

## งานบริการ และ Observable

บทนี้เน้นการสร้างงานบริการ ที่นำไปใช้ในคอมโพเนนต์ต่าง ๆ โดยสมมติให้งานบริการ ให้บริการข้อมูลที่มาจากรายการ การอ่านงานบริการนี้เป็นการอ่านข้อมูลโดยตรง หรือที่เรียกว่า ซินโครนัส (Synchronous) ยังมีการอ่านข้อมูลอีกแบบที่ตรงกันข้ามคือ อซินโครนัส (Asynchronous) โดยเรียกทำงานผ่านคลาส Observable ซึ่งทำงานในลักษณะ การสมัครสมาชิกรับข้อมูล (subscriber) กับผู้ให้บริการข้อมูล (publisher) ผู้สมัครจะได้รับข้อมูล ซึ่งในบทต่อไปจะนำไปใช้งานร่วมกับ HttpClient ในการดึงข้อมูลผ่านเว็บเซิร์ฟเวอร์

### มูลเหตุแห่งการสร้างงานบริการ

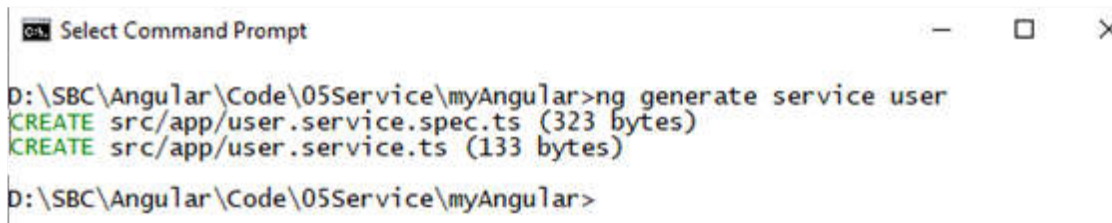
ใช้ในการดำเนินการกับข้อมูล ไม่ว่าจะเป็นการอ่าน ลบ เพิ่ม ปรับปรุง แทนที่จะให้คอมโพเนนต์ทำงานแทน เพราะหน้าที่หลักของคอมโพเนนต์คือการ แสดงผล จึงแยกหน้าที่ไม่เกี่ยวกับการแสดงผลออกไปให้งานบริการทำงานด้านการดำเนินการกับข้อมูล นอกจากนี้งานบริการที่ให้บริการด้านอื่นๆ ที่ไม่ใช่หน้าที่หลักของคอมโพเนนต์ เช่น งานด้านการรักษาความปลอดภัย งานด้านส่งอีเมล งานด้านให้บริการเฉพาะด้าน ก็สามารถสร้างเป็นงานบริการได้ นอกจากการสร้างงานบริการเพื่อเรียกใช้บริการแล้ว งานบริการก็สามารถที่เรียกใช้งานบริการด้วยตัวเองได้ด้วย

### สร้างงานบริการ

เพื่อเป็นการจำลองการสร้างงานบริการ เพื่อดำเนินการกับข้อมูล User จึงเริ่มต้นด้วยการสร้างงานบริการชื่อ user ในการสร้างนี้ ใช้ Angular CLI สร้างบริการขึ้น

จากตัวอย่างต่อไปนี้ ใช้ CLI ในโฟลเดอร์ myAngular ซึ่งเป็นโฟลเดอร์ที่โปรแกรมของ Angular ที่สร้างไว้แล้ว ในชื่อ myAngular สร้างงานบริการชื่อ user ด้วยคำสั่ง

```
ng generate service user
```



```

C:\> Select Command Prompt

D:\SBC\Angular\Code\05Service\myAngular>ng generate service user
CREATE src/app/user.service.spec.ts (323 bytes)
CREATE src/app/user.service.ts (133 bytes)
D:\SBC\Angular\Code\05Service\myAngular>
```

รูป 1 การสร้างบริการ user

ในรูปนี้ สร้างบริการจาก โฟลเดอร์ 05Service ซึ่งขึ้นอยู่กับว่า สร้างเว็บแอปฯ ที่โฟลเดอร์ใด ก็ให้สร้างบริการที่โฟลเดอร์นั้น

หลังจากสร้างงานบริการ user จะได้ ไฟล์ สำคัญคือ user.service.ts ในไฟล์นี้ คือคลาส UserService มีการประกาศนำเข้า Injectable ดังตัวอย่างต่อไปนี้

#### Code 1. src/app/user.service.ts

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

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

```
})
```

```
export class UserService {  
  constructor() { }  
}
```

ด้วยการลงทะเบียนด้วย @Injectable เพื่อให้ใช้ได้ทั้งแอปพลิเคชัน ด้วยการระบุ providedIn: root ทำให้คลาสอื่นในโปรแกรมนี้อีกเรียกใช้งานได้ และทำให้ไม่จำเป็นต้องลงทะเบียนในโมดูล (app.module.ts) ในส่วนอาร์เรย์ providers อีก

## DI (Dependency Injection)

DI เป็นแนวทางหลักที่แองกูลาร์เลือกใช้งานบริการ ในความหมายของ DI คือการส่งต่อออบเจกต์ เข้ามาทำงาน โดยออบเจกต์ที่ส่งมานั้นเป็นอิสระเพราะเกิดขึ้นจากภายนอก (ไม่ได้ถูกสร้างจากภายในผู้ใช้บริการ) ยกตัวอย่างเช่นออบเจกต์ User ถูกส่งเข้ามาทำงานในคลาส MyUser

```
class MyUser{  
  constructor(public users: User[]){}  
}
```

จากคลาส MyUser รับค่าสมาชิก users เป็นอาร์เรย์ของออบเจกต์ User เข้ามาใช้งานผ่านคอนสตรัคเตอร์ โดยที่ users เป็นออบเจกต์ที่ถูกสร้างจากที่อื่น ไม่ได้ถูกสร้างภายใน MyUser จึงถือว่า users เป็น DI

กรณีที่สร้างออบเจกต์ users ในคลาส MyUser จะไม่ถือว่าเป็น DI เช่นตัวอย่างต่อไปนี้ มีการสร้างออบเจกต์ users ภายในคอนสตรัคเตอร์ ทำให้ users ไม่เป็นอิสระ เพราะเมื่อออบเจกต์ MyUser ถูกทำลาย ก็จะทำให้ออบเจกต์ users หายไปด้วย

```
class MyUser{  
  public users: User[];  
  constructor(){  
    this.users = [  
      new User(1, 'Pol', 'L.', 'pol@gmail.com'),  
      new User(2, 'Mon', 'T.', 'Mon@gmail.com')];  
  }  
}
```

