Learning Consolidation

Build Reusable Application Logic Using Angular Services

In this Sprint, you learned to...

- Explain the need for creating an Angular service to perform application tasks
- Create an Angular service to perform a synchronous task
- Consume services in Angular components
- Make asynchronous calls to the server using HttpClient service and Observable
- Utilize component lifecycle method `ngOnInit()` to initialize component properties.
- Create and consume an Angular service to make the server calls
- Handle HTTP error response in components

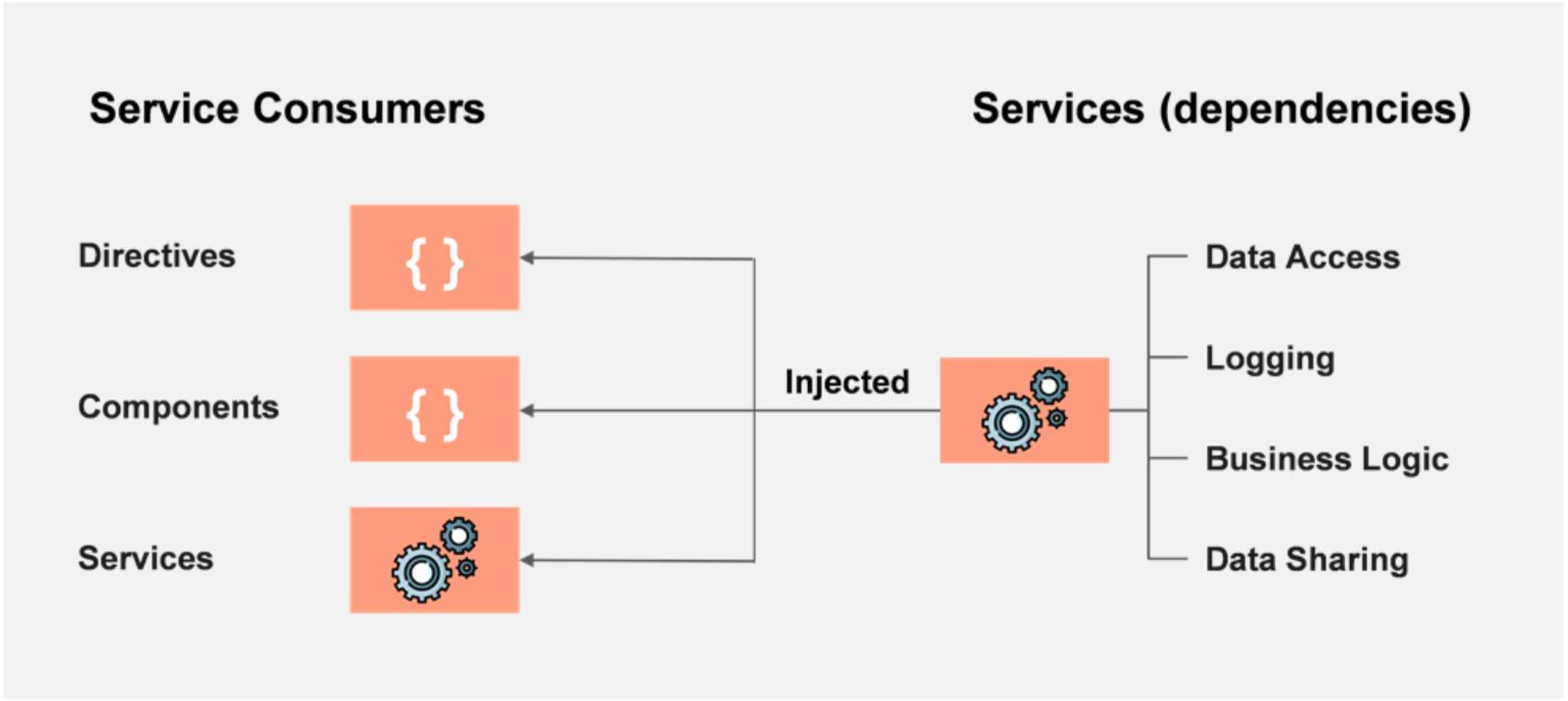
Slide Note

Services are made responsible for carrying out various application tasks.

These services are injected into the Directives, Components, or Services who are said to be the consumer of the services.

