**Question 1: What is the use of Angular?**

**Answer:** Angular is a TypeScript-based open-source front-end web application framework. It is mainly used for building dynamic, single-page applications (SPAs) with rich user interfaces. Angular allows developers to create reusable components, manage the application's data binding and rendering efficiently, and ensure a clean, modular codebase for large-scale applications. It supports two-way data binding, dependency injection, modular development, routing, and testing.

**Question 2: What are directives in Angular?**

**Answer:** Directives in Angular are special markers in the DOM that allow Angular to manipulate elements, attributes, and components dynamically. They help in adding behavior to the DOM elements. Angular identifies them by the prefix ng-. Directives enhance HTML by adding new syntax and making it more dynamic.

**Question 3: Explain the different types of Angular directives?**

**Answer:** There are three main types of directives in Angular:

1. **Component Directives**: These are custom components that contain a view (HTML) and associated logic (TypeScript). Every component is a directive with a template.
   * Example: @Component()
2. **Structural Directives**: These alter the structure of the DOM by adding or removing elements. They are prefixed with \*.
   * Example: \*ngIf, \*ngFor
3. **Attribute Directives**: These change the appearance or behavior of an existing element.
   * Example: ngClass, ngStyle

**Question 4: Explain the importance of the NPM and node\_modules folder?**

**Answer:**

* **NPM (Node Package Manager)**: NPM is the default package manager for Node.js and is used in Angular for managing dependencies, libraries, and packages. It provides access to a vast repository of open-source packages for Angular development.
* **node\_modules folder**: This folder contains all the libraries and dependencies listed in the package.json file. It includes the packages required to run, build, and maintain the Angular project.

**Question 5: Explain the importance of the package.json file in Angular?**

**Answer:** The package.json file is essential in Angular as it:

1. Lists all the dependencies (libraries and modules) required for the project.
2. Contains metadata about the project such as name, version, description, etc.
3. Defines the scripts for building, testing, and running the application.
4. Maintains versioning of the installed packages. This file ensures consistent package management across different environments.

**Question 6: What is TypeScript and why do we need it?**

**Answer:** **TypeScript** is a superset of JavaScript that adds static typing, classes, interfaces, and other features to make JavaScript more structured and easier to maintain. It is the primary language used in Angular because it:

1. Enables type safety, which helps catch errors early during development.
2. Supports OOP (Object-Oriented Programming) principles like inheritance and interfaces.
3. Provides better tooling for IDEs, enabling features like autocompletion and refactoring.

**Question 7: Explain the importance of Angular CLI?**

**Answer:** Angular CLI (Command Line Interface) is a powerful tool for Angular developers that simplifies the development workflow. Its importance includes:

1. **Project Scaffolding**: Quickly generates the basic structure of an Angular project.
2. **Automation**: Automates repetitive tasks like generating components, services, modules, etc.
3. **Build Process**: Provides commands to build, serve, and deploy the application.
4. **Testing**: Simplifies running unit and end-to-end tests.
5. **Optimization**: Helps with production-ready builds by enabling ahead-of-time (AOT) compilation, code minification, and tree shaking.

**Question 8: Explain the importance of Components and Modules?**

**Answer:**

* **Components**: Components are the building blocks of an Angular application. Each component consists of HTML, CSS, and TypeScript files to handle the UI and logic for a specific part of the application. Components are reusable, which promotes a modular design.
* **Modules**: Angular modules (@NgModule) are used to group related components, services, and other modules together. They help in organizing the application into functional units, making it scalable and maintainable. The AppModule is the root module that bootstraps the entire Angular application.

**Question 9: What is a decorator in Angular?**

**Answer:** A **decorator** in Angular is a special type of declaration attached to a class, method, or property to modify its behavior. Decorators provide metadata about a class or function, which Angular uses to configure the application. Examples include:

* @Component (to define a component)
* @NgModule (to define a module)
* @Injectable (to define a service)

**Question 10: What are annotations or metadata?**

**Answer:** Annotations or **metadata** in Angular are additional information attached to a class or component to describe how it should be processed, instantiated, or used at runtime. They are provided using decorators like @Component and @NgModule. For example, the metadata of a component specifies the selector, template URL, styles, and other configuration settings necessary for Angular to render and manage the component.

**Question 11: What is a template?**

**Answer:** In Angular, a **template** is an HTML view that defines the user interface (UI) for a component. It contains HTML tags, Angular directives, and bindings that allow you to display data and interact with the logic defined in the component class. Templates can either be written inline, within the component file, or be linked to an external HTML file using the templateUrl property in the component decorator.

