Angular and Web Development

Part 2

SWEN-261k
Introduction to Software Engineering

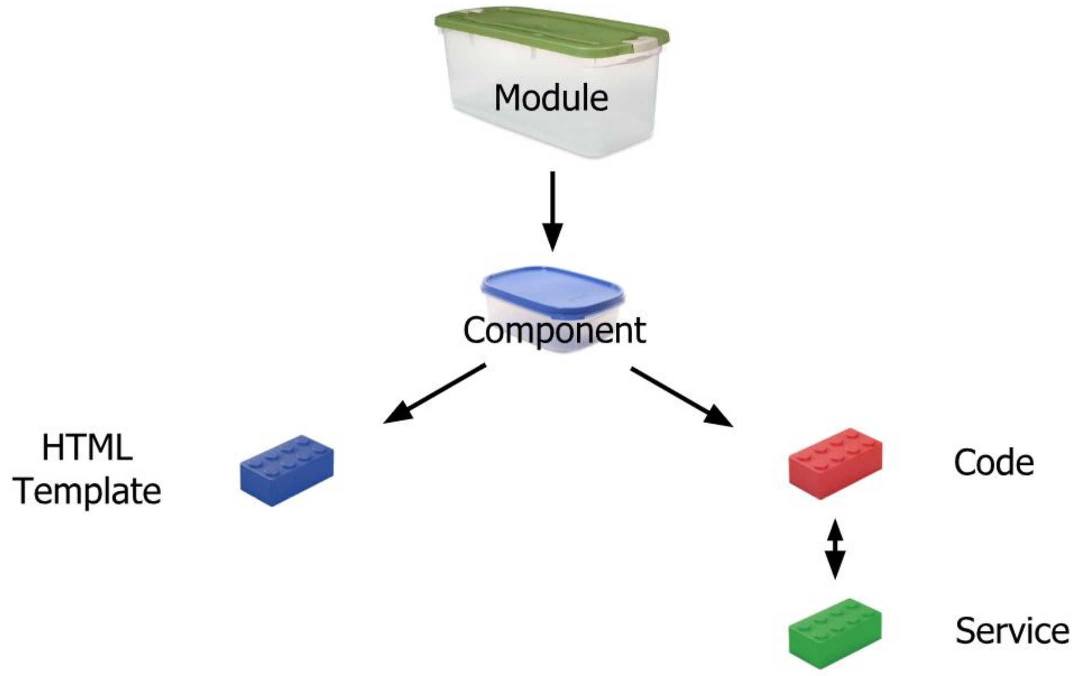
Department of Software Engineering
Rochester Institute of Technology

Software Engineering
Rochester Institute of Technology

Summary

- **Angular** is a component-based framework that is used for developing single page applications employing TypeScript and HTML template language
  - Typescript is a language that compiles to JavaScript. It is **strongly typed**, object oriented and compiled language
  - HTML (Hypertext Markup Language) is the code that is used to structure a web page and its content
  - CSS (Cascading Style Sheets) is the language we use to style a Web page
- Last class we covered **Modules** and **Components**
  - Modules are objects that help you to **organize dependencies** into discrete units
  - Components are new elements that will compose the majority of your application’s structure and logic

Modules vs Components



Module	Component
A module is a **collection** of components, services, directives, pipes and so on	A component in Angular is a building block of the Application with an **associated template**
Denoted by **@NgModule**	Denoted by **@Component**
An Angular application will contain many modules, each dedicated to a **single purpose**	Each component can **use other components**, which are declared in the same module. To use components declared in other modules, they need to be ***exported*** from this module and the module needs to be ***imported***. **Note**: (**>=** v17 defaults to standalone use *ng new --no-standalone* for use of @NgModule)

Angular – What’s next

- Data binding
- Services
- Routing
- Observables

Angular – Data Binding

- **Data binding** automatically keeps your page up-to-date based on your application's state. You use data binding to specify things such as the source of an image, the state of a button, or data for a particular user
- There are four types of data binding available in Angular:
  - **Event binding -** This data binding type is when information flows from the view to the component when an **event is triggered**
  - **Interpolation -** Text **representing variables** in components are placed in between double curly braces in the template
  - **Two-way data binding -** Two-way binding is a mechanism where data flows **both ways** from the component to the view and back
  - **Property binding -** Property binding is a one-way mechanism that lets you set the property of a **view element**