งานบริการของ แองกูลาร์ใช้รูป DI ทั้งหมด รวมทั้งส่วนประกอบอื่น ๆ ก็ใช้รูปแบบนี้เช่นกัน งานบริการที่เป็นอิสระนี้มีลักษณะ เป็นออบเจกต์เดี่ยว (singleton) เพราะไม่ได้สร้างขึ้นมามากหลายรอบ หรือสร้างเพียงครั้งเดียว และนำไปลงทะเบียนใช้ทั้งแอปพลิเคชัน การลงทะเบียนจะลงทะเบียนกับโมดูลหลักในรูปแบบ DI ผ่านตัวแปร providers: [ ] หรือจะลงทะเบียนผ่าน providedIn: 'root' ผ่านในคลาสนงานบริการเองก็ได้ ถือว่าได้ผลเหมือนกัน

## จำลองข้อมูลในงานบริการ

สร้างคลาส User จะสร้างโดยตรงจาก CLI หรือสร้างขึ้นเองจากสร้างไฟล์ใหม่ก็ได้ แต่ถ้าสร้างจาก CLI จะเขียนดังนี้

```
ng generate class user
```

คลาส User กำหนดให้มีข้อมูลอย่างง่าย เพียง 2 สมาชิก คือ id, fname, lname, email เมื่อต้องการสร้างข้อมูลตัวอย่าง ให้ใช้ข้อมูลสมมติ user-data.ts ที่นำข้อมูลมาจากคลาส User โดยการกำหนดเป็นค่าคงที่ในชื่อ USERS ที่เป็นอาร์เรย์ของออบเจกต์ user ตัวสองตัวอย่างต่อไปนี้

#### Code 2. src/app/user.ts

```
export class User {
  constructor(
    public id:number,
    public fname:string,
    public lname:string,
    public email:string)
  {}
}
```

#### Code 3. src/app/user-data.ts

```
import { User } from './user';
export const USERS: User[] = [
  new User(1,'Pol', 'L.', 'pol@gmail.com'),
  new User(2,'Mon', 'T.', 'Mon@gmail.com'),
  new User(3,'Tee', 'F.', 'Tee@gmail.com'),
  new User(4,'Kon', 'A.', 'Kon@gmail.com'),
  new User(5,'Jel', 'T.', 'Jel@gmail.com'),
];
```

สำหรับคลาส UserService ต้องการให้ ข้อมูลจากค่าคงที่ USERS และค่าคงที่เก็บข้อมูลในรูปอาร์เรย์ ที่ประกอบไปด้วยคลาส User ดังนั้นจึงต้องนำเข้าคลาส User และค่าคงที่ USERS เพิ่มเติมจากเดิม

#### Code 4. src/app/user.service.ts

```
import { User } from './user';
import { USERS } from './user-data';
```

และสร้างเป็นฟังก์ชัน getUsers( ) เพื่ออ่านค่าข้อมูล Users ไว้ในคลาส UserService ด้วย ดังเป็นเป้าหมายของคลาสนี้ ที่ต้องการใช้ให้ดำเนินการกับฐานข้อมูล

#### Code 5. src/app/user.service.ts

```
getUsers(): User[] { return USERS; }
```

#### นำงานบริการไปใช้กับคอมโพเนนต์

เพื่อที่จะให้ข้อมูล จาก USERS แสดงผลได้ในหน้าเว็บ จึงควรสร้างคอมโพเนนต์ ขึ้นมาใหม่ โดยให้ชื่อว่า users ดังให้ Angular CLI สร้างขึ้นมาให้ดังตัวอย่างรูปต่อไปนี้

```
ng generate component users
```

เมื่อสร้างคอมโพเนนต์ใหม่มาแล้ว และต้องการให้แสดงผลเป็นหน้าแรก ก็ต้องให้ไฟล์ app.component.html ประกาศอิลีเมนต์ของคอมโพเนนต์ใหม่ด้วย ดังแก้ไขการแสดงผลได้ใหม่คือ:

#### Code 6. src/app/app.component.html

```
<div style="text-align:center">
```

```

    <h1>
      Welcome to {{ title }}!
    </h1>
  </div>
<h2>User List</h2>
<app-users></app-users>

```

ในส่วน selector เดิมทั้งหมด ในชื่อ app-root จากไฟล์ app.component.ts โดยที่ <app-users> คือ คอมโพเนนต์เพิ่มเติม ของไฟล์ users.component.ts ซึ่งจะเป็นการเริ่มต้นหน้าเว็บใหม่ การประกาศอัสเม้นท์ ของคอมโพเนนต์ สามารถประกาศได้หลาย อัสเม้นท์ได้ เช่น การประกาศอัสเม้นท์ <app-product> เพิ่มอีกหนึ่งอัสเม้นท์ ทำให้การแสดงผลมี สองส่วนอัสเม้นท์ของคอมโพเนนต์



รูป 2 หน้าเว็บเริ่มต้นใหม่

ต่อมา เมื่อต้องการให้ คอมโพเนนต์ user เรียกใช้บริการ UserService ได้ จึงต้องประกาศ นำเข้าบริการนี้เพิ่มเติม ด้วย และในคอมโพเนนต์นี้มีการอ้างอิง ชนิดข้อมูล User จึงต้องนำเข้าคลาส User ด้วย

**Code 7. src/app/users/user.component.ts**

```

import { User } from '../user';
import { UserService } from '../user.service';

```

การนำเข้า UserService นี้ ถือว่า นำไปใช้ได้ทันที เพราะถือเป็นออบเจกต์ที่ได้ลงทะเบียนไว้แล้ว ไม่จำเป็นต้องสร้าง เป็นออบเจกต์ขึ้นมาใหม่อีกครั้ง และให้สังเกตว่าใช้ เส้นทางคลาส UserService ด้วยจุด 2 จุดติดกัน เพราะต้องออกจาก โฟลเดอร์ไปหนึ่งโฟลเดอร์ ต่อมาให้สร้าง สมาชิก users เป็นข้อมูลอาร์เรย์ เพื่อเตรียมรับข้อมูลจากงานบริการ UserService

**Code 8. src/app/users/users.component.ts**

```

users: User[];

```

**เพิ่มสมาชิกให้คอนโพเนนต์**

การสร้างสมาชิกเพิ่ม ให้สร้างแบบทางลัด คือสร้างในตัวแปรเข้าของคอนสตรัคเตอร์ แต่ให้ใช้ได้เฉพาะคลาสตัวเอง ดังนั้นจึงมีค่าการเข้าถึงแบบ private ให้กับคลาส UsersComponet

**Code 9. src/app/users/users.component.ts**

```

constructor(private userService: UserService) { }

```

เพิ่มสมาชิกประเภทฟังก์ชัน ชื่อ getUsers( ) เพื่อให้ทำการดึงข้อมูลจากงานบริการ ที่เพ่งใส่ในคอนสตรัคเตอร์ การ ดึงข้อมูลนี้ เรียกฟังก์ชัน getUsers( ) ในงานบริการ UserService

