### **key concepts of Angular**

* components, modules, templates, data binding, services, dependency injection, and routing
* These concepts form the foundation of Angular development and help in building dynamic and scalable web applications.

**Directives In Angular :**

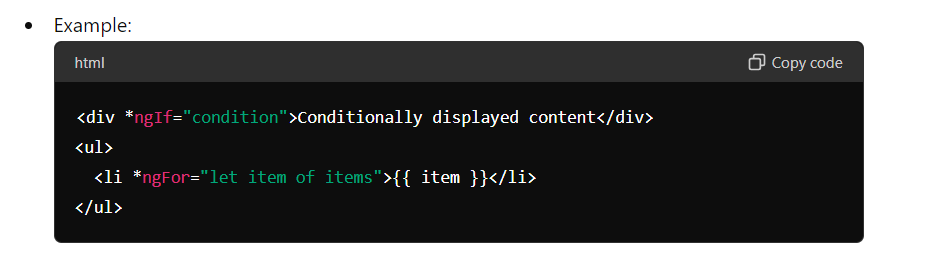
n Angular, directives are a fundamental feature that allow developers to attach behavior to elements in the DOM. They can be used to manipulate the DOM, create reusable components, and manage dynamic data binding. Angular directives are classified into three main types:

1. **Component Directives**:
   * These are the most commonly used directives in Angular.
   * They are defined with the **@Component** decorator.
   * Components are directives with a template. They are the building blocks of an Angular application.



**Structural Directives**:

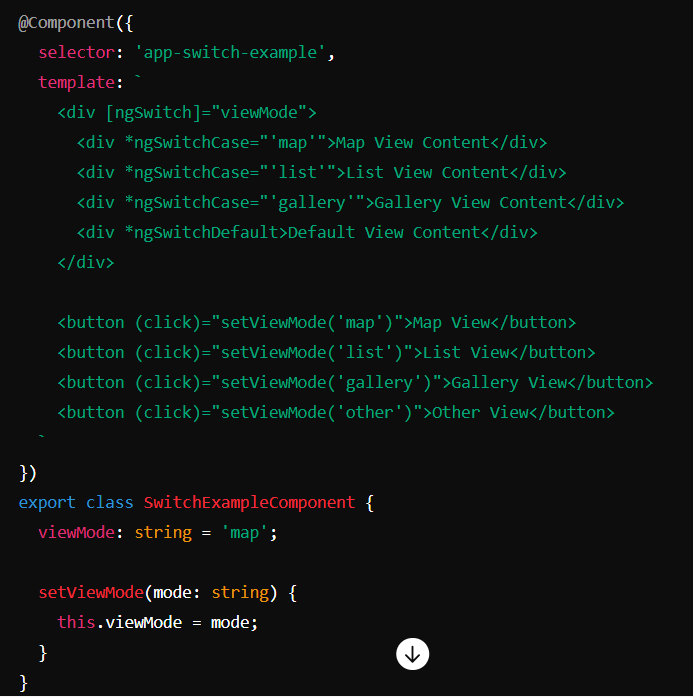
* Structural directives are responsible for HTML layout. They shape or reshape the DOM's structure, typically by adding, removing, or manipulating elements.
* They are applied using a **\*** prefix in the template.
* Common structural directives include **\*ngIf**, **\*ngFor**, and **\*ngSwitch**.



**Attribute Directives**:

* Attribute directives change the appearance or behavior of an element, component, or another directive.
* They are used as attributes on elements.
* Common attribute directives include **ngClass**, **ngStyle**, and custom attribute directives.

**NgSwitch :**

****

**Static Web Application :**

A static website is one where web pages are delivered exactly as they are stored, with no real-time content changes.

**Examples :**

* Personal blog sites.
* Professional portfolios.
* Information sites, such as wikis or reference sites.
* Basic corporate sites that display information about the company and its products or services.
* Galleries of images or videos.

**Dynamic Web Application :**

a dynamic website generates content in real time, typically using databases and scripting languages to provide interactivity and personalized experiences.

**Examples :**

* Facebook
* Tinder
* YouTube
* Uber Eats
* LinkedIn
* Netflix

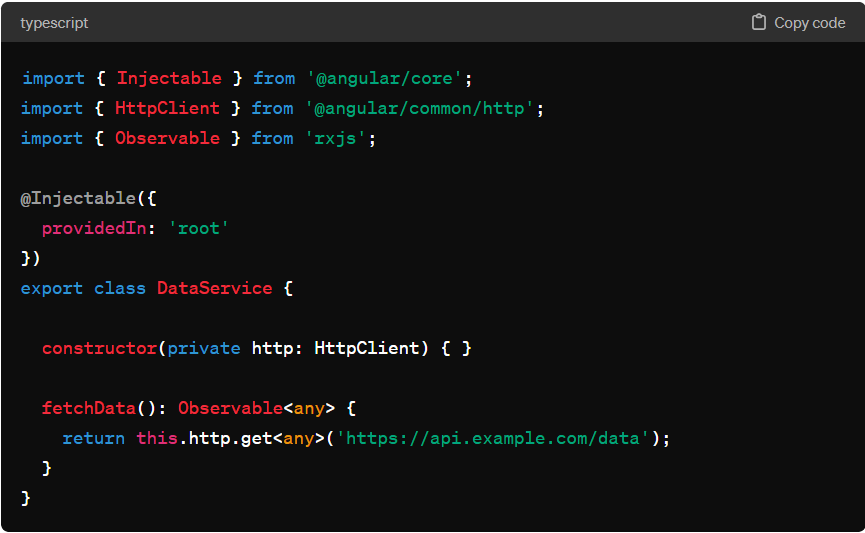
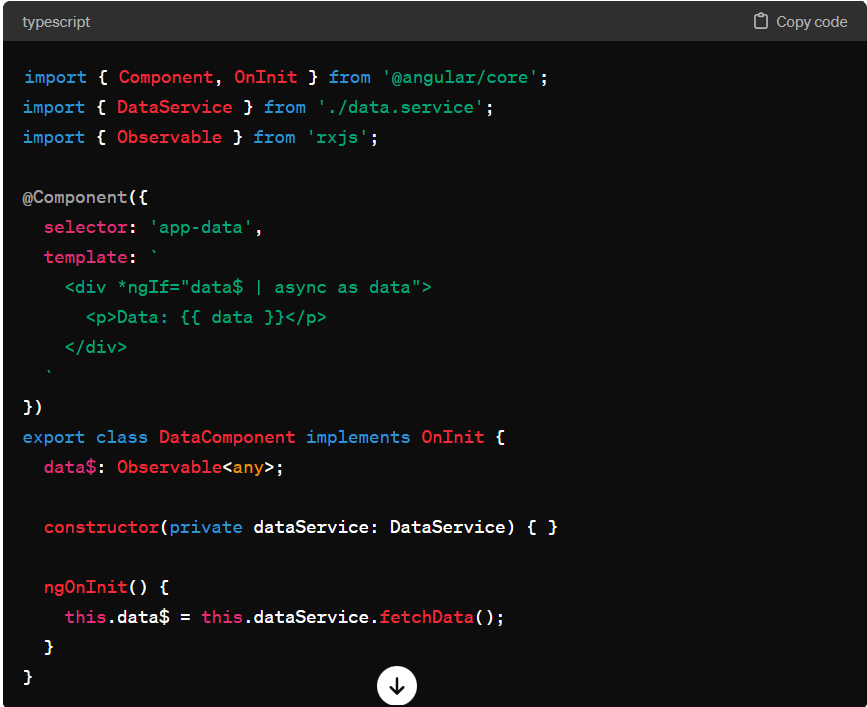
**Scalability :**

