1 Explain the FS module with suitable code snippets.

The `fs` (File System) module in Node.js is used to interact with the file system. It allows you to read, write, update, delete, and perform many other operations on files and directories. Below are some common use cases with code snippets to illustrate how to use the `fs` module in web development.

1. Importing the `fs` Module

To use the `fs` module, you first need to import it:

const fs = require('fs');

2. Reading Files

fs.readFile('example.txt', 'utf8', (err, data) => {

if (err) {

console.error(err);

return;

}

console.log(data); });

---

2 Explain the steps to create a new module in NodeJS

1. **Open a Text Editor**: Start by opening a text editor like Visual Studio Code or Sublime Text.
2. **Create a New File**: Create a new file and save it with a `.js` extension (e.g., `math.js`).
3. **Write Your Code**: In the file, define the function or object that you want to export using `module.exports`. For example:

JavaScript

```
// math.js
function add(a, b) {
    return a + b;
}

module.exports = {
    add: add
};
```

4. **Save the File**: Save your file.

3 Explain the HTTP module with suitable code snippets.

---

4 Discuss the problems related to Dynamic types in java scripts.

While dynamic typing in JavaScript offers flexibility, it can also lead to several problems during development and maintenance of applications. Here's a breakdown of some key issues:

1. Runtime Errors:

- **Type Mismatches:** Since types are not checked until runtime, errors like trying to call a method on a non-object or performing operations on incompatible data types (e.g., adding a string and a number) will only manifest during execution. This can lead to unexpected behavior and crashes.

2. Debugging Challenges:

- **Hidden Errors:** Typos or mistakes in variable assignments might not be caught until runtime, making it harder to identify the root cause of issues during debugging.
- **Less IDE Support:** Static typing allows Integrated Development Environments (IDEs) to provide features like code completion and type checking, which can be less effective with dynamic typing.

3. Maintenance Difficulties:

- **Reduced Code Clarity:** As codebases grow, it can be harder to understand the intended behavior of variables and functions without explicit type information. This can make maintenance and collaboration on large projects more challenging.
- **Refactoring Risks:** Without static types, unintended side effects can be introduced during code refactoring, as changes might break assumptions about the types of data being used.

4. Integration Issues:

- **Interoperability Challenges:** When integrating JavaScript code with libraries or frameworks written in statically typed languages, type mismatches can occur at runtime, causing integration problems.

---

5 Explain the core features of AngularJS

AngularJS, though not actively maintained anymore, was a popular framework for building Single Page Applications (SPAs). Here are some of its core features:

1. Model-View-Controller (MVC) Architecture:

- **Model:** Represents the data of your application.
- **View:** The user interface elements that display the data.
- **Controller:** Handles user interaction and updates the model, which in turn reflects on the view.

AngularJS enforces this separation of concerns, promoting cleaner and more maintainable code.

2. Two-way Data Binding:

- AngularJS automatically synchronizes data between the model and the view.
- Changes made in the view (e.g., user input) are reflected in the model, and vice versa.

3. Directives:

- AngularJS directives are markers on HTML elements that extend their functionality.
- They offer a powerful way to manipulate the DOM and create dynamic UIs.

4. Dependency Injection:

- AngularJS promotes loose coupling between components by using dependency injection.
- This improves testability and makes code more modular.

5. Routing:

- AngularJS allows you to define different views for different parts of your application.
- The URL determines which view is displayed, providing a seamless user experience for navigating within the SPA.

6. Services:

- Services are reusable components that encapsulate application logic and data access.
- They can be shared across different parts of your application, promoting code reuse and separation of concerns.

7. Templates:

- AngularJS uses HTML templates to define the view of your application.
- These templates can be embedded within the HTML or stored in separate files.

8. Filters:

- Filters are functions that can be applied to data before it's displayed in the view.

- They can be used for formatting, sorting, or transforming data to improve its presentation.

6 Explain any 5 built-in angular Directives.

Here's an explanation of 5 commonly used built-in directives in Angular:

1. **ngIf:**
    - **Purpose:** Conditionally includes or removes an element from the DOM based on a provided expression.
    - **Syntax:** `<element *ngIf="condition">...</element>`
    - **Example:**

    HTML

    ```
    <div *ngIf="isLoggedIn">Welcome, {{ username }}!</div>
    ```

    - This code snippet displays the welcome message with the username only if the `isLoggedIn` expression evaluates to true.
2. **ngFor:**
    - **Purpose:** Iterates over a collection of data and instantiates a template for each item.
    - **Syntax:** `<element *ngFor="let item of items">...</element>`
    - **Example:**

    HTML

    ```
    <ul>
      <li *ngFor="let product of products">{{ product.name }}</li>
    </ul>
    ```

    - This code iterates over the `products` array and displays the name of each product in a list item.
3. **ngSwitch:**
    - **Purpose:** Acts like a switch statement, displaying one element from a set of elements based on a provided expression.
    - **Syntax:**

      HTML

      ```
      <div [ngSwitch]="selectedTab">
        <p *ngSwitchCase="'tab1'">Content for tab 1</p>
        <p *ngSwitchCase="'tab2'">Content for tab 2</p>
        <p *ngSwitchDefault>Default content</p>
      </div>
      ```

    - **Example:**
    - This code displays content specific to the value of `selectedTab`.
4. **ngClass:**

- **Purpose:** Conditionally applies CSS classes to an element based on a provided expression or object.
- **Syntax:**

  HTML

  ```
  <div [ngClass]="{'class1': condition1, 'class2': condition2}">...</div>
  ```

- **Example:**

HTML

```
<button [ngClass]="{'active': isActive}">Click me</button>
```

- This code adds the `active` class to the button if the `isActive` expression is true.

5. **ngStyle:**
   - **Purpose:** Conditionally applies inline styles to an element based on a provided expression or object.
   - **Syntax:**

     HTML

     ```
     <div [ngStyle]="{'font-weight': isActive ? 'bold' : 'normal'}">...</div>
     ```

   - **Example:**
   - This code applies bold font weight if the `isActive` expression is true.

These are just a few of the many built-in directives available in Angular. They provide a powerful way to manipulate the DOM and create dynamic user interfaces.

---

7 Explain any 5 built-in angular filters.

Here's an explanation of 5 commonly used built-in filters in Angular:

1. **currency:**
   - **Purpose:** Formats a number as a currency value with a specific symbol and formatting options.
   - **Syntax:** `{{ expression | currency }}`
   - **Example:**

   HTML

   ```
   <p>The price is: {{ product.price | currency:'USD':'1.2-2' }}</p>
   ```

   - This code snippet formats the `product.price` as a US dollar value with two decimal places and thousands separators.