**Question 12: Explain the four types of Data Bindings in Angular?**

**Answer:** Angular provides four types of data bindings:

1. **Interpolation (One-way Data Binding)**: This is the process of embedding expressions into HTML to display dynamic values from the component's TypeScript class.
   * Example: {{ expression }}
2. **Property Binding**: This binds a property of an HTML element to a property in the component class. The value flows from the component to the view.
   * Example: [src]="imagePath"
3. **Event Binding**: This binds a DOM event (like a click or input event) to a method in the component's TypeScript class, allowing interaction between the user and the component.
   * Example: (click)="onClick()"
4. **Two-way Data Binding**: This allows bidirectional communication between the component and the view. It is achieved using the [(ngModel)] directive and allows changes in the view to update the component and vice versa.
   * Example: [(ngModel)]="userName"

**Question 13: Explain the architecture of Angular?**

**Answer:** Angular's architecture follows a modular design. Key components include:

1. **Modules (@NgModule)**: The building blocks that organize the app into cohesive blocks of functionality (e.g., AppModule).
2. **Components**: UI elements that handle data, logic, and HTML rendering.
3. **Templates**: Define the view (HTML structure) associated with a component.
4. **Services**: Handle business logic and are injected into components using dependency injection.
5. **Directives**: Modify the DOM or component behavior (e.g., ngIf, ngFor).
6. **Routing**: Manages navigation within the application.
7. **Dependency Injection (DI)**: A design pattern to manage service instances in a modular way, making the application more testable and maintainable.

**Question 14: What is SPA in Angular?**

**Answer:** SPA (Single Page Application) is a web application or website that interacts with the user by dynamically rewriting the current page rather than loading entire new pages from the server. In Angular, SPA is implemented such that only the necessary parts of the page are updated or re-rendered when navigating, improving performance and user experience by reducing page reloads.

**Question 15: How to implement SPA in Angular?**

**Answer:** To implement an SPA in Angular:

1. **Create Components**: Each view in the application should be represented as a component.
2. **Routing**: Use Angular’s Router to map URLs to components. The RouterModule in Angular allows you to configure the routes and navigate between them without reloading the whole page.
3. **RouterOutlet**: Use the <router-outlet></router-outlet> directive in the main template to specify where the routed component should be displayed.

**Question 16: How to implement routing in Angular?**

**Answer:** To implement routing in Angular:

1. Import RouterModule and Routes in the app module:



1. Define the routes in the Routes array:



1. Add the RouterModule.forRoot(routes) in the imports section of the @NgModule.
2. Use <router-outlet></router-outlet> in the main template for rendering components.

**Question 17: Explain Lazy Loading?**

**Answer:** **Lazy Loading** is a performance optimization technique in Angular where feature modules are loaded on demand rather than upfront during the initial application load. This reduces the initial load time of the application and improves performance, especially in large applications.

**Question 18: How to implement Lazy Loading in Angular?**

**Answer:** To implement lazy loading:

1. Create a module for the feature you want to lazy load (e.g., AdminModule).
2. In the AppRoutingModule, use the loadChildren property in the route configuration to load the module lazily.



1. Ensure that the feature module uses RouterModule.forChild(routes) for child routing.

**Question 19: Define Services?**

**Answer:** A **service** in Angular is a class that contains reusable business logic, data access methods, or utility functions that can be shared across multiple components. Services are typically used to separate the business logic from the component and are injected into components via dependency injection.

Services are created using the @Injectable decorator, making them injectable in other components or services.

**Question 20: What is Dependency Injection?**

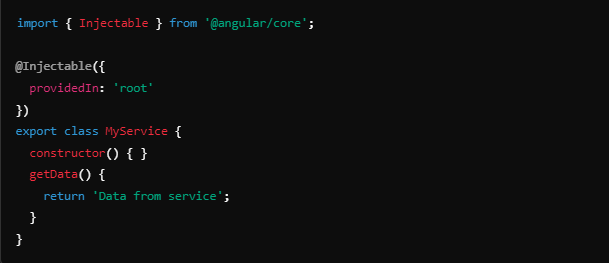
**Answer:** **Dependency Injection (DI)** is a design pattern used in Angular to provide instances of services or objects to components or other services at runtime. Instead of a component creating its dependencies, they are injected by Angular, making the code more modular, testable, and easier to maintain.

DI works by defining providers in the module (or component) that are responsible for creating instances of services. These instances are then injected into components or other services when needed.

**Question 21: How to implement Dependency Injection?**

**Answer:** To implement **Dependency Injection (DI)** in Angular:

1. **Create a Service**:



1. **Inject the Service into a Component**: Use the service in a component by injecting it into the constructor.



