Angular life-cycle →

→ Once a ~~Angular~~ new component is instantiated/ angular goes for a couple of different phases in this creation process, and it will actually give us a chance to hook into these phases and execute some code

1. `ngOnChanges` → called after a bound input property changes

2. `ngOnInit` → Called once the component is initialized

3. `ngDoCheck` → executed a lot because this will run whenever change detection runs.
   → For many events like timer completes, observable resolved → A lot of time this will run because you clicked some button which doesn't change something, But still it's event and due to these event angular has to check if some changes happens something changed or not, so ngDoCheck and angular change will run.
   detection events

4. `ngAfterContentInit` → called after Content (ng-content) has been projected into view. (only once)

5. `ngAfterContentChecked` → called every time the projected content has been checked.

6. `ngAfterViewInit` → called after the component's view (and child views) has been initialized.

7. `ngAfterViewChecked` → called every time the view & child view have been checked.

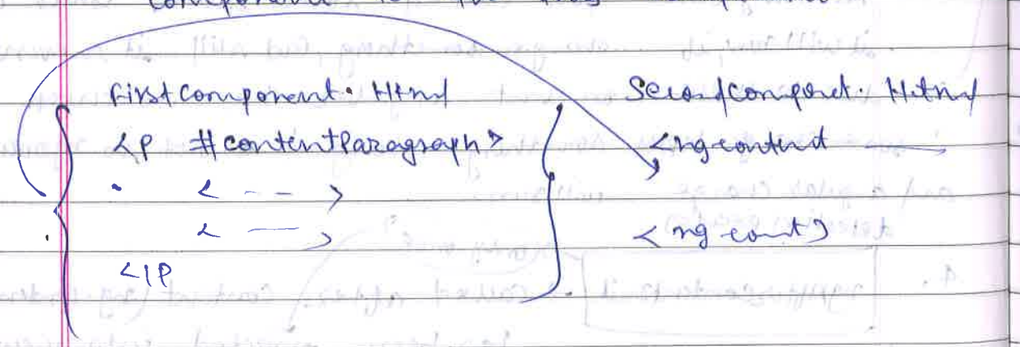
@ViewChild:-

→ from this we can get ~~refere~~ elementrefere of HTML DOM from the local reference or template variable.

Ex.

```
<input type="text" #inputElement>
<button ---->
@ViewChild('inputElement') inputElement: ElementRef
click() {
  Console.log(this.inputElement.nativeElement.value);
}
```

@ContentChild :- It's give access to the DOM element's of ng-content that will be projected by the other component to the this component.



SecondComponent.ts

@ContentChild('contentParagraph') paragraph: ElementRef;

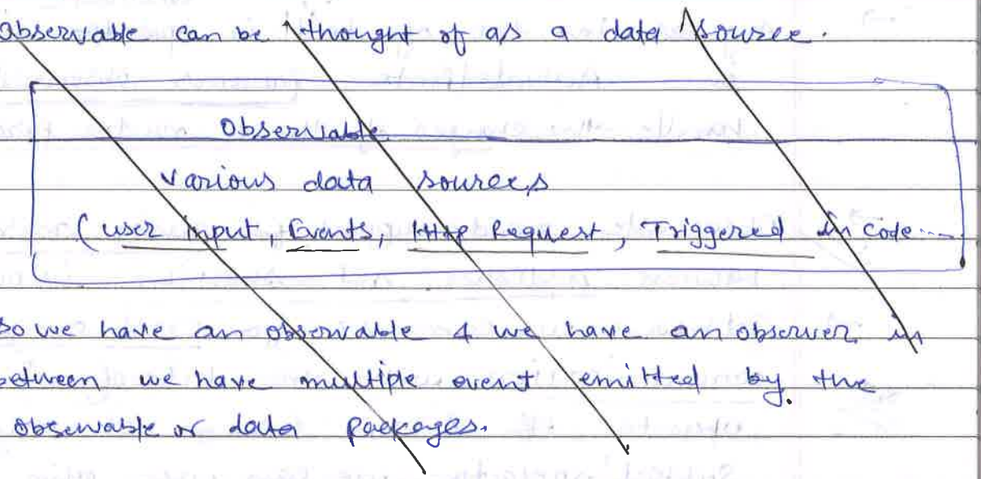
value of This will be ~~accessible~~ accessible in ngAfterContentInit() lifecycle ~~or~~ hook and ~~but~~ latter hooks.

b/c ~~or~~ once (<ng-content>) ~~is~~ is projected by the ~~to~~ View than only we can access Template

Observables → follows Observable Pattern

Date:
Page No:

→ Observable can be thought of as a data source.



