I can provide you with an introduction to Node.js, its advantages and disadvantages, the Node.js process model, the traditional web server model, and a step-by-step installation guide. However, I can't directly provide you with a document in DOC format. I can provide the content, and you can copy and paste it into a document editor of your choice, such as Microsoft Word or Google Docs. Here's the information you requested:

Introduction to Node.js
Node.js is an open-source, server-side runtime environment that is built on Chrome's V8 JavaScript engine. It allows developers to run JavaScript on the server, enabling them to build scalable, networked applications. Node.js is known for its non-blocking, event-driven architecture, making it well-suited for building real-time applications and high-performance web servers.

Advantages of Node.js
1. **High Performance**: Node.js is built on a non-blocking, asynchronous model, which makes it highly performant and efficient, especially for I/O-intensive applications.

2. **Single Language**: Using JavaScript on both the client and server sides can simplify development, code sharing, and maintenance.

3. **Large Ecosystem**: Node.js has a vast ecosystem of libraries and packages available through npm, making it easy to find solutions for various tasks.

4. **Scalability**: Node.js is well-suited for building scalable applications, thanks to its event-driven architecture.

5. **Community and Support**: It has a strong and active community with ample online resources and support.

Disadvantages of Node.js
1. **Not Ideal for CPU-Intensive Tasks**: Node.js may not be the best choice for CPU-bound tasks, as it's primarily designed for I/O-bound operations.

2. **Callback Hell**: Managing callbacks in deeply nested code can lead to callback hell, which can be challenging to maintain.

3. **Limited Multithreading**: Node.js is single-threaded, which means it may not fully utilize multicore processors without additional effort.

Node.js Process Model
Node.js uses a single-threaded event loop to handle multiple concurrent connections asynchronously. This means that it can efficiently handle many connections without the need for creating a separate thread for each.

1. **Event Loop**: Node.js' event loop continuously checks the event queue for incoming events and processes them one by one.

2. **Non-Blocking I/O**: When Node.js performs I/O operations, it does not block the event loop. It delegates I/O tasks to the underlying system and continues to process other events.

3. **Callbacks**: Callback functions are used to handle the results of asynchronous operations, ensuring that the event loop is not blocked.

Traditional Web Server Model
In traditional web server models (e.g., Apache), each incoming connection spawns a new thread or process to handle it. This approach can be resource-intensive and is less efficient when dealing with many concurrent connections.

Installation of Node.js (Step-by-Step)

Follow these steps to install Node.js on your system:

1. **Download Node.js**:
   - Visit the official Node.js website (https://nodejs.org/).
   - Download the recommended version for your operating system (e.g., Windows, macOS, or Linux).

2. **Install Node.js**:
   - Run the installer you downloaded.
   - Follow the installation prompts, which typically involve accepting the license agreement and choosing the installation directory.

3. **Verify Installation**:
   - Open a command prompt or terminal.
   - Run the following commands to verify the installation:
     ```
     node -v
     npm -v
     ```
     These commands should display the installed Node.js version and npm (Node Package Manager) version.

Node.js is now installed on your system, and you can start building applications using it.

To print today's date and time using the Node.js REPL (Read-Eval-Print Loop), follow these steps:

1. Open your terminal.

2. Start the Node.js REPL by typing `node` and pressing Enter.

3. Use JavaScript's `Date` object to get the current date and time. Type the following command and press Enter:

```javascript
new Date().toLocaleString()
```

This command will display the current date and time in your terminal.

To print the given pattern in JavaScript, you can use nested loops. Here's a program to print the pattern:

```javascript
const n = 5; // Number of rows

for (let i = 1; i <= n; i++) {
  let row = '';

  for (let j = 1; j <= n - i + 1; j++) {
    row += j + ' ';
  }

  console.log(row);
}
```

Copy and paste this code into a JavaScript file (e.g., `pattern.js`) and run it using Node.js by executing the following command in your terminal:

```shell
node pattern.js
```

This program will print the desired pattern:

```
```

```
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```

You can adjust the value of `n` to change the number of rows in the pattern.

Here are two Node.js programs, one to print the first 20 Fibonacci numbers and another to print prime numbers between 1 to 100. You can execute these programs in your terminal.

1. **Print the First 20 Fibonacci Numbers**:

   To print the first 20 Fibonacci numbers, you can use the following code:

   ```javascript
   function generateFibonacci(n) {
     const fib = [0, 1];

     for (let i = 2; i < n; i++) {
       fib[i] = fib[i - 1] + fib[i - 2];
     }

     return fib;
   }

   const n = process.argv[2] || 20; // Get input from the command line or default to 20
   const fibonacciSeries = generateFibonacci(n);

   console.log(`First ${n} Fibonacci numbers:`);
   console.log(fibonacciSeries.join(', '));
   ```

   To run this program and print the first 20 Fibonacci numbers, use the following command:

   ```shell
   node fibonacci.js
   ```

   If you want to specify a different number of Fibonacci numbers, pass it as an argument when running the script:

   ```shell
   node fibonacci.js 15
   ```

2. **Print Prime Numbers Between 1 to 100**:

   Here's a program to print prime numbers between 1 and 100:

   ```javascript
   function isPrime(num) {
   ```