- *By defining such processing tasks in* ***an injectable service class,*** *the tasks can be made available to any component.*

In Angular, Services Help Perform Additional Tasks



Slide Note

Working with Angular Service

Menu

00:04 00:00:32 04/ 32

Slide Note

Step 1: Providing Dependency

- Services are dependencies for the components.
- These services need to be defined as injectable and should be provided to make them available to consumers (components).
- To create a new Angular service, run the Angular CLI command.

  ```
  ng generate service <service-name> -or- ng g s <service-name>
  ```

- The command creates a class with the `@Injectable` decorator.
- This injectable class is made available through the metadata `providedIn`.
- The value `root` indicates the service is visible throughout the application and is therefore available to all the components.

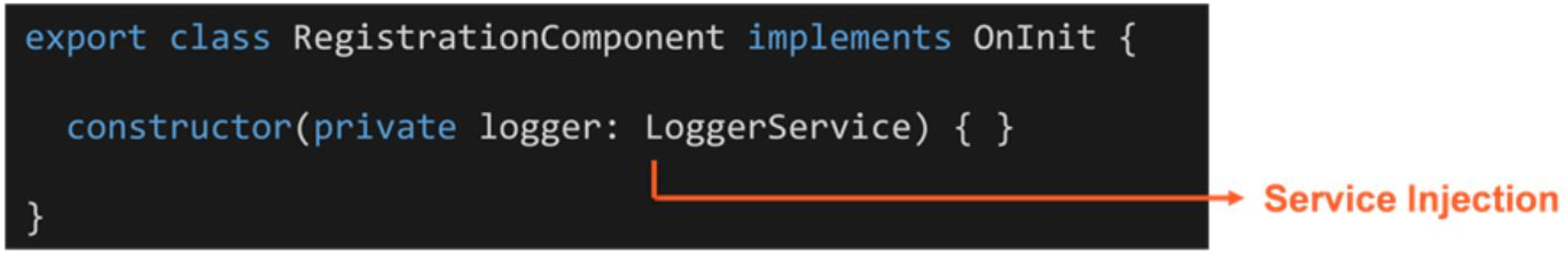
```
@Injectable({
  providedIn: 'root'
})
export class LoggerService {
  constructor() { }
  logMessage(message: string) {
    console.log(message);
  }
}
```

Menu

Step 2: Injecting Service (Dependency)

Service is a class, so did the Angular component create an instance using the 'new' keyword?

- The service was injected into the component through dependency injection.
- Angular uses a constructor injection mechanism for injecting the services.
- **Components consume services:** You can inject a service into a component through its constructor, giving the component access to that service class.



```
export class RegistrationComponent implements OnInit {

  constructor(private logger: LoggerService) { }

}
```

Slide Note

How Many Instances of Dependencies Get Created?

- Angular Framework uses the Singleton pattern for creating a service (dependency) and injects it into different Components through Constructor Injection.
- It means Angular creates and provides a single service instance that is shared across applications.

What if services are stateful and two different modules or components want to maintain separate instances for individual states?

Is it possible to get service instances to be provided at the module level or component level?

Slide Note

Providing Services

To The Application

Register the provider in the service metadata in the `@Injectable()` decorator.
The single instance of service is shared by any application class that asks for it.

To The NgModule

Register the provider using the providers' property of the `@NgModule()` decorator.
The same service instance is available to all the module components.

To The Component

Register the provider in the providers' property of the `@Component()` metadata.
A new instance of the service is obtained with each new instance of the component.

Menu

Slide Note

Try relating the questions with the Logging service as the dependency.

A logging service instance is not created using the new operator, so then who created the instance?

How did this person know which out of all dependencies (say there can be Logging as one dependency, but others as well)?