→ So we have an observable & we have an observer in between we have multiple event emitted by the observable or data packages.

→ Three ways to handling data packages:-

→ There are observables too example hook up to a normal button which never complete. b/c how would you know when it completes. A user could click the button how often the user wants right.

→ Other observable is like Http request will have a clear end will complete eventually.

→ we use it to handle asynchronous tasks because all these data sources, your user events are triggred & or Http request are asynchronous tasks you don't know when they will happen. and you don't know how long they will take.

→ Therefore we need methods to handling such async tasks. And historically you might have used callbacks or promises, its not bad to use them, Observable is just a different approach of handly it.

also we need to subscribe — EventEmitter class extends Observable

EventEmitter, http

→ Angular has lot of built in Observable operator like in ActivatedRoute params Observable to handle the changes of these router parameters.

→ Observable provide support for passing message between publisher and subscriber in our app.

→ It means we can use for cross component communication with the help of Subject operator of 'rxjs' lib or not only with 'Subject' operator we can use other available operators also

→ Angular makes use of observables as an interface to handle a variety of common asynchronous op's:-

(i) The EventEmitter class extends Observable

(ii) The HTTP module uses Observable to handle async req. & response.

(iii) The Router & FormsModule uses observable to listen for and response to user input events.

⇒ Subject:- is a basically an Observable but it allows us to conviently push it to emit new data.

Ex:
```
class UserService {
    userActivated = new Subject();
}
```

Component1:-
```
this.UserService.userActivated.next("some data");
```

Component2:-
```
this.UserService.userActivated.subscribe(
    (data) => {},
    (error) => {},
);
```

Diff between Promises & Observable

(1) Observable can have multiple values over time but promises always return only one value.

(2) Another thing is that Observable are cancelable. If the result of an HTTP request to a server or some other expensive async operation isn't needed anymore, the subscription of an observable allows to cancel the subscription, while a promise will eventually call the success or failed callback even when we don't need them.
→ There are 'promise' libraries out there that support cancellation, but ES6 'promise' doesn't so far.

(3) Observables provides a bunch of useful operators like map, forEach, reduce etc... (retry(), replay())

(4) We can always convert Observables to promise.

→ Observable operators simple returns new Observable, you can of course also chain those operators.

Ex:
```
const myNumbers = Observable.interval(1000)
    .map(
        (data: number) => { return data * 2; }
    );
myNumbers.subscribe(
    ...
);
```

Brilliant

async pipe

for Observables

→ It would subscribe automatically and after 2 seconds it would simply recognize that something changed the promise resolved in in the case of an Observable that data sent from the subscription and it will present the data on the screen. And also unsubscribe the observable subscription.

Ex. =>

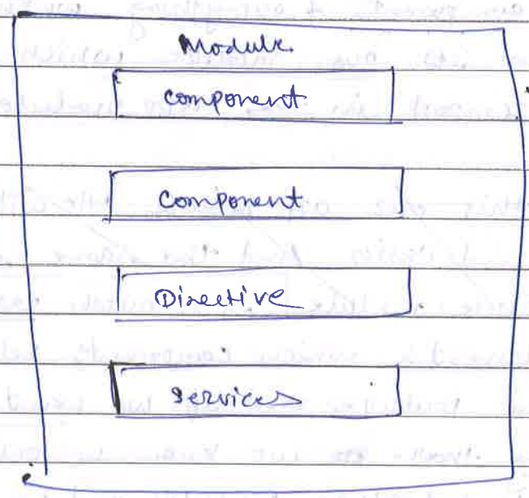
```
Status = new promise((resolve, reject) => {
    setTimeOut(() => {
        resolve("promise resolved");
    }, 2000);
})
```