```
  if (num <= 1) {
    return false;
  }
  if (num <= 3) {
    return true;
  }
  if (num % 2 === 0 || num % 3 === 0) {
    return false;
  }
  let i = 5;
  while (i * i <= num) {
    if (num % i === 0 || num % (i + 2) === 0) {
      return false;
    }
    i += 6;
  }
  return true;
}

console.log('Prime numbers between 1 and 100:');
for (let num = 1; num <= 100; num++) {
  if (isPrime(num)) {
    console.log(num);
  }
}
```

To run this program and print prime numbers between 1 and 100, use the following command:

```shell
node primes.js
```

These programs will display the requested Fibonacci numbers and prime numbers in your terminal.

EXP-1 over

To create a Node.js application that demonstrates the use of the built-in OS module to retrieve information about the computer's operating system, you can follow these steps:

1. **Create a new directory for your project and navigate to it in your terminal**:

   ```bash
   mkdir os_module_demo
   cd os_module_demo
   ```

2. **Initialize a Node.js project by running**:

   ```bash
   npm init -y
   ```

   This will create a `package.json` file for your project.

3. **Create a JavaScript file (e.g., `os_info.js`) in the project directory**.

4. **Open the `os_info.js` file and write the code to use the OS module**. In this code, we will use five methods from the OS module to print information about the computer's operating system.

   ```javascript
   // Import the 'os' module
   const os = require('os');

   // Get the operating system's hostname
   const hostname = os.hostname();

   // Get the operating system's type
   const osType = os.type();

   // Get the operating system's platform
   const platform = os.platform();

   // Get the operating system's release
   const release = os.release();

   // Get the operating system's total memory (in bytes)
   const totalMemory = os.totalmem();
   ```

```
// Print the retrieved information
console.log('Hostname:', hostname);
console.log('Operating System Type:', osType);
console.log('Platform:', platform);
console.log('Release:', release);
console.log('Total Memory (bytes):', totalMemory);
```

5. **Save the `os_info.js` file**.

6. **Run the program using Node.js**:

   In your terminal, execute the following command to run the program and see the information about the computer's operating system:

   ```bash
   node os_info.js
   ```

   You will see the hostname, operating system type, platform, release, and total memory information printed to the terminal.

This demonstration illustrates how to use the built-in OS module in Node.js to access information about the computer's operating system. You can explore more methods and properties provided by the OS module for gathering system information.

To print "Hello" every 500 milliseconds using the Timer Module in Node.js and print the message exactly 10 times using `setInterval`, `clearInterval`, and `setTimeout`, you can follow these steps:

1. **Create a new JavaScript file (e.g., `timer_demo.js`)** in your project directory.

2. **Open the `timer_demo.js` file and write the code** to use `setInterval`, `clearInterval`, and `setTimeout`:

   ```javascript
   // Initialize a counter to keep track of the number of times "Hello" is printed
   let count = 0;

   // Create a function to print "Hello"
   function printHello() {
     console.log("Hello");

     // Increment the count
     count++;

     // Check if "Hello" has been printed 10 times
     if (count === 10) {
       // If it has, clear the interval to stop further printing
       clearInterval(intervalId);
     }
   }

   // Use setInterval to print "Hello" every 500 milliseconds
   const intervalId = setInterval(printHello, 500);

   // Use setTimeout to stop the interval after a certain time (e.g., 5 seconds)
   setTimeout(() => {
     clearInterval(intervalId);
     console.log("Printed 'Hello' 10 times. Interval stopped.");
   }, 5000);
   ```

3. **Save the `timer_demo.js` file**.

4. **Run the program using Node.js**:

   In your terminal, execute the following command to run the program:

   ```bash
   ```