**Code 10.** src/app/users/users.component.ts

```
getUsers(): User[] {return this.userService.getUsers();}
```



ในระหว่างการทำงาน ให้บันทึกไฟล์ต่าง ๆ ที่สร้าง และให้สังเกตว่า ที่หน้าต่าง CLI มีการแจ้งความผิดพลาดอะไรบ้าง ให้ทำการแก้ไข ถ้ามีความผิดพลาด ตามที่ข้อมูลแจ้งความผิดพลาด ก่อนการดำเนินการต่อไป

### ngOnInit

เมื่อเราสร้างสมาชิกใหม่ ชื่อ userService แล้วในคอนสตรัคเตอร์ การที่จะให้คอนสตรัคเตอร์ทำงานต่อด้วยการเรียกฟังก์ชัน getUsers() อีกก็ย่อมทำได้ แต่เป็นวิธีการที่ไม่ดี เพราะมีอีกฟังก์ชันหนึ่ง คือ ngOnInit() ของคอมโพเนนต์ ที่ทำหน้าที่กำหนดค่าเริ่มต้นหลังจากที่คอมโพเนนต์เริ่มแสดงผล ซึ่งเหมาะที่จะเรียกใช้บริการต่าง ๆ เช่น บริการ HTTP อย่างในกรณีตัวอย่างนี้ เรียกบริการ UserService ผ่านสมาชิกชื่อ getUser() ของคลาสตัวเอง (UsersComponent) และกำหนดค่าให้สมาชิกอีกตัวหนึ่งคือ users ซึ่งเป็นอาร์เรย์ของ User[]

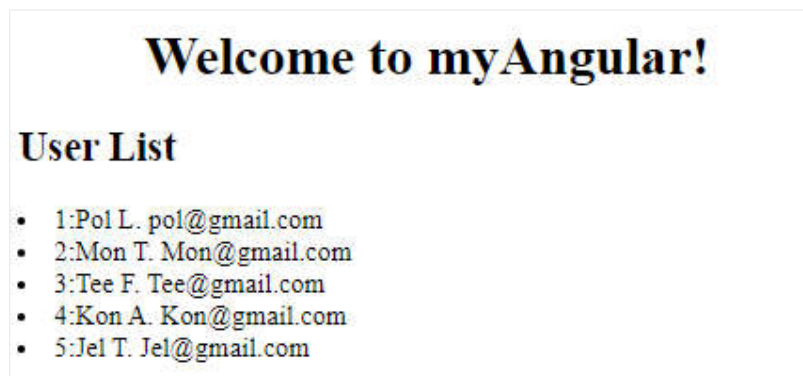
**Code 11.** src/app/users/users.component.ts

```
ngOnInit() {  
  this.users = this.getUsers();  
}
```

ตอนนี้ข้อมูลพร้อมบริการแสดงผลแล้ว จาก การใช้ฟังก์ชัน ngOnInit() ซึ่งจะได้ค่าไปเก็บที่ this.users ค่านี้เป็นอาร์เรย์ ซึ่งคอมโพเนนต์ จากไฟล์ users.components.html เรียกผ่านชื่อนี้ได้โดยตรง โดยไม่ต้องมีคำว่า this นำหน้า ดังตัวอย่างต่อไปนี้เรียก แสดงผลเฉพาะ ข้อมูลแรก แสดง id และ name ด้วยการผูกข้อมูลทางเดียว

**Code 12.** src/app/users.component.html

```
<p>  
  ID:{{users[0].id}} Name:{{users[0].fname}}  
</p>
```



รูป 3 การแสดงข้อมูลทั้งหมดของอาร์เรย์ users

หรือต้องการให้แสดงผลทั้งหมด ก็ให้ใช้คำสั่ง \*ngFor เพื่อวนซ้ำในอาร์เรย์ users ดังตัวอย่างต่อไปนี้

**Code 13.** src/app/users.component.html

```
<p>
```

```
<li *ngFor="let u of users">
  {{u.id}}:{{u.fname}} {{u.lname}} {{u.email}}</li>
</p>
```

### การดำเนินการแบบอซิงโครนัส

การเรียกใช้บริการที่ผ่านมาใน ngOnInit ถือว่าเป็นการเรียกใช้บริการโดยตรงและทำงานทันที แต่ในความเป็นจริง การเรียกใช้บริการมักทำผ่าน HTTP ซึ่งข้อมูลอาจมีความล่าช้า การได้ข้อมูลเพื่อแสดงผลในคอมโพเนนต์จึงหยุดค้างไปชั่วขณะ ได้วิธีการที่ดีคือให้ดำเนินการแบบ อซิงโครนัส (Asynchronous)

การดำเนินการแบบอซิงโครนัส จะได้ข้อมูลไม่ทันทีทันใด ไม่แน่นอนขึ้นอยู่กับระบบสื่อสาร การทำงานทำได้ผ่านการเรียกฟังก์ชันเรียกฟังก์ชัน (callback) ซึ่งจะคืนค่าเป็นกระแสข้อมูลที่ส่งผ่านการสื่อสารแบบ HTTP แบบอซิงโครนัสที่สังเกตการณ์ได้ว่าดำเนินไปถึงขั้นตอนใด (Observable) และต่อไปการอ่านค่าข้อมูลต่างๆ ต้องใช้งานแบบนี้เสมอ โดยเฉพาะกับข้อมูลที่สื่อสารผ่านระบบอินเทอร์เน็ต ซึ่งแน่นอนว่าเนื้อหาต่อไปนี้นี้สำคัญมาก

### ตัวให้สังเกตการณ์ (Observable)

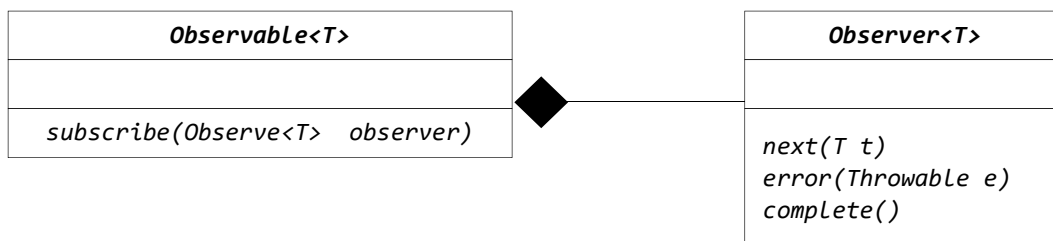
ตัวให้สังเกตการณ์ ทำงานคล้าย การสมัครสมาชิกรับข้อมูล (subscriber) กับผู้ให้บริการข้อมูล (publisher) ผู้สมัครจะได้รับข้อมูล จากการทำงานของฟังก์ชันที่กำหนดจากผู้ให้บริการ การรับข้อมูลจะเสร็จสิ้นเมื่อฟังก์ชันนี้ทำงานเสร็จ หรือผู้สมัครยกเลิกสมัคร