2. **date:**
    - **Purpose:** Formats a date object according to a specified format string.
    - **Syntax:** `{{ expression | date:'format' }}`
    - **Example:**

    HTML

    ```
    <p>Today's date is: {{ currentDate | date:'dd/MM/yyyy' }}</p>
    ```

    - This code displays the current date in the format "dd/MM/yyyy" (day/month/year).
3. **lowercase:**
    - **Purpose:** Converts a string to lowercase.
    - **Syntax:** `{{ expression | lowercase }}`
    - **Example:**

    HTML

    ```
    <p>The message in lowercase: {{ userInput | lowercase }}</p>
    ```

    - This code converts the user input to lowercase letters.
4. **uppercase:**
    - **Purpose:** Converts a string to uppercase.
    - **Syntax:** `{{ expression | uppercase }}`
    - **Example:**

    HTML

    ```
    <p>The message in uppercase: {{ userInput | uppercase }}</p>
    ```

    - This code converts the user input to uppercase letters.
5. **limitTo:**
    - **Purpose:** Limits the output to a specified number of characters or items from an array.
    - **Syntax:**
        - For strings: `{{ expression | limitTo: limit }}`
        - For arrays: `{{ expression | limitTo: limit: start }}` (start is optional)
    - **Example:**

    HTML

    ```
    <p>First 10 characters: {{ product.name | limitTo: 10 }}</p>
    ```

    - This code displays only the first 10 characters of the product name.

---

8 Describe the architecture of Node.js and its relationship with the V8 engine.

Node.js follows a single-threaded, event-driven architecture built around two core components:

1. **V8 Engine:**
    - It's a high-performance JavaScript engine developed by Google, originally for Chrome.
    - V8 is responsible for parsing, compiling, and executing JavaScript code.
    - It manages memory allocation, garbage collection, and provides the core JavaScript runtime environment.
2. **Libuv:**
    - It's an asynchronous I/O library written in C++.
    - Libuv enables Node.js to perform non-blocking I/O operations efficiently.
    - This means Node.js can handle multiple concurrent requests without blocking the main thread.
    - Libuv provides an event loop that monitors for I/O events (like network requests or file system operations) and executes callbacks when those events occur.

Here's how these components work together:

- **Event Loop:** The core of Node.js is the event loop managed by Libuv.
- **Execution of JavaScript Code:** When JavaScript code is received (e.g., from a web server request), the V8 engine parses and compiles it into machine code.
- **Non-Blocking I/O:** If the code involves I/O operations, Node.js doesn't wait for them to complete. Instead, it delegates them to Libuv and continues executing other code.
- **Event Queue:** Libuv places callbacks associated with the I/O operations in an event queue.
- **Event Handling:** Once the I/O operation finishes, Libuv signals the event loop, which then pulls the appropriate callback from the queue and schedules it for execution on the main thread when it's free.

Benefits of this Architecture:

- **Scalability:** Node.js can handle a large number of concurrent connections efficiently due to the non-blocking I/O model.
- **Performance:** V8 is a highly optimized engine, and Libuv's asynchronous I/O avoids blocking the main thread, leading to faster response times.
- **Event-Driven:** The event loop provides a clear model for handling asynchronous operations and simplifies development with callbacks or Promises.

Relationship with V8 Engine:

- V8 acts as the heart of Node.js, responsible for the core JavaScript execution capabilities.
- Node.js extends V8 with additional functionalities specific to server-side development, like modules for interacting with the file system, network, and other system resources.
- While Node.js leverages V8 for JavaScript execution, it's not limited to just JavaScript. You can use transpiled languages like TypeScript or languages that compile to JavaScript (like CoffeeScript) within Node.js.

9 Explain the significance of static typing in TypeScript.

1. Early Error Detection:

- TypeScript enforces type annotations on variables and function arguments/returns.
- This allows the compiler to identify potential type mismatches during the compilation phase, before the code is even run.
- Catching errors early in the development process saves time and effort compared to debugging runtime errors that might occur in dynamically typed languages like JavaScript.

2. Improved Code Clarity:

- Explicit type annotations make the purpose and usage of variables and functions more self-documenting.
- Developers can easily understand the expected data types involved in different parts of the code without relying solely on comments or conventions.
- This can improve code readability and maintainability, especially for larger projects or when working in teams.

3. Better IDE Support:

- Static typing enables Integrated Development Environments (IDEs) to provide more advanced features like:
    - **Type checking:** IDEs can highlight potential type errors as you write code, offering immediate feedback.
    - **Code completion:** Based on types, IDEs can suggest relevant variable names, function calls, and property completions, improving development efficiency.
    - **Refactoring:** Static type information allows IDEs to perform safe refactoring operations like renaming variables or functions while ensuring type consistency throughout the codebase.

4. Increased Developer Confidence:

- With static typing, developers can have more confidence in the correctness of their code.
- The compiler's type checks act as a safety net, reducing the chance of runtime errors caused by type mismatches.
- This can lead to a more productive development experience and fewer headaches during debugging.

5. Integration Benefits:

- When working with libraries or frameworks written in statically typed languages, TypeScript can help prevent type-related issues at integration points.
- Explicit types make it clearer how data should be exchanged between different parts of the application, reducing integration challenges.

10. Summarize the steps to install TypeScript using npm.

Here's a summary of the steps to install TypeScript using npm:

1. Pre-requisite:

- Ensure you have Node.js and npm installed on your system. You can check this by running `node -v` and `npm -v` in your terminal.

2. Installation:

There are two main approaches to install TypeScript using npm:

- **Global Installation (Optional):**
  - This installs TypeScript globally and allows you to use the `tsc` compiler command from anywhere in your terminal.
  - Run the following command:

  Bash

  ```
  sudo npm install -g typescript
  ```

  - **Note:** This approach might require administrative privileges (sudo) depending on your system configuration.
- **Local Installation (Recommended):**
  - This installs TypeScript only for the current project directory and its dependencies.
  - Navigate to your project directory in the terminal.
  - Run the following command:

  Bash

  ```
  npm install typescript --save-dev
  ```

  - The `--save-dev` flag adds TypeScript as a development dependency to your project's `package.json` file.

3. Verification:

- Once installed, you can verify the installation by running:

  Bash

  ```
  tsc -v
  ```

  - This command should print the installed version of the TypeScript compiler

---

11 Explain how TypeScript improves upon vanilla JavaScript for large applications.

TypeScript offers several advantages over vanilla JavaScript for developing large applications. Here's a breakdown of how it improves the development process:

1. Enhanced Type Safety:

- **Static Typing:** TypeScript enforces type annotations on variables, functions, and properties. This allows the compiler to catch potential type mismatches during compilation, preventing runtime errors caused by incorrect data usage. This is crucial for large applications where complex data flows can be error-prone.