Data Binding – Event Binding

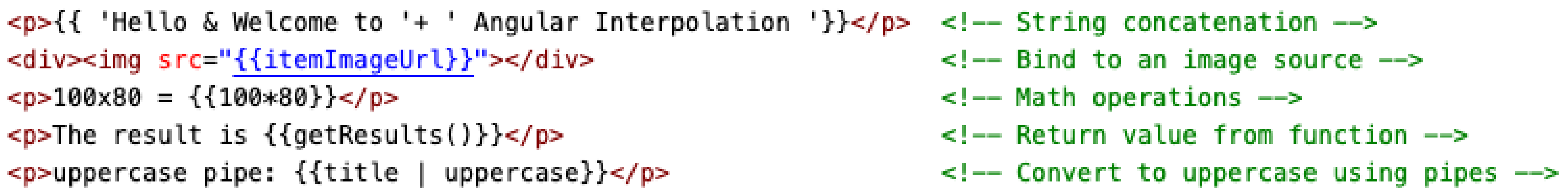
- To bind to an **event**, you use the Angular event binding syntax
- This syntax consists of a **target event** name within parentheses to the left of an equal sign, and a quoted **template statement** to the right
- In the following example, the target event name is click and the template statement is onSave():

$$\texttt{<button }\underbrace{\texttt{(click)}}_{\text{Target event name}}\texttt{ = "}\underbrace{\texttt{onSave()}}_{\text{Template statement}}\texttt{">Save</button>}$$

- Template statements are methods or properties that you can use in your HTML to respond to user events

Data Binding – Interpolation

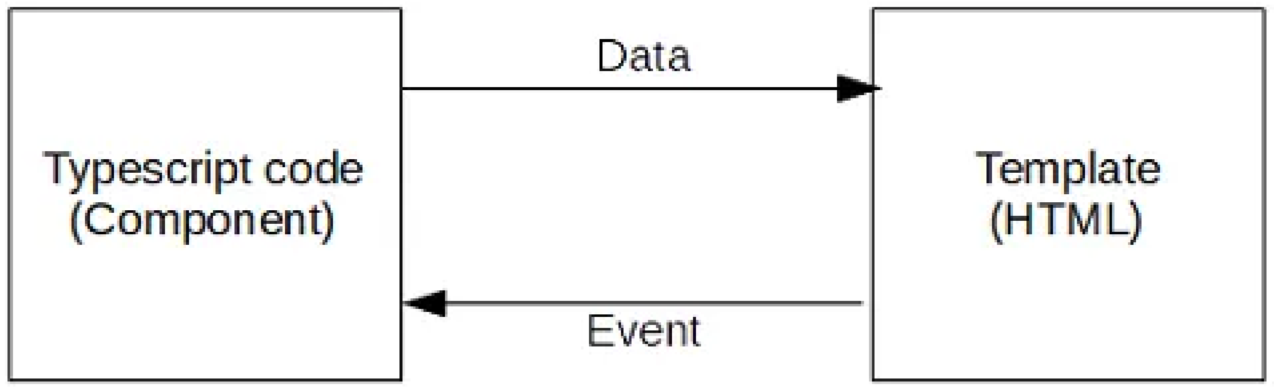
- **Interpolation** is used for one-way data binding
- It moves data in one direction from our components to HTML elements
- Angular evaluates the expressions into a string and replaces it in the original string and updates the view
- Angular uses the **{{ }}** in the template to denote the interpolation
- Examples:



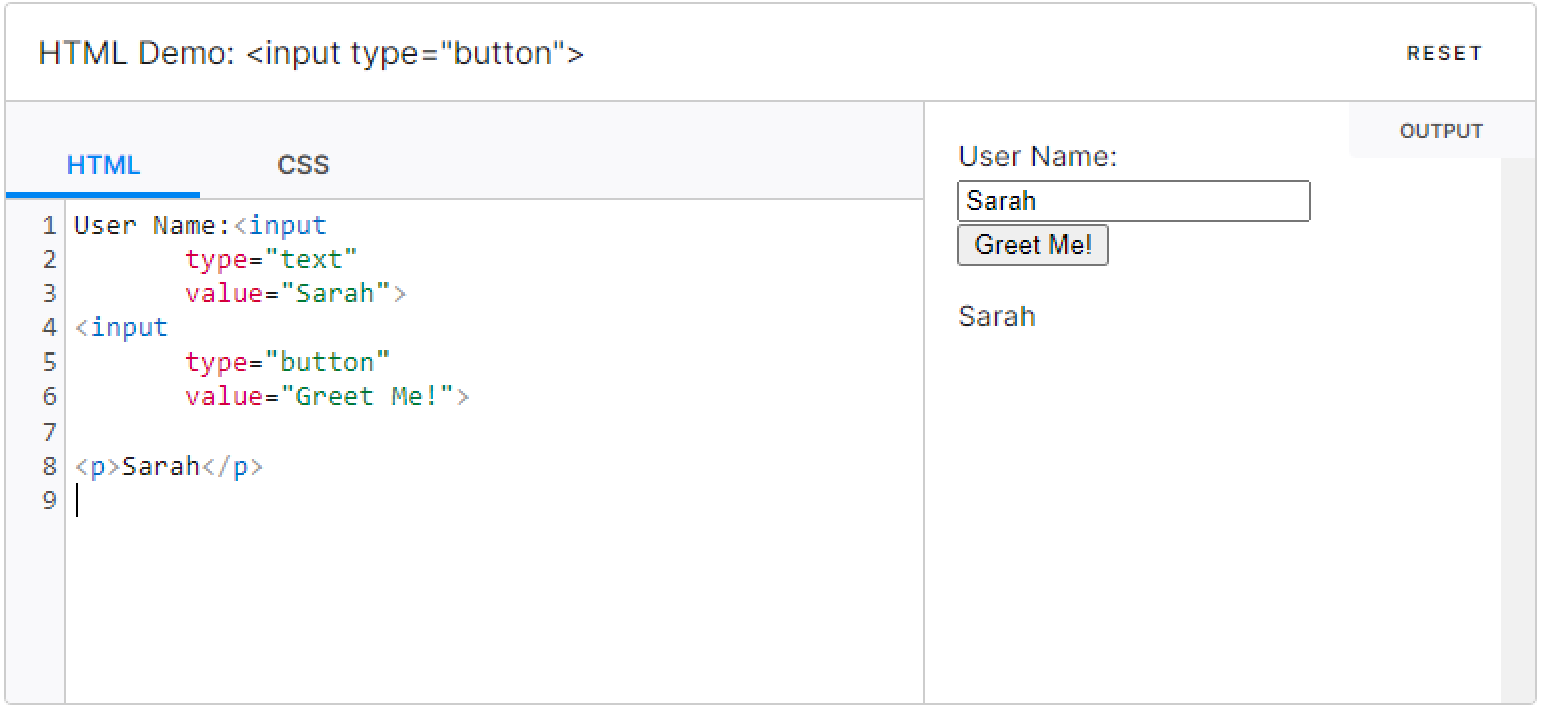
```
<p>{{ 'Hello & Welcome to '+ ' Angular Interpolation '}}</p>  <!-- String concatenation -->
<div><img src="{{itemImageUrl}}"></div>                        <!-- Bind to an image source -->
<p>100x80 = {{100*80}}</p>                                     <!-- Math operations -->
<p>The result is {{getResults()}}</p>                          <!-- Return value from function -->
<p>uppercase pipe: {{title | uppercase}}</p>                   <!-- Convert to uppercase using pipes -->
```

