Slide Note

A Couple of Scenarios that Refer to Different Kinds of Services Consumed

Slide Note

Is the waiter serving on only one table or at multiple tables?

Is the waiter doing the cooking, cleaning, and billing job as well?

Different Modus Operandi Followed by the Restaurants

Self-Service

Waiter Service

Slide Note

Responses suggested for the above questions:

How does the experience vary between visiting a restaurant with self-service counters and a fine-dining restaurant with waiter service?

With self-service counters, the customers have to themselves search for tables, place order at the counter, make payments at the payment counter, get token, wait for the order to be ready, collect the order and carry it to the table.

With waiter service, all these requirements are handled by the waiter.

What would happen if the waiters in the fine-dining restaurants also handled the cooking and bill payments?

This would keep the waiters more engaged and they would not be able to dedicatedly serve the customers, which can lead to poor experience for the customers.

Would the waiter be serving on a single table or multiple tables?

Yes, a waiter can serve multiple tables and help multiple customers to enjoy their meals at the restaurant

Think and Tell

- How does the experience vary between visiting a restaurant with self-service counters and a fine-dining restaurant with waiter service?
- What would happen if the waiters in the fine-dining restaurants also handled the cooking and bill payments?
- Would the waiter be serving at a single table or multiple tables?

Slide Note

Here is another popular service scenario...

Menu

00:04:00:00:42

04/ 42

Slide Note

Responses suggested for the above questions:

How do organizations help employees reach the workplace on time?

The organizations provide a bus service to its employees for easing out commutation hassles.

Would the same bus for company employees pick and drop school students on the way?

No, since employees and students have different timings to adhere

Also, mixing up different types of commuters in one bus would be a messy affair.

Can the same service be consumed by employees working in different shifts?

Yes, this way the resources can be well utilized and would be an economical option.

Pick and Drop Service

- How do organizations help employees reach the workplace on time?
- Would the same bus be used by company employees to pick up and drop off school students on the way?
- Can the same service be consumed by employees working different shifts?

Slide Note

Responses suggested for the above questions:

Do these services handle a single responsibility or multiple responsibilities?

Service in particular handles a single responsibility

For example,

A the restaurant, a waiter only handles the requests of its customers.

The bus service provided by organization only helps the organization's employees with commutation

Are these services being consumed by a single consumer or multiple consumers?

Services can be consumed by multiple consumers

For example,

Restaurant waiter can serve at multiple tables

Bus service can help employees working in different shifts to commute.

Do these services help in building a better ecosystem?

Yes, definitely.

Makes the resource consumption more optimized and hence economical, without leading to any mess.

Think and Tell

- Do these services handle a single responsibility or multiple responsibilities?
- Are these services being consumed by a single consumer or by multiple consumers?
- Do these services help build a better ecosystem?

Slide Note

Build Reusable Application Logic Using Angular Services

Slide Note

Learning Objectives

- 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 Http Client service and Observable
- Utilize the component lifecycle method `ngOnInit()` to initialize component properties.
- Create and consume an Angular service to make server calls
- Handle HTTP error response in components

Slide Note

The additional responsibilities are background tasks and are not visible to the end user.

These additional responsibilities could be

Logging

Allows all external and internal transactions / activities to be logged on a console, file or even database

Authentication (security)

Allows ensuring access to requested parts of applications only to valid users.

