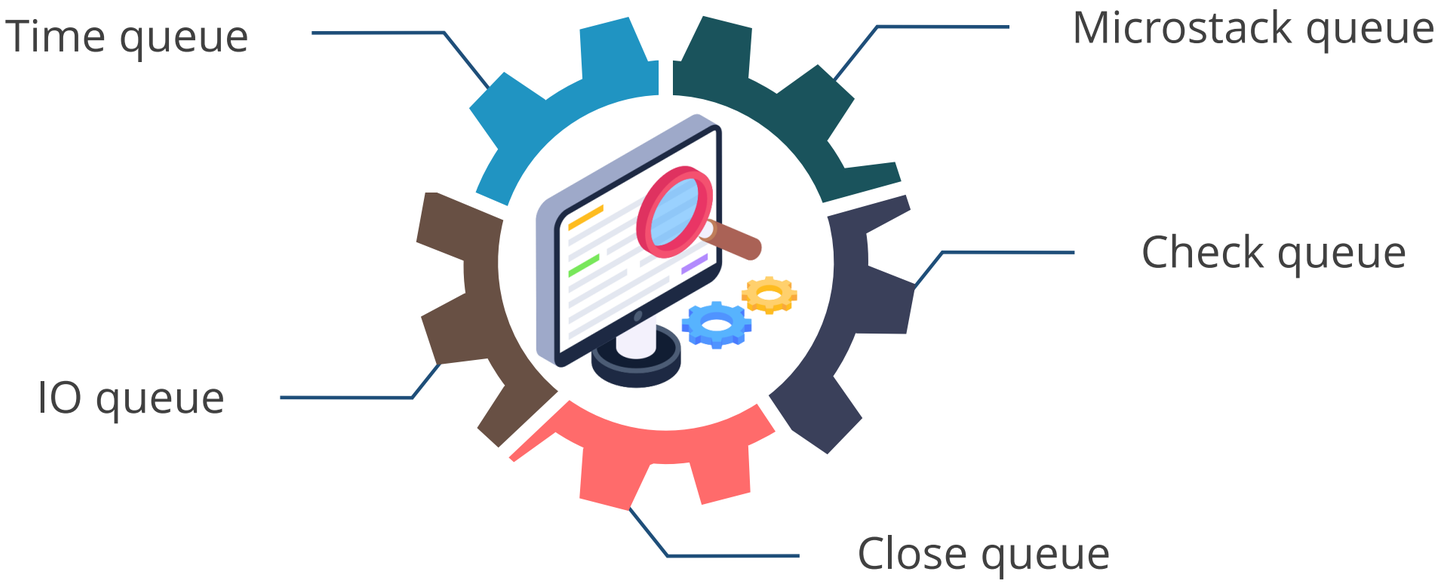
When asynchronous operations are finished in the background, they hold callback functions for those operations.



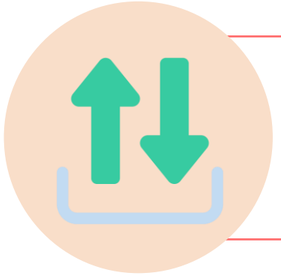
They follow the first-in, first-out (FIFO) principle.

Callback Queue: Types

Callback queue is divided into the following types:



Callback Queue: Types



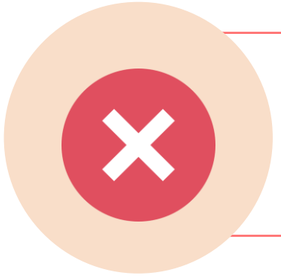
IO queue: IO operations involve external devices like the computer's internals.

Timer queue: Operations involving the timer feature of Node.js are added to the timer queue.

Microtask queue: A queue of tasks executed after the current task

Callback Queue: Types

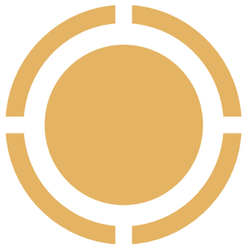
Check queue: The callback functions in this queue are executed after all such functions in the IO queue.



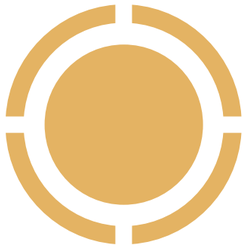
Close queue: A queue that stores functions that are associated with close event operations

Event Loop

Event loop adds a function from the callback queue to the call stack by checking for it.



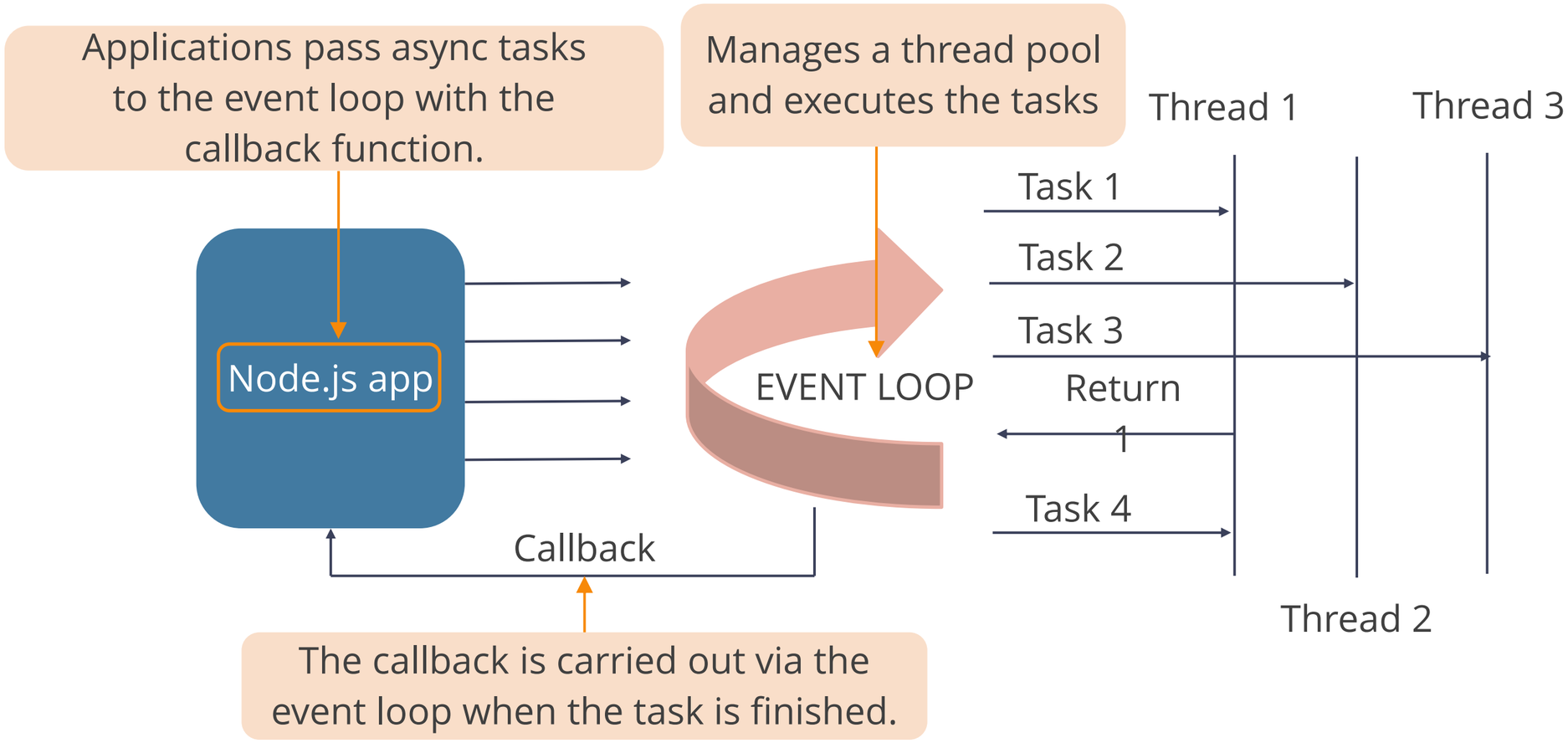
The event loop examines the queues when all synchronous operations have been executed.



