MEAN Stack:

Angular

Agenda

- Nested/Child Component
- Component Communication
  - Passing data to child component
  - Passing data to parent component
- Component Lifecycle
- Dependency Injection
  - Introduction of Dependency Injection(DI)
  - Benefits of DI
  - Angular Dependency Framework
- Service in Angular
  - What is Service?
  - Angular Providers
  - Angular Injectors
- Interacting with server using HTTP
  - A brief overview of the Http object's API
  - Observables and Observers
  - Catching Http Errors

Child/Nested Components

- The Angular follows component based Architecture, where each component manages a specific task or workflow. Each component is an independent block of the reusable unit.
- In real life angular application, we need to break our application into a small child or nested components. Then the task of root components is to just host these child components
- You can render it using child component's selector into parent component.

app.component.html

```
1
2 <h1>{{title}}. </h1>
3
4 <customer-list></customer-list>
5
```

Passing data to a child component

- In Angular, the Parent Component can communicate with the child component by setting its Property. To do that the Child component must expose its properties to the parent component. The Child Component does this by using the **©Input decorator**
- **In the Child Component**
  - Import the ©Input module from ©angular/Core Library
  - Mark those property, which you need data from parent as input property using ©Input decorator
- **In the Parent Component**
  - Bind the Child component property in the Parent Component when instantiating the Child

Various ways to use ©Input Decorator

- We used input ©Input annotation to mark the property in child component as input property. There are two ways you can do it Angular.
  - Using the ©Input decorator to decorate the class property

```
export class ChildComponent {
     @Input('MyCount') count: number;
}
```

  - Using the input array meta data of the component decorator

child.component.ts

```
1
2  @Component({
3       selector: 'child-component',
4       inputs: ['count'],
5       template: `<h2>Child Component</h2>
6       current count is {{ count }}
7  `
8  })
9  export class ChildComponent {}
10
```

©input() Example

child.component.ts

```ts

import { Component, Input  } from '@angular/core';

@Component({
    selector: 'child-component',
    template: `<h2>Child Component</h2>
               current count is {{ count }}
    `
})
export class ChildComponent {
    @Input() count: number;
}

```

app.component.ts

```ts

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

@Component({
  selector: 'app-root',
  template: `
        <h1>Welcome to {{title}}!</h1>
        <button (click)="increment()">Increment</button>
        <button (click)="decrement()">decrement</button>
        <child-component [count]=Counter></child-component>` ,
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Component Interaction';
  Counter = 5;

  increment() {
    this.Counter++;
  }
  decrement() {
    this.Counter--;
  }
}

```

Passing data to parent component

- There are three ways in which parent component can interact with the child component
  - Parent Listens to Child Event
  - Parent uses Local Variable to access the child
  - Parent uses a ©ViewChild to get reference to the child component

Parent listens for child event

- The Child Component exposes an EventEmitter Property. This Property is adorned with the ©Output decorator. When Child Component needs to communicate with the parent it raises the event. The Parent Component listens to that event and reacts to it
- **How to Pass data to parent component using ©Output**
  - Declare a property of type EventEmitter and instantiate it
  - Mark it with a ©Output annotation
  - Raise the event passing it with the desired data

```
countChanged: EventEmitter<number> = new EventEmitter()
```

Example

child.component.ts

```
1
2  import { Component, Input, Output, EventEmitter  } from '@angular/core';
3
4  @Component({
5      selector: 'child-component',
6      template: `<h2>Child Component</h2>
7                 <button (click)="increment()">Increment</button>
8                 <button (click)="decrement()">decrement</button>
9                 current count is {{ count }}
10     `
11 })
12 export class ChildComponent {
13     @Input() count: number;
14
15     @Output() countChanged: EventEmitter<number> =    new EventEmitter();
16
17     increment() {
18         this.count++;
19         this.countChanged.emit(this.count);
20       }
21     decrement() {
22         this.count--;
23         this.countChanged.emit(this.count);
24     }
25 }
```

