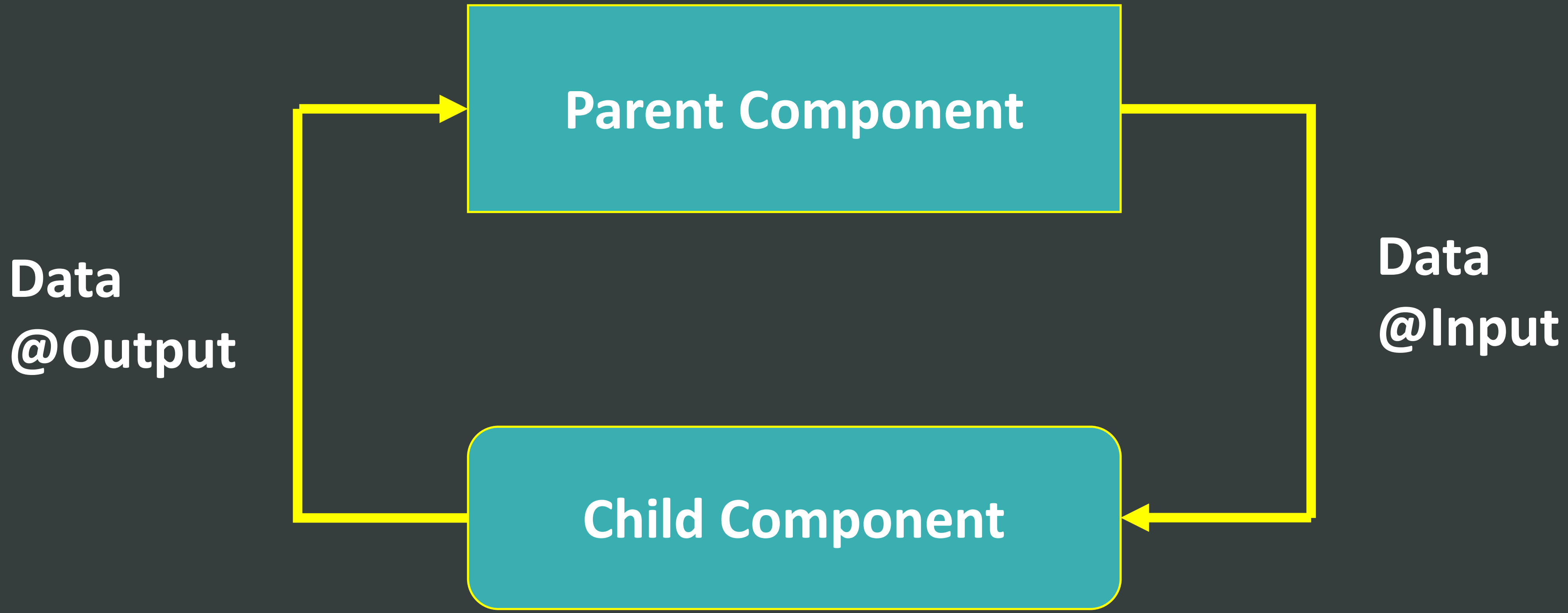
Angular (Part – 3)

PROF. P. M. JADAV
ASSOCIATE PROFESSOR
COMPUTER ENGINEERING DEPARTMENT
FACULTY OF TECHNOLOGY
DHARMSINH DESAI UNIVERSITY, NADIAD

Content

- Component Interaction
- Services
- HTTP Client

Component Interaction



Component Interaction (Parent to Child)

- Data flow from Parent to Child Component (through child selector)

Creation of Child Component

- ng g c child

app.component.ts

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

@Component({
        selector: 'app-root',
        templateUrl: './app.component.html',
        styleUrls: ['./app.component.css']
})
export class AppComponent {
        title = 'app'
        name = "App Component"
}
```

app.component.html

```
<div style="text-align:center">
        <h1>
                Welcome to {{ title }}!
        </h1>