It enables non-blocking I/O operations for Node.js by offloading tasks to the kernel whenever possible.

How Event Loop Works?

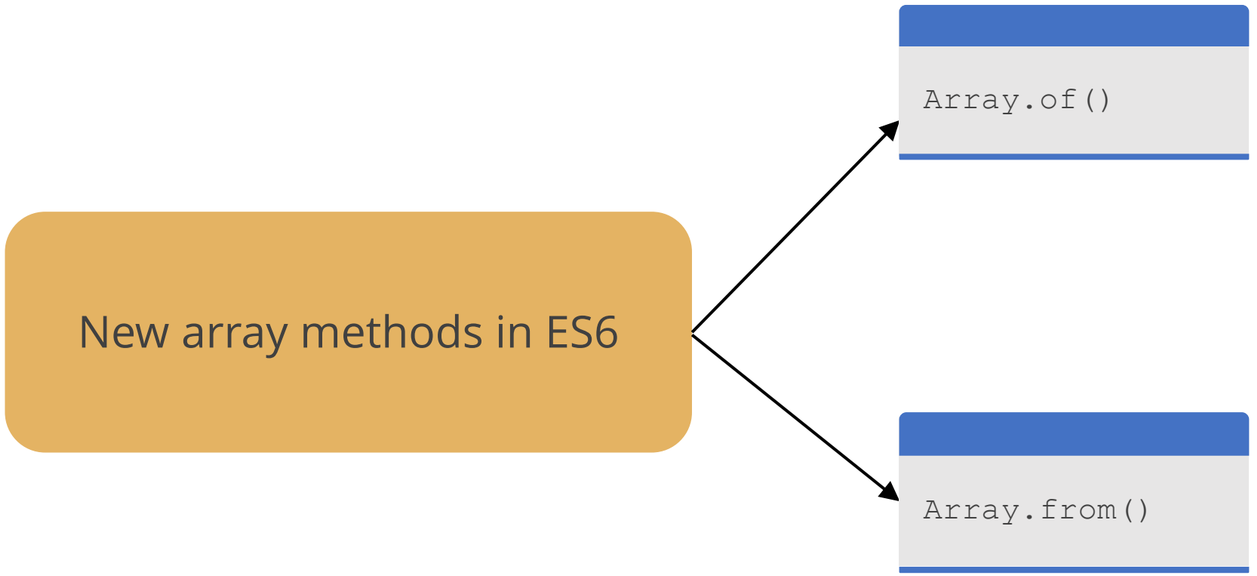
Following are the steps in which the event loop works:



ES6 Array

An array is a homogeneous collection of values.

A single variable called an ES6 array can store many elements.



Array.from()

Array.form() enables the creation of a new array from an array-like object.

It converts array-like values and iterable values like **Set** and **Map**, to arrays.

Example: Printing alphabets of a string

```
for (let element of Array.from('NodeJS')) {
console.log(element)
}
```

Array.find()

Array.find() is an iterative method.

It gets the first element back, which causes the given callback function to return true.

Example: Finding an odd number from the elements

```
var numbers = [10, 15, 20];
var result = numbers.find((x) => x % 2 == 1);
console.log(result); // 15
```

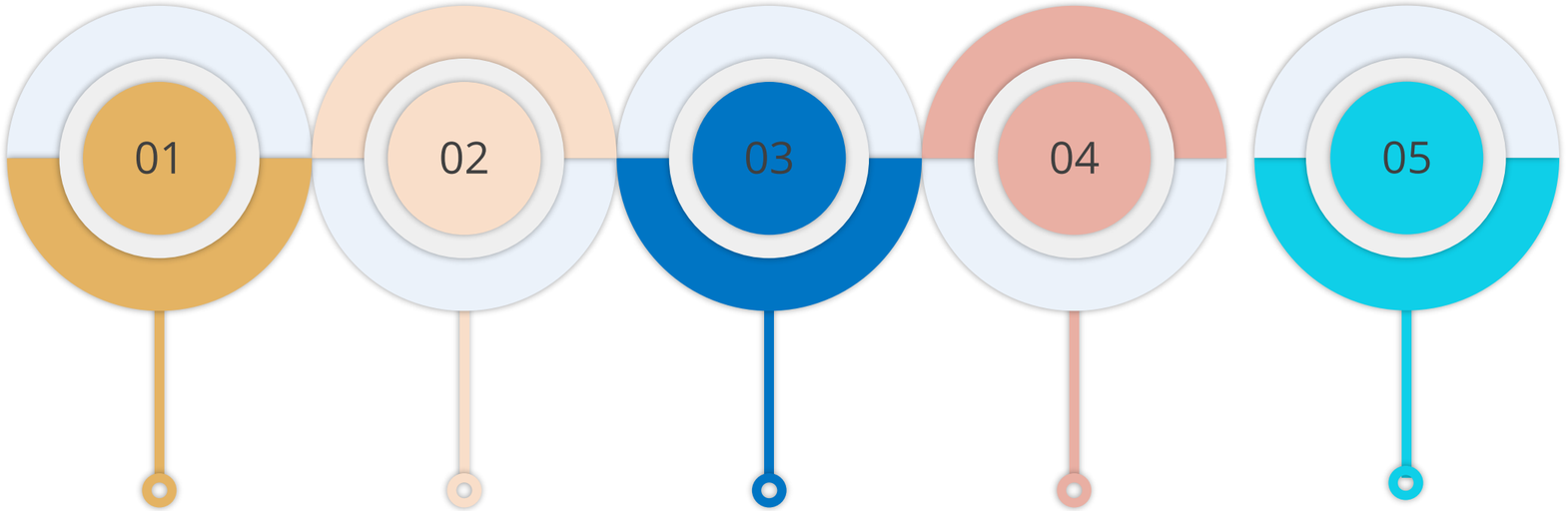
Files

A file is a container in a computer system to store information.