**Question 23: What’s the benefit of Dependency Injection?**

**Answer:** The benefits of **Dependency Injection** are:

* **Decoupling**: It decouples the creation of services from their usage, promoting a modular and maintainable architecture.
* **Reusability**: Services can be reused across components.
* **Testability**: It makes components more testable by allowing services to be easily mocked or replaced during testing.
* **Flexibility**: It allows easy switching of service implementations without changing component code.
* **Manageability**: Centralized control of service instances reduces the risk of redundant or inconsistent service usage.

**Question 24: Differentiate between ng serve and ng build?**

**Answer:**

* **ng serve**:
  + Used to compile and serve the application in development mode.
  + It hosts the app in memory and provides hot-reloading for live updates when changes are made.
  + No physical files are generated on disk.
* **ng build**:
  + Used to build the application for deployment.
  + Generates static files (.js, .html, .css, etc.) in the dist/ directory.
  + You can specify additional build options (like --prod for production build).

**Question 25: Explain the --prod parameter in ng build?**

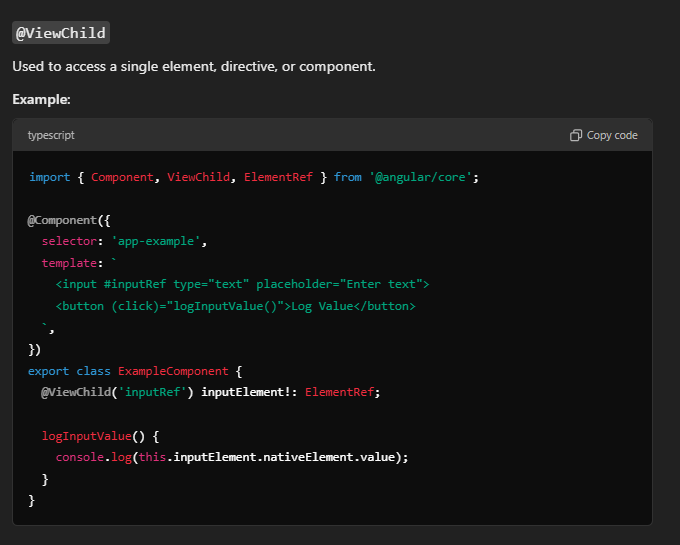
**Answer:** The --prod flag in ng build performs a **production build**, which includes optimizations such as:

* **Ahead-of-Time (AOT) Compilation**: Converts TypeScript and HTML into JavaScript at build time.
* **Minification**: Reduces file size by removing whitespace and shortening variable names.
* **Tree Shaking**: Removes unused code from the final build.
* **Dead Code Elimination**: Optimizes the code by eliminating code paths that are never executed.
* **Caching**: Generates hashed filenames to ensure proper browser caching of files.

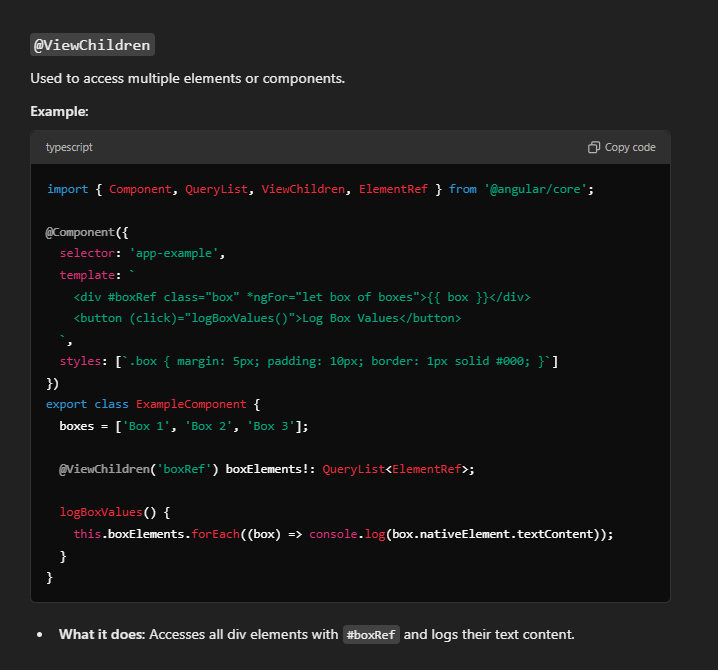
**Question 26: Explain ViewChild and ViewChildren?**

**Answer:**

* **@ViewChild**: Allows you to access a single child component, directive, or element from the DOM in the parent component.
  + Example:



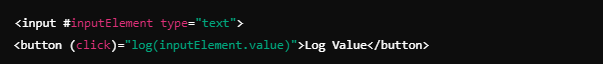
* **@ViewChildren**: Allows you to access multiple child components, directives, or elements as a list (QueryList).
  + Example:



**Question 27: Why do we need Template Reference Variables?**

**Answer:** **Template Reference Variables** (#variableName) are used to reference a DOM element or component instance within the template. They allow easy access to that element or component within the same template to perform actions like calling methods, accessing properties, or interacting with DOM elements without explicit data binding.