2. Improved Code Maintainability:

- **Increased Readability:** Explicit type annotations make code more self-documenting. Developers can easily understand the expected data types involved without relying solely on comments or conventions. This improves code clarity and reduces the cognitive load, especially for large codebases or when working in teams.

3. Better IDE Support:

- **Advanced Features:** Static typing enables IDEs to provide functionalities like:
    - **Type checking:** IDEs can highlight potential type errors as you write, offering immediate feedback.
    - **Code completion:** Based on types, IDEs suggest relevant names, function calls, and property completions, boosting development efficiency.
    - **Refactoring:** Static type information allows safe refactoring operations like renaming variables or functions while ensuring type consistency throughout the codebase.

4. Reduced Debugging Time:

- **Early Error Detection:** By catching type errors during compilation, TypeScript saves time and effort compared to debugging runtime errors that can be harder to identify in large applications.

5. Improved Team Collaboration:

- **Consistent Coding Style:** Type annotations enforce a more consistent coding style, as developers must explicitly define data types. This improves code clarity and reduces the risk of misunderstandings within a team.

6. Integration Benefits:

- **Clearer Data Exchange:** When working with libraries or frameworks written in statically typed languages, TypeScript ensures type compatibility at integration points. Explicit types make it clearer how data should be exchanged between different parts of the application, reducing integration challenges.

7. Modularization and Reusability:

- **Type Annotations in Interfaces:** TypeScript allows defining interfaces for components, specifying their expected types. This promotes better code organization and reusability by ensuring type consistency across modules.

8. Scalability:

- **Gradual Adoption:** TypeScript can be introduced gradually into existing JavaScript codebases. This allows developers to migrate code to a type-safe approach at their own pace.

Trade-offs to Consider:

- **Initial Setup:** Adding type annotations can introduce some overhead, especially for simple projects.
- **Learning Curve:** Developers unfamiliar with static typing might have a learning curve when adopting TypeScript.

---

12 Describe the purpose of the enum type in TypeScript.

In TypeScript, enums (enumerations) are a special kind of data type that allows you to define a set of named constants. They offer several advantages for managing constant values within your code:

1. Improved Readability:

- Enums provide meaningful names for constant values, making your code more readable and self-documenting.
- Instead of using magic numbers (e.g., `const STATUS_PENDING = 0`), you can define an enum like:

TypeScript

```
enum Status {
  Pending = 0,
  Success = 1,
  Failed = 2,
}
```

- This makes the code easier to understand and the purpose of each constant value is clear.

2. Type Safety:

- Once defined, the enum type restricts the values that can be assigned to variables associated with it.
- This helps prevent accidental assignment of incorrect values and improves code reliability.

3. Improved Developer Experience:

- IDEs and code editors can provide better code completion and suggestions when using enums, as they recognize the valid constant names within the enum type.

4. Increased Maintainability:

- If you need to change a constant value, you only need to modify it in one place within the enum definition. This makes code updates easier and reduces the risk of introducing errors by accidentally changing the value in multiple places.

Types of Enums:

- **Numeric Enums:** By default, enums assign numeric values to their members sequentially starting from 0. You can also explicitly assign values:

TypeScript
```
enum UserRole {
  Admin = 10,
  Editor = 20,
  Guest = 30,
}
```

- **String Enums:** Enums can also hold string values:

TypeScript
```
enum FileType {
  PDF = "pdf",
  JPG = "jpg",
  PNG = "png",
}
```

Using Enums:

You can access enum members using dot notation:

TypeScript
```
const user: UserRole = UserRole.Editor;

if (user === UserRole.Admin) {
  console.log("User has admin privileges");
}
```

---

13 Summarize the process of compiling and running a TypeScript file.

Here's a summary of the process for compiling and running a TypeScript file:

1. Tools:

- You'll need two main tools:

- **TypeScript compiler (tsc):** This tool translates your TypeScript code into JavaScript.
- **Node.js (optional):** If you want to run the compiled JavaScript code directly, you'll need Node.js installed on your system.

2. Compilation:

There are two main approaches to compile a TypeScript file:

- **Command Line:**
  - Open your terminal and navigate to the directory containing your TypeScript file (`.ts`).
  - Run the following command, replacing `source.ts` with your actual filename:

    Bash

    ```
    tsc source.ts
    ```

  - This will create a JavaScript file with the same name by default (e.g., `source.js`).
- **Build Tools (Optional):**
  - For larger projects, you might use build tools like Gulp or Webpack to automate the compilation process and manage dependencies.

3. Running the Compiled JavaScript:

- Once you have the compiled JavaScript file:
  - **Node.js:** If your code interacts with the Node.js environment (e.g., file system access, server-side logic), you can run the JavaScript file using Node.js:

    Bash

    ```
    node source.js
    ```

  - **Browser (for some cases):** If your TypeScript code compiles to vanilla JavaScript without external dependencies, you might be able to run it directly in a web browser. However, this is not always the case, and some browser features might not be supported.

---

14 Describe how the ng-controller directive works in AngularJS.

The `ng-controller` directive in AngularJS is a fundamental building block for creating controllers in your application. Here's how it works:

1. Attaching a Controller:

- The `ng-controller` directive is placed as an attribute on an HTML element, typically the `<body>` tag or a specific section of your view.

- It takes an expression as its value, which specifies the name of the controller function you want to associate with that element and its child elements.

Example:

HTML
```
<body ng-controller="MyController">
  </body>
```

- In this example, the `MyController` function will be attached to the `<body>` element and its children.

2. Controller Function:

- The controller function is a JavaScript function defined in your AngularJS application.
- It typically takes a single argument, which is an AngularJS scope object.
- The scope object acts as a communication bridge between your view (HTML) and the controller's logic.

Example:

JavaScript
```
angular.module('myApp', [])
  .controller('MyController', function($scope) {
    $scope.message = "Hello from AngularJS!";
  });
```

- In this example, the `MyController` function injects the `$scope` service and assigns a value to the `message` property on the scope.

3. Accessing Data in the View:

- Within the view (HTML), you can access the properties and methods defined on the scope object using double curly braces (`{{ }}`) for one-way data binding or property binding syntax (`[ ]`) for two-way data binding.

Example:

HTML
```
<p>{{ message }}</p>
```

- This code snippet displays the value of the `message` property defined on the scope within the `MyController` function.