There are different types of files storing different information:

Files

In Node.js file system, a module can be used to perform the below tasks:



Read files

Create files

Update files

Delete files

Rename files

node:fs

node:fs facilitates communication with the file system using typical POSIX functions that include:

- Synchronous forms
- Callback forms
- Promise-based forms

They are accessed by both **CommonJS** syntax and **ES6 Modules** (ESM).

node:fs: Usage

node:fs can be used in two different ways:

Promise-based APIs

```
import * as fs from 'node:fs/promises';
```

Callback and sync APIs

```
import * as fs from 'node:fs';
```

Callback-Based Versions

If maximum performance is required, node:fs module APIs are preferred over promise APIs.

Example:

```
const { unlink } = require('node:fs');

unlink('/tmp/data', (err) => {

if (err) throw err;

console.log('successfully deleted /tmp/data’);

});
```

Callback-Based Versions: Example

Read the HTML file and return the response using the node:fs module

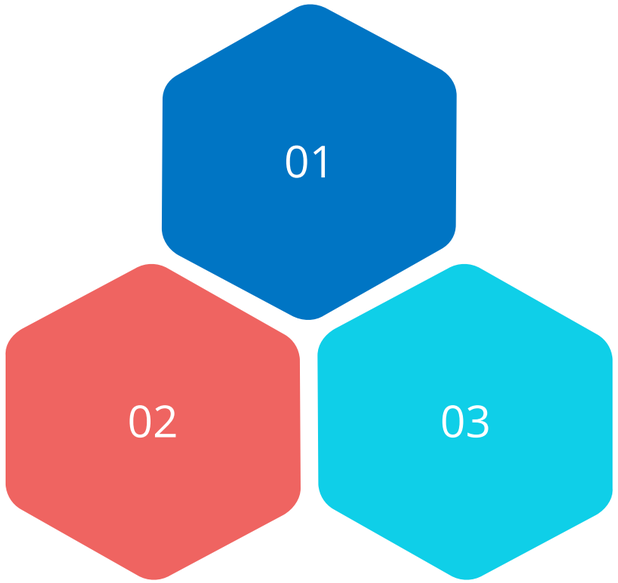
```
var http = require('http');
var fs = require('fs');
http.createServer(function (req, res) {
fs.readFile('index.html', function(err, data) {
res.writeHead (200, {'Content-Type': 'text/html'});
res.write(data);
return res.end();
});
}).listen(8000);
```

Stream

Stream is the node's I/O abstraction.

Streams in Node.js represent:

Open files



HTTP connections

stdin, stdout, and stderr

Pipes

Pipes takes a readable stream and connects itself to a writeable stream using the **stream.pipe()** method.

stream.pipe() uses HTTP request and response objects.

Pipe() Method

This method helps to pipe streams together within the node application.

Two arguments for this method are:

01

A necessary writable stream serves as the destination for the data.

02

An optional object is used to pass the option.

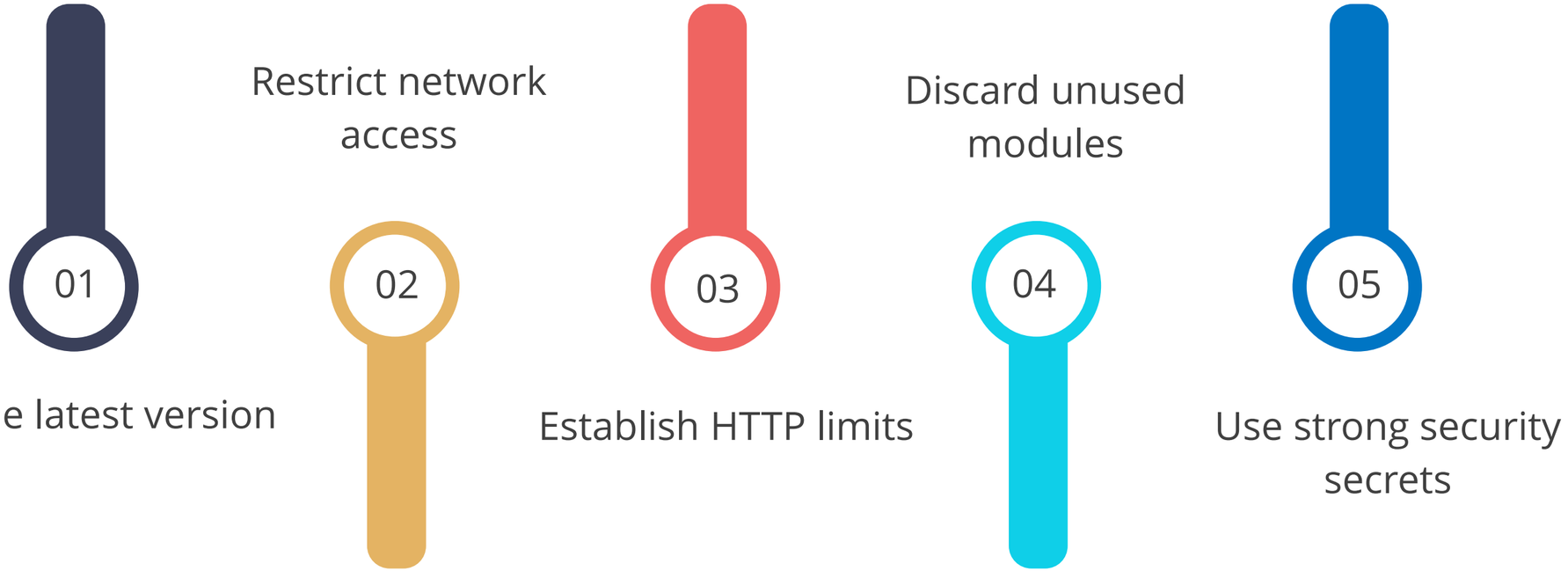
Pipes: Example

Transfer data from one file to other using the pipes:

```
var fs = require('fs');
var readStream = fs.createReadStream('/Users/john/Documents/input.txt’);
var writeStream = fs.createWriteStream('/Users/john/Documents/output.txt');
readStream.pipe(writeStream);
```