* Example:



**Question 28: What is Content Projection?**

**Answer:** **Content Projection** is an Angular feature that allows you to pass content from a parent component into a child component. It enables flexible component designs by letting you insert content into predefined slots of the child component.

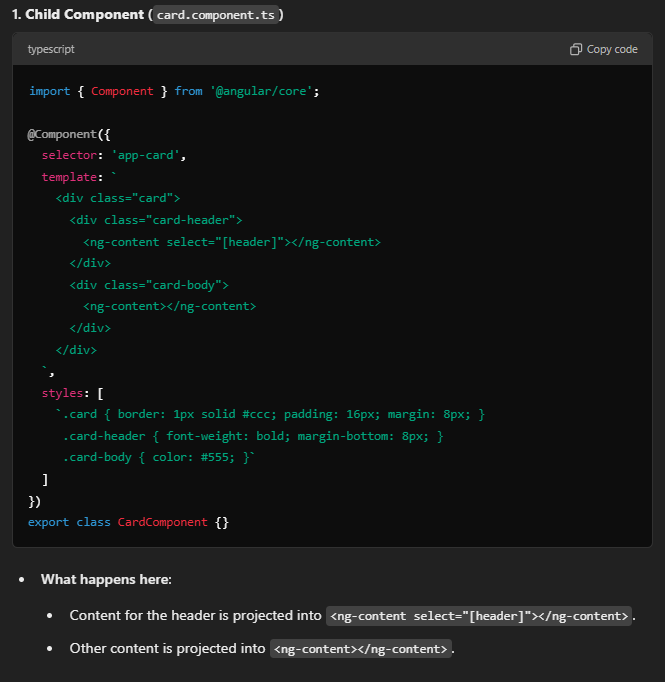
* Example:

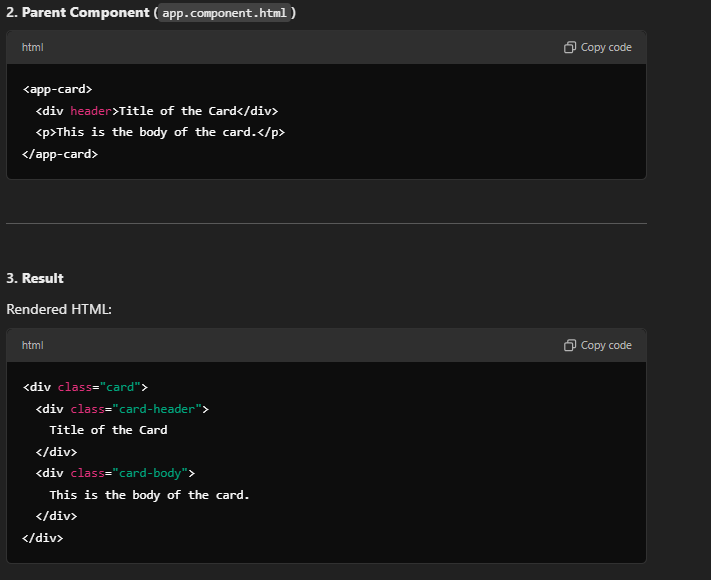


**Question 29: Explain Content Projection Slot?**

**Answer:** A **Content Projection Slot** allows you to define multiple placeholders in a component where content can be projected using <ng-content> with the select attribute. This allows more granular control over where content is inserted.

