1

Outline

1. Couse Introduction
2. Review of JavaScript fundamentals
3. Node.js and its architecture
4. Setting up a basic development environment
5. Hello World example using Node.js

 2

2

Review Javascript fundamentals

 3

3

Javascript

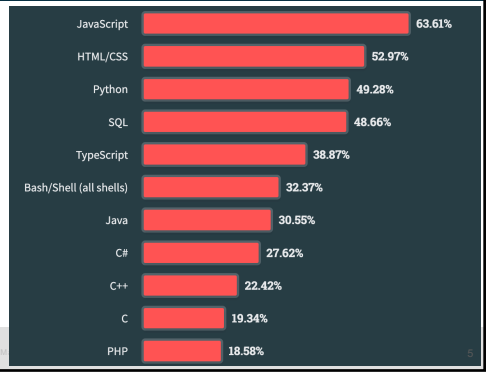
- JavaScript is a client-side scripting language that is used to make web pages interactive.
- It is interpreted by the browser, which means that it does not need to be compiled into machine code before it can be executed.
- JavaScript can be used to manipulate the DOM, add dynamic content to web pages, and create interactive effects.
- It is a powerful and versatile language that is used by millions of developers around the world.
- JavaScript is easy to learn and can be used to create a wide variety of web applications.

 4

4

Most Common Programming Language 2023

- https://survey.stackoverflow.co/2023/#programming-scripting-and-markup-languages



 5

5

Javascript Uses

- **Client-side JavaScript**:
  - Used to make web pages interactive.
  - Manipulates the DOM, adds dynamic content, and creates interactive effects.
  - Can be used to validate forms, create animations, and track user interactions.
- **Server-side JavaScript**:
  - Used to build back-end web applications and provide server-side services.
  - Can be used to process data, generate dynamic content, and communicate with databases.
  - Is often used with Node.js, a runtime environment that allows JavaScript to be run on the server.

 6

6

Javascript 2015 (ES6)

1. let and const keywords
2. Template Literals
3. Objects and JSON
4. Destructing Assignments
5. Higher order functions
6. Arrow functions

 7

7

var, let and const

 8

8

let and const keywords

- **let**: Creates a variable with block scope.
- **const**: Creates a constant variable with block scope.
- **Block scope**: Variables declared with let or const are only accessible within the block in which they are declared.
- **Reassignment**: Variables declared with let can be reassigned, while variables declared with const cannot be reassigned.
- **Hoisting**: Variables declared with let or const are not hoisted.
- **Use**: Let and const should be used instead of var for most variable declarations.

9

let and const keywords

```
function myFunction() {
        // let variable
        let x = 10;

        // const variable
        const y = 20;

        // x can be reassigned
        x = 20;

        // y cannot be reassigned
        // y = 30; // Error: Assignment to constant variable.
}
```


10

let vs. var keywords

- **let**: Block scope, reassignable
- **var**: Function scope, reassignable
- **Use**: Use let for most variable declarations

```
if (true)
{
        // a and b are local variales
        var a = 10
        let b = 20

        console.log(a, b) // 10 20
}

console.log(a) // 10 (okay)
console.log(b) // b is not defined
```


11

Template Literals

12

Template Literals

- A new feature in JavaScript that allow you to create strings that can contain embedded expressions.
- They are enclosed in backticks ` instead of double or single quotes.
- You can use template literals to create multi-line strings, interpolate variables and expressions, and escape characters.
- Template literals are a powerful tool that can be used to improve the readability and flexibility of your JavaScript code.

```
const name = "John Doe"
const age = 30
const message = `Hello, my name is ${name}. I am ${age} years old.`
```


13

Template Literals: Multiline Strings

- Template literals are particularly handy when working with multiline strings.

```
const message = `
  This is a multiline
  string using template literals.
`;

console.log(message);

// This is a multiline
// string using template literals.
```


14

Template Literals: Expression Interpolation

- You can include expressions inside ${} placeholders to dynamically insert values into your template literals:

```
const num1 = 10;
const num2 = 5;

const result = `${num1} + ${num2} = ${num1 + num2}`;
console.log(result); // Output: 10 + 5 = 15
```