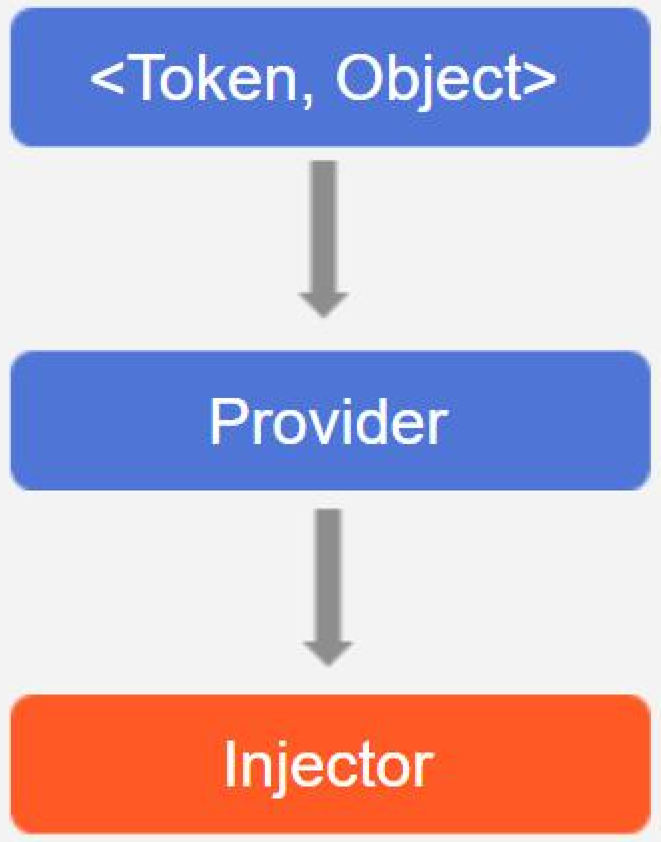
So, who provides it with this information about dependencies?

When the consumer requests the dependency, how is it located and provided?

How many instances of dependencies get created?

How Does Dependency Injection Work?

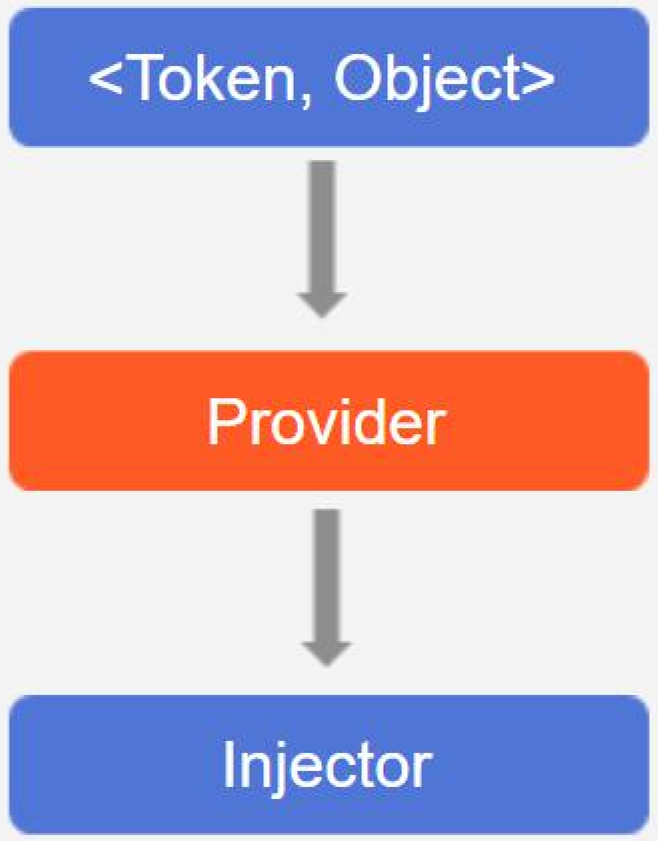
- Who creates dependency?
- How is the dependency created?
- How is the dependency located?



Who Creates a Dependency?

- The **injector** is the main mechanism for implementing DI.
- An injector creates dependencies and maintains the container of reusable dependencies.
- The injector uses a provider to create new instances for the dependencies.
- Angular creates injectors while bootstrapping the application; developers do not create them.

Slide Note



How Is Dependency Created?