* Example:

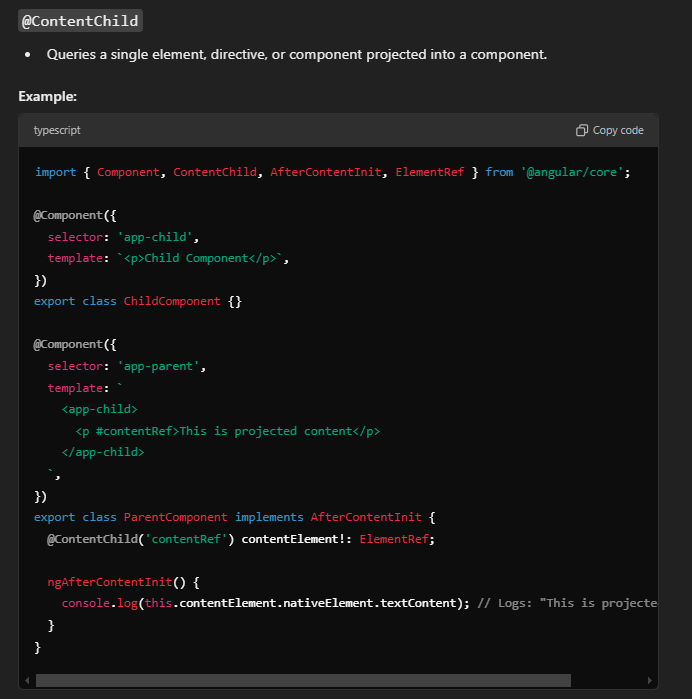




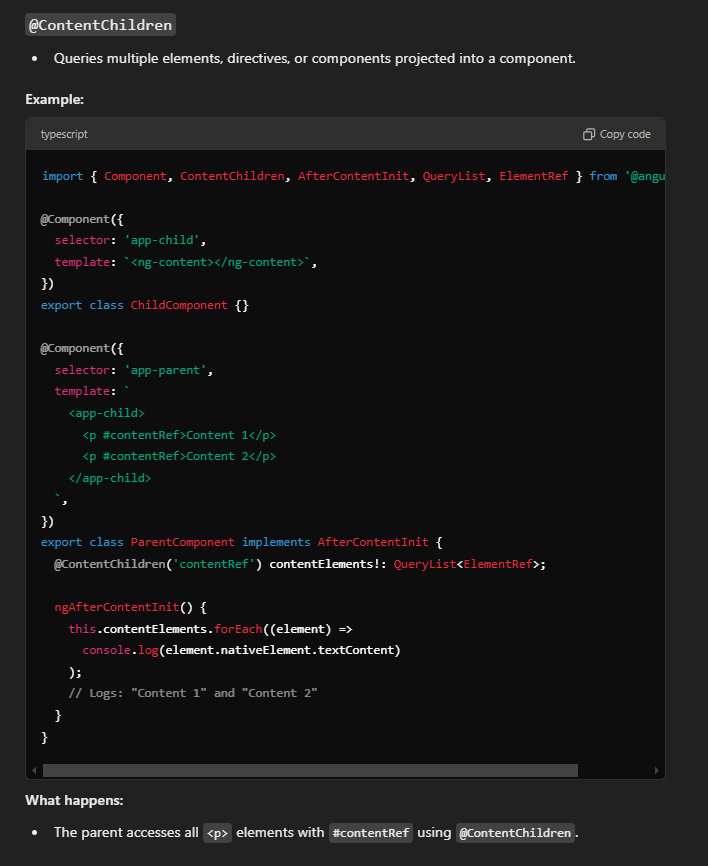
**Question 30: What is ContentChild and ContentChildren?**

**Answer:**

* **@ContentChild**: Used to query and access a single projected content element (DOM element, component, or directive) in the parent component.



* **@ContentChildren**: Used to access multiple projected content elements as a QueryList.



**Question 31: ViewChild vs ViewChildren vs ContentChild vs ContentChildren?**

**Answer:**

* **ViewChild**: Accesses a single child component or DOM element that is part of the component's own template.
* **ViewChildren**: Accesses multiple child components or DOM elements in the component's own template.
* **ContentChild**: Accesses a single projected child component or DOM element passed from the parent component.
* **ContentChildren**: Accesses multiple projected child components or DOM elements passed from the parent component.

**Question 32: Explain the importance of Component Lifecycle?**

**Answer:** The **Component Lifecycle** in Angular defines a series of methods (called lifecycle hooks) that provide control over key moments in a component’s creation, updating, and destruction. Understanding the lifecycle is important for managing state, making API calls, handling DOM interactions, and other tasks.

**Question 33: Explain events and sequence of component lifecycle?**

**Answer:** The major Angular component lifecycle hooks and their sequence:

1. **ngOnChanges**: Invoked when input properties change.
2. **ngOnInit**: Called after the component is initialized (used for initialization logic).
3. **ngDoCheck**: Invoked on every change detection cycle.
4. **ngAfterContentInit**: Called after content projection into the component.
5. **ngAfterContentChecked**: Called after the projected content is checked.
6. **ngAfterViewInit**: Called after the component’s view and child views are initialized.
7. **ngAfterViewChecked**: Called after the component’s view and child views are checked.
8. **ngOnDestroy**: Called just before the component is destroyed.

