Introduction

Welcome to the world of MERN full stack development! In this book, we will delve into the intricacies of the MERN stack and learn how to build robust web applications using this powerful technology stack.

MERN is an acronym for four popular technologies: MongoDB, Express, React, and NodeJS. These technologies work together seamlessly to form a full stack framework that enables developers to build scalable and performant web applications.

JavaScript is at the heart of the MERN stack, and it is one of the most widely used programming languages in the world. JavaScript is a versatile language that can be used for both front-end and back-end development, making it an essential tool for full stack development.

Throughout this book, we will cover the fundamental concepts of MERN development, including creating RESTful APIs with NodeJS and Express, designing dynamic user interfaces with React, and working with MongoDB to store and retrieve data.

In addition to learning these core concepts, we will also explore the applications of the MERN stack in various real-world scenarios, such as building e-commerce platforms, social media applications, and more.

Whether you are an experienced developer looking to expand your skillset or a newcomer to the world of web development, this book will provide you with the knowledge and tools needed to create robust, scalable, and performant web applications using the MERN stack.

JavaScript Basics

JavaScript is a high-level, dynamic, and interpreted programming language that is primarily used for creating interactive client-side web applications, but it can also be used on the server-side through NodeJS. It was created in 1995 by Brendan Eich, who was then working at Netscape Communications Corporation.

Core features of JavaScript

1. Object-Oriented Programming (OOP): JavaScript is an object-oriented programming language, which means that it supports the creation of objects with their own properties and methods. Objects are a fundamental concept in JavaScript and are used extensively in the language.
2. Dynamic Typing: JavaScript is dynamically typed, which means that the data type of a variable can change at runtime. This makes JavaScript very flexible and easy to use.
3. First-Class Functions: JavaScript functions are first-class citizens, which means that they can be passed around like any other data type. Functions can also be used as arguments to other functions, and they can be returned as values from functions.
4. Asynchronous Programming: JavaScript has built-in support for asynchronous programming, which means that it can execute multiple operations simultaneously without blocking the execution of other code. This is typically used in web applications to make requests to servers without blocking the UI.
5. Prototypal Inheritance: JavaScript uses prototypal inheritance, which is a different approach to inheritance than classical inheritance used in languages like Java and C++. This allows for more flexible object creation and is a key feature of the language.

Data types in JavaScript

Primitive data types

1. String: used to represent textual data, such as names or messages. Strings are enclosed in single or double quotes.

2. Number: used to represent numerical data, including integers and floating-point numbers.
3. Boolean: used to represent true/false values.
4. Undefined: used to represent the absence of a value. A variable that has not been assigned a value is undefined.
5. Null: used to represent the intentional absence of any object value.
6. Symbol: a new data type added in ES6, which represents a unique identifier. Symbols are often used as keys in objects.

JavaScript also has one non-primitive data type

7. Object: used to represent complex data structures, such as arrays, functions, and objects. Objects are a key feature of JavaScript, and many of the language's features are built around them.

Example of how variables are declared, initialized and accessed in JavaScript:

```
// Declare a variable called 'name'
let name;

// Initialize the variable 'name' with the value 'John'
name = 'John';

// Declare and initialize a variable called 'age' with the value 30
let age = 30;

// Print the values of 'name' and 'age' to the console
console.log(name);
console.log(age);
```

Let's break down each line and see what it does:

1. let name; declares a variable called 'name' without initializing it. This is known as a variable declaration.
2. name = 'John'; assigns the value 'John' to the variable 'name'. This is known as variable initialization.
3. let age = 30; declares and initializes a variable called 'age' with the value 30.

4. console.log(name); prints the value of 'name' to the console. The console is a tool used in web development that displays messages and logs in real-time, and is accessible through the developer console of most modern web browsers.

5. console.log(age); prints the value of 'age' to the console.

When you run this code in a JavaScript environment, such as a web browser console or a NodeJS environment, you will see the values of 'name' and 'age' printed to the console:

```
John
30
```

In this example, we used the let keyword to declare variables, which is the recommended way to declare variables in modern JavaScript. The let keyword allows us to declare variables that can be reassigned to a new value later on in the program.

Variable Declaration and Initialization in JS

In JavaScript, variables are stored in memory as part of the execution context. When a variable is declared, space is reserved in memory to store its value.

When a variable is accessed in JavaScript, the engine looks up the variable in memory and retrieves its value. This process is known as variable resolution.

Here's what happens behind the scenes of JavaScript when a variable is declared and defined:

1. Declaration: When a variable is declared using the let, const, or var keyword, the JavaScript engine creates a new identifier in memory with the name of the variable. The identifier is assigned a reference to a memory location where the value of the variable will be stored.

2. Initialization: When a variable is initialized, the engine assigns a value to the memory location that was reserved for the variable. If the variable is not initialized at the time of declaration, its value is set to undefined.

3. Assignment: When a value is assigned to a variable using the = operator, the value is stored in the memory location reserved for the variable.

4. Scope: Variables in JavaScript are scoped to the function or block in which they are declared. This means that variables declared inside a function are not accessible outside of the function.

5. Garbage Collection: When a variable is no longer needed, JavaScript automatically removes it from memory through a process called garbage collection. This frees up

memory that can be used by other parts of the program.

It's important to note that variables in JavaScript are dynamically typed, which means that their data type can change during the execution of the program. This is in contrast to statically typed languages, where the data type of a variable is fixed at compile time.

Data type: String

In JavaScript, a string is a sequence of characters enclosed in single or double quotes. Strings are one of the most common data types in JavaScript and are used to represent text.

Common string manipulation in JS are:

- Concatenation: You can concatenate two or more strings using the + operator. For example, if you have two strings "hello" and "world", you can concatenate them like this:

```
let greeting = "hello";
let name = "world";
let message = greeting + " " + name;
console.log(message); // Output: "hello world"
```

- String Length: You can find the length of a string using the length property. For example:

```
let myString = "This is a string";
console.log(myString.length); // Output: 16
```

- Accessing Characters: You can access individual characters in a string using square brackets and the index of the character you want. For example: javascript

```
let myString = "hello";
console.log(myString[0]); // Output: "h"
console.log(myString[3]); // Output: "l"
```

- Substring: You can extract a portion of a string using the substring method. For example:

```
let myString = "hello world";
let subString = myString.substring(0, 5); // Extracts the first 5 characters
console.log(subString); // Output: "hello"
```

- Replace: You can replace one or more occurrences of a substring in a string using the replace method. For example:

```
let myString = "hello world";
let newString = myString.replace("world", "javascript");
console.log(newString); // Output: "hello javascript"
```

Important features of string in JS are:

1. Strings are immutable: This means that once you create a string, you cannot change the individual characters in it. Instead, you must create a new string with the desired changes.
2. Strings are iterable: This means that you can loop over the characters in a string using a for loop or other iterable methods like forEach or map.
3. Strings have many built-in methods: JavaScript provides many methods for working with strings, such as substring, replace, and split. These methods make it easy to manipulate strings in various ways.
4. Strings can be converted to other data types: You can convert a string to a number using the parseInt or parseFloat methods. You can also convert a string to an array using the split method.

Overall, strings are an essential data type in JavaScript and are used extensively in web development. Understanding how to manipulate strings is crucial for building effective JavaScript applications.

Data type: Undefined and Null

In JavaScript, both undefined and null are special values that represent the absence of a value or an empty value. However, they are not interchangeable, and they have slightly different meanings.

Undefined

undefined is a primitive value that represents a variable that has been declared but has not been assigned a value. It also represents a function that has been defined but has no return statement.

Here's an example of a variable that is declared but not assigned a value:

```
let foo;
console.log(foo); // Output: undefined
```

In this example, the variable foo is declared but not assigned a value. When we try to log the value of foo to the console, we get undefined.

Here's an example of a function that does not have a return statement:

```
function bar() {}
console.log(bar()); // Output: undefined
```

In this example, the function bar is defined but does not have a return statement. When we call the function and try to log the return value to the console, we get undefined.

Null

null is also a primitive value that represents the intentional absence of any object value. It is often used to indicate that a variable or object property has no value.

Here's an example of a variable that is explicitly set to null:

```
let bar = null;
console.log(bar); // Output: null
```

In this example, the variable bar is explicitly set to null. When we log the value of bar to the console, we get null.

Here's an example of an object property that is set to null:

```
const person = {
  name: 'John',
  age: null,
};
console.log(person.age); // Output: null
```

In this example, the age property of the person object is set to null. When we log the value of person.age to the console, we get null.

Difference between Undefined and Null

The main difference between undefined and null is that undefined is the default value of a variable that has not been assigned a value, whereas null is an intentional absence of any object value. undefined is also a primitive value, whereas null is an object value. In general, you should use undefined to represent the absence of a value that should have been assigned, and use null to represent the intentional absence of any object value (empty).