        <app-child [parentData]="name"></app-child>
</div>
```

child.component.ts

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

@Component({
        selector: 'app-child',
        template: `   <h2> Hello {{ parentData }} </h2>        `,
        styles: []
})
export class ChildComponent implements OnInit {
        @Input() public parentData: string;
        constructor() { }
        ngOnInit() {}
}
```

child.component.ts (Alias for parent data)

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

@Component({
        selector: 'app-child',
        template: `   <h2> Hello {{ name }} </h2>      `,
        styles: []
})
export class ChildComponent implements OnInit {
        @Input('parentData') public name
        constructor() { }
        ngOnInit() {}
}
```

Intercept Input Properties in Child Component

```
export class ChildComponent implements OnInit {
        private _name

        @Input()
        set name(name: string) {
                this._name = (name && name.trim()) || '<no name set>';
        }

        get name(): string { return this._name; }

        …
}
```

Component Interaction (Child to Parent)

- Data flow from Child to Parent Component (through events)

child.component.ts (part-1)

```
import { Component, OnInit, Input, Output, EventEmitter }
                                                    from '@angular/core';
~~import {EventEmitter} from 'events';~~

@Component({
        selector: 'app-child',
        template: `<h2> Hello {{ name }} </h2>
        <button (click)="fireEvent()" >Send data </button>`,
        styles: []
})
```

child.component.ts (part-2)

```
export class ChildComponent implements OnInit {
        @Input('parentData') public name
        @Output() public childEvent = new EventEmitter()

        constructor() { }
        ngOnInit() {}
        fireEvent() {
                this.childEvent.emit('Hi from child component')
        }
}
```

app.component.ts (part – 1)

```
import { Component } from '@angular/core';
@Component({
            selector: 'app-root',
           template : `
                     <h1>
                                {{ childMsg }}!
                     </h1>
<app-child (childEvent)="msgFromChild($event)" [name]="name">
              </app-child>
           `,
            styleUrls: ['./app.component.css']
})
```

app.component.ts (part – 2)

```
export class AppComponent {
        name          = " Parent: My name is App Component "
        childMsg    = "child msg not received";

        msgFromChild(msg) {
                this.childMsg = msg
        }
}
```

Component Interaction via Template Reference Variable

counter.component.ts

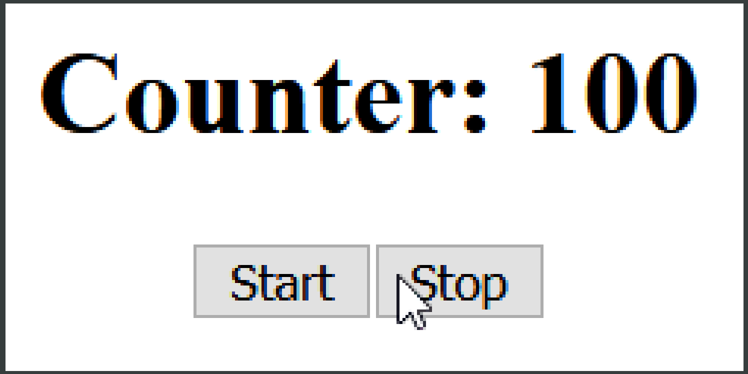
```
export class CounterComponent  {
	seconds = 100
	timerRef

	start() {
		this.timerRef = setInterval( () => {
							this.seconds--
						}, 1000)
	}
	stop() {		window.clearInterval(this.timerRef)	}
}
```

Component Interaction via Template Reference Variable

app.component.ts

```
@Component({
        selector: 'app-root',
        template : `
                <h1>    Counter: {{ timer.seconds }}   </h1>
                <button (click)="timer.start()">Start</button>
                <button (click)="timer.stop()">Stop</button>
                <app-counter #timer></app-counter>
          `