4. Controller Responsibilities:

- Controllers typically handle application logic like:
  - Data manipulation
  - Event handling (e.g., user interactions)
  - Interaction with services (data fetching, business logic)

- They don't directly manipulate the DOM; instead, they update the scope object, and AngularJS automatically reflects those changes in the view through data binding.

---

15 Explain the purpose of the ng-repeat directive in AngularJS.

The `ng-repeat` directive in AngularJS is a powerful tool for iterating over collections of data and dynamically generating HTML content based on each item in the collection. Here's a breakdown of its purpose:

1. Looping through Data:

- `ng-repeat` allows you to specify a collection of data as an expression and iterate over each item within that collection.
- This is particularly useful for displaying lists of items, tables of data, or any scenario where you need to repeat a template structure for each element in a collection.

2. Template Instantiation:

- You define a template structure within the `ng-repeat` directive.
- For each item in the collection, AngularJS creates a separate instance of this template and populates it with the corresponding data from the item.

3. Syntax:

The basic syntax for `ng-repeat` is:

HTML
```
<element ng-repeat="item in collection">
  </element>
```

- `element`: This can be any HTML element where you want to repeat the template.
- `item`: This represents the variable name you'll use to access data from each item in the collection within your template.
- `collection`: This is an expression that evaluates to the collection of data you want to iterate over (e.g., an array, object property).

4. Accessing Data within the Template:

- Inside the template defined within `ng-repeat`, you can access properties of the current item using the variable name you specified (e.g., `item.name`, `item.price`).

Example:

HTML
```
<ul>
  <li ng-repeat="product in products">
    {{ product.name }} - {{ product.price }}
  </li>
</ul>
```

- In this example:
    - The `ng-repeat` iterates over the `products` array.
    - Inside the `<li>` element, the template uses `product.name` and `product.price` to display product information for each item in the `products` array.

5. Additional Features:

- **Track by:** You can use the `track by` property with `ng-repeat` to improve performance for large lists by specifying how AngularJS should track changes within the collection.
- **Filters:** You can combine `ng-repeat` with AngularJS filters to manipulate data before displaying it in the view.

---

16 Describe the benefits of using TypeScript's static types.

TypeScript's static type system offers a multitude of benefits for developers working on large and complex applications. Here's a closer look at some of the key advantages:

1. Early Error Detection:

- TypeScript enforces type annotations on variables, functions, and properties.
- This allows the compiler to identify potential type mismatches during the compilation phase, before the code is even run.
- Catching errors early saves time and effort compared to debugging runtime errors that can be harder to identify in large codebases.

2. Improved Code Readability:

- Explicit type annotations make the purpose and usage of variables and functions more self-documenting.
- Developers can easily understand the expected data types involved in different parts of the code without relying solely on comments or conventions.
- This enhances code clarity and maintainability, especially for larger projects or when working in teams.

3. Better IDE Support:

- Static typing enables Integrated Development Environments (IDEs) to provide more advanced features like:
    - **Type checking:** IDEs can highlight potential type errors as you write code, offering immediate feedback.
    - **Code completion:** Based on types, IDEs can suggest relevant variable names, function calls, and property completions, improving development efficiency.
    - **Refactoring:** Static type information allows IDEs to perform safe refactoring operations like renaming variables or functions while ensuring type consistency throughout the codebase.

4. Increased Developer Confidence:

- With static typing, developers can have more confidence in the correctness of their code.
- The compiler's type checks act as a safety net, reducing the chance of runtime errors caused by type mismatches.
- This can lead to a more productive development experience and fewer headaches during debugging.

5. Integration Benefits:

- When working with libraries or frameworks written in statically typed languages, TypeScript can help prevent type-related issues at integration points.
- Explicit types make it clearer how data should be exchanged between different parts of the application, reducing integration challenges.

6. Improved Code Organization:

- TypeScript allows defining interfaces for components, specifying their expected types. This promotes better code organization and reusability by ensuring type consistency across modules.

7. Maintainability:

- By catching errors early and promoting clear code structure, static types contribute to more maintainable codebases.
- As projects grow, the risk of regressions due to type mismatches is reduced.

8. Scalability:

- The type system helps manage the complexity of large applications by providing clear contracts between different parts of the code.
- This can improve the scalability of your project as it grows in size and complexity.

---

17 Explain the difference between number[] and Array<number> in TypeScript.

In TypeScript, both `number[]` and `Array<number>` represent an array of numbers. There's no semantic difference between these two notations. They are essentially interchangeable.

Here's a breakdown:

- **`number[]`:**
  - This is a shortcut syntax provided by TypeScript.
  - It leverages the fact that JavaScript arrays can hold any type of data.
  - By using `number[]`, you explicitly tell the TypeScript compiler that this array should only contain numbers.

- **`Array<number>`:**
  - This is the generic way to define an array type in TypeScript.
  - `Array` is a built-in generic type that takes a type parameter specifying the element type of the array.
  - In this case, `<number>` specifies that the array elements should be of type `number`.

Here's an example of how both notations work:

TypeScript
```
let numbers1: number[] = [1, 2, 3];
let numbers2: Array<number> = [4, 5, 6];

console.log(numbers1); // Output: [1, 2, 3]
console.log(numbers2); // Output: [4, 5, 6]
```

---

18 Explain the purpose of the ng-class directive in AngularJS.

The `ng-class` directive in AngularJS is a powerful tool for dynamically applying CSS classes to HTML elements based on various conditions or expressions. It offers flexibility in controlling the element's appearance based on data or user interactions.

Here's a breakdown of its purpose:

1. Dynamic Class Binding:

- Unlike static class attributes, `ng-class` allows you to bind the element's class list to an expression that evaluates to a set of CSS classes.
- This expression can be a string, an object, or an array, providing different ways to control which classes are applied.

2. String Syntax:

- In its simplest form, `ng-class` can take a string expression containing space-separated CSS class names.
- The element's class list will be set to the classes specified in the string.

Example:

HTML
```
<div ng-class="isActive ? 'active' : ''">This element is active</div>
```

- In this example:
  - The `isActive` variable determines the class applied.
  - If `isActive` is true, the `active` class is added to the `<div>`. Otherwise, the class list is empty.

3. Object Syntax:

- You can use an object literal with `ng-class` to conditionally apply classes based on boolean properties.
- The property keys represent CSS class names, and the property values represent boolean expressions.
- A class is added to the element's class list only if the corresponding property evaluates to true.