Data Binding – Two-way Data Binding

- **Two-way data binding** in Angular will help users to exchange data from the component to view and from view to the component
- It will help users to establish communication bi-directionally.
    - If a **property** in the component is **changed** that change flows to the view
    - Same way **change** in view is reflected in the bound property in the component



Input example on HTML


HTML Demo: <input type="button">

```
User Name:<input
       type="text"
       value="Sarah">
<input
       type="button"
       value="Greet Me!">

<p>Sarah</p>

```

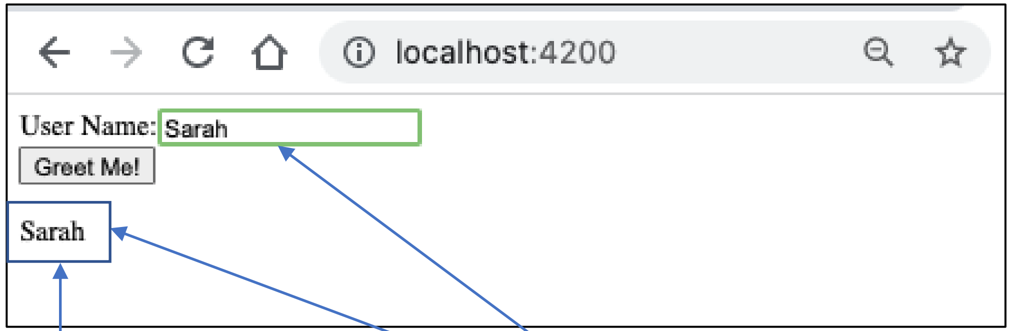
```
User Name:
Sarah
Greet Me!

Sarah
```

Data Binding – Two-way Data Binding

- In Angular, **`ngModel`** directive is used for two-way bindings
- It simplifies creating two-way data bindings on form elements like input elements

Two-way data binding for "name" element

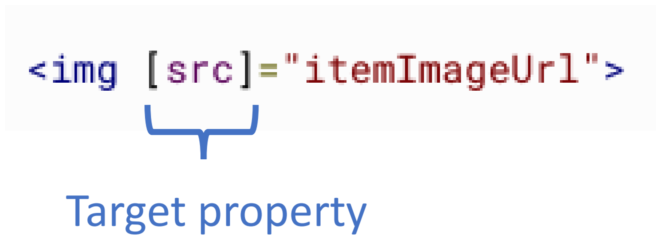
```
    Enter Your Names: <input type="text" [(ngModel)]="name"><br/>
    <button (click)="greet()">Greet Me!</button>

    <p>{{name}}</p>
</div>
```



As you type in a new value for "name", all references are immediately updated in template and component class

Data Binding – Property Binding

- **Property binding** in Angular helps you set values for **properties of HTML** elements or directives
- Use property binding to do things such as **toggle** button functionality, **set paths** programmatically, and **share values** between components
- Property binding moves a value in **one direction**, from a component's property into a target element property
- To bind to an element's property, enclose it in square brackets, [], which identifies the property as a target property