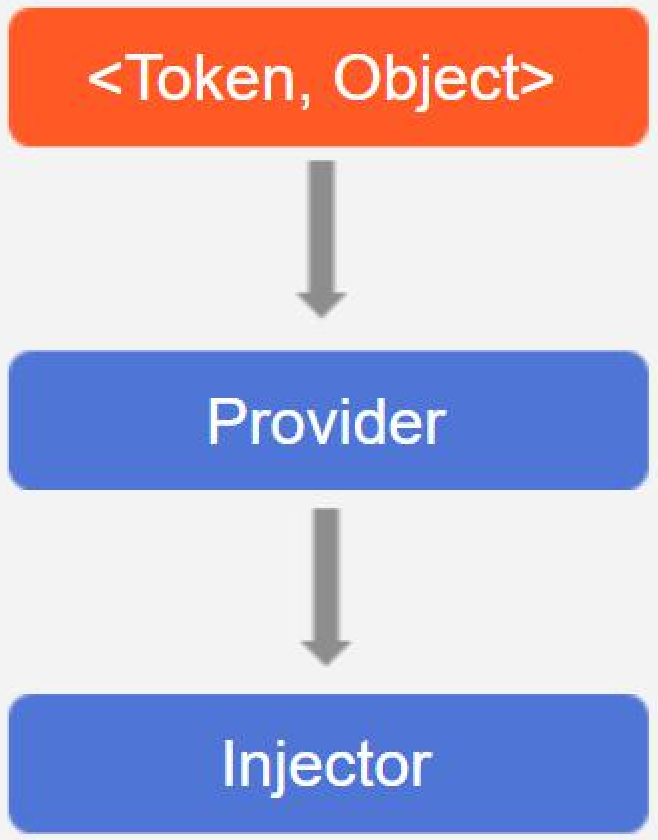
- A **provider** is an object that tells an injector how to obtain or create a dependency.
- By default, for a service, the service class itself is a provider.
- The provider can be explicitly declared in the provider's metadata.
- For example,

providers :[{ provide: LoggerService, useClass: LoggerService }]

- The syntax has two properties:
  - provide (`provide: LoggerService`)
  - provider (`useClass: LoggerService`)

Menu 00:11 00:00:32 11/ 32

Slide Note



How Is Dependency Located?

- While configuring an injector with a provider, the provider gets associated with a dependency injection token.
- Angular creates a map of dependencies with the help of an injector, and the DI token is the key to the map.
- In the syntax,

providers :[{ provide: LoggerService, useClass: LoggerService }]

The first property is `provided` that holds the DI Token

- The DI systems need the key to locate the provider in the provider's array.

Menu

00:12 00:00:32 12/ 32

Slide Note

The JavaScript Promise object allows associating handlers that get invoked when the asynchronous task completes (successfully or as a failure)

The concept was discussed in Sprint 6 of Course 3.

Service Method Calls

- Calls to service methods can be both synchronous as well as asynchronous.
- If the service logs data on console, then it is a synchronous task.
- However, if the same data is logged on to a database, the execution will be asynchronous since it will be an external interaction with a remote server.
- The frontend web application frequently makes HTTP calls to REST APIs for interacting with the backend server.
- These asynchronous calls can be made with the help of
    - JavaScript Promise object
        - This object represents the eventual completion of an asynchronous operation.
    - Observables
        - It's an RxJS type that produces values asynchronously only when requested.

Slide Note

Making Asynchronous Calls Using RxJS Observable

Menu

00:14 00:00:32 14/ 32

What Is RxJS?

- RxJS is a Reactive Extensions library for JavaScript.
- It is a library for composing asynchronous and event-based programs by using observable sequences.
- An Observable is a Producer of multiple values, "pushing" them to Observers (Consumers).
- It provides the core type, **Observable.**

Slide Note

In a restaurant, a menu is observed and orders are placed.

The invoice needs to account for changes in the cart. The invoice must also be updated as the cart changes.

This manager is the Observable.

Think and Tell

- Who is the connecting link between the kitchen and the order in the restaurant?
- If a cart during online shopping is updated, how does the invoice know about it and get updated accordingly?
- Is there an entity between producer and consumer that manages the communication?

Slide Note

Observable Simplified

- Observables are lazy push collections of multiple values.
    - Lazy push is the term used to indicate that the push operation is not performed immediately (only when requested).
- They are like functions that execute when called.
    - Remember, in a restaurant, the food item is not delivered until the order is placed.
- They are the producers who produce the values only when the consumer subscribes to it.
    - The chef prepares food only when the order is placed.
- Observables can deliver values synchronously as well as asynchronously.
    - Drinks that are ready are delivered just in time.
    - Food items take time to get prepared and are delivered asynchronously.
- They can return zero or an infinite number of values.

Menu

Slide Note

What is an Observable?

- In an application, often, there are background processes that execute to produce data.
  - For example, fetching data from the server.
- The data can be produced synchronously as well as asynchronously.
- In JavaScript, functions are defined that contain code to produce a single value synchronously.
- The caller (consumer) calls the function and pulls the value produced by the function (producer).
- However, a more complex code is required that produces multiple values synchronously or even asynchronously for the consumers who need not be aware when exactly the values are created.
- The consumer observes the producer and will receive the value when pushed to it.
- For example, fetching data from the server.
  - The JavaScript library RxJS provides a class Observable that produces multiple values.
- The consumer of these values has to subscribe to Observable to receive the values when produced by the Observable.