Data type: Number

The Number data type in JavaScript is used to represent numeric values, both integers and floating-point numbers.

Some examples of numbers declaration in JavaScript:

```
let integer = 42; // integer number
let floatingPoint = 3.14; // floating-point number
let negativeNumber = -10; // negative number
let scientificNotation = 2.998e8; // scientific notation
```

The core features of the Number data type in JavaScript include the ability to perform arithmetic operations such as addition, subtraction, multiplication, and division. It also supports several mathematical methods such as Math.sqrt() (to find the square root), Math.pow() (to raise a number to a power), and Math.abs() (to find the absolute value of a number).

Examples of number-related manipulations in JavaScript:

```
let x = 10;
let y = 5;

// Addition
let sum = x + y; // 15

// Subtraction
let difference = x - y; // 5

// Multiplication
let product = x * y; // 50

// Division
let quotient = x / y; // 2

// Remainder (modulo)
let remainder = x % y; // 0

// Exponentiation
let exponentiation = x ** y; // 100000

// Square root
let squareRoot = Math.sqrt(x); // 3.1622776601683795

// Absolute value
let absoluteValue = Math.abs(-10); // 10
```

Additionally, JavaScript provides several methods for converting strings to numbers and vice versa. For example, the parseInt() method can be used to convert a string to an integer, and the parseFloat() method can be used to convert a string to a floating-point number.

```
let stringNumber = "42";
let parsedInteger = parseInt(stringNumber); // 42

let stringFloat = "3.14";
let parsedFloat = parseFloat(stringFloat); // 3.14
```

It is worth noting that JavaScript's Number data type has some limitations due to the way it represents numbers internally. Specifically, it can only represent numbers up to a certain precision (approximately 15-17 decimal digits), and certain arithmetic operations can lead to rounding errors. Therefore, it is important to be aware of these limitations when working with large numbers or performing precise calculations.

Data type: Boolean

Boolean data type is used to represent logical values - true or false. It is a primitive data type, meaning that it is not an object and does not have any methods or properties.

Examples of Boolean values in JavaScript:

```
let isTrue = true;
let isFalse = false;
```

In addition to these literal values, Boolean values can also be the result of logical expressions. For example:

```
let x = 10;
let y = 5;

let greaterThan = x > y; // true
let lessThan = x < y; // false
let isEqual = x === y; // false
let notEqual = x !== y; // true
```

Here, the expressions x > y, x < y, x === y, and x !== y all evaluate to Boolean values - either true or false - based on the comparison being made.

Boolean values are often used in conditional statements to control the flow of a program. For example:

```
let age = 18;

if (age >= 18) {
  console.log("You are old enough to vote.");
} else {
  console.log("You are not old enough to vote.");
}
```

Basics Of Objects

An object is a data type that represents a collection of related data and/or functionality. Objects are used to organize and store data in a structured way.

Important features of objects in JavaScript:

- Properties: Objects have properties, which are key-value pairs. The key is a string (or symbol), and the value can be any data type, including other objects. Properties are accessed using dot notation or bracket notation. Example:

```
let person = {
  name: "John",
  age: 30,
  address: {
    street: "123 Main St",
    city: "Anytown",
    state: "CA"
  }
};

console.log(person.name); // "John"
console.log(person.address.city); // "Anytown"
```

- Methods: Objects can have methods, which are functions that are properties of the object. Methods are accessed using dot notation. Example:

```
let person = {
  name: "John",
  age: 30,
  sayHello: function() {
    console.log(`Hello, my name is ${this.name}`);
  }
};

person.sayHello(); // "Hello, my name is John"
```

- Constructors: Objects can be created using constructor functions, which are functions that are used to create new objects. Constructor functions use the new keyword to create a new instance of the object. Example:

```
function Person(name, age) {
  this.name = name;
  this.age = age;
  this.sayHello = function() {
    console.log(`Hello, my name is ${this.name}`);
  };
}

let person1 = new Person("John", 30);
let person2 = new Person("Jane", 25);

console.log(person1.name); // "John"
console.log(person2.age); // 25
person1.sayHello(); // "Hello, my name is John"
```

- Prototypes: Objects can inherit properties and methods from a prototype object, which is a template object that is used to create new objects. Prototypes are accessed using the prototype property of a constructor function. Example:

```
function Person(name, age) {
  this.name = name;
  this.age = age;
}

Person.prototype.sayHello = function() {
  console.log(`Hello, my name is ${this.name}`);
};

let person1 = new Person("John", 30);
let person2 = new Person("Jane", 25);

console.log(person1.name); // "John"
console.log(person2.age); // 25
person1.sayHello(); // "Hello, my name is John"
```

Some common manipulations related to objects in JavaScript include:

1. Adding and deleting properties:

```
let person = {
  name: "John",
  age: 30
};

person.address = "123 Main St";
delete person.age;

console.log(person); // { name: "John", address: "123 Main St" }
```

2. Looping over properties:

```
let person = {
  name: "John",
  age: 30
};

for (let prop in person) {
  console.log(`${prop}: ${person[prop]}`);
}

// Output:
// name: John
// age: 30
```

3. Cloning objects:

```
let person = {
  name: "John",
  age: 30
};

let clone = Object.assign({}, person);

console.log(clone); // { name: "John", age: 30 }
```

Loop / Iterate through JS Objects

There are different ways to loop through JS objects.

1. for...in loop: This loop is used to iterate over the properties of an object. Example:

```
let person = {
  name: "John",
  age: 30,
  gender: "male"
};

for (let prop in person) {
  console.log(`${prop}: ${person[prop]}`);
}

// Output:
// name: John
// age: 30
// gender: male
```

2. Object.keys() method: This method returns an array of the object's own enumerable properties (keys), which can be looped through using a for...of loop or forEach() method. Example:

```
let person = {
  name: "John",
  age: 30,
  gender: "male"
};

let keys = Object.keys(person);

for (let key of keys) {
  console.log(`${key}: ${person[key]}`);
}

// Output:
// name: John
// age: 30
// gender: male
```

3. Object.values() method: This method returns an array of the object's own enumerable property values, which can be looped through using a for...of loop or forEach() method. Example:

```
let person = {
  name: "John",
  age: 30,
  gender: "male"
};

let values = Object.values(person);

for (let value of values) {
  console.log(value);
}

// Output:
// John
// 30
// male
```

4. Object.entries() method: This method returns an array of the object's own enumerable property key-value pairs as arrays, which can be looped through using a for...of loop or forEach() method. Example:

```
let person = {
  name: "John",
  age: 30,
  gender: "male"
};

let entries = Object.entries(person);

for (let [key, value] of entries) {
  console.log(`${key}: ${value}`);
}

// Output:
// name: John
// age: 30
// gender: male
```

Basics Of Array

> An array is a collection of elements of any data type that are stored in contiguous memory locations. Arrays are commonly used to store and organize related data.

Arrays in JavaScript are stored as objects in memory, with each element being a property of the array object. The index of each element is used as the property name.

Example of creating an array in JavaScript:

```
let myArray = [1, 2, 3, 4, 5];
```

In the example above, we have created an array called myArray that contains five elements of type number.

To access elements of an array in JavaScript, we use square brackets with the index of the element we want to access. The first element of an array has an index of 0, the second element has an index of 1, and so on.

- Example of accessing elements of an array in JavaScript:

```
let myArray = [1, 2, 3, 4, 5];

console.log(myArray[0]); // Output: 1In the example above, we are accessing the first and third elements of the myArray array using their index.
```

- We can also change the value of an array element by assigning a new value to it using its index: console.log(myArray[2]); // Output: 3

```
let myArray = [1, 2, 3, 4, 5];

myArray[0] = 6;

console.log(myArray); // Output: [6, 2, 3, 4, 5]
```

In the example above, we are changing the value of the first element of the myArray array from 1 to 6.

- We can also add elements to an array using the push() method, remove elements using the pop() method, or insert elements using the splice() method.

Examples of using these methods:

```
let myArray = [1, 2, 3, 4, 5];

myArray.push(6);
console.log(myArray); // Output: [1, 2, 3, 4, 5, 6]

myArray.pop();
console.log(myArray); // Output: [1, 2, 3, 4, 5]

myArray.splice(2, 0, 7);
console.log(myArray); // Output: [1, 2, 7, 3, 4, 5]
```

In the examples above, we are adding an element to the end of the myArray array using push(), removing the last element of the array using pop(), and inserting an element at index 2 using splice().

JavaScript Execution

JavaScript is an interpreted language, which means that it executes code directly, without needing to compile it beforehand. When a program or script is executed in JavaScript, there are several steps involved in the process.

JavaScript Engine

The JavaScript engine is just one of the many components that make up a modern web browser. The browser engine is the main component responsible for rendering web pages, and it consists of several different modules that work together to process HTML, CSS, and JavaScript code.

