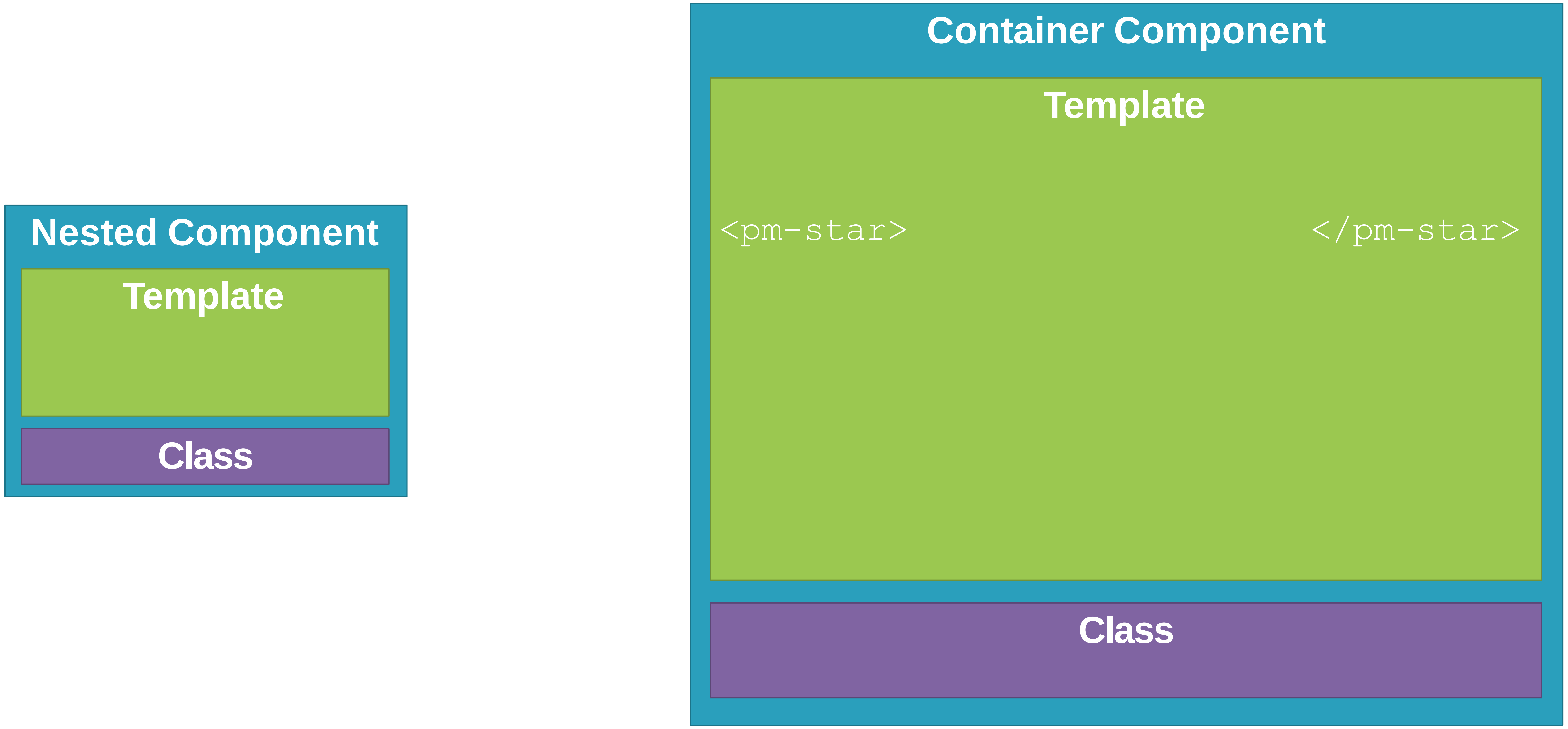
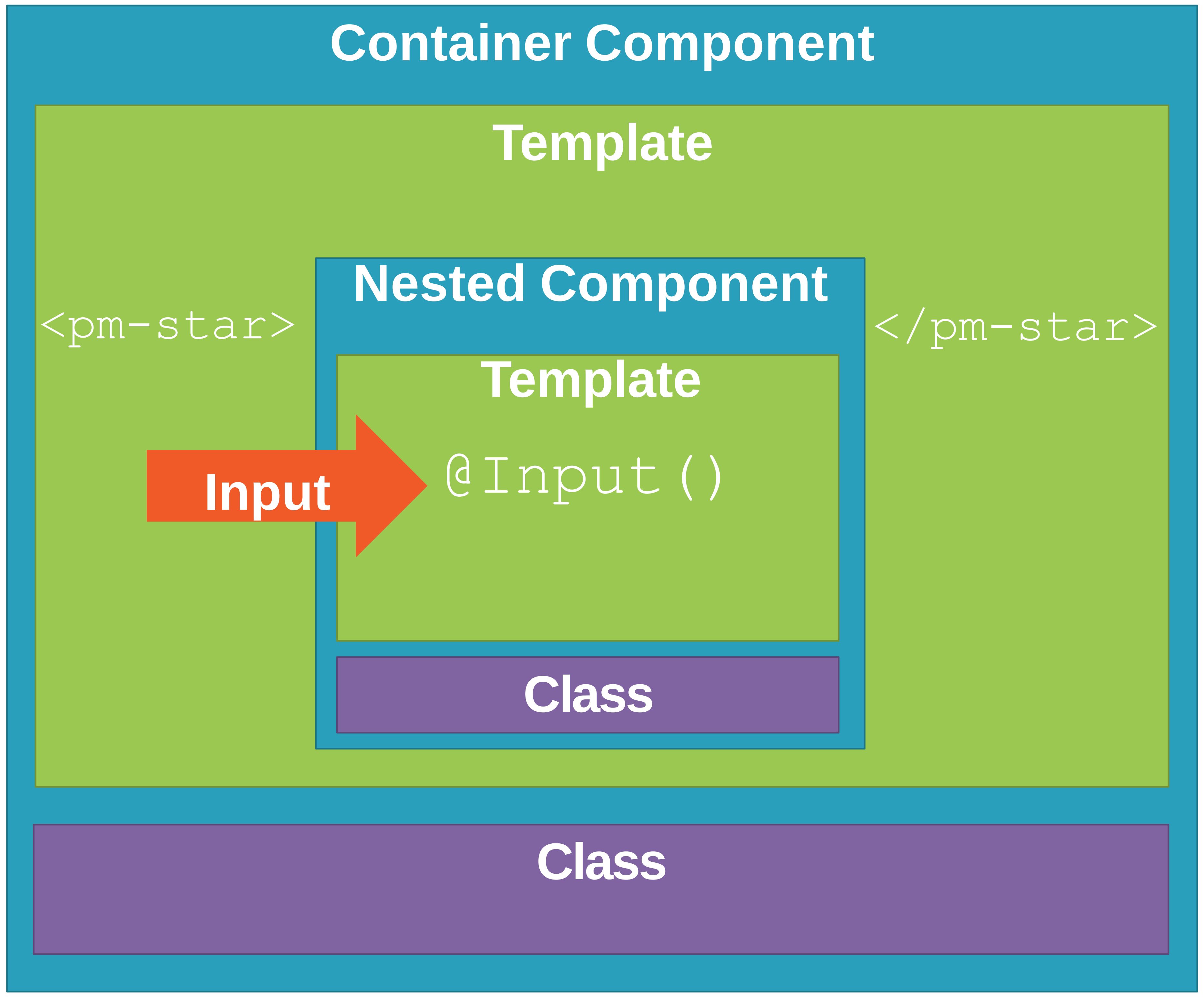
Building Nested Components