Example

app.component.ts

```
1
2  import { Component} from '@angular/core';
3
4  @Component({
5    selector: 'app-root',
6    template: `
7          <h1>Welcome to {{title}}!</h1>
8          <p> current count is {{ClickCounter}} </p>
9          <child-component [count]=Counter (countChanged)="countChangedHandler($eve
   nt)"></child-component>` ,
10   styleUrls: ['./app.component.css']
11 })
12 export class AppComponent {
13   title = 'Component Interaction';
14   Counter = 5;
15
16   countChangedHandler(count: number) {
17     this.Counter = count;
18     console.log(count);
19   }
20 }
21
```

Parent uses local variable to access the Child in Template

- Parent Template can access the child component properties and methods by creating the template reference variable
- We have created a local variable, #child, on the tag <child-component>. The “child” is called template reference variable, which now represents the child component

app.component.ts

```
1
2         <child-component #child></child-component>` ,
3
```

- Now you can use the local variable elsewhere in the template to refer to the child component methods and properties as shown below

app.component.ts

```
1
2         <p> current count is {{child.count}} </p>
3         <button (click)="child.increment()">Increment</button>
4         <button (click)="child.decrement()">decrement</button>
5
```

Example

child.component.ts

```ts

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

@Component({
    selector: 'child-component',
    template: `<h2>Child Component</h2>
               current count is {{ count }}
    `
})
export class ChildComponent {
    count = 0;

     increment() {
        this.count++;
      }
    decrement() {
        this.count--;
    }
}

```

app.component.ts

```ts

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

@Component({
  selector: 'app-root',
  template: `
        <h1>{{title}}!</h1>
        <p> current count is {{child.count}} </p>
        <button (click)="child.increment()">Increment</button>
        <button (click)="child.decrement()">decrement</button>
        <child-component #child></child-component>` ,
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Parent interacts with child via local variable';
}

```

©ViewChild() to get reference to Child Component

- **©ViewChild** is the another technique used by the parent to access the property and method of the child component
- The **©ViewChild** decorator takes the name of the component/directive as its input. It is then used to decorate a property.
- Decorate parent property with **©ViewChild** decorator passing it the name of the component to inject

```
@ViewChild(ChildComponent) child: ChildComponent;
```

- Now, when angular creates the child component, the reference to the child component is assigned to the child property.

Example

child.component.ts

```ts
import { Component} from '@angular/core';

@Component({
    selector: 'child-component',
    template: `<h2>Child Component</h2>
               current count is {{ count }}
    `
})
export class ChildComponent {
    count = 0;

     increment() {
        this.count++;
      }
    decrement() {
        this.count--;
    }
}
```

app.component.ts

```ts
import { Component, ViewChild } from '@angular/core';
import { ChildComponent } from './child.component';

@Component({
  selector: 'app-root',
  template: `
        <h1>{{title}}</h1>
        <p> current count is {{child.count}} </p>
        <button (click)="increment()">Increment</button>
        <button (click)="decrement()">decrement</button>
        <child-component></child-component>` ,
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Parent calls an @ViewChild()';

  @ViewChild(ChildComponent) child: ChildComponent;

  increment() {
    this.child.increment();
  }

  decrement() {
    this.child.decrement();
  }
}
```

Component Lifecycle Hook

- A component has a lifecycle managed by Angular.
- Angular creates it, renders it, creates and renders its children, checks it when its data-bound properties change, and destroys it before removing it from the DOM.
- Angular offers lifecycle hooks that provide visibility into these key life moments and the ability to act when they occur.
- When the angular application starts it creates and renders the root component. It then creates and renders it's Children. For each loaded component, it checks when its data bound properties change and updates them. It destroys the component and removes it from the DOM when no longer in use.