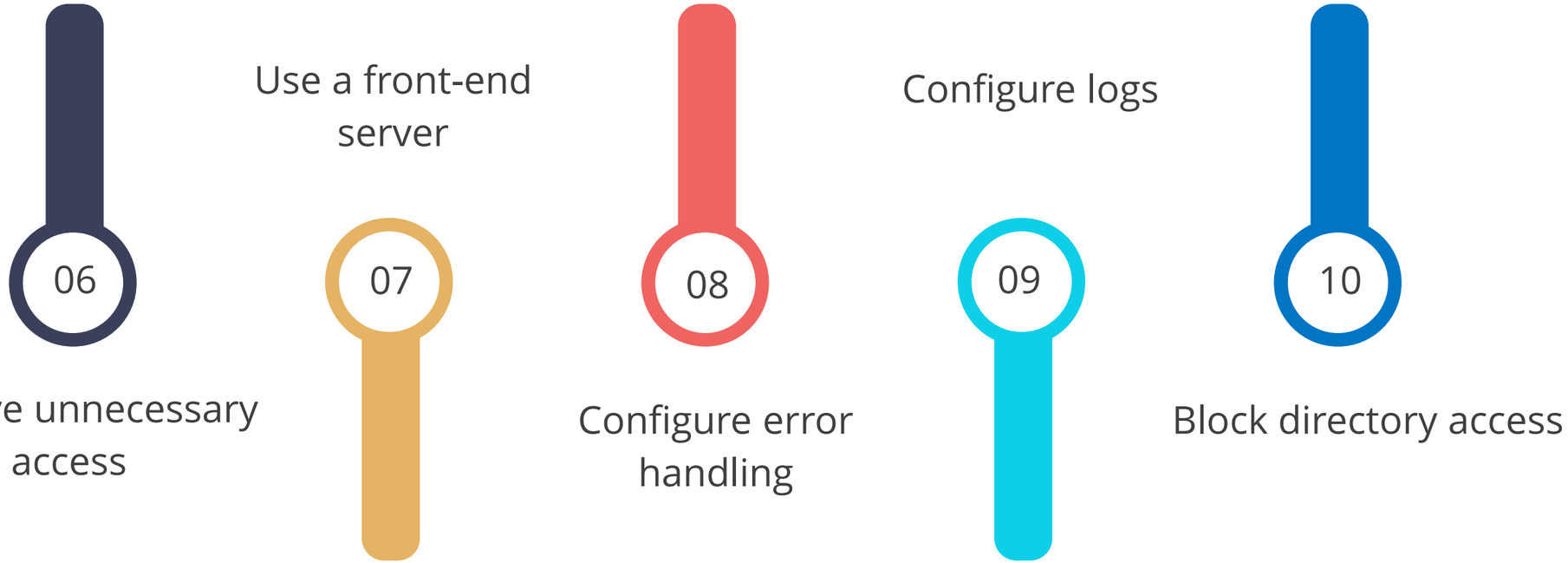
Web Server Checklist

Some features of the web server checklist are:



Web Server Checklist

Some features of the web server checklist are:



Blocking and Non-blocking Operations

Problem Statement:

Duration: 15 min.

You need to execute blocking and non-blocking operations in Node.js.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

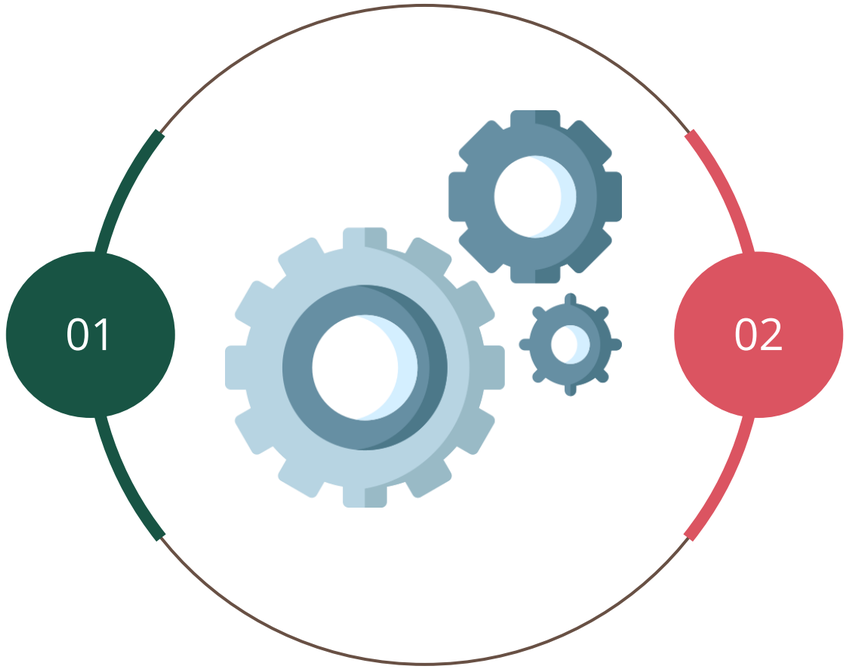
1. Create a new project
2. Import file system module
3. Implement readfilesync to understand blocking operation
4. Implement readfile to understand the non-blocking operation
5. Execute the code using the node command

Asynchronous Node.js: Part 2

Libuv

Libuv is a multi-platform support library focusing on asynchronous I/O.

It allows the CPU and the other resources to be used simultaneously while performing I/O operations.



It facilitates an event-driven approach and performs other activities using callback notifications.

Features of Libuv

Following are some of the features of Libuv:

01 Full-featured event loop

02 Asynchronous file and file system operations

03 Asynchronous TCP and UDP sockets

04 File system events

05 Asynchronous DNS resolution

Features of Libuv

Following are some of the features of Libuv:

06 Child process

07 High-resolution clock

08 Thread pool

09 Threading and synchronization primitives

10 Signal handling

Asynchronous Execution of a Program

In asynchronous execution, the program is not necessarily executed line by line.

The program moves on to the next operation without waiting for the called function to return.

Non-blocking asynchronous

- It refers to the program that does not block the execution of further operations.
- It gets executed asynchronously.

Asynchronous Execution: Example

File system module is used to demonstrate non-blocking in Node.js.