```
node timer_demo.js
```

The program will print "Hello" every 500 milliseconds, and after exactly 10 times, it will stop printing and display the message "Printed 'Hello' 10 times. Interval stopped."

This code demonstrates the use of `setInterval` to print "Hello" at a specified interval, `clearInterval` to stop the interval after a specific condition is met, and `setTimeout` to ensure that the interval is stopped after a certain amount of time.

To create a custom Calculator Node.js module with functions for addition, subtraction, multiplication, and division, and then use this module in another Node.js file, follow these steps:

1. **Create a new directory for your project and navigate to it in your terminal**:

   ```bash
   mkdir calculator_app
   cd calculator_app
   ```

2. **Initialize a Node.js project by running**:

   ```bash
   npm init -y
   ```

   This will create a `package.json` file for your project.

3. **Create a JavaScript file (e.g., `calculator.js`) for your custom module**:

   ```javascript
   // calculator.js

   // Function to add two numbers
   function add(a, b) {
     return a + b;
   }

   // Function to subtract two numbers
   function subtract(a, b) {
     return a - b;
   }

   // Function to multiply two numbers
   function multiply(a, b) {
     return a * b;
   }

   // Function to divide two numbers
   function divide(a, b) {
     if (b === 0) {
       throw new Error("Division by zero is not allowed.");
     }
     return a / b;
   ```

```
}

// Export the functions as an object
module.exports = {
  add,
  subtract,
  multiply,
  divide,
};
```

4. **Create another JavaScript file (e.g., `app.js`) to use the Calculator module**:

```javascript
// app.js

// Import the Calculator module
const calculator = require('./calculator');

// Use the functions from the module
const a = 10;
const b = 5;

console.log(`Addition: ${a} + ${b} = ${calculator.add(a, b)}`);
console.log(`Subtraction: ${a} - ${b} = ${calculator.subtract(a, b)}`);
console.log(`Multiplication: ${a} * ${b} = ${calculator.multiply(a, b)}`);
console.log(`Division: ${a} / ${b} = ${calculator.divide(a, b)}`);
```

5. **Save the `calculator.js` and `app.js` files**.

6. **Run the program using Node.js**:

   In your terminal, execute the following command to run the application:

```bash
node app.js
```

   You will see the results of the addition, subtraction, multiplication, and division operations using the functions provided by the custom Calculator module.

This demonstrates how to create a custom Node.js module (calculator) and use it in another Node.js file (app.js). You can expand the calculator module with additional functions as needed for your application.

You can create a custom Circle module in Node.js with functions to find the area and perimeter (circumference) of a circle and then use it in another Node.js file. Here's how you can do it:

1. **Create a new directory for your project**:

   ```bash
   mkdir circle_module_app
   cd circle_module_app
   ```

2. **Initialize a Node.js project by running**:

   ```bash
   npm init -y
   ```

   This will create a `package.json` file for your project.

3. **Create a JavaScript file (e.g., `circle.js`) for your custom module**:

   ```javascript
   // circle.js

   // Function to calculate the area of a circle
   function calculateArea(radius) {
     return Math.PI * radius * radius;
   }

   // Function to calculate the circumference (perimeter) of a circle
   function calculateCircumference(radius) {
     return 2 * Math.PI * radius;
   }

   // Export the functions as an object
   module.exports = {
     calculateArea,
     calculateCircumference,
   };
   ```

4. **Create another JavaScript file (e.g., `app.js`) to use the Circle module**:

   ```javascript
   // app.js
   ```

```
// Import the Circle module
const circle = require('./circle');

// Use the functions from the module
const radius = 5;

const area = circle.calculateArea(radius);
const circumference = circle.calculateCircumference(radius);

console.log(`Circle with a radius of ${radius}`);
console.log(`Area: ${area.toFixed(2)}`);
console.log(`Circumference: ${circumference.toFixed(2)}`);
```

5. **Save the `circle.js` and `app.js` files**.

6. **Run the program using Node.js**:

   In your terminal, execute the following command to run the application:

   ```bash
   node app.js
   ```

   The program will calculate and display the area and circumference of a circle with a radius of 5 (you can change the radius value as needed).

This demonstrates how to create a custom Node.js module (circle) and use it in another Node.js file (app.js) to calculate and display the area and circumference of a circle. You can reuse this module for circle-related calculations in your projects.

EXP 2 OVER

To demonstrate various Node.js events using the `EventEmitter` class, you can create an application that listens for and emits events. Here's an example that demonstrates the use of events:

1. **Create a new directory for your project**:

   ```bash
   mkdir event_emitter_demo
   cd event_emitter_demo
   ```

2. **Initialize a Node.js project by running**:

   ```bash
   npm init -y
   ```

   This will create a `package.json` file for your project.

3. **Create a JavaScript file (e.g., `event_demo.js`)**.

4. **Open the `event_demo.js` file and write the code to demonstrate events**:

   ```javascript
   // Import the 'events' module
   const events = require('events');