})
export class AppComponent {}
```

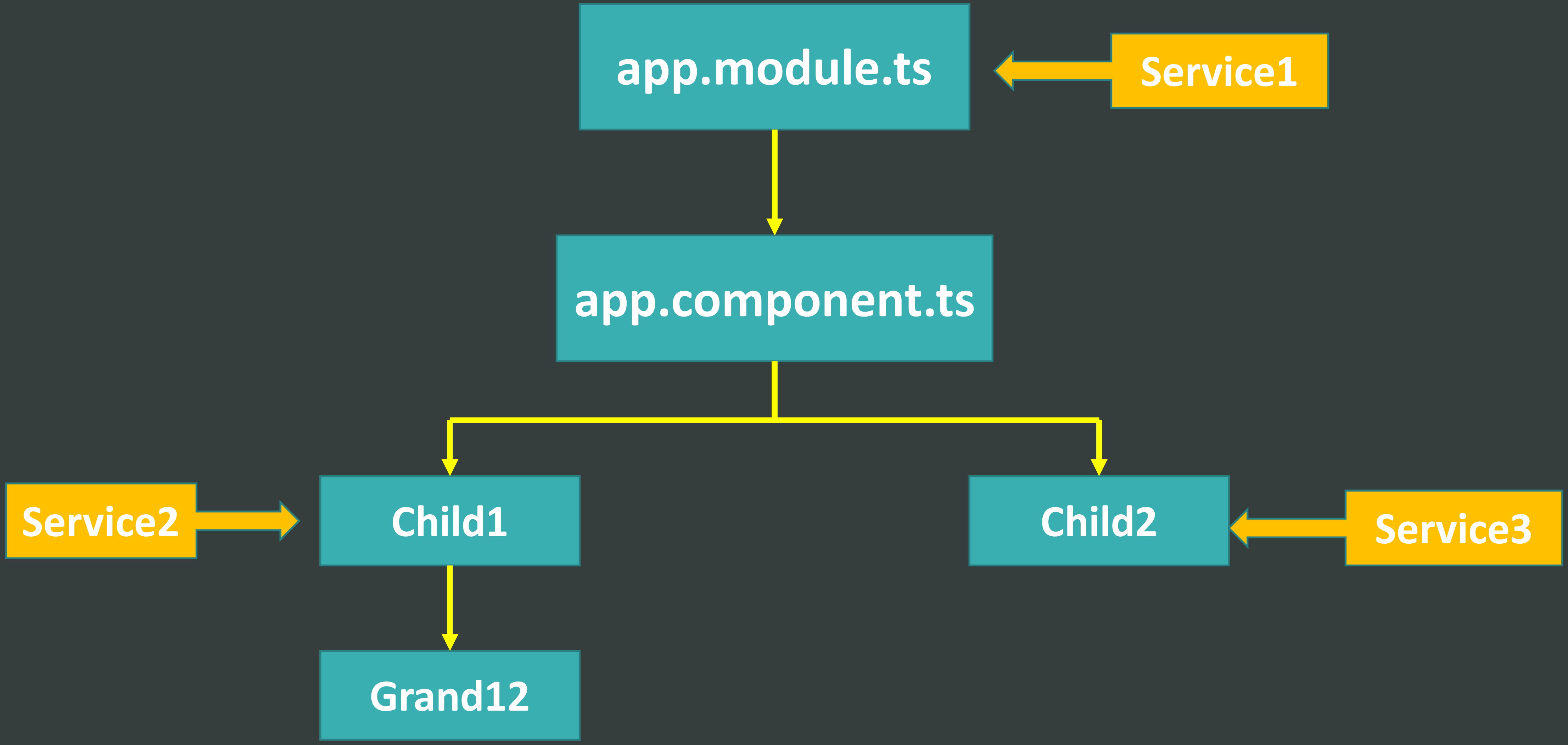
Component Interaction via Template Reference Variable



Services

- Service is a broad category encompassing any value, function, or feature that an app needs
- A service is typically a class with a narrow, well-defined purpose
- It should do something specific and do it well
- Services are a great way to share information among classes that *don't know each other*
- In Angular Dependency Injection (DI) framework is used to inject services

Hierarchical Dependency Injection



Providing Services

- You must register at least one *provider* of any service you are going to use
- The provider can be part of the
  1. service's own metadata through @[Injectable]() decorator (making that service available everywhere)
  2. register providers with specific modules (in the @[NgModule]())
  3. register providers with specific components (in @[Component]())

Providing Services at Root level

- By default, the Angular CLI command ng generate service registers a provider with the root injector for your service by including provider metadata in the @Injectable() decorator.

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

- In this case Angular creates a single, shared instance of Service and injects it into any class that asks for it.
- It also allows Angular to optimize an app by removing the service from the compiled app if it isn't used.

Providing Services at Module Level

- When you register a provider with a specific NgModule, the same instance of a service is available to all components in that NgModule.
- To register at this level, use the providers property of the @NgModule() decorator,

