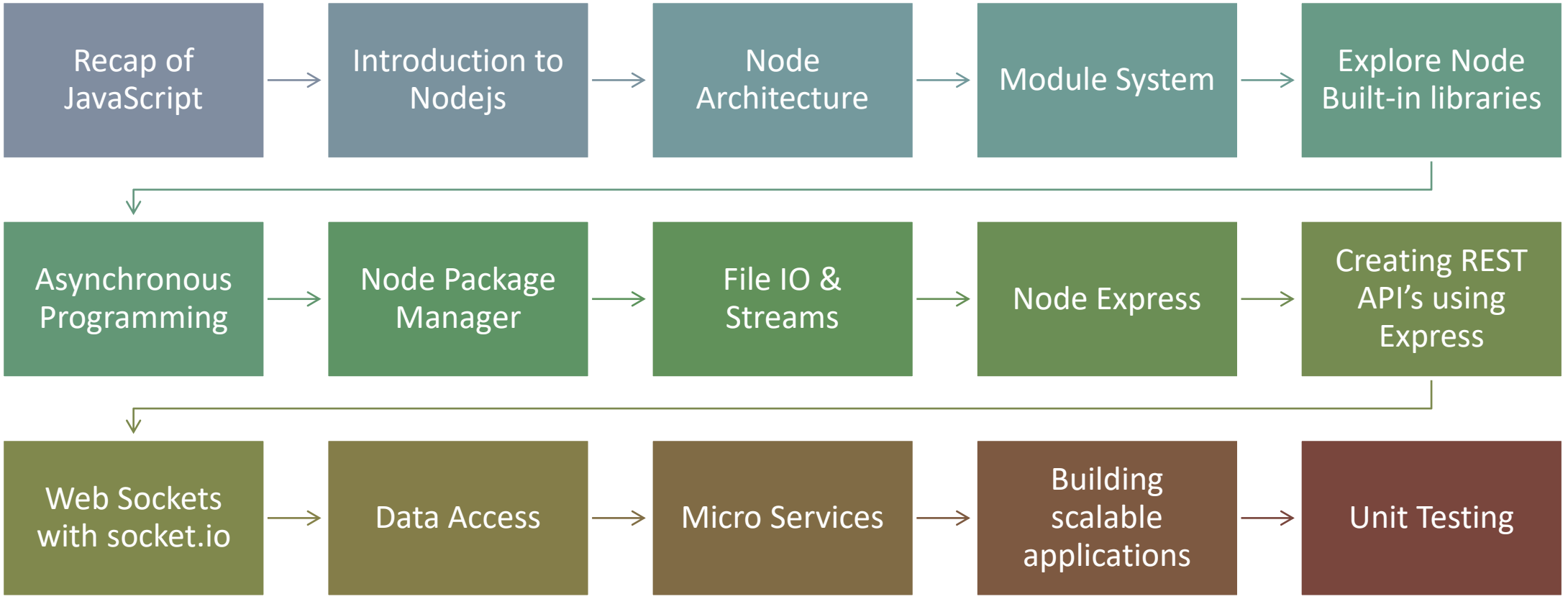
Node.js

ANIL JOSEPH

Timmins Training Consulting

Agenda


Recap of JavaScript
Introduction to Nodejs
Node Architecture
Module System
Explore Node Built-in libraries
Asynchronous Programming
Node Package Manager
File IO & Streams
Node Express
Creating REST API's using Express
Web Sockets with socket.io
Data Access
Micro Services
Building scalable applications
Unit Testing

Introduction

Anil Joseph

- Over 20 years of experience in Training and Development
- Technologies
  - C++
  - Java
  - .NET and .NET Core
  - Node, Node Express and other frameworks
  - UI Technologies: React, Angular, ExtJS
  - Mobile: Native Android, React Native, Xamarin
- Worked on numerous projects
- Conducted training for corporates(700+)

Software

Nodejs

Visual Studio Code

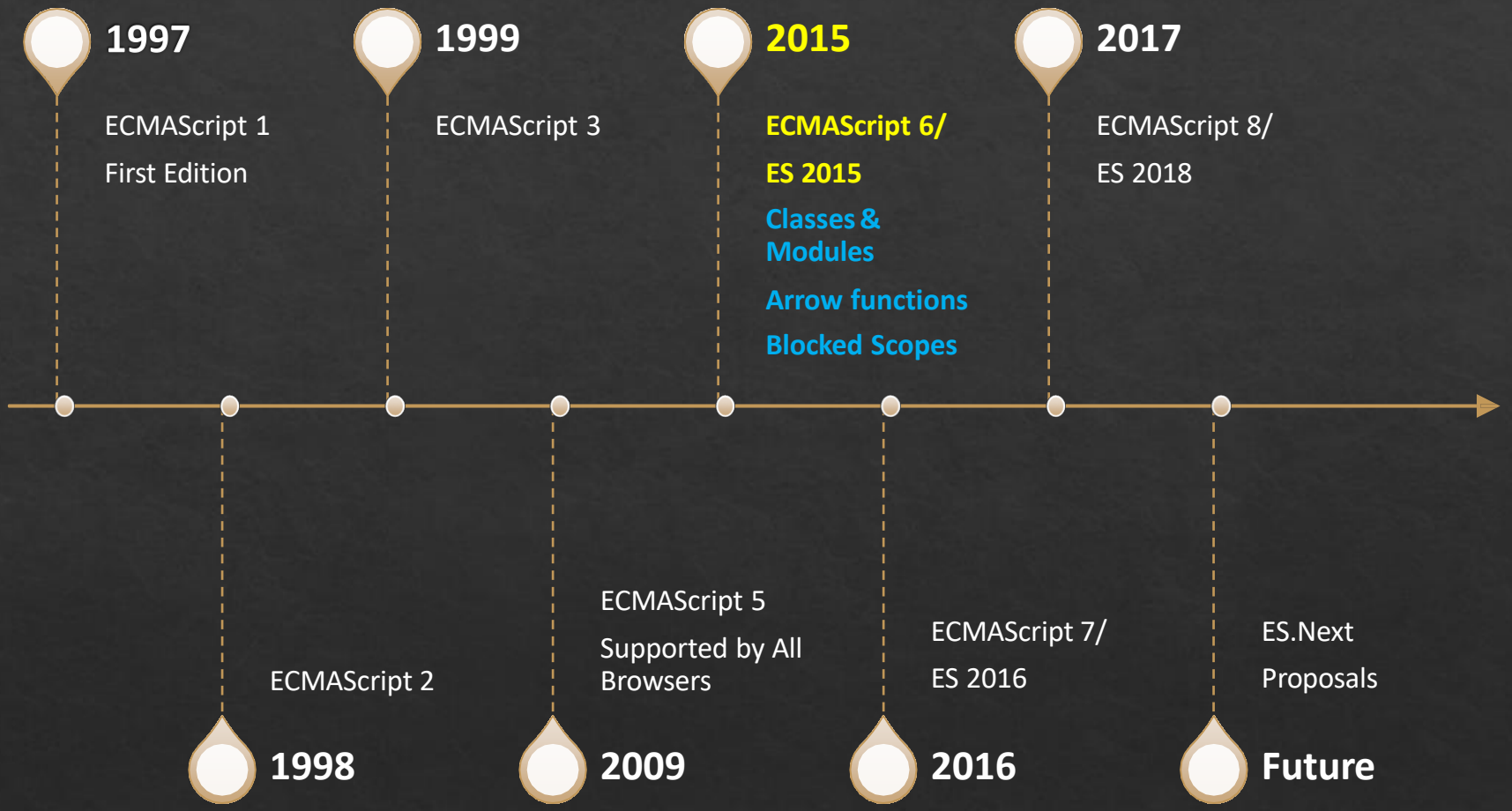
Postman

MongoDB

JavaScript

- An interpreted language
- Dynamic
- Object-oriented
- Supports Functional style of programming
- Available on the browsers and Node.js

ECMAScript Versions



Getting Started

- Install Node
  - https://nodejs.org/en/
- Installs Node and NPM
- Verify by opening a command prompt and executing the command
  - node –version
- To execute a JavaScript program
  - node hello.js

Primitive Data Types

Number	• Floating Points, Decimals and Integer
String	• Sequence of characters
Boolean	• Logical: true/false
Undefined	• Not Initialized
Null	• Absence of an object value
Symbol	• A unique identifier

Non-Primitive Data Types

Object	• Collection of key-value pairs
Function	• A type of object that can be invoked(called)
Array	• A ordered collection of values
Date	• Represents Dates and Times
Regular Expression	• Represents regular expressions.

Scopes

- Scope of a variable determines where it can be accessed and modified. There are several types of scope:
- Global
  - Variables declared outside of any function or block have global scope. They can be accessed from anywhere in the code.
- Function
  - Variables declared inside a function using var are scoped to that function. They cannot be accessed outside the function.
- Block(ES6)
  - Variables declared inside a block (i.e., a pair of curly braces {}) using let or const are scoped to that block. They cannot be accessed outside the block.

Scopes

- Lexical
  - JavaScript uses lexical scoping, which means that the scope of a variable is determined by its location within the source code, and nested functions have access to variables declared in their outer scope.
- Module(ES6)
  - When using ES6 modules, variables declared inside a module are scoped to that module and are not accessible outside it unless explicitly exported.

Hoisting

- Hoisting is a JavaScript behavior where variable and function declarations are moved to the top of their containing scope during the compilation phase, before the code is executed.
- This means that you can use variables and functions before they are declared in the code,
- Depending on whether you use var, let, const, or function declarations, the behavior changes.

- Variables declared with var are hoisted to the top of their function or global scope and are initialized with undefined.
- Variables declared with let and const are hoisted to the top of their block scope, but are not initialized.

global Object

The global object is unique to the Node.js environment and represents the global namespace.

Provides access to globally available objects, functions, and variables across all Node.js modules.

Includes properties like console, process, Buffer, and timer functions (setTimeout, setInterval, etc.).

Global Scope

- Variables attached to the global object can be accessed anywhere in the Node.js application, similar to the window object in browsers.

Global Scope

```
// Defining a global function
global.myGlobalFunction = function() {
    console.log("This is a global function");
};

// Calling the global function
myGlobalFunction(); // Outputs: This is a global function

// Defining a global variable
global.myGlobalVariable = "I am global";
console.log(global.myGlobalVariable); // Outputs: I am global

// Accessing built-in global properties
console.log(global.process.version); // Outputs the Node.js version
console.log(global.setTimeout); // Outputs the setTimeout function
```

Global and globalThis

global is specific to the Node.js environment.

globalThis is part of the JavaScript standard.

Functions

- Functions are one of the fundamental building blocks in JavaScript.
- They allow you to encapsulate code into reusable blocks that can be called and executed from different parts of your program.
- Functions can take input, process it, and return output.

Functions

- Function declaration
  - A function declaration defines a named function. It is hoisted, meaning it can be called before it is defined in the code.
- Function Expressions
  - A function expression defines a function inside an expression and can be named or anonymous. It is not hoisted, so it cannot be called before it is defined.
- Arrow functions(ES6)
  - Arrow functions are a shorter syntax for function expressions and do not have their own this, arguments keywords. They are always anonymous.

Functions as objects

- First-Class Objects:
  - Functions can be assigned to variables, passed as arguments, and returned from other functions.
- Properties and Methods:
  - Functions can have properties and methods, such as call, apply, and bind.
- Higher-Order Functions:
  - Functions that take other functions as arguments or return functions.
- Function Properties:
  - Functions have properties like length and name.
- Constructors:
  - Functions can act as constructors when used with the new keyword.

“this” keyword

- The **“this”** keyword in JavaScript behaves differently depending on the context in which it is used.
- It essentially refers to the object that is executing the current function.
- Understanding how this works in different contexts is crucial for writing effective JavaScript code.

