# Angular Asynchronous Handling Guide

## 1. Using Observables

Observables are a powerful way to manage asynchronous data streams. They allow you to subscribe to data changes, making them particularly useful for handling HTTP requests and real-time data.

### Example: HTTP Requests with Observables

1. Create a Service for HTTP Requests:

Generate a service to handle HTTP requests:

ng generate service data

In data.service.ts:

import { Injectable } from '@angular/core';  
import { HttpClient } from '@angular/common/http';  
import { Observable } from 'rxjs';  
  
export interface Item {  
 id: number;  
 name: string;  
}  
  
@Injectable({  
 providedIn: 'root',  
})  
export class DataService {  
 private apiUrl = 'https://api.example.com/items'; // Replace with your API URL  
  
 constructor(private http: HttpClient) {}  
  
 getItems(): Observable<Item[]> {  
 return this.http.get<Item[]>(this.apiUrl);  
 }  
}

2. Subscribe to the Observable in a Component:

In item.component.ts:

import { Component, OnInit } from '@angular/core';  
import { DataService, Item } from '../data.service';  
  
@Component({  
 selector: 'app-item',  
 templateUrl: './item.component.html',  
})  
export class ItemComponent implements OnInit {  
 items: Item[] = [];  
 loading = false;  
 error: string | null = null;  
  
 constructor(private dataService: DataService) {}  
  
 ngOnInit(): void {  
 this.loadItems();  
 }  
  
 loadItems() {  
 this.loading = true;  
 this.dataService.getItems().subscribe({  
 next: (data) => {  
 this.items = data;  
 this.loading = false;  
 },  
 error: (err) => {  
 this.error = 'Failed to load items!';  
 console.error(err);  
 this.loading = false;  
 },  
 });  
 }  
}

3. Template Example:

In item.component.html:

<div \*ngIf="loading">Loading...</div>  
<div \*ngIf="error">{{ error }}</div>  
<ul>  
 <li \*ngFor="let item of items">{{ item.name }}</li>  
</ul>

## 2. Using Promises

Promises are another way to handle asynchronous operations. However, they are less commonly used in Angular for HTTP requests since Observables provide more features.

### Example: HTTP Requests with Promises

1. Create a Service that Returns Promises:

In data.service.ts:

import { Injectable } from '@angular/core';  
import { HttpClient } from '@angular/common/http';  
import { Item } from './item.model'; // Adjust according to your model structure  
  
@Injectable({  
 providedIn: 'root',  
})  
export class DataService {  
 private apiUrl = 'https://api.example.com/items';  
  
 constructor(private http: HttpClient) {}  
  
 getItems(): Promise<Item[]> {  
 return this.http.get<Item[]>(this.apiUrl).toPromise();  
 }  
}

2. Handle the Promise in the Component:

In item.component.ts:

import { Component, OnInit } from '@angular/core';  
import { DataService, Item } from '../data.service';  
  
@Component({  
 selector: 'app-item',  
 templateUrl: './item.component.html',  
})  
export class ItemComponent implements OnInit {  
 items: Item[] = [];  
 loading = false;  
 error: string | null = null;  
  
 constructor(private dataService: DataService) {}  
  
 ngOnInit(): void {  
 this.loadItems();  
 }  
  
 loadItems() {  
 this.loading = true;  
 this.dataService.getItems()  
 .then(data => {  
 this.items = data;  
 this.loading = false;  
 })  
 .catch(err => {  
 this.error = 'Failed to load items!';  
 console.error(err);  
 this.loading = false;  
 });  
 }  
}

## 3. Using Async Pipe

When working with Observables, you can also use the async pipe in your templates to handle subscriptions automatically, which reduces boilerplate code.

### Example: Using Async Pipe

1. Update the Component:

In item.component.ts:

import { Component, OnInit } from '@angular/core';  
import { DataService, Item } from '../data.service';  
import { Observable } from 'rxjs';  
  
@Component({  
 selector: 'app-item',  
 templateUrl: './item.component.html',  
})  
export class ItemComponent implements OnInit {  
 items$: Observable<Item[]> | undefined; // Expose the observable  
  
 constructor(private dataService: DataService) {}  
  
 ngOnInit(): void {  
 this.items$ = this.dataService.getItems(); // Assign the observable  
 }  
}

2. Template Example Using Async Pipe:

In item.component.html:

<div \*ngIf="!(items$ | async); else loading">Loading...</div>  
<ng-template #loading>  
 <div>Loading items...</div>  
</ng-template>  
<ul>  
 <li \*ngFor="let item of (items$ | async)">{{ item.name }}</li>  
</ul>  
=========

**Promise**

A [Promise](https://www.geeksforgeeks.org/javascript-promise/) represents a single value in the future, that may not be available at present but is expected to be resolved or rejected in the future

 the **resolve** and **reject**. It offers a structured way to handle resolved or rejected states. It has “[**then ()**](https://www.geeksforgeeks.org/javascript-promise-then-method/)” to handle resolved states and “[**catch ()**](https://www.geeksforgeeks.org/javascript-promise-catch-method/)” to handle rejected ones

const myPromise = new Promise((resolve, reject) => {

// Asynchronous operation

if (success) {

resolve(response);

} else {

reject(error);

}

});

myPromise.then(result => {

// Handle success here

}).catch(error => {

// Handle error here

});

**Observable**

An [Observable](https://www.geeksforgeeks.org/angular-7-observables/) is a powerful concept that is in Angular through the *[RxJS library](https://www.geeksforgeeks.org/rxjs-beginners-guide/)*. These are a sequence of values that can be arrived at over time. These can emit values either synchronously or asynchronously. Mainly used to handle streams of data that change over time.

import { Observable } from 'rxjs';

const myObservable = new Observable(observer => {

// Asynchronous operation

if (success) {

observer.next(result);

} else {

observer.error(error);

}

});

myObservable.subscribe(result => {

// Handle success here

}, error => {

// Handle error here

});

| **Promise** | **Observable** |
| --- | --- |
| **Handling multiple values** | Handles single value. | Handle multiple values at a time. |
| **Asynchronous support** | Suitable for asynchronous communication | Suitable for both synchronous and asynchronous communication |
| **Cancellation** | Cannot be canceled once initiated. | Can be canceled whenever we want. |
| **Complex data transformation** | Limited support. | Wide range of support. |
| **Error Handling** | The catch() method is used for handling errors. | This offers different mechanisms. |
| **Conciseness** | Simple and concise syntax. | More complex due to extensive support. |
| **Use Cases** | Suitable for one-time tasks like reading files. | Suitable for continuous real-time updates like in stock market dashboards. |