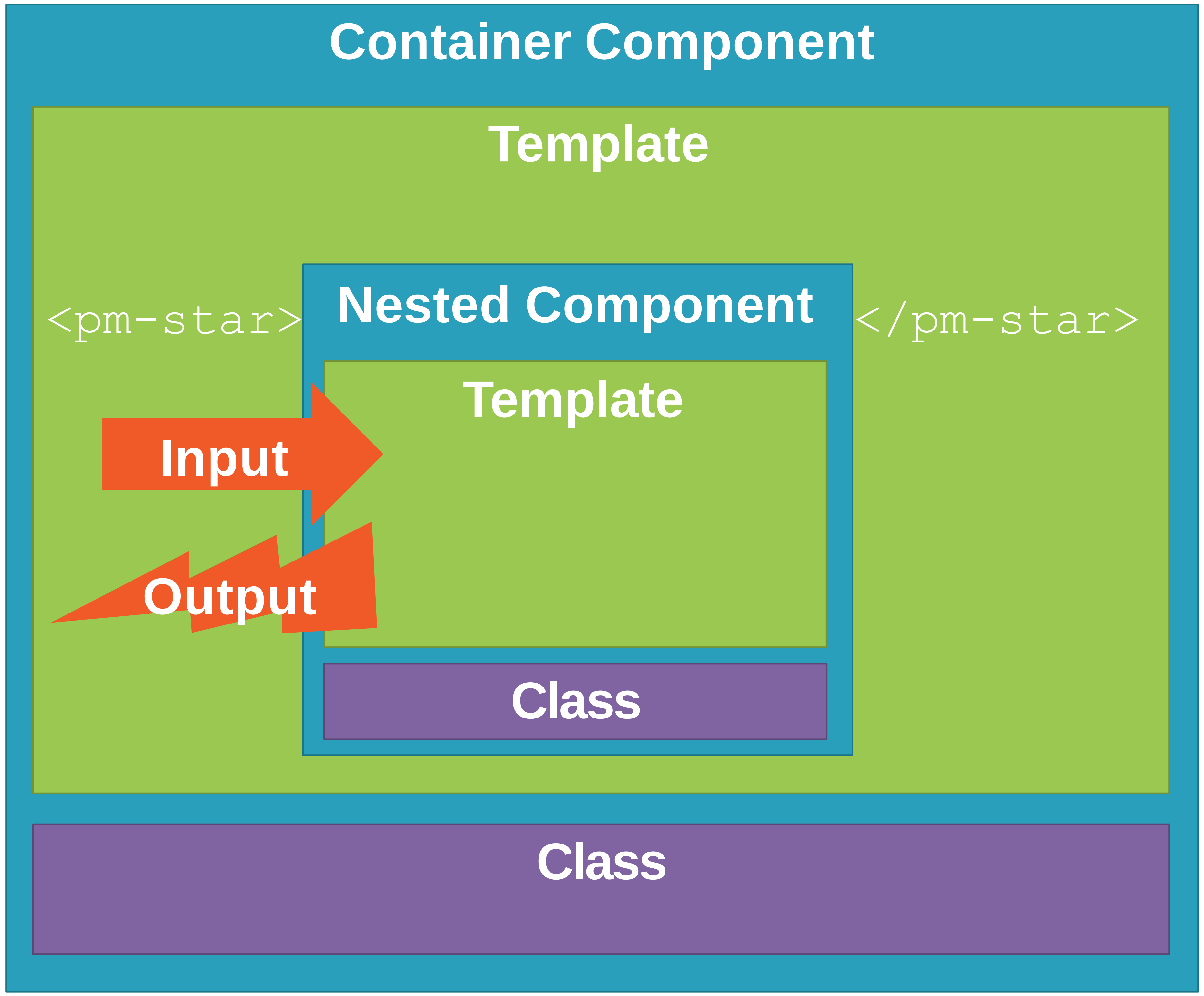
Building a Nested Component



Passing Data to a Nested Component (`@Input`)



