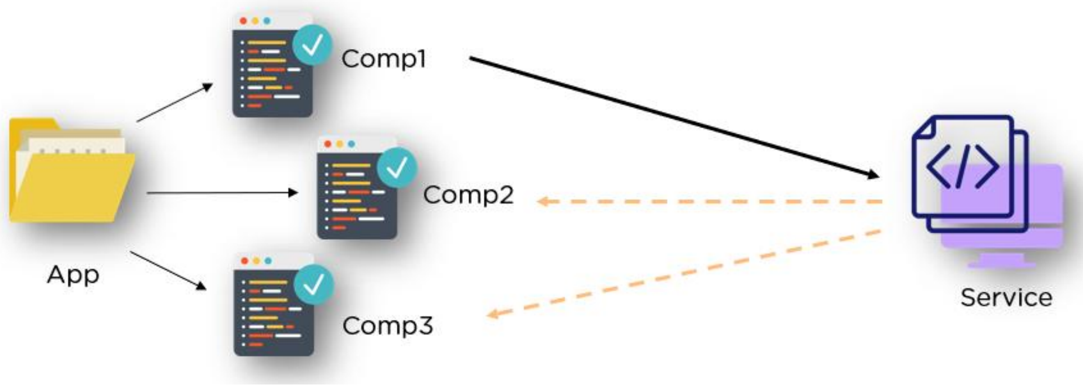
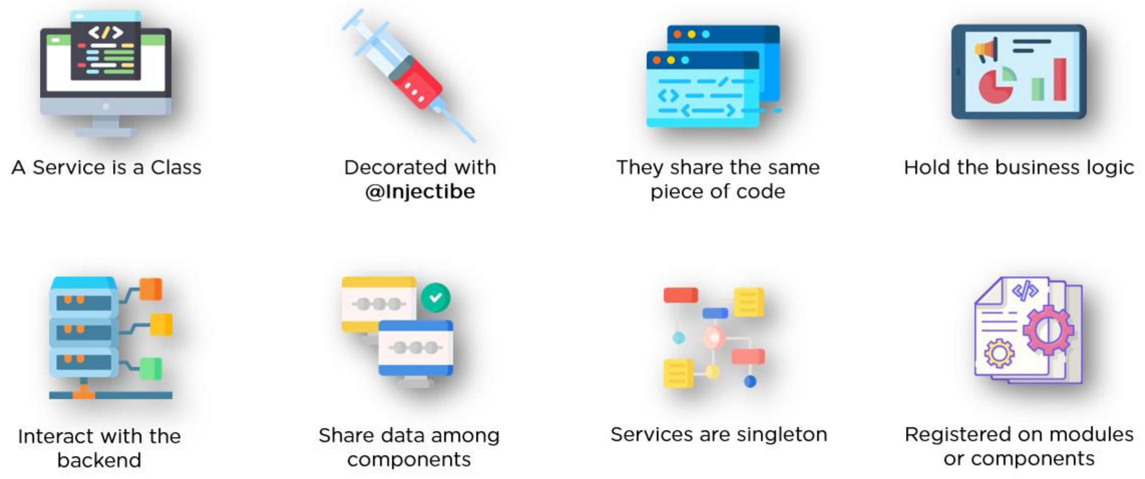
1. Understanding Services

In Angular, services are singleton objects that are used to organize and share code across the application. They are typically used to encapsulate reusable functionality that doesn't belong in a component, such as data fetching, logging, or authentication.



2. Purpose of Using Services

- **Code Organization**: Services help to keep your codebase clean and maintainable by separating concerns.
- **Reuse**: Services can be injected into multiple components, making it easy to reuse code.
- **Dependency Injection**: Services are typically injected into components using Angular's dependency injection system, which makes it easier to manage dependencies and test components in isolation.


A Service is a Class
Decorated with @Injectibe
They share the same piece of code
Hold the business logic
Interact with the backend
Share data among components
Services are singleton
Registered on modules or components

The main objective of a service is to organize and share business logic, models, or data and functions with different components of an Angular application. They are usually implemented through dependency injection.

3. Creating Services in Angular

To create a service in Angular, you can use the Angular CLI to generate a new service file. For example, to create a new data.service.ts file, you can use the following command:

```
ng generate service data
```

This will create a new file data.service.ts with a basic service skeleton. You can then add your service logic to this file.