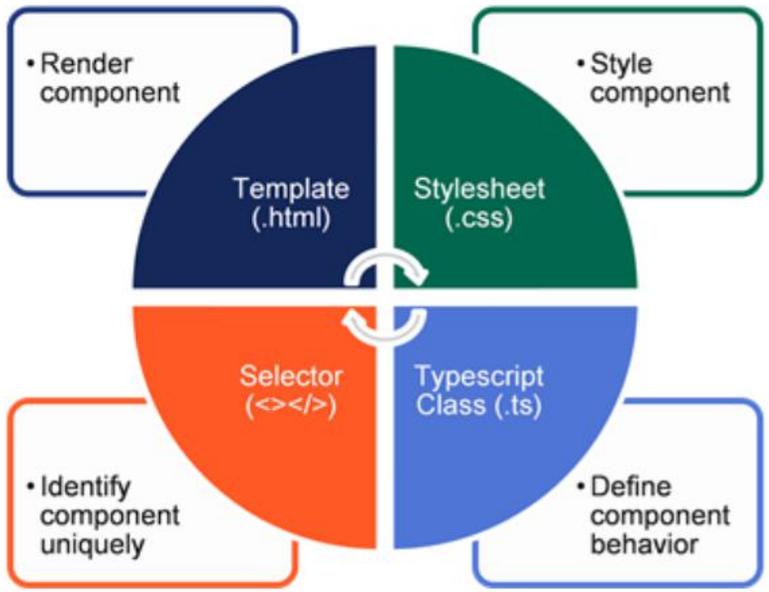
Server interactions

Allows making requests and receiving responses to server applications running remotely.

Responsibility of an Angular Component in SPA

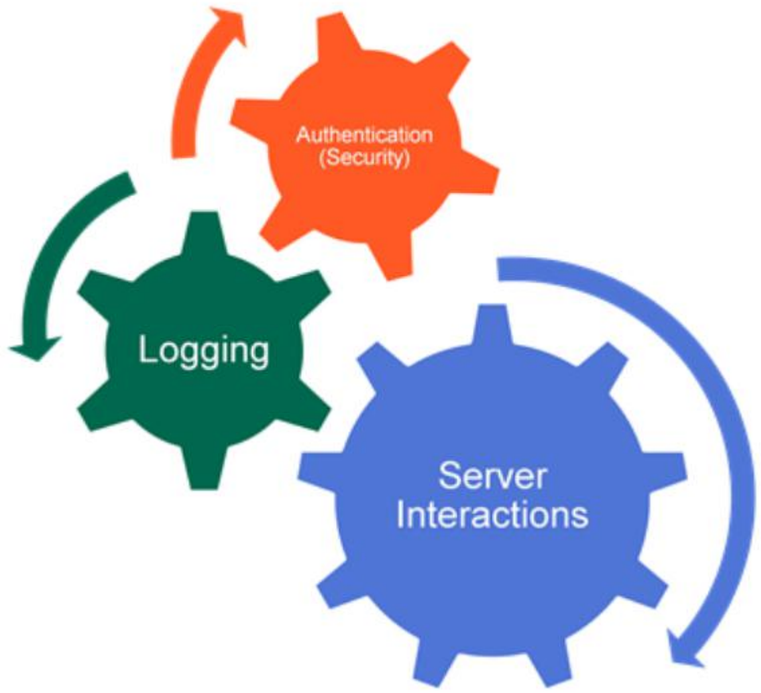
Primary Responsibility

- Manage View(s) of SPA



Additional Responsibilities

- Manage Background Tasks



Slide Note

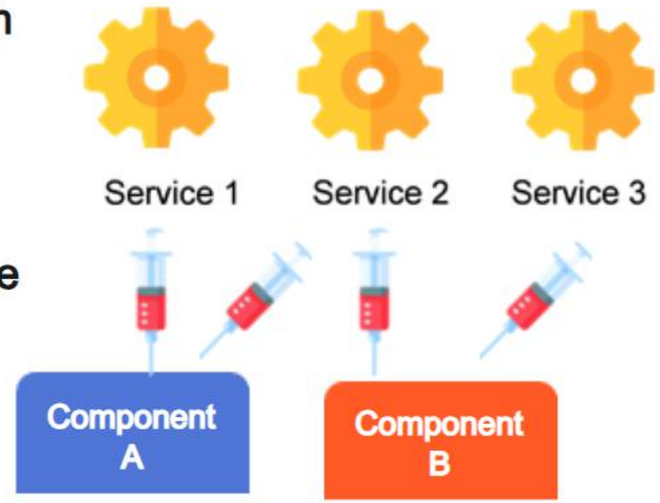
Think and Tell

- Where should the application logic for these additional tasks reside?
- Should it be the responsibility of the component to handle these tasks?
- If so, then the Single Responsibility Principle – SRP is violated.
  - The responsibility of the component is to manage views.
- Would this application logic be reusable by other components if it resided within them?
- Would it not be a better approach to provide the application logic in a separate class and get it consumed by the component class?