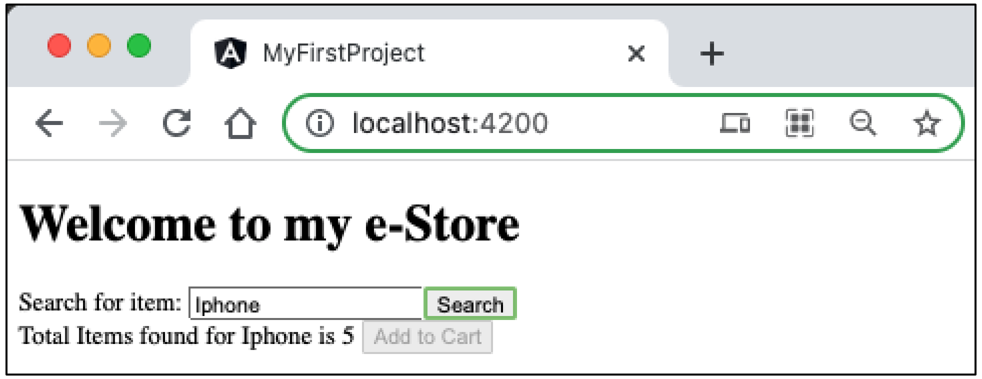
```
<img [src]="itemImageUrl">
```

Angular – Data Binding Example

- Using Data binding, we can pass data between the component and template

greet.component.ts

```
import { Component, OnInit } from '@angular/core';
import { LogService } from '../log.service';

@Component({
  selector: 'app-greet',
  templateUrl: './greet.component.html',
  styleUrls: ['./greet.component.css']
})
export class GreetComponent implements OnInit {

  constructor(private logger: LogService) { }

  ngOnInit(): void {
  }

  title: string = "Welcome to my e-Store";
  isDisabled = true;
  item: string = "item";
  searchItem: string ="";
  numItems = 0;

  searchItems(): void {
    this.numItems = 5;
    this.searchItem = this.item;
  }
}
```



Welcome to my e-Store

Search for item: Iphone

Total Items found for Iphone is 5

greet.component.html

```
<h1 [innerText]="title"></h1>

<div>
    Search for item: <input [(ngModel)]="item"/>
    <button (click)="searchItems()">Search</button><br/>
    Total Items found for {{searchItem}} is {{numItems}}
    <button [disabled]="isDisabled">Add to Cart</button>
</div>
```

Event binding

Property binding

Interpolation

Two-way data binding

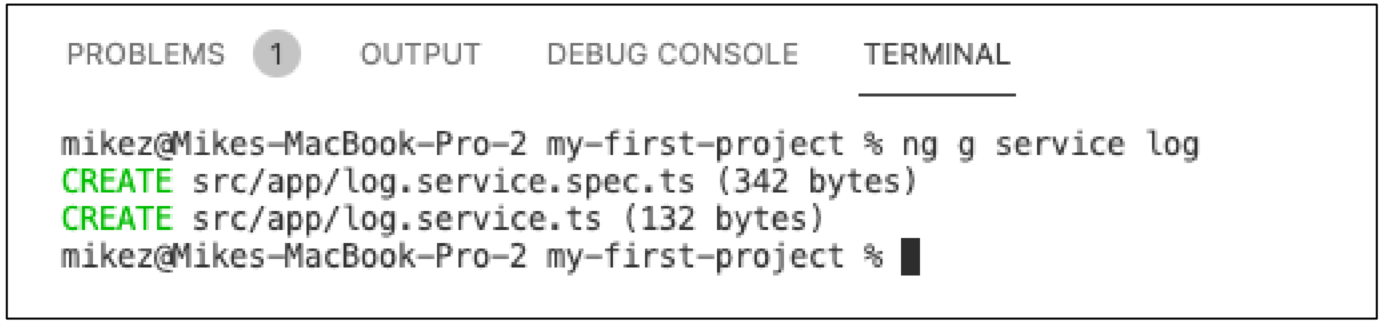
Services

- Angular **services** are singleton objects that get instantiated only once during the lifetime of an application
- They contain methods that maintain data **throughout the life of an application**, i.e. data does not get refreshed and is available all the time
- The main objective of a service is to organize and **share** business logic, models, or data and functions with **different components** of an Angular application
- Services are a great way to share information among classes that *don't know each other*