Slide Note

Subscribing to Observables

- Observable produces the values synchronously or asynchronously.
- The consumer must subscribe to the observable to consume these values.
- To subscribe, the consumer must call the `subscribe()` method on the observable.
- The `subscribe()` method accepts an object known as an `observer` to receive the notifications.
- The observer object has 3 properties for 3 different types of notifications:
  - `next`: this is required and is associated with a handler that gets called for each value delivered.
  - `error`: this is optional and is related to a handler for handling errors.
  - `complete`: this is optional and is associated with a handler that notifies the completion of execution.

Menu

Slide Note

<> notation is defined in TypeScript language for Generics.

Generics allow the creation of functions and classes that can work over a variety of types.

Observable is a Generic type that produces values of a variety of types.

While working with Generic type, the specific type of data that needs to be passed or returned should be specified in <>.

While working with the HttpClient service object, the type needs to be specified while calling the HTTP methods so that the response received is converted to the type specified.

HTTP Calls With HttpClient

- Angular provides the built-in service `HttpClient` in `@angular/common/http` that helps make HTTP calls.
- `HttpClient` is a service class in `HttpClientModule`.
- The `HttpClient` contains methods that help make GET, POST, PUT, and DELETE HTTP requests to the application server.
- Each of these methods returns an Observable of type `HttpResponse` and not just the JSON data contained in the body.
- The observables can be subscribed to receive the responses, which could be the expected data or an error.

Inside Service class:

```
getFruits(): Observable<Array<Fruit>> {
    return this.http.get<Array<Fruit>>(this.URL);
}
```

Making GET request

The HTTP response is transformed to type Array of type Fruit

Inside Component class:

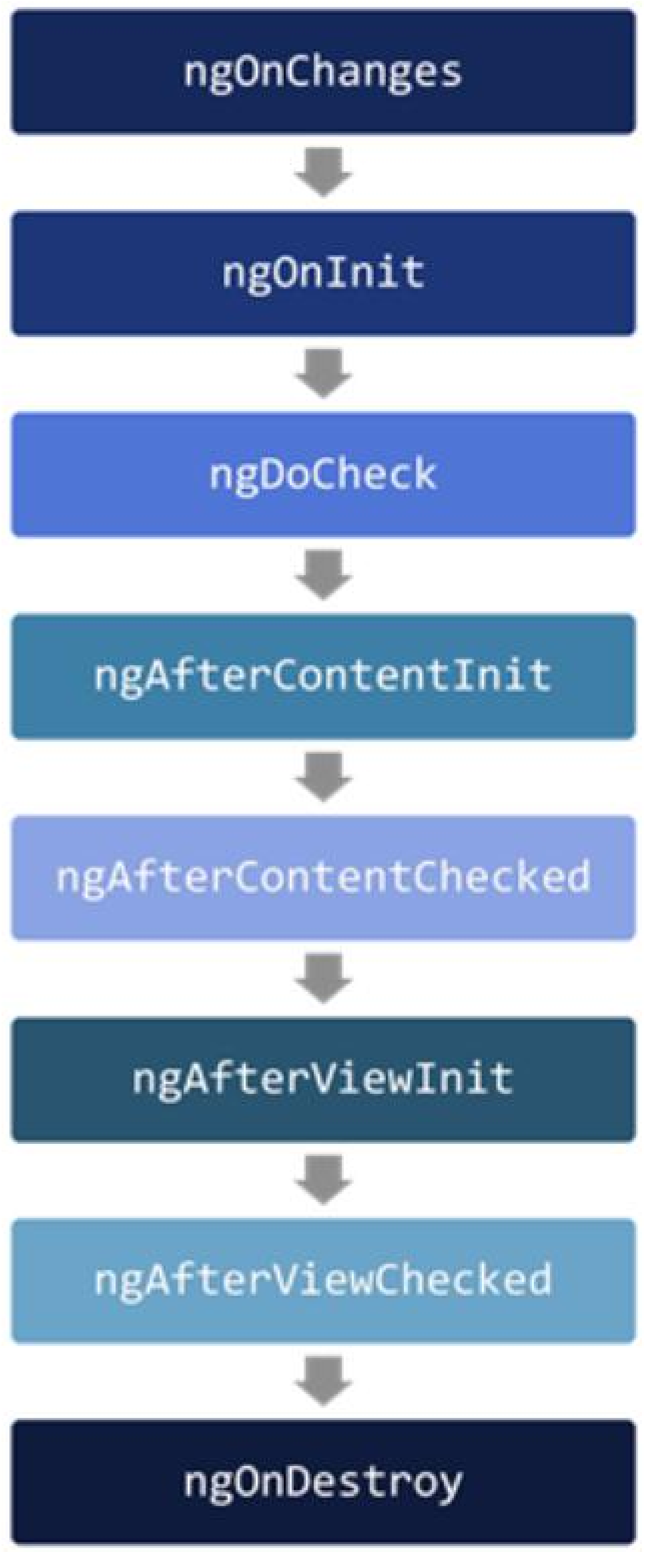
```
this.fruitService.getFruits().subscribe(data => {
      this.fruits = data;
});
```