Slide Note

Handling Additional Responsibilities – A Better Approach

- Additional tasks such as logging, authentication, and interactions with the external server can be defined as a separate class.
- In Angular, this class is known as a Service.
- Just as medicines are injected in human body, services in Angular are injected into Angular components.
- This mechanism enables adherence to:
  - DRY principle
    - The code is not tightly coupled with the component.
    - The service can be injected into multiple components, making it reusable.
  - SRP principle
    - Component continues to handle its primary responsibility.
    - For different application tasks, different services can be designed.



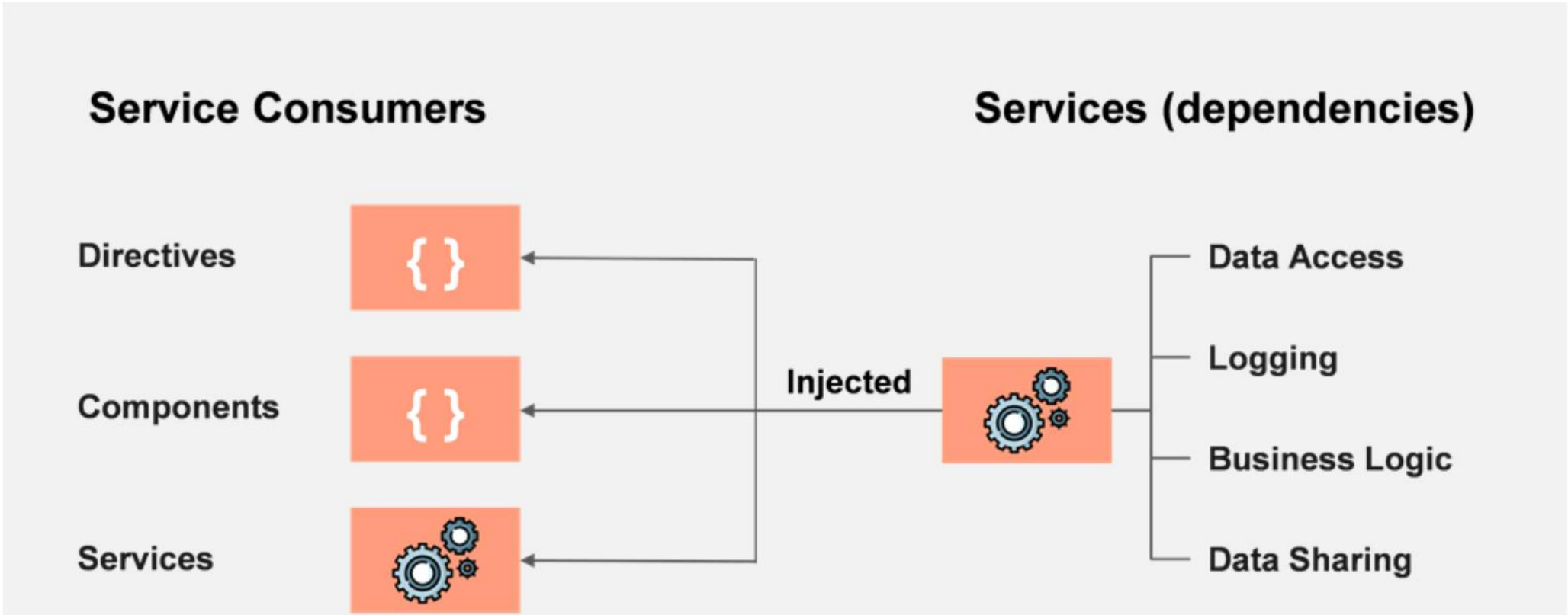
Slide Note

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

Logging Scenario

- How can a developer monitor the time taken for a component to load?
  - The app should record the time when the request is made and the time when the component is loaded and calculate the difference.
- How can a developer get to know number of times registration has failed as well as the reasons of failures using an app's registration component?
  - The app should record the failed transactions with reasons
- These details help developer, in general, to monitor the performance of app as well as track user activities.
- The task of recording the details by the app, while the app is known as logging.
- Logs can be written to console, a file or a database.