Example:

HTML
```
<button ng-class="{ 'btn-primary': isSelected, 'btn-secondary': !isSelected }}">Select</button>
```

- This button applies the `btn-primary` class if `isSelected` is true and `btn-secondary` if it's false.

4. Array Syntax:

- You can use an array expression with `ng-class` to dynamically add or remove classes from the element.
- The array elements should be strings representing CSS class names.

Example:

HTML
```
<span ng-class="['error', showError ? 'visible' : 'hidden']">Error message</span>
```

- This code snippet displays an error message with the `error` class always applied.
- If `showError` is true, the `visible` class is added, making the message visible. Otherwise, the `hidden` class is added, hiding the message.

5. Combining Syntaxes:

- You can even combine these syntaxes within a single `ng-class` expression for more complex scenarios.

---

19 Summarize the key features of Node.js.

Here's a summary of the key features of Node.js:

- **Event-Driven Architecture:** Node.js uses an event-driven, non-blocking I/O model. This means it can handle multiple concurrent requests without blocking the main thread. This is achieved through:
    - **Libuv:** An asynchronous I/O library written in C++ that handles I/O operations efficiently.
    - **Event Loop:** The core of Node.js, it manages a queue of events and executes callbacks when I/O operations are complete.

- **JavaScript on the Server:** Node.js allows you to write server-side code using JavaScript, a familiar language for many web developers. This can simplify development and reduce the need for multiple languages (backend and frontend).
- **Scalability:** Due to its non-blocking I/O model, Node.js can handle a large number of concurrent connections efficiently. This makes it suitable for building scalable applications.
- **Single-Threaded (but Highly Scalable):** While Node.js uses a single-threaded event loop, it doesn't limit scalability. The non-blocking I/O model avoids blocking the main thread, allowing it to handle many connections efficiently.
- **Cross-Platform Compatibility:** Node.js applications can run on different operating systems (Windows, macOS, Linux) without modification, thanks to JavaScript's inherent portability.
- **Rich Ecosystem of Packages:** Node.js has a vast and active open-source community that contributes a wide variety of packages (modules) to the Node Package Manager (npm). This ecosystem provides pre-built solutions for many common tasks, saving development time.
- **Fast Data Streaming:** Node.js is efficient at handling real-time data streaming due to its event-driven architecture and the V8 JavaScript engine's optimization for fast code execution.
- **Full-Stack JavaScript:** Node.js, along with JavaScript frameworks like Express.js, enables building both the backend (server-side) and frontend (client-side) of an application using JavaScript. This can streamline development for some projects.

These features make Node.js a popular choice for building various applications, including:

- Web servers and APIs
- Real-time applications (chat, collaboration tools)
- I/O bound applications (data streaming)
- Microservices architectures

10 Marks

1. Explain how the event loop in Node.js works and its implications for writing code.

The Event Loop in Node.js

The event loop is a fundamental concept in Node.js that underpins its ability to handle multiple concurrent requests efficiently. Here's how it works and how it affects your code:

1. Non-Blocking I/O:

- Unlike traditional web servers, Node.js utilizes a non-blocking I/O model. This means it doesn't wait for I/O operations (like reading from a file or making a network request) to complete before processing other tasks.

2. Libuv and the Event Loop:

- Node.js leverages a library called Libuv written in C++. Libuv efficiently manages asynchronous I/O operations, initiating them and notifying Node.js when they're finished.
- The event loop is a core component of Node.js. It's an infinite loop that performs three main tasks:
    - **1. Checking for Timers and Pending Callbacks:**
        - The loop checks for timers that have expired and callbacks associated with them. These callbacks are then added to a queue for execution.
    - **2. Processing Pending I/O Operations:**
        - Libuv informs the event loop when I/O operations are complete. The loop then retrieves any associated callback functions from a queue and adds them to the execution queue.
    - **3. Executing Callbacks:**
        - The event loop processes the callbacks in the execution queue one by one. This is where your application's code gets executed.

3. Implications for Writing Code:

- **Callback-Based Programming:**
    - Due to the event loop's reliance on callbacks, Node.js code often involves writing functions that are passed as arguments to be executed later when an event occurs (e.g., an I/O operation completes).
- **Asynchronous vs. Synchronous:**
    - Node.js excels at handling asynchronous operations efficiently. However, it's important to be mindful of synchronous code within callbacks, as it can block the event loop and hinder performance.
- **Error Handling:**
    - When working with asynchronous operations, proper error handling is crucial. You need to manage potential errors within callback functions to prevent unexpected behavior.

4. Benefits:

- **Scalability:** The event loop allows Node.js to handle a large number of concurrent connections without blocking, making it suitable for scalable applications.
- **Efficiency:** By not waiting for I/O, Node.js can handle multiple requests efficiently, making it ideal for real-time applications and I/O bound tasks.

5. Considerations:

- **Callback Hell:** Nesting callbacks can make code difficult to read and maintain. Techniques like promises and async/await can help manage asynchronous code flow more effectively.
- **Mental Model Shift:** Coming from a synchronous programming background, understanding the asynchronous nature of Node.js can require a shift in how you think about code execution.

---

2. Explain built-in modules of Node JS with suitable example.

Node.js comes with a rich set of built-in modules that provide essential functionalities for various tasks. Here's an explanation of some common built-in modules with examples:

****1. http:****

This module allows you to create web servers and handle HTTP requests.

****Example:****

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

const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.write('Hello World!');
  res.end();
});

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

This code creates a simple HTTP server that listens on port 3000 and responds with "Hello World!" when a request is received.

****2. fs (File System):****

This module offers functionalities for interacting with the file system, including reading, writing, creating, and deleting files.

****Example:****

```javascript
const fs = require('fs');
fs.readFile('data.txt', 'utf8', (err, data) => {
  if (err) {
    console.error(err);
  } else {
    console.log(data);
  }
});
```

This code reads the contents of a file named "data.txt" and logs them to the console.

3. path:

This module provides utilities for working with file and directory paths.

Example:

```javascript
const path = require('path');
const filePath = path.join(__dirname, 'data', 'user.json');
console.log(filePath);
```

This code constructs a file path by joining the current directory (`__dirname`), a subdirectory named "data", and a filename "user.json".

4. events:

This module allows you to create custom event emitters and listeners for building event-driven applications.

Example:

```javascript

const EventEmitter = require('events');

const myEmitter = new EventEmitter();

myEmitter.on('message', (data) => {

  console.log(`Received message: ${data}`);

});

myEmitter.emit('message', 'Hello from the event emitter!');

```

This code creates an event emitter and defines a listener for the "message" event. The emitter then triggers the event, sending the message to the listener.