**Question 34: Constructor vs ngOnInit()?**

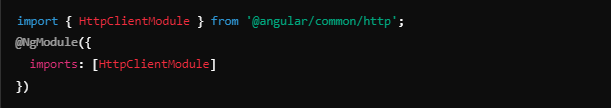
**Answer:**

* **Constructor**: Used for basic initialization of the class and for injecting dependencies. It runs once when the component class is instantiated.
* **ngOnInit()**: It is a lifecycle hook that is specifically used for component initialization, especially when input bindings are set. This is the preferred place to put initialization logic for components.

**Question 35: How to make HTTP calls using Angular?**

**Answer:** To make HTTP calls in Angular:

1. Import HttpClientModule in app.module.ts:



1. Use HttpClient service to make HTTP requests:

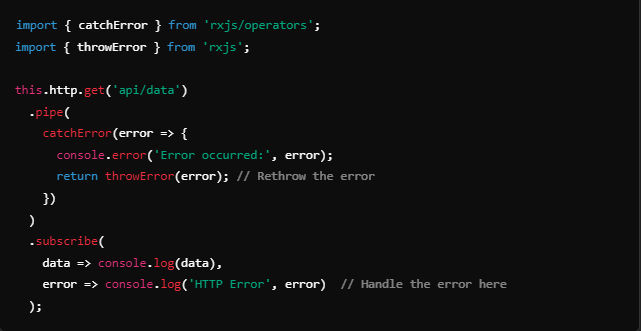


**Question 36: What is the need of Subscribe function?**

**Answer:** The **subscribe()** function is used to **listen** to changes in an observable and receive emitted data. In Angular, observables are commonly used for handling asynchronous data streams, such as HTTP requests or events. The subscribe() method allows the component to react when the observable emits values, handles data, and even manages error or completion scenarios.

**Question 37: How to handle errors when HTTP fails?**

**Answer:** To handle errors in Angular’s HTTP requests, you use the catchError operator from RxJS:



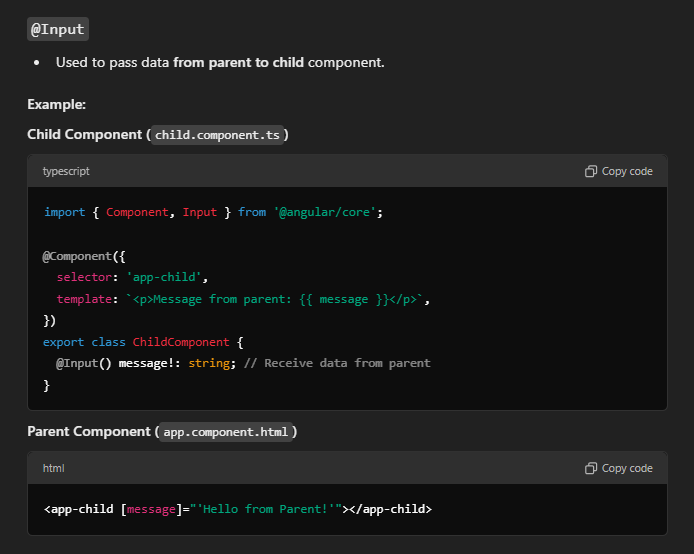
**Question 38: How to pass data between components?**

**Answer:** Data can be passed between Angular components in several ways:

1. **Using @Input and @Output**: Parent-to-child or child-to-parent communication.
2. **Using a Service**: Sharing data through a centralized service.
3. **Using Route Parameters**: Passing data through routing.
4. **Local Storage/Session Storage**: When data needs to be preserved across multiple components.

**Question 39: Explain the importance of @Input, @Output, and Event Emitters?**

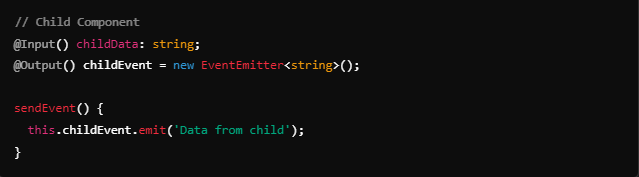
**Answer:**

* **@Input**: Decorator that allows a parent component to pass data to a child component.
* 
* **@Output**: Decorator used in child components to emit events to the parent component.



* **EventEmitter**: Used in conjunction with @Output to emit custom events from a child component.