Components of JavaScript Engine

1. Parser: The parser component of the engine is responsible for converting JavaScript source code into an Abstract Syntax Tree (AST). This is the first step in the execution process, and it enables the engine to analyze and optimize the code.

2. Interpreter: The interpreter component of the engine reads the AST generated by the parser and executes the code. It does this by converting the AST into machine code, which can be executed by the computer's CPU.

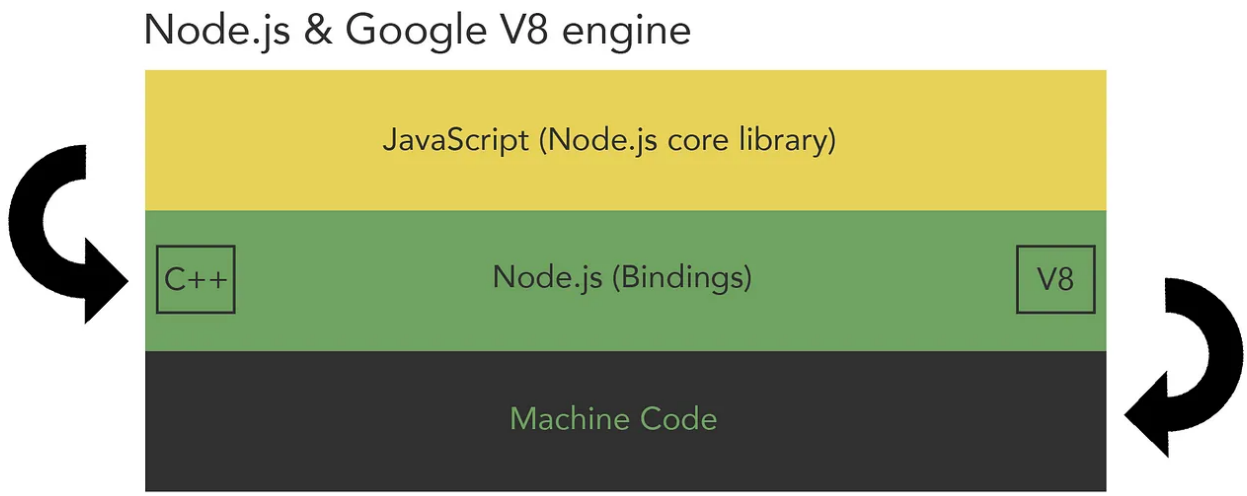
3. Compiler: The compiler component of the engine is responsible for optimizing the code by analyzing its structure and behavior. The engine uses a technique called "Just-in-time" (JIT) compilation to improve the performance of JavaScript code. This involves analyzing the code at runtime and generating optimized machine code that can be executed more quickly.

4. Garbage Collector: The garbage collector component of the engine is responsible for managing memory usage in the JavaScript program. It monitors the objects created by the program and frees up memory that is no longer in use.

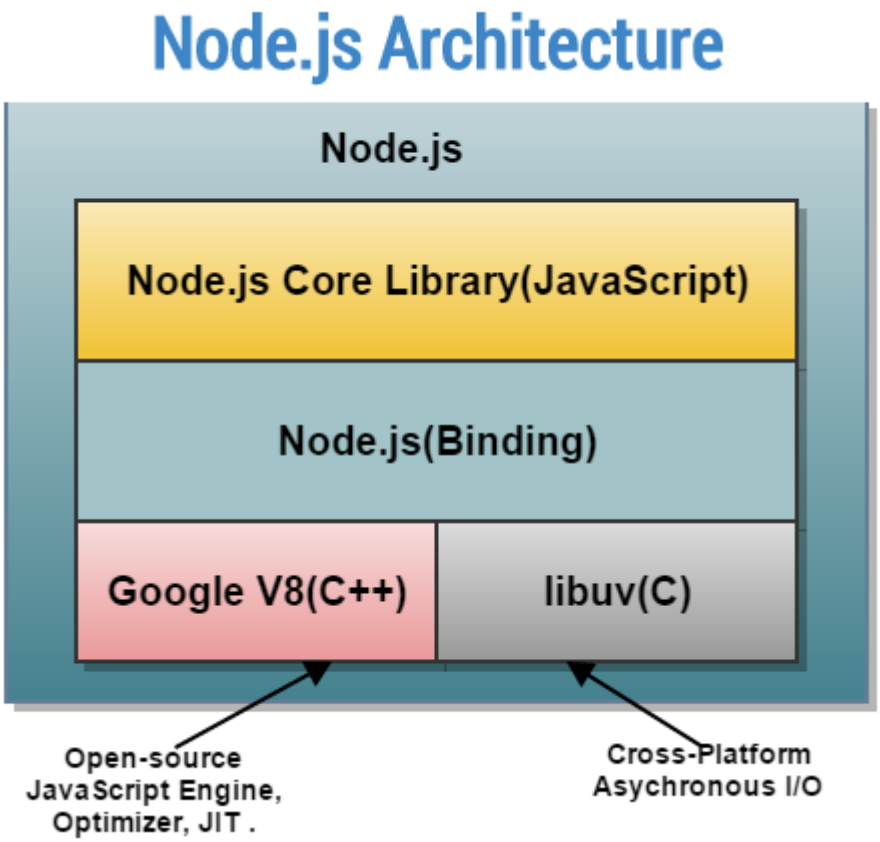
5. Profiler: The profiler component of the engine is used to monitor the performance of JavaScript code. It provides detailed information about the execution time of different parts of the code, allowing developers to identify and optimize performance bottlenecks.

Chrome V8 Engine

JavaScript Chrome V8 engine is an open-source JavaScript engine developed by Google that powers the Google Chrome web browser and other web applications. It is written in C++ and is designed to be fast, efficient, and scalable. The V8 engine is made up of several components, each of which plays a specific role in the JavaScript execution process.



Components of Chrome V8 Engine



1. NodeJs Core or JavaScript Core: This contains core of JS functionalities and utilities.

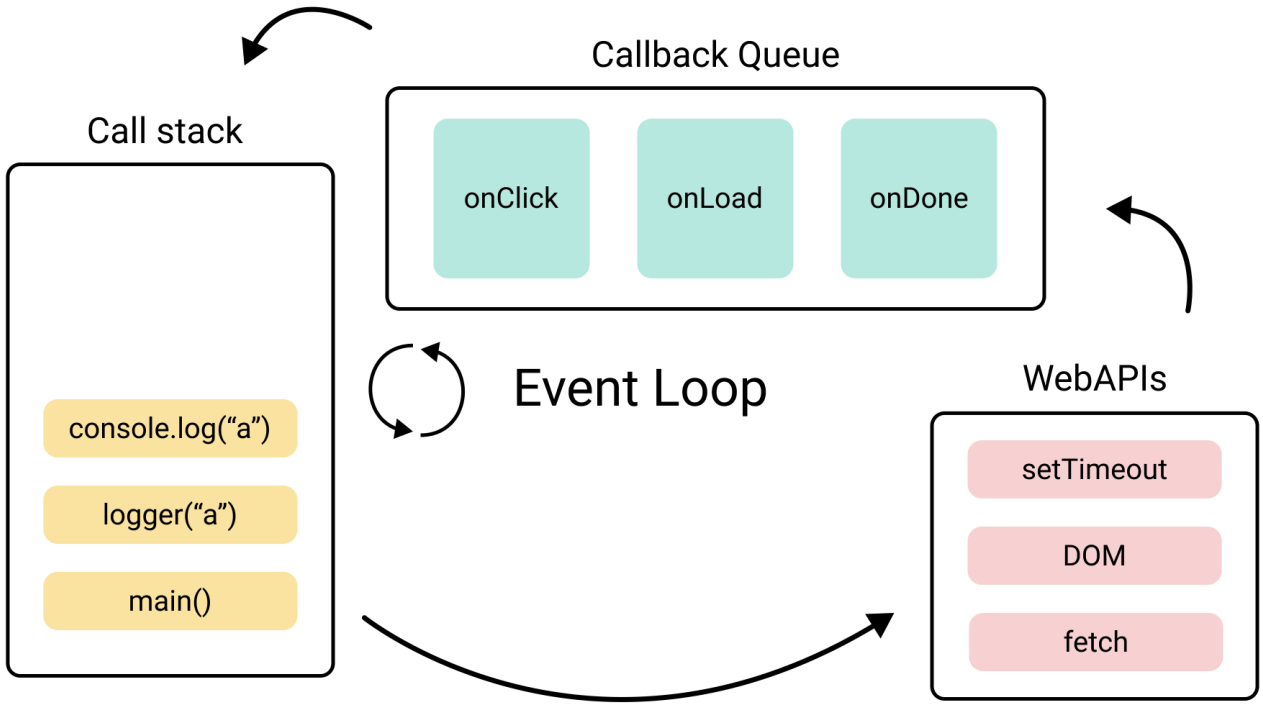
2. NodeJs Bindings: This helps to process functions from JS or NodeJS to C++.
3. Google V8 (C++): This is the core of JS engine which helps to the run the code exactly.
4. libuv: This is the pool of utilities or also worker threads which helps to run additional functionality or asynchronous tasks.