Calling Service method and subscribing to Observable returned

Slide Note

The detailed discussion on lifecycle methods is beyond the scope of this course.

Component Life Cycle



- A lifecycle of an Angular component defines the stages it goes through – from initialization to destruction.
- There are 8 different lifecycle stages.
- Each stage is a lifecycle hook event.
- Angular applications can use lifecycle hook methods to tap into lifecycle events.
- After executing the constructor, Angular executes its lifecycle hook methods in the order shown in the image.

Slide Note

```
interface IMyInterface {
    getData(): Array<string>;
    postData(value: string): string;
    updateData(id: number, value: string): boolean;
    deleteData(id: number): boolean;
}

class MyClass implements IMyInterface {
    getData(): string[] {
        console.log("fetching data from server");
        return ["some","data"];
    }
    postData(value: string): string {
        console.log("posting data to server");
        return "value";
    }
    updateData(id: number, value: string): boolean {
        console.log("updating data at server");
        return true;
    }
    deleteData(id: number): boolean {
        console.log("deleting data from server");
        return true;
    }
}
```

How to Work With Component Life Cycle Methods?

- In Angular, each lifecycle hook method is associated with an interface provided by the Angular core library.
    - The interface is like a contract that specifies which methods need to be defined if the class implements the interface.
- Each interface has exactly one lifecycle hook method.
- The name of the interface is the same without the ng prefix.
    - For example, the `OnInit` interface has a hook method named `ngOnInit()`.

Slide Note

Implementing `ngOnInit()` to Fetch Data

- Constructors should only handle the responsibility of initializing local variables with simple values.
- Code to fetch data should not be written in a component constructor.
  - To ensure the safe construction of a component, the constructor, while constructing a component, should not contact a remote server.
  - The failure to connect with the server can hamper the component construction.
- In Angular, each lifecycle hook method is associated with an interface provided by the Angular core library.
- Each interface has exactly one lifecycle hook method.
- The lifecycle hook, `ngOnInit()`, is an excellent place to perform complex initialization tasks such as fetching initial data from the server.

Slide Note

Error Handling With HttpClient

- The `HttpClient` object allows making API calls using `get()`, `post()`, `put()` and `delete()` meth ods to perform popular CRUD operations on data.
- The request sent by calling these methods may responded with an error.
- Errors resulting from observable execution should be handled using the error property of the observer in the `subscribe()` method.
- Providing error handling code is crucial as it helps notify the user about the request failure (preferably with the reason of failure).