Contexts of “this”

- Global Context: this refers to the global object (window in browsers).
- Function Context: this refers to the global object in non-strict mode and undefined in strict mode.
- Method Context: this refers to the object the method belongs to.
- Constructor Context: this refers to the newly created instance.
- Arrow Function Context: this is inherited from the surrounding lexical context.
- Event Handler Context: this refers to the element that received the event.
- Explicit Setting: Use call, apply, or bind to explicitly set this.

Closures

- A closure is a fundamental concept in JavaScript that allows functions to "remember" the environment in which they were created.
- Closures enable functions to access variables from an outer function even after the outer function has returned.
- This is possible because the inner function maintains a reference to its lexical environment.
- **A function that retains access to its lexical scope, even when that function is executed outside its lexical scope.**

Closure

```
function outerFunction() {
  let outerVariable = 'I am outside!';

  function innerFunction() {
    console.log(outerVariable); // Accesses outerVariable from the outer scope
  }

  return innerFunction;
}

const closure = outerFunction();
closure(); // Output: I am outside!
```

Objects

- Objects are fundamental building blocks for representing real-world entities, data structures, and more complex constructs.
- An object is a collection of key-value pairs where each key is a string (also called a property) and each value can be any data type, including another object, function, array, etc.

Creating Objects

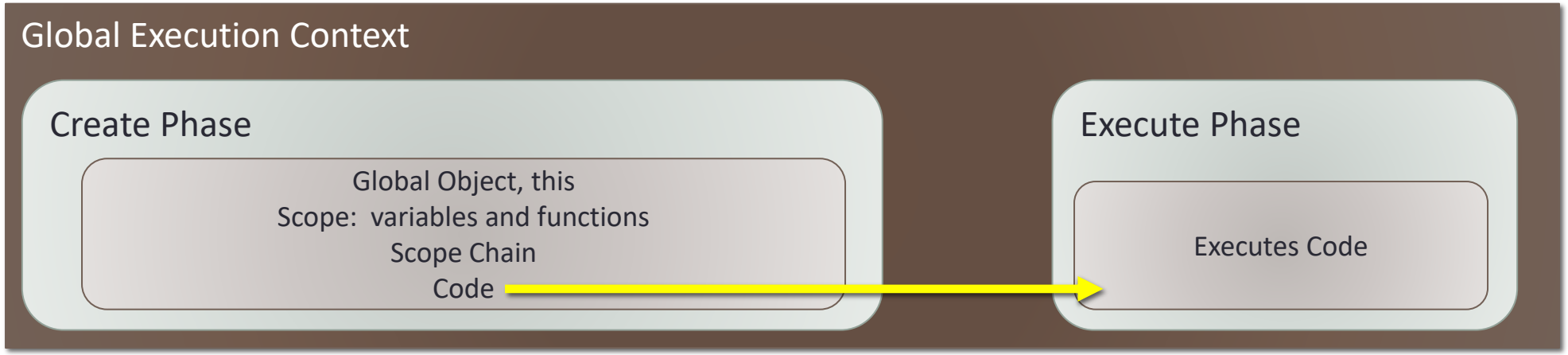
- Object Literals:
  - Define objects directly with {}.
- Constructor Functions:
  - Use functions with new to create objects.
- ES6 Classes:
  - Provide a clear syntax for creating and managing objects.
- Object.create Method:
  - Create objects with a specific prototype.

Execution Context

Execution context (EC) is defined as the environment in which JavaScript code is executed.

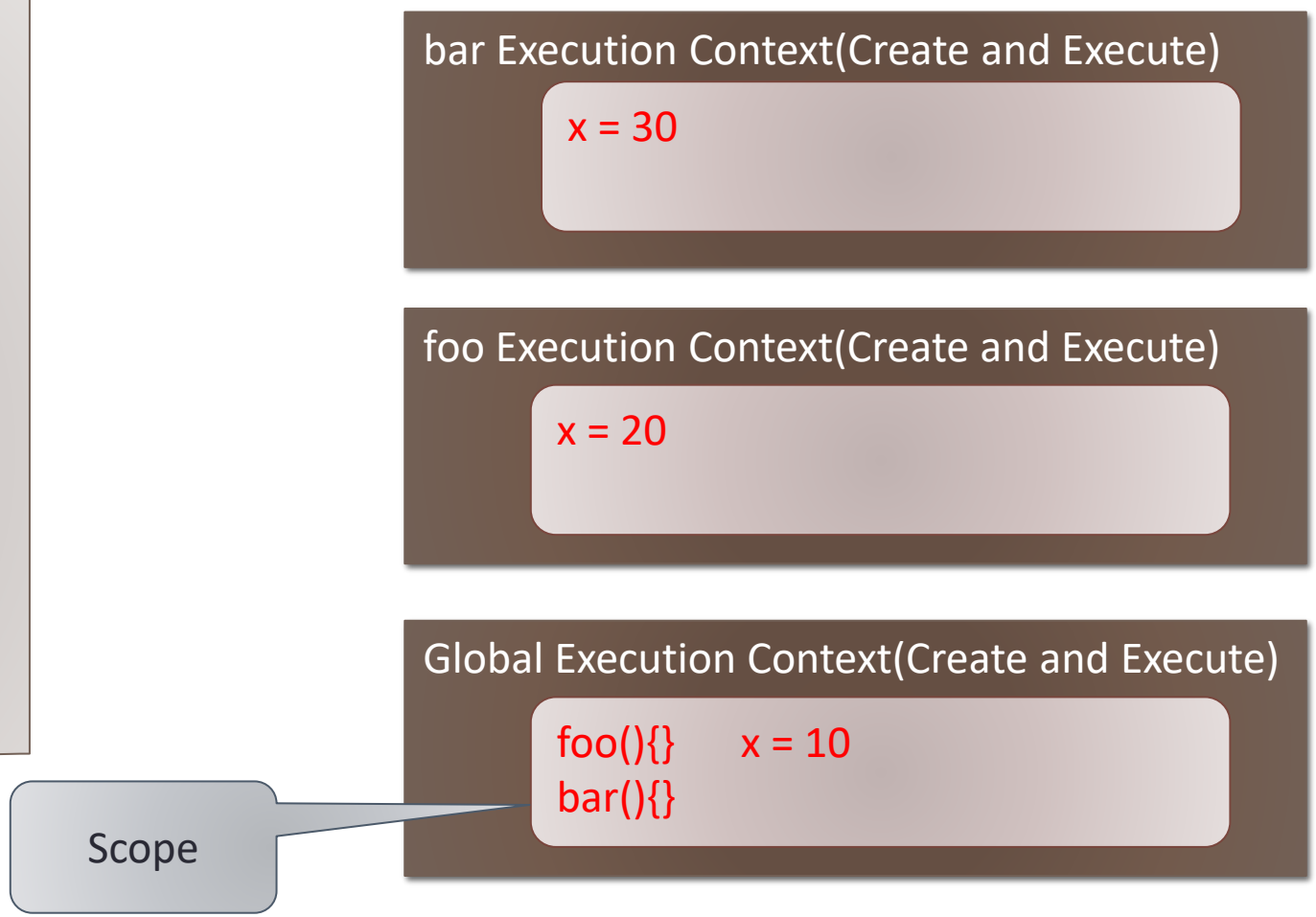
Execution context in JavaScript are of two types.

- **Global execution context**
- **Functional execution context**



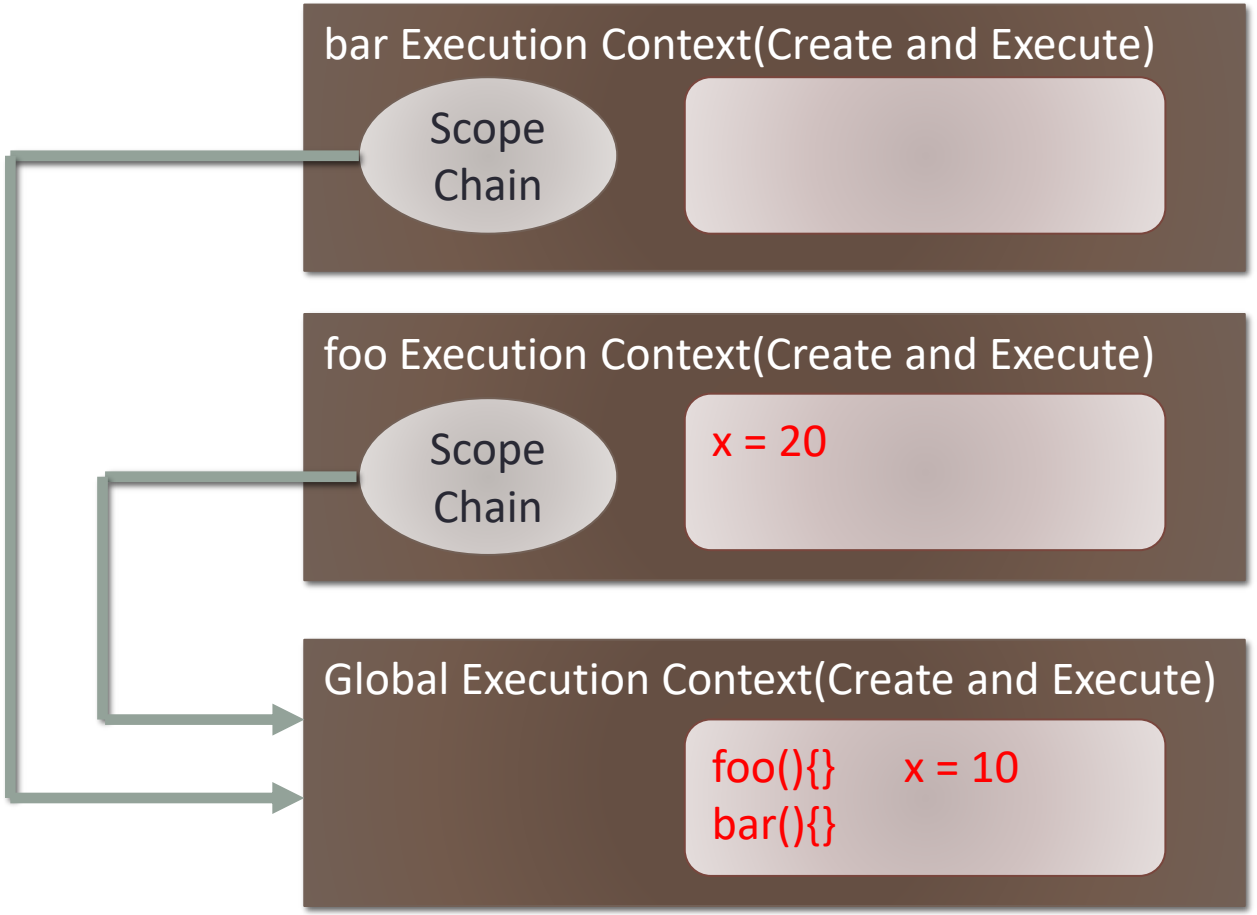
Execution Context

```
function foo(){
      var x = 20;
      bar();
}

function bar(){
      var x = 30
}

var x = 10;
foo();
```