   // Create an instance of the EventEmitter class
   const eventEmitter = new events.EventEmitter();

   // Define an event listener function
   const myEventHandler = function () {
     console.log('Custom event handler invoked');
   };

   // Add the event listener to the 'customEvent' event
   eventEmitter.on('customEvent', myEventHandler);

   // Emit the 'customEvent' event
   eventEmitter.emit('customEvent');

   // Remove the event listener from the 'customEvent' event
   eventEmitter.removeListener('customEvent', myEventHandler);
   ```

```
// Attempt to emit the 'customEvent' event after removing the listener
eventEmitter.emit('customEvent'); // This won't trigger the listener

// Define an event listener function with arguments
const withArgumentsHandler = function (arg1, arg2) {
  console.log('Event with arguments:', arg1, arg2);
};

// Add the event listener to the 'eventWithArguments' event
eventEmitter.on('eventWithArguments', withArgumentsHandler);

// Emit the 'eventWithArguments' event with arguments
eventEmitter.emit('eventWithArguments', 'Hello', 'World');

// List all event names for the EventEmitter
const eventNames = eventEmitter.eventNames();
console.log('Event Names:', eventNames);
```

5. **Save the `event_demo.js` file**.

6. **Run the program using Node.js**:

   In your terminal, execute the following command to run the application:

   ```bash
   node event_demo.js
   ```

   The program demonstrates various events, including registering an event listener, emitting events, removing event listeners, and emitting events with arguments. It also lists the event names registered with the EventEmitter.

This demonstrates how to use the `EventEmitter` class in Node.js to handle and trigger events in your applications. You can use this pattern to create event-driven applications and handle various asynchronous operations efficiently.

To create functions for sorting, reversing, and searching for an element in an array and register and trigger these functions using events, you can follow these steps:

1. **Create a new directory for your project**:

   ```bash
   mkdir array_operations_app
   cd array_operations_app
   ```

2. **Initialize a Node.js project by running**:

   ```bash
   npm init -y
   ```

   This will create a `package.json` file for your project.

3. **Create a JavaScript file (e.g., `array_operations.js`)**.

4. **Open the `array_operations.js` file and write the code to define functions and register them with events**:

   ```javascript
   const events = require('events');
   const eventEmitter = new events.EventEmitter();

   // Sample array
   const sampleArray = [5, 2, 9, 1, 5, 6, 8];

   // Function to sort the array
   function sortArray() {
     const sortedArray = [...sampleArray].sort((a, b) => a - b);
     console.log('Sorted Array:', sortedArray);
   }

   // Function to reverse the array
   function reverseArray() {
     const reversedArray = [...sampleArray].reverse();
     console.log('Reversed Array:', reversedArray);
   }

   // Function to search for an element in the array
   function searchArray(element) {
   ```

```
  const index = sampleArray.indexOf(element);
  if (index !== -1) {
    console.log(`Element ${element} found at index ${index}`);
  } else {
    console.log(`Element ${element} not found in the array`);
  }
}

// Register the functions with events
eventEmitter.on('sort', sortArray);
eventEmitter.on('reverse', reverseArray);
eventEmitter.on('search', searchArray);

// Trigger the events
eventEmitter.emit('sort');
eventEmitter.emit('reverse');
eventEmitter.emit('search', 6);
```

5. **Save the `array_operations.js` file**.

6. **Run the program using Node.js**:

   In your terminal, execute the following command to run the application:

   ```bash
   node array_operations.js
   ```

   The program will demonstrate sorting, reversing, and searching for an element in the array using events.

This demonstrates how to create functions for array operations (sorting, reversing, and searching), register these functions with events using the `EventEmitter` class, and trigger the events to perform these operations on an array. You can expand this concept to handle more array operations and events as needed.

EXP 3 over

To create a Node.js application that demonstrates the use of a timer function to display a message every 10 seconds, you can use the `setInterval` function. Here's a step-by-step guide:

1. **Create a new directory for your project**:

   ```bash
   mkdir timer_function_app
   cd timer_function_app
   ```

2. **Initialize a Node.js project by running**:

   ```bash
   npm init -y
   ```

   This will create a `package.json` file for your project.

3. **Create a JavaScript file (e.g., `timer_function.js`)**.

4. **Open the `timer_function.js` file and write the code to display a message every 10 seconds**:

   ```javascript
   // Function to display a message every 10 seconds
   function displayMessage() {
     console.log('Message displayed every 10 seconds.');
   }