Component Lifecycle Hook

- The Angular life cycle hooks are nothing but callback function, which angular invokes when a certain event occurs during the component's life cycle.
- For example
  - When component is initialized, Angular invokes ngOnInit
  - When a component's input properties change, Angular invokes ngOnChanges
  - When a component is destroyed, Angular invokes ngOnDestroy

Order of Execution of Lifecycle Hooks

ngOnChanges	Called after a bound input property changes
ngOnInit	Called once the component is initialized
ngDoCheck	Called during every change detection run
ngAfterContentInit	Called after content (ng-content) has been projected into view
ngAfterContentChecked	Called every time the projected content has been checked
ngAfterViewInit	Called after the component's view (and child views) has been initialized
ngAfterViewChecked	Called every time the view (and child views) have been checked
ngOnDestroy	Called once the component is about to be destroyed

ngOnChanges

- The Angular invokes this life cycle hook whenever any data-bound property of the component or directive changes.

- The parent component can communicate with the child component, if child component exposes a property decorated with ©Input decorator. This hook is invoked in the child component, when the parent changes the Input properties. We use this to find out details about which input properties have changed and how they have changed.

ngOnInit

- This hook is called when the component is created for the first time. This hook is called after the constructor and first ngOnChanges hook.
- This is a perfect place where you want to add any initialisation logic for your component.
- Note that ngOnChanges hook is fired before ngOnInit. Which means all the input properties are available to use when the ngOnInit is hook is called
- This hook is fired only once
- This hook is fired before any of the child directive properties are initialized

ngDoCheck

- Detect and act upon changes that Angular can't or won't detect on its own.

- Called during every change detection run, immediately after ngOnChanges() and ngOnInit().

ngAfterContentInit

- This Life cycle hook is called after the Component's content has been fully initialized. This hook is called after the properties marked with ContentChild and ContentChildren are fully initialized.

- The Angular Component can include the external content from the child Components by transuding them using the <ng-content></ng-content> element. This hook is fired after these projected contents are initialized.

- This is a component only hook and Called Only once.

ngAfterContentChecked

- This life cycle hook is called after the Components Content is checked by the Angular's Change detection module. It is called immediately after ngAfterContentInit and after every subsequent ngDoCheck().

- This is a component only hook

ngAfterViewInit

- Similar to ngAfterContentInit, but invoked after Angular initializes the component's views and all its child views. This is Called once after the first ngAfterContentChecked.
- A component-only hook.

ngAfterViewChecked

- The Angular fires this hook after it checks the component's views and child views. This event is fired after the ngAfterViewInit and after that for every subsequent ngAfterContentChecked hook.
- This is a component only hook.

ngOnDestroy

- This hook is called just before the Component/Directive instance is destroyed by Angular.
- You can Perform any cleanup logic for the Component here. This is the correct place where you would like to Unsubscribe Observables and detach event handlers to avoid memory leaks.

How to Use Lifecycle Hooks

1. Import Hook interfaces
   - Import hook interfaces from the core module. The name of the Interface is hook name without ng. For example interface of ngOnInit hook is OnInit

```
import { Component,OnInit } from '@angular/core';
```

2. Declare that Component/directive Implements lifecycle hook interface
   - Next, define the AppComponent to implement OnInit interface

```
export class AppComponent implements OnInit {
```

3. Create the hook method (e.g. ngOnInit())
   - The life cycle hook methods must use the same name as the hook.

```
ngOnInit() {
  console.log("AppComponent:OnInit");
}
```

Dependency Injection

- **Dependency Injection (DI) is a technique in which we provide an instance of an object to another object, which depends on it. This is technique is also known as “Inversion of Control” (IoC).**
- Dependency injection is an important application design pattern. It's used so widely that almost everyone just calls it *DI*.
- Angular has its own dependency injection framework, and you really can't build an Angular application without it.
- Dependency Injection is a combination of two terms, those are “Dependency” and “Injections”. Dependency is an object or service that can be used in any another object and Injection is a process of passing the dependency to a dependent object. It creates a new instance of class along with its required dependencies.

Benefits of DI

- **Loose coupling and reusability**
  - The main goal of DI is to avoid tightly coupled components by injecting dependencies rather than instantiating them directly in consuming component**s**

- **Testability**
  - DI increases the testability of your components in isolation. You can easily inject mock objects if their real implementations aren’t available or you want to unit-test your code.