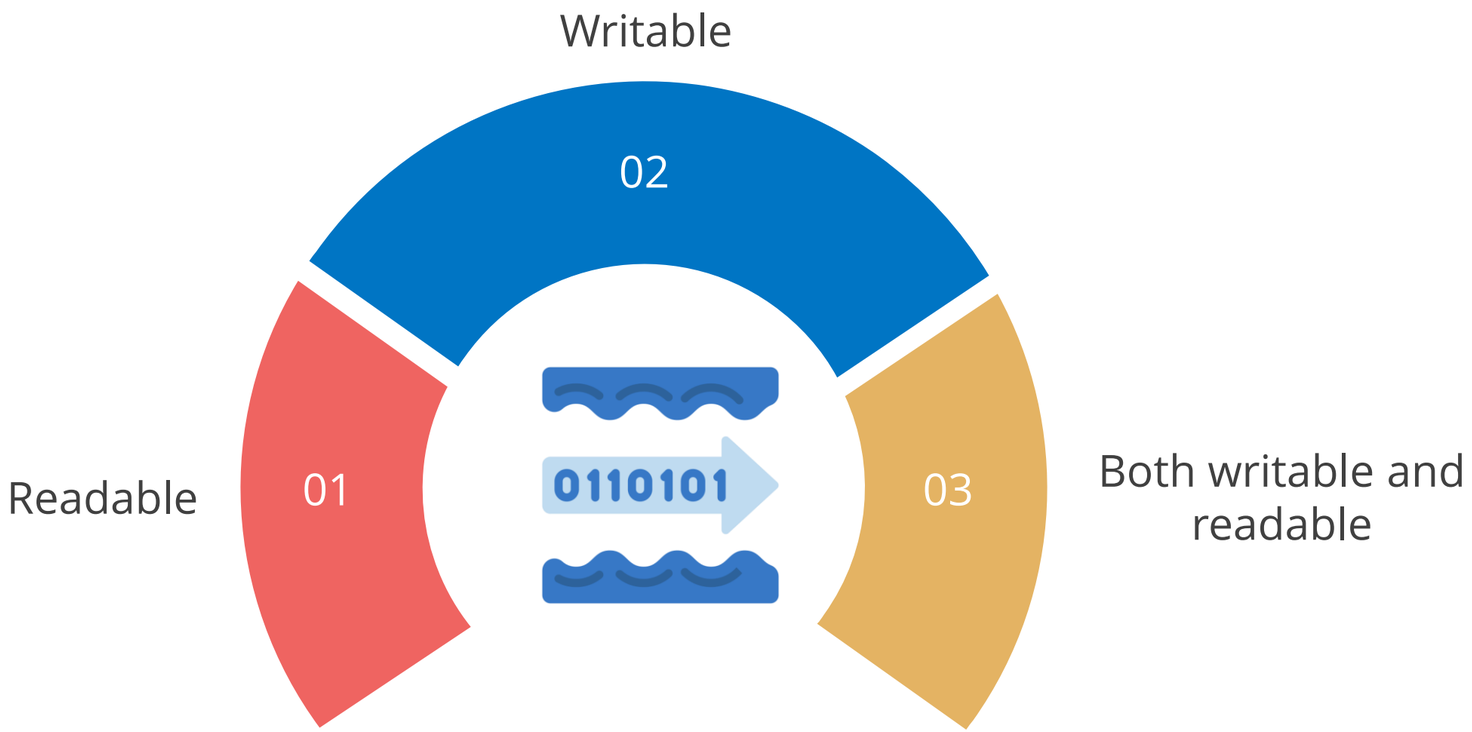
```
const fs = require('fs');
fs.readFile('/file.md', (err, data) => {
if (err) throw err;
});
```

Based on the condition, the user can decide if the error should be thrown or not.

Streams

Streams are abstract interfaces for working with streaming data in Node.js.

These can be implemented using the node:stream and are of the following types:



node:stream

node:stream can be accessed using this command:

```
const stream = require ('node:stream');
```

This helps to create new types of stream instances.

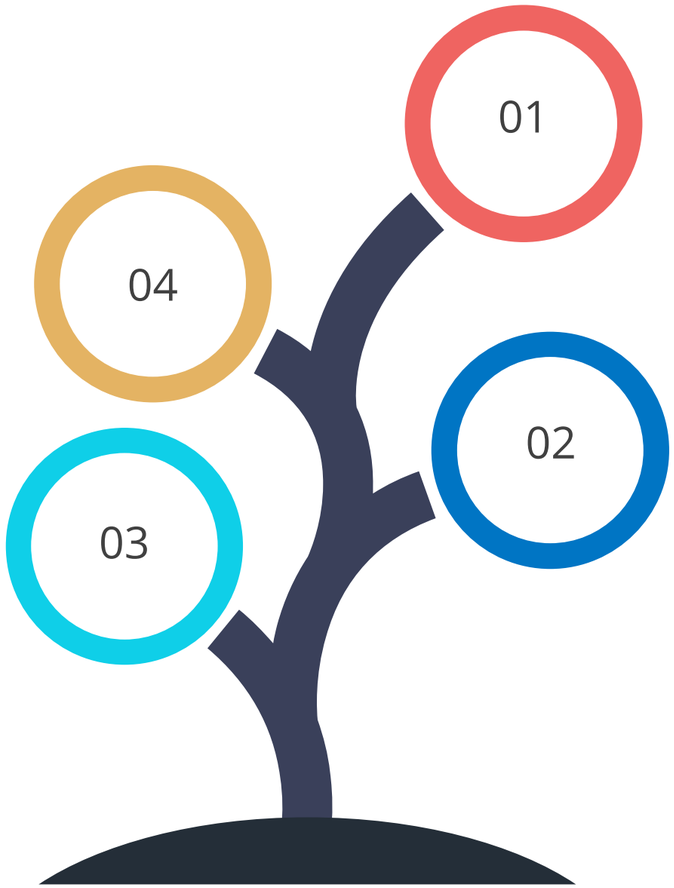
To consume streams, this module is typically not required.

Types of Streams

Four fundamental stream types are:

Transform
Modification is possible for duplex streams.

Duplex
Data permits both reading and writing.



Writable
Data allows for writing.

Readable
Data allows for reading.

Streams: Utility Functions

The streams module includes the utility functions:

```
stream.pipeline()
```

```
stream.finished()
```

```
stream.Readable.form()
```

```
stream.addAbortSignal()
```

Buffers

It is a temporary memory that streams use to store data until they are consumed.

A buffer memory in Node.js works on strings, buffers, and JavaScript objects.

To do so, set the property **objectMode** on the stream object to true.

Binary Data

Binary data is a set or collection of 1 and 0; each 1 and 0 in a set is called a bit.

The computer converts the data into a binary format to store and perform the operations.

Examples of binary data: 10, 01, 001, 1110, 00101011

Binary Data: Example

Fetching binary content from request in node:

```
http.get(url.parse('http://example.com:8000/app'), function(res) {
res.setEncoding('binary');
var data = [];
res.on('data', function(chunk) {
data.push(chunk);
}).on('end', function() {
var binaryData = Buffer.concat(data);
console.log(binaryData.toString('base64'));
});
});
```

Set encoding to respond as binary

Receive binary data in response to the request raised

Use Buffer.concat() to make a new buffer for binary data, and then convert it to the desired type

Character Sets

Character sets refer to the number of characters supported by the software and hardware in a computer.