template → `<h2> Promise Status : {{ status | async }} </h2>`

Modules

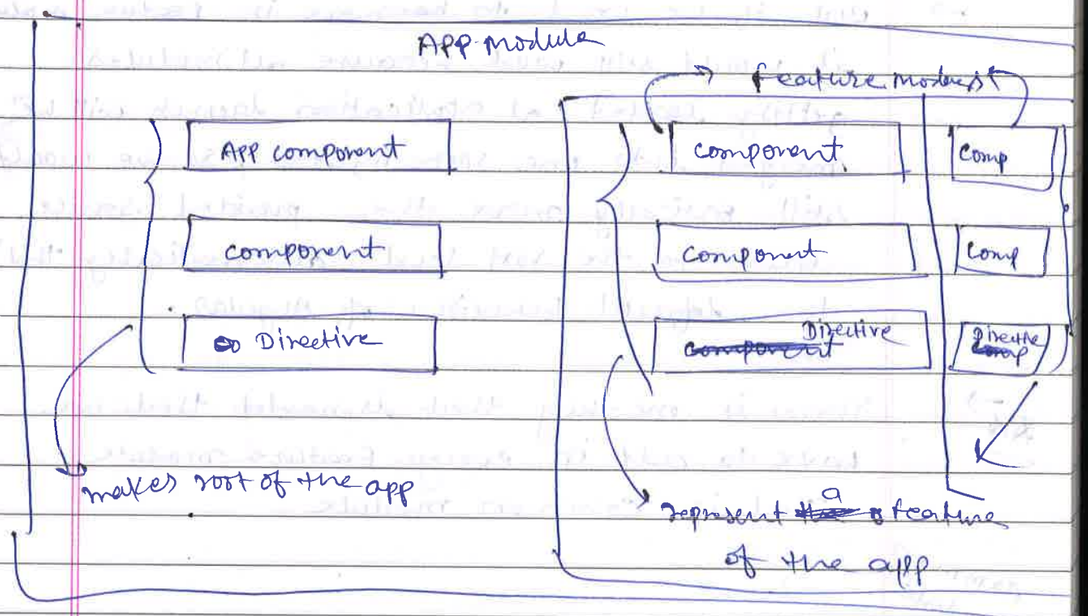
Style
Date:
Page No:

→ Decrease the file size & also restructure our code a better way and an easier to maintain way.



→ we can put together (feature module) in a module, related components, services, pipes, and directives which belongs to the same functionality.

→ So looking at the module we can get which components, services, pipes and directives this module uses.



→ ~~This is a good case to be outsourced into a feature module.~~ So now we would put all these components & everything which belongs to it in its own module which we then simply import in the app module.

→ due to this our app module file will become leaner and easier And the same is true for the feature module. it's much easier to see which elements which components belong to a certain feature. and if we want to change something there we know we only have to go to that feature module and we quickly see which dependencies we have. which components belongs to the rest of his feature and so on.

→ If in our app we have a service related to a feature but used in whole app we should keep in appmodule providers array.

→ But if we want to more in feature module, it would still work because all modules getting loaded at application lacunch will be merged into one root injector, so we would still basically move the provided service back to the root level automatically that's the default behaviour of Angular.

→ There is one thing that is needed that we have to add in every feature module, That is 'Common module'

common module

common module →

→ we don't have to add that to every feature module but what the common module does it it gives you access to these common directives like ngClass, ngFor, ngIf and chances are pretty high that basically every feature module we have will use one of these directives.

common vs browser?

→ Why we don't use common module in the AppModule we have browserModule instead of common module.

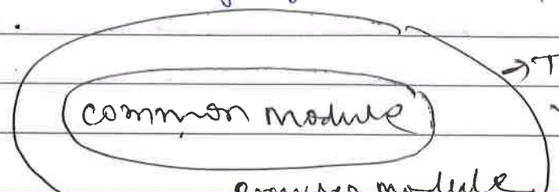
→ BrowserModule is basically contains all the features of the common module and then some additional features which are only needed at the point of time the app starts, therefore only needed in app module.

use only in appmodule

→ A module is only able to use what we define in that module, services are kind of an exception.