Slide Note

Logging Implementation Considerations

- Let's take up a scenario, wherein, the developer of a shopping cart application wants to create a ‘registration’ component that accepts registration details such as username, email, and contact number and sends them to a server for storing them.
- During the development process, the developer wants to log the messages, warnings, or errors onto the console to ensure the correctness of the application logic written.
- In the future, the logging functionality can also be implemented in other components such as login component, cart component, payment component. etc.
- Therefore, the application code for logging must be reusable.
  - *A code is reusable if it resides as a separate unit of code.*
- A separate logging class can be defined that handles the logging activity.
- The ‘registration’ component will consume the ‘logging’ class. *It makes the registration component dependent on the‘logging’ class.*
  - In other words, the 'logging' class is a dependency for the registration component

Slide Note

Implement Logging Using Angular Service

Create a registration component that logs the important information captured during user registration.

The Angular service should handle the responsibility of logging functionality.

Click here for the demo solution.

DEMO

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);
  }
}
```

Slide Note

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) { }

}
```

Service Injection

Slide Note

Quick Check

A service class is decorated with the _____ decorator.

1. `@Service`
2. `@Inject`
3. `@Injectable`
4. `@Serviceable`

Slide Note

Quick Check: Solution

A service class is decorated with the _____ decorator.

1. `@Service`
2. `@Inject`
3. `@Injectable`
4. `@Serviceable`

Slide Note

Think and Tell

- Were the calls to Logging service synchronous or asynchronous?
  - Answer: Synchronous
- If the logging service, writes the logs to a file or database, would it be synchronous or asynchronous
  - Answer: asynchronous, as the response can be delayed.
- How does an Angular app handle asynchronous data communication?

Slide Note

Between frontend and server, data communication is frequently done

Slide Note

Data Communication

- Using Function
  - Function, as the Producer, produces a single value at a time synchronously.
  - A value produced by the function is consumed by the calling program who is termed as a Consumer.
  - Function produces the value only when it is called. (lazy approach)
- Using Promise
  - Promise produces single value when resolved successfully and pushes the value(s) to the registered callbacks.
  - Promise is the Producer and registered callbacks are the Consumers.
  - Promise produces the value whenever the data is available and pushes it to the consumers.
  - Consumer is unaware when the data will be available.
- Angular uses Observable for data communication

Slide Note

Let's say there is a function 'fn' of object `objA` that returns a value.

Here, objA.fn() means give me one value synchronously.

Now let's say there is an Observable `obs`

Here, obs.subscribe() means the consumer is ready to receive a number of values produced synchronously or asynchronously.

They are the producers who produce the values only when the consumer subscribes to them.

The chef prepares food only when the order is placed.

Observables can deliver values synchronously as well as asynchronously:

In a restaurant, soft drinks are ready and are delivered just in time.

However, pizzas take time to be prepared and so are delivered asynchronously.

What is an Observable?

- Observable is a function that produces values.
  - Hence, it is a producer.
  - Values are produced synchronously as well as asynchronously.
- The Consumers of Observable need to subscribe to Observable to consume the values produced.
  - Unless subscribed, values are not produced.

Slide Note

Why Observable?

- Like function, Observable is lazy.
  - Observable is declarative since it produces values only when subscribed (called).
    - *Subscribing to an Observable is analogous to calling a Function.*
  - However, unlike function, it can produce zero to infinite values over time.
- Like the consumers of Promise, consumers of Observable would not be aware when the data would be produced.
  - Once subscribed, the control making data available lies with the Observable.
  - The subscriber or consumer gets the notification once the values are ready.
  - The consumer can even unsubscribe from the observable when it wants to stop receiving the value. (The feature is not possible with Promise).
- Based on the above listed features, Observable is frequently used in Angular primarily for asynchronous operations, such as:
  - Event handling, making server calls and handling multiple values.

Slide Note

Asynchronous Communication in Angular SPA

- The components or services in an Angular often need to interact with the server for sending or fetching data.
  - In other words, Angular components or services are the consumers of data or response provided by the server.
  - An HTTP request (GET / POST / PUT or DELETE) needs to be made to the server to communicate asynchronously
- Is there an API that can make the request to the server and provide the necessary data to the Angular components or services?