```
@NgModule({
    providers: [
        BackendService,
        Logger
    ],
    …
})
```

Providing Services at Component Level

- When you register a provider at the component level, you get a new instance of the service with each new instance of that component.
- At the component level, register a service provider in the providers property of the @Component() metadata.

```
@Component({
        selector:   'app-test',
        templateUrl: './test.component.html',
        providers:  [ LoggerService ]
})
```

Service Creation

- ng g s employee

g – generate

s – service

employee – name of the service

employee.service.ts (generated)

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

@Injectable(
        // { providedIn: 'root'}
        // If registered at root level, then available everywhere
)

export class EmployeeService {

        constructor() { }
}
```

employee.service.ts (edited)

```
import { Injectable } from '@angular/core';
@Injectable({
        //providedIn: 'root'
})
export class EmployeeService {
        public employees = [
                { id:1, name: "John", designation : "manager"},
                { id:2, name: "Mac", designation : "accountant"},
                { id:3, name: "Tom", designation : "clerk"}    ]
        constructor() { }
        getEmployee() {
                return this.employees
        }
}
```

app.module.ts (Registering the Service)

```
                         …
import { EmployeeService } from './employee.service';

@NgModule({
    …
    providers: [EmployeeService],
    bootstrap: [AppComponent]
})
export class AppModule { }
```

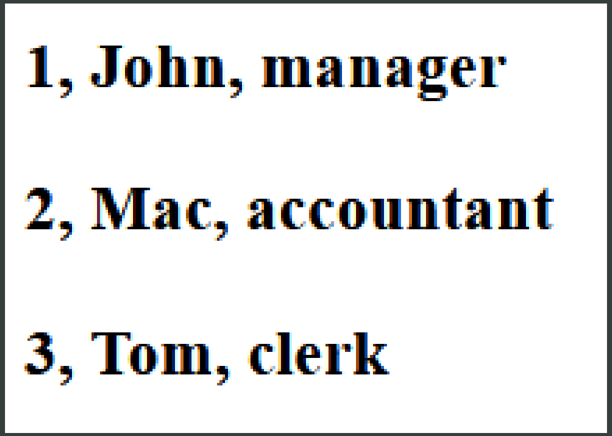
app.component.ts (part-1) (Using the Service)

```
import { EmployeeService } from './employee.service';

@ Component({
        selector: 'app-root',
        template : `
                <h3 *ngFor="let emp of empList">
                   {{ emp.id }}, {{ emp.name }}, {{emp.designation}}
                </h3>
        `,
})
```

app.component.ts (part-2) (Using the Service)

```
export class AppComponent implements OnInit {
        public empList = [];
        public empService : EmployeeService
        constructor(empServ:EmployeeService) {
                this.empService = empServ;
        }
        ngOnInit() {
                this.empList = this.empService.getEmployee();
        }
}
```