ตัวให้สังเกตการณ์ เป็นรูปแบบการออกแบบอย่างหนึ่ง ที่เก็บรายการเรื่องต่างๆ (subject) ที่เป็นอิสระในต่อละเรื่อง และมีกรลงทะเบียนสมัคร (subscribe) จากผู้สังเกตการณ์ (Observer)

ตัวอย่างจาก รูป 1 ผู้สังเกตการณ์จะต้องการรับสมัครข้อมูลในเรื่องต่างๆ เช่น Observer 1 ต้องการรับสมัครเรื่อง Subject 1 จาก Observable และยังมีเรื่อง Subject 2 ที่มี Observer 2 รับสมัคร ทั้ง Subject 1 และ 2 เป็นอิสระต่อกันที่ถูกเก็บรวมอยู่ใน Observable



รูป 1 รูปแบบ Observable



รูป 2 รูปแบบ Observable บนแผนภาพคลาส/อินเทอร์เฟซ

จากรูปที่ 2 Observable ประกอบด้วยอินเทอร์เฟซ Observer<T> ภายในอินเทอร์เฟซนี้มีเมธอด next, error, complete ทั้งสามตัวใช้จัดการข้อมูลที่สังเกตการณ์

next ใช้จัดการแต่ละข้อมูล error ใช้จัดการแจ้งความผิดพลาด เมธอดนี้เป็นทางเลือกไม่ใช่ก็ได้ complete ใช้จัดการแจ้งผลสำเร็จ เมธอดนี้เป็นทางเลือกไม่ใช่ก็ได้

ข้อมูลที่ส่งผ่าน Observable สามารถส่งข้อมูลชนิดใดก็ได้ เช่น ข้อความ ออบเจกต์ หรือแม้กระทั่งอีเวนต์ (event) การส่งข้อมูลทำได้ทั้งในลักษณะ ซินโครนัส และอซินโครนัส

Observable เป็นคลาสสำคัญคลาสหนึ่งในไลบรารี rxjs ซึ่งมักใช้งานมากกับงานบริการ HTTP ผ่านฟังก์ชัน HttpClient ในตัวอย่างนี้จะใช้ คลาส Observable เพื่ออ่านงานบริการ UserService ซึ่งสมมุติว่าจำลองมาจากเซิร์ฟเวอร์ที่หนึ่ง ในการใช้งานคลาสนี้ จะต้องประกาศนำเข้าคลาสจากไลบรารี rxjs และนอกจากคลาส Observable และยังต้องการ ฟังก์ชัน of() เพื่อคืนค่าของออบเจกต์ Observable

---

#### Code 14. src/app/user.service.ts

```
import { Observable, of } from 'rxjs';
```

เมื่อนำเข้าคลาส และฟังก์ชัน (ใน JavaScript คลาส และฟังก์ชัน มีฐานะเหมือนกัน) ที่ต้องการใช้สำหรับการดำเนินการแบบ อซินโครนัส แล้ว ก็สามารถนำใช้อ่านข้อมูลจากเซิร์ฟเวอร์ได้ แต่ในตัวอย่างนี้จำลองไว้ว่าค่าคงที่ USERS แทนข้อมูลจากเซิร์ฟเวอร์

จากเดิมที่เคยอ่านค่าคงที่โดยตรงผ่านฟังก์ชัน getUser() ให้เปลี่ยนมาใช้เรียกแบบใหม่แบบ อซินโครนัส ด้วยการเรียกผ่านฟังก์ชัน getUserObservable() ซึ่งจะคืนค่าอาร์เรย์ Observable ด้วยฟังก์ชัน of(Array<T>)

---

#### Code 15. src/app/user.service.ts

```
getUserObservable(): Observable<User[]> {  
    return of(USERS);  
}
```

#### สร้างตัวให้สังเกตการณ์ (Observable)

การใช้ of(USERS) ถือว่าเป็นการสร้างออบเจกต์ Observable อย่างหนึ่ง แทนการสร้างโดยตรงจาก จากคำสั่ง new Observable( (observer)=>{ } ) ด้วยการใช้ of() นี้ จะคืนค่าข้อมูลที่รับสมัคร ระบุตามตัวแปรเข้า ในตัวอย่างนี้คือ อาร์เรย์ของออบเจกต์ User ซึ่งคือ observer.next ตามชื่อตัวแปรเข้า

ตัวอย่างต่อไปนี้จะใช้ of(USERS) แต่เขียน โดยการสร้างคลาสใหม่ ซึ่งจะดูยาวขึ้น แต่กับ สามารถเขียนกำกับรายละเอียดได้มากกว่า โดยสามารถตรวจสอบก่อนว่ามีข้อมูลมากกว่าศูนย์ แต่ถ้าไม่ใช่ ให้ส่งเป็นความผิดพลาดแทน

---

#### Code 16. src/app/user.service.ts

```
getUserObservable(): Observable<User[]>{  
    const myObservable = new Observable<User[]>(observer=>{  
        if(USERS.length>0){  
            observer.next(USERS);  
            observer.complete();  
        }  
        else {  
            observer.error("User is empty");  
        }  
    });  
}
```

```

    return myObservable;
}

```

อย่างไรก็ตามวิธีแรกใช้สะดวกกว่า (การใช้ of(USERS)) เพราะความผิดพลาด หรือด้านอื่นใด ควรเป็นที่ของผู้สังเกตการณ์ ให้จัดการด้วยตัวเองดีกว่า

จากตัวอย่างที่ผ่านมารับข้อมูลเป็นออบเจกต์ USER แต่จากที่กล่าวก่อนหน้านี้ ตัวสังเกตการณ์ สามารถรับข้อมูลได้ทุกประเภท ตัวอย่างต่อไปนี้จะแสดงตัวอย่างการรับข้อมูลประเภทอีเวนต์ (event) โดยสมมติให้รับเหตุการณ์ เคลื่อนเมาท์ไปยังอีลีเมนต์ <p> โดยใช้การอ่านด้วย document.getElementsByTagName("p") การอ่านด้วยวิธีนี้จะได้ค่า <p> ในรูปอาร์เรย์ ซึ่งถ้าหน้าเว็บมี <p> เพียงตัวเดียว ก็จะได้ ค่าดัชนีที่ศูนย์ หรืออาร์เรย์ที่ศูนย์นั่นเอง ในตัวอย่างต่อไปนี้จะไม่ได้สร้างในคลาส UserService แต่สร้างเป็นฟังก์ชันหนึ่งในคลาส UserComponent และต้องนำเข้า fromEvent ด้วย

```
import { fromEvent } from 'rxjs';
```