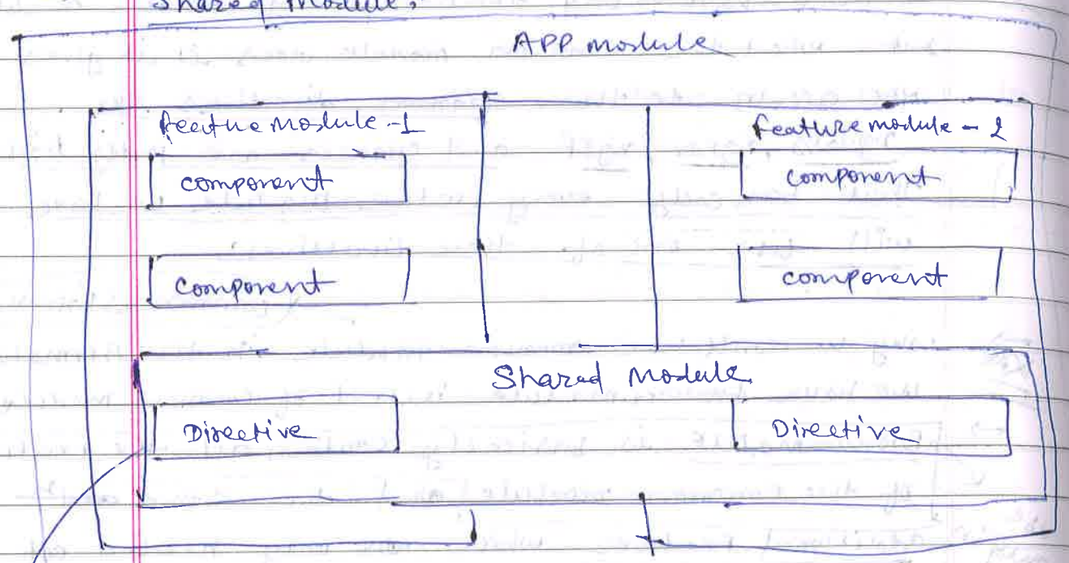
→ we must not declare components, pipes or directives in more than one module.

→ Simple rule in our app we must only call for RouterModule.~~for~~Root() in our root module. If we ever register routes anywhere else and you keep using router module we must use for child like RouterModule.forChild() because you are not on the ~~root~~ root ~~router~~ module anymore, But we are on the child module because in the end everything will be imported in AppModule.



→ That's why in app module we are not using common

→ Shared module:-



→ put these directives into a shared module which basically is a module not containing a feature but only something which is shared across multiple module.

Lazy loading:-

→ If the user never visits some feature/part of that means a lot of the code might never be used because if the user never visits the recipe section all the code related to that including all the template and everything will never be used, so we downloaded too much at the beginning.

→ That is where lazy loading comes into play.

→ That means the modules is only loaded if we actually visit a route loading to this module.

app-routing.module.ts

{path: 'recipes', loadChildren: './recipes/recipe.module#RecipesModule'}

→ If we want to use route protection (canActive) on lazy load route we should add before loading the code means after loadChildren:

{path: '', loadChildren: '', canLoad: [AuthGuard]}

→ at the point of time our app launches it is bootstraped with the app module

→ everything listed here in imports of app module is imported.

→ That is why we have no performance difference if we create multiple - feature module because in the end those started together anyways

→ Now we don't want to load ~~to eger~~ eger eagerly anymore so we somehow need to remove it from this import array here.

→ After removing webpack doesn't add this recipe's module to our initial bundle anymore. ~~since~~.

→ Since webpack doesn't add this to our bundle anymore it also means everything we reference here

→ we can load it easily by going to the app routing file and here we reintroduce a route we had there before.

→ It points to recipe, but now we no longer load the recipe component here.

Ex. =

```
const approutes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'recipe', component: RecipesComponent },
  { path: 'shopping-list', component: ... }
];
```

→ don't do this, that would again load it eagerly.

→ Instead here we use another property. loadChildren → it takes a string not a type like in component: HomeComponent

So in loadChildren property we put a string that points to the module that we want to load dynamically, lazily and at the point of the time we visit this route.

Ex. =

```
const approutes: Routes = [
  { path: ... , loadChildren: './recipe/recipe.module' },
];
```

Ex. 2

```
const approute: Routes = [
  { path: '', component: HomeComponent },
  { path: 'recipes', loadChildren: './recipe/recipes.module#RecipesModule' },
  { path: ... },
];
```

to separate path from the class name

class name of recipe.module

with this in place now whenever we visit '/recipe' route it will dynamically load the recipes module. ~~but only once we entered this into our url.~~

→ we can protect lazy (loading) routes with canLoad

~~→ we can~~ → we can prevent loading with canLoad

→ ~~We can add canActive to the lazy~~ loaded routes but that of course means, ~~that you might load code~~ It would be better to check that before loading the code →

Ex. { path: 'recipes', loadChildren: ()

'./recipes/recipes.module#RecipesModule',

canLoad : [AuthGuard] }

→ In this example, the AuthGuard should implement the CanLoad interface.

Modules- examples:-

→ It bundled certain functionalities and we can import them.