Service Example Output

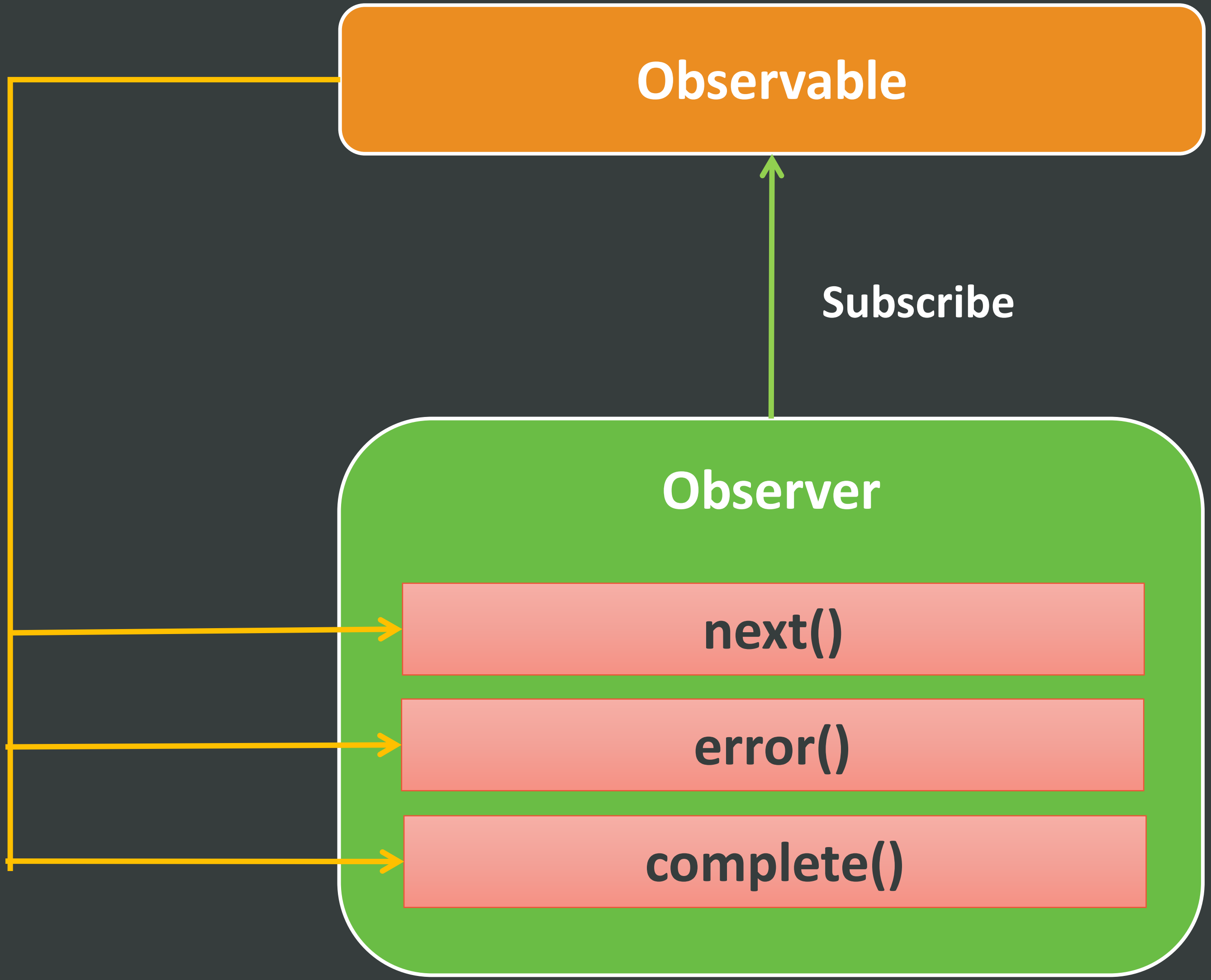


```
1, John, manager

2, Mac, accountant

3, Tom, clerk
```

HttpClient

- HttpClient is Angular's mechanism for communicating with a remote server over HTTP.