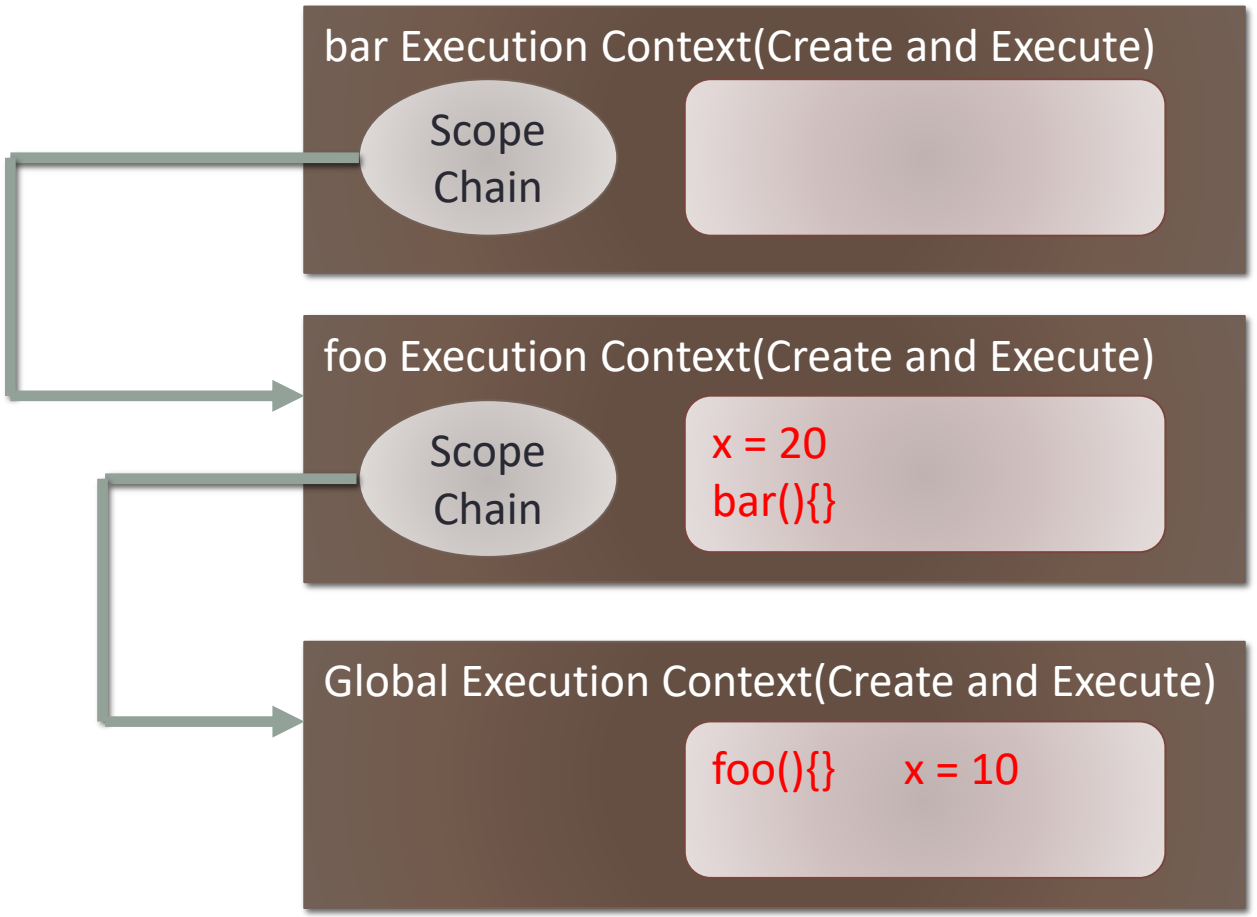
Execution Context

```
function foo(){
      var x = 20;
      bar();
}

function bar(){
            console.log(x);
}

var x = 10;
foo();
```



Execution Context

```
function foo(){

        function bar(){
                console.log(x);
        }

    var x = 20;
    bar();
}


var x = 10;
foo();
```



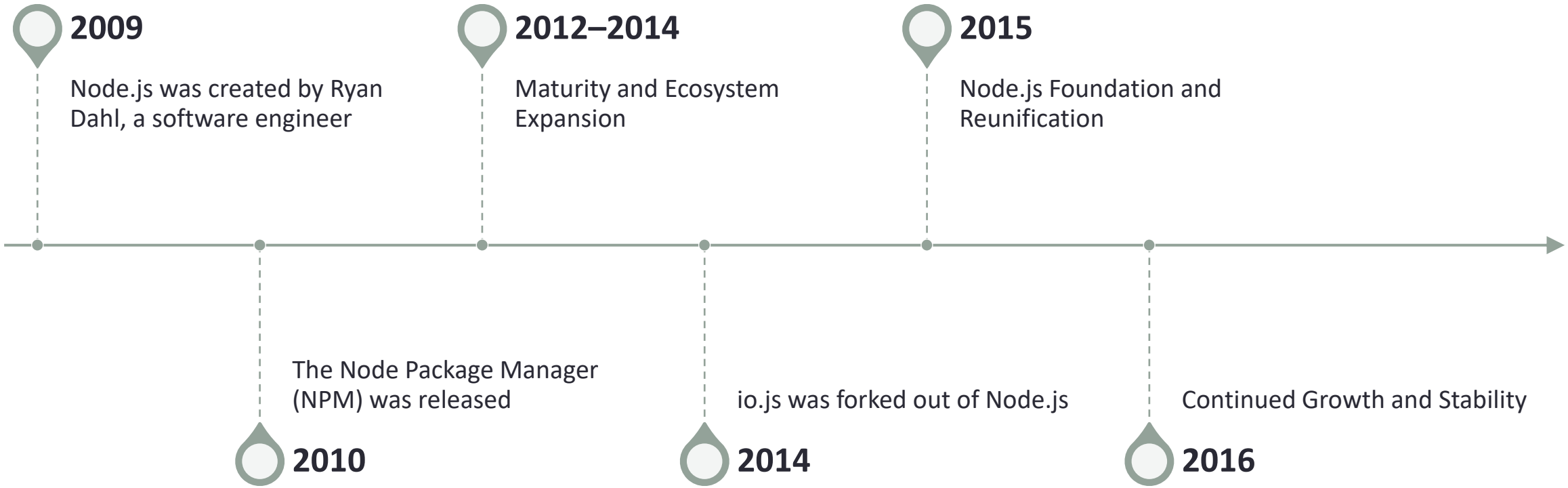
Node.js

JavaScript Runtime: Node.js is a powerful runtime for executing JavaScript on the server-side.

Non-blocking I/O: Utilizes an event-driven, non-blocking I/O model for handling multiple concurrent connections efficiently.

Scalable and Fast: Ideal for building scalable network applications, real-time systems, and APIs with high performance and low resource consumption.

Node.js History



Node.js vs Traditional Server-side Programming

- Traditional Server-Side Programming
  - **Multi-threaded**: Most traditional server-side frameworks (e.g., Java, .NET) use multiple threads to handle concurrent requests.
- Node.js
  - **Single-threaded**: Node.js uses a single thread to handle all incoming requests.

Node.js vs Traditional Server-side Programming

- Traditional Server-Side Programming
  - **Blocking I/O:** I/O operations block the thread until the operation completes, which can lead to inefficiencies, especially with high concurrency.
- Node.js
  - **Non-blocking I/O:** Operations do not block the execution thread. Instead, they are offloaded to the event loop or worker threads, allowing the server to handle other tasks concurrently.
  - **Efficient for I/O-bound tasks:** Ideal for applications with heavy I/O operations, such as APIs, real-time applications, and data streaming.

Node.js vs Traditional Server-side Programming

- Traditional Server-Side Programming
  - **Various Languages**: Uses languages like Java, C#, PHP, Python, etc., each with its own syntax and paradigms.
  - Synchronous model: Often follows a synchronous programming model, although asynchronous capabilities are available in most modern frameworks.
- Node.js
  - **JavaScript**: Uses JavaScript for both client-side and server-side programming, allowing for a unified development stack.

Node.js vs Traditional Server-side Programming

- Traditional Server-Side Programming
  - **Vertical scaling**: Often relies on vertical scaling (adding more resources to a single server) though horizontal scaling is possible.
  - **Thread management:** Requires careful thread management to achieve high concurrency and performance.
- Node.js
  - **Horizontal scaling:** Easily scales horizontally by spawning multiple instances of the application.
  - Event-driven model: Handles a large number of concurrent connections efficiently, making it suitable for real-time applications.

Key Features

- Cross Platform
  - Runs on all Operating Systems
- Asynchronous and Event-Driven
  - Light-weight and efficient for creating a real-time applications with concurrent users.
- Built on Chrome's V8 JavaScript Engine
  - The V8 JavaScript high performance engine compiles code to machine-code
- Fast and Scalable Applications
  - Capable of handling huge number of simultaneous connections with high throughput

Key Features

- Node API
  - Built-in library for common and complex functionalities
- Rich Ecosystem
  - Node.js has a vast ecosystem of libraries and modules, available through the Node Package Manager (npm).
- Full Stack Development
  - Build a complete application(end to end) using JavaScript

Node.js Applications

- Real-time Chat Applications
  - Example: Slack, Discord, or any live chat support system.
  - Use Websockets
- Streaming Data Applications
  - Example: Video streaming services like Netflix, or audio streaming services like Spotify.
- RESTful APIs and Microservices
  - Example: RESTful APIs for web services like Google Maps API, or microservices in an e-commerce platform.
- Handling Multiple File Uploads
  - Example: File upload services like Google Drive or Dropbox.
- Data-Intensive Real-time Dashboards
  - Example: Financial trading platforms or monitoring dashboards
- Serverless Architectures
  - Example: Serverless functions running on cloud platforms

Modules

Node.js uses a modular architecture to encapsulate and organize code into reusable pieces called modules.

Facilitates code reuse, maintainability, and separation of concerns.

Each module has its own scope, preventing variable conflicts.

Modules

Node.js supports two primary module systems:

- CommonJS (CJS)
- ECMAScript Modules (ESM)

CommonJs

Default Module System: Initially designed for server-side JavaScript in Node.js.

Synchronous Loading: Modules are loaded synchronously.

Syntax:

Importing: const module = require('module');

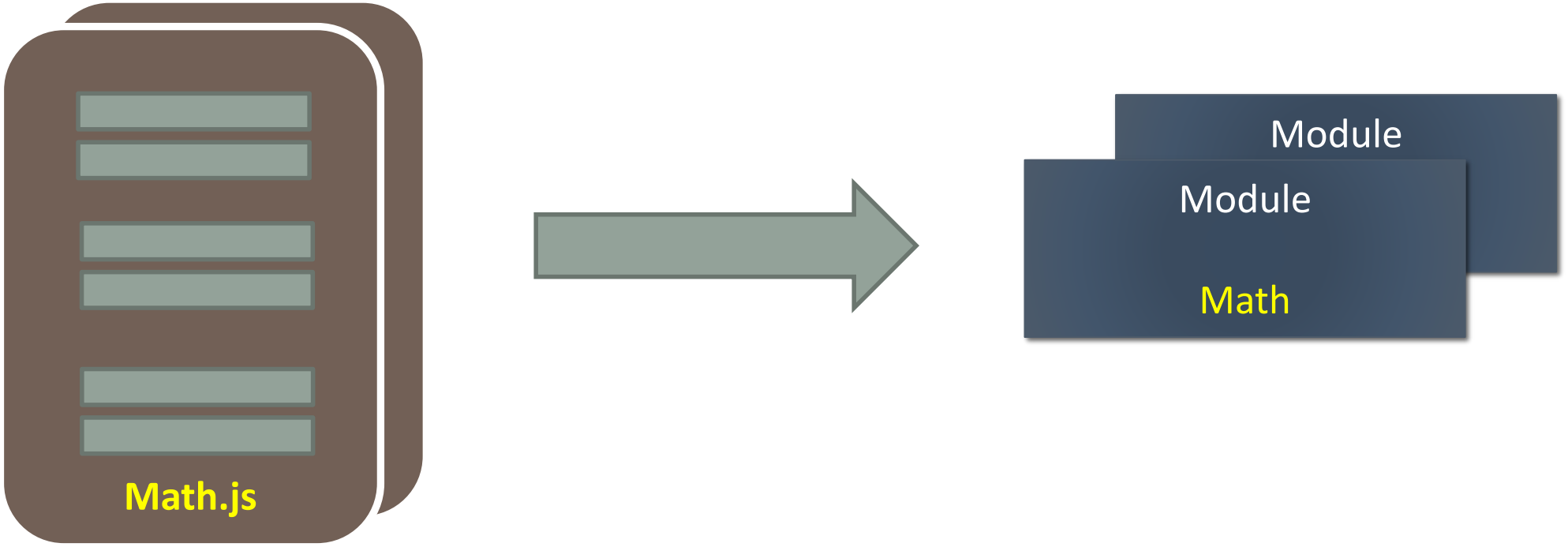
Exporting: module.exports = value; or exports.value = value;

Extensions: Files typically use .js extension. No specific configuration is required.