Note: when talking only about JS (not NodeJS) the libuv does not exist in the engine and it is replaced by Browser Web APIs.

JavaScript Event Loop

The event loop in JS is a single main thread which is responsible for the execution of the code.

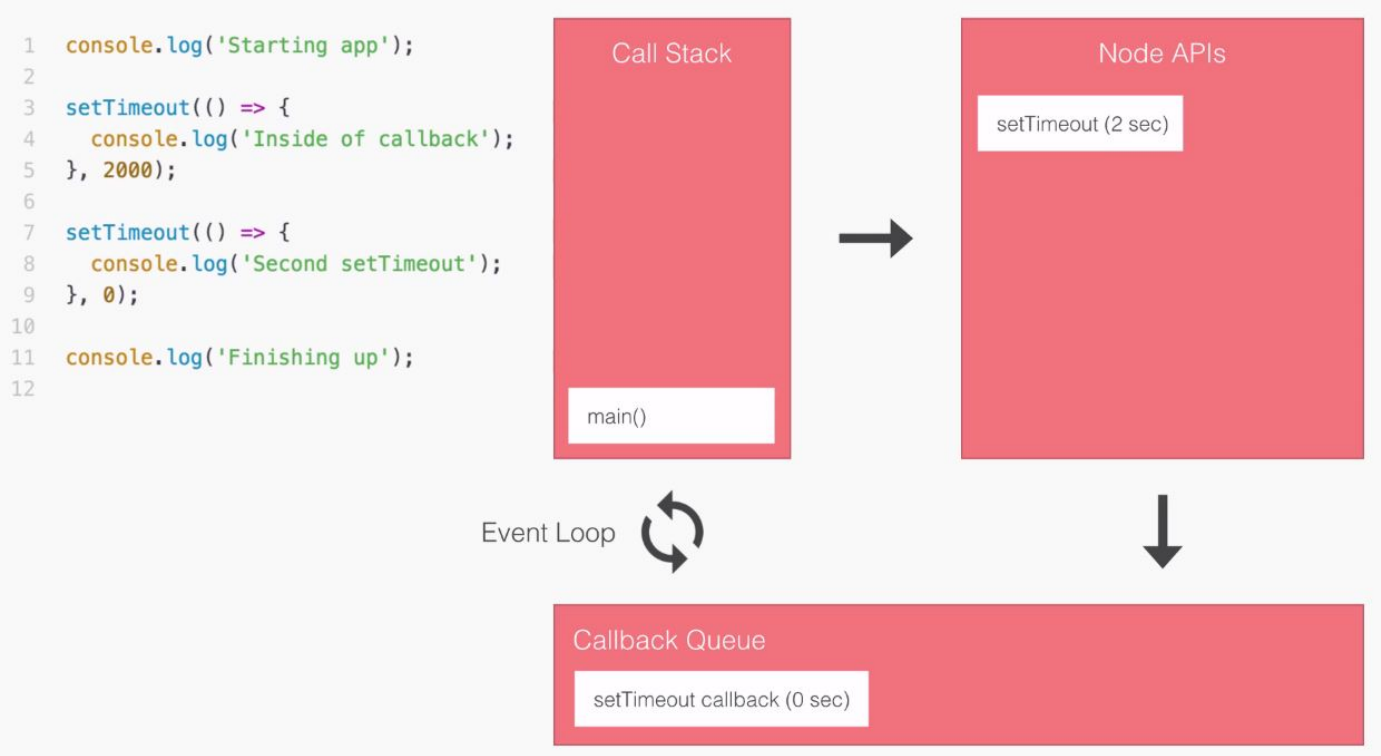
Major components of JS event loop



- Main Thread (Event Loop) : responsible for execution of the flow
- Call Stack : the stack which hold the lastest part of the code which needs to be executed
- Callback Queues : holds the response generated from Web APIs once the task/function is completed
- Web APIs : runs the additional functionality required and handles the asynchronous aspect of the execution

Callbacks

What is callbacks ?



- Callbacks are function which are passed as arguments in other functions and these callback is trigger when the task/request is completed.
- A triggered callback will contain, error if any, data and other options if specified.
- This callback returns the data to main process.

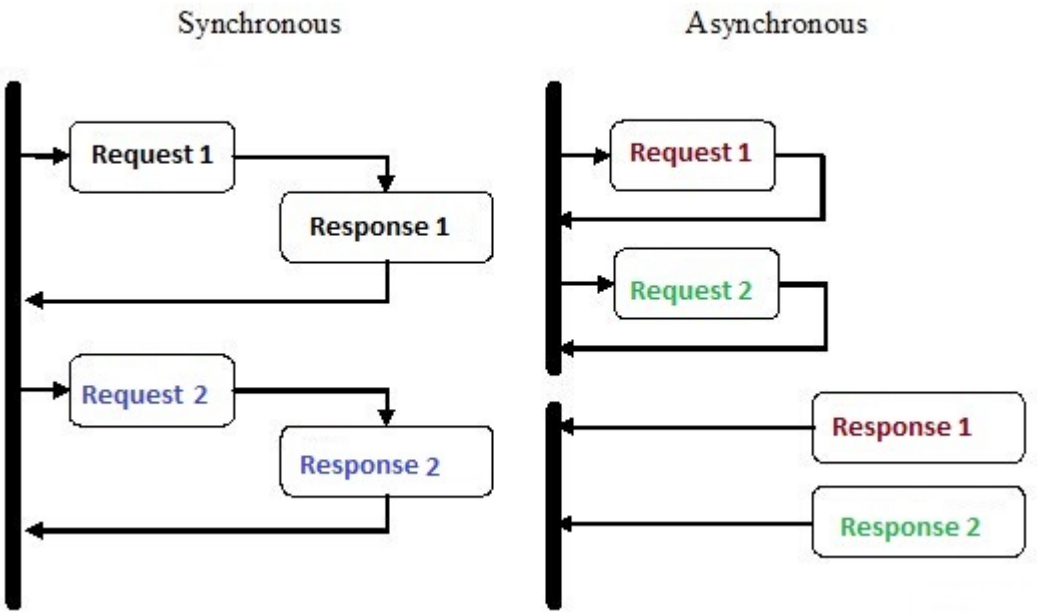
Drawback of Callbacks

Callback Hell

```
getData(function(a) {
     getMoreData(a, function(b) {
       getMoreData(b, function(c) {
         getMoreData(c, function(d) {
           getMoreData(d, function(e) {
             // ...
           })
         })
       })
     })
   })
```

- Even though callbacks provide flexibility to handle async situation but it is a hard concept to understand.
- Using callbacks can lead code unreadability, it becomes difficult to keep track on things.

Asynchronous Programming



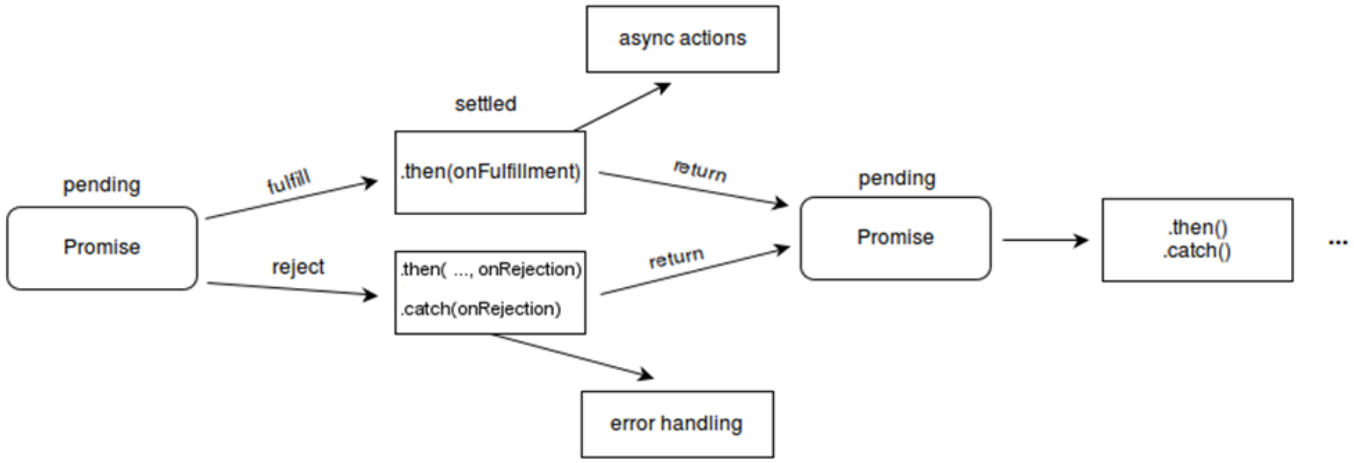
- In sync operations, the process completes the task then it hands over back the main control, till then no other task can be performed.
- In async operations, the process is handed over to worker threads and they completes the task and returns the output to the process back.
- In this while the main process is free and hence it can perform other tasks.