Scalability is a web application's ability to handle large data loads of growing digital traffic without any interruptions and degrading in app performance.

**Async :**In Angular, "async" is a feature that helps you handle asynchronous operations, like fetching data from a server or dealing with user input events. When you mark a component property as "async," Angular automatically subscribes to an Observable or a Promise and manages the subscription for you.

Here's a simple example to illustrate:

Suppose you have a service in your Angular application that fetches some data from a server. Let's call it **DataService**. It might have a method like this:

****Now, let's say you want to display this fetched data in a component. You can use the **async** pipe in the template to handle the asynchronous data retrieval. Here's how you would do it in your component:  
****

In this example:

* We inject the **DataService** into our component.
* In the **ngOnInit()** lifecycle hook, we call the **fetchData()** method of the service, which returns an Observable.
* We assign the result to the **data$** property.
* In the template, we use the **async** pipe to subscribe to the Observable (**data$**). Angular automatically subscribes to the Observable, retrieves the data, and updates the view when the data arrives.

So, by using the **async** pipe, Angular takes care of subscribing, unsubscribing, and updating the view with the latest data automatically, making your code cleaner and more manageable.

**Pipe :**

In Angular, a pipe is a feature that allows you to transform data in your application before displaying it in the UI. Think of it as a filter or a formatter for your data.

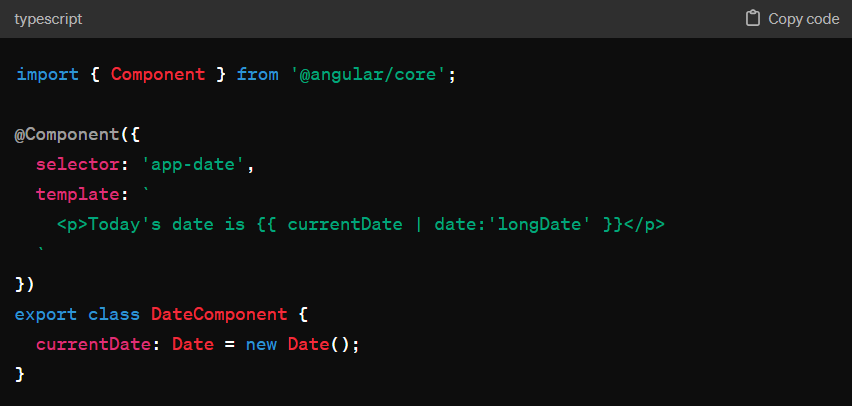
Here's a simple example to help illustrate how pipes work:  
  
**Date Pipe**

Suppose you have a component that displays the current date:

****

This component simply displays the current date. But what if you want to format the date differently, say in a more user-friendly format like "April 16, 2024" instead of the default format provided by JavaScript's **Date** object?

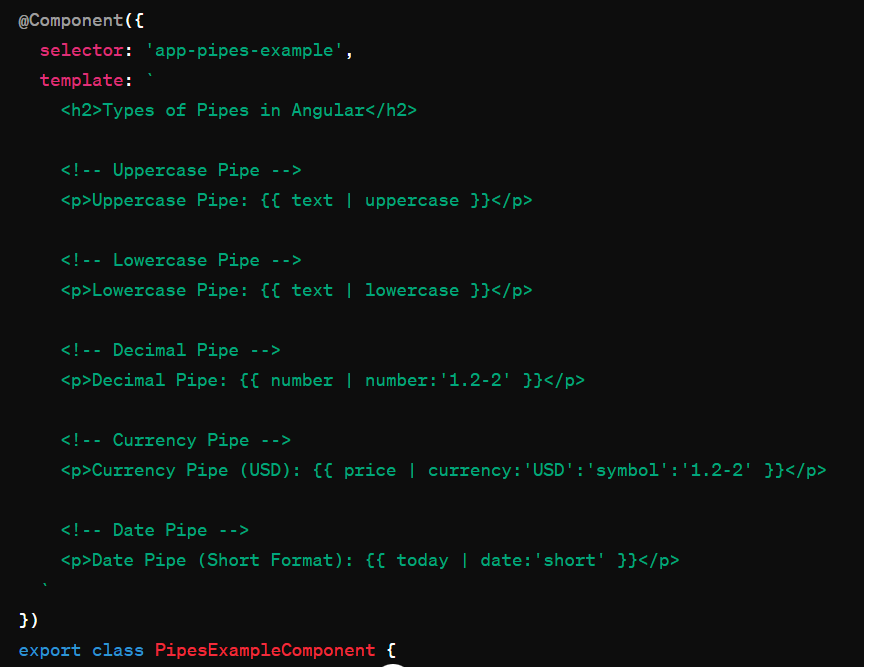
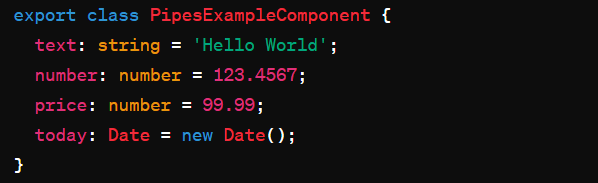
This is where pipes come in handy. Angular provides a built-in pipe called **DatePipe** that you can use to format dates. Here's how you can use it in your component:

****

In this updated component, we've added **| date:'longDate'** after the **currentDate** property. This is using the **DatePipe** to format the date in the "longDate" format, which will display the date in the format like "April 16, 2024".