Building a Nested Component



Services and Dependency Injection

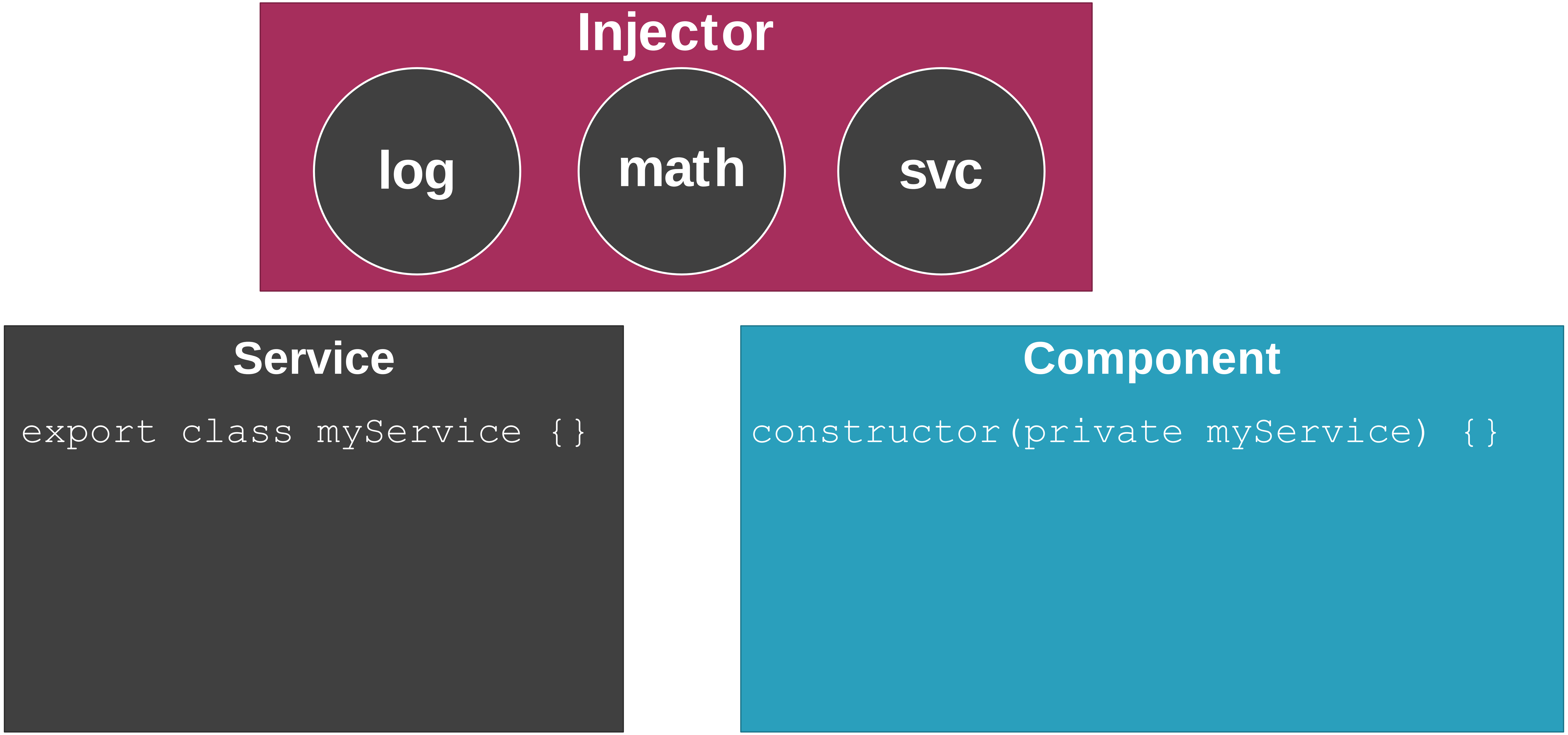
Service

A class with a focused purpose.

Used for features that:

- **Are independent from any particular component**
- **Provide shared data or logic across components**
- **Encapsulate external interactions**

How Does It Work?



```
export class myService {}
```

```
constructor(private myService) {}
```

Dependency Injection

A coding pattern in which a class receives the instances of objects it needs (called dependencies) from an external source rather than creating them itself.