Promises

Asynchronous programming can be handled with promises instead of creating a lot of callbacks in functions.

When there is a lengthy chain of callback functions used to implement numerous asynchronous tasks, it becomes difficult to maintain the application. We can utilise promise to address this issue. It is a neat technique to use JavaScript to do asynchronous programming.

A Promise in JavaScript is an object representing the eventual completion or failure of an asynchronous operation and its resulting value. Promises provide a cleaner and more organized way to write asynchronous code than using callbacks. They also allow for more advanced flow control and error handling.



Here's an example of how to use Promises in JavaScript:

```
function fetchUserData(userId) {
  return new Promise((resolve, reject) => {
    const url = `https://api.example.com/users/${userId}`;
    const xhr = new XMLHttpRequest();
    xhr.open('GET', url);
    xhr.onload = function() {
      if (xhr.status === 200) {
        const data = JSON.parse(xhr.responseText);
        resolve(data);
      } else {
        reject(new Error('Failed to fetch user data'));
      }
    };
    xhr.onerror = function() {
      reject(new Error('Network error'));
    };
    xhr.send();
  });
}

fetchUserData(123)
  .then((data) => {
    console.log('User data:', data);
  })
  .catch((error) => {
    console.error('Error:', error);
  });
```

In this example, we define a function fetchUserData that returns a Promise. Inside the Promise, we make an AJAX request to fetch user data from a server. If the request is successful, we call the resolve method and pass the resulting data as an argument. If the request fails, we call the reject method and pass an error object as an argument.

We then call the fetchUserData function with a user ID of 123, and use the then method to handle the resolved Promise with a callback function that logs the user data in then block and error in catch block.

Server Requests and HTTP

The core of the Internet is making requests. All you are doing when you browse the internet is asking servers, which are essentially connected computers, for information. It serves as the basis for how webpages function. A straightforward example is when a user types "www.google.com" and is directed to the Google search page. A web server receives a request and responds with the files needed to create the page, which the browser will subsequently display as a web page. This is made feasible by the HTTP protocol (HyperText Transfer Protocol).

> The HTTP protocol is described as follows on the Mozilla Development Network website: HTTP is a protocol which allows the fetching of resources, such as HTML documents. It is the foundation of any data exchange on the Web and it is a client-server protocol, which means requests are initiated by the recipient, usually the Web browser. A complete document is reconstructed from the different sub-documents fetched, for instance text, layout description, images, videos, scripts, and more.

To understand requests in JS we need to learn about following:

AJAX

AJAX stands for Asynchronous JavaScript and XML. It is a technique that allows web pages to be updated asynchronously without requiring a full page refresh. AJAX is commonly used for fetching data from the server and updating parts of a web page dynamically without requiring a full page reload.

XMLHttpRequest (XHR)

XMLHttpRequest (XHR) is the JavaScript API used to make AJAX requests. It allows JavaScript to send HTTP requests to the server and receive responses asynchronously without blocking the user interface. XMLHttpRequest was originally designed to work with XML, but it can also handle other data formats, such as JSON and plain text.

Here's an example of how to use XMLHttpRequest to fetch data from a server:

```
const xhr = new XMLHttpRequest();
xhr.open('GET', 'https://api.example.com/data');
xhr.onreadystatechange = function() {
  if (xhr.readyState === 4 && xhr.status === 200) {
    const data = JSON.parse(xhr.responseText);
    console.log(data);
  }
};
xhr.send();
```

In this example, we create a new instance of the XMLHttpRequest object and set the request method to 'GET'. We then set the onreadystatechange property to a callback function that will be called each time the readyState property of the XHR object changes. The callback function checks if the readyState is equal to 4 (which indicates that the request has been completed) and if the status is equal to 200 (which indicates that the request was successful). If both conditions are true, the responseText property of the XHR object is parsed as JSON and logged to the console.

This is just a basic example of how to use XMLHttpRequest to make an AJAX request. There are many other options and properties available that can be used to customize the request and handle errors or timeouts.

NodeJS

NodeJS is an open-source, cross-platform, server-side runtime environment that executes JavaScript code outside of a web browser. NodeJS uses the V8 JavaScript engine from Google Chrome to interpret and run JavaScript code on the server-side.

Important features of NodeJS:

1. Asynchronous and Event-Driven: NodeJS is designed to handle asynchronous programming with an event-driven architecture, which means it can handle multiple connections and requests at the same time without blocking the execution of other code.
2. Fast and Scalable: NodeJS is built on the V8 JavaScript engine, which is known for its speed and performance. It also has a built-in clustering module, which allows developers to scale applications across multiple CPU cores.
3. Lightweight and Efficient: NodeJS is a lightweight and efficient runtime environment that uses very little memory, making it ideal for building applications that require high scalability and performance.
4. Large and Active Community: NodeJS has a large and active community of developers who contribute to its development and maintenance. This means that there are many libraries and tools available for NodeJS that can be used to simplify development and speed up the process.
5. Easy to Learn and Use: NodeJS is easy to learn and use, especially if you already have experience with JavaScript. It also has a simple and intuitive API that makes it easy to build complex applications.

JavaScript and NodeJS: What's the difference?

JavaScript and NodeJS are often confused as the same thing, but they are actually different in several ways.

JavaScript is a programming language that was originally developed for use in web browsers to make web pages interactive. It is primarily used on the client-side, which means it is executed within a user's web browser. JavaScript is used for creating dynamic web applications, including user interface components and client-side validation.

NodeJS, on the other hand, is a runtime environment for executing JavaScript code on the server-side. It is built on the V8 JavaScript engine and provides an event-driven, non-blocking I/O model that makes it highly scalable and efficient for building server-side applications. NodeJS is used for building web servers, command-line tools, and other server-side applications.

Differences between JavaScript and NodeJS:

1. Environment: JavaScript is primarily used on the client-side within web browsers, while NodeJS is used on the server-side.
2. Modules: JavaScript supports modules, but they are loaded asynchronously, while NodeJS has built-in support for modules and they are loaded synchronously.
3. I/O: JavaScript is not designed for I/O operations, while NodeJS provides a non-blocking I/O model that makes it highly efficient for I/O-intensive applications.
4. APIs: JavaScript has a limited set of APIs, while NodeJS provides a comprehensive set of APIs for building server-side applications.

NPM

NPM stands for Node Package Manager, which is a package manager for the NodeJS ecosystem. It is used for installing, managing, and sharing packages of reusable code for NodeJS applications.

NPM comes bundled with NodeJS, so when you install NodeJS, you also get NPM. NPM provides access to a vast library of open-source packages that can be easily installed and used in NodeJS projects.

Common commands used in NPM for NodeJS:

1. npm init: This command is used to create a new NodeJS project and generate a package.json file that describes the project's dependencies and configuration. Example: To create a new NodeJS project with npm init, run the following command in the terminal:

```
npm init
```

2. npm install: This command is used to install dependencies for a NodeJS project from the NPM registry. Example: To install the Express web framework and save it as a project dependency, run the following command in the terminal:

```
npm install express --save
```

3. npm uninstall: This command is used to remove a package from a NodeJS project. Example: To remove the Express web framework from a project, run the following command in the terminal:

```
npm uninstall express
```

4. npm start: This command is used to start a NodeJS project, as defined in the "start" script in the package.json file. Example: To start a NodeJS project with the "start" script defined as "node index.js", run the following command in the terminal:

```
npm start
```

5. npm uninstall --save-dev: This command uninstalls a specific package and removes it from the devDependencies section of the package.json file.

```
npm install ts --save-dev
```

6. npm run : This command runs a script that is defined in the package.json file.

```
npm run <script-name>
```

---

package.json file

In NodeJS, the package.json file is a JSON file that is used to manage the dependencies, metadata, and configuration of a NodeJS project. The package.json file is located in the root directory of the project, and it is usually the first file created when starting a new NodeJS project.

Features of the package.json file:

1. Dependency Management: The package.json file lists all the dependencies required by the project, along with their version numbers. This allows other developers to easily install and use the same dependencies in their own projects.

2. Project Metadata: The package.json file contains information about the project, such as the project name, version, description, author, and license. This metadata helps other developers understand what the project is about and how it can be used.

3. Scripts: The package.json file can define custom scripts that can be run with the npm run command. These scripts can be used for tasks such as building the project, running tests, and deploying the application.

4. Configuration: The package.json file can contain configuration settings for various tools and libraries used in the project, such as Babel or ESLint.