**Code 17. src/app/users/user.component.ts**

```

observableEventConstant() : Observable<Event>{
    const el = document.getElementsByTagName("p");
    const mouseMoves = fromEvent(el[0], 'mousemove');
    return mouseMoves;
}

```

การสร้างอีเวนต์ที่รับอ่านค่ามาจากอีลีเมนต์ <p> นี้จะเห็นว่าในหน้า HTML มีค่าตายตัวอยู่แล้วจึงสามารถอ่านได้โดยไม่มีปัญหา แต่เมื่อใช้อ่านกับค่า ที่ไดนามิก เช่น <li> ซึ่งมาจากการวนซ้ำอ่านของ \*ngFor และการวนซ้ำนี้ก็มาจากการอ่านของ Observable ซึ่งได้ค่าไม่ทันทีทันใด การอ่านแบบนี้จะไม่สามารถสร้างเหตุการณ์ให้รับสมัครบริการได้ในทันที การแก้ไขจะต้องใช้การเวลาระยะหนึ่งให้แน่ใจว่าได้ข้อมูล <li> ครบก่อน

**Code 18. src/app/users/user.component.ts**

```

observableEventDynamic():Observable<Event>{
    const el = document.getElementsByTagName("li");
    const mouseOver = fromEvent(el[0], 'mouseover');
    return mouseOver;
}

```

### สมัครรับบริการ

เมื่องานบริการแบบใหม่ได้สร้างขึ้นมาแล้วในแบบ Observable คอมโพเนนต์จากที่เคยเรียกใช้บริการแบบอ่านค่าโดยตรงจาก user-data.ts ก็ปรับมาเรียกใช้แบบ การสมัครรับข้อมูลจาก Observable แทน

การให้คอมโพเนนต์ที่ใช้บริการแบบ สมัครรับข้อมูลจาก Observable ใช้ฟังก์ชัน subscribe( ) ในตัวอย่างนี้ผู้ให้บริการใช้ฟังก์ชันชื่อ getUserObservable( ) การสมัครรับข้อมูลจึงต้องลงทะเบียนรับสมัครกับฟังก์ชันนี้ ฟังก์ชัน subscribe( function( ) ) ซึ่งเป็นฟังก์ชันเรียกฟังก์ชัน (callback function) ดังนั้นแล้ว คอมโพเนนต์ user เขียนฟังก์ชัน getUser( ) ใหม่ดังนี้

**Code 19. src/app/users/user.component.ts**

```

getUserObservable():void{
    this.userService
        .getUserObservable()

```



```

        .subscribe(users => this.users = users);
    }

```

ภายในฟังก์ชัน subscribe() มีการกำหนดค่าให้ this.user ให้เท่ากับ ค่าที่อ่านได้จาก Observable ซึ่งคือค่า users ในรูปแบบการเขียนเชิงฟังก์ชัน หรือแลมบ์ดา (lambda)

นอกจากนี้ ในฟังก์ชัน ngOnInit() จากเดิมที่รับข้อมูลจากการเรียกโดยตรงจากฟังก์ชัน getUser() ให้เปลี่ยนมาเป็นเรียกฟังก์ชัน getUserObservable() ที่เพิ่งสร้างก่อนหน้านี้ ดังแก้ไข ข้อมูลใหม่คือ:

**Code 20. src/app/users/user.component.ts**

```

ngOnInit() {
    //this.users = this.getUser();
    this.getUserObservable();
}

```

เมื่อบันทึกการเปลี่ยนโปรแกรมแล้ว จะพบว่า การแสดงผลของ หน้า users.component.html จะยังคงแสดงเหมือนเดิม

อีกวิธีหนึ่ง ที่จะสร้างตัวรับสมัครข้อมูล โดยการกำหนดค่ารับแจ้งข้อมูลเตือนอื่น ๆ นอกจากรับข้อมูลที่ขอรับบริการ อย่างเดียว เช่น การแจ้งความผิดพลาด การแจ้งผลการสำเร็จของการได้รับข้อมูล ตัวอย่างต่อไปนี้จะทำงานได้เหมือนกับที่ผ่านมา

**Code 21. src/app/users/user.component.ts**

```

getUserObservable():void{
    const myObserver = {
        next: x =>this.users = x,
        error: err=>console.log("Observer got an error"),
        complete:()=>console.log("Completed notification")
    };

    this.userService
        .getUserObservable()
        //.subscribe(users => this.users = users);
        .subscribe(myObserver);
}

```

ในตัวอย่างนี้ next ใช้เพื่ออ่านข้อมูลอาร์เรย์ของ Users แล้วกำหนดให้ค่าเท่ากับตัวแปร users ของคลาสนี้ มีการรับแจ้งผลความผิดพลาด ด้วย error เพื่อรับการแจ้งผลการทำงานผิดพลาด และ complete เพื่อรับการแจ้งผลการทำงานที่สำเร็จ

หรือจะเขียนอย่างย่อ โดยรวม myObserver ที่เขียนก่อนหน้านี้ รวมอยู่ฟังก์ชัน subscribe() ไปได้เลย โดยไม่ต้องเขียนไม่ต้องเขียนแบบมีตัวแปร myObserver

**Code 22. src/app/users/user.component.ts**

```

getUserObservable():void{
    this.userService
        .getUserObservable()
        .subscribe( x=> this.users = x,
                    err=>console.log("Observer got an error."),
                    ()=>console.log("Completed notification")
        );
}

```

```
}
```

นอกจากนี้การเขียนรวมได้ใน subscribe() แล้ว จะไม่ต้องใส่ชื่อแทนฟังก์ชัน next, error, และ complete ก็ได้ โดยใส่เป็นฟังก์ชันไม่มีชื่อไปเลย (lambda)

การสร้างผู้สังเกตการณ์ประเภทอีเว้นท์ โดยรับสมัครจากฟังก์ชัน getEventConstants() ที่ได้สร้างมาก่อนหน้านี้ จะรับสมัครได้โดยตรงคล้ายกับการเข้าสมัครที่ผ่านมา ดังตัวอย่างต่อไปนี้ ซึ่งเพิ่มเติมลงในฟังก์ชัน ngOnInit()