Observables, Observers and Subscriptions



Observables

1. Observable - is a wrapper around data source (stream of data source that continuously emits data).
2. Observer - executes some piece of code whenever we receive some data, error or if observable reports that this is done.
   - implements three methods next() , error() and complete()
3. Subscription - Observer is connected to an observable through subscription.
   - Each Subscription causes an independent execution of the observable.

There is a contract between observer and observable so that Observable knows which method to call.

Observables

- provide support for passing messages between publishers and subscribers
- offer significant benefits for event handling, asynchronous programming, and handling multiple values
- are declarative—that is, you define a function for publishing values, but it is not executed until a consumer subscribes to it. The subscribed consumer then receives notifications until the function completes, or until they unsubscribe
- An observable can deliver multiple values of any type—literals, messages, or events, depending on the context.

Observables

- The API for receiving values is the same whether the values are delivered synchronously or asynchronously.
- Setup and teardown logic are both handled by the observable, your application code only needs to worry about subscribing to consume values, and when done, unsubscribing.
- The stream can be keystrokes, an HTTP response, or an interval timer, the interface for listening to values and stopping listening is the same

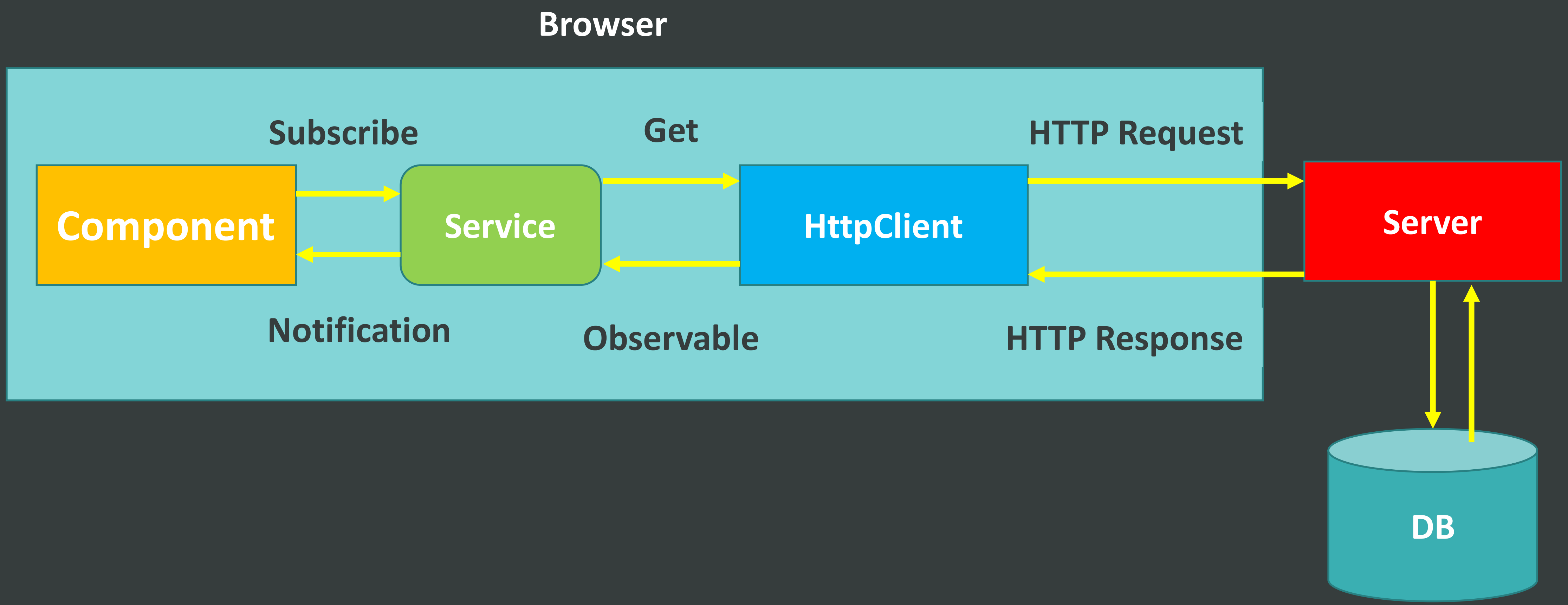
RxJS

- Reactive programming is an asynchronous programming paradigm concerned with data streams and the propagation of change.
- RxJS (Reactive Extensions for JavaScript) is a library for reactive programming using observables that makes it easier to compose asynchronous or callback-based code.
- RxJS provides an implementation of the Observable type, which is needed until the type becomes part of the language and until browsers support it.