→ With this module concept we can ~~defi~~ divide our appn in multiple feature that means multiple feature-module.

→ Through module concept, ~~it~~ →

- Decrease the file size (Separation of Concern)
- easier to maintain (restructure code)
- lazy loading implementation (performance improvement)

→ helpful in lazy loading of feature-modules.

Structure → app.module.ts

```
import { } from ' ';
   ⋮ → some import statements.
@NgModule({
    declarations: [ ],
    imports : [ ],
    providers : [ ],
    bootstrap : [AppComponent]
})
export class AppModule { }
```

(providers: defines what services we may use)

→ If we create a feature module we also have to move our routes related to this feature module.

Example → feature module (recipe)

recipes-routing.module.ts

```
import { } from '';
   ⋮

const recipesRoutes: Routes = [
    { path: '', component: RecipesComponent,
      children: [
      { path: '', component: RecipeStartComponent },
      { path: 'new', component: ———, canActive: [—] },
      { path: 'id', ——————————— },
    ] },
]
@NgModule ({
        imports: [
            RouterModule.forChild(recipesRoutes)
        ],
        exports: [RouterModule]
})
export class RecipesRoutingModule { }
```

recipes.module.ts

```
import ....
   ⋮
@NgModule({
    declarations: [
        RecipesComponent,
        ⋮
    ],
    imports: [
        CommonModule,
        RecipesRoutingModule,
        ShareModule,
        ReactiveFormsModule
    ]
})
```

→ app-routing.module.ts

```
import ....
   ⋮
const appRoutes: Routes = [
    { path: '', component: HomeComponent },
    { path: 'recipes', loadChildren: './recipes/recipes.module#
                                    RecipesModule' },
    { path: 'shopping-list', component: ShoppingListComponent }
];

@NgModule ({
    imports: [RouterModule.forRoot(appRoutes)],
    exports: [RouterModule]
})

export class AppRoutingModule { }
```

→ app.module.ts

```
import ....
   ⋮
@NgModule({
    declarations: [
        AppComponent,
        HeaderComponent,
        HomeComponent
    ],
    imports: [
        BrowserModule, HttpModule, AppRoutingModule,
        ShareModule, AuthModule ],
providers: [ShoppingListService, RecipeService, AuthGuard .... ],
bootstrap: [AppComponent]
})

export class AppModule { }
```

Style
Date:
Page No:

Share.module.ts

```
import ....
:
@NgModule({
    declaration: [ DropDownDirective ],
    export: [ CommonModule,
              DrowdownDirective
            ]
})
export class ShareModule { }
```

Preloading Lazy Loaded Routes:-

~~we implemented lazy loading,~~

→ In case of lazy loading, when user clicks ~~the~~ or visits /recipe you load this whole chunk at this point of time.

→ So that might give us a little glitch where the application kind of hangs because that code needs to be downloaded and depending on the connection that may take a couple of millisecond or second. that is not great.

→ इसलिए इस glitch से बचने के लिए हम App launch होने के बाद जब user is busy with other area of the app तब recipe module को preload कर लेंगे।

~~modules & Service injection~~ Style Date: Page No:

→ Inside app-routing.module.ts

import In RouterModule.forRoot(appRoutes, { })

preloadingStrategy : PreloadAllModules

→ we can pass a JavaScript object as second argument where we can configure this router module.

→ here one important configuration that is very useful it's the ~~preloading strategy~~ property

preloadingStrategy

↓

This property takes a value

↓

here we can pass the type of pre-loading strategy we want to use.

→ default Strategy is don't preload.

→ we can add PreloadModules → which we need to import from '@angular/router' module.

This is the Strategy which as the name implies pre-load all lazy-loaded modules, After the app has been loaded.

→ Now we can make (write) our own strategies that's more advanced if you want to control which features should get preloaded when,