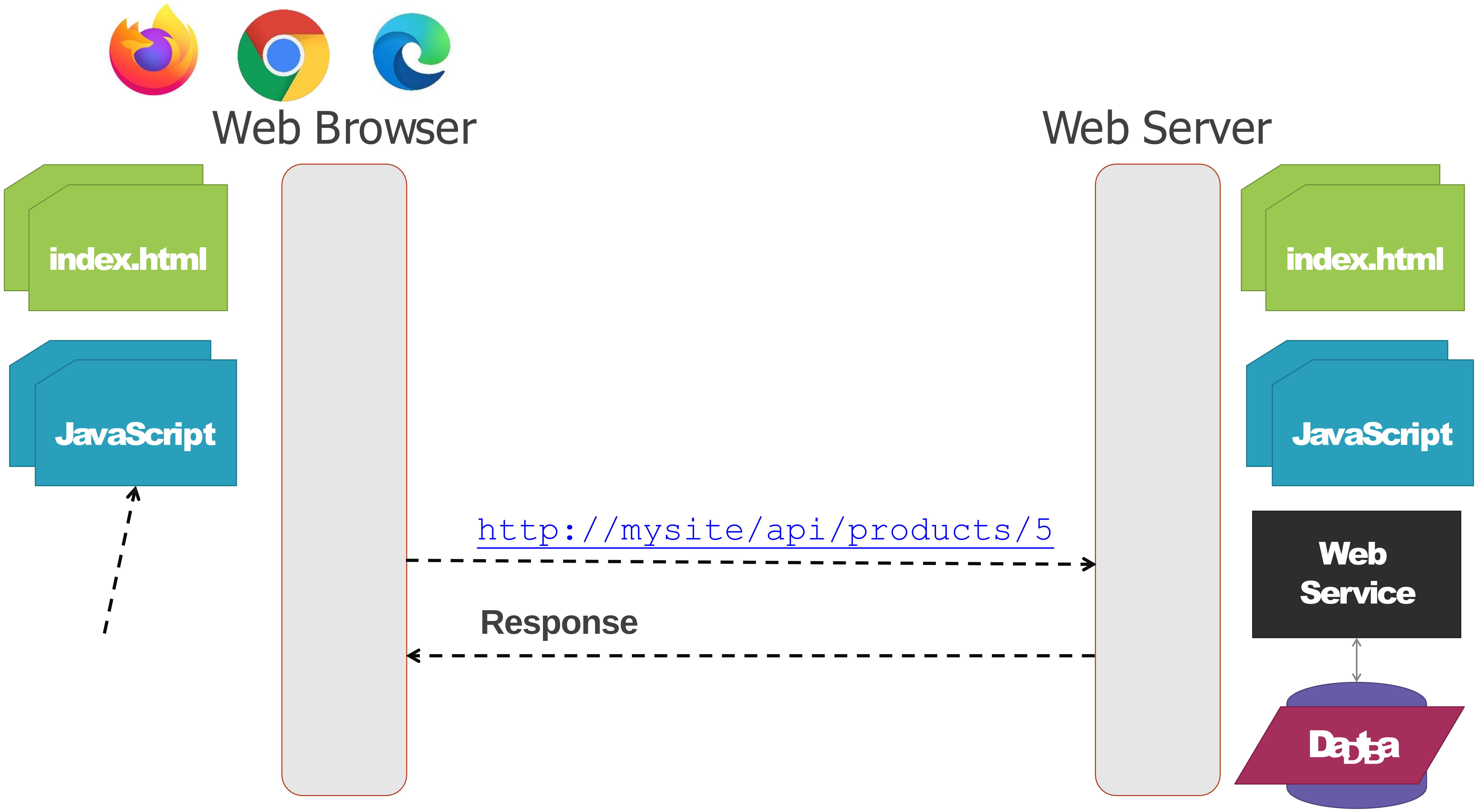
Building a Service

Create the service class

Define the metadata with a decorator

Import what we need

Retrieving Data Using HTTP


Web Browser
Web Server
index.html
JavaScript
index.html
JavaScript
http://mysite/api/products/5
Response
Web Service
Data
DB

To understand the HTTP code,
it's important to understand
Reactive Extensions and
Observables

Reactive Extensions (RxJS)

A library for composing data using observable sequences

And transforming that data using operators

- Similar to .NET LINQ operators

Angular uses Reactive Extensions for working with data

- Especially asynchronous data

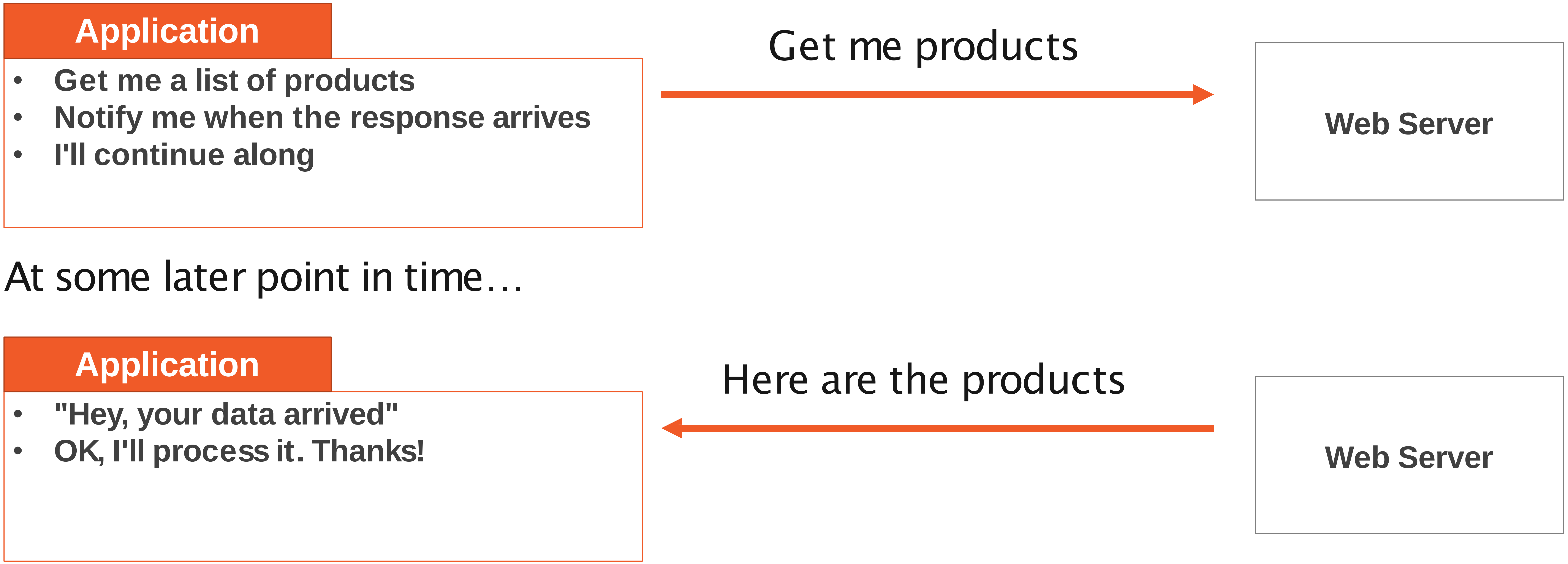
Synchronous vs. Asynchronous

Synchronous: real time

Asynchronous: No immediate response

HTTP requests are asynchronous: request and response

Getting Data



Application

- Get me a list of products
- Notify me when the response arrives
- I'll continue along

Get me products

Web Server

At some later point in time…

Application

- "Hey, your data arrived"
- OK, I'll process it. Thanks!