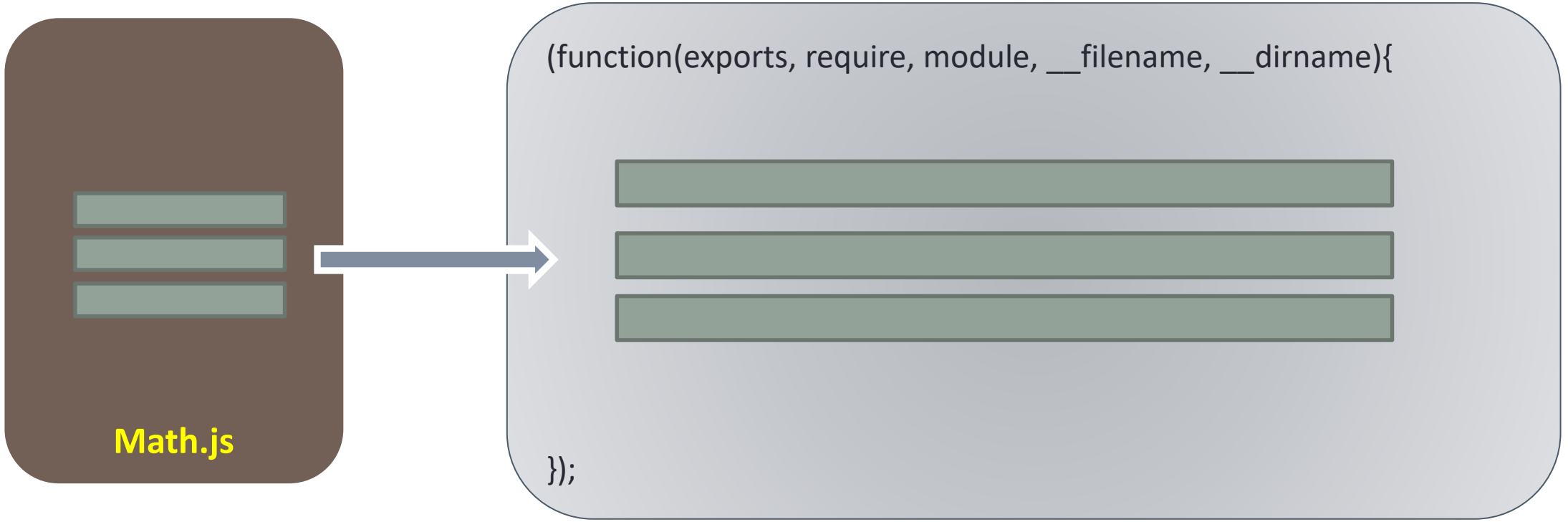
ECMAScript Modules (ESM)

ECMAScript Modules (ESM)	Modern Module System: Introduced in ECMAScript 2015 (ES6).
Asynchronous Loading:	Can be loaded asynchronously, Allows for better performance and optimization.
Syntax:	Importing: import { value } from 'module'; Exporting: export const value = ...; or export default value;
Extension	Files use .mjs extension or .js with "type": "module" in package.json.

Module



Module Wrapper



The Module Wrapper Function

- Every module in Node.js is wrapped in a function before it is executed. This wrapper function provides module-specific variables like
  - exports,
  - require,
  - module,
  - __filename,
  - __dirname.

The Module Wrapper Function

```
(function (exports, require, module, __filename, __dirname) {
  // Module code actually lives here
});
```

Sources of Modules

Built-in modules

- Comes pre-packaged with Node
- Examples fs, http, crypto, os

The project files

- Each .js file is a module
- JavaScript files can export variables, functions and objects.
- JavaScript files can be imported by providing the path.

Module folder

- A folder with a package.json file containing a main field.
- A folder with an index.js file in it.

External Modules

- Install using NPM or yarn
- Installed into node-modules folder

Module Interoperability

- Node.js allows interoperability between CJS and ESM:
- Importing CommonJS in ESM: Use dynamic import() or createRequire.
- createrequire

```
import { createRequire } from 'module’;
const require = createRequire(import.meta.url);
const { add } = require('./math.js’);
```

- Dynamic Import

```
import('path/to/math.js’)
    .then((mathModule) =>
    {
    const { add, multiply } = mathModule;
    }
     .catch((err) => {});
```

Module Interoperability

- Importing ESM in CommonJS: Use dynamic import().

```
async function loadModule()
{
    const { add } = await import('./math.mjs’);
     console.log(add(2, 3)); // 5
    }
loadModule();
```

Benefits of Modules

Encapsulation: Keeps code organized and modular.

Reusability: Promotes reuse of code across different parts of the application.

Maintainability: Makes the codebase easier to maintain and update.

Namespace Management: Prevents global namespace pollution by encapsulating variables and functions.

Node Architecture: Core constituents

V8 Engine: For executing JavaScript code.

libuv: For managing asynchronous I/O operations.

Node.js Core Libraries: Providing essential modules and APIs.

C/C++ Bindings: Enabling integration and interaction between V8 and libuv.

Node Architecture: V8 Engine

- The V8 engine is a critical component of the Node.js runtime
- Responsible for executing JavaScript code.
- Developed by Google
- V8 is designed for high performance and efficiency.

Node Architecture: V8 Engine

- Compilation
  - V8 compiles JavaScript code directly into machine code before execution.
  - This identifies Hotspots(frequently executed code)
- Execution
  - Executes the compiled machine code
- JIT Compilation
  - V8 uses Just-In-Time (JIT) compilation to optimize JavaScript code dynamically as it runs.
  - Inline Caching: speeds up property access
- Garbage Collection
  - Memory Management

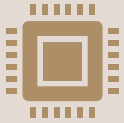
Node Architecture: V8 Engine

- ECMAScript Compliance
  - Compliant with the latest ECMAScript standards
- Integrates with libuv
  - For execution of asynchronous I/0 operation
- Efficient
  - Fast startups and optimized code
- Memory Efficient
  - Heap Management and low-latency GC.

Node Architecture: libuv

The backbone of Node.js's asynchronous I/O.

libuv is a multi-platform library that focuses on asynchronous I/O operations



It abstracts the complexities of asynchronous I/O operations across different operating systems

Node Architecture: libuv

- Event Loop
  - The event loop is the heart of libuv and Node.js.
  - The event loop is single-threaded
  - It is responsible for handling and dispatching events to appropriate callback functions.
- Event Queues
  - All asynchronous tasks are added to an event queue.
  - The event loops picks tasks from the queue.
  - There are multiple queue
- Thread Pool
  - libuv uses a thread pool to handle blocking operations.
  - libuv offloads blocking operations to the thread-pool

Node Package Manager(NPM)

It is the default package manager for Node.js.

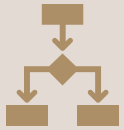
npm helps in managing dependencies and packages for JavaScript applications.

It allows developers to share and reuse code.

npm is included with Node.js installation.

npm: Initializing a Project

Use npm init to create a package.json file, which manages project metadata and dependencies.



Command: npm init

To use default settings: npm init -y

package.json

The package.json file is a crucial part of any Node.js project.

It serves as the manifest file for the project, containing metadata and information about the project's

Dependencies,

Scripts,

and more.

package.json: key entries

name: The name of your project or package.

version: The current version of your project, following semantic versioning.

description: A brief description of your project.

keywords: An array of keywords that describe your project and help in searchability.

package.json: key entries

main: The entry point of your application.

The file loaded when your package is required by another application.

scripts: Defines a set of script commands executed using npm run <script-name>.

Common scripts include start, test, build, etc.

dependencies: Specifies the packages required by your project to run.

These are installed when you run npm install.

package.json: key entries

devDependencies: Specifies the packages required for development and testing.

These are installed when you run npm install with –save-dev option.

repository: Information about the source code repository.

author: Information about the project author.

license: Specifies the license under which the project is distributed.

engines: Specifies the versions of Node.js or npm required to run the project.

npm commands

Node Libraries

- File System(fs)
  - Provides functions to interact with the file system, including reading, writing, and manipulating files and directories.
- HTTP/HTTPS
  - Purpose: Enables the creation of HTTP and HTTPS servers and clients.
- Events(events)
  - Provides a way to handle and emit custom events using the EventEmitter class.
  - Observer pattern

Node Libraries

- Streams
  - Handles streaming data, allowing efficient reading and writing of large files, network data, etc.
- Path
  - Provides utilities for working with file and directory paths.
- Process
  - Provides information and control over the current Node.js process.
- Timers
  - Schedules the execution of code at specified times or intervals.

Node Libraries

- OS
  - Provides information about the operating system.
- Crypto
  - Provides cryptographic functionality, including hashing, encryption, and signing.
- Child Processes
  - Allows the creation of child processes to execute commands and scripts.
- Cluster
  - Allows for creation of child processes that share the same server port, useful for improving performance on multi-core systems.

Asynchronous Programming

Asynchronous programming allows multiple operations to be executed concurrently.

Useful in environments where operations can take an unpredictable amount of time

Examples: reading files from disk, querying a database, or making network requests

Asynchronous programming is crucial for building scalable and high-performance applications due to its non-blocking I/O model.

Asynchronous Programming

Callbacks

- A function invoked to notify the result of the async call

Promises

- Promise represents the result of asynchronous function.
- Promises can be used to avoid chaining of callbacks.

Async-await

- Syntactic features to write better async code

Events (Observer Pattern)

- Node.js provides **EventEmitter** module that can help you to program using events.

Async programming: Callbacks

- A callback is a function passed as an argument to another function, which is executed after the completion of an asynchronous operation.

```
const fs = require('fs’);
fs.readFile('example.txt', 'utf8', (err, data) =>
  {
     if (err) { console.error('Error reading file:', err);
     return;
    }
    console.log('File content:', data);
});
console.log('Reading file...');
```

Async Programming: Callback hell

- Callback Hell refers to a situation where multiple nested callbacks make the code difficult to read, maintain, and debug.
- This problem often arises in asynchronous programming when a series of asynchronous operations depend on each other.
- Avoiding callback hell
  - Use promises
  - Use async-await

Promise

- A Promise in JavaScript is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value.
- It allows you to write asynchronous code in a more manageable and readable way compared to traditional callbacks.
- Promise States
  - **Pending**: The initial state, neither fulfilled nor rejected.
  - **Fulfilled**: The operation completed successfully, and the promise has a resulting value.
  - **Rejected**: The operation failed, and the promise has a reason for the failure (an error).

Creating a Promise

```
const myPromise = new Promise((resolve, reject) => {
        const success = true;
        if (success) {
            resolve('Operation was successful!');
        }
        else {
           reject('Operation failed.');
        }
});
```

Using Promises

- You interact with a Promise using its then(), catch(), and finally() methods.
- **then**(onFulfilled, onRejected):
  - Attaches callbacks for the resolution and/or rejection of the Promise.
- **catch**(onRejected):
  - Attaches a callback for only the rejection of the Promise.
- **finally**(onFinally):
  - Attaches a callback that is invoked when the Promise is settled (fulfilled or rejected).