15

Objects

- Objects are a way to group data and functions together.
- Object properties are key-value pairs that store data.
- Objects can be created using the {} syntax.
- Objects are a powerful tool for organizing data and behavior in JavaScript.

```
const student = {
   id: 12345,
   name: "John Doe",
   age: 20,
   score: 90
};
console.log(student)
console.log(student.name)
console.log(student.age)
```

```
{ id: 12345, name: 'John Doe', age: 20, score: 90 }
John Doe
20
```


16

Objects and JSON

17

JSON

- JSON is a lightweight data-interchange format.
- It is easy for humans to read and write.
- It is easy for machines to parse and generate.

```
const student = {
   id: 12345,
   name: "John Doe",
   age: 20,
   score: 90,
   sports: ["Soccer", "Badminton"]
};
```

The student object

```
{
   "id": 12345,
   "name": "John Doe",
   "age": 20,
   "score": 90,
   "sports": [
      "Soccer",
      "Badminton"
   ]
}
```

A JSON String from the student object

18

JSON

- JSON is a text format, so it is easy to transmit over a network.
- JSON is a standard format, so it is widely supported by programming languages and applications.
- JavaScript objects can be converted to JSON using the **JSON.stringify()** method.
- JSON can be converted to JavaScript objects using the **JSON.parse()** method.

19

Converting an object to JSON

- There are many reasons to convert from JavaScript object to JSON:
  - To transmit data over a network
  - To store data in a file
  - To pass data to a web service
  - To use data with other JavaScript libraries

20

Converting an object to JSON

- Here is an example of how to convert a JavaScript object to JSON:

```
const obj = {
   name: "John Doe",
   age: 30,
   address: "123 Main Street"
};
const json = JSON.stringify(obj);
```

- The JSON.stringify() method takes a JavaScript object as input and returns a JSON string as output. The JSON string will have the same structure as the JavaScript object.

21

Converting a JSON String to JS Object

- Here is an example of how to convert a JSON to a JavaScript object:

```
const json = '{"name":"John Doe","age":30,"address":"123 Main Street"}';
const obj = JSON.parse(json);
console.log(obj.address)
```

- The JSON.parse() method takes a JSON string as input and returns a JavaScript object as output. The JavaScript object will have the same structure as the JSON string.

22

Destructuring Assignment

23

Destructuring Assignment

- A JavaScript feature that allows you to unpack values from arrays or objects into distinct variables.

```
const user = { name: "John Doe", age: 30, address: "123 Main Street"};

const { name, age, address} = user;
console.log(name)
```

- This can be useful for making your code more concise and readable, and for avoiding the need to use nested loops or conditional statements.

24

Destructuring Assignment

- There are two types of destructuring assignment: array destructuring and object destructuring.
- Array destructuring allows you to unpack the elements of an array into distinct variables.

```
const numbers = [1, 2, 3];

const [n1, n2, n3] = numbers;
console.log(n1)
```

- Object destructuring allows you to unpack the properties of an object into distinct variables.

```
const user = { name: "John Doe", age: 30, address: "123 Main Street"};

const { name, age, address } = user;
console.log(name)
```


25

Skipping Elements

- You can skip elements in an array by using commas to separate the variables that you want to unpack. For example, the following code skips the first and third elements of the numbers array:

```
const numbers = [1, 2, 3, 4, 5];


const [, n2, , n4] = numbers;
console.log(n2, n4) // 2 4
```


26

Rest syntax

- The rest syntax is a special syntax that can be used to unpack the remaining elements of an array into a single variable. The rest syntax is represented by three dots (...)

```
const user = {
  name: "John Doe",
  age: 30,
  address: "123 Main Street"
};

const { name, ...otherProps } = user;
console.log(name) // John Doe
console.log(otherProps) // { age: 30, address: '123 Main Street' }
```

```
const numbers = [1, 2, 3, 4, 5];
const [n1, n2, ...others] = numbers;

console.log(n1, n2) // 1 2
console.log(others) // [ 3, 4, 5 ]
```


27

Renaming variables

- You can rename variables in destructuring assignment by using the colon (:) syntax. For example, the following code renames the name property of the user object to username:

```
const user = {
   name: "John Doe",
   age: 30,
   address: "123 Main Street"
};

const { name: username } = user;
console.log(username)
```