RxJS

- The library also provides utility functions for creating and working with observables
- These utility functions can be used for:
  - Converting existing code for async operations into observables
  - Iterating through the values in a stream
  - Mapping values to different types
  - Filtering streams
  - Composing multiple streams

HTTP Mechanism



HTTP Mechanism

1. HTTP Get request from EmployeeService
2. Receive the observable and cast it into an employee array
3. Subscribe to the observable from AppComponent
4. Assign the employee array to a local variable

app.module.ts

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

@NgModule({
        declarations: [   AppComponent  ],
        imports: [   BrowserModule,   HttpClientModule  ],
        providers: [EmployeeService],
        bootstrap: [AppComponent]
})

export class AppModule { }
```

assets/emp.json

```
[
        { "id":1, "name": "John", "designation" : "manager"          },
        { "id":2, "name": "Mac", "designation" : "accountant"      },
        { "id":3, "name": "Tom", "designation" : "clerk"                 },
        { "id":4, "name": "Jane", "designation" : "project manager"}
]
```

employee.ts

```
export interface IEmployee {
        id : number,
        name : string,
        designation : string
}
```

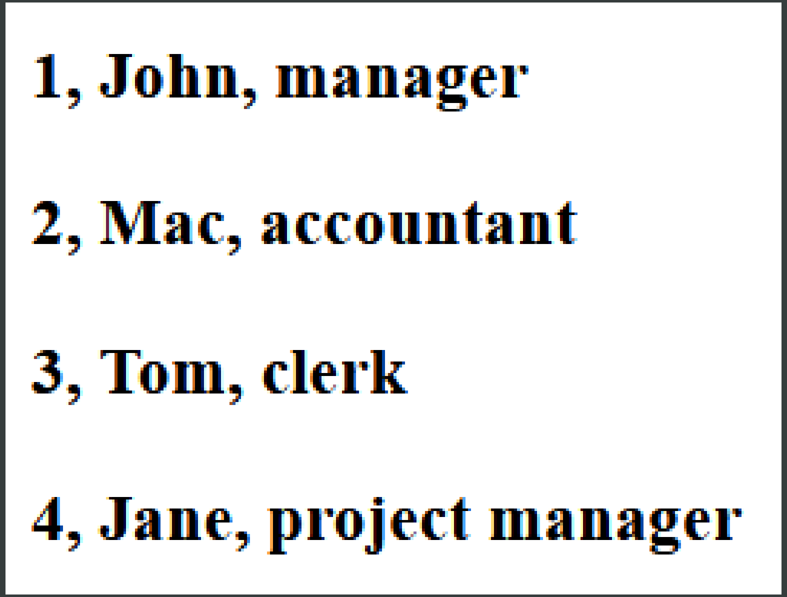
employee.service.ts

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { IEmployee } from './employee'
import { Observable } from 'rxjs';

@Injectable()
export class EmployeeService {
        private _url :string = "/assets/emp.json"

        constructor(private http: HttpClient) {   }

        getEmployees():Observable<IEmployee[]> {
                return this.http.get<IEmployee[]>(this._url)
        }
}
```

app.component.ts (Class)

```
export class AppComponent implements OnInit {
        public empList = [];

        constructor(public empServ:EmployeeService) { }

        ngOnInit() {
                this.empServ.getEmployees()
                        .subscribe(data => this.empList = data);
        }
}
```

Output



```
1, John, manager

2, Mac, accountant

3, Tom, clerk

4, Jane, project manager
```

References

- https://angular.io/docs