Using Promise

```
myPromise
  .then(result => {
    console.log(result); // 'Operation was successful!'
  })
  .catch(error => {
    console.error(error); // 'Operation failed.'
  })
  .finally(() => {
    console.log('Promise has been settled.');
  });
```

Chaining Promises

- Promises can be chained to handle a series of asynchronous operations.
- Each then method returns a new Promise, allowing for sequential execution of asynchronous tasks.

```
promise1
  .then(result1 => {
    console.log(result1); // 'First promise resolved'
    return promise2;
  })
  .then(result2 => {
    console.log(result2); // 'Second promise resolved'
  })
  .catch(error => {
    console.error(error);
  });
```

Handling multiple promises

- Promise.all(iterable):
  - Waits for all promises in the iterable to be fulfilled or for any to be rejected. Returns a single Promise that resolves with an array of the results.
- Promise.race(iterable):
  - Returns a Promise that resolves or rejects as soon as one of the promises in the iterable resolves or rejects.
- Promise.any(iterable):
  - Returns a Promise that resolves as soon as one of the promises in the iterable resolves. If no promises resolve, it rejects with an AggregateError.

async - await

- **async** and **await** are syntactic features in JavaScript that provide a more straightforward and readable way to work with asynchronous operations.
- They are built on top of Promises,
- Make it easy to write and understand asynchronous code.
- Avoids deeply nested callbacks or chains of .then() and .catch().

async - await

- The async keyword is used to declare an asynchronous function.
- An asynchronous function implicitly returns a Promise.
- If the function returns a value, the Promise is resolved with that value.
- If the function throws an error, the Promise is rejected with that error.
- The await keyword can only be used inside an async function.
- It pauses the execution of the async function
  - waiting for the Promise to resolve or reject.

Event Emitter

- The **EventEmitter** is a core module in Node.js that allows objects to communicate with each other through events.
- It provides a mechanism to implement the observer pattern,
- The emitter emits events
- Objects (listeners) listen for those events and respond accordingly.
- **Events**:
  - Named signals that an EventEmitter object can emit.Listeners:
- **Functions(callbacks)**
  - that are registered to be called when a specific event is emitted.

Event Emitter methods

- **on**(event, listener):
  - Adds a listener to the specified event.
  - Multiple listeners can be added for the same event.
- **once**(event, listener):
  - Adds a one-time listener to the specified event.
  - The listener is removed after it is called the first time.
- **emit**(event, [...args]):
  - Emits the specified event,
  - calling all the listeners registered for that event with the provided arguments.
- **removeListener**(event, listener):
  - Removes a specific listener from an event.
- **removeAllListeners**([event]):
  - Removes all listeners for the specified event.
- **listenerCount**(event):
  - Returns the number of listeners for the specified event.

Timer

- Node.js provides several timer functions that allow us to schedule the execution of code at specified intervals or after a certain delay.
- These timer functions are built on the timers module,
- They leverages the event loop to manage the scheduling and execution of these functions.

Timer Methods

- setTimeout(callback, delay, [arg1, arg2, ...]):
  - Schedules a one-time callback after the specified delay.
- clearTimeout(timeoutId):
  - Cancels a scheduled timeout.
- setInterval(callback, interval, [arg1, arg2, ...]):
  - Schedules a callback to be executed repeatedly at the specified interval.
- clearInterval(intervalId):
  - Cancels a scheduled interval.
- setImmediate(callback, [arg1, arg2, ...]):
  - Schedules a callback to be executed immediately after the current event loop cycle.
- clearImmediate(immediateId):
  - Cancels a scheduled immediate callback.

process.nextTick

- process.nextTick() is a function in Node.js that schedules a callback to be invoked in the next iteration of the event loop, but before any I/O operations, timers, or other tasks.

Micro Tasks

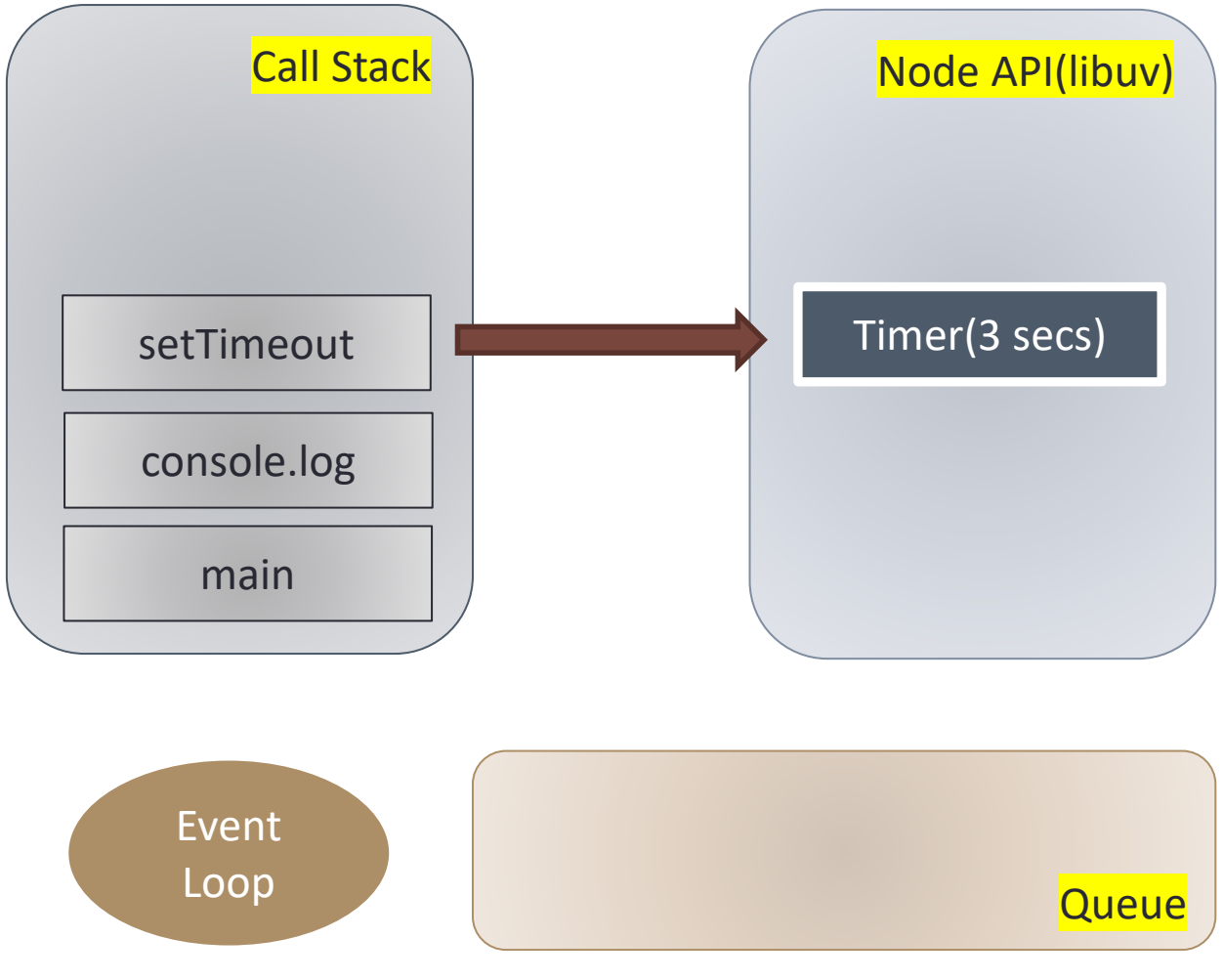
- Microtasks are a type of task that is prioritized over macrotasks in the event loop.
- They are executed immediately after the currently executing script and before any I/O operations, timers, or other macrotasks.
- In JavaScript, microtasks include process.nextTick and Promises (.then(), .catch(), .finally()).

Macro Tasks

- Macrotasks (also known as tasks) are executed in their respective phases of the event loop.
- These include I/O operations, timers (setTimeout, setInterval), and setImmediate.

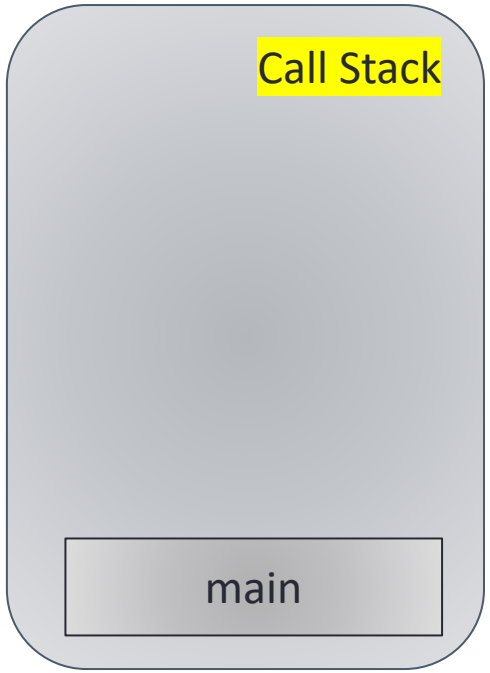
Asynchronous Execution

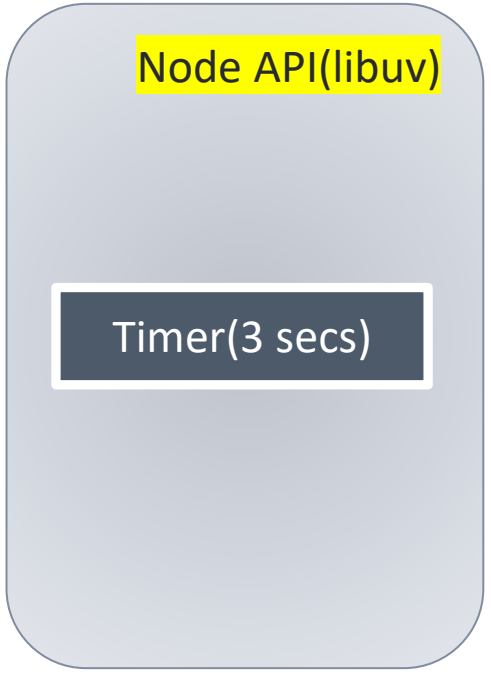
```
console.log("App started");

setTimeout(function(){
    console.log("timeout after 3 sec")
}, 3000);

setTimeout(function(){
    console.log("timeout after 0 sec")
}, 0);

console.log("App Completed");
```

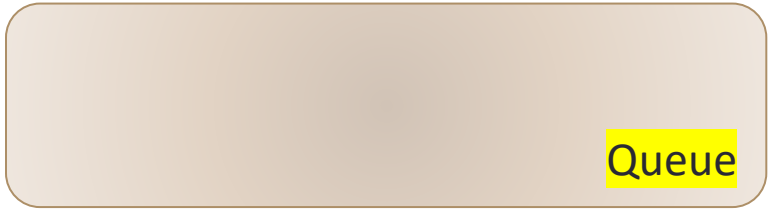


Asynchronous Execution

```
console.log("App started");

setTimeout(function(){
    console.log("timeout after 3 sec")
}, 3000);

setTimeout(function(){
    console.log("timeout after 0 sec")
}, 0);

console.log("App Completed");
```