Angular pipes are versatile and can be used for various purposes like formatting dates, numbers, currency, and even custom transformations. They help keep your templates clean and concise by separating the logic for transforming data from the presentation logic.

So, in simple terms, an Angular pipe is like a tool that helps you transform your data into a format that's more suitable for displaying in your application's UI.

**Types of Pipe :  
  
  
**In this example:

* **Uppercase Pipe**: Converts the text to uppercase.
* **Lowercase Pipe**: Converts the text to lowercase.
* **Decimal Pipe**: Formats the number to have at least one digit before the decimal point, and a minimum of two digits after the decimal point, and a maximum of two digits after the decimal point.
* **Currency Pipe**: Formats the number as currency. In this case, it's formatted as USD currency with the dollar symbol, with at least one digit before the decimal point, and a minimum of two digits after the decimal point, and a maximum of two digits after the decimal point.
* **Date Pipe**: Formats the date. In this case, it's formatted in the short date format.

Each pipe is applied to its corresponding data property (**text**, **number**, **price**, **today**) in the template using the pipe operator **|**. This demonstrates the usage of different types of built-in pipes in Angular.

Certainly! Let's break down the **number** pipe with the **'1.2-2'** parameter:

* **1** before the dot (.) indicates the minimum number of integer digits.
* **2-2** after the dot (.) indicates the minimum and maximum number of digits after the decimal point.

So, **'1.2-2'** specifies the following formatting rules:

1. **Minimum one digit before the decimal point:** If the number is less than 1, it will be displayed as 0.x.
2. **Minimum two digits after the decimal point:** At least two digits will be displayed after the decimal point, even if they are zeros.
3. **Maximum two digits after the decimal point:** No more than two digits will be displayed after the decimal point.

Let's see an example to better understand:

Suppose we have a number **123.4567**.

* **Minimum one digit before the decimal point**: This condition is satisfied.
* **Minimum two digits after the decimal point**: We have four digits (4567) after the decimal point. So, we take the first two, which are 45.
* **Maximum two digits after the decimal point**: We've already taken two digits (45), so we stop here.

After applying the **number:'1.2-2'** pipe to **123.4567**, the output will be **123.45**.

In the expression **{{ price | currency:'USD':'symbol':'1.2-2' }}**, the **currency** pipe is being applied to the **price** variable with the following parameters:

* **'USD'**: Specifies that the currency format should be in US dollars.
* **'symbol'**: Indicates that the currency symbol (in this case, the dollar sign **$**) should be displayed.
* **'1.2-2'**: Defines the format for the number part of the currency. Let's break it down:
  + **1 before the dot (.)**: Specifies the minimum number of integer digits. If the number is less than 1, it will be displayed as 0.x.
  + **2-2 after the dot (.)**: Specifies the minimum and maximum number of digits after the decimal point. At least two digits will be displayed after the decimal point, and no more than two digits will be displayed.

So, the **currency** pipe with **'1.2-2'** parameter ensures that the price is formatted in the following manner:

1. Minimum one digit before the decimal point.
2. Minimum two digits after the decimal point.
3. Maximum two digits after the decimal point.

For example, if **price** is **99.99**, after applying the **currency:'USD':'symbol':'1.2-2'** pipe, it will be displayed as **$99.99**.

**Sync vs Async :**In Angular, the terms "sync" and "async" refer to how code execution is managed, particularly in handling asynchronous operations.

**Sync (Synchronous)**:

* In synchronous code execution, tasks are performed one after another in a sequential manner.
* The program waits for each task to finish before moving on to the next one.
* Synchronous operations block the main thread until they are completed, potentially causing the application to become unresponsive if tasks take too long to finish.

Example: Performing a series of calculations where each calculation depends on the result of the previous one.

**Async (Asynchronous)**:

* In asynchronous code execution, tasks are performed concurrently or independently of each other.
* Asynchronous operations allow the program to continue executing other tasks while waiting for certain operations to complete.
* Asynchronous operations are typically non-blocking, meaning they don't halt the execution of the program.

Example: Making an HTTP request to fetch data from a server, where the application continues to respond to user interactions while waiting for the response.

In Angular, asynchronous operations are common when dealing with tasks such as HTTP requests, timers, or other operations that involve waiting for external resources or events. Angular provides mechanisms such as Observables, Promises, and async/await syntax to handle asynchronous code in a more manageable way.

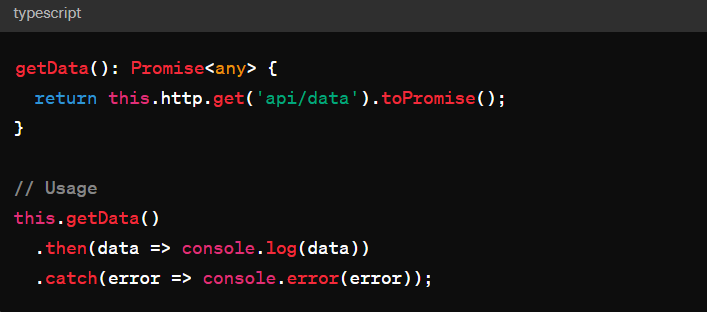
Understanding the difference between synchronous and asynchronous operations is crucial for building responsive and efficient Angular applications. It helps developers make informed decisions about how to structure their code and manage tasks that may have varying execution times.

**Promise vs observable :**

In Angular, both Promises and Observables are used for handling asynchronous operations, but they have some differences in how they work and what features they offer.