Angular 2 DI Framework

- **Angular 2 Dependency Injection Framework** helps us to implement the **Dependency injection Pattern**.
- This framework consists of four main parts:
  - **Consumer**: The Component that needs the Dependency
  - **Dependency**: The Service that is being injected.
  - **Provider**: Maintains the list of Dependencies. It provides the instance of dependencies to the injector
  - **Injector**: Responsible for injecting the instance of the Dependency to the Consumer

Service

- Services allow for greater separation of concerns for your application and better modularity by allowing you to extract common functionality out of components.
- Service is a piece of reusable code with a Focused Purpose. A code that you will use it in many components across your application.
- Services may have their associated properties and the methods, which can be included in the component. Services are injected, using DI (Dependency Injection).
- Services share data or functions between different parts of angular application.

Service Cont..

- **Services are used for**
  - Features that are independent of components such a logging services
  - Share logic and data across components
  - Encapsulate external interactions like data access

- **Advantages of Service**
  - Services are easier Test.
  - Services are easier to Debug.
  - You can reuse the service.

Service Example

- **©Injectable()** decorator use to create service. Without this decorator dependency injection will not work.

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

@Injectable()
    export class ProductService{
}
```

- Now our AppComponent needs to ask for the dependencies. This is done in the constructor as shown below

```
constructor(private productService:ProductService) { }
```

Angular Providers

- The Angular Providers allow you to define set of Dependencies and provides the concrete, runtime version of that dependencies when asked for it.
- To Provide an instance of the service, the Angular Providers needs to know where to find the provider and how to instantiates it.
- In Angular, we can register our dependency at two levels
  - **At App Level:**

```
@NgModule({
  ...
  providers: [ProductService]
})
```

  - **At Component Level:**

```
providers:[ProductService]
```

Angular Providers Cont..

- The previous syntax is an actual shorthand notation for the following syntax

```
providers :[{ provide: ProductService, useClass: ProductService }]
```

- The above syntax has two properties.
  - **Token:** The First property Provide is known as the Token. The token is used by the injector to locate the provider
  - **Provider:** The second property useClass is Provider. It tells how and what to inject.
- The injector maintains an **internal collection of token-provider map**. The token acts as a key to that collection. Injector uses the Token (key) to locate the provider.

Angular Providers Cont..

- The Angular-Dependency Framework provides several types of providers.
  - **Class Provider(useClass)**
    - The Class Provider is used, when you want to provide an instance of the class.

```
providers :[{ provide: ProductService, useClass: ProductService }]
```

- In the above, example *ProductService* is the **Token** (or key) and it maps to the *ProductService* Class. In this case both the Class name and token name match.

Angular Providers Cont..

- **Value Provider (useValue)**
  - Value Provider is used, when you want to provide a Simple value
  - This property can be used when you want to provide URL of Service class, Application wide Configuration Option etc.

```
providers :[ {provide:'USE_FAKE', useValue: true}]
```

  - Another example

```
const APP_CONFIG = { serviceURL: "www.serviceUrl.com\api", IsDevleomentMode: true
};
providers: [{ provide: AppConfig, useValue: APP_CONFIG }]
```

Angular Injectors

- The injector object is used to create an instance of a dependency.
- An injector maintains a container of service instances that it created
- The injector passes the request to the injector of the parent component. If the provider is found, the request returns the instance of the Provider. If not found then the request continues until the request reaches the top most injector in the injector chain.

```
constructor(private productService:ProductService) { }
```

Interacting with server using HTTP

- Most front-end applications communicate with backend services over the HTTP protocol.
- In order to start making HTTP calls from our Angular app we need to import the angular/http module and register for HTTP services. It supports both XHR and JSONP requests exposed through the HttpModule and JsonpModule respectively.
- HttpModule serves the purpose to perform HTTP requests.
- Asynchronous HTTP requests can be implemented using callbacks, promises or observables.

Making GET requests with Http object

- When Angular's Http object makes a request, the response comes back as Observable, and the client's code will handle it by using the subscribe() method.
- To be able to use the Http service within your components we first need to include the HttpModule in the Angular application. First we need to import Http module in the application's root module in file app.module.ts:
- Once imported you can make use of *Http* in your components. To make *Http* available in the component class you need to inject it into the class constructor.

Making GET requests with Http object

- To make HTTP requests we will use the Http service.