Here are the products

Web Server

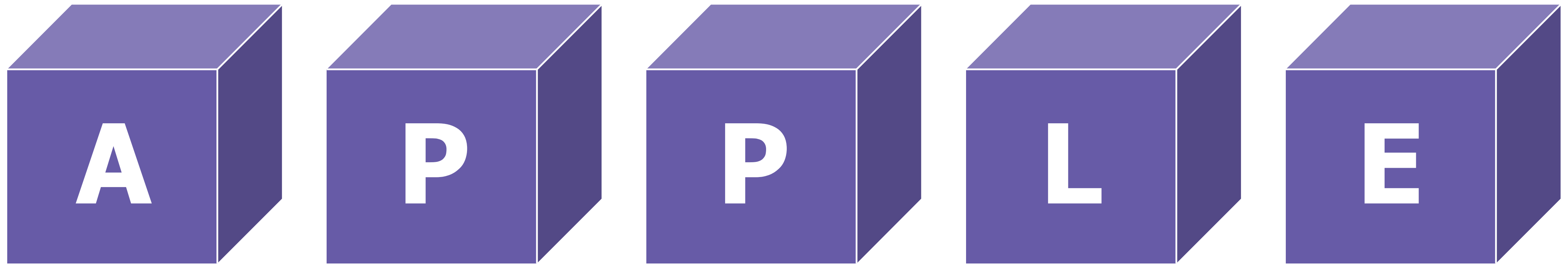
Observable

A collection of items over time

- **Unlike an array, it doesn't retain items**
- **Emitted items can be observed over time**

Array: [A, P, P, L, E]

Observable:



What Does an Observable Do?

