Node.js Basics

Node.js is a powerful, open-source, cross-platform JavaScript runtime environment that allows you to run JavaScript on the server-side, outside of the browser. It is built on Chrome's V8 JavaScript engine and is commonly used to build scalable and high-performance network applications, such as web servers and APIs.

1. Setting Up Node.js

Step 1: Install Node.js

- Download the Node.js installer from the official website.
- The installer also includes `npm` (Node Package Manager), which is used to manage Node.js packages.

Step 2: Verify Installation

After installation, open a terminal or command prompt and run:

bash

```
node -v
npm -v
```

This will display the installed versions of Node.js and npm, confirming that the installation was successful.

2. Writing Your First Node.js Program

You can start by creating a simple "Hello, World!" program.

Step 1: Create a JavaScript File

Create a file named `app.js`:

javascript

```
console.log("Hello, World!");
```

Step 2: Run the Program

In the terminal, navigate to the directory where `app.js` is located and run:

bash

```
node app.js
```

This will execute the JavaScript file, and you should see "Hello, World!" printed in the terminal.

3. Core Concepts of Node.js

a. Non-Blocking I/O

Node.js uses a non-blocking, event-driven architecture. This means that I/O operations (like reading from a file or network requests) do not block the execution of the code. Instead, they are handled asynchronously, allowing the application to process other tasks simultaneously.

b. Event Loop

The event loop is the heart of Node.js. It allows Node.js to perform non-blocking I/O operations by offloading tasks to the operating system whenever possible.

4. Creating a Simple Web Server

One of the most common use cases for Node.js is creating web servers.

Step 1: Set Up a Basic HTTP Server

In `app.js`, write the following code:

javascript

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

// Create a server
const server = http.createServer((req, res) => {
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/plain');
```

```
    res.end('Hello, World!\n');
});

// Start the server on port 3000
server.listen(3000, '127.0.0.1', () => {
    console.log('Server running at http://127.0.0.1:3000/');
});
```

Step 2: Run the Server

Run the server using:

bash

```
node app.js
```

Visit `http://127.0.0.1:3000/` in your web browser. You should see "Hello, World!" displayed.

5. Node.js Modules

a. Built-in Modules

Node.js comes with a set of built-in modules, such as `http`, `fs` (file system), `path`, and `os`.

Example: File System Module

javascript

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

fs.readFile('example.txt', 'utf8', (err, data) => {
    if (err) {
        console.error(err);
        return;
    }
    console.log(data);
```

```
});
```

This code reads the contents of `example.txt` asynchronously and prints it to the console.

b. Creating Your Own Modules

You can create your own modules by exporting functions, objects, or values.

Example: Custom Module Create a file `math.js`:

javascript

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

module.exports = add;
```

Then, use it in `app.js`:

javascript

```
const add = require('./math');

console.log(add(2, 3)); // Outputs: 5
```

c. NPM (Node Package Manager)

NPM is the default package manager for Node.js, allowing you to install third-party packages.

Example: Installing and Using Express Express is a popular web framework for Node.js.

Initialize a New Project:
bash

```
npm init -y
```

1. This creates a `package.json` file.

Install Express:
bash

```
npm install express
```

2.

Create a Simple Express Server:
javascript

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

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

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

3.

Run the Server:
bash

```
node app.js
```

4.

Visit `http://localhost:3000/` to see "Hello, World!" served by the Express server.

6. Asynchronous Programming

Node.js heavily relies on asynchronous programming, primarily using callbacks, promises, and async/await.

a. Callbacks

A callback is a function passed as an argument to another function, which is then executed after the completion of that function.

Example:

javascript

```
function fetchData(callback) {
    setTimeout(() => {
        callback("Data received");
    }, 2000);
}

fetchData((data) => {
    console.log(data); // Outputs: Data received after 2 seconds
});
```

b. Promises

Promises provide a cleaner way to handle asynchronous operations, especially when dealing with multiple operations.

Example:

javascript

```
const fetchData = () => {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve("Data received");
        }, 2000);
    });
};
```

```
fetchData().then((data) => {
    console.log(data); // Outputs: Data received after 2 seconds
}).catch((err) => {
    console.error(err);
});
```

c. Async/Await

Async/await is syntactic sugar over promises, making asynchronous code look more like synchronous code.

Example:

javascript

```
const fetchData = () => {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve("Data received");
        }, 2000);
    });
};

const getData = async () => {
    const data = await fetchData();
    console.log(data); // Outputs: Data received after 2 seconds
};

getData();
```

7. Handling Errors

Proper error handling is crucial in Node.js to avoid crashing your application.

Example: Try-Catch with Async/Await:

javascript

```
const fetchData = () => {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            reject("Error occurred");
        }, 2000);
    });
};

const getData = async () => {
    try {
        const data = await fetchData();
        console.log(data);
    } catch (err) {
        console.error(err); // Outputs: Error occurred
    }
};

getData();
```

Summary

- **Setup:** Install Node.js and npm, and verify the installation.
- **Core Concepts:** Understand the non-blocking I/O and event-driven architecture of Node.js.
- **Basic Web Server:** Use the `http` module to create a simple server.
- **Modules:** Utilize built-in modules, create custom modules, and manage packages with npm.
- **Asynchronous Programming:** Handle asynchronous tasks using callbacks, promises, and async/await.
- **Error Handling:** Ensure your application handles errors gracefully.