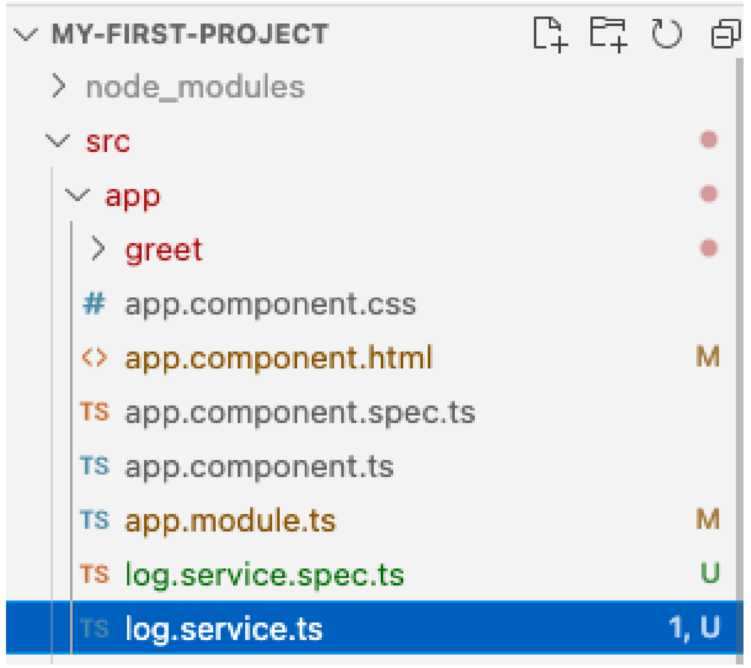
Angular Example – Create Service

- Use the Angular CLI to generate a service for a simple logger

```
ng g service log
```



```
mikez@Mikes-MacBook-Pro-2 my-first-project % ng g service log
CREATE src/app/log.service.spec.ts (342 bytes)
CREATE src/app/log.service.ts (132 bytes)
mikez@Mikes-MacBook-Pro-2 my-first-project % 
```



```
MY-FIRST-PROJECT
  > node_modules
  v src
    v app
      > greet
      app.component.css
      app.component.html          M
      app.component.spec.ts
      app.component.ts
      app.module.ts               M
      log.service.spec.ts         U
      log.service.ts           1, U
```

Angular Example – Service Details

- Add a new log() method to log messages to the console

log.service.ts

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

@Injectable({
  providedIn: 'root'
})
export class LogService {

  constructor() { }

  log(msg: any) {
    console.log(new Date() + ": " + JSON.stringify(msg));
  }
}
```

@Injectable() decorator to provide the metadata that allows Angular to inject it into a component as a dependency

New log method

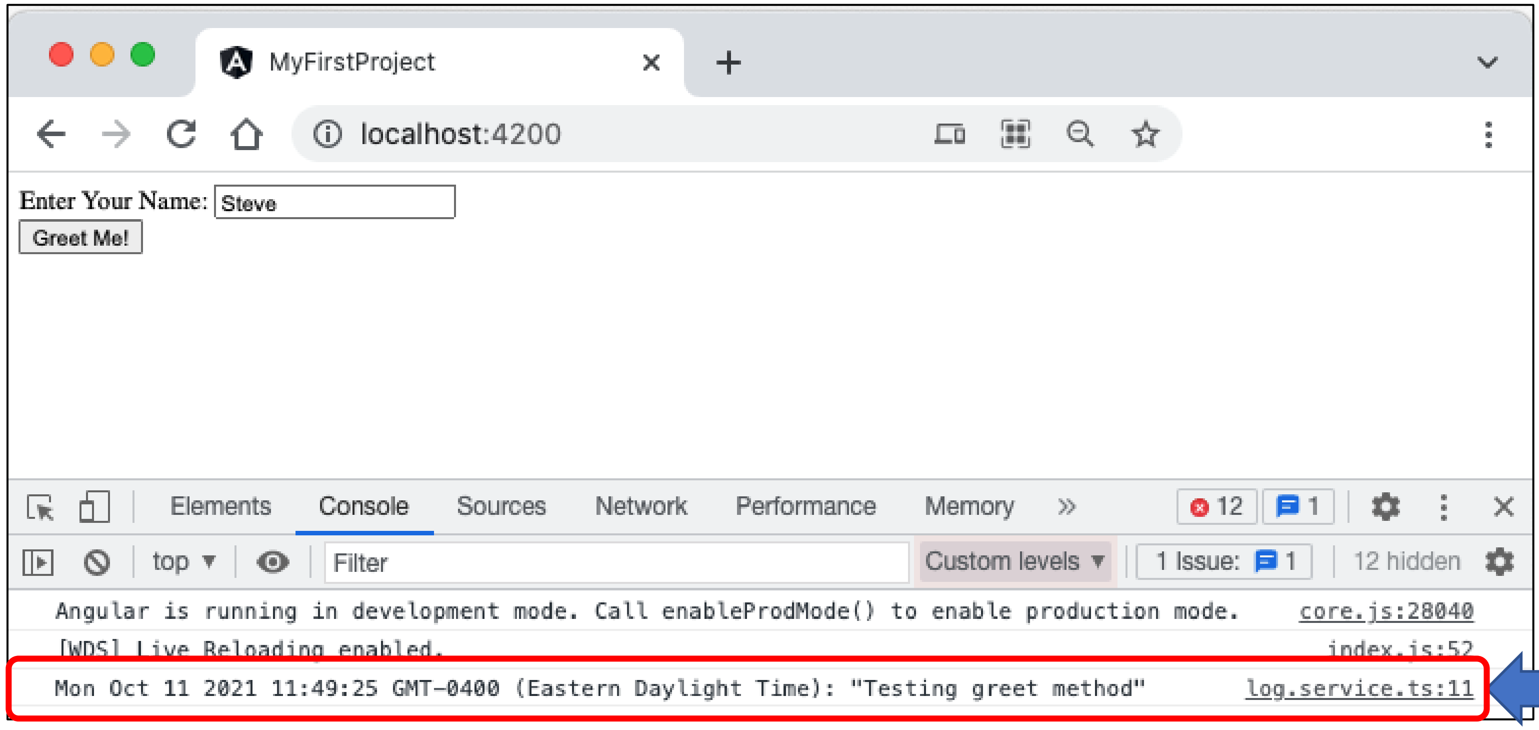
greet.component.ts