Nothing until we subscribe

`next`: **Next item is emitted**

`error`: **An error occurred and no more items are emitted**

`complete`: **No more items are emitted**

Getting Data

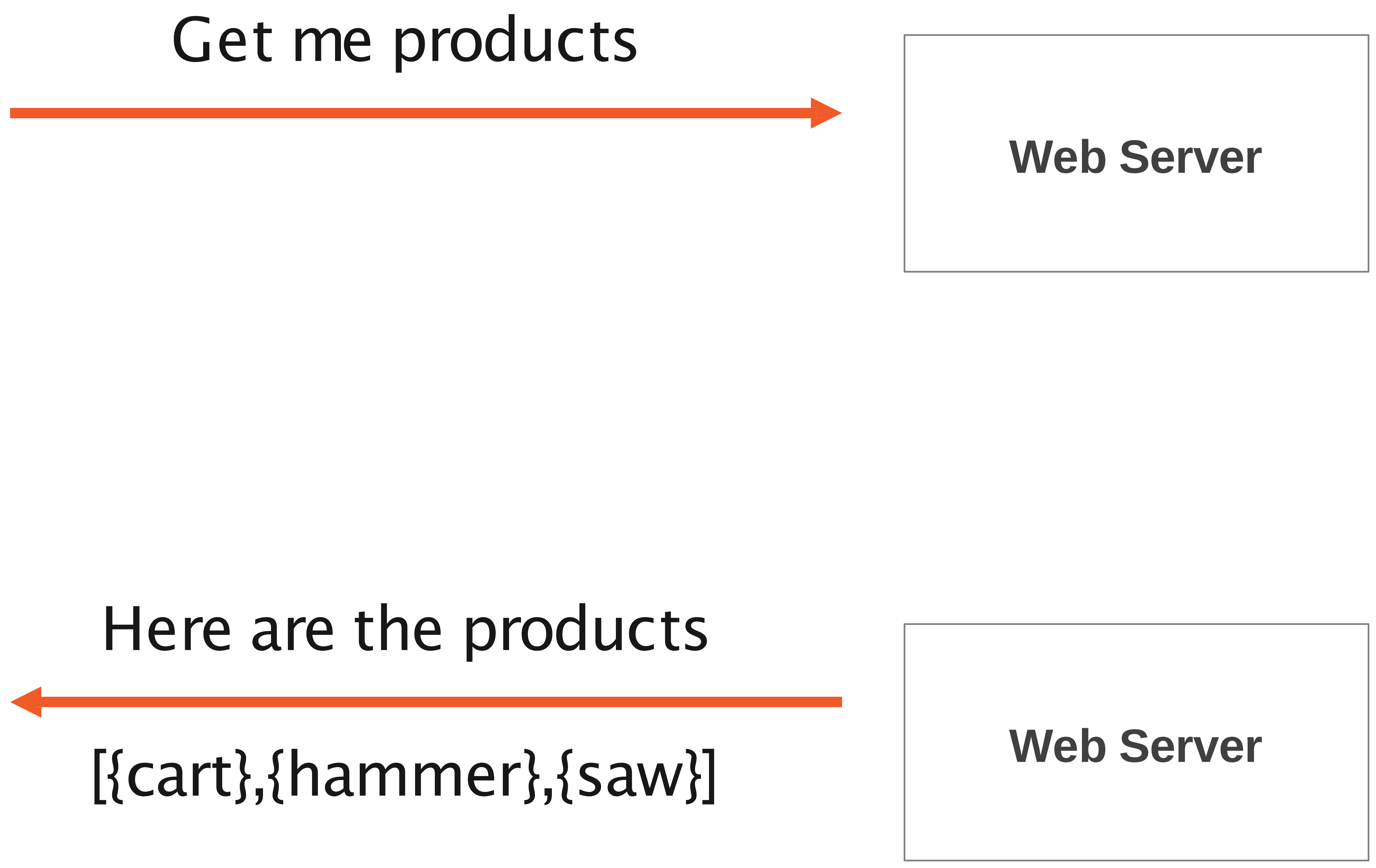
Application

- Call http get
- http get returns an Observable, which will emit notifications