Example of a simple package.json file:

```
{
  "name": "my-project",
  "version": "1.0.0",
  "description": "A simple NodeJS project",
  "main": "index.js",
  "scripts": {
    "start": "node index.js"
  },
  "dependencies": {
    "express": "^4.17.1"
  }
}
```

In this example, the name, version, and description fields provide metadata about the project. The main field specifies the entry point for the project, and the scripts field defines a custom script named start that runs the index.js file. Finally, the dependencies field lists the express library as a dependency, along with its version number.

NodeJS REPL

REPL stands for "Read-Eval-Print Loop" and it is a built-in feature of NodeJS that provides a simple interactive programming environment in the command line. The REPL allows developers to test out code and experiment with NodeJS features without having to create a separate file or application.

To start the NodeJS REPL, simply type node in the command line and press enter. This will open up the REPL prompt, where you can enter and execute JavaScript code.

Examples of how to use the NodeJS REPL:

1. Basic Math Operations:

```
> 2 + 2
4
> 10 - 5
5
> 3 * 6
18
> 10 / 2
5
```

2. Defining Variables:

```
> let a = 10;
undefined
> a
10
> let b = 5;
undefined
> a + b
15
```

3. Using Built-In NodeJS Modules:

```
> const fs = require('fs');
undefined
> fs.writeFileSync('test.txt', 'Hello world!');
undefined
> fs.readFileSync('test.txt', 'utf-8')
'Hello world!'
```

In this example, we use the require function to import the built-in fs (file system) module, which provides functions for reading and writing files. We then use the fs.writeFileSync function to write the string "Hello world!" to a file named "test.txt". Finally, we use the fs.readFileSync function to read the contents of the "test.txt" file and log it to the console.

Modules

In NodeJS, a module is a reusable block of code that encapsulates related functionality and can be loaded and used in other NodeJS applications. Modules are the building blocks of NodeJS applications, and they help to keep code organized, maintainable, and scalable.

NodeJS provides two types of modules: core modules and external modules. Core modules are built-in modules that are part of the NodeJS installation, while external modules are third-party modules that can be installed and loaded using the Node Package Manager (NPM).

Example of how to create and use a simple module in NodeJS:

1. Creating a Module:

```
// myModule.js

const greeting = 'Hello, world!';

function greet() {
  console.log(greeting);
}

module.exports = {
  greet
};
```

2. In this example, we define a module that exports a greet function that logs the string "Hello, world!" to the console. The module.exports object is used to export the greet function so that it can be used in other parts of the application.

Using a Module:

```
// index.js

const myModule = require('./myModule');

myModule.greet();
```

In this example, we load the myModule module using the require function and store it in a variable called myModule. We then call the greet function from the myModule module to log the "Hello, world!" message to the console.

3. In addition to exporting functions, modules can also export variables, objects, and classes. Here's an example of how to export a variable from a module:

```
// myModule.js

const message = 'Hello, world!';

module.exports = message;
```

```
// index.js

const message = require('./myModule');

console.log(message);
```

In this example, we export the message variable from the myModule module and load it into the message variable in the index.js file using the require function. We then log the message variable to the console to output the "Hello, world!" message.

Finally, it's worth noting that NodeJS provides several built-in modules that can be used, like "path", "fs", etc.

Basic Server in NodeJS

Example of how to create a server in vanilla NodeJS without using any frameworks or libraries, and some example API calls:

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

const hostname = 'localhost';
const port = 3000;

// Create a server object
const server = http.createServer((req, res) => {
  // Set the response header content type
  res.setHeader('Content-Type', 'application/json');

  // Handle the request method and URL path
  if (req.method === 'GET' && req.url === '/') {
    res.statusCode = 200;
    res.end(JSON.stringify({ message: 'Hello, World!' }));
  } else if (req.method === 'POST' && req.url === '/api/users') {
    let body = '';
    req.on('data', chunk => {
      body += chunk.toString();
    });
    req.on('end', () => {
      const user = JSON.parse(body);
      // Save the user to the database or perform other actions
      res.statusCode = 201;
      res.end(JSON.stringify(user));
    });
  } else {
    res.statusCode = 404;
    res.end(JSON.stringify({ message: 'Not found' }));
  }
});

// Start the server
server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

1. First, we import the built-in http module in NodeJS.
2. We define the hostname and port number that our server will run on.
3. We create a server object using the createServer method of the http module. This method takes a callback function as an argument, which will be called each time a request is received.
4. Inside the callback function, we first set the response header content type to application/json.
5. We then check the request method and URL path to determine what action to take. In this example, we handle a GET request to the root path / by sending a JSON response with a message of "Hello, World!". We also handle a POST request to the path

/api/users by parsing the request body as JSON, saving the user to the database (not shown in this example), and sending a JSON response with the saved user data.
6. If the request method and URL path do not match any of our defined routes, we set the status code to 404 and send a JSON response with a message of "Not found".
7. Finally, we start the server by calling the listen method on our server object and passing in the hostname and port number to listen on. We also log a message to the console to indicate that the server is running.

This example demonstrates how to create a simple server in vanilla NodeJS and handle basic API calls using only the built-in modules. You can expand on this code to add more routes and functionality as needed.

RESTful (REST API Calls)

RESTful API is an architectural style for building APIs that use HTTP requests to perform operations on resources. REST stands for Representational State Transfer, which means that each HTTP request contains enough information to retrieve or modify a resource on the server.

In NodeJS, we can build RESTful APIs using the http module to handle HTTP requests, and using JSON to represent our resources.

Some important aspects of REST APIs:

1. Resources: Resources are the key concept in RESTful API design. A resource can be any object or data that can be accessed through a URL. Each resource has a unique identifier, known as a Uniform Resource Identifier (URI).
2. HTTP Methods: RESTful APIs use standard HTTP methods to perform operations on resources. The most commonly used HTTP methods are GET (to retrieve a resource), POST (to create a new resource), PUT (to update an existing resource), and DELETE (to delete a resource).
3. Representation of Resources: RESTful APIs use a variety of data formats to represent resources, including XML, JSON, and plain text.
4. Uniform Interface: RESTful APIs have a uniform interface, which means that the same interface is used for all resources. This makes it easy to develop and maintain APIs, as well as to integrate them with other systems.
5. Stateless: RESTful APIs are stateless, which means that the server does not maintain any information about the client between requests. Each request contains all the information needed for the server to process it.

Important features of RESTful APIs include:

1. Scalability: RESTful APIs are highly scalable because they use standard HTTP methods and can be distributed across multiple servers.
2. Flexibility: RESTful APIs can be used with any programming language and can work with any type of data format.

3. Cacheability: RESTful APIs are cacheable, which means that responses can be stored in a cache and reused for subsequent requests.
4. Security: RESTful APIs can be secured using standard HTTP security mechanisms, such as SSL/TLS and OAuth.
5. Discoverability: RESTful APIs are discoverable, which means that clients can explore the available resources and their relationships using hypermedia links.

What is an API ?

> API stands for Application Programming Interface, which is a set of rules, protocols, and tools for building software applications. In the context of JavaScript and NodeJS, an API can be thought of as a collection of functions and methods that can be used by developers to interact with external systems or services.

In NodeJS, APIs are typically used to interact with databases, file systems, network protocols, and other external services. For example, the built-in http module in NodeJS provides an API for creating and managing HTTP servers and clients, while the fs module provides an API for reading and writing files.

In JavaScript, APIs can also refer to interfaces provided by web browsers, such as the Document Object Model (DOM) API, which allows developers to manipulate HTML and XML documents using JavaScript.

APIs can be accessed through various protocols, such as REST, SOAP, or GraphQL, which define how the data is transmitted and formatted. In general, APIs provide a way for developers to access the functionality of external systems or services without having to understand their internal workings or implement them from scratch.

Examples of RESTful API calls in NodeJS:

1. Get a list of users:

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

const hostname = 'localhost';
const port = 3000;

const server = http.createServer((req, res) => {
  if (req.method === 'GET' && req.url === '/users') {
    // Retrieve the list of users from the database
    const users = [
      { id: 1, name: 'John Doe' },
      { id: 2, name: 'Jane Smith' },
      { id: 3, name: 'Bob Johnson' }
    ];
    res.setHeader('Content-Type', 'application/json');
    res.statusCode = 200;
    res.end(JSON.stringify(users));
  } else {
    res.statusCode = 404;
    res.end();
  }
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

This code defines a route for the GET request to /users, which returns a list of users as a JSON response.

2. Get a single user by ID:

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

const hostname = 'localhost';
const port = 3000;

const server = http.createServer((req, res) => {
  if (req.method === 'GET' && req.url.startsWith('/users/')) {
    const userId = req.url.split('/')[2];
    // Retrieve the user from the database by ID
    const user = { id: userId, name: 'John Doe' };
    res.setHeader('Content-Type', 'application/json');
    res.statusCode = 200;
    res.end(JSON.stringify(user));
  } else {
    res.statusCode = 404;
    res.end();
  }
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

This code defines a route for the GET request to /users/:id, where :id is a placeholder for the user ID. It extracts the user ID from the URL path and retrieves the user from the database by ID, then returns the user as a JSON response.