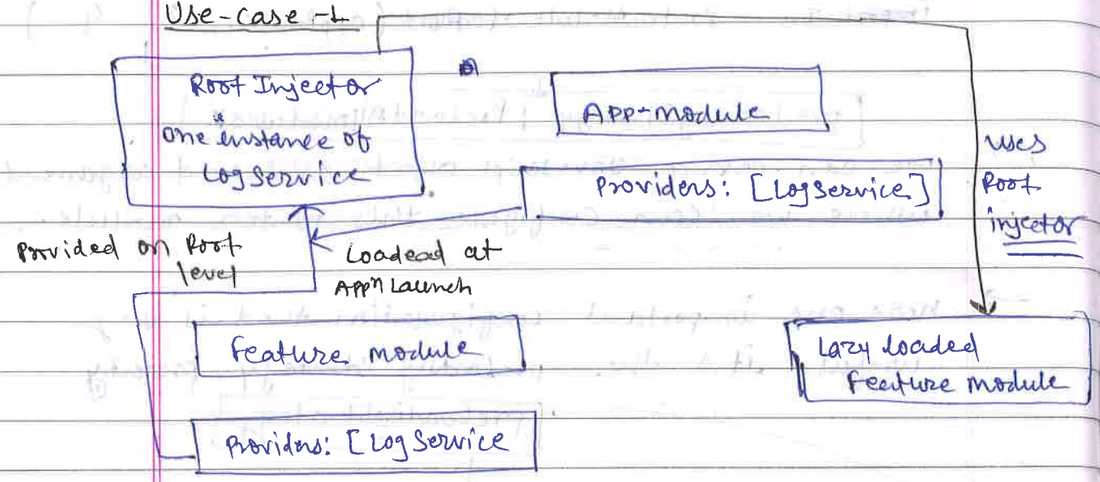
but this is a great Strategy to basically tell angular please make sure that you preload the modules which are loaded lazily.

```
@NgModule({                    ⇓
  imports: [ RouterModule.forRoot(appRoutes,
      {preloadStrategy: preloadAllModules})
```

Brilliant

Module & Service Injection

Use-case-1



→ What happens is that in the end we have a route injector for the whole appn which is created by angular at the point of time app starts.

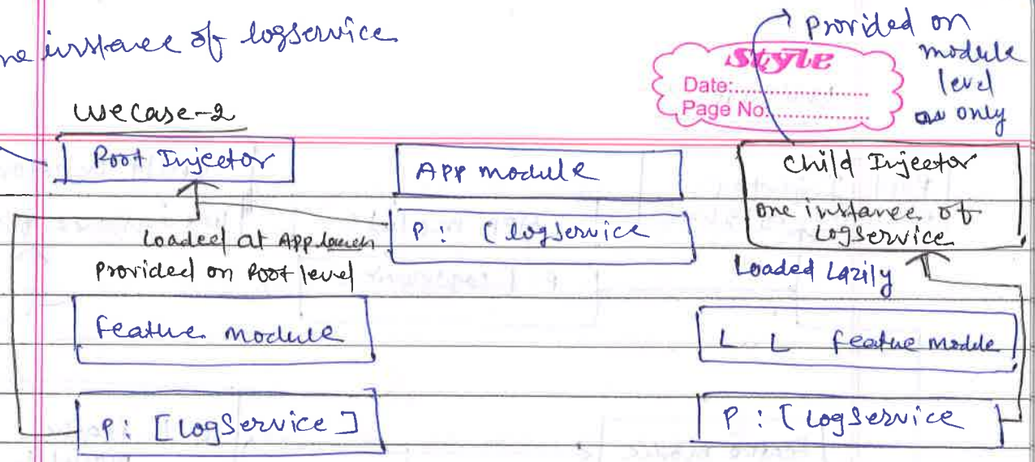
→ There is only one Root injector and all the services we provide in either of the module is added to the root injector that means in the appn you are going to use same instance.

→ No special instance for feature-module

→ This Root injector available for lazy-loaded feature module also. we would use this instace in the whole application including the lazy loading modules.

one instance of logservice

Usecase-2



Provided on module level as only

→ ~~But~~ for the App module & eager loaded feature module we will still merge the providers together and basically get one root injector for this service

→ But at the point of time we load the lazily ~~load~~ loaded ~~modu~~ feature module angular will treat a child injector for this module.