- Subscribe to start the Observable and the get request is sent
- Code continues along

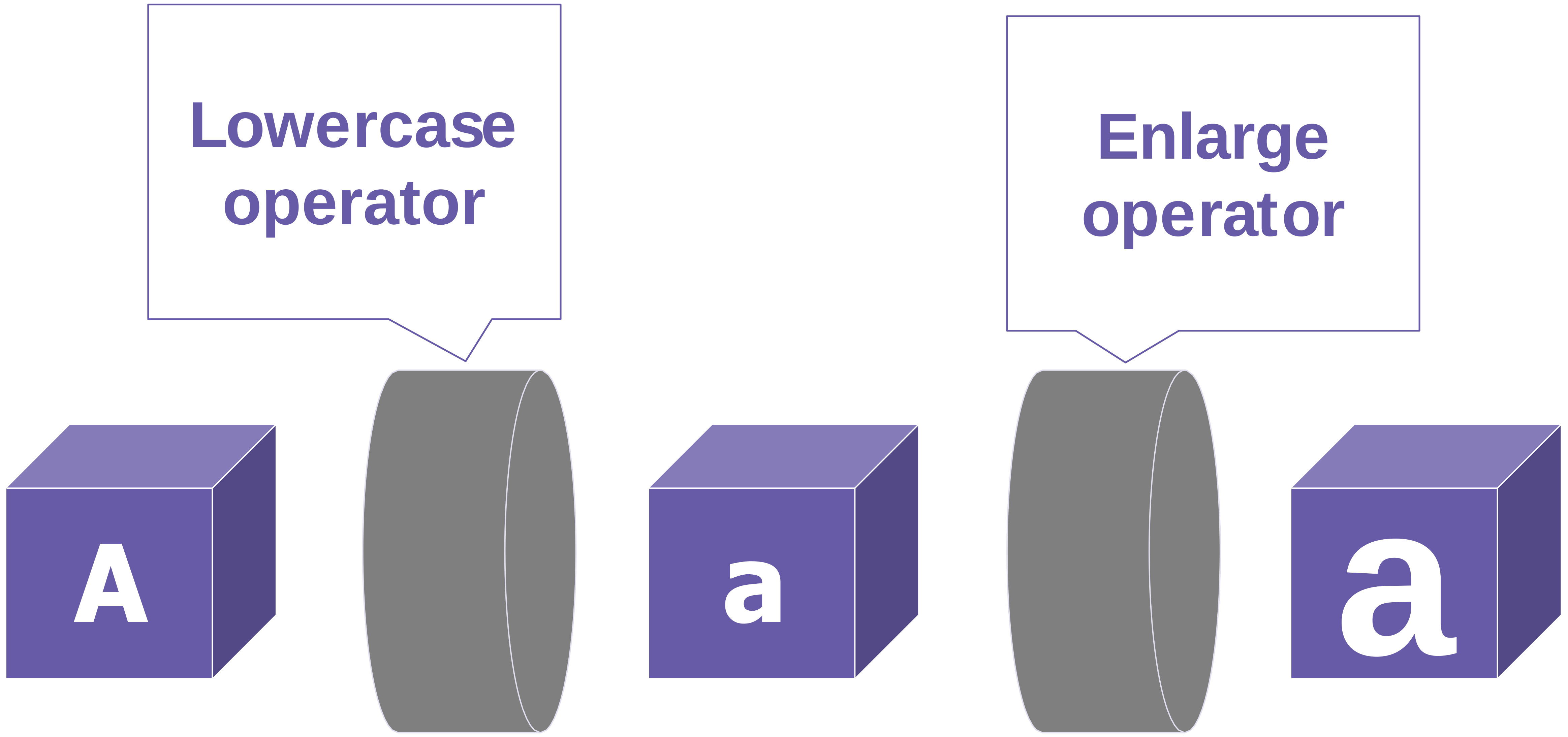
At some later point in time…

Application

- The response is returned
- The Observable emits a `next` notification
- We process the emitted response