3. Create a new user:

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

const hostname = 'localhost';
const port = 3000;

const server = http.createServer((req, res) => {
  if (req.method === 'POST' && req.url === '/users') {
    let body = '';
    req.on('data', chunk => {
      body += chunk.toString();
    });
    req.on('end', () => {
      const user = JSON.parse(body);
      // Save the user to the database
      user.id = 4;
      res.setHeader('Content-Type', 'application/json');
      res.statusCode = 201;
      res.end(JSON.stringify(user));
    });
  } else {
    res.statusCode = 404;
    res.end();
  }
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

This code defines a route for the POST request to /users, which expects a JSON request body containing user data. It parses the request body as JSON, saves the user to the database, and returns the new user as a JSON response with a status code of 201 (Created).

Basics of ExpressJS

ExpressJS is a popular web framework for NodeJS. It provides a simple and flexible API to build web applications and APIs. ExpressJS is built on top of NodeJS, which allows developers to use the powerful features of NodeJS in their applications.

Important features of ExpressJS are:

1. Routing: ExpressJS provides a flexible and powerful routing system that allows developers to define routes for their applications. It provides a simple API to define routes based on HTTP methods and URL patterns.
2. Middleware: ExpressJS provides a middleware system that allows developers to add functionality to their applications at various stages of the request-response cycle. Middleware functions can be used to handle authentication, logging, compression, and many other tasks.
3. Templating engines: ExpressJS supports a variety of templating engines such as Pug, EJS, and Handlebars. Templating engines allow developers to create dynamic views that can be rendered on the server and sent to the client.
4. Error handling: ExpressJS provides a robust error-handling system that allows developers to handle errors in a centralized location. Developers can define error-handling middleware functions that can handle various types of errors and provide appropriate responses.
5. Modularity: ExpressJS is a modular framework that allows developers to use only the components they need. Developers can choose to use only the features they need, without having to include unnecessary modules.

Why should we use ExpressJS ?

1. Simplifies server-side development: ExpressJS provides a simple and easy-to-use interface for building web applications with NodeJS. It provides a range of tools and features that simplify server-side development, such as routing, middleware, and templating engines.
2. Supports middleware: ExpressJS supports middleware, which is a function that sits in between the request and response objects and can perform various tasks like

authentication, logging, error handling, and more. Middleware functions can be chained together to create a pipeline of functions that execute in a specific order.

3. Supports routing: With ExpressJS, you can define routes that map to specific URL paths and HTTP methods. This makes it easy to build RESTful APIs that follow a specific pattern.

4. Easily integrates with other NodeJS modules: ExpressJS is designed to be modular, which means it can easily integrate with other NodeJS modules like databases, authentication libraries, and more.

Examples of how ExpressJS can be used in MERN stack development:

- Building RESTful APIs: ExpressJS is a great choice for building RESTful APIs in the MERN stack. You can use it to define routes, handle requests and responses, and integrate with a database like MongoDB.

- Creating server-side rendering: With ExpressJS, you can easily create server-side rendered applications in the MERN stack. You can use a templating engine like EJS or Pug to render dynamic HTML pages and serve them to the client.

- Handling authentication: ExpressJS makes it easy to handle authentication in the MERN stack. You can use middleware functions like passport.js to authenticate users and protect certain routes.

- In summary, ExpressJS is a powerful and flexible web application framework that simplifies server-side development, supports middleware and routing, and easily integrates with other NodeJS modules. It is a great choice for building RESTful APIs, creating server-side rendered applications, and handling authentication in the MERN stack.

Creating a basic server in ExpressJS:

Step 1: Install ExpressJS To create a server in ExpressJS, you first need to install it. You can do this using npm by running the following command:

```
npm install express
```

Step 2: Create a server file Create a new file called server.js and add the following code:

```
const express = require('express');
const app = express();

app.get('/', (req, res) => {
  res.send('Hello World!');
});

app.listen(3000, () => {
  console.log('Server listening on port 3000');
});
```

In the above code, we first require the express module and create a new instance of express called app. Then, we define a route for the root URL (/) using the app.get() method. This route sends the text "Hello World!" as a response.

Finally, we start the server by calling the app.listen() method and passing it the port number (in this case, 3000) and a callback function to log a message when the server starts.

Step 3: Start the server Save the server.js file and run the following command to start the server:

```
node server.js
```

You should see the message "Server listening on port 3000" in the console.

Step 4: Test the server Open your web browser and navigate to http://localhost:3000/. You should see the text "Hello World!" displayed in your browser.

Request and Response in ExpressJS

> In ExpressJS, the request and response objects are two important parameters that are passed to each route handler function. The request object represents the HTTP request that was made by the client, and the response object represents the HTTP response that will be sent back to the client.

Examples of how to use the request and response objects in ExpressJS:

Example 1: Handling GET requests The following code shows how to handle a GET request for the root URL (/) and send a response back to the client:

```
app.get('/', (req, res) => {
  res.send('Hello World!');
});
```

In this example, the app.get() method defines a route for the root URL (/). The route handler function takes two parameters - req and res. The req object contains information about the incoming request, such as the request method, URL, headers, and query parameters. The res object is used to send a response back to the client. In this case, we're using the res.send() method to send the text "Hello World!" as the response body.

Example 2: Handling POST requests The following code shows how to handle a POST request for the /login URL and send a JSON response back to the client:

```
app.post('/login', (req, res) => {
  const username = req.body.username;
  const password = req.body.password;

  // Authenticate the user

  res.json({ success: true });
});
```

In this example, the app.post() method defines a route for the /login URL. The route handler function takes two parameters - req and res. The req object contains information about the incoming request, such as the request method, URL, headers, and request body. In this case, we're using the req.body object to extract the username and password values from the request body.

We then perform some authentication logic and use the res.json() method to send a JSON response back to the client with a success property set to true.

In summary, the request and response objects are two important parameters that are passed to each route handler function in ExpressJS. The request object contains information about the incoming request, while the response object is used to send a response back to the client. You can use these objects to handle various types of HTTP requests and send different types of responses, such as text, HTML, JSON, and more.

Basic API: GET one and GET all API in ExpressJS:

Step 1: Install necessary packages You'll need to install two packages for this example: express and body-parser. You can do this by running the following command:

```
npm install express body-parser
```

Step 2: Create a data source For the purposes of this example, we'll create an array of users that will serve as our data source:

```
const users = [
  { id: 1, name: 'Alice' },
  { id: 2, name: 'Bob' },
  { id: 3, name: 'Charlie' }
];
```

Step 3: Set up the Express app Create a new file called server.js and add the following code:

```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();
const port = 3000;

app.use(bodyParser.json());

app.get('/users', (req, res) => {
  res.json(users);
});

app.get('/users/:id', (req, res) => {
  const id = parseInt(req.params.id);
  const user = users.find(user => user.id === id);

  if (user) {
    res.json(user);
  } else {
    res.sendStatus(404);
  }
});

app.listen(port, () => {
  console.log(`Server listening on port ${port}`);
});
```

In this code, we require the express and body-parser packages and create a new instance of the express app. We also set the port number to 3000.

We then use the app.use() method to tell Express to use the body-parser middleware for parsing JSON data in the request body.

Next, we define two routes - one for getting all users (/users) and one for getting a single user by ID (/users/:id). The app.get() method takes two parameters - the route URL and a route handler function.

In the route handler function for /users, we simply send back the entire users array as a JSON response using the res.json() method.

In the route handler function for /users/:id, we first extract the id parameter from the URL using req.params.id. We then use the Array.find() method to search for the user with the matching ID. If we find the user, we send back a JSON response with the user data using res.json(). If we don't find the user, we send back a 404 status code using res.sendStatus().

Finally, we start the server using the app.listen() method.

Step 4: Test the API Save the server.js file and start the server by running the following command:

```
node server.js
```

You should see the message "Server listening on port 3000" in the console.

You can now test the API using a tool like Postman or by making HTTP requests directly from your client-side code. Example requests:

- GET all users: GET http://localhost:3000/users
- GET a single user by ID: GET http://localhost:3000/users/2 In response to the first request, you should see the entire users array as a JSON response. In response to the second request, you should see a JSON response with the user whose ID is 2.

That's it! You've now created a basic ExpressJS API with two routes for getting all users and getting a single user by ID.

Routing and Path

> Routing in ExpressJS refers to the process of mapping HTTP requests to corresponding functions that handle those requests. A route is a combination of a URL path and an HTTP method that is used to define an API endpoint.

In ExpressJS, routes are defined using the app.METHOD(PATH, HANDLER) syntax, where app is an instance of ExpressJS, METHOD is an HTTP method (such as GET, POST, PUT, DELETE, etc.), PATH is a URL path, and HANDLER is a function that is executed when the route is matched.