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.
- The HttpClient service class belongs to 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

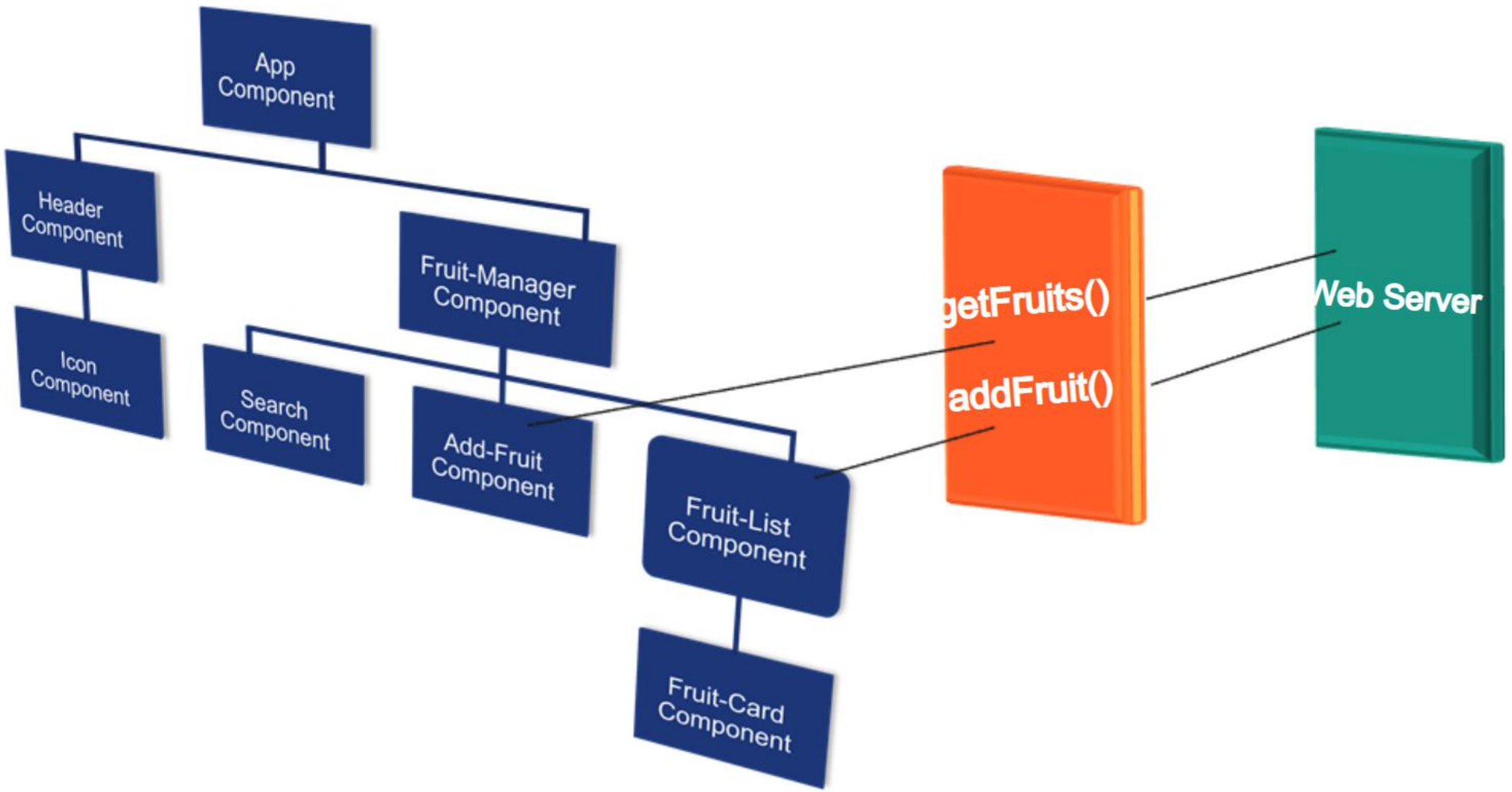
Inside Component class:

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

Calling Service method and subscribing to Observable returned

Slide Note

Implementing Persistence in Fruit-Fantasy App



Slide Note

Fruit-Fantasy – Add Fruit

Create a service that makes an HTTP call to 'fruits' API launched using `json-server` to add fruit data.

The service should be consumed by the component that accepts input from the user and sends it to the server to store the data.

Click here for the demo solution.

DEMO

Slide Note

Subscribing to Observables

- Observables produce 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.