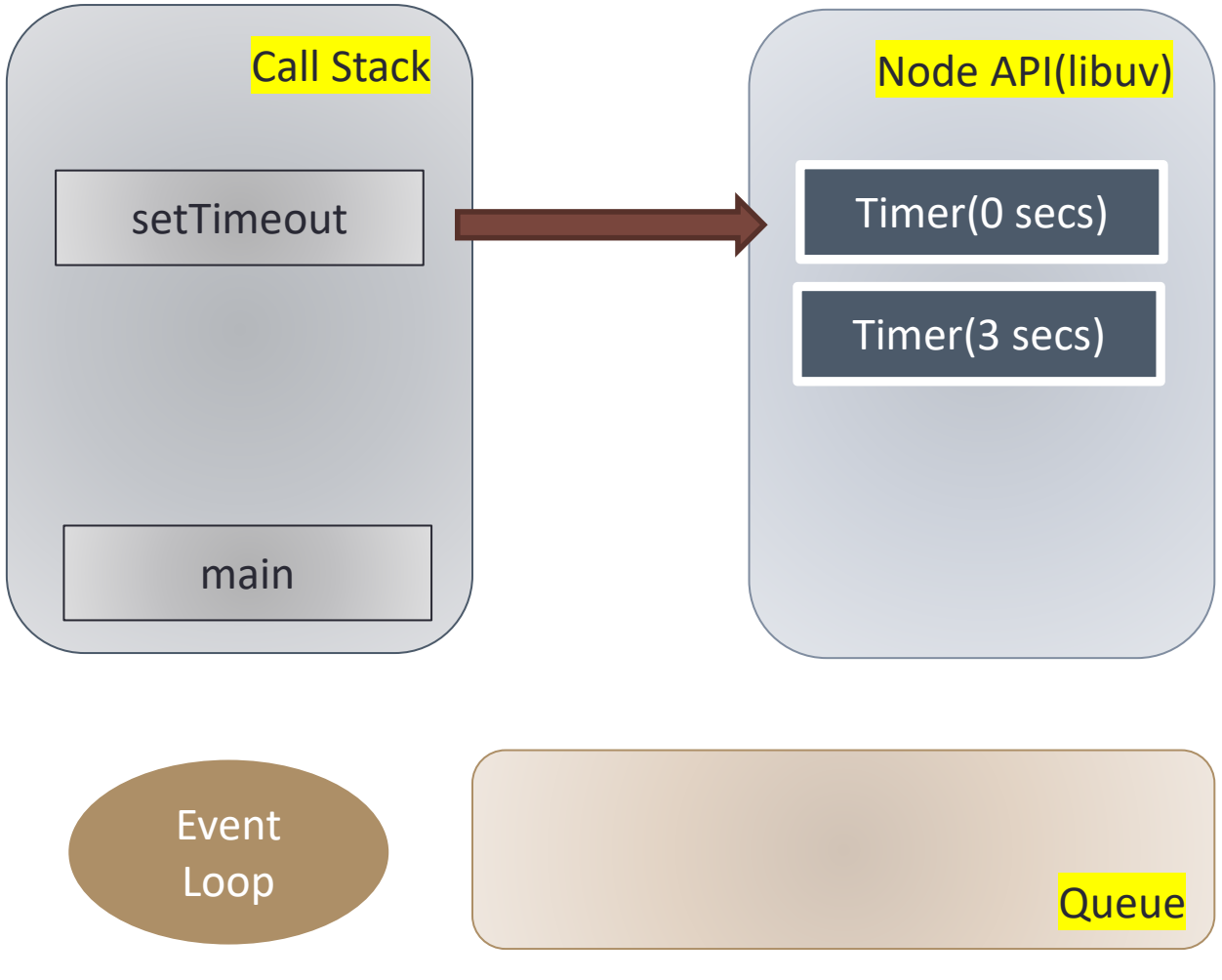






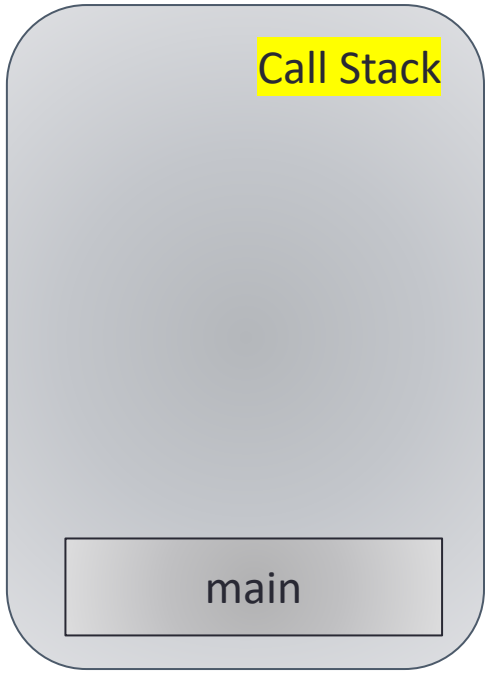
Asynchronous Execution

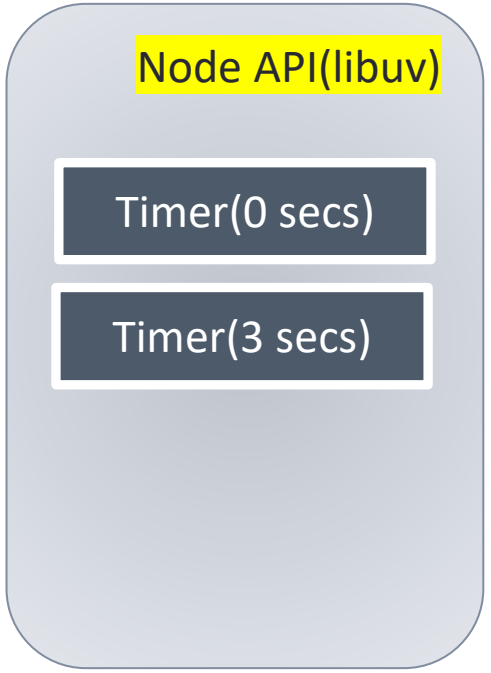
```
console.log("App started");

setTimeout(function(){
    console.log("timeout after 3 sec")
}, 3000);

setTimeout(function(){
    console.log("timeout after 0 sec")
}, 0);

console.log("App Completed");
```

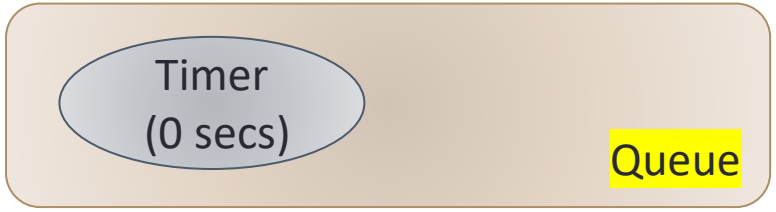


Asynchronous Execution

```
console.log("App started");

setTimeout(function(){
    console.log("timeout after 3 sec")
}, 3000);

setTimeout(function(){
    console.log("timeout after 0 sec")
}, 0);

console.log("App Completed");
```

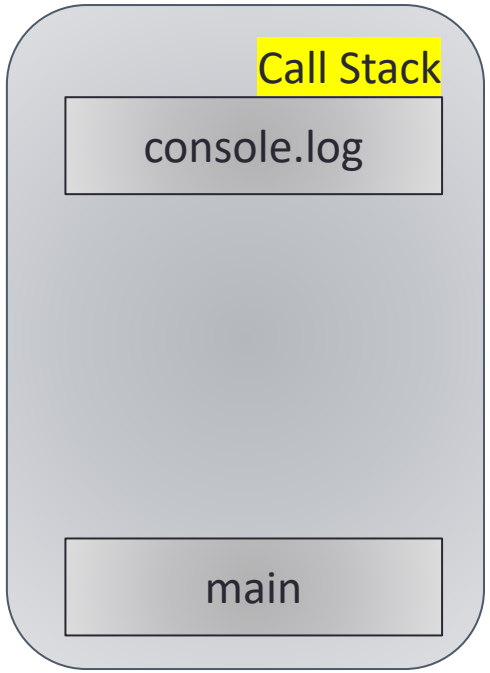


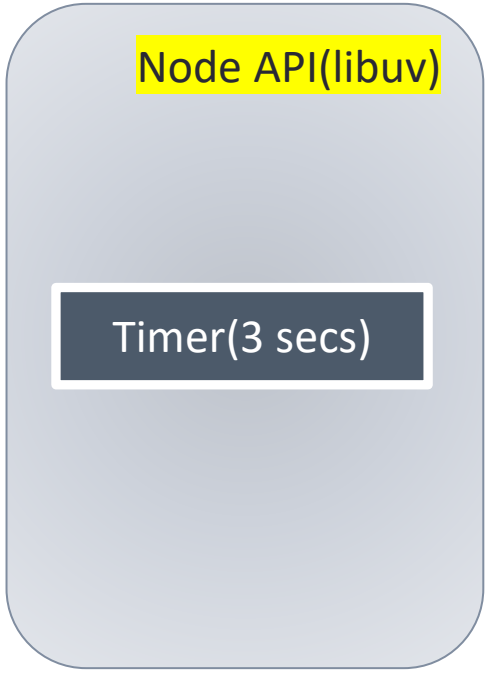


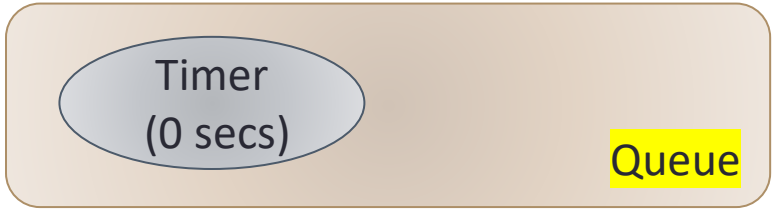


Asynchronous Execution

```
console.log("App started");

setTimeout(function(){
    console.log("timeout after 3 sec")
}, 3000);

setTimeout(function(){
    console.log("timeout after 0 sec")
}, 0);

console.log("App Completed");
```

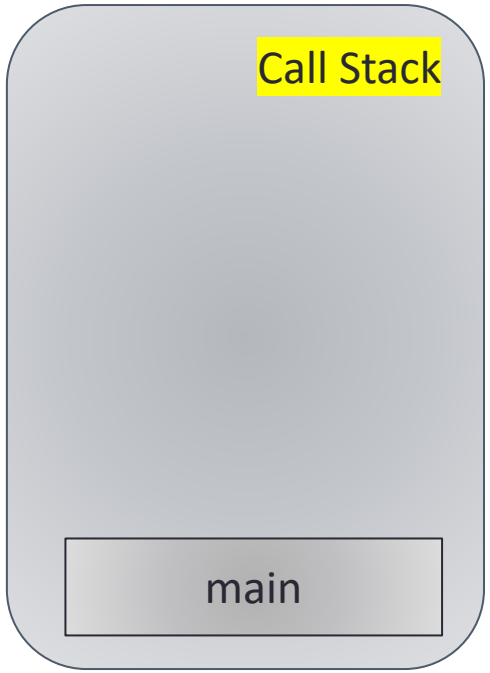


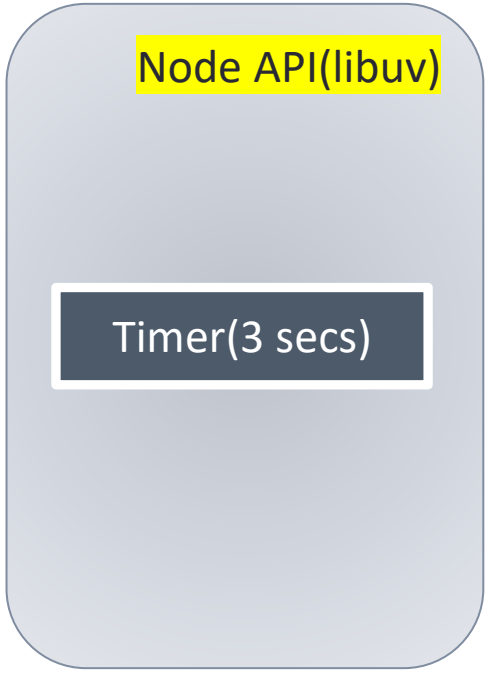


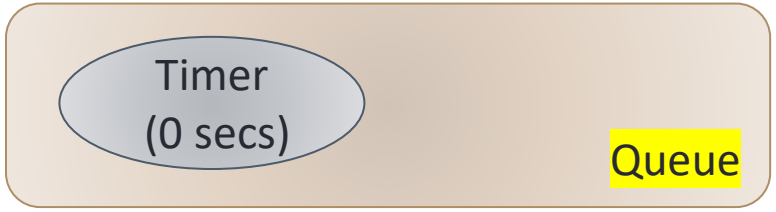


Asynchronous Execution

```
console.log("App started");

setTimeout(function(){
    console.log("timeout after 3 sec")
}, 3000);

setTimeout(function(){
    console.log("timeout after 0 sec")
}, 0);

console.log("App Completed");
```