Example of how to define a basic route in ExpressJS:

```
const express = require('express');
const app = express();

app.get('/', (req, res) => {
  res.send('Hello, world!');
});
```

In this example, we create an instance of ExpressJS and define a route for the root URL path (/). We use the app.get() method to define a route handler function that sends the string "Hello, world!" as the response to any GET request to the root URL path.

Paths in ExpressJS are simply the URL paths that are used to define routes. Paths can include route parameters, which are denoted by a colon (:) followed by a parameter name.

Example of how to define a route with a path parameter in ExpressJS:

```
app.get('/users/:id', (req, res) => {
  const id = req.params.id;
  res.send(`User ${id}`);
});
```

In this example, we define a route for /users/:id, where :id is a path parameter that can be used to match any value in the URL path. We use the req.params object to access the value of the id parameter and send it back as the response.

ExpressJS also provides the ability to define routes with multiple path parameters, optional parameters, and regular expressions for more complex routing scenarios.

Example of how to define a route with multiple path parameters in ExpressJS:

```
app.get('/users/:id/:action', (req, res) => {
  const { id, action } = req.params;
  res.send(`User ${id} ${action}`);
});
```

In this example, we define a route for /users/:id/:action, where :id and :action are both path parameters. We use destructuring assignment to extract the values of both parameters from the req.params object and send them back as the response.

URL Query

In ExpressJS, a query refers to the set of key-value pairs that are passed in the URL as part of a request. The query string is the portion of the URL that comes after the question mark (?) and contains the key-value pairs separated by ampersands (&). The query can be accessed using the req.query object in an ExpressJS route handler.

In NodeJS, the query string is a built-in module that provides methods for working with query strings. The query string module provides methods for parsing and formatting URL query strings. It can be used to parse the query string from a URL and convert it to an object or string, and vice versa.

Here's an example of how to use the query string in NodeJS:

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

const query = 'name=john&age=30';
const parsedQuery = querystring.parse(query);
console.log(parsedQuery);
// Output: { name: 'john', age: '30' }

const obj = { name: 'jane', age: 25 };
const stringifiedQuery = querystring.stringify(obj);
console.log(stringifiedQuery);
// Output: 'name=jane&age=25'
```

In this example, we require the query string module and use the querystring.parse() method to parse a query string and convert it to an object. We also use the querystring.stringify() method to convert an object to a query string.

Render HTML Using ExpressJS

To render HTML files using ExpressJS, we need to use a templating engine. There are many templating engines available, but one of the most popular is EJS (Embedded JavaScript).

Here's an example of how to render an HTML file using EJS in ExpressJS:

1. Install EJS using NPM:

```
npm install ejs
```

2. Set up your ExpressJS app:

```
const express = require('express');
const app = express();

app.set('view engine', 'ejs'); // Set the view engine to EJS
app.use(express.static(__dirname + '/public')); // Set the static folder
```

3. Create a simple HTML file in the views folder (by default in ExpressJS):

```
<!-- views/index.ejs -->
<html>
  <head>
    <title>My App</title>
  </head>
  <body>
    <h1>Welcome to my app!</h1>
  </body>
</html>
```

4. Create a route in your ExpressJS app that renders the HTML file:

```
app.get('/', (req, res) => {
  res.render('index');
});
```

5. Run your ExpressJS app and navigate to http://localhost:3000 to see the rendered HTML file. Here's a complete example of an ExpressJS app that renders an HTML file using EJS:

```
const express = require('express');
const app = express();

app.set('view engine', 'ejs');
app.use(express.static(__dirname + '/public'));

app.get('/', (req, res) => {
  res.render('index');
});

app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

Note that in the above example, we set the view engine to EJS using app.set('view engine', 'ejs'), and we used res.render('index') to render the index.ejs file in the views folder. The express.static middleware is used to serve static files (like CSS, JavaScript, and images) from the public folder.

Error Handling

Error Handling in NodeJS

Error handling in NodeJS is the process of detecting and handling errors that occur during the execution of a program. In NodeJS, errors are typically represented by JavaScript Error objects.

There are several ways to handle errors in NodeJS, including try-catch blocks, error events, and callback functions. Here are some examples of error handling in NodeJS:

1. Using try-catch blocks:

```
try {
  // Some code that might throw an error
} catch (error) {
  // Handle the error here
  console.error(error);
}
```

In this example, the try block contains the code that might throw an error. If an error is thrown, the catch block will be executed, and the error object will contain information about the error.

2. Using error events:

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

const readStream = fs.createReadStream('file.txt');

readStream.on('error', (error) => {
  // Handle the error here
  console.error(error);
});
```

In this example, we're creating a Readable stream from a file using the fs module. If an error occurs while reading the file, the error event will be emitted, and we can handle the error in the callback function.

3. Using callback functions:

```
function readFile(path, callback) {
  fs.readFile(path, (error, data) => {
    if (error) {
      // Handle the error here
      return callback(error);
    }

    // Continue with normal execution
    callback(null, data);
  });
}

readFile('file.txt', (error, data) => {
  if (error) {
    console.error(error);
  } else {
    console.log(data);
  }
});
```

In this example, we're defining a readFile function that takes a file path and a callback function as parameters. The fs.readFile function is used to read the file, and if an error occurs, we call the callback function with the error object as the first argument. If no error occurs, we call the callback function with null as the first argument and the file data as the second argument. We can then handle the error or the file data in the callback function passed to readFile.

Error Handling in ExpressJS

n ExpressJS, error handling is done by middleware functions that have an additional err parameter as the first argument. Examples of error handling in ExpressJS:

- Using the built-in error handler:

```
const express = require('express');
const app = express();

// Define routes...

// Error handler
app.use((err, req, res, next) => {
  console.error(err);
  res.status(500).send('Something went wrong');
});

// Start the server
app.listen(3000, () => {
  console.log('Server started on port 3000');
});
```

In this example, we're defining an error handler middleware function that will be called whenever an error occurs in the application. The err parameter contains information about the error. We're logging the error to the console and sending a 500 status code with an error message to the client. This error handler should be defined last in the middleware chain.

- Using a custom error handler middleware function:

```
// Import the Express module
const express = require('express');

// Create a new instance of the Express application
const app = express();

// Define a middleware function that handles errors
app.use((err, req, res, next) => {
  console.log("Middleware Error Handling");
    const errStatus = err.statusCode || 500;
    const errMsg = err.message || 'Something went wrong';
    res.status(errStatus).json({
        success: false,
        status: errStatus,
        message: errMsg,
        stack: process.env.NODE_ENV === 'development' ? err.stack : {}
    })
});

// Start the server
app.listen(3000, () => {
  console.log('Server started on port 3000');
});
```

In this example, we define a middleware function that takes four arguments: err, req, res, and next. The err argument contains the error that was thrown, and the req and res arguments are the request and response objects, respectively.

We log the error to the console using console.error, and then send a response to the client with a 500 status code and a message indicating that an internal server error occurred.

Assignments

Assignment: Ecommerce Website

Requirements

- Frontend using JavaScript
- Custom CSS or Bootstrap

Description

Create a ecommerce website where you can display list (or grid) of products. Each product should be clickable and should go to new page where the products all details are diplay. Create a website with all components: Header, Footer and Main.

Add an admin panel (dashboard) where you provide CRUD operations for products. This should be separate page.

Extras: try to add cart, login, user profile if possible.

Feel free to use any themes or you can follow themes of Flipkart, Amazon or Jio Mart

---

Example Code

```
<!DOCTYPE html>
<html>

<head>
    <title>AJAX</title>
</head>

<body>
    <div id="products">

    </div>

    <script>
        const httpRequest = new XMLHttpRequest()
        const productsDiv = document.querySelector("#products")

        httpRequest.onreadystatechange = function () {
            if (httpRequest.readyState === XMLHttpRequest.DONE)
                if (httpRequest.status === 200){
                    alert(httpRequest.statusText)
                    const productsArr =
JSON.parse(httpRequest.responseText.toString())
                    console.log("response : ", productsArr)

                    productsArr.map(function(prod) {
                        productsDiv.appendChild(createProductElement(prod))
                    })

                }
                else
                    alert("some error occured")
        }

        httpRequest.open("GET", "https://fakestoreapi.com/products")

        httpRequest.send()

        function createProductElement(prod){
            const prodDiv = document.createElement("div")

            prodDiv.innerHTML = `
            <h1>${prod.title}</h1>
            <img src="${prod.image}" height="100px" />
            <p>${prod.description}</p>
            <h3>Rs. ${prod.price}</h3>
            `
            return prodDiv
        }
    </script>
</body>
</html>
```

Note: Use Fake Store API to products, cart and login data. Here is the link for Fake Store API.

Note: Make use of DOM, Array and Object functions, Callbacks, Promises, wherever required.