It consists of codes, bit patterns, or numbers used to define character and works based on:

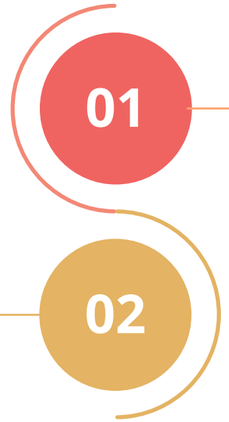
- Charset
- Character map
- Character code

Character Sets

Popular character sets used in the industry are:

Character Encoding

Character encoding determines:



How to represent a character in character set as binary data

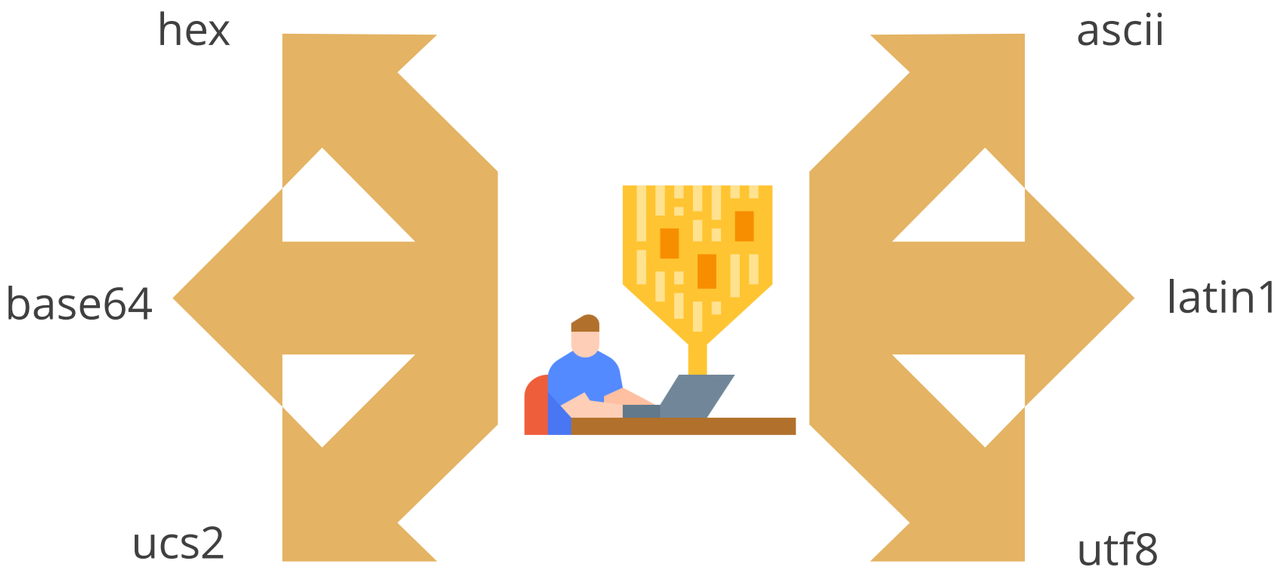
How many bits are to be used to represent a number

Example: UTF-8

setEncoding Function

setEncoding function is available in the stream module to set the encoding of character data stored inside the readable stream.

Character encodings supported by Node.js include:



Event Loop

Problem Statement:

Duration: 15 min.

You are given a task to execute event loop in Node.js to convert data into the desired type.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Create a new project
2. Import file system module
3. Implement the setImmediate function while reading the file
4. Implement the setTimeout function while reading the file
5. Execute the code using the node command

Modules

Modules

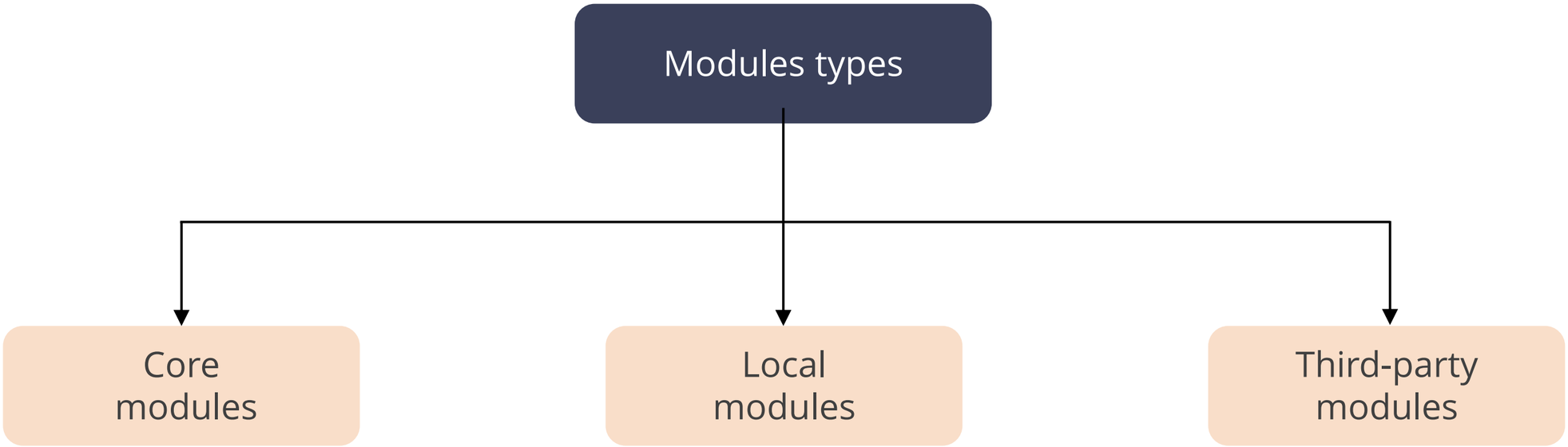
Modules are the blocks of the encapsulated code that communicate with an external application based on their related functionality.

They can be a single file or a collection of multiple files or folders.

They help in code reusability, as they can break down complex code into manageable chunks.

Modules: Types

There are three types of modules:



Core Modules

Core modules are built-in modules and can be loaded into the program using the **require** function.

Example:

```
const http = require('node:http');
```

Node.js uses two core modules, the **required** module and the **module** module, for managing module dependencies. These are available on the global scope.

Local Modules

Local modules are created locally in the Node.js application.

Example:

```
var utilsModule = require('./utils.js');
```

This module conceals functionality that is unnecessary outside of the module.

Third-party Modules

The third-party modules are available online using the Node Package Manager (NPM).

Syntax:

```
npm install module_name

OR

npm i module_name
```

Example:

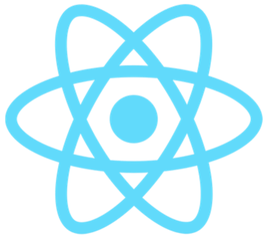
```
//single module
npm i express
npm i mysql

//multiple modules at once
npm i express mysql
```

These modules can be installed in the project folder or globally.