****5. os:****

This module provides information about the operating system your Node.js application is running on.

****Example:****

```javascript

const os = require('os');

const hostname = os.hostname();

const platform = os.platform();

console.log(`Hostname: ${hostname}`);

console.log(`Platform: ${platform}`);

```

This code retrieves the hostname and platform (operating system) of the machine where the Node.js application is running.

These are just a few examples of built-in Node.js modules. The Node.js documentation provides a comprehensive list and detailed explanations of all available modules: [URLnodejs org doc api]

By understanding and utilizing these modules effectively, you can streamline development and create robust Node.js applications that handle various tasks.

---

3. **You are creating a feedback form for a website. The form includes a text input for the user's name, a textarea for comments, and a checkbox to agree to terms.**
    - **Display a summary of the feedback dynamically as the user fills out the form.**
        - **Requirements:**
            - **Define the Angular application.**
    - **Bind the text input to capture the user's name.**
    - **Bind the textarea to capture the user's comments.**
    - **Bind the checkbox to capture the user's agreement to terms.**
    - **Dynamically display a summary of the feedback, including the name, comments, and agreement status.**
    - **Ensure the summary updates in real-time as the user interacts with the form.**

To create a feedback form using AngularJS that dynamically updates a summary as the user fills it out, we will:

1. Define the AngularJS application.

2. Bind the text input to capture the user's name.

3. Bind the textarea to capture the user's comments.

4. Bind the checkbox to capture the user's agreement to terms.

5. Dynamically display a summary of the feedback, including the name, comments, and agreement status.

6. Ensure the summary updates in real-time as the user interacts with the form.

Complete Implementation

HTML

```html
<!DOCTYPE html>
<html ng-app="feedbackApp">
<head>
  <title>Feedback Form</title>
  <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></script>
  <style>
    .container {
      max-width: 600px;
      margin: 0 auto;
      padding: 20px;
      border: 1px solid #ccc;
      border-radius: 5px;
    }
    .summary {
      margin-top: 20px;
      padding: 10px;
      border: 1px solid #ddd;
      border-radius: 5px;
    }
  </style>
</head>
<body ng-controller="FeedbackController">

  <div class="container">
```

```
    <h2>Feedback Form</h2>

    <form>

      <label for="name">Name:</label><br>

      <input type="text" id="name" ng-model="feedback.name"><br><br>


      <label for="comments">Comments:</label><br>

      <textarea id="comments" ng-model="feedback.comments"></textarea><br><br>


      <label>

        <input type="checkbox" ng-model="feedback.agree"> I agree to the terms and
conditions

      </label><br><br>

    </form>


    <div class="summary" ng-if="feedback.name || feedback.comments || feedback.agree">

      <h3>Feedback Summary</h3>

      <p><strong>Name:</strong> {{ feedback.name }}</p>

      <p><strong>Comments:</strong> {{ feedback.comments }}</p>

      <p><strong>Agreement:</strong> {{ feedback.agree ? 'Agreed' : 'Not Agreed' }}</p>

    </div>

  </div>


  <script>

    angular.module('feedbackApp', [])

      .controller('FeedbackController', function($scope) {

        $scope.feedback = {
```

```
        name: '',
        comments: '',
        agree: false
      };
    });
  </script>
</body>
</html>
```

Explanation

1. **Define the AngularJS Application**:
   - The `ng-app` directive initializes the AngularJS application named `feedbackApp`.
   - The `ng-controller` directive attaches the `FeedbackController` to the HTML.

2. **Bind the Text Input to Capture the User's Name**:
   - The `ng-model` directive binds the input field to `feedback.name`.

3. **Bind the Textarea to Capture the User's Comments**:
   - The `ng-model` directive binds the textarea to `feedback.comments`.

4. **Bind the Checkbox to Capture the User's Agreement to Terms**:
   - The `ng-model` directive binds the checkbox to `feedback.agree`.

5. **Dynamically Display a Summary of the Feedback**:

- The summary section uses AngularJS expressions to display the values of `feedback.name`, `feedback.comments`, and `feedback.agree`.

- The `ng-if` directive ensures the summary is only displayed if any of the feedback fields have values.

---

4. **You are creating a survey form where users can add their answers dynamically. Display the answers and bind each answer to an input field.**
    - **Requirements:**
        - **Define the Angular application.**
    - **Bind an input field to a new answer.**
    - **Iterate over a list of answers and display them.**
    - **Add a button to add the new answer to the list and display the updated list dynamically.**

To create a dynamic survey form in AngularJS where users can add their answers, we will follow these steps:

1. Define the AngularJS application.

2. Bind an input field to capture a new answer.

3. Iterate over a list of answers and display them.

4. Add a button to add the new answer to the list and display the updated list dynamically.

Here's the complete implementation:

HTML

```html

**<!DOCTYPE html>**

**<html ng-app="surveyApp">**

**<head>**

**<title>Survey Form</title>**
```

```
  <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></script>
  <style>
    .container {
      max-width: 600px;
      margin: 0 auto;
      padding: 20px;
      border: 1px solid #ccc;
      border-radius: 5px;
    }
    .answers {
      margin-top: 20px;
      padding: 10px;
      border: 1px solid #ddd;
      border-radius: 5px;
    }
    .answer-item {
      margin-bottom: 10px;
    }
  </style>
</head>
<body ng-controller="SurveyController">

  <div class="container">
    <h2>Survey Form</h2>
    <form ng-submit="addAnswer()">
```

```
    <label for="newAnswer">New Answer:</label><br>
    <input type="text" id="newAnswer" ng-model="newAnswer">
    <button type="submit">Add Answer</button>
  </form>

  <div class="answers" ng-if="answers.length > 0">
    <h3>Answers</h3>
    <div class="answer-item" ng-repeat="answer in answers track by $index">
      <input type="text" ng-model="answer">
    </div>
  </div>
</div>

<script>
  angular.module('surveyApp', [])
    .controller('SurveyController', function($scope) {
      $scope.answers = [];
      $scope.newAnswer = '';

      $scope.addAnswer = function() {
        if ($scope.newAnswer.trim() !== '') {
          $scope.answers.push($scope.newAnswer);
          $scope.newAnswer = '';
        }
      };
```

});