Example:



**Question 40: How to pass data during routing?**

**Answer:** You can pass data during routing using **route parameters**, **query parameters**, or **router state**.



**Question 41: Is it a good practice to pass data using services?**

**Answer:** Yes, using **services** to share data between unrelated or sibling components is considered a good practice. Services allow centralized data sharing, reducing the coupling between components. This makes the architecture cleaner, testable, and maintainable, especially when sharing complex or stateful data.

**Question 42: What is the need of Angular Pipes?**

**Answer:** **Pipes** in Angular are used to **transform data** in the template before displaying it to the user. Common use cases include formatting dates, numbers, currencies, and strings, or filtering and sorting arrays.

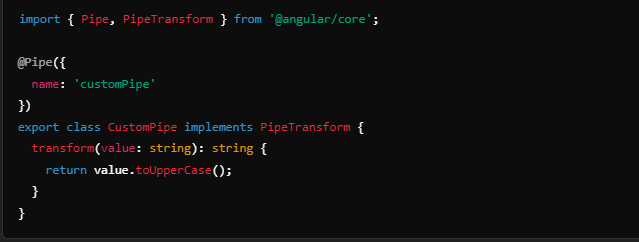
**Question 43: Can you name some built-in Angular Pipes?**

**Answer:** Some built-in Angular pipes include:

* **date**: Formats a date value.
* **uppercase / lowercase**: Converts text to uppercase or lowercase.
* **currency**: Formats a number as a currency.
* **json**: Converts an object to JSON format.
* **async**: Unwraps asynchronous values (like observables or promises).
* **slice**: Extracts a section of an array.

**Question 44: How to create Custom Pipes in Angular?**

**Answer:** You can create a custom pipe by implementing the PipeTransform interface.

Then, use the pipe in a template:



**Question 45: What's the full form of RxJS?**

**Answer:** The full form of **RxJS** is **Reactive Extensions for JavaScript**. It's a library for composing asynchronous and event-based programs using observables.

**Question 46: What is the purpose of RxJS?**

**Answer:** The purpose of **RxJS** is to handle **asynchronous events** and **data streams** in a reactive programming style. It simplifies the complexity of managing asynchronous code (such as HTTP requests, timers, events) through observables, which allow better control over data flow and error handling.

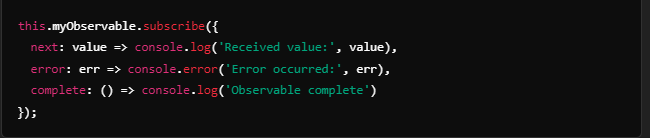
**Question 47: What are Observables and Observers?**

**Answer:**

* **Observable**: Represents a data stream that emits values over time. It can be subscribed to by an observer to receive updates.
* **Observer**: An object that subscribes to an observable to listen for emitted values, errors, or completion.

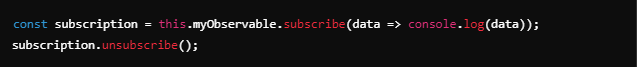
**Question 48: Explain the use of Subscribe with sample code.**

**Answer:** subscribe() is used to listen to an observable and react to emitted values. Here’s an example:



**Question 49: How to unsubscribe in RxJS?**

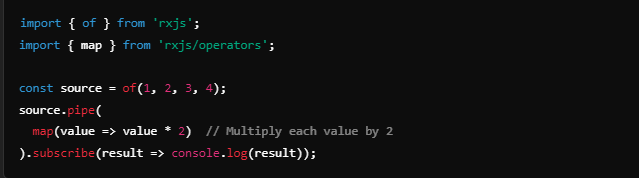
**Answer:** To unsubscribe from an observable, you can store the subscription and call the unsubscribe() method.

This prevents memory leaks, especially in long-lived components.

**Question 50: Explain the concept of operators with sample code.**

**Answer:** **Operators** in RxJS are functions that allow transforming, filtering, and managing data from observables. Common operators include map, filter, mergeMap, catchError.

Example with map:

This will output:



**Question 51: How to install RxJS?**

**Answer:** RxJS is included with Angular by default, so if you have an Angular project, RxJS is already available. However, if you need to install or update it, you can use npm:

**npm install rxjs**

You can also specify a version if needed:

**npm install rxjs@<version>**

**Question 52: Differentiate between Promise and RxJS?**

**Answer:**

1. **Asynchronous Handling**:
   * **Promise**: Represents a single asynchronous operation that will complete in the future. It can only resolve or reject once.
   * **RxJS**: Represents a stream of values over time, allowing multiple values to be emitted over a period.
2. **Multiple Values**:
   * **Promise**: Handles a single value; once resolved, it cannot emit further values.
   * **RxJS**: Can emit multiple values over time, making it suitable for continuous data streams.
3. **Cancellation**:
   * **Promise**: Cannot be canceled once initiated.
   * **RxJS**: Provides cancellation through unsubscribing from the observable.
4. **Error Handling**:
   * **Promise**: Uses .catch() for error handling.
   * **RxJS**: Uses operators like catchError for error handling within the stream.