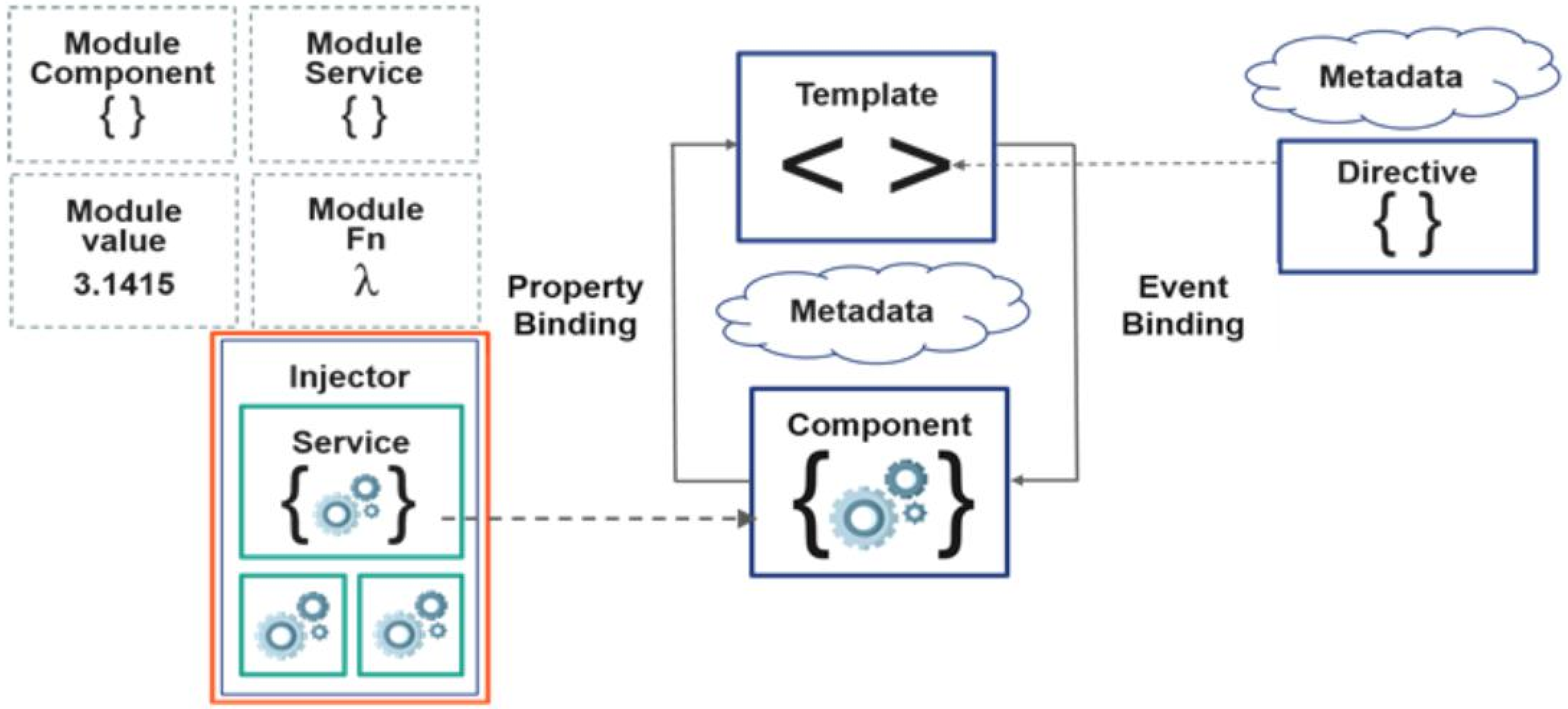
```
ngOnInit(): void {
   this.fruitService.getFruits().subscribe({
     next: data => {
       this.fruits = data;
     },
     error: e => {
       alert("Network Error !! Please Try
          Again Later");
     }
   });
 }
```

Menu

00:24 00:00:32 24/ 32

Role of Angular Service in Angular Application

- Angular helps follow design principles by making it easy to factor your application logic into services.
- Services are reusable and can be injected into multiple components.
- A Service is a class in Angular that is created with a well-defined purpose. It is the piece of code or logic that is used to perform some specific task.



Slide Note

Self-Check

An Angular application cannot exist without services.

True / False

Menu

00:26 00:00:32 26/ 32

Slide Note

Self-Check: Solution

An Angular application cannot exist without services.

True / False

False

Service is not mandatory with an Angular application.
However, Angular makes it easy to factor the application logic into services that helps you follow design principles. These services are made available to components through dependency injection.

Slide Note

Self-Check

What type of data is returned by the `put()` method of Angular's `HttpClient` class?

1. Promise
2. void
3. Observable
4. Array

Menu

Slide Note

Self-Check: Solution

What type of data is returned by the `put()` method of Angular's `HttpClient` class?

1. Promise
2. void
3. **Observable**
4. Array

Menu

Slide Note

Self-Check

Which property of the observer object of `subscribe()` method is used to implement error handling?

1. `onError`
2. `onException`
3. `error`
4. `exception`

Menu

Slide Note

Self-Check: Solution

Which property of the observer object of `subscribe()` method is used to implement error handling?

1. `onError`
2. `onException`
3. **`error`**
4. `exception`

Menu