</script>

</body>

</html>

```

### Explanation

1. **Define the AngularJS Application**:

   - The `ng-app` directive initializes the AngularJS application named `surveyApp`.

   - The `ng-controller` directive attaches the `SurveyController` to the HTML.

2. **Bind an Input Field to Capture a New Answer**:

   - The `ng-model` directive binds the input field to `newAnswer`, a variable in the scope.

3. **Iterate Over a List of Answers and Display Them**:

   - The `ng-repeat` directive iterates over the `answers` array and displays each answer.

   - Each answer is bound to an input field to allow real-time editing.

4. **Add a Button to Add the New Answer to the List**:

   - The form element uses the `ng-submit` directive to call the `addAnswer` function when the form is submitted.

   - The `addAnswer` function checks if the `newAnswer` is not empty, then pushes it to the `answers` array and resets the `newAnswer` field.

---

## 5. List core features in Angular JS and explain any three.
```

AngularJS is a powerful framework for building dynamic web applications. Here are some of the core features of AngularJS:

1. **Two-Way Data Binding**
2. **MVC Architecture**
3. **Directives**
4. **Expressions**
5. **Modules**
6. **Controllers**
7. **Services**
8. **Filters**
9. **Routing**
10. **Dependency Injection**
11. **Templates**
12. **Custom Directives**

Explanation of three features:

1. **Two-way Data Binding**: Two-way data binding in AngularJS ensures that any changes made to the model (via controller or user interaction) reflect instantly in the view, and vice versa. This bidirectional synchronization eliminates the need for manual DOM manipulation and keeps the application state and UI in sync effortlessly.
2. **Directives**: AngularJS directives allow developers to extend HTML with custom attributes and tags, encapsulating complex UI components or behaviors into reusable units. For example, directives can be used to create custom input validations, integrate third-party libraries seamlessly, or define reusable UI components like tabs or modals.
3. **Dependency Injection (DI)**: Dependency Injection in AngularJS facilitates the creation of loosely coupled components by managing their dependencies externally. Components declare their dependencies (like services or other components), and AngularJS injects these dependencies when instantiating the component. This approach promotes modular design, enhances code reusability, and simplifies unit testing by allowing dependencies to be easily mocked or substituted.

---

6. Summarize the process of setting up an AngularJS application with controllers and services.

Setting up an AngularJS application with controllers and services involves several key steps:

1. Load AngularJS Library

Include the AngularJS library in your HTML file. This can be done by linking to a CDN or downloading the library and referencing it locally.

2. Define the AngularJS Module

Create an AngularJS module which will serve as the main container for your application. This module is defined in JavaScript and serves as a namespace for your application components.

3. Create Controllers

Controllers in AngularJS are responsible for managing the data and behavior of your application. They are defined within the module and attached to parts of your HTML using the `ng-controller` directive. Controllers use `$scope` to pass data and functions to the view.

4. Create Services

Services in AngularJS are used to share data and functions across different parts of your application. They are singletons and can be created using different methods such as factory, service, or provider. Services are injected into controllers or other services where they are needed.

5. Integrate Controllers and Services

Use dependency injection to include services in your controllers. This allows controllers to call functions and access data defined in services, promoting code reuse and separation of concerns.

6. Bootstrap the Application

Ensure your HTML is properly set up to use AngularJS by including the `ng-app` directive on an element that encompasses your application. This initializes the AngularJS application.

Summary

1. **Load AngularJS Library**: Add AngularJS to your HTML.

2. **Define the Module**: Create an AngularJS module as the main container.

3. **Create Controllers**: Define controllers to manage data and behavior.

4. **Create Services**: Develop services to share functionality and data.

5. **Integrate Controllers and Services**: Use dependency injection to connect them.

6. **Bootstrap the Application**: Initialize the application with the `ng-app` directive.

Following these steps, you can set up a well-structured AngularJS application that leverages controllers and services for modular, maintainable code.

---

7. Describe the main differences between synchronous and asynchronous code in Node.js.

In Node.js, understanding the differences between synchronous and asynchronous code execution is crucial due to its single-threaded, event-driven nature. Here are the main differences:

Synchronous Code Execution:

1. **Blocking Nature:**

Synchronous code execution operates sequentially. Each operation blocks the execution of subsequent code until it completes. For example, if there's a synchronous operation that takes a long time (like reading a large file or making a slow database query), the entire application's execution will pause until that operation finishes.

2. **Return Values:**

Synchronous functions return control to the caller only after completing their tasks. This makes the flow predictable and easier to reason about since operations are performed in a deterministic order.

3. **Error Handling:**

Errors in synchronous code are straightforward to handle because they typically throw exceptions that can be caught using `try-catch` blocks. This makes error handling more synchronous in nature and easier to manage within the same execution context.

Asynchronous Code Execution:

1. **Non-Blocking Nature:**

Asynchronous code does not wait for tasks to complete and allows other operations to continue while waiting for the completion of the asynchronous task. This is achieved using callbacks, promises, or async/await syntax in Node.js.

2. **Callback Mechanism:**

Asynchronous functions accept callback functions or return promises that allow registering callbacks. These callbacks are invoked when the asynchronous operation completes, allowing the program to continue executing other tasks in the meantime.

3. **Error Handling:**

Errors in asynchronous code are typically handled through callbacks, promise rejections, or try-catch blocks inside async functions (using async/await). Error handling in asynchronous code can sometimes be more complex due to the need to propagate errors properly across callback chains or promise chains.

---

8. Explain various inheritance in Typescript with example.

   In TypeScript, inheritance can be implemented using several mechanisms, including class inheritance, interface inheritance, and mixin patterns. Let's explore each of these with examples:

   ### 1. Class Inheritance:

   Class inheritance in TypeScript allows a class (subclass) to inherit properties and methods from another class (superclass). It supports both single and multiple inheritances through extending classes.

   ```typescript

   // Superclass

   class Animal {

```
    name: string;

    constructor(name: string) {
        this.name = name;
    }

    move(distance: number = 0) {
        console.log(`${this.name} moved ${distance} meters.`);
    }
}

// Subclass inheriting from Animal
class Dog extends Animal {
    bark() {
        console.log(`${this.name} barked!`);
    }
}

// Usage
let dog = new Dog("Buddy");
dog.bark(); // Output: Buddy barked!
dog.move(10); // Output: Buddy moved 10 meters.
```

Explanation:

- In this example, `Animal` is the superclass, and `Dog` is the subclass that extends `Animal`.

- The `Dog` class inherits the `name` property and the `move` method from `Animal`.

- The `bark` method is unique to the `Dog` class.

- Instances of `Dog` can access methods from both `Animal` and `Dog` itself.

2. Interface Inheritance:

Interfaces in TypeScript can also inherit from other interfaces. This allows for the creation of new interfaces that extend or specialize existing ones.