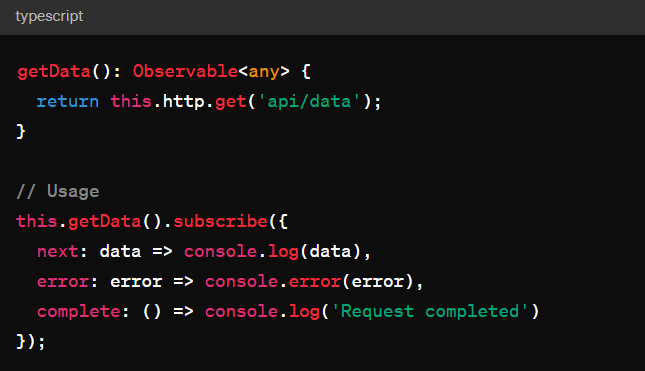
1. **Promise**:
   * Promises are objects that represent the eventual completion (or failure) of an asynchronous operation and allow you to handle its result asynchronously.
   * They have a single value and are not lazy, meaning they execute immediately once created.
   * Promise objects have two main methods: **then()** for handling a successful result and **catch()** for handling errors.
   * Promises are usually used for handling a single asynchronous operation, such as an HTTP request.

Example of using a Promise in Angular:

****

1. **Observable**:
   * Observables represent a stream of data that can be observed over time. They are lazy, meaning they only execute when subscribed to.
   * They provide powerful operators for transforming, combining, and managing streams of data.
   * Observables support multiple values over time and are cancellable.
   * They are used extensively in Angular for handling events, as well as asynchronous operations like HTTP requests.

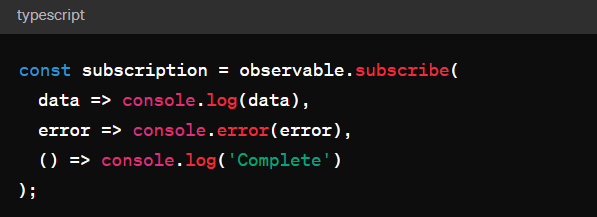
Example of using an Observable in Angular:

****

In general, Observables are preferred in Angular applications due to their flexibility, composability, and support for more complex use cases. However, Promises are still widely used, especially when dealing with APIs or libraries that return Promises.  
  
**Subscribe vs Un Subscribe :**

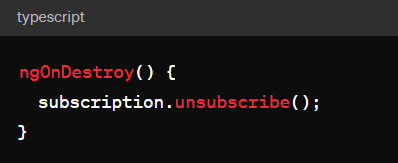
In Angular, when you subscribe to an Observable, you start listening to the stream of data emitted by that Observable. It's important to manage subscriptions properly to avoid memory leaks and unnecessary resource consumption. Angular provides mechanisms to unsubscribe from Observables when they are no longer needed.

1. **Subscribe**: When you subscribe to an Observable, you typically provide a callback function or an observer object with methods to handle the data emitted by the Observable.

****

In the above example, **observable** is the Observable instance, and **subscribe()** is called on it. The callback functions handle the next data emitted, errors, and completion of the Observable respectively.

**Unsubscribe**: It's important to unsubscribe from Observables when you're done listening to them to prevent memory leaks, especially when dealing with Observables that have ongoing emissions such as those from HTTP requests or WebSocket connections.

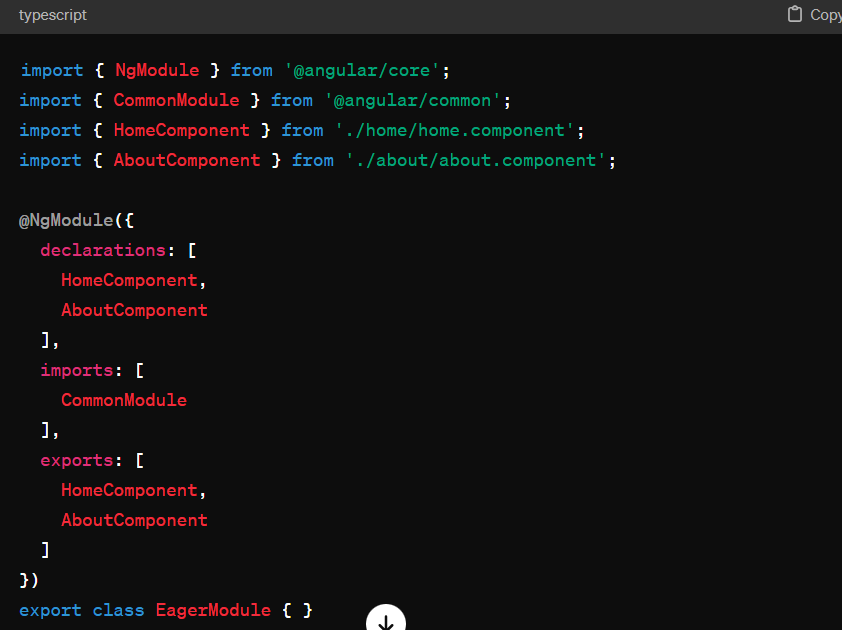
****The **unsubscribe()** method is called on the subscription object within the Angular component's **ngOnDestroy()** lifecycle hook. This ensures that the subscription is unsubscribed when the component is destroyed.

**Lazy vs Eager Loading :**

Lazy loading and eager loading are two strategies for loading modules in Angular applications. They have different characteristics and are suited for different scenarios.

1. **Eager Loading**:
   * Eager loading loads all the necessary modules and components when the application starts.
   * All the modules are bundled together and loaded upfront, increasing the initial load time of the application.
   * Eager loading is suitable for smaller applications or when the size of the application bundle is not a concern.
   * Eager loading is the default loading strategy in Angular.

Example of lazy loading:



**Lazy Loading**:

* Lazy loading loads modules and their components on demand, i.e., when they are required by the application.
* Modules are loaded asynchronously, reducing the initial load time of the application and improving performance.
* Lazy loading is suitable for larger applications with many features or when minimizing the initial bundle size is important for improving page load performance.
* Lazy loading requires additional configuration using Angular's routing system to specify which modules should be loaded lazily.

Example of lazy loading:

****

1. In this example, the **loadChildren** property of the route specifies the module to be lazily loaded when the route is accessed.

