**Interceptors**

In Angular, interceptors are a powerful feature that allows you to intercept and modify HTTP requests and responses globally across your application. They provide a way to pre-process or post-process HTTP requests or responses before they are sent or received by the actual HTTP client.

Interceptors in Angular are implemented as classes that implement the `HttpInterceptor` interface. This interface requires the implementation of the `intercept` method, which receives an `HttpRequest` object and a `HttpHandler` object as parameters.

The `intercept` method allows you to inspect, modify, or even short-circuit the request or response flow. You can modify request headers, add authentication tokens, handle errors, perform logging, and more. The `intercept` method returns an observable of `HttpEvent`, which allows you to chain multiple interceptors or handle the response.

Interceptors can be used for various purposes, such as:

1. Authentication: You can add authentication headers or tokens to outgoing requests or handle authentication-related errors.

2. Logging: Interceptors can log request and response information for debugging or analytics purposes.

3. Error handling: You can intercept and handle HTTP errors globally, providing a centralized error handling mechanism.

4. Caching: Interceptors can implement caching strategies, allowing you to cache responses and serve them without making additional requests.

5. Response transformation: You can modify the response data, such as parsing, transforming, or filtering the data before it reaches the client code.

To use an interceptor, you need to provide it as a provider in your Angular module using the `HTTP\_INTERCEPTORS` token. By specifying `multi: true`, you allow multiple interceptors to be registered.

import { HTTP\_INTERCEPTORS } from '@angular/common/http';

@NgModule({

// ...

providers: [

// ...

provide: HTTP\_INTERCEPTORS,

useClass: MyInterceptor,

multi: true,

},

})

export class AppModule {}

The order of the interceptors in the provider array determines the order of execution. The interceptors are executed in the order they are registered.

For example if you provide three interceptors called A, B, C in that order, as the Angular Docs say: "... Requests flow in A->B->C and responses flow C->B->A

By using interceptors, you can centralize common functionalities related to HTTP requests and responses and make your code more modular, reusable, and maintainable.

**Guards**

In Angular, guards are used to control the navigation and access to routes in your application. They provide a way to protect routes based on certain conditions or permissions, allowing you to implement authentication, authorization, and other route-related policies.

Angular offers several types of guards, each serving a specific purpose:

1. CanActivate: This guard is used to determine if a user is allowed to navigate to a particular route. It checks if a certain condition is met before allowing access. For example, you can use it for implementing authentication guards to ensure that only authenticated users can access certain routes.
2. CanActivateChild: Similar to CanActivate, but specifically for child routes. It allows you to control access to child routes of a parent route.
3. CanDeactivate: This guard is used to determine if a user is allowed to navigate away from a particular route. It can be used to implement confirmation dialogs or unsaved changes checks before leaving a page.
4. CanLoad: This guard is used to control if a module can be loaded lazily. It is useful when you have lazy-loaded modules and want to control if the module should be loaded based on certain conditions, such as authentication or user roles.
5. Resolve: This guard is used to fetch some data before a route is activated. It allows you to retrieve necessary data for a route from a remote server or any other data source and make it available to the route component.

To implement a guard, you create a class that implements the corresponding guard interface and provides the necessary logic within the guard's methods. For example, if you want to implement a CanActivate guard, you would implement the CanActivate interface and define the canActivate method to check if the user is authenticated.

To use a guard, you specify it in the route configuration using the canActivate property or other applicable guard properties.

const routes: Routes = [

{

path: 'admin',

component: AdminComponent,

canActivate: [AuthGuard]

},

// Other routes...

];

In the example above, the AuthGuard is specified in the canActivate property of the admin route. This ensures that the AuthGuard is executed before allowing access to the AdminComponent route.

Guards provide a powerful mechanism to control route access and implement various navigation-related policies in your Angular application.

**Directives**

In Angular, directives are a way to extend and manipulate the behavior of HTML elements. They allow you to attach custom behaviors to elements, modify their appearance, or manipulate their structure.

There are three types of directives in Angular:

1. **Component** Directives: These directives are used to create reusable UI components. They consist of a template and associated logic encapsulated within a class. Component directives are used as custom HTML elements in your application.

2. **Attribute** Directives: These directives modify the behavior or appearance of an element or component. They are applied as attributes on HTML elements and are used to enhance the existing functionality of elements.

3. **Structural** Directives: These directives modify the structure of the DOM by adding or removing elements based on certain conditions. They are used to control the rendering and visibility of elements based on conditions or iteratively rendering elements based on collections of data.

Directives can be created using the `@Directive` decorator in Angular. They can define properties, methods, and lifecycle hooks to interact with the element they are applied to or to perform certain actions during the component's lifecycle.

To use a directive in your templates, you apply them as attributes to HTML elements using square brackets (`[]`) for attribute directives or with an asterisk (`\*`) for structural directives.