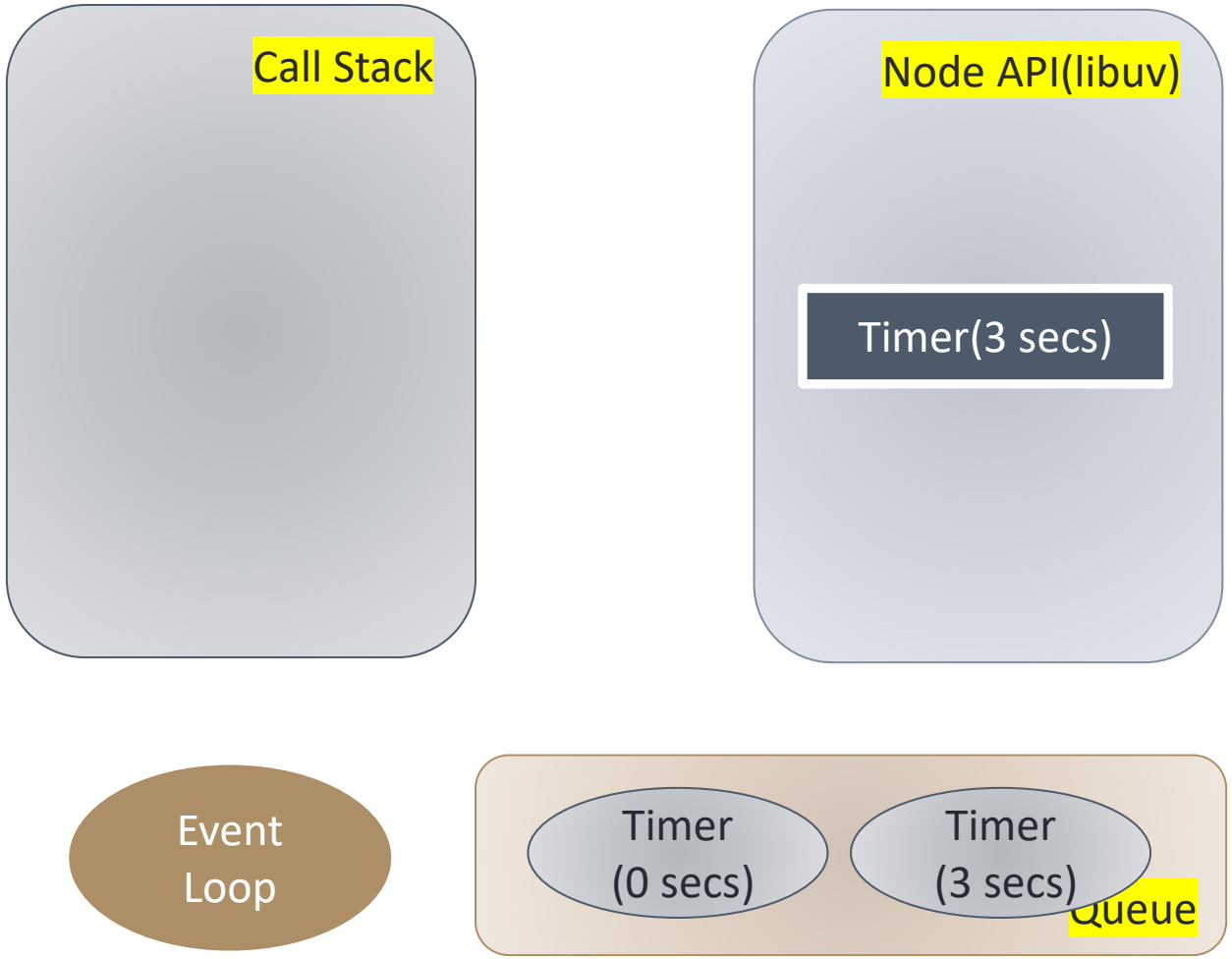






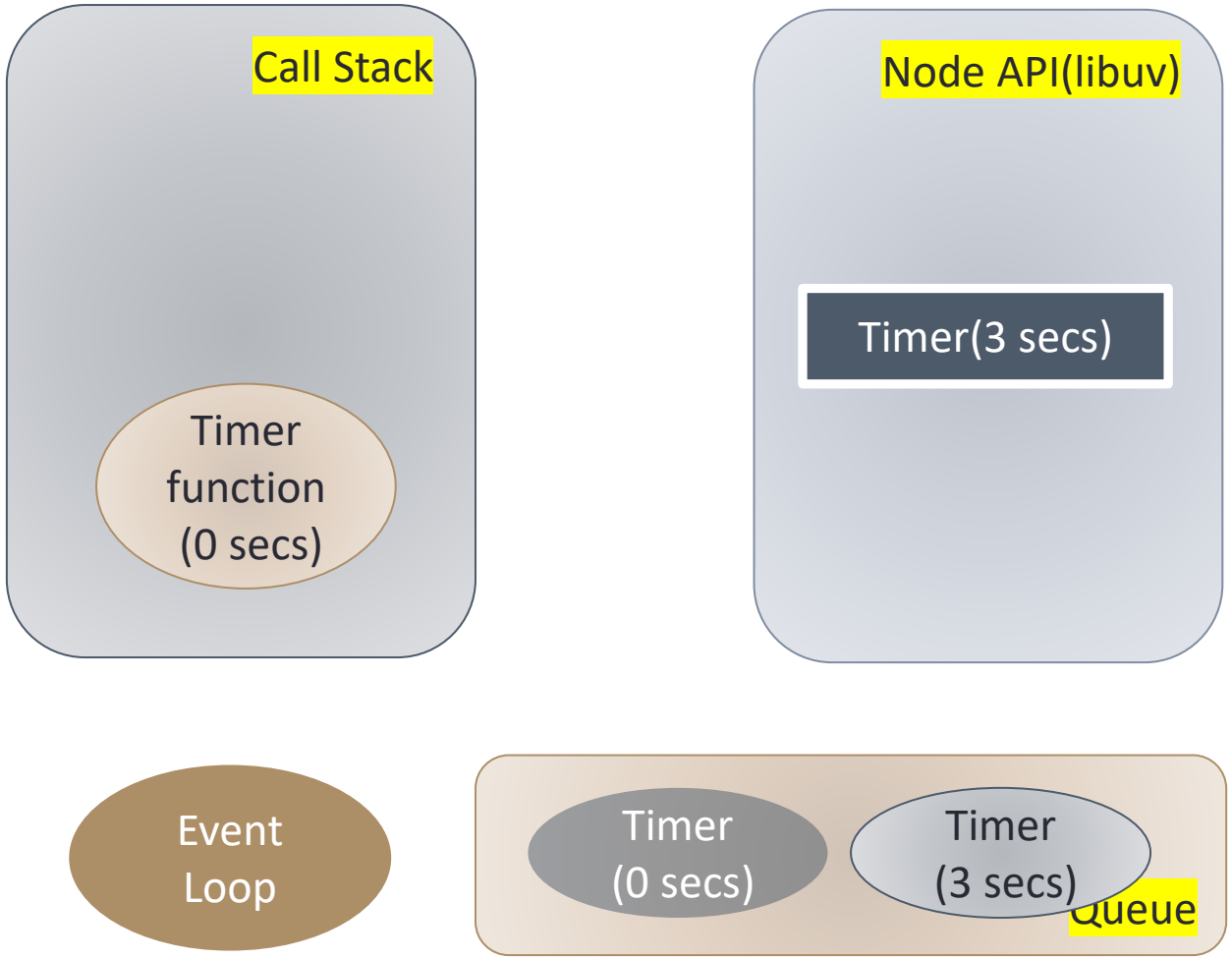
Asynchronous Execution

```
console.log("App started");

setTimeout(function(){
    console.log("timeout after 3 sec")
}, 3000);

setTimeout(function(){
    console.log("timeout after 0 sec")
}, 0);

console.log("App Completed");
```



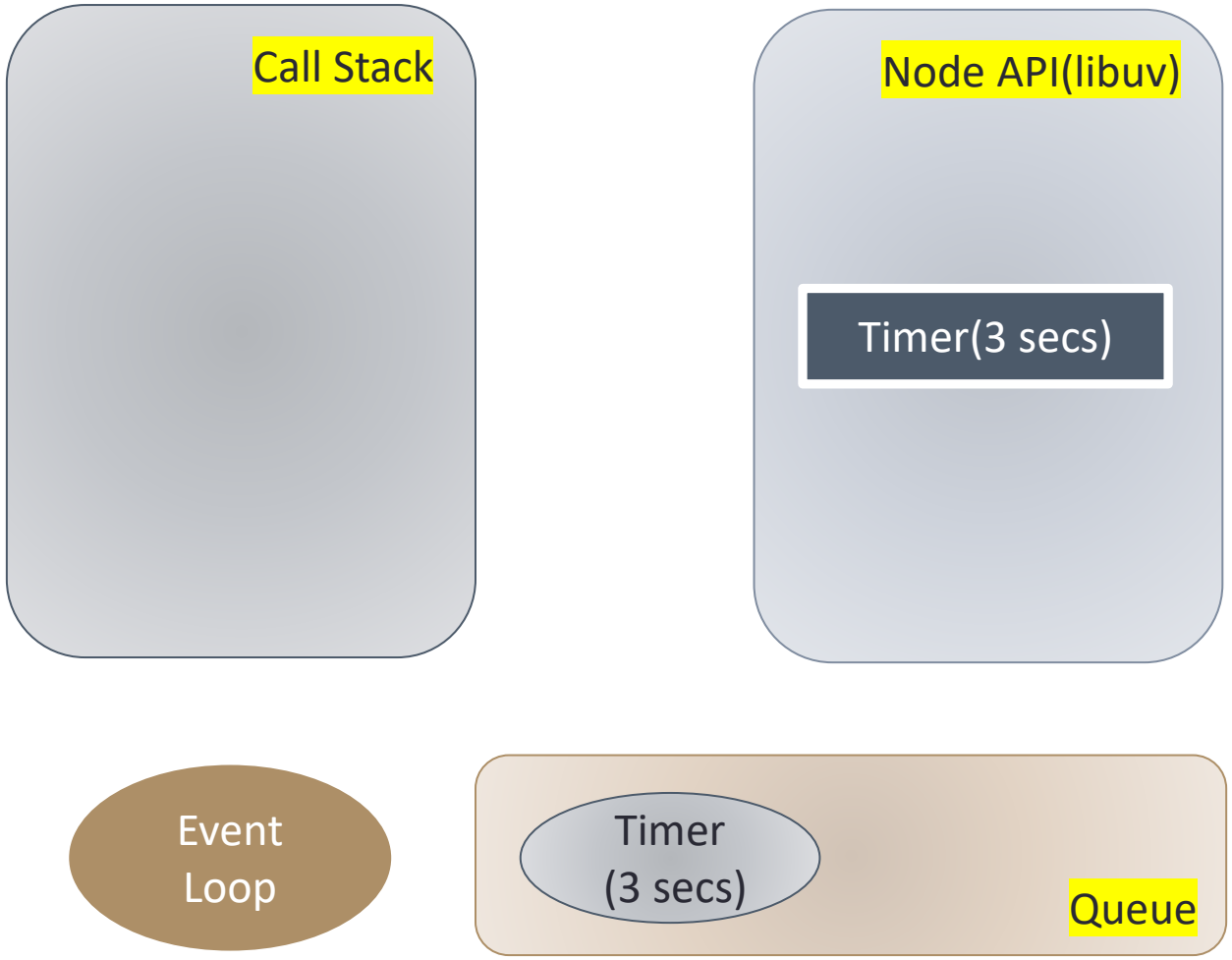
Asynchronous Execution

```
console.log("App started");

setTimeout(function(){
    console.log("timeout after 3 sec")
}, 3000);

setTimeout(function(){
    console.log("timeout after 0 sec")
}, 0);

console.log("App Completed");
```