```
import { Component, OnInit } from '@angular/core';
import { LogService } from '../log.service';


@Component({
  selector: 'app-greet',
  templateUrl: './greet.component.html',
  styleUrls: ['./greet.component.css']
})
export class GreetComponent implements OnInit {

  constructor(private logger: LogService) { }

  ngOnInit(): void {
  }

  name: string = "Steve";

  greet(): void {
    this.logger.log("Testing greet method");
  };

}
```

The log service is **injected** into the greet component

Angular Example – Service Details

```
ng serve
```

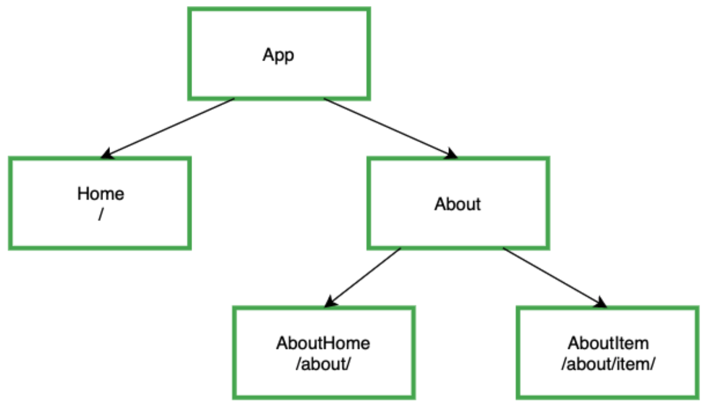


```
Angular is running in development mode. Call enableProdMode() to enable production mode.   core.js:28040
[WDS] Live Reloading enabled.                                                             index.js:52
Mon Oct 11 2021 11:49:25 GMT-0400 (Eastern Daylight Time): "Testing greet method"       log.service.ts:11
```

Using Chrome's developer tools, we can see our message logged to the console when the button is clicked

Routing

- Most applications require the ability to navigate between different pages during the lifecycle of the application.
- Typically, an application has at least a few basic pages, such as a login page, home page, user’s account page, and so forth.
- **Routing** is the term used to describe the capability for the application to change the content on the page as the user navigates around.
- The Angular **router** is a core part of the Angular platform



Routing

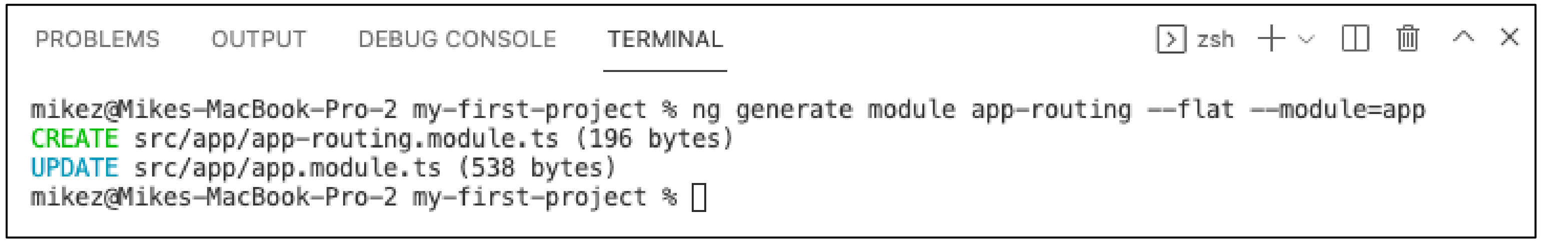
- In Angular, the best practice is to load and configure the router in a separate, top-level **module** that is dedicated to routing and imported by the root AppModule`
- Use the Angular CLI to generate

Note: (>= v17 defaults to standalone use ***ng new my-app --no-standalone –routing*** to generate **app.module.ts** and **app-routing.module.ts**)

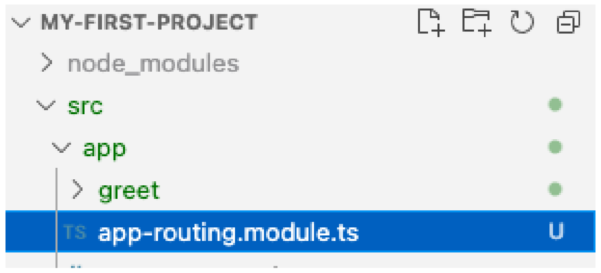
```
ng generate module app-routing --flat --module=app
```

Puts the file in **src/app**

register it in the imports array of the AppModule.