Here’s an example of a simple service that provides a method to fetch data from an API:

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Injectable({
  providedIn: 'root'
})
export class DataService {
  constructor(private http: HttpClient) {}

  fetchData() {
    return this.http.get('https://api.example.com/data');
  }
}
```

In this example, the DataService class is defined with a method fetchData that uses Angular's HttpClient service to make an HTTP GET request to an API endpoint.

Using the Service in a Component

To use the DataService service in a component, you first need to inject it into the component's constructor. For example:

```
import { Component } from '@angular/core';
import { DataService } from './data.service';

@Component({
  selector: 'app-root',
  template: `
    <button (click)="fetchData()">Fetch Data</button>
    <div *ngIf="data">{{ data }}</div>
  `
})
export class AppComponent {
  data: any;

  constructor(private dataService: DataService) {}

  fetchData() {
    this.dataService.fetchData().subscribe((data) => {
      this.data = data;
    });
  }
}
```

In this example, the AppComponent injects the DataService service and uses it to fetch data from the API when a button is clicked. The fetched data is then displayed in the template using Angular's *ngIf directive.

This is a basic example of how services are used in Angular to encapsulate reusable functionality and share code across components.

Overview of Singleton Object

In software design, a singleton is a class that can have only one instance in the application, usually to control access to some shared resource such as a database connection or a service. In Angular, services are implemented as singletons by default, meaning that Angular creates a single instance of the service and shares it throughout the application.

5. Understanding Dependency Injection

Dependency injection (DI) is a design pattern used to achieve inversion of control (IoC) in software development. In Angular, DI is used to provide instances of dependencies (such as services) to classes that need them. This allows for better code organization, reusability, and testability.

6. Injectors and Providers

In Angular, an injector is responsible for creating and managing dependencies. It is the mechanism that Angular's DI system uses to provide instances of dependencies to classes. Providers are used to configure injectors with the information they need to create and manage dependencies. Providers can be registered at various levels in an Angular application, such as at the module level or the component level.

API stands for Application Programming Interface. It's a set of rules and protocols that allow software applications to communicate with each other. APIs are a key part of modern technology and business infrastructure.

How do APIs work?

- APIs act as an intermediary between two applications.
- When one application wants to access a resource from another application, it sends a request.
- The other application responds to the request by sending back the requested resource.

What are the benefits of APIs?

- APIs make it easier to develop applications by allowing developers to use data and functionality from other applications.
- APIs make it easier for application owners to share data and functions with other departments, business partners, or third parties.
- APIs help to keep systems secure by only sharing the information that is necessary.

Examples of APIs

- Restaurant APIs allow users to search for restaurants, query datasets, and display information.
- Database APIs allow applications to communicate with database management systems.

Understanding Asynchronous Programming

In the realm of modern web development, efficient handling of asynchronous tasks is paramount for responsive applications. Angular, a popular front-end framework, offers two powerful tools for this purpose: Promises and Observables. But before we dive into these concepts, let's understand why asynchronous programming is essential in Angular.

Web applications often require tasks like fetching data from APIs, processing inputs, and managing animations. These tasks should not block the main program flow, as it can lead to poor user experiences. Asynchronous programming allows tasks to operate concurrently, making apps dynamic and user-friendly.

Asynchronous programming lets tasks initiate and proceed independently, preventing slowdowns. Angular utilises this approach to enhance performance, responsiveness, and user satisfaction.

Observables: Embracing Reactive Programming

Observables are the heart of reactive programming in Angular, enabling efficient data flow and event handling.They provide a powerful way to manage asynchronous operations and respond to changes in data, user interactions, or any event-based scenarios.

Observables in Angular

Angular makes use of observables as an interface to handle a variety of common asynchronous operations. For example:

- The HTTP module uses observables to handle AJAX requests and responses
- The Router and Forms modules use observables to listen for and respond to user-input events In Angular, Observables are part of the Reactive Extensions for JavaScript (RxJS) library.
- · Observables provide support for passing messages between parts of your application.
- · Observables are a powerful feature used extensively in reactive programming to handle asynchronous operations and data streams.
- · Observables provide a way to subscribe to and receive notifications when new data or events are emitted, enabling you to react to changes in real-time.
- The basic usage of Observable in Angular is to create an instance to define a **subscriber function**. Whenever a consumer wants to execute the function the **subscribe()** method is called. This function defines how to obtain messages and values to be published.
- To make use of the observable, all you need to do is to begin by creating notifications using subscribe() method, and this is done by passing observer as discussed previously. The notifications are generally Javascript objects that handle all the received notifications. Also, the unsubscribe() method comes along with subscribing () method so that you can stop receiving notifications at any point in time.

HTTP

Angular's HttpClient returns observables from HTTP method calls. For instance, http.get('/api') returns an observable. This provides several advantages over promise-based HTTP APIs:

- Observables do not mutate the server response (as can occur through chained .then() calls on promises). Instead, you can use a series of operators to transform values as needed.
- HTTP requests are cancellable through the unsubscribe() method
- Requests can be configured to get progress event updates
- Failed requests can be retried easily

HTTP in Angular is a crucial component for making web requests to communicate with backend services, such as RESTful APIs. Angular provides the HttpClient module, which is a robust tool for making HTTP requests. This module offers methods like get , post , put , delete , etc., allowing developers to interact with remote servers by sending and receiving data. The HttpClient module simplifies the process of making HTTP calls and handling responses, including error handling, request cancellation, and the ability to transform response data.