Here's a basic example of an attribute directive that changes the background color of an element:

import { Directive, ElementRef, Renderer2 } from '@angular/core';

@Directive({

selector: '[appHighlight]'

})

export class HighlightDirective {

constructor(private elementRef: ElementRef, private renderer: Renderer2) {

this.renderer.setStyle(this.elementRef.nativeElement, 'background-color', 'yellow');

}

}

In the above example, the `HighlightDirective` is an attribute directive that applies a yellow background color to any element it is applied to using the `Renderer2` API.

You can then use the `appHighlight` directive in your HTML like this:

<p appHighlight>Highlighted Text</p>

This would result in the `<p>` element having a yellow background color applied to it.

Overall, directives in Angular provide a powerful way to extend and customize the behavior and appearance of HTML elements and components in your application.

**The \* syntax**

The `\*` syntax is a shorthand way of writing a structural directive. When you use the `\*` syntax with a directive selector, such as `\*appHasRole`, Angular automatically expands it into a `<ng-template>` element with the directive applied.

`<h1 \*appHasRole="'hr'"`>HRTEST</h1> directive is expanded into:

<ng-template [appHasRole]="'hr'">

<h1>HRTEST</h1>

</ng-template>

The `appHasRole` directive is applied to the `<ng-template>` element, and the `<h1>` element becomes its content.

The role of the `\*` syntax is to indicate that the directive is a structural directive and to trigger the expansion of the directive into the corresponding `<ng-template>` element.

By using the `\*` syntax, you can apply structural directives to HTML elements in a more concise and readable way. It makes the code easier to understand and maintain by clearly indicating that the directive has a structural effect on the DOM.

It signals that the directive has a structural effect on the DOM and triggers the expansion of the directive into a <ng-template> element.

**PIPE**

In general, the `pipe` function is a common functional programming concept that allows you to chain multiple functions or operators together to transform or manipulate data. It takes the output of one function and passes it as the input to the next function, creating a sequence of operations.

In the context of TypeScript and RxJS (Reactive Extensions for JavaScript), the `pipe` function is provided by the RxJS library and is used to compose operators for handling asynchronous streams of data, often referred to as observables. Observables are used to represent sequences of values that are emitted over time.

The `pipe` function in RxJS allows you to apply multiple operators to an observable in a concise and readable manner. Each operator takes the input observable, performs a specific operation on the emitted values, and returns a new observable. These operators can include transformations, filtering, mapping, combining, and more.

By using the `pipe` function, you can create a pipeline of operations on an observable, where the output of each operation becomes the input for the next one. This allows you to apply a series of transformations or actions to the data emitted by the observable, achieving complex data processing or manipulation.

In summary, the `pipe` function is a powerful tool for composing operations on observables in RxJS, enabling you to chain and combine multiple operators to transform, filter, or process asynchronous data streams.

**TAP**

In general, the `tap` function is a utility function that allows you to perform side effects or actions on the elements of a stream without modifying the stream itself. It is commonly used in functional programming and reactive programming paradigms.

In the context of TypeScript and RxJS (Reactive Extensions for JavaScript), the `tap` function is provided by the RxJS library and is used as an operator on observables. It allows you to intercept the values emitted by an observable and perform actions or side effects based on those values, such as logging, updating external state, or triggering other operations.

The `tap` function takes a callback function as its argument. This callback function is executed for each value emitted by the observable, allowing you to perform custom actions on those values. The value emitted by the observable is passed as an argument to the callback function.

The key aspect of the `tap` function is that it does not modify the values emitted by the observable or the stream itself. It is purely used for performing actions on the values while allowing them to continue flowing through the stream. This makes it useful for debugging, logging, or updating external state based on the values emitted by the observable.

In summary, the `tap` function in RxJS is an operator that allows you to perform side effects or actions on the values emitted by an observable without modifying the observable or its stream. It provides a way to perform custom actions based on the emitted values, such as logging or updating external state, while preserving the original data flow.

**‘next’ method**

In general, the next method is used in the context of an Observable or a Subject to emit or publish a value to the subscribers of that Observable or Subject. It is a fundamental method in reactive programming libraries like RxJS.

In RxJS, an Observable represents a sequence of values that can be observed over time, and a Subject is a special type of Observable that allows values to be multicasted to multiple subscribers. Both Observables and Subjects provide the next method to emit values.

The next method typically takes a single argument, which is the value to be emitted. When the next method is called, the provided value is emitted to all the subscribers of the Observable or Subject. Each subscriber will receive the emitted value as part of their subscription's data flow.

In summary, the next method is used to emit values in the context of Observables and Subjects. It allows you to publish values to the subscribers of the Observable or Subject, enabling the propagation of data through the reactive streams.