**Question 53: In Angular where have you used RxJS?**

**Answer:** RxJS is used in Angular for various purposes, including:

* **HTTP Requests**: Using HttpClient to handle API calls that return observables.
* **Reactive Forms**: Handling form value changes using observables.
* **Event Handling**: Listening to events through observables, such as user input or button clicks.
* **State Management**: Using observables to manage state and communicate between components.

**Question 54: Which operators have you used from RxJS?**

**Answer:** Commonly used RxJS operators include:

* **map**: Transforms emitted values.
* **filter**: Emits only values that pass a condition.
* **mergeMap**: Flattens observables into a single observable.
* **switchMap**: Switches to a new observable, unsubscribing from the previous one.
* **catchError**: Handles errors in the observable stream.
* **tap**: Allows side effects (e.g., logging) without affecting the stream.

**Question 55: What is Push/reactive vs Pull/Imperative?**

**Answer:**

* **Push/Reactive**: In a push model, data is pushed to the observer when it's available (e.g., through observables). The observer subscribes to the data source and receives updates automatically.
* **Pull/Imperative**: In a pull model, the consumer pulls data from the source when needed (e.g., using promises). The consumer explicitly requests data, and it only receives what it asks for at that moment.

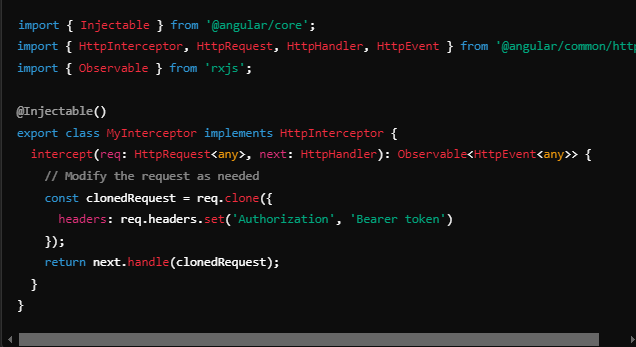
**Question 56: What are Interceptors in Angular?**

**Answer:** **Interceptors** are a feature in Angular that allow you to intercept and modify HTTP requests and responses before they are handled by the application. This can be used for various purposes, such as adding authentication tokens, logging, error handling, or modifying request/response data.

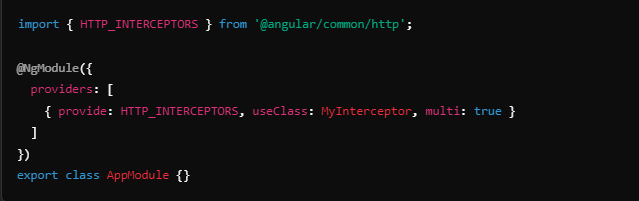
**Question 57: How to implement Interceptors?**

**Answer:** To implement an interceptor, create a class that implements the HttpInterceptor interface and define the intercept method. Here’s an example:

1. Create the interceptor:



1. Register the interceptor in your AppModule:



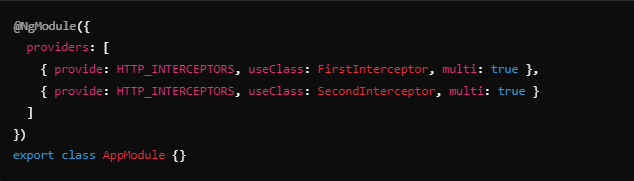
**Question 58: Give some use of Interceptors?**

**Answer:** Interceptors can be used for:

* **Authentication**: Adding authorization tokens to requests.
* **Logging**: Logging request and response data for debugging purposes.
* **Error Handling**: Intercepting and handling HTTP errors globally.
* **Request/Response Transformation**: Modifying request payloads or response data as needed.
* **Caching**: Implementing caching strategies to optimize network requests.

**Question 59: Can we provide multi-Interceptors?**

**Answer:** Yes, you can provide multiple interceptors in Angular. When you register interceptors, set the multi option to true, allowing Angular to handle an array of interceptors. Each interceptor will be executed in the order they are provided:

In this case, FirstInterceptor will run before SecondInterceptor.