In summary, eager loading loads all modules upfront, while lazy loading loads modules on demand. Eager loading is suitable for smaller applications or when initial load time is not a concern, while lazy loading is preferable for larger applications or when minimizing initial load time is important.

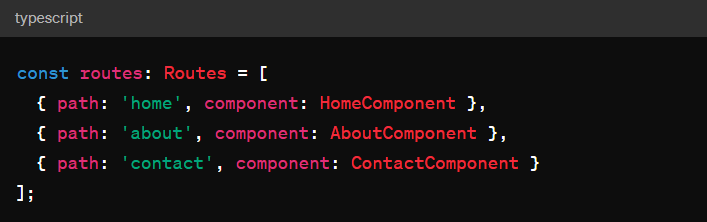
**SPA:**

Single Page Application (SPA) architecture is a common approach used in modern web development, and Angular is well-suited for building SPAs. In an SPA, the entire application runs within a single web page, and navigation between different views or pages is handled dynamically without the need for full page reloads.

Here are some key aspects of building an SPA in Angular:

1. **Component-Based Architecture**:
   * Angular applications are built using components, which are self-contained units of functionality that can be composed together to create complex user interfaces.
   * Each page or view in an Angular application is typically represented by a component.
2. **Routing**:
   * Angular's built-in router allows you to define the navigation structure of your application.
   * You can define routes for different views or components and map them to specific URLs.
   * When a user navigates to a different route, Angular dynamically updates the view without reloading the entire page.

Example of defining routes in Angular:

****

1. **Lazy Loading**:
   * As mentioned earlier, Angular supports lazy loading, which allows you to load modules and their components on demand.
   * This helps to keep the initial bundle size small and improves the performance of the application by loading only the necessary code when needed.
2. **Data Binding and Services**:
   * Angular provides powerful data binding capabilities, allowing you to bind data between components and templates.
   * Services are used to encapsulate business logic, data fetching, and other shared functionality across components.
   * Data binding and services enable seamless communication between different parts of the application.
3. **State Management**:
   * As your application grows, managing application state becomes increasingly important.
   * Angular provides tools like RxJS for handling state in a reactive and predictable manner.
   * Additionally, libraries like NgRx provide a state management solution based on the Redux pattern for managing complex application states.

By leveraging Angular's features such as component-based architecture, routing, lazy loading, data binding, and state management, you can build robust and scalable SPAs that provide a smooth and responsive user experience.

1. **State Management :   
   What is State?**  
   State refers to the data that defines the current condition or snapshot of your application at any given moment. It could be user input, server responses, UI state, or any other piece of information that influences how your application behaves and looks.
2. **Why Manage State?**  
   As your Angular application grows, managing state becomes crucial for maintaining a predictable and scalable application. Without proper state management, it's easy to end up with scattered, inconsistent, or hard-to-debug code.
3. **State Management in Angular:**  
   In Angular, you typically manage state through services, components, and Angular features like RxJS Observables or Angular's built-in mechanisms.
   * **Services**: Angular services are used to store and manage data that needs to be shared across multiple components. They act as singletons, meaning there's only one instance of each service throughout the application.
   * **Component State**: Components can manage their own state using properties and methods. This is suitable for local UI state or data specific to a particular component.
   * **RxJS Observables**: RxJS is a library for reactive programming that Angular heavily relies on. Observables are streams of data that you can subscribe to and reactively update your application based on changes. You can use observables to manage asynchronous data flow and state changes in your application.
   * **State Management Libraries**: Angular developers often use external libraries like NgRx, Akita, or Redux to manage complex application states. These libraries provide patterns and tools for centralizing, organizing, and updating state in a predictable and maintainable way.
4. **Simple Example**:  
   Let's say you have a shopping cart feature in your Angular app. You might have a service called **CartService** to manage the state of the shopping cart. This service could store the items in the cart and provide methods to add, remove, or update items. Components that need access to the shopping cart data can inject and use this service to interact with the cart state.

In essence, state management in Angular involves organizing and updating your application's data in a way that makes it easy to understand, maintain, and scale as your application grows. It's about keeping your data organized, consistent, and easily accessible throughout your application.

**Components :**

In Angular, a component is a fundamental building block of an application's UI. It encapsulates the template, data, and behavior of a part of the user interface. Each component consists of three main parts:

1. **Template**: This defines the HTML that determines the structure of the component's view.
2. **Class**: This is the TypeScript code that provides the component's functionality. It contains properties and methods that can be bound to the template.
3. **Metadata**: This provides additional information about the component, such as its selector (the custom HTML tag used to represent the component), its template URL or inline template, and other configuration options.

Components are reusable and composable, meaning you can create complex UIs by nesting components within each other. They follow a hierarchical structure, where parent components can communicate with and pass data to child components via input properties, and child components can emit events to notify parent components of changes using output properties.

Overall, components help in organizing the UI into smaller, manageable pieces, promoting modularity, reusability, and maintainability in Angular applications.

**Modules :**

In Angular, modules are used to organize an application into cohesive blocks of functionality. A module is a container for a group of related components, directives, pipes, and services that work together to provide a specific feature or functionality to the application. Angular apps are typically built by composing multiple modules together.

There are two main types of modules in Angular:

1. **Root Module**: Also known as the AppModule, it is the entry point of an Angular application. It bootstraps the application and provides the initial configuration. The root module imports other modules and specifies the components that should be loaded when the application starts.
2. **Feature Modules**: These are additional modules created to organize the application into smaller, reusable chunks. Feature modules group together components, directives, pipes, and services that are related to a specific feature or workflow within the application. They help in keeping the codebase modular and maintainable by encapsulating related functionality.

Modules are created using the **@NgModule** decorator, which takes a metadata object that specifies various properties such as:

* **declarations**: An array of components, directives, and pipes that belong to the module.
* **imports**: An array of other modules that this module depends on. This can include both Angular core modules and third-party modules.
* **providers**: An array of services that are available to all components and services within the module.
* **exports**: An array of components, directives, and pipes that should be accessible to other modules that import this module.
* **bootstrap**: Specifies the root component(s) that should be bootstrapped when the module is loaded.