```
mikez@Mikes-MacBook-Pro-2 my-first-project % ng generate module app-routing --flat --module=app
CREATE src/app/app-routing.module.ts (196 bytes)
UPDATE src/app/app.module.ts (538 bytes)
mikez@Mikes-MacBook-Pro-2 my-first-project % 
```



Routing

- In this example, we will create a routes to a home, about and dashboard page by updating the new app-routing module

app-routing.modules.ts

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

import { GreetComponent } from './greet/greet.component';
import { AboutComponent } from './about/about.component';
import { DashboardComponent } from './dashboard/dashboard.component';

const routes: Routes = [
{ path: 'home', component: GreetComponent },
{ path: 'about', component: AboutComponent },
{ path: 'dashboard', component: DashboardComponent }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

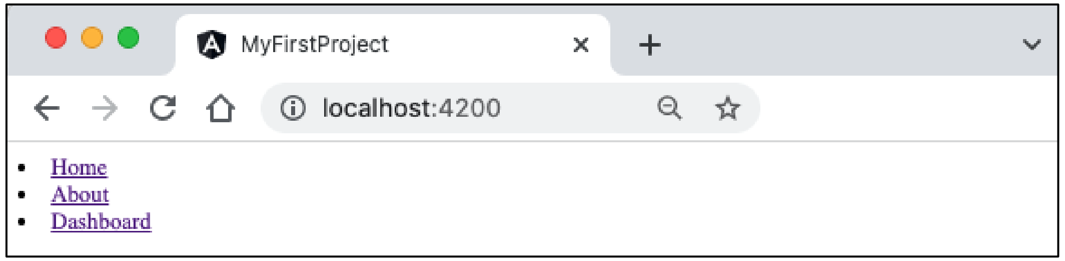
Import components we want to route to

Each **route** has 2 properties:
- **path** – String that matches URL in browser. Maps to a component
- **component** – the component the router should created when navigating to this route

- **import** - Register the top-level routes and return the routing module that should be imported by the root module of the application
- **export** - exports RouterModule so it will be available throughout the application

Routing

- Our new start page links to other pages



MyFirstProject

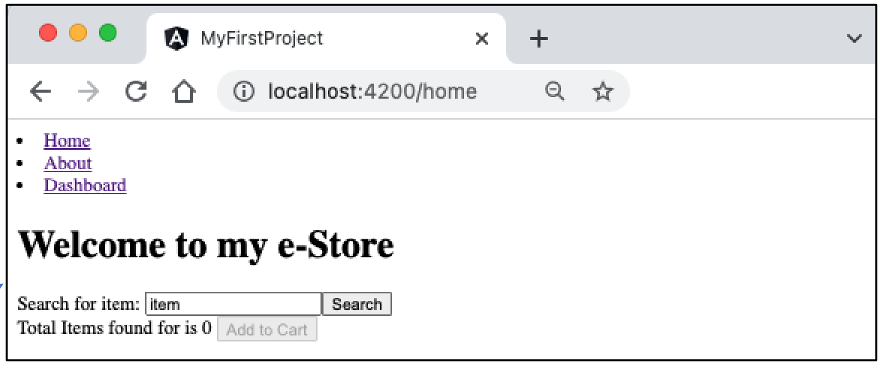
localhost:4200

- Home
- About
- Dashboard

app.component.html