28

Destructuring Function Parameters

- A versatile feature that can greatly simplify your code when working with arrays, objects, and function parameters in JavaScript.

```
function printFullName({ first, last }) {
    console.log(first + ' ' + last);
}

const person = { first: 'Jane', last: 'Smith', age: 30 };
printFullName(person); // Jane Smith
```


29

Higher Order Functions

30

Higher Order Functions

- Higher order functions are functions that take other functions as arguments or return functions.

```
function add(a, b) { // normal function
     return a + b
}

function multiply(a, b) { // normal function
     return a * b
}

function calcuate(a, b, operation) // high order function
{
     return operation(a, b)
}

console.log(calcuate(4,5, add)) // use 'add' as a parameter
console.log(calcuate(4,5, multiply)) // use 'multiply' the third parameter
```


31

Higher Order Functions

- They are a powerful tool that can be used to abstract code and make it more reusable.
- Some common examples of higher order functions in JavaScript include map(), filter(), and reduce().
- Higher order functions can be used to simplify complex code and make it easier to understand.
- They can also be used to create more dynamic and interactive applications.
- If you are looking for ways to improve your JavaScript skills, learning about higher order functions is a great place to start.

32

Higher Order Functions

```
function square(number) {
    return number * number;
}

function map(array, callback) {
    const result = [];
    for (const item of array) {
        result.push(callback(item));
    }
    return result;
}
```

```
const array = [1, 2, 3, 4, 5];

const array = [1, 2, 3, 4, 5];
const result = map(array, square);
console.log(result); // [1, 4, 9, 16, 25]
```


33

Arrow Functions

34

Arrow functions

- Arrow functions are a concise way to write JavaScript functions.
- They are defined using the **=>** arrow syntax.
- Arrow functions do not have their own this keyword.
- They can be used in place of traditional functions in most cases.
- Arrow functions are a good choice for small, anonymous functions.
- They can also be used to create higher-order functions.

```
const add = (a, b) => a + b;
console.log(add(1, 2)); // 3
```


35

Arrow functions

- The basic syntax of an arrow function looks like this:

```
const functionName = (param1, param2) => {
    // function body
    return returnValue;
};
```

- For single-expression functions, the curly braces and return keyword can be omitted:

```
const square = (x) => x * x;
```


36

Arrow functions

- If the function takes a single parameter, the parentheses around the parameter can be omitted:

```
const greet = (name) => “Hello “ + name;
```

```
const greet = name => “Hello “ + name;
```

- If the function has no parameters, you still need to include empty parentheses:

```
const sayHello = () => "Hello!";
```


37

Arrow functions: Simplifying Callbacks

- Using an arrow function in this case eliminates the need for the function keyword and shortens the syntax, making the code more concise and readable.

```
const numbers = [1, 2, 3, 4, 5];

// Traditional function as a callback
const squared = numbers.map(function(num) {
    return num * num;
});

// Equivalent arrow function
const squaredArrow = numbers.map(x => x * x);
```


38

Arrow functions: Implicit Return

- Arrow functions automatically return the value of single expressions, allowing you to omit the return keyword and curly braces for simpler operations.

```
// Traditional function with explicit return
function multiply(a, b) {
    return a * b;
}

// Equivalent arrow function with implicit return
const multiplyArrow = (a, b) => a * b;
```


39

Concise Object Literal Functions

- Arrow functions can be used within object literals to create concise methods, making the object structure cleaner and more readable.

```
// Traditional function inside an object literal
const calculator = {
    add: function(a, b) {
        return a + b;
    },
    subtract: function(a, b) {
        return a - b;
    }
};

// Equivalent arrow functions
const calculatorArrow = {
    add: (a, b) => a + b,
    subtract: (a, b) => a - b
};
```


40

Arrow functions: Array Manipulation

- Arrow functions are great for concise operations within functions like reduce, map, and filter.

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
// Traditional function to calculate the sum of all even numbers
const evenSum = numbers.reduce(function(acc, num) {
   if (num % 2 === 0) {
      return acc + num;
   } else {
      return acc;
   }
}, 0);

// Equivalent arrow function
const evenSumArrow = numbers.reduce((acc, num) => (num % 2 === 0 ? acc + num : acc), 0);
```


41

Callbacks

42

Callbacks

- Callbacks are functions that are passed as parameters to other functions.

