## Async Pipe

**Learning Objectives**

* When to use the *async* pipe.
* How to use async pipe with Promises and also Observables.

**Overview**

Normally to render the result of a promise or an observable we have to:

1. Wait for a *callback*.
2. Store the result of the callback is a *variable*.
3. *Bind* to that variable in the template.

With AsyncPipe we can use promises and observables directly in our template, without having to store the result on an intermediate property or variable.

AsyncPipe accepts as argument an observable or a promise, calls subcribe or attaches a then handler, then waits for the asynchronous result before passing it through to the caller.

### AsyncPipe with promises

Lets first create a component with a promise as a property.

@Component({

selector: 'async-pipe',

template: `

<div class="card card-block">

<h4 class="card-title">AsyncPipe</h4>

<p class="card-text" ngNonBindable>{{ promiseData }}</p>

①

<p class="card-text">{{ promiseData }}</p>

②

</div>

`

})

class AsyncPipeComponent {

promiseData: string;

constructor() {

this.getPromise().then(v => this.promiseData = v);

③

}

getPromise() {

④

return new Promise((resolve, reject) => {

setTimeout(() => resolve("Promise complete!"), 3000);

});

}

}

1. We use ngNonBindable so we can render out {{ promiseData }} as is without trying to bind to to the property promiseData
2. We bind to the property promiseData
3. When the promise resolves we store the data onto the promiseData property

④getPromise returns a promise which 3 seconds later resolves with the value "Promise complete!"

In the constructor we wait for the promise to resolve and store the result on a property called promiseData on our component and then bind to that property in the template.

To save time we can use the async pipe in the template and bind to the promise *directly*, like so: [source,javascript]cript.ts

@Component({

selector: 'async-pipe',

template: `

<div class="card card-block">

<h4 class="card-title">AsyncPipe</h4>

<p class="card-text" ngNonBindable>{{ promise }}</p>

<p class="card-text">{{ promise | async }}</p>

①

</div>

`

})

class AsyncPipeComponent {

promise: Promise<string>;

constructor() {

this.promise = this.getPromise();

②

}

getPromise() {

return new Promise((resolve, reject) => {

setTimeout(() => resolve("Promise complete!"), 3000);

});

}

}

1. We pipe the output of our promise to the async pipe.
2. The property promise is the actual unresolved *promise* that gets returned from getPromise without then being called on it.

The above results in the same behaviour as before, we just saved ourselves from writing a then callback and storing intermediate data on the component.

### AsyncPipe with observables

To demonstrate how this works with *observables* we first need to setup our component with a simple *observable*, like so:

import { Observable } from 'rxjs/Rx';

.

.

.

@Component({

selector: 'async-pipe',

template: `

<div class="card card-block">

<h4 class="card-title">AsyncPipe</h4>

<p class="card-text" ngNonBindable>{{ observableData }}

<p class="card-text">{{ observableData }}</p>

①

</div>

`

})

class AsyncPipeComponent {

observableData: number;

subscription: Object = null;

constructor() {

this.subscribeObservable();

}

getObservable() {

②

return Observable

.interval(1000)

.take(10)

.map((v) => v \* v);

}

subscribeObservable() {

③

this.subscription = this.getObservable()

.subscribe( v => this.observableData = v);

}

ngOnDestroy() {

④

if (this.subscription) {

this.subscription.unsubscribe();

}

}

}

1. We render the value of observableData in our template.
2. We create an observable which publishes out a number which increments by one every second then squares that number.
3. We subscribe to the output of this observable chain and store the number on the property observableData. We also store a reference to the subscription so we can unsubscribe to it later.
4. On destruction of the component we unsubscribe from the observable to avoid memory leaks.

We should also be destroying the subscription when the component is destroyed.

Otherwise we will start leaking data as the old observable, which isn’t used any more, will still be producing results.

Again by using AsyncPipe we don’t need to perform the subscribe and store any intermediate data on our component, like so:

@Component({

selector: 'async-pipe',

template: `

<div class="card card-block">

<h4 class="card-title">AsyncPipe</h4>

<p class="card-text" ngNonBindable>{{ observable | async }}

<p class="card-text">{{ observable | async }}</p>

①

</div>

`

})

class AsyncPipeComponent {

observable: Observable<number>;

constructor() {

this.observable = this.getObservable();

}

getObservable() {

return Observable

.interval(1000)

.take(10)

.map((v) => v\*v)

}

}

① We pipe our observable directly to the async pipe, it performs a subscription for us and then returns whatever gets passed to it.

By using AsyncPipe we: 1. Don’t need to call subscribe on our observable and store the intermediate data on our component. 2. Don’t need to remember to unsubscribe from the observable when the component is destroyed.

**Summary**

AsyncPipe is a convenience function which makes rendering data from observables and promises much easier.

For promises it automatically adds a then callback and renders the response.

For Observables it automatically subscribes to the observable, renders the output and then also unsubscribes when the component is destroyed so we don’t need to handle the clean up logic ourselves.

That’s it for the built-in pipes, next up we will look at creating out own custom pipes.

**Listing**

<http://plnkr.co/edit/gHIalfn10CfocBwCE6UG?p=preview>

*script.ts*

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

import {BrowserModule} from '@angular/platform-browser';

import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';

import { Observable } from 'rxjs/Rx';

@Component({

selector: 'async-pipe',

template: `

<div class="card card-block">

<h4 class="card-title">AsyncPipe</h4>

<p class="card-text" ngNonBindable>{{ promise | async }} </p>

<p class="card-text">{{ promise | async }} </p>

<p class="card-text" ngNonBindable>{{ observable | async }} </p>

<p class="card-text">{{ observable | async }}</p>

<p class="card-text" ngNonBindable>{{ observableData }} </p>

<p class="card-text">{{ observableData }}</p>

</div>

`

})

class AsyncPipeComponent implements OnDestroy {

promise: Promise<string>;

observable: Observable<number>;

subscription: Object = null;

observableData: number;

constructor() {

this.promise = this.getPromise();

this.observable = this.getObservable();

this.subscribeObservable();

}

getObservable() {

return Observable

.interval(1000)

.take(10)

.map((v) => v \* v);

}

// AsyncPipe subscribes to the observable automatically

subscribeObservable() {

this.subscription = this.getObservable()

.subscribe((v) => this.observableData = v);

}

getPromise() {

return new Promise((resolve, reject) => {

setTimeout(() => resolve("Promise complete!"), 3000);

});

}

// AsyncPipe unsubscribes from the observable automatically

ngOnDestroy() {

if (this.subscription) {

this.subscription.unsubscribe();

}

}

}

@Component({

selector: 'app',

template: `

<async-pipe></async-pipe>

`

})

class AppComponent {

imageUrl: string = "";

}

@NgModule({

imports: [BrowserModule],

declarations: [AppComponent,

AsyncPipeComponent

],

bootstrap: [AppComponent],

})

class AppModule {

}

platformBrowserDynamic().bootstrapModule(AppModule);