Observable Pipe



Common Observable Usage

Start the Observable (subscribe)

Pipe emitted items through a set of operators

Process notifications: next, error, complete

Stop the Observable (unsubscribe)

Example

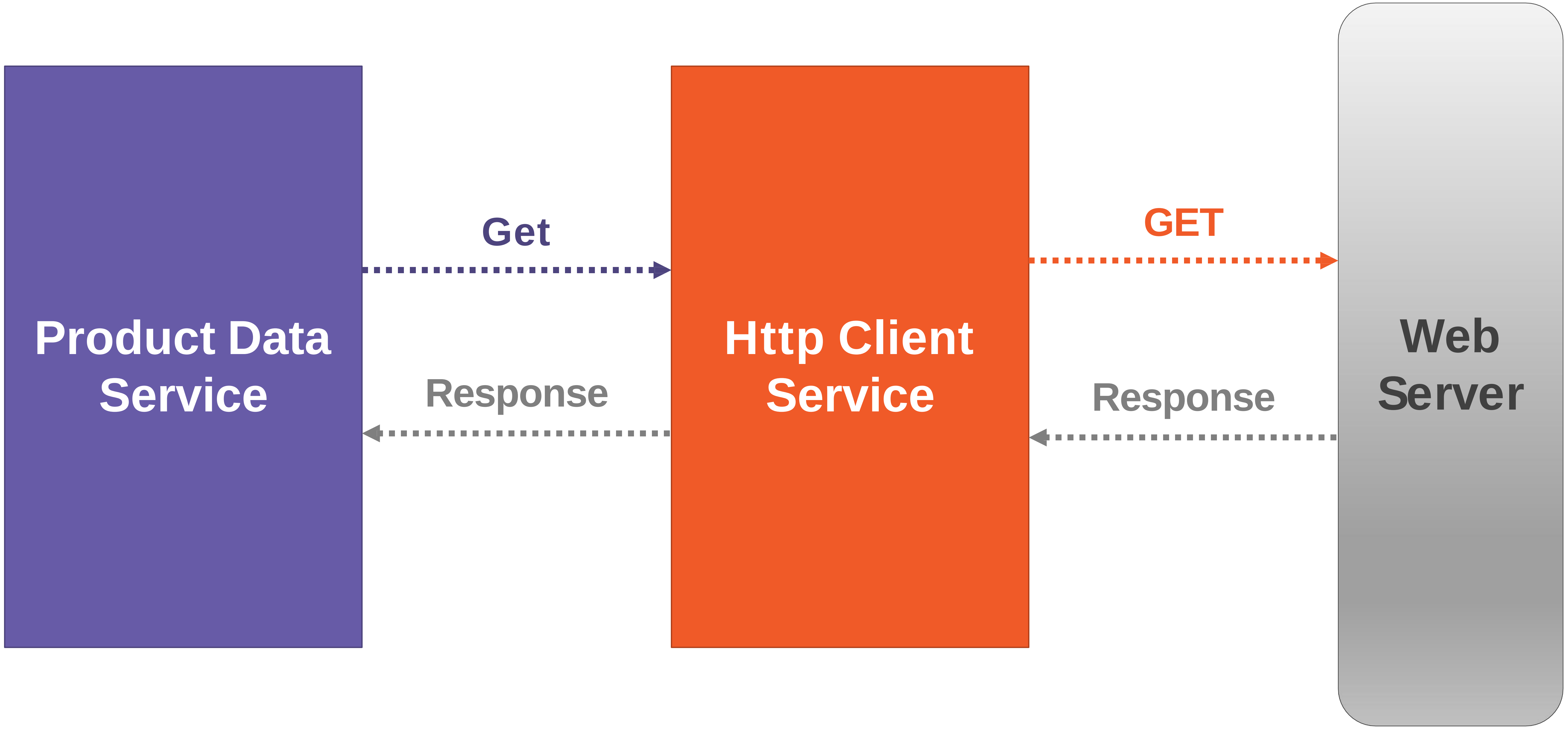
Example

```
import { Observable, range, map, filter } from 'rxjs';


const source$: Observable<number> = range(0, 10);


source$.pipe(
   map(x => x * 3),
   filter(x => x % 2 === 0)
).subscribe(x => console.log(x));
```

Result

```
0
6
12
18
24
```

Sending an HTTP Request



Subscribing to an Observable

```
x.subscribe()

x.subscribe(Observer)

x.subscribe({
    nextFn,
    errorFn,
    completeFn
})

const sub = x.subscribe({
     nextFn,
     errorFn,
     completeFn
})
```

Unsubscribing from an Observable

Store the subscription in a variable

Implement the OnDestroy lifecycle hook

Use the subscription variable to unsubscribe