Slide Note

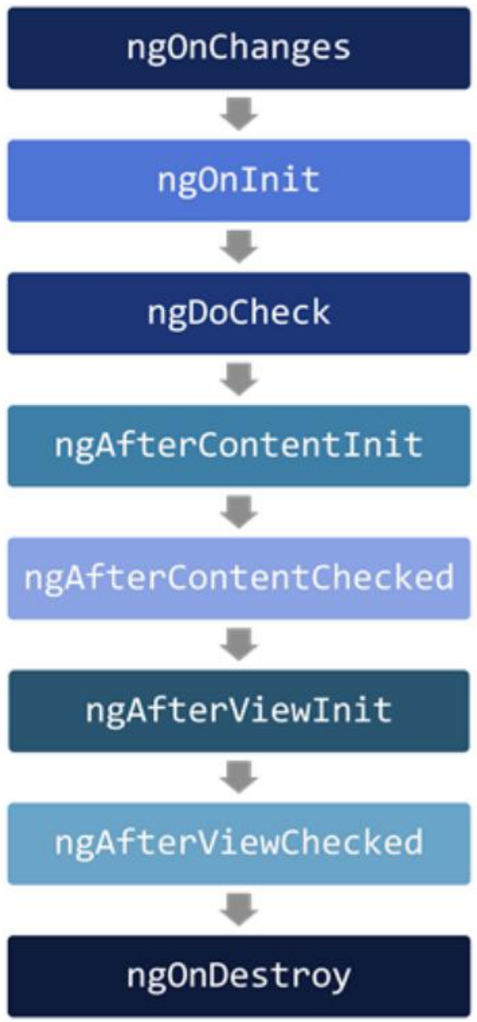
Think and Tell

- Posting data to the server can be done with a click of a button when the user has provided the inputs.
- But when should the data from the server be read?
- No user interaction is required to fetch data. It should be done immediately after the component creation is completed.
- The data needs to be fetched after the component instance is created and then passed to the template for presentation.
- Which method will get invoked once the component is created?

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

The code on the slide is a sample code to show how an interface is defined and implemented by a class.

Note: This demo is not related to the ongoing Angular discussions on Services and LifeCycle methods.

```
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

Fetching Data From Server

- 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.
- A `ngOnInit()` is an excellent place to perform complex initialization tasks such as fetching initial data from the server.
- This lifecycle hook event is triggered once, after the first `ngOnChanges()` is called.

Slide Note

Fruit-Fantasy – Fetch Fruits

Create a service that makes an HTTP call to 'fruits' API launched using `json-server` to fetch the fruit data.

The component should consume the service, which then displays the fruits on the view.

Click here for the demo solution.

DEMO

Slide Note

Quick Check

What is the return type from the following call to the `get()` method of `HttpClient`, if the request is successfully processed by `json-server`?

```
this.httpClient.get('http://localhost:9090/customers' )
```

1. Observable with Object as data
2. Observable with an Array of objects as data
3. Observable with an Array of Customer objects as data
4. Observable with Customer object as data

Slide Note

Quick Check: Solution

What is the return type from the following call to the `get()` method of `HttpClient`, if the request is successfully processed by `json-server`?

```
this.httpClient.get('http://localhost:9090/customers' )
```

1. Observable with Object as data
2. Observable with an Array of objects as data
3. Observable with an Array of Customer objects as data
4. Observable with Customer object as data

Slide Note

Time to Think

What would you do if:

- The server serving the fruit data is down?
- The application runs into an unexpected error?

Slide Note

Error Handling With HttpClient

- The `HttpClient` object allows making API calls using `get()`, `post()`, `put()` and `delete()` methods 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");
     }
   });
}
```

Slide Note

Fruit-Fantasy – Handle Error Response

While making API requests, the response need not always be as expected. The answer may return an error message due to network unavailability or an incorrect request.

Modify the frontend code that handles the error response and raises an alert to notify the user about the same.

Click here for the demo solution.

DEMO

Slide Note

Quick Check

Which property of the observer object is used to handle errors?

1. next
2. data
3. error
4. complete

Slide Note

Quick Check: Solution

Which property of the observer object is used to handle errors?

1. next
2. data
3. **error**
4. complete