By organizing code into modules, Angular promotes modularity, reusability, and maintainability of the application. It also facilitates lazy loading, where modules are loaded on-demand, improving the application's performance by reducing the initial load time.

**Templates :**

In Angular, templates are the part of components that define the user interface (UI) of your application. They represent the HTML views that Angular renders dynamically based on data and logic. Templates define the structure, layout, and appearance of the UI that users interact with.

Here's a breakdown of what templates in Angular entail:

1. **HTML Markup**: Templates primarily consist of HTML markup, just like traditional HTML files. You can use standard HTML tags, attributes, and elements to create the visual elements of your application.
2. **Interpolation**: Angular allows you to embed expressions within double curly braces (**{{ }}**) directly in your HTML templates. These expressions are evaluated by Angular and replaced with their corresponding values from the component's data. For example, **{{ user.name }}** would display the name of the user.
3. **Binding**: Angular supports both one-way and two-way data binding, allowing you to bind data from your component's TypeScript code to your HTML template. One-way data binding (**{{ }}**) updates the UI whenever the data changes in the component, while two-way data binding (**[(ngModel)]**) synchronizes changes between the UI and the component's data.
4. **Directives**: Angular directives are special markers on DOM elements that tell Angular to do something with that element or its children. Directives can manipulate the DOM, modify element behavior, or conditionally render elements based on certain conditions. Angular provides built-in directives like **ngIf**, **ngFor**, and **ngClass** that you can use in your templates.
5. **Event Binding**: You can bind DOM events such as clicks, keypresses, or input changes to methods in your component's TypeScript code using Angular event binding syntax **(event)="methodName()"**. This allows you to handle user interactions and trigger actions in response to those interactions.
6. **Structural Directives**: Angular also provides structural directives like **\*ngIf** and **\*ngFor** that allow you to conditionally include or exclude elements from the DOM or repeat a set of elements based on collection data.

Overall, templates in Angular provide a powerful mechanism for creating dynamic and interactive user interfaces by combining HTML markup with Angular-specific features like data binding, directives, and event handling. They play a crucial role in separating the presentation logic from the business logic in your Angular applications.

**Data binding :**

Data binding in Angular is the process of synchronizing data between the component's TypeScript code (the data model) and the HTML template (the view). It allows you to keep the UI in sync with the underlying data and respond to user input or changes in the application state.

There are mainly two types of data binding supported in Angular:

1. **One-Way Data Binding**: One-way data binding binds data from the component's TypeScript code to the HTML template in a single direction. Changes in the component's data model update the UI, but changes in the UI do not affect the data model. One-way data binding is achieved using interpolation (**{{ }}**) and property binding (**[ ]**).
   * Interpolation: Data from the component's class is interpolated directly into the HTML template. For example, **{{ username }}** in the template will display the value of the **username** property from the component.
   * Property Binding: Allows you to bind the value of an HTML attribute or property to a property in the component's class. For example, **[disabled]="isDisabled",<img [src]=”url”>** binds the **disabled** property of an HTML element to the **isDisabled** property in the component, enabling/disabling the element based on the value of **isDisabled**.



1. **Two-Way Data Binding**: Two-way data binding combines both property binding and event binding to establish a synchronization between the data model and the UI in both directions. Changes in the component's data model update the UI, and changes in the UI update the data model. Two-way data binding is typically used with form elements such as input fields, checkboxes, and select boxes.
   * Banana-in-a-Box Syntax (**[(ngModel)]**): Angular provides a special syntax called banana-in-a-box (**[(ngModel)]**) for two-way data binding. It binds the value of an input element to a property in the component's class and listens for changes in the input element's value, updating the component's property accordingly.

Data binding in Angular helps streamline the development process by automatically updating the UI when the underlying data changes, reducing the need for manual DOM manipulation and ensuring a consistent and reactive user experience. It is one of the core features that make Angular a powerful and productive framework for building dynamic web applications.

**Event Binding :**

Event binding in Angular is a mechanism that allows you to listen for and respond to DOM events triggered by user interactions such as clicks, keypresses, mouse movements, and form submissions. It enables you to execute custom logic or trigger actions in response to these events.

Event binding is typically implemented in the HTML templates of Angular components using Angular's event binding syntax, which involves binding event handlers defined in the component's TypeScript code to specific DOM events in the template.

Here's a basic example of event binding in an Angular template:

<button (click)="onClick()">Click me!</button>  
You can bind to various DOM events using event binding syntax, such as **(click)**, **(mouseover)**, **(keydown)**, **(input)**, etc. Event binding allows you to handle a wide range of user interactions and respond to them dynamically.

**Services :**

In Angular, services are a fundamental part of the architecture used for organizing and sharing code across an application. Services are TypeScript classes that are responsible for encapsulating reusable functionality, such as data fetching, logging, authentication, and other business logic. They provide a way to centralize code that can be shared across multiple components, directives, or other services within an Angular application.

Here are some key characteristics and purposes of services in Angular:

1. **Reusability**: Services promote code reusability by encapsulating common functionality into standalone units that can be injected into any component or service within the application.
2. **Separation of Concerns**: Services help maintain a clear separation of concerns by keeping business logic and data manipulation separate from the presentation layer (components) of the application.
3. **Singleton Pattern**: By default, Angular services are singletons, meaning there is only one instance of a service created and shared throughout the application. This ensures that data and state are consistent across components and allows for efficient memory management.
4. **Dependency Injection (DI)**: Angular's dependency injection system is used to provide instances of services to components or other services that depend on them. This promotes loose coupling between components and services, making the application more modular and easier to maintain.
5. **HTTP Communication**: Services are commonly used to interact with external APIs or backend services using Angular's built-in HTTP client module. They encapsulate HTTP request logic, making it easier to manage API interactions and handle responses.
6. **Business Logic**: Services often contain business logic, such as data manipulation, validation, or calculations, that is independent of any specific component. This allows the same logic to be reused across different parts of the application.

To create a service in Angular, you typically use the Angular CLI to generate a new service file (**ng generate service serviceName**) or create it manually. Then, you define the service class with its methods and properties. Finally, you register the service with Angular's dependency injection system by providing it in the **providers** array of an Angular module or component.