#### Code 23. src/app/users/user.component.ts

```
ngOnInit() {
  //this.users = this.getUsers();
  this.getUserObservable();

  const subscription = this.observableEventConstant()
    .subscribe((evt: MouseEvent) => {
      // แสดงตำแหน่ง x, y ของเมาส์
      console.log(`Coords: ${evt.clientX}:${evt.clientY}`);
      // เมื่อเมาส์เคลื่อนไปซ้ายของหน้าจอ ให้หยุดรับใช้บริการ
      if (evt.clientX < 40 && evt.clientY < 40){
        subscription.unsubscribe();
      }
    });
}
```

จากตัวอย่างผู้สังเกตการณ์สมัครรับอีเว้นท์โดยตรงกับ observableEventConstant() ภายใต้ฟังก์ชัน subscribe() โดยมีตัวแปรเข้า (evt) เป็นเหตุการณ์ MouseEvent และสามารถยกเลิกการรับสมัครเมื่อมีเหตุการณ์ เคลื่อนเมาส์ ในตำแหน่งที่กำหนดตามเงื่อนไข if

สำหรับการสมัครกับอีเว้นท์ที่เป็นไดนามิก observableEventDynamic() ซึ่งรับเหตุการณ์จาก <li> อันเป็นข้อมูลไดนามิก ทำให้เหตุการณ์หรืออีเว้นท์ยังไม่สามารถได้สำเร็จทันที ต้องรอจนกว่าจะมี <li> ครบทุกรายการ การหน่วงเวลาจนกว่าจะแสดงข้อมูล <li> ครบ

การหน่วงเวลา ใช้การนำเข้า interval จาก rxjs ซึ่งต้องนำเข้ามาด้วย

```
import { interval } from 'rxjs';
```

ตัวอย่างต่อไปนี้ใส่เพิ่มเติมลงในฟังก์ชัน ngOnInit() ต่อท้ายจากตัวอย่างโปรแกรมที่ผ่านมา

#### Code 24. src/app/users/user.component.ts

```
const secondsCounter = interval(1000);
const subscriptionCounter = secondsCounter.subscribe(n =>{
  console.log(`It's been ${n} seconds since subscribing!`);
  this.observableEventDynamic().subscribe(
    (evt:MouseEvent)=>{
      console.log(`Mouseover:${evt.clientX}:${evt.clientY}`);
    });
  if (n>5) subscriptionCounter.unsubscribe();
});
```

จากตัวอย่างโปรแกรมนี้นี้ ใช้การหน่วงเวลา ก่อนที่จะสมัครจาก `observEventDynamic()` และในส่วน การหน่วงเวลา เมื่อ ค่าเวลาผ่านไป มากกว่า 5 วินาที ให้สังเกตการณ์ `secondsCounter` จะยกเลิกสมัครรับเหตุการณ์หน่วงเวลา

### สมัครรับบริการอย่างอัตโนมัติ

ที่ผ่านมาใช้การรับสมัครเต็มรูปแบบ คือมีการใช้ฟังก์ชัน `subscribe()` เพื่อรับข้อมูลสังเกตการณ์ แต่ก็มีอีกริธีที่ใช้รับสมัครโดยไม่ต้องเรียกใช้ฟังก์ชัน `subscribe()` แต่ใช้การสมัครผ่าน หน้า HTML ด้วยการต่อเชื่อมกับ คำสำคัญ **async**

ด้วยวิธีการใหม่มีข้อดีอย่างแรกคือ ทำการรับสมัครอัตโนมัติ และหยุดรับสมัครอัตโนมัติ เมื่อไม่ใช้งานคอมโพเนนต์ ทำให้เราไม่จำเป็นต้องเขียนหยุดรับสมัครเอง ผลคือลดการใช้หน่วยความจำ

ขั้นตอนการทำงานรับสมัครมีเพียง 2 ขั้นตอนที่ต้องแก้ไข คือแก้ไขบน คอมโพเนนต์ ทั้งในไฟล์ `ts` และ `html` เริ่มต้นกันที่ไฟล์ `users.component.ts` ให้แก้ไขโดยไม่ต้องเขียนการรับสมัคร

#### Code 25. `src/app/usres/users.component.ts`

```
users: Observable<User[]>;
getUserObservable: Observable<User[]>{
  console.log("Async..");
  return this.userService.getUserObservable();
}
ngOnInit(){
  this.users = this.getUserObservable();
}
```

ตอนนี้มีส่วนที่แก้ไขสำคัญคือ `users` มีชนิดเป็น `Observable<User[]>` แทนที่จะเป็น `User[]` เพราะรับค่า `Observable<User>` ในส่วน การรับสมัครตัดออกไป ใช้การคืนค่าแทน (`return`) และส่วน `ngOnInit()` ใช้ การส่งค่าให้กับ `users`

สำหรับการผูกข้อมูลบนเทลทเพลท `user.component.html` ใช้ การไปป์ ไปยัง `async` เพื่อแทนการรับสมัครอัตโนมัติ เพียงเท่านี้ก็อ่านค่าได้ผลเหมือนกับการรับสมัคร

สำหรับไฟล์ `user.service.ts` ไม่ได้เปลี่ยนแปลงอะไร เพราะเป็นตัวให้รับสมัคร หรือตัวให้ตัวสังเกตการณ์เท่านั้น

#### Code 26. `src/app/users/users.component.ts`

```
<p>
  <li *ngFor="let u of users_ | async">
    {{u.id}}:{{u.fname}} {{u.lname}} {{u.email}}</li>
</p>
```

### ตัวดำเนินการของตัวให้สังเกตการณ์

ตัวดำเนินการหมายถึงตัวจัดการกับข้อมูลของข้อมูลของผู้ให้สังเกตการณ์ ซึ่งมีหลายตัว เช่น `map`, `filter`, `tap`, `of` ทั้งหมดต้องนำเข้าจาก `rxjs/operators` หรือ `rxjs` สำหรับตัวที่เคยใช้แล้ว เช่น `of` จัดอยู่ในกลุ่มตัวดำเนินการผู้สร้าง (`creation`) จะคืนข้อมูลแต่ละตัวที่ให้สังเกตการณ์ แต่การใช้ `of` อาจให้ความเข้าใจผิดว่าส่งข้อมูลออกในรูปอาร์เรย์ และการใช้ ฟังก์ชัน `map()` ซึ่งอยู่ในกลุ่มตัวดำเนินการเพื่อการเปลี่ยนแปลง (`transformation`) เช่นตัวอย่างต่อไปนี้

#### Code 27.

```
import { map } from 'rxjs/operators';
```

```
const nums = of(1, 2, 3);
const squareValues = map((val: number) => val * val);
const squaredNums = squareValues(nums);
squaredNums.subscribe(x => console.log(x));
// Logs // 1 // 4 // 9
```