Third-party Modules

Some popular third-party modules are:

Mongoose



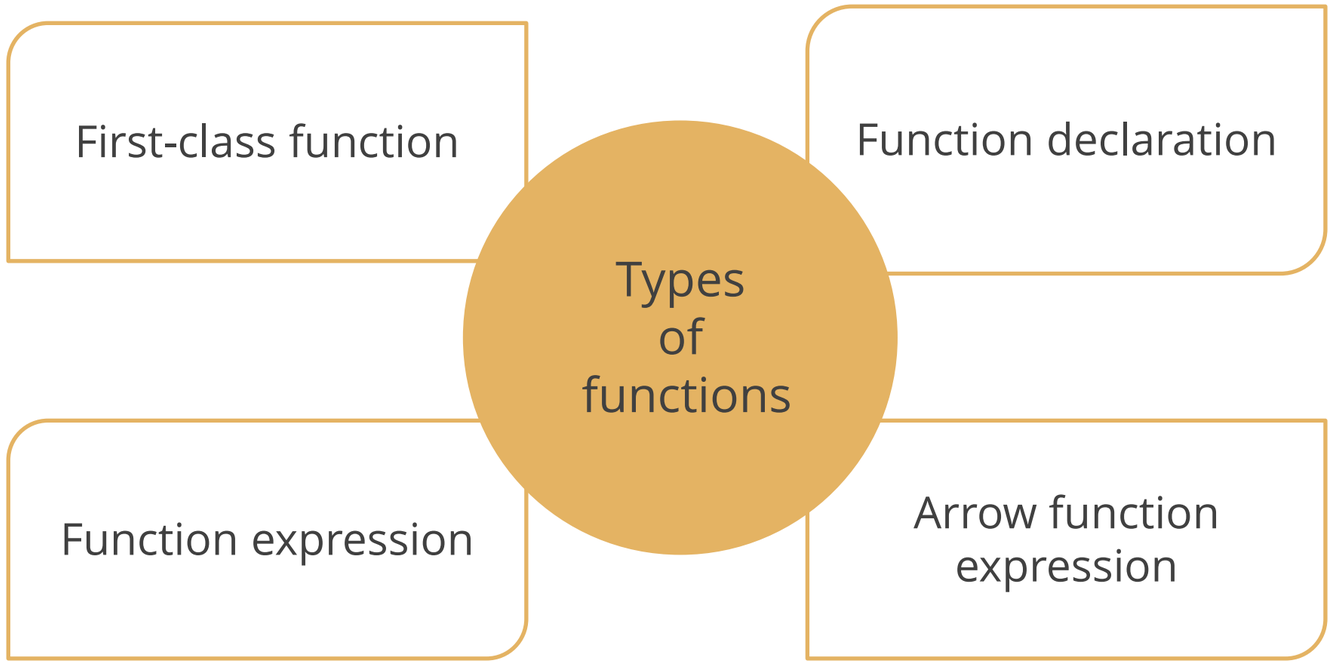
React

Express

Angular

Function

A function refers to an executable code snippet. It accepts data known as parameters.
Different types of functions used are:

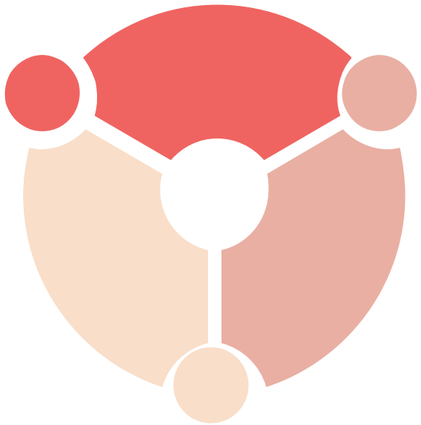


First-Class Function

First-class function in a programming language can be treated like any other variable and can be:

Passed as an argument

Assigned to any other variable



Returned by another function

JavaScript treats the function as a simple value or as a type of object.

First-Class Function: Example

The following example shows how to create first-class function:

Syntax:

```
const hello = () => {
  console.log("Hello from First Class Function");
};
hello();
```

This variable represents the function and can be used to execute it.

Function Declaration

The function declaration defines a function with the specified parameters.

Example:

```
function functionName(x,y) {
    statements……
    return (y)
};
```

A function defined with function declaration has all the properties, methods, and behavior of the function objects.

Function Expressions

Function expressions are used to define a named or anonymous function.

Example: Function Expression (Named)

```
let variableName = function
functionName(x,y){

statements…..

returns (g)

};
```

Example: Function Expression (Anonymous)

```
let variableName = function(x,y) {

    statements…..

    return (x)

};
```

The function name can be omitted in function expressions to create anonymous functions, which is the primary distinction between a function expression and a function declaration.

Arrow Function Expression

The arrow function expression is an alternative to the functional expression.

Example:

```
let variableName = (x,y) => {
    statements….
    return (g)
};
```

They cannot be used as constructors and throw a **TypeError** if called with a **new** operator.

Objects

An object is a container for named values known as properties and can be created using the **Object()** constructor.

Example:

```
const object = {
  eid: 101,
  name: 'John Watson',
  email: 'john@example.com',
  salary: 30000,
};
```

It is used to store various keyed collections and more complex entities.

Objects

A function can be created inside the objects.

Example:

```
const object = {
  eid: 101,
  name: 'John Watson',
  email: 'john@example.com',
  salary: 30000,
  getDetails : function() {
    return this.name + " " + this.email + " " + this.salary;
  }
};
```

Object Literal

The object literal is one of the most used patterns for constructing objects.
ES6 makes the object literal more concise and powerful by extending the syntax.

Example:

```
function createDish(name, description, price) {
    return {
        name,
        description,
    price
    };
}
```

Object Literal

Object literal can be constructed using local variables as well.

Example:

```
let name = 'Noodles', description = 'Veggie Hakka Noodles with Medium Spices', price = 200;


let dish = {
   name,
   description,
   price
};
```

When a property of an object has the same name as a local variable, ES6 eliminates duplication by including the name without a colon and value.

Working with Objects

Problem Statement:

Duration: 15 min.

You are given a task to work with objects.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Create a new project
2. Create the structure of an object as key-value pairs
3. Implement nested object structure
4. Implement operations like update details in the object
5. Execute the code using the node command

Prototypal Inheritance

Prototype inheritance is a technique to link a child object with the prototypes of the parent object to access the parent properties.

Example:

```
// membership object
let membership = {
     name: "prime",
     price: 1000,
     buyMembership: function () {
       console.log("Redicrecting to Payment Gateway...");
     },
};
```

Initially, the **membership object** will be used as a parent.