Overall, services play a crucial role in Angular applications by facilitating code organization, reusability, and maintainability, and they are an essential part of building scalable and modular applications.

**Dependency injection :**

In Angular, Dependency Injection (DI) is a design pattern and a core concept that allows you to inject dependencies (such as services or other objects) into components, directives, or other services. Angular's DI system provides a way to manage the dependencies of your application, making it easier to create modular, reusable, and testable code.

**Routing :**

Routing in Angular refers to the process of navigating between different views (or pages) in a single-page application (SPA). It allows you to define navigation paths, associate them with specific components, and handle routing events such as navigation to a URL, navigation within the application, and passing parameters to routes.

**Flow of Angular Application :**

1. **Bootstrapping**: Angular bootstraps the root module (**AppModule**) of the application.
2. **Initialization of AppModule**: The root module (**AppModule**) initializes and configures the application, including setting up dependencies, importing other modules, and specifying components, services, and providers.
3. **Initialization of AppComponent**: The root component (**AppComponent**) of the application initializes, and its associated template is rendered.
4. **Optional: Initialization of Angular Router**: If the application uses Angular's router, the router module initializes, and configured routes are set up. This includes parsing the URL, matching it to defined routes, and rendering the appropriate component for the requested route inside the **<router-outlet>**.
5. **Dependency Injection**: Angular's dependency injection system starts providing instances of services, components, and other objects as needed and injects their dependencies using the defined providers.
6. **Change Detection**: Angular's change detection mechanism begins running, monitoring changes to the application's data and updating the view accordingly.
7. **HTTP Requests (Optional)**: If the application makes HTTP requests to a server, Angular's HTTP client module sends requests to the server and handles responses asynchronously.
8. **Continuous Running**: Angular continues to monitor for changes, handle user interactions, and respond to events as long as the application is running in the browser.

These components and processes work together to initialize and run an Angular application, providing a smooth and responsive user experience.

**Guards :**

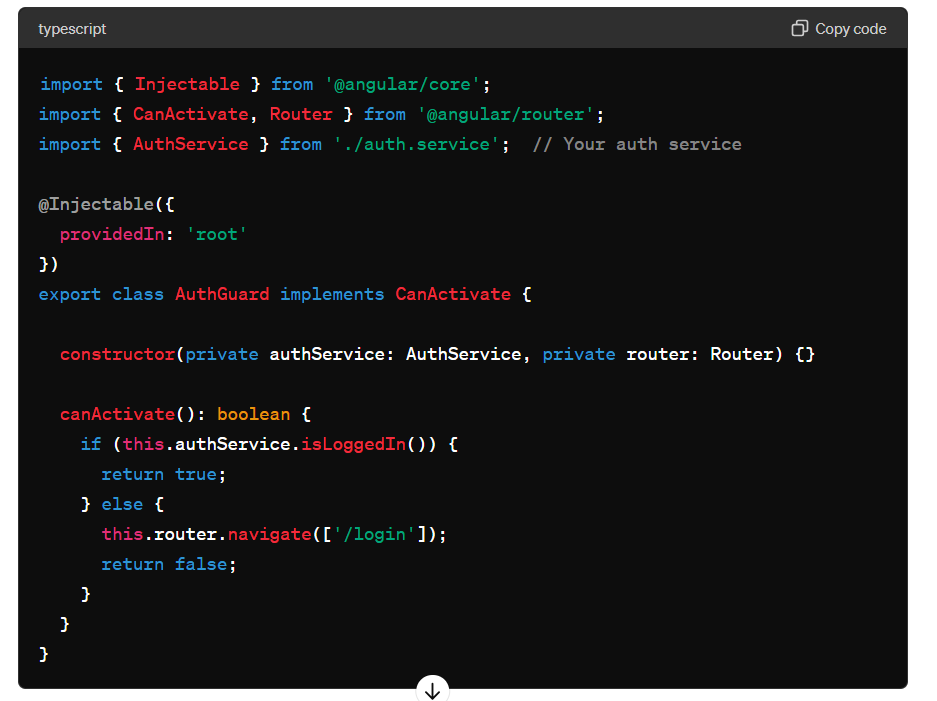
In Angular, guards are a key part of routing that allow you to control access to various parts of your application. They are used to implement logic that determines whether a route can be activated, deactivated, loaded, or if a user can leave a particular route. There are four main types of guards in Angular:

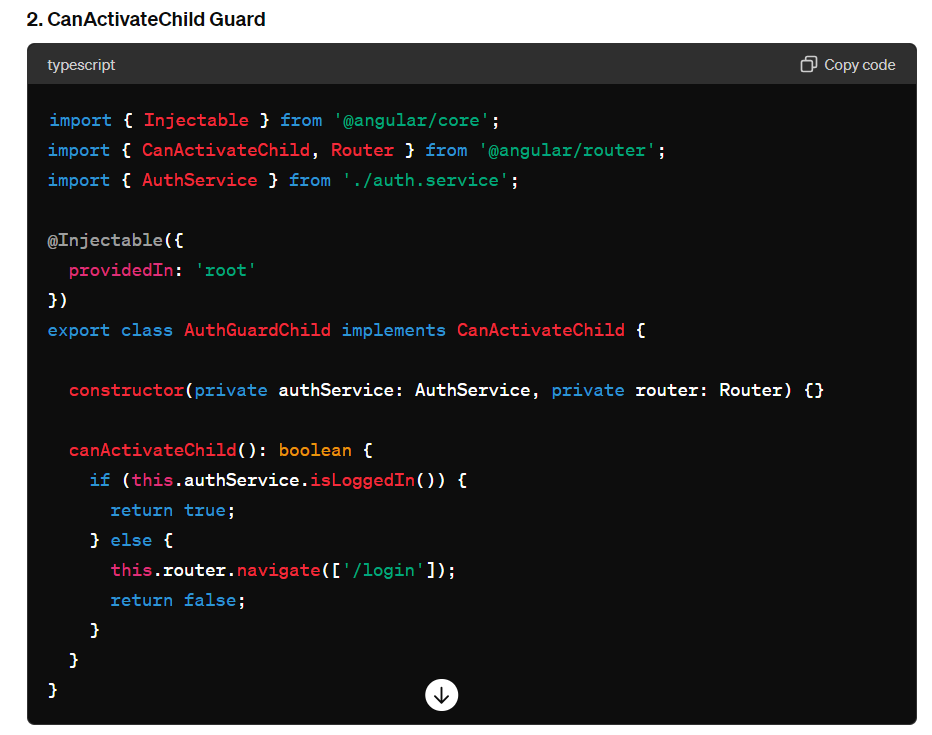
1. **CanActivate**: This guard determines if a route can be activated. It’s useful for checking if a user is authenticated before allowing access to certain routes.
2. **CanActivateChild**: This guard works like **CanActivate** but is used to check if child routes can be activated.
3. **CanDeactivate**: This guard is used to check if a route can be deactivated. It’s commonly used to prompt users if they have unsaved changes and try to navigate away.
4. **Resolve**: This guard is used to perform data retrieval before a route is activated. It ensures that the necessary data is available before the route is loaded.
5. **CanLoad**: This guard determines if a module can be loaded asynchronously. It’s typically used for lazy loading of routes.