การใช้ `of()` ในลักษณะนี้ส่งข้อมูลออกเป็นตัว ๆ (ไม่ใช่อาร์เรย์) ดังหลังจากสมัครรับข้อมูลด้วย ด้วยฟังก์ชัน `subscribe()` ให้พิมพ์ผลผ่าน `console.log()` ก็จะได้ข้อมูลเลขยกกำลังสองของตนเอง จากการใช้ `map()` โดยมีตัววิ่ง เป็นตัวเลขแต่ละตัวของ `nums` เมื่อได้แนวการใช้งานนี้ เรามาลองใช้กับ ข้อมูลของ `User[]` ซึ่งเป็นอาร์เรย์ของ `User` การใช้งานก็น่าจะคล้ายๆ กัน แต่ทดลองใช้ในการกรองข้อมูล `filter` ซึ่งจัดเป็นตัวดำเนินการกลุ่มการกรองข้อมูล (filtering) ที่ `user.id > 2` คือเลือกรายที่มี `id` มากกว่า 2 ดังนั้น การแก้ไขควรเป็นดังนี้

#### Code 28.

---

```
import { map, filter } from 'rxjs/operators';

const users = of(User[]);
const filterOf = filter((user: User) => user.id>2);
const userObservable = filterOf (users);
userObservable.subscribe(x => console.log(x.id));
```

การใช้ `of()` ในลักษณะนี้ใช้ส่งข้อมูลออกเป็นอาร์เรย์ จึงไม่สามารถทำได้ เนื่องจาก `filterOf` มีการใช้ตัวแปรเข้าเป็น `users` เป็นชนิด `User[]` แต่ตอนนิยาม `filter` ใช้ตัวแปรเข้าเป็น `user` มีชนิดเป็น `User` ซึ่งทั้งสองตัวแปรเป็นคนละชนิดกันจึงไม่สามารถดำเนินการได้ เมื่อพบปัญหาแบบนี้แล้ว จะต้องแก้ไขให้ตัวแปรชนิดเดียวกัน ดังแนวทางต่อไปนี้

#### Code 29. src/app/users/user.service.ts

---

```
getUserObservable(): Observable<User[]> {
  const myObservable = of(USERS);
  const idGreaterThan2 = map((users:User[])=>
    users.filter((user:User)=>user.id>2));
  return idGreaterThan2(myObservable);
}
```

จากตัวอย่างนี้ ได้แก้ไขใหม่ โดยใช้ ฟังก์ชัน `map()` เป็นตัวรับข้อมูลเข้าเป็นอาร์เรย์ `User[]` แล้วค่อยใช้ฟังก์ชัน `filter()` รับข้อมูลเข้าเป็น `User` ซึ่งแทนตัววิ่งใน `User[]` อีกทีเพื่อแก้ปัญหานี้ ข้อสังเกตว่า `filter()` นี้ไม่ได้เป็นตัวดำเนินการของ `rxjs` แต่เป็นตัวดำเนินการของอาร์เรย์ทั่วไป

ยังมีการรวมตัวดำเนินการไว้ด้วยกันด้วยการใช้ฟังก์ชัน `pipe()` ซึ่งเป็นการต่อรวมฟังก์ชันหลายตัวเข้าด้วยกัน

#### Code 30. src/app/users/user.service.ts

---

```
getUserObservable():Observable<User[]>{
  const myObservable = of(USERS).pipe(
    map((users:User[])=>users.filter((user:User)=>user.id>2)),
    map((users:User[])=>users.filter((user:User)=>user.lname=="T."))
  );
  return myObservable;
}
```

ด้วยการกรองข้อมูลสองชั้น คือชั้นแรกเลือกเฉพาะรายที่มี id<2 และชั้นที่สอง รายที่มี lname เป็น T. และรวมสอง การกรองข้อมูลด้วย pipe( ) และผลลัพธ์ของการใช้ฟังก์ชัน getUserObservable( ) นี้จะได้คือ

5: Jel T. Jel@gmail.com

ตัวดำเนินการอีกตัวที่คล้ายกับ map คือ mergeMap ตัวดำเนินการนี้ (ต้องนำเข้าตัว mergeMap ด้วย)

```
import { filter, map, mergeMap } from 'rxjs/operators';
```

mergeMap สร้างค่าให้สังเกตการณ์ใหม่ จากการเปลี่ยนแปลงแต่ละค่าภายในค่าสังเกตการณ์ เช่น ต้องการ เปลี่ยนแปลง id ให้มีค่าเพิ่มขึ้น 10 ทุกรายการ

**Code 31.** src/app/users/user.service.ts

```
getUserObservable():Observable<User[]>{  
  const myObservable = of(USERS).pipe(  
    mergeMap((users:User[])=> {  
      users.forEach(u=>u.id+=10);  
      return of(users);  
    })  
  );  
  return myObservable;  
}
```

ให้สังเกตว่า ภายในแลมบ์ดา ของ mergeMap ใช้เครื่องหมายปีกกาเพราะมีหลายคำสั่ง (สองบรรทัด) และต้องใส่ค่า ว่า return หน้า of(users) เพราะต้องเขียนเต็มรูปแบบ แต่ถ้ามีเพียงบรรทัดเดียว ไม่จำเป็นต้องมีปีกกา และไม่ต้องมี return

### สร้างคอมโพเนนต์ MessagesComponent

อีกครั้ง ให้สร้างคอมโพเนนต์ Messages เพื่อแสดงข้อความในหน้าเว็บ ด้วยคำสั่ง CLI สำหรับ Windows ใช้ Command Prompt

```
ng generate component message
```

เมื่อสร้างเสร็จแล้ว ให้เพิ่มคอมโพเนนต์ลงในไฟล์ app.component.html โดยวางในตำแหน่งล่างสุด เพื่อแสดง ข้อมูลที่ตำแหน่งล่างสุดของหน้าเว็บ

**Code 32.** /src/app/app.component.html

```
<h2>User List</h2>  
<app-users></app-users>  
<app-message></app-message>
```

ผลการเพิ่ม <app-message> เข้าไปใหม่ จะมีข้อความ **message works!** แสดงเพิ่มเติมต่อท้าย ซึ่งเป็น ข้อความที่มาจาก message.component.html

### สร้างบริการ MessageService

สร้างงานบริการอีกตัว เพื่อแจ้งผลการอ่านข้อมูล User ด้วยการใช้ CLI ดังรูปต่อไปนี้

ng generate service message

เมื่อสร้างงานบริการเสร็จให้เปิดไฟล์ message.service.ts แล้วแก้ไขตามไฟล์ต่อไปนี้

**Code 33.** /src/app/message.service.ts

---

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

@Injectable({
  providedIn: 'root',
})
export class MessageService {
  messages: string[] = [];

  add(message: string) {
    this.messages.push(message);
  }

  clear() {
    this.messages = [];
  }
}
```

จากไฟล์ที่แก้ไขใหม่นี้ ทำหน้าที่เก็บข้อความแบบอาร์เรย์ (messages) ในรูปอาร์เรย์ ผ่านฟังก์ชัน add( ) และมีฟังก์ชัน clear( ) เพื่อล้างข้อมูลทั้งหมด

#### ให้งานบริการเรียกงานบริการ

ให้งานบริการ UserService เรียกงานบริการ MessageService การจะเรียกใช้ ก็ต้องทำการนำเข้าด้วยคำสั่งต่อไปนี้

**Code 34.** /src/app/user.service.ts

---