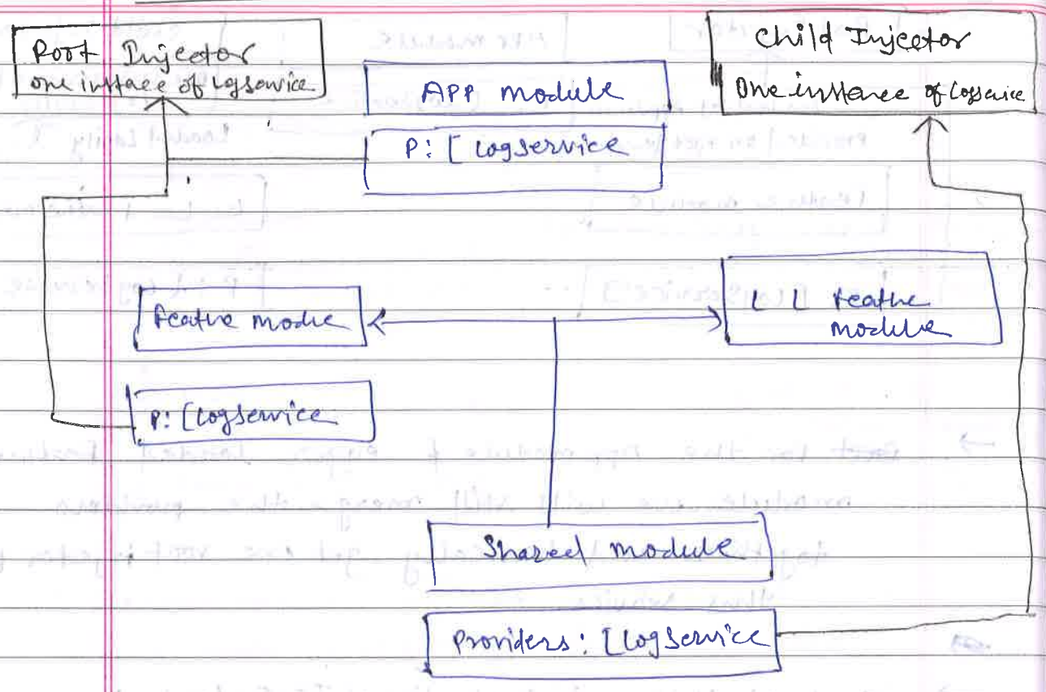
⇓

So all the component of this module will use this different instance of logservice

⇓

Of course it's absolutely find to add the provider/s service to ~~lazy~~ lazily loaded module but you have to keep in mind that this will means you are not going to use the same instance ~~of the~~ as the rest of your appn.

Usecase-3:



→ we should avoid this kind of situations

→ Expected Behaviour :- Lazily loaded module uses Root injector

→ Real Behaviour :- Lazy loaded module uses child injector

* when we visit the route which loads it angular will give you a child injector because we import the share module which provides the service.

⇓

(So this is same like usecase-2.

⇓

We got a different instance of service for lazy-loaded module that is not the behaviour we wanted.

→ Because why would we have added a service so the providers array of a shared module if we did not ~~want to~~ use share the same instance of the service in the whole appn.

⇓

we can go for usecase-1

Brilliant

Hierarchical Injector in services

→ Angular dependency injector actually is a hierarchical injector that means if we provide a service in some place let's say on one component so the same instance of this service will be available to this component and child components.

App module – Same instance of service is available Application-wide

↓

all component, directories, pipes, services will receive same instace

App Component – Same instance of service is available for all component (but not for other services)

→ The instances don't propagate up. They only go down till last component with no-child.

Any other component – Same instace of service is available for the component and all its child component

Observables - (2)

→ we use promise / observable

to handle ↓ to work with

asynchronous operation

→ Asynchronous operations :-

When we call the server there is going to be a delay.

↓

That delay might be half of second, it might be 3 or 20 seconds depending on the connection speed.

⇓

In situation like this the process that is executing our code, does not want to block while waiting for the result coming back from the server.

⇓

bcoz ~~the process~~ if that process blocks than user can't interact with browser.

⇓

So that process is going to call the server behind the scene, and when the result is ready, it's going to display to the user.

↳ This is asynchronous operation.

Style
Date:
Page No:

Asynchronous means → Non-blocking

Asynchronous operations in JS →
- Calling the server (AJAX)
- Timer function (SetTimeout)
- Events.

⇓

- Normal button click event which never completes b/c how would you know when it completes.
- A user could click the button whenever he wants

Reason

- These event Observables never completes

* Other Observable used with ~~org dope~~ Http requests will have a clear end, they will end eventually.

∞ So, observable we use to handle asynchronous operations or tasks, bcoz we don't know when user events are triggered, or when Http request or Timer fn or some other async task when they will happen and how long they will take we do not ~~no~~ know.

Brilliant

Angular makes use of observables to handle a variety of asynchronous operations:

(1) The EventEmitter class extends Observables.

⇒ Angular provides an EventEmitter class that is used when publishing values from a component through a @Output decorator.

⇒ EventEmitter extends Observables, adding an emit() method so, when we call emit(), it passes the emitted value to the next() method of any subscriber observer.