```
import { Injectable } from '@angular/core';
import { Http, Response } from '@angular/http';
import 'rxjs/add/operator/map';

@Injectable()
export class UserService {
  constructor (
    private http: Http
  ) {}

  getUser() {
    return this.http.get(`https://conduit.productionready.io/api/profiles/eric`)
    .map((res:Response) => res.json());
  }

}
```

- In addition to Http.get(), there are also Http.post(), Http.put(), Http.delete(), etc. They all return observables.

What is observables?

- Observable help us to manage async data. You can think of Observables as an array of items, which arrive asynchronously over time.
- The observables implement the observer design pattern, where observables maintains a list of dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.
- Observables provide support for passing messages between publishers and subscribers in your application. Observables offer significant benefits over other techniques for event handling, asynchronous programming, and handling multiple values.

Observables Operators

- Operators are methods that operate on an Observable and return an observable. Each Operator modifies the value it receives. These operators are applied one after the other in a chain.
- The RxJs Provides several Operators, which allows you to filter, select, transform, combine and compose Observables. Examples of Operators are **map, filter, take, merge** etc.
- RxJS is a great tool for managing data with help of the Observer pattern. Instead of keeping state in a variable, it stores data in a stream

```
import { Observable} from 'rxjs/Rx';
```

What is Observers?

- Observable instance begins publishing values only when someone subscribes to it.
  - **Eg. subscribe(success, error, complete)**
- The Subscriber Operator is the glue that connects an observer to an Observable.
- We need to Subscribe to an observable to see the results of observables.

```
.subscribe((response) => {this.repos=response;},
          (error) => {this.errorMessage=error; this.loading=false; },
          () => {this.loading=false;})
```

Observers Cont..

- The subscribe method has three arguments. Each specifies the action to be taken when a particular event occurs
  - **Success:** This event is raised whenever observable returns a value. We use this event to assign the response to the repos

```
(response) => {this.repos=response;}
```

  - **Failure:** This event occurs, when observable is failed to generate the expected data or has encountered some other error

```
(error) => {this.errorMessage=error; this.loading=false; },
```

  - **Completed:** This event fires, when the observables complete its task. We disable the loading indicator here.

```
() => {this.loading=false;})
```

Catching Http Error

- A HTTP request can fail. Because of a poor network connection or other circumstances which can not be foreseen.
- Therefore you should always include code which handlers an error situation. Adding a second callback method to the subscribe method is the way we should do:

```
validateUser(payload) {
    return this.http.post("/api/login", payload)
        .map((res: Response) => res.json())
        .subscribe(
              authData=>this.storeToken(authData.token),
              err=>console.error(err),
               ()=>console.log("Authentication Completed.."))
     }
```

Catching Http Error Cont..

- We also have the option of using the .catch operator of RxJS. It allows us to catch errors on an existing stream, do something, and pass the exception onwards.

```
getUsers() {
    return this.http.get("/api/users")
        .map((res: Response) => res.json())
        .catch(err => this.handleError(err));
     }

handleError(error: Response) {

    // Do messaging and error handling here
    return Observable.throw(error)
}
```

Http Status Code

- Most of the users are aware about what HTTP Status Codes are all about. Status codes such as 404 and 500 errors are popular among most of the tech geeks.

Code	Description	Code	Description
200	OK	400	Bad Request
201	Created	401	Unauthorized
202	Accepted	403	Forbidden
301	Moved Permanently	404	Not Found
303	See Other	410	Gone
304	Not Modified	500	Internal Server Error
307	Temporary Redirect	503	Service Unavailable