   // Use setInterval to call the function every 10 seconds (10000 milliseconds)
   const intervalId = setInterval(displayMessage, 10000);

   // Stop the timer after a certain time (e.g., 30 seconds)
   setTimeout(() => {
     clearInterval(intervalId);
     console.log('Timer stopped after 30 seconds.');
   }, 30000);
   ```

5. **Save the `timer_function.js` file**.

6. **Run the program using Node.js**:

   In your terminal, execute the following command to run the application:

```bash
node timer_function.js
```

The program will display the message "Message displayed every 10 seconds" every 10 seconds, and after 30 seconds, it will stop and display "Timer stopped after 30 seconds."

This demonstrates how to use the `setInterval` function in Node.js to create a timer function that performs a specific action (in this case, displaying a message) at a regular interval (every 10 seconds). You can adjust the time intervals and actions as needed for your application.

EXP 4 over

Certainly! Here's a Node.js application that demonstrates basic file operations, including creating a file, writing to it, reading from it, and deleting it.

To run this code, make sure you have Node.js installed on your system.

1. **Create a new directory for your project**:

   ```bash
   mkdir file_handling_demo
   cd file_handling_demo
   ```

2. **Initialize a Node.js project by running**:

   ```bash
   npm init -y
   ```

   This will create a `package.json` file for your project.

3. **Create a JavaScript file (e.g., `file_operations.js`)**.

4. **Open the `file_operations.js` file and write the code to demonstrate basic file operations**:

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

   // File path
   const filePath = 'sample.txt';

   // Data to write to the file
   const contentToWrite = 'This is a sample text content.\n';

   // Create a file and write to it
   fs.writeFile(filePath, contentToWrite, (err) => {
     if (err) {
       console.error('Error creating file:', err);
     } else {
       console.log('File created and content written successfully.');

       // Read from the file
       fs.readFile(filePath, 'utf8', (err, data) => {
         if (err) {
   ```

```
        console.error('Error reading file:', err);
      } else {
        console.log('File content:', data);

        // Delete the file
        fs.unlink(filePath, (err) => {
          if (err) {
            console.error('Error deleting file:', err);
          } else {
            console.log('File deleted successfully.');
          }
        });
      }
    });
  }
});
```

5. **Save the `file_operations.js` file**.

6. **Run the program using Node.js**:

   In your terminal, execute the following command to run the application:

   ```bash
   node file_operations.js
   ```

   This program will create a file named `sample.txt`, write some content to it, read the content from the file, and then delete the file. You will see log messages indicating the success of each operation.

Make sure to adjust the `filePath` and `contentToWrite` variables if you want to use a different file path or content.

OR

Certainly! I'll break down the process into individual steps for each file operation: creating, writing, reading, and deleting a file in Node.js.

Step 1: Create a Directory
```bash
mkdir file_handling_demo
cd file_handling_demo
```

Step 2: Initialize a Node.js Project
```bash
npm init -y
```

Step 3: Create a JavaScript File
```bash
touch file_operations.js
```

Step 4: Open the `file_operations.js` File in a Text Editor

Step 5: Add Code for Creating a File and Writing Content
In the `file_operations.js` file, add the following code to create a file and write content to it:

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

// File path
const filePath = 'sample.txt';

// Data to write to the file
const contentToWrite = 'This is a sample text content.\n';

fs.writeFile(filePath, contentToWrite, (err) => {
  if (err) {
    console.error('Error creating file:', err);
  } else {
    console.log('File created and content written successfully.');
  }
});
```

Step 6: Save the `file_operations.js` File

Step 7: Run the Program to Create and Write to the File
In your terminal, execute the following command to create a file and write content to it:

```bash
node file_operations.js
```

Step 8: Add Code for Reading from the File
In the same `file_operations.js` file, add the following code to read content from the file:

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

Step 9: Save the `file_operations.js` File

Step 10: Run the Program to Read from the File
In your terminal, execute the following command to read content from the file:

```bash
node file_operations.js
```

Step 11: Add Code for Deleting the File
In the same `file_operations.js` file, add the following code to delete the file:

```javascript
fs.unlink(filePath, (err) => {
  if (err) {
    console.error('Error deleting file:', err);
  } else {
    console.log('File deleted successfully.');
  }
});
```

Step 12: Save the `file_operations.js` File

Step 13: Run the Program to Delete the File
In your terminal, execute the following command to delete the file:

```bash
node file_operations.js
```

Now, you have separate steps for creating, writing, reading, and deleting a file in Node.js. Each step represents a specific file operation, and you can run each step individually to observe the file handling operations.

EXP 5 over- EXP 6 & 7 skip

Setting up a TypeScript development environment involves several steps, including installation and configuration. Additionally, you can create TypeScript programs that include decision-making (if statements), functions, and classes with objects. Here are step-by-step instructions:

Step 1: Install Node.js and npm (if not already installed)

If you haven't already installed Node.js and npm (Node Package Manager), you can download and install them from the official Node.js website: https://nodejs.org/

Step 2: Install TypeScript Globally

Open your terminal and use npm to install TypeScript globally:

```bash
npm install -g typescript
```

This installs the TypeScript compiler (`tsc`) globally.

Step 3: Create a TypeScript Project

Create a new directory for your TypeScript project and navigate to it in the terminal.