```
import { MessageService } from './message.service';
```

กำหนดให้งานบริการ MessageService เป็นส่วนหนึ่งของ UserService ได้ผ่านคอนสตรัคเตอร์ แบบเขียนวิธีลัดผ่านคอนสตรัคเตอร์ (อยู่ภายในวงเล็บ ไม่ใช่ในปีกกา) และกำหนดให้ใช้ได้เฉพาะ คลาส UserService ด้วยการกำหนดการเข้าถึงแบบ private

**Code 35.** /src/app/user.service.ts

---

```
constructor(private messageService: MessageService) { }
```

เมื่อได้สมาชิกใหม่ตามที่กำหนดในคอนสตรัคเตอร์แล้ว ก็สามารถเรียกใช้บริการใหม่นี้ได้ ในที่นี้เลือกใช้บริการผ่านฟังก์ชัน add( ) ในฟังก์ชัน getUser( ) ก่อนจะคืนค่าคงที่ USERS แบบอซินโครนัส

**Code 36.** /src/app/user.service.ts

---

```
getUsers(): Observable<User[]> {
  // TODO: send the message _after_ fetching the users
  this.messageService.add('UserService: fetched users');
  return of(USERS);
}
```

## คอมโพเนนต์ที่ใช้บริการ MessageService

คอมโพเนนต์ MessageComponent ที่สร้างไว้ก่อนหน้านี้ให้เรียกใช้บริการ MessageService ได้จะต้องนำเข้าก่อนดังเพิ่มคำสั่งต่อไปนี้

```
Code 37. /src/app/message/message.component.ts
import { MessageService } from '../message.service';
```

ต่อมา ตามด้วยกำหนดสมาชิกใหม่ของคอมโพเนนต์ผ่านคอนสตรัคเตอร์ ในครั้งนี้กำหนดสมาชิกใหม่มีการเข้าถึงเป็น private การทำเช่นนี้ ทำให้ ส่วนอื่นของทั้งแอปพลิเคชัน เข้าถึงได้ ซึ่งจะเรียกใช้ผ่าน เทมเพลต (template) หรือไฟล์ HTML ของคอมโพเนนต์นี้

```
Code 38. /src/app/message/message.component.ts
constructor(public messageService: MessageService) {}
```

```
Code 39. src/app/message/message.component.html
<h2>Messages</h2>
<div *ngIf="messageService.messages.length">
  <button class="clear"
    (click)="messageService.clear()">clear</button>
  <div *ngFor='let message of messageService.messages'> {{message}} </div>
</div>
```

สำหรับเทมเพลตนี้ มีการผูกข้อมูลกับ messageService ซึ่งเป็นสมาชิกใหม่ที่เพิ่งสร้างมา การผูกข้อมูลใช้คำสั่งปีกกาคู่ {{ }} แต่มีเงื่อนไข ตามคำสั่ง \*ngIf เพื่อแสดงผลแบบมีเงื่อนไขถ้า (if) โดยกำหนดเงื่อนไขของงานบริการว่า messageService.messages.length อันหมายถึงมีขนาดมากกว่าศูนย์ เมื่อเงื่อนไขเป็นจริง จะทำการวนซ้ำผ่าน คำสั่ง \*ngFor เพื่อวนซ้ำข้อความในอาร์เรย์ ในตัวอย่างนี้ ใช้ วนซ้ำแบบ forEach มีการใช้ขอบเขตตัวแปรแบบ let เพื่อบ่งบอกว่าตัวแปร message นี้ใช้ได้เฉพาะในการวนซ้ำนี้เท่านั้น



รูป 4 ผลการทำงานของเพิ่มคอมโพเนนต์ message

นอกจากนี้ ยังผูกเหตุการณ์คลิก (click) ไว้กับฟังก์ชัน clear( ) ของงานบริการ MessageService อีกด้วยเพื่อล้างข้อมูลข้อความทั้งหมด

### สรุป

ในบทนี้ได้พยายามอธิบายการสร้างงานบริการ ที่มีการเรียกใช้งานทั้งในลักษณะซินโครนัส และแบบอซินโครนัส ความแตกต่างหลักของงานบริการนี้คือ แบบซินโครนัส นั้นเรียกใช้บริการได้โดยตรง ในขณะที่แบบอซินโครนัสนั้นเรียกใช้งานผ่านออบเจกต์ Observable ซึ่งจำเป็นต้องมีการทำงานแบบรับส่งข้อมูลแบบอะซิงโครนัส สำหรับงานเว็บควร์ที่จะใช้การทำงานแบบอซินโครนัส โดยเฉพาะการอ่านข้อมูล เนื่องด้วยความไม่แน่นอนของเวลาส่งผ่านข้อมูลผ่านระบบอินเทอร์เน็ต ดังนั้นการทำความเข้าใจการทำงานแบบอซินโครนัส จึงสำคัญอย่างยิ่ง ในบทนี้ ได้เน้นการทำงานแบบอซินโครนัส โดยยกตัวอย่างถึงสองตัวอย่าง คืองานบริการ UserServer และ MessageService โดยตัวอย่างบริการตัวหลังนี้เป็นการเรียกใช้งานบริการผ่านงานบริการ UserService อีกต่อหนึ่ง

### แบบฝึกหัด

1. ให้สร้างงานบริการสำหรับการเพิ่มราย User โดยให้รับ ข้อมูลเดิมให้เป็นค่า อาร์เรย์ทั่วไป ให้ตัดส่วนที่เป็นค่าคงที่ออก และใส่คีย์เวิร์ด let แทน โดยดำเนินการแบบ ซินโครนัส
2. เมื่อ มีการเพิ่มข้อมูล User ดังทำในข้อหนึ่งแล้ว ให้เพิ่ม ข้อความในอาร์เรย์ของ message ของคลาส MessageService เพื่อแจ้งว่าได้มี User รายใหม่ได้เพิ่มแล้ว เช่น เพิ่มข้อความว่า : new user is added

### แหล่งข้อมูลเพิ่มเติม

1. Angular. (11 Aug. 2020). The RxJS library. <https://angular.io/guide/rx-library>.
2. RxJs. (11 Aug. 2020). API List. <https://rxjs.dev/api>.