```
function multiply(a, b) {
    return a * b
}
function calcuate(a, b, operation)
{
    return operation(a, b)
}
console.log(calcuate(4,5, multiply)) // 'multiply' is used as a CALLBACK
```


43

Callbacks

- Callbacks can be used to handle asynchronous events, such as AJAX requests or DOM changes.
- They can also be used to pass data between functions.
- Callbacks can be anonymous or named functions.
- They are a powerful tool that can be used to improve the performance and readability of JavaScript code.

44

Callbacks

- Here's a simple example of how callbacks work:

```
function displayData(data) {
    console.log("Processing data:", data);
}

function loadData(callback) {
    // Simulate an asynchronous operation
    setTimeout(() => {
        const data = "Some fetched data";
        callback(data); // Call the provided callback function
    }, 1000);
}

loadData(displayData);
```


45

Handling Errors with Callbacks

- Callbacks can also be used to handle errors that might occur during asynchronous operations

```
function displayData(data) {
    console.log("Processing data:", data);
}

function handleError(error) {
    console.error("Error:", error);
}
```

```
function loadData(callback, errorCallback) {
    setTimeout(() => {
        const error = null; // Simulate no error
        if (error) {
            errorCallback("An error occurred");
        } else {
            const data = "Some fetched data";
            callback(data);
        }
    }, 1000);
}
loadData(displayData, handleError);
```


46

Callback Hell

47

Callback Hell

- Callback hell occurs when you have a series of asynchronous operations that depend on each other's results.
- Each operation requires a callback function to handle its completion, leading to deeply nested code structures

```
asyncOperation1(function(result1) {
    asyncOperation2(result1, function(result2) {
        asyncOperation3(result2, function(result3) {
            // ... and so on
        });
    });
});
```


48

The Drawbacks of Callback Hell

- Readability: Deeply nested code becomes hard to read and understand, leading to maintenance challenges and reduced code quality.
- Debugging: Identifying the source of errors and bugs within nested callbacks can be challenging and time-consuming.
- Error Handling: Properly handling errors across multiple layers of callbacks can be complex and error-prone.
- Scalability: As your codebase grows, callback hell can become overwhelming and hinder your ability to maintain or expand your application.

49

Mitigating Callback Hell

- Refactor your anonymous functions into named functions. This approach makes your code more modular and easier to read.

```
function handleResult1(result1) {
    asyncOperation2(result1, handleResult2);
}

function handleResult2(result2) {
    asyncOperation3(result2, handleResult3);
}

asyncOperation1(handleResult1);
```


50

Mitigating Callback Hell

- Promises provide a cleaner way to work with asynchronous operations. They allow you to chain asynchronous operations together in a more readable manner.

```
asyncOperation1()
    .then(result1 => asyncOperation2(result1))
    .then(result2 => asyncOperation3(result2))
    .then(result3 => {
        // ... Process 'result3'
    })
.catch(error => {
   // Handle errors
});
```


51

Mitigating Callback Hell

- Async/await builds upon Promises and provides a more synchronous-like syntax for handling asynchronous code.