```bash
mkdir typescript_project
cd typescript_project
```

Step 4: Initialize a TypeScript Configuration File

Run the following command to generate a `tsconfig.json` file for your project:

```bash
tsc --init
```

Step 5: Write TypeScript Programs

Now, you can create TypeScript programs that include decision-making, functions, and classes with objects. Let's create some example programs:

Example 1: Decision Making (If Statements)

Create a TypeScript file (e.g., `decision.ts`) and add the following code to demonstrate decision-making:

```typescript
// decision.ts
let num = 10;

if (num > 0) {
    console.log("Number is positive.");
} else if (num < 0) {
    console.log("Number is negative.");
} else {
    console.log("Number is zero.");
}
```

Example 2: Functions

Create a TypeScript file (e.g., `functions.ts`) and add the following code to demonstrate functions:

```typescript
// functions.ts
function add(a: number, b: number): number {
    return a + b;
}

const result = add(5, 3);
console.log(`Result of addition: ${result}`);
```

Example 3: Classes and Objects

Create a TypeScript file (e.g., `classes.ts`) and add the following code to demonstrate classes and objects:

```typescript
// classes.ts
class Person {
    constructor(public name: string, public age: number) {}

    displayInfo(): void {
        console.log(`Name: ${this.name}, Age: ${this.age}`);
    }
```

```
}

const person1 = new Person("Alice", 30);
const person2 = new Person("Bob", 25);

person1.displayInfo();
person2.displayInfo();
```

Step 6: Compile TypeScript Code

In your terminal, navigate to the project directory and compile the TypeScript code to JavaScript using the TypeScript compiler (`tsc`):

```bash
tsc decision.ts
tsc functions.ts
tsc classes.ts
```

This will generate JavaScript files (`decision.js`, `functions.js`, and `classes.js`) based on your TypeScript code.

Step 7: Run the Programs

You can now run the programs:

```bash
node decision.js
node functions.js
node classes.js
```

This completes the installation and setup of TypeScript and demonstrates the use of decision-making, functions, and classes with objects in TypeScript. You can modify the code as needed and explore more TypeScript features.

EXP-8 over

Introduction to Angular:

Angular is a popular open-source web application framework developed and maintained by Google. It's often used for building dynamic, single-page web applications. Angular offers a comprehensive solution for front-end development and provides a rich ecosystem of tools and libraries. Here's a brief introduction to Angular:

- **Angular Features:**
  - **Declarative UI:** Angular allows developers to build user interfaces declaratively using HTML and TypeScript, making it easier to understand and maintain code.
  - **Two-Way Data Binding:** Angular provides two-way data binding, which means changes in the model automatically update the view and vice versa.
  - **Component-Based Architecture:** Angular follows a component-based architecture, where the application is organized into reusable components.
  - **Dependency Injection:** Angular has a built-in dependency injection system that helps manage the application's components and services.
  - **RxJS:** Angular uses Reactive Extensions for JavaScript (RxJS) to handle asynchronous operations, making it efficient for working with data streams.

Setup for Local Development Environment:

To set up your local development environment for Angular, follow these steps:

1. **Install Node.js and npm:**
   Angular requires Node.js and npm. You can download and install them from the official website: https://nodejs.org/

2. **Install the Angular CLI:**
   The Angular CLI (Command Line Interface) simplifies the development process. You can install it globally using npm:

   ```bash
   npm install -g @angular/cli
   ```

3. **Create an Angular Project:**
   Use the Angular CLI to create a new Angular project:

   ```bash
   ng new my-angular-app
   ```

4. **Navigate to Your Project Directory:**