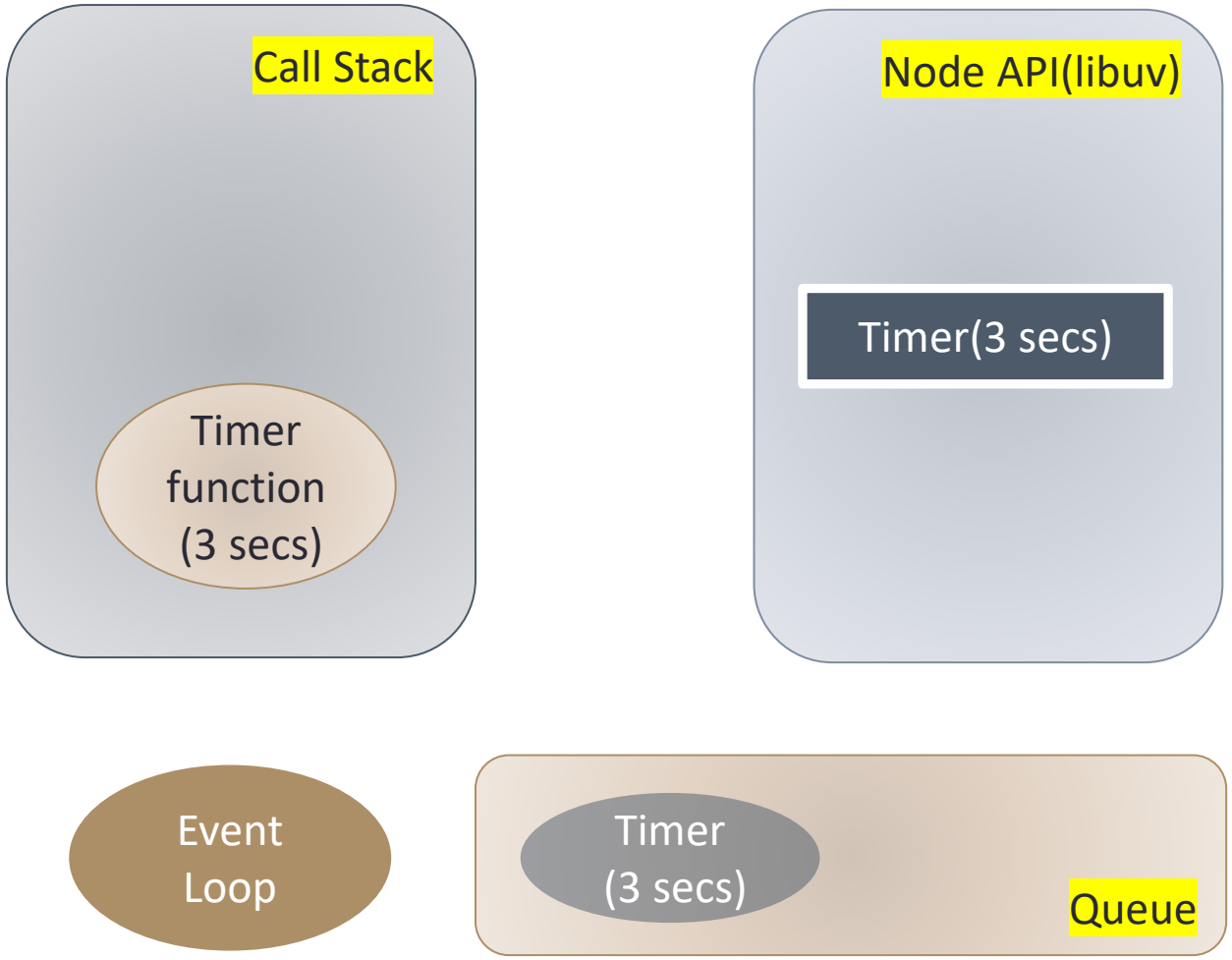
Asynchronous Execution

```
console.log("App started");

setTimeout(function(){
    console.log("timeout after 3 sec")
}, 3000);

setTimeout(function(){
    console.log("timeout after 0 sec")
}, 0);

console.log("App Completed");
```



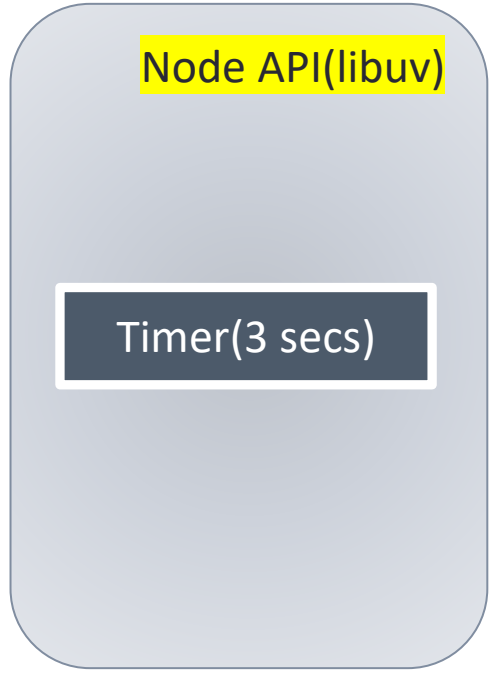
Asynchronous Execution

```
console.log("App started");

setTimeout(function(){
    console.log("timeout after 3 sec")
}, 3000);

setTimeout(function(){
    console.log("timeout after 0 sec")
}, 0);

console.log("App Completed");
```

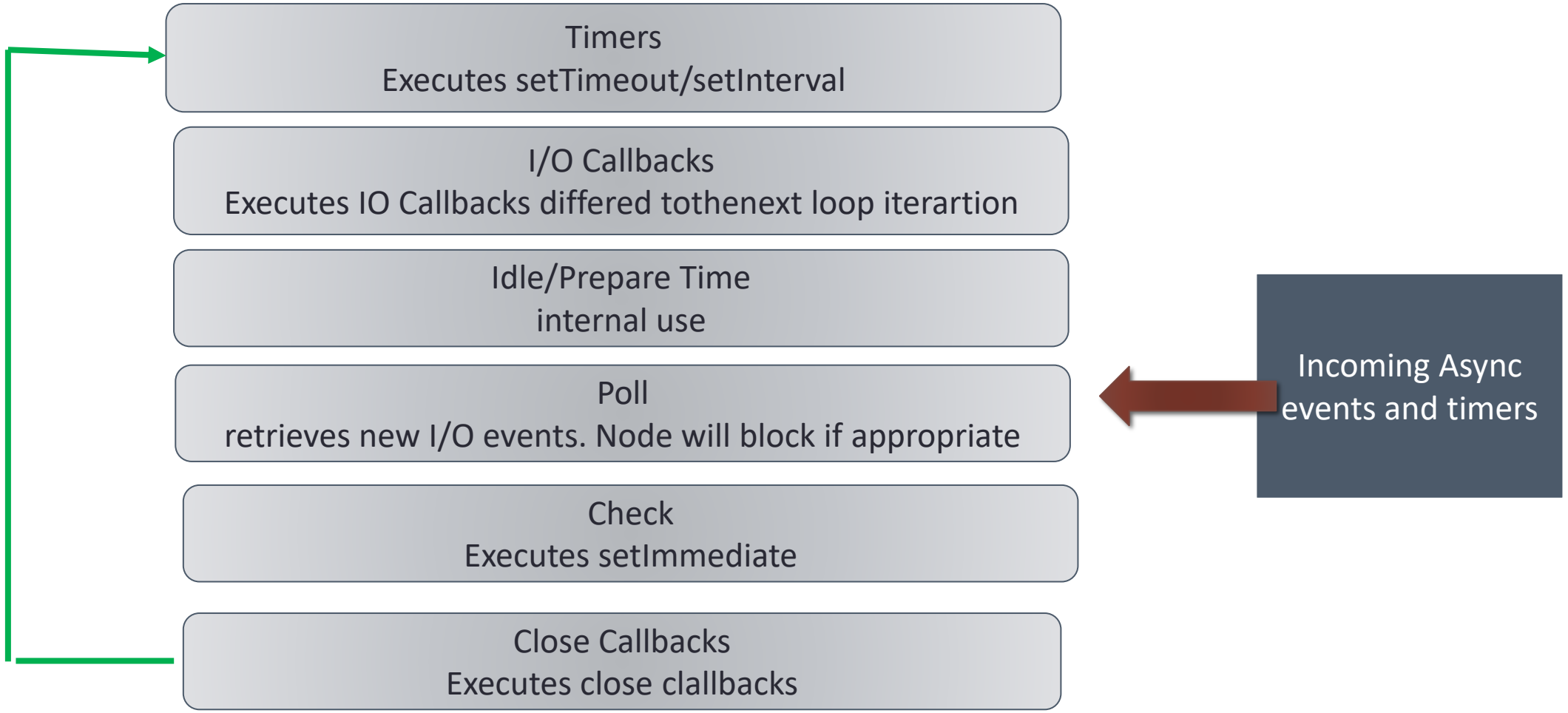


Asynchronous Execution

```
console.log("App started");

setTimeout(function(){
    console.log("timeout after 3 sec")
}, 3000);

setTimeout(function(){
    console.log("timeout after 0 sec")
}, 0);

console.log("App Completed");
```



Event Loop


Timers
Executes setTimeout/setInterval
I/O Callbacks
Executes IO Callbacks differed tothenext loop iterartion
Idle/Prepare Time
internal use
Poll
retrieves new I/O events. Node will block if appropriate
Check
Executes setImmediate
Close Callbacks
Executes close clallbacks
Incoming Async events and timers

Streams

- The Streams module in Node.js provides a way to handle streaming data.
- Streams are a powerful allows us to work with data in a more efficient way, especially when dealing with large files or data sources.
- Streams are instances of EventEmitter and can be used to read, write, and transform data efficiently.

Types of Streams

Readable Streams: Used for reading data sequentially.

Reading Files

Receiving HTTP request

Writable Streams: Used for writing data sequentially.

Writing Files

Sending HTTP responses

Duplex Streams: Can be used both for reading and writing data.

Network sockets

Transform Streams: A type of duplex stream where the output is computed based on the input.

Compresses or decompresses data(zlib)

Flow in Streams

- Streams are used to handle continuous data flows.
- There are two main modes in which a readable stream can operate:
  - flowing
  - paused.
- Flowing Mode:
  - Data is read from the source and provided to the application as soon as it is available, without waiting for the application to explicitly request it.
- Paused Mode:
  - Data is read from the source only when the application explicitly requests it.

Switching Between Modes

- When a readable stream is created, it starts in paused mode.
- Adding a 'data' event listener switches the stream to flowing mode.
- Removing all 'data' event listeners switches the stream back to paused mode.
- Calling .pause() on a readable stream switches it to paused mode.
- Calling .resume() on a readable stream switches it to flowing mode.
- Piping the stream to a writable stream also switches it to flowing mode.

Streams-Pipes

- Pipes are a mechanism provided by Node.js streams to connect the output of one stream directly to the input of another.
- The pipe method is available on readable streams

Anil Joseph

anil.jos@gmail.com

WhatsApp : +91 98331 69784

Thank You