```
async function fetchData() {
    try {
       const result1 = await asyncOperation1();
       const result2 = await asyncOperation2(result1);
       const result3 = await asyncOperation3(result2);
       // ...
     } catch (error) {
        // Handle errors
     }
}
```


52

Node.js and its architecture

53

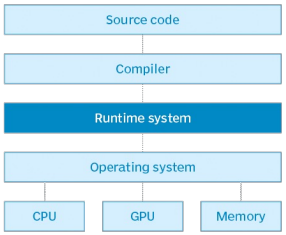
Introduction to Node.js

- Node.js is one of the JavaScript runtime environments.
- It was built in order to use JavaScript in server-sides.
- Node.js is NOT a programming language.

54

Runtime Environments

- This is the environment in which a program or application is executed.
- Different runtime environments for different languages
- Important for software development
- Ensures programs run correctly and efficiently
- Improves performance and scalability



55

Common Runtime Environments

Programming Language	Runtime Environment
Python	CPython, Jython, IronPython, PyPy, etc.
Java	Java Virtual Machine (JVM)
Kotlin	Java Virtual Machine (JVM)
Scala	Java Virtual Machine (JVM)
C#	Common Language Runtime (CLR)
PHP	PHP Runtime Environment (Phalcon)
Dart	Dart Virtual Machine (Dart VM)
JavaScript	**Node.js (for server-side), Web browsers (for client-side)**

56

Javascript Runtime Environments

Web Browser	JavaScript Runtime Engine
Google Chrome	V8
Mozilla Firefox	SpiderMonkey
Microsoft Edge (Chromium)	V8
Apple Safari	JavaScriptCore (Nitro)
Opera	V8
Brave	V8
Microsoft Edge (Legacy)	Chakra
Node.js	**V8**

57

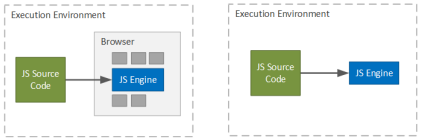
Introduction to Node.js

- Node.js is an open-source, cross-platform runtime environment that allows developers to create all kinds of server-side tools and applications in JavaScript.
- It is based on the V8 JavaScript engine, which is also used in Google Chrome and other browsers.
- Node.js is single-threaded and event-driven, which makes it very efficient for handling concurrent requests.
- It is also well-suited for building real-time applications, as it can easily handle multiple concurrent connections.
- Node.js is a popular choice for building APIs, web applications, and streaming applications.
- It is a powerful tool that can be used to build a wide variety of applications.

58

JavaScript Runtime Environments

- A JavaScript runtime environment is an environment where your JavaScript code is executed:
  - **Browser Environment**: This is the environment where JavaScript code is executed within a web browser. It includes the DOM and BOM.
  - **Node.js Environment**: This is the environment where JavaScript code is executed on a server using the Node.js runtime.



59

Browser Javascript vs. Node.js Javascript

Aspect	Browser JS	Node.js
Execution Environment	Runs in web browsers	Runs on the server-side
Global Objects	'window' object for browser context	'global' object as global context
DOM Manipulation	Interaction with DOM and BOM	Lacks direct DOM, BOM manipulation
Modules	ES modules, 'require' for modularization	CommonJS modules, 'require'
File System Access	Restricted due to security	Extensive file system access
Concurrency Model	Async operations with callbacks/Promises	Event-driven async architecture

60

Node.js Architecture

- Single-Threaded & Non-blocking I/O
- Event Loop
- Modules
- Event Emitters and Event Listeners

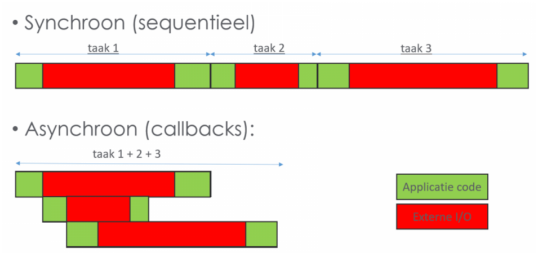
61

Node.js Single Threaded

- Node.js is single-threaded.
- Node.js uses an event loop to handle multiple requests.
- Node.js can handle a large number of concurrent requests.
- Single-threaded execution can be a bottleneck for CPU-intensive tasks.
- Node.js provides ways to offload CPU-intensive tasks to other processes or threads.

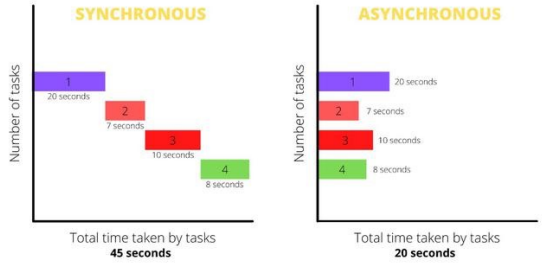
62

Single Threaded & Non-blocking IO



63

Single Threaded & Non-blocking IO



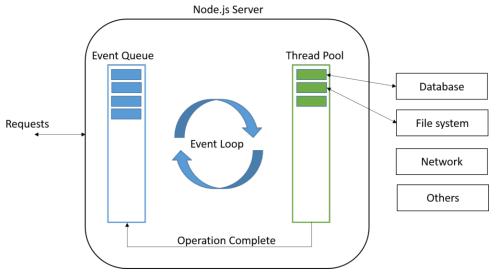
64

NonBlocking vs Multi Thread

Aspect	Multi-threaded (Java)	Single-threaded Non-blocking (Node.js)
Performance	Good for CPU-bound tasks.	Good for I/O-bound tasks.