```bash
cd my-angular-app
```

5. **Start the Development Server:**
   You can start the development server by running:

   ```bash
   ng serve
   ```

   Your Angular application will be available at `http://localhost:4200/`.

Angular Architecture:

Angular follows a component-based architecture, which is the foundation of building Angular applications. Here's a high-level overview of the key architectural components:

- **Modules:** Angular applications are organized into modules, which group related components, services, and other functionality. The root module is called `AppModule`.

- **Components:** Components are the building blocks of an Angular application. They consist of HTML templates, styles, and TypeScript code. Each component represents a part of the user interface.

- **Templates:** Templates define how the user interface is rendered. They are written in HTML and may include Angular directives and components.

- **Services:** Services are used to encapsulate and share business logic and data across components. They can be used to make HTTP requests, manage state, and more.

- **Dependency Injection:** Angular provides a built-in dependency injection system for managing component and service dependencies.

- **Directives:** Directives are markers on the DOM elements that tell Angular to do something with the element, such as changing its appearance or behavior. Examples include `ngIf`, `ngFor`, and `ngModel`.

- **RxJS:** Angular applications often use RxJS to handle asynchronous operations, such as working with observables for handling data streams.

- **Router:** Angular provides a powerful router for creating single-page applications with multiple views and navigation.

- **Forms:** Angular has an advanced form handling system for creating and validating forms in applications.

- **HTTP Client:** Angular provides an HTTP client module for making HTTP requests and handling responses.

This is a high-level overview of Angular's architecture. As you work with Angular, you'll gain a deeper understanding of how these components come together to create robust and maintainable web applications.

To create an Angular application that demonstrates the usage of directives and pipes, follow these steps:

Step 1: Set Up Your Angular Development Environment

If you haven't already, set up your Angular development environment as described in the previous response under "Setup for Local Development Environment."

Step 2: Create a New Angular Application

Open your terminal and use the Angular CLI to create a new Angular application. Let's name it "DirectiveAndPipeDemo":

```bash
ng new DirectiveAndPipeDemo
```

Navigate to the project directory:

```bash
cd DirectiveAndPipeDemo
```

Step 3: Create a Component

Now, let's create a component to work with directives and pipes. Generate a new component named "user-list" using the Angular CLI:

```bash
ng generate component user-list
```

This will create the necessary component files.

Step 4: Update the Component

Edit the `user-list.component.html` file to add some HTML and use directives and pipes. Here's an example:

```html
<h2>User List</h2>

<!-- Using *ngFor directive to loop through users -->
<ul>
```

```
  <li *ngFor="let user of users">
    {{ user.name | uppercase }} <!-- Using uppercase pipe -->
    <span *ngIf="user.isAdmin"> (Admin)</span> <!-- Using *ngIf directive -->
  </li>
</ul>

<!-- Using *ngIf directive to conditionally display content -->
<p *ngIf="users.length === 0">No users found.</p>

<!-- Using ngClass directive to conditionally apply CSS classes -->
<button [ngClass]="{'active': isActive, 'inactive': !isActive}" (click)="toggleStatus()">Toggle
Status</button>
```

Step 5: Update the Component Class

Edit the `user-list.component.ts` file to define the component class and populate the `users` array. Also, create a method to toggle the status:

```typescript
import { Component } from '@angular/core';

@Component({
  selector: 'app-user-list',
  templateUrl: './user-list.component.html',
  styleUrls: ['./user-list.component.css']
})
export class UserListComponent {
  users = [
    { name: 'Alice', isAdmin: true },
    { name: 'Bob', isAdmin: false },
    { name: 'Charlie', isAdmin: true },
    { name: 'David', isAdmin: false }
  ];

  isActive = true;

  toggleStatus() {
    this.isActive = !this.isActive;
  }
}
```

Step 6: Use Pipes

In the component's HTML, you can see the usage of the `uppercase` pipe to transform the user names to uppercase. Angular provides various built-in pipes for transforming and formatting data. You can also create custom pipes.

Step 7: Use Directives

In the component's HTML, you can see the usage of several directives:
- `*ngFor` is used to loop through the `users` array and display user names.
- `*ngIf` is used to conditionally display elements based on the `users` array's length.
- `ngClass` is used to conditionally apply CSS classes to the button based on the `isActive` variable.

Step 8: Run the Application

Start your Angular development server using the following command:

```bash
ng serve
```

Open your web browser and navigate to `http://localhost:4200/` to see the "User List" component in action. You'll observe how directives and pipes are applied to the user data.

This application demonstrates the usage of Angular directives and pipes in a simple scenario. You can expand on these concepts and explore more directives and pipes to enhance your Angular applications.

Demonstrate features of Angular forms with a program

Angular provides a powerful and flexible system for working with forms. You can create both template-driven forms and reactive forms. I'll provide an example of a template-driven form.

Step 1: Create a New Angular Application

If you haven't already, set up your Angular development environment and create a new Angular application as described in the previous responses.

Step 2: Create a Form Component

Generate a new component named "contact-form" using the Angular CLI:

```bash
ng generate component contact-form
```

Step 3: Define the Form in the Component's HTML

Edit the `contact-form.component.html` file to define your form:

```html
<h2>Contact Form</h2>

<form #contactForm="ngForm" (ngSubmit)="onSubmit()">
  <div class="form-group">
    <label for="name">Name:</label>
    <input type="text" id="name" name="name" class="form-control"
[(ngModel)]="formData.name" required>
  </div>

  <div class="form-group">
    <label for="email">Email:</label>
    <input type="email" id="email" name="email" class="form-control"
[(ngModel)]="formData.email" required>
  </div>

  <div class="form-group">
    <label for="message">Message:</label>
    <textarea id="message" name="message" class="form-control"
[(ngModel)]="formData.message" required></textarea>
  </div>

  <button type="submit" class="btn btn-primary">Submit</button>
```

```
</form>

<div *ngIf="submitted" class="alert alert-success mt-3">
  Form submitted successfully!
</div>
```

Step 4: Define the Form Model in the Component Class

Edit the `contact-form.component.ts` file to define the component class:

```typescript
import { Component } from '@angular/core';

@Component({
  selector: 'app-contact-form',
  templateUrl: './contact-form.component.html',
  styleUrls: ['./contact-form.component.css']
})
export class ContactFormComponent {
  formData = {
    name: '',
    email: '',
    message: ''
  };

  submitted = false;

  onSubmit() {
    // Handle form submission logic here (e.g., send data to a server)

    // For this example, we'll just set the submitted flag
    this.submitted = true;
  }
}
```

Step 5: Add Styles (Optional)

Edit the `contact-form.component.css` file to add some styling to your form.

```css
.form-group {
  margin-bottom: 15px;
```

```
}

.alert-success {
  margin-top: 20px;
}
```

Step 6: Import Required Modules

To use forms in Angular, make sure that you import the necessary modules. In your `app.module.ts` file, import `FormsModule` from `@angular/forms` and add it to the `imports` array.

```typescript
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms'; // Import FormsModule

import { AppComponent } from './app.component';
import { ContactFormComponent } from './contact-form/contact-form.component';

@NgModule({
  declarations: [
    AppComponent,
    ContactFormComponent
  ],
  imports: [
    BrowserModule,
    FormsModule // Add FormsModule here
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Step 7: Run the Application

Start your Angular development server:

```bash
ng serve
```

Open your web browser and navigate to `http://localhost:4200/` to see the "Contact Form" component in action. You can fill out the form and submit it.

This is a basic example of a template-driven form in Angular. You can explore more form controls, validation, and custom form handling as your application requirements grow.

Create an application to demonstrate SPA.

Creating a Single Page Application (SPA) in Angular is a common use case for demonstrating SPAs. Angular provides a powerful framework for building SPAs with client-side routing. Here, I'll guide you through creating a simple SPA with Angular.

Step 1: Create a New Angular Application

If you haven't already, set up your Angular development environment and create a new Angular application as described in the previous responses.

Step 2: Create Components

In Angular, different components represent different views in your SPA. For this example, let's create two components: a home page and a contact page.

Generate a component for the home page:

```bash
ng generate component home
```

Generate a component for the contact page:

```bash
ng generate component contact
```

Step 3: Define Routes

In Angular, you define routes in the `app-routing.module.ts` file. You specify the route paths and which components should be displayed when those paths are navigated to.

Edit the `app-routing.module.ts` file:

```typescript
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

import { HomeComponent } from './home/home.component';
import { ContactComponent } from './contact/contact.component';

const routes: Routes = [
  { path: '', component: HomeComponent }, // Home page
  { path: 'contact', component: ContactComponent }, // Contact page
```

```
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Step 4: Create Navigation Links

Edit the `app.component.html` file to create navigation links. You can use the `routerLink` directive to navigate between different views.

```html
<nav>
  <a routerLink="/">Home</a>
  <a routerLink="/contact">Contact</a>
</nav>

<router-outlet></router-outlet>
```

Step 5: Run the Application

Start your Angular development server:

```bash
ng serve
```

Open your web browser and navigate to `http://localhost:4200/`. You'll see the home page, and you can click the "Contact" link to navigate to the contact page. The content is loaded without a full page refresh, demonstrating the SPA behavior.

This is a basic example of a Single Page Application (SPA) in Angular. You can add more routes and components to build more complex SPAs, as well as add route guards, lazy loading, and other features to enhance your application.