Ex: class commonService {
    statusUpdated = new EventEmitter<string>();
}

→ service

→ whom emit the event

class firstComponent {
    strValue : string;
    constructor(private commonService : CommonService)
    { }

    ShareData() {
        this.commonService.statusUpdated.
            emit(this.strValue);
    }
}

Subscribed for the Observable to receive the data when emitted by first component

class SecondComponent implements OnInit, OnDestroy {
    strValue : any;
    constructor(private commonService : CommonService)
    { }

    ngOnInit() {
        this.commonservice.statusUpdated.subscribe(
            (data.string) => this.strValue = data);
    }

    ngOnDestroy() : void {
        this.commonService.statusUpdated.unsubscribe();
    }
}

next()

(2) Http module uses Observable to handle Asynchronous request & response.

Ex: export interface Config {
    herosUrl : string;
    textfile : string;
}

config.json

{
    "herosUrl" : "api/heroes",
    "textfile" : "assets/textfiles.txt"
}

config.service.ts

⇒ export class ConfigService {

configUrl = 'assets/config.json';

constructor (private http: HttpClient) { }

getConfig( ) : Observable<Configs> {

return this.http.get<Configs>(this.configUrl);

}

}

⇒ config.component.ts

export class ConfigComponent {

config : Config;

ShowConfig( ):

constructor (private configService : ConfigService)
{ }

showConfig ( ) {

this.configService.getConfig( )

.Subscribe (data ⇒ this.config = {

herosUrl : data 'herosUrl',

textfile : data.textfile

},

error ⇒ this.error = error

);

}

Style
Date:
Page No:

(3) The Router and Forms modules use Observables to listen for and respond to user-input events.

⇒ For Router →

→ Router.events provides events as Observables. We can use the filter( ) operator from RXJS to look for events of interest.

→ The ActivatedRoute is an injected router service that makes use of observables to get information about a route path & parameter.

→ for example, ActivateRoute.url contains an observable that reports the route path.

Ex.

constructor (private activatedRoute : ActivatedRoute)
{ }

ngOnInit ( ) {

this.activatedRoute.url.

Subscribe (url => console.log ('The URL changed to: ' + url));

}

⇒ Other Observable in ActivatedRoute

- params → R.T.U.
- queryParams
  - Observable<Params>
  - An observable of the query parameters

Brilliant

Params → Observable<Params>

↓

an observable of the ~~query~~ matrix parameters scoped to this route.

Ex:

```
id: Number;
constructor(private route: ActivatedRoute) { }

ngOnInit() {

  this.route.params
    .subscribe((params: Params) => {

      this.id = +params['id'];
    });
}
```

Defining observers :-

→ A handler for receiving observable notifications implements the Observer interface.

⇓

It is an object that defines callbacks methods to handle the three types of notifications that an observable can send.

(Handler) = Notification type	Description
next →	Required, A handler for each delivered value. Called zero or more time after execution starts.
error →	Optional. A handler for an error notification. An error halts execution of the observable instance.
complete →	Optional. A handler for the execution-complete notification.

→ An observer object can define any combination of these handlers.

→ If we don't supply a handler for a notification type, the observer ignores notifications of that types.

Subscribing →

→ An Observable instance begins publishing values only when someone subscribes to it.

→ we can subscrib by calling subscribe() method, passing an observer object to recieve the notification.

Ex. (1)

```
// simple observable that emits three values
const myObservable = observable.of(1,2,3);
// create Observer object
const myObserver = {
    next : x => console.log('value :' + x),
    error : err => console.log(*err);
    complete : () => console.log('observer got a
                        complete notification'),
};
// Execute with observer object
myObservable.subscribe(myObserver);
```

Ex-2.

```
myObservable.subscribe(
    x => console.log('value :' + x),
    err => console.log(err);
    () => console.log('observer got a
                        complete notification')
);
```

Ex-1.

Creating Observables :-

```
const myObservable = Observable.of(1,2,3);
```

↓↓ to create observable equivalent to above something like this also we can do ↓

```
// this fn runs when subscribe() is called
function sequenceSubscriber(observer){
    observer.next(1);
    ————(2);
    ————(3);
    ——.complete();
}
// create a new Observable
const sequence = new Observable(sequenceSubscriber);
sequence.subscribe({
    next(num) {console.log(num);},
    complete() {console.log('Finished sequence');}
});
```

Brilliant