```
import { HttpClient } from '@angular/common/http';

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

import { Observable } from 'rxjs';

@Injectable({ providedIn: 'root',})

export class DataService {

 private apiUrl = 'https://api.example.com/data';

constructor(private http: HttpClient) {}

getData(): Observable<any>

 { return this.http.get<any>(this.apiUrl);

}}
```

What are HTTP services in Angular, and how do they work?

HTTP services in **Angular** are used to make **HTTP requests** from a client-side application to a server. They are typically used to fetch data from an API or send data to a server, using **RESTful** web services. In Angular, we use the **HttpClient** module, which simplifies the process of making HTTP requests and handling responses. Angular's **HttpClient** service automatically handles the complexities of making network requests, like handling headers, sending parameters, and dealing with different request types.

When I use **HttpClient**, I can make GET, POST, PUT, DELETE requests, and more. For example, to fetch data from an API, I can use the **http.get()** method, which returns an **Observable**. I'll then subscribe to this Observable to receive the response from the server. This makes it easy to handle asynchronous operations, such as waiting for data to arrive from an API. Angular also provides easy ways to deal with errors, transform responses, and retry failed requests.

What is an Observable in Angular, and why is it used with HTTP?

In **Angular,** an **Observable** is a way to handle asynchronous operations. It's part of **RxJS (Reactive Extensions for JavaScript)**, which is a powerful library for handling streams of data. When I make an HTTP request, the response isn't immediate. That's where Observables come in handy, as they allow me to "subscribe" to the response and act upon it once it arrives. With Observables, I can also handle multiple values over time, making it a flexible tool for **asynchronous** programming.

The reason **Angular** uses Observables for HTTP is that it allows for greater flexibility. With an Observable, I can manage multiple values, cancel requests, retry failed requests, and chain operations, all with simple, clean syntax. I can also transform the data received using operators like **map() , filter() ,** and **tap()** , which are part of **RxJS**.

How do you create an HTTP request in Angular using HttpClient?

To create an **HTTP request** in **Angular**, I first need to import the **HttpClientModule** in my **app.module.ts** file and include it in the imports array. Once that's done, I can inject the **HttpClient** service into my components or services where I need to make HTTP requests. The **HttpClient** service provides methods like **get() , post() , put() , delete()** to interact with a RESTful API.

Here's an example of how I make a **GET** request:

```
import { HttpClient } from '@angular/common/http';


constructor(private http: HttpClient) {}


getData() {
  this.http.get('https://api.example.com/data').subscribe(response => {
    console.log(response);
  });
}
```

How can you handle errors in HTTP requests using Observables?

Handling errors in **HTTP requests** using **Observables** is a crucial part of building robust applications. When making an HTTP request with Angular's **HttpClient** , errors can occur for various reasons, such as network failures or invalid responses from the server. The good thing about **Observables** is that they provide built-in mechanisms to handle these errors gracefully.

I can catch and manage errors by using **RxJS operators** like **catchError()** . This allows me to intercept the error and decide how to handle it. Here's an example of handling errors in a **GET** request:

```
getData() {
  this.http.get('https://api.example.com/data').pipe(
    catchError((error) => {
      console.error('Error occurred:', error);
      return throwError(error);
    })
  ).subscribe((data) => {
    console.log('Data:', data);
  });
}
```

How do you cancel an HTTP request in Angular with an Observable?

Canceling an **HTTP request** in Angular is possible when using **Observables**. One of the powerful features of Observables is their ability to be canceled by unsubscribing. This is particularly useful in scenarios where I have multiple HTTP requests happening simultaneously, and I want to cancel one or more of them to save resources or avoid unwanted updates.

To cancel a request, I need to create a **Subscription** to the Observable and then call **unsubscribe()** when I no longer need the data. Here's an example:

```
let subscription = this.http.get('https://api.example.com/data').subscribe(
  (data) => console.log(data)
);


// Later in the code, cancel the request
subscription.unsubscribe();
```

note:use provideHttpClient() in app.config.ts

```
export const appConfig: ApplicationConfig = {
  providers: [provideRouter(routes),provideHttpClient()]
};
```