Prototypal Inheritance

Create another object, **superPrimeMembership**, that can be used as a child object

Example:

```
//superPrimeMembership object
let superPrimeMembership = {
     name: "Super Prime",
     price: 1500,
     coupons: 20,
     movies: 5,
     redeem: function () {
       this.coupons--;
       console.log("Redeem a Coupon...");
     },
};
```

Prototypal Inheritance

Next, perform prototype inheritance

Syntax:

```
//superPrimeMembership object inherits membership object
superPrimeMembership.__proto__ = membership;
console.log(superPrimeMembership);

// calling method from membership object using superPrimeMembership object.
superPrimeMembership.buyMembership();
```

Prototypal Inheritance

Define a membership object which has an attribute name with a value as **prime** and **buyMembership**

```
// membership object
 let membership = {
 name: "prime",
 buyMembership: function () {
  console.log("Redirecting to Payment Gateway...");
 },
};
```

Prototypal Inheritance

Now, create a **superPrimeMembership** object which has more details and a redeem attribute that reduces the coupon count

```
//superPrimeMembership object
 let superPrimeMembership = {
  name: "Super Prime",
  price: 1500,
  coupons: 20,
  movies: 5,
  redeem: function () {
    this.coupons--;
    console.log("Redeem a Coupon...");
  },
},
```

Prototypal Inheritance

Finally, assign **superPrimeMembership.__proto__** with membership to access properties of the **membership** object using the **superPrimeMembership** object

```
//superPrimeMembership object inherits membership object
 superPrimeMembership.__proto__ = membership;
 console.log(superPrimeMembership);
  // calling method from membership object using superPrimeMembership object.
 superPrimeMembership.buyMembership();
```

Prototypal Inheritance

This is the output of the previous execution:

Output:

```
{
  name: 'Super Prime',
  price: 1500,
  coupons: 20,
  movies: 5,
  redeem: [Function: redeem]
}
Redirecting to Payment Gateway...
```

Function Constructors

The Function() constructor creates a new Function object.

Directly calling the constructor creates functions dynamically at run time.

Example:

```
const taxes = new Function('a', 'b', 'return a + a*b');
console.log(taxes(100, 0.18));
// output: 118
```

Constructor Function

The constructor function is used to create objects.

Example:

```
// create a constructor function
function Dish () {
    this.name = 'Veggie Burger',
    this.price = 200
}



// create a Dish object
const dish = new Dish();
```

By Value and By Reference

Ways to execute functions are as follows:

Pass by value

- Pass by value is called directly by passing the variable's value as an argument.
- Any changes made inside the function do not affect the original value.
- The parameters passed as arguments create their copy.

Pass by reference

- Pass by reference is called directly by passing the reference or address of the variable as an argument.
- Any change of the value inside the function also changes the original value.

Pass By Value: Example

Example:

```
function swapByValue(x,y) {
   let temp;
    temp = y;
    y = x;
    x = temp;
    console.log("Inside Pass by value function -> x = '+x+'  y = '+y);
}
let x = 4;
let y = 5;
console.log('Before calling Pass by value Function -> x = '+x+'  y = '+y);
swapByValue(x,y);
console.log('After calling Pass by value Function ->  x = '+x+'  y = '+y);
```

Pass By Value: Output

Output:

```
Before calling Pass by value Function -> x = 4   y = 5
Inside Pass by value function -> x = 5   y = 4
After calling Pass by value Function -> x = 4   y = 5
```

Pass By Reference: Example

Example:

```
function swapByReference(obj) {
   let temp;
    temp = obj.y;
    obj.y = obj.x;
    obj.x = temp;
    console.log('Inside Pass by Reference function -> x = '+obj.x+'  y = '+obj.y);
}
let obj = { x: 20, y:40 }
console.log('Before calling Pass by Reference Function -> x = '+obj.x+'  y = '+obj.y);
swapByReference(obj);
console.log('After calling Pass by Reference Function ->  x = '+obj.x+'  y = '+obj.y);
```

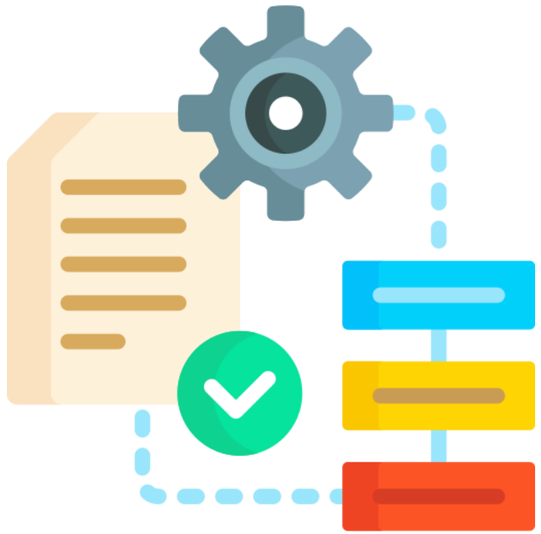
Pass By Reference: Output

```
Before calling Pass by Reference Function -> x = 20  y = 40
Inside Pass by Reference function -> x = 40  y = 20
After calling Pass by Reference_Function -> x = 40  y = 20
```

Immediately Invoked Function Expressions (IIFEs)

IIFEs are used to avoid variable hoisting from within the blocks.

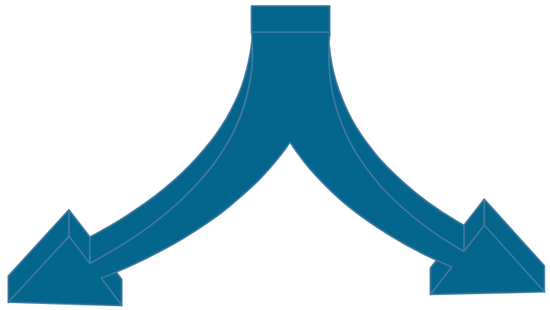
They allow the public to access methods retaining the privacy of variables.



IIFE is a design pattern known as the **Self-Executing Anonymous Function.**

Immediately Invoked Function Expressions (IIFEs)

IIFEs contains two major parts:



The first part is the anonymous function having a lexical scope enclosed within the grouping operator().

The second part creates the IIFE by which the JavaScript engine will interpret the function directly.

Immediately Invoked Function Expressions (IIFEs): Example

Syntax:

```
(function()
{
})();


(() => {/*....*/})();
```

Example:

```
(function()
{
  console.log("Welcome to company");
})();
```

Module Pattern

Require is used to include modules.

Simplest module

```
console.log('Welcome');
require('test.js');
```

Pattern 1: Define a global

```
test = function () {
console.log('this is testing
function..');
}
require('./test.js');
test();
```

Module Pattern

Pattern 2: Export an anonymous function

```
module.exports = function () {
  console.log('Check if Connected to Internet');
}
var con = require('./connectivity.js');
con();
```

Pattern 3: Export a named function

```
exports.connectivity = function() {
  console.log('Check if Connected to Internet');
}
var con = require('./connectivity.js');
con();
```

Module Pattern

Pattern 4: Export an anonymous object

```
var Connect = function() { };
  Connect.prototype.log = function() {
  console.log('connect!');
};
module.exports = new Connect();
var connect = require('./connect.js');
connect.log();
```

Creating and Testing Modules

Problem Statement:

Duration: 20 min.

You have been asked to create and test a module in Node.js.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Create a new project
2. Import test and assert module
3. Write various unit tests
4. Create subtests and skip tests
5. Implement syntax for testing
6. Execute the code using the node command

Key Takeaways

- Node.js is a cross-platform, open-source server to build scalable applications.
- Web servers using node.js can be created using the http.createServer function.
- Node.js comes with various roles and has practical usage in the industry.
- Node.js works on asynchronous and non-blocking IO concepts.

Thank You