### Implementing Angular Guards

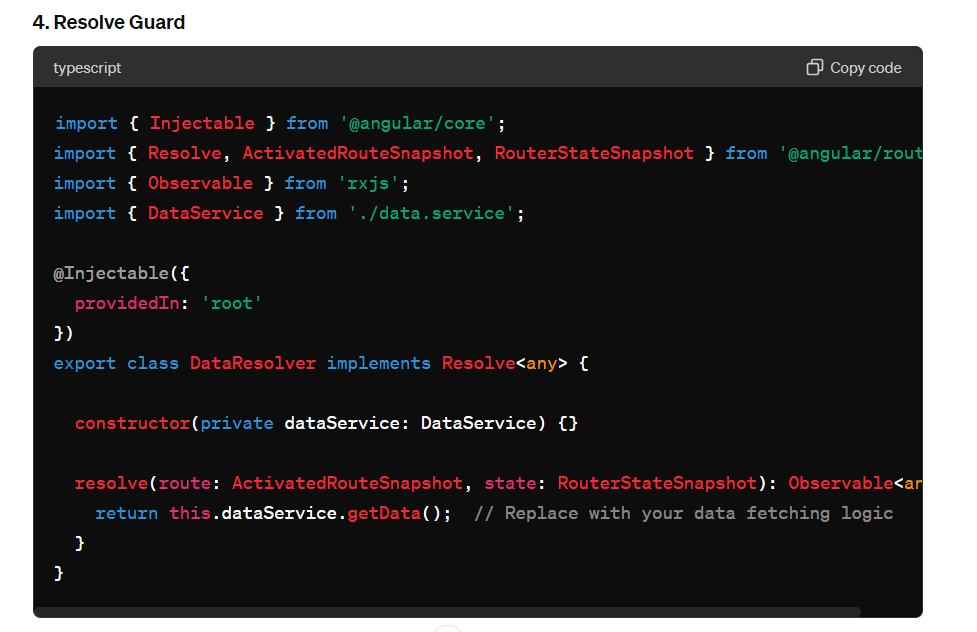
Here's a brief overview of how to implement these guards:

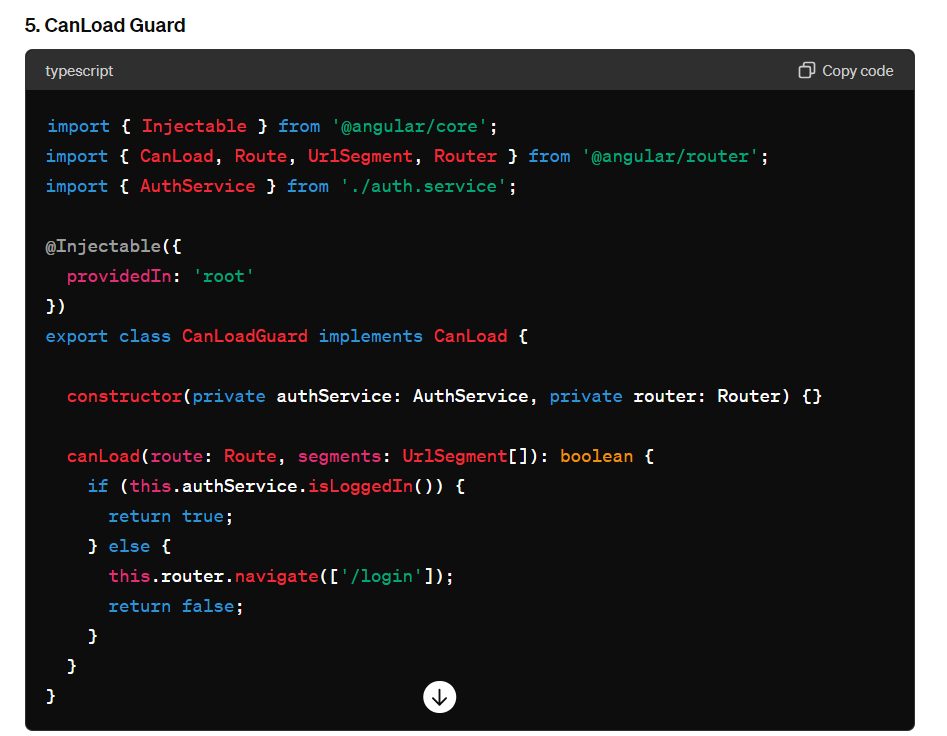
#### 1. CanActivate Guard

****

****

****

****

****

**Snapshot :**

In Angular, a snapshot refers to a static snapshot of the route state taken at a particular moment. The snapshot is used to retrieve route parameters, query parameters, and other data that do not change during the lifetime of a component. This is particularly useful when you need to access the route parameters or other route data synchronously without subscribing to the **Observable** provided by the Angular Router.

When you want to access the route parameters using a snapshot, you typically use the **ActivatedRoute** service. The **ActivatedRoute** service provides access to the current route associated with the component. By using its **snapshot** property, you can get the current state of the route.

constructor(private route: ActivatedRoute) {}

ngOnInit() {

this.userId = this.route.snapshot.paramMap.get('id');

}

ngOnInit() {

this.category = this.route.snapshot.queryParamMap.get('category');

}