Scalability	Can achieve high concurrency, but may face thread management overhead.	Can handle high concurrency efficiently.
Synchronization	Requires careful synchronization to avoid data races and deadlocks.	Generally less need for synchronization due to single-threaded nature and event loop.
Complexity	Higher complexity due to concurrency issues.	Generally lower complexity for handling I/O operations.
Blocking Operations	Threads can block while waiting for I/O, potentially wasting resources.	Non-blocking operations prevent resource wastage.
Context Switching	Thread context switching can be costly.	Minimal context switching due to single-threaded nature.
Resource Consumption	Threads consume more memory due to their stack and other resources.	Lower memory consumption due to single thread.
Error Handling	Complex error handling due to shared resources and threads.	Simplified error handling through callback or Promise-based mechanisms.
Development Speed	May be slower due to concurrency management and debugging.	Generally faster development due to single-threaded and event-driven nature.

65

Node.js Event Loop



66

Node.js Modules

- Built-in modules are a collection of JavaScript modules that are included in the Node.js runtime.
- These modules provide a wide range of functionality, such as file system access, networking, and cryptography.
- Node.js built-in modules can be accessed using the require() function.
- For example, to access the fs module, you would use the following code:

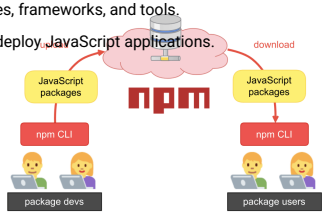
```
const fs = require('fs');
```

- Node.js built-in modules are a powerful tool for building Node.js applications.

67

Node Package Manager

- npm is a package manager for the JavaScript programming language.
- It is used to install, update, and manage JavaScript packages.
- npm has a large repository of packages, including libraries, frameworks, and tools.
- npm is a powerful tool that can help you to develop and deploy JavaScript applications.



68

Settiing Up Development Environment

69

Choose a Text Editor or an IDE

- Choosing the right text editor or IDE is crucial as it greatly impacts your coding experience. Some popular choices include:
  - Visual Studio Code (VS Code): A highly customizable and lightweight IDE with a massive extension library.
  - Sublime Text: A sleek and fast text editor with a wide range of features.
  - Atom: A hackable text editor from GitHub, known for its user-friendly interface.
  - IntelliJ IDEA: A powerful IDE for Java development, among other languages.

70

Installation

- Follow these steps to install Node.js on your machine:
  - Download: **https://nodejs.org/en/download**
  - Installation: Run the installer and follow the on-screen instructions to install Node.js (includes both Node.js and npm)
  - Verify Installation: Open a terminal or command prompt and run the following commands:

```
node -v
npm -v

v20.5.1
9.8.0
```


71

Create Your First Node.js Application

72

Create ”Hello World” App

- **Create a Project Directory**: Create a new directory for your Node.js application
- **Initialize Node.js**: Inside your project directory, run the npm init to initialize a new Node.js project.
- **Create an Entry File**: Inside your project directory, create a new JavaScript file (e.g. app.js)
- **Write Your Code**: In the app.js file, write the some code to create a simple app.
- **Run the Application**: Run the **node app.js** command to tart your Node.js application:

73

Sample Web App

```
// app.js
const http = require("http");

const server = http.createServer((req, res) => {
    res.writeHead(200, { "Content-Type": "text/plain" });
    res.end("Hello, Node.js Server!");
});

server.listen(3000, () => {
    console.log("Server is running on http://localhost:3000");
});
```


74

Sample app.js

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

fs.readFile('data.txt', 'utf8', (err, data) => {
    if (err) {
        console.log('Error reading file: ', err)
    } else {
        console.log(data)
    }
});
```


75