```typescript
// Base interface
interface Shape {
    color: string;
}

// Derived interface inheriting from Shape
interface Square extends Shape {
    sideLength: number;
}

// Usage
let square: Square = {
    color: "blue",
    sideLength: 10,
};
console.log(`Square color: ${square.color}, Side length: ${square.sideLength}`);
```

Explanation:

- Here, `Square` extends `Shape`, inheriting the `color` property from `Shape`.

- `Square` adds its own `sideLength` property.

- Instances of `Square` must have both `color` (inherited) and `sideLength` (specific to `Square`).

3. Mixin Pattern:

Mixins in TypeScript allow classes to extend multiple classes (or interfaces) to combine their features. This is achieved using a function that copies properties and methods from source classes/interfaces into a target class.

```typescript
// Mixin example: Mixin function
function withPrintable(Base: new () => any) {
    return class extends Base {
        print() {
            console.log(`Printing from ${this.name}`);
        }
    };
}

// Base class
class Person {
    name: string;
```

```
    constructor(name: string) {
        this.name = name;
    }
}

// Applying mixin to create a new class
interface Printable {
    print(): void;
}

const PrintablePerson = withPrintable(Person);

// Usage
let person = new PrintablePerson("Alice");
person.print(); // Output: Printing from Alice
```

Explanation:

- `withPrintable` is a function that takes a base class `Base` and returns a new class extending `Base`, adding a `print` method.

- `Person` is a base class with a `name` property.

- `Printable` is an interface defining the `print` method.

- `PrintablePerson` is created by applying the `withPrintable` mixin to `Person`, creating a new class with `name` and `print` functionality.

- Instances of `PrintablePerson` can access both `name` and `print` functionality.

---

9. **You are developing an online store application. Display a list of products and bind the product name to an input field for adding new products.**

Requirements:

- **Define the Angular application.**
- **Iterate over an array of products and display their names.**
- **Bind an input field to a new product name.**
- **Add a button to add the new product to the list.**
- **Ensure the list updates dynamically when new products are added.**

To meet the requirements for developing an online store application using Angular, we'll create a simple implementation that displays a list of products, allows adding new products via an input field, and updates the list dynamically. Below are the steps and code snippets to achieve this

Step-by-Step Implementation:

1. Define the Angular Application:

First, set up a new Angular application if you haven't already. You can use the Angular CLI for this purpose:

```bash

**ng new online-store-app**

**cd online-store-app**

```

2. Create Product Service and Component:

Create a service to manage products and a component to display and add new products.

Product Service (`product.service.ts`):

```typescript

// product.service.ts

**import { Injectable } from '@angular/core';**

**@Injectable({**

  **providedIn: 'root'**
```

```
})
export class ProductService {

  private products: string[] = ['Product 1', 'Product 2', 'Product 3'];

  constructor() { }

  getProducts(): string[] {
    return this.products;
  }

  addProduct(newProduct: string): void {
    this.products.push(newProduct);
  }
}
```

Product Component (`product.component.ts` and `product.component.html`):

```typescript
// product.component.ts

import { Component } from '@angular/core';
import { ProductService } from './product.service';
```

```
@Component({
  selector: 'app-product',
  templateUrl: './product.component.html',
  styleUrls: ['./product.component.css']
})
export class ProductComponent {

  products: string[];
  newProduct: string = '';

  constructor(private productService: ProductService) {
    this.products = this.productService.getProducts();
  }

  addProduct(): void {
    if (this.newProduct.trim()) {
      this.productService.addProduct(this.newProduct);
      this.newProduct = ''; // Clear input field after adding
    }
  }
}
```

```html
<!-- product.component.html -->
```

```
<div>
  <h2>Product List</h2>
  <ul>
    <li *ngFor="let product of products">{{ product }}</li>
  </ul>
</div>

<div>
  <h2>Add New Product</h2>
  <input type="text" [(ngModel)]="newProduct" placeholder="Enter product name">
  <button (click)="addProduct()">Add Product</button>
</div>
```

3. Update `app.module.ts`:

Ensure that the `ProductComponent` and `ProductService` are declared and provided respectively in your `app.module.ts`.

```typescript
// app.module.ts

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

```
import { AppComponent } from './app.component';
import { ProductComponent } from './product.component';
import { ProductService } from './product.service';

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

4. Run the Application:

Run `ng serve` from the command line in your project directory and navigate to `http://localhost:4200` in your browser to see the online store application in action.

Explanation:

- **Product Service**: Manages the list of products (`products` array) and provides methods (`getProducts`, `addProduct`) to retrieve and modify the list.

- **Product Component**: Displays the list of products using `ngFor`, binds the `newProduct` input field using `ngModel`, and implements `addProduct` method to add new **products to the list.**

- **FormsModule**:** Imported into `AppModule` to enable `ngModel` for two-way data binding in the input field.

---

10. Write a TypeScript interface User with properties username (string) and password (string). Then, create a function login that takes a User object and returns a message "Login successful" if the username is "admin" and the password is "password123", otherwise it returns "Login failed".

Here's how you can define the TypeScript interface `User` and implement the `login` function based on the given requirements:

Define TypeScript Interface:

// user.interface.ts

```
interface User {
  username: string;
  password: string;
}
```

Create Login Function:

```
// auth.service.ts
function login(user: User): string {
  if (user.username === "admin" && user.password === "password123") {
    return "Login successful";
```

```
    } else {
        return "Login failed";
    }
}
```

Usage Example:

```
// main.ts (or any other entry point)

import { login } from './auth.service';

// Example 1: Successful login
let user1: User = { username: "admin", password: "password123" };
console.log(login(user1)); // Output: Login successful

// Example 2: Failed login due to wrong password
let user2: User = { username: "admin", password: "wrongpassword" };
console.log(login(user2)); // Output: Login failed

// Example 3: Failed login due to wrong username
let user3: User = { username: "user", password: "password123" };
console.log(login(user3)); // Output: Login failed
```

Explanation:

1. **Interface `User`:**

- Defines a TypeScript interface `User` with two properties: `username` (string) and `password` (string). This structure ensures that any object conforming to this interface must have these two properties.

2. **Login Function:**

- `login` function takes a `User` object as a parameter.

- Checks if the `username` is "admin" and the `password` is "password123".

- Returns "Login successful" if both conditions are met; otherwise, returns "Login failed".

3. **Usage Example:**

- Demonstrates how to use the `login` function with different `User` objects.

- Outputs "Login successful" for the correct credentials and "Login failed" for incorrect credentials.

This setup ensures type safety and clarity in the code, leveraging TypeScript's strong typing capabilities with interfaces while implementing straightforward login logic based on provided username and password criteria.