```
<div>
<nav>
  <li><a routerLink="home">Home</a></li>
  <li><a routerLink="about">About</a></li>
  <li><a routerLink="dashboard">Dashboard</a></li>
</nav>
  <router-outlet></router-outlet>
</div>
```

The <router-outlet> tells the router where to display routed views



MyFirstProject

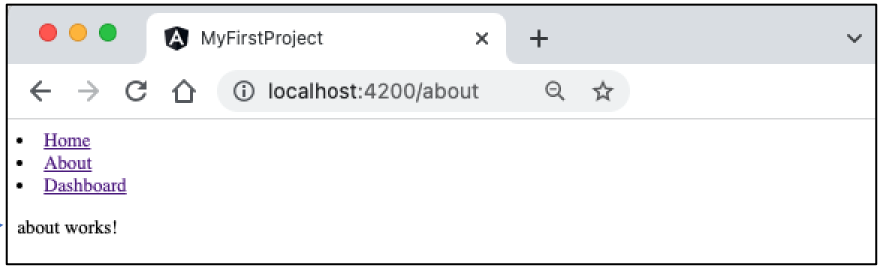
localhost:4200/home

- Home
- About
- Dashboard

Welcome to my e-Store

Search for item: item Search

Total Items found for is 0 Add to Cart

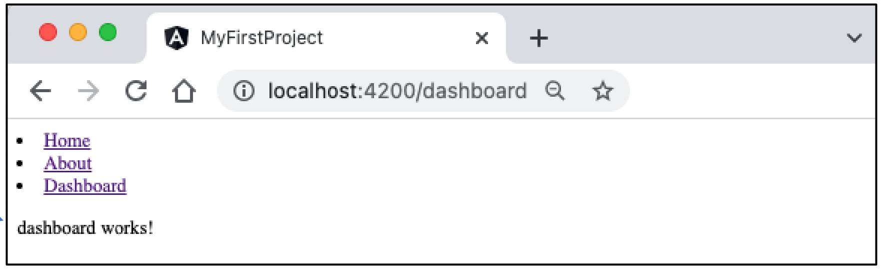


MyFirstProject

localhost:4200/about

- Home
- About
- Dashboard

about works!



MyFirstProject

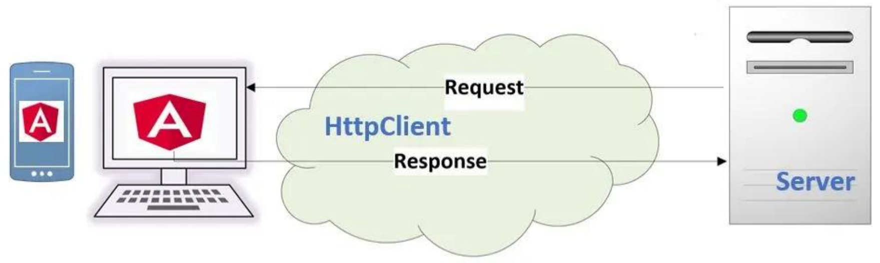
localhost:4200/dashboard

- Home
- About
- Dashboard

dashboard works!

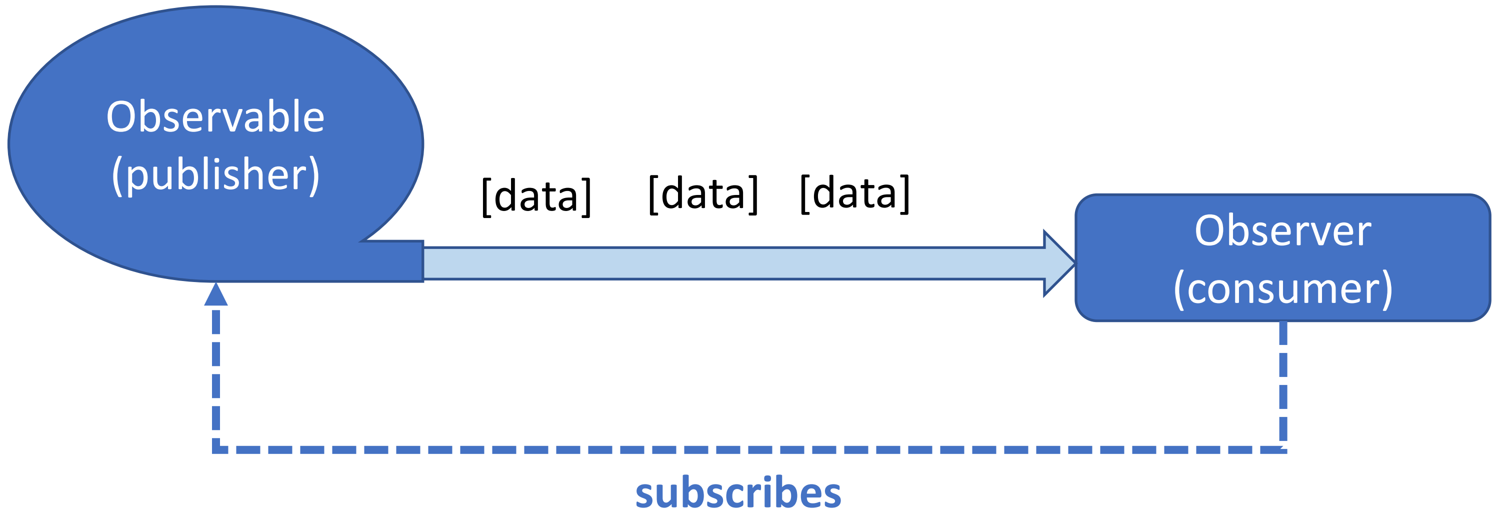
Observables

- **Observables** provide support for passing messages between parts of your application
- They are used frequently in Angular and are a technique for **event handling**, **asynchronous programming**, and handling **multiple values**
- For example, consider requesting data from a server via HTTP
  - If the content was retrieved synchronously (following the request), the browser could freeze the UI while it waited for the server's response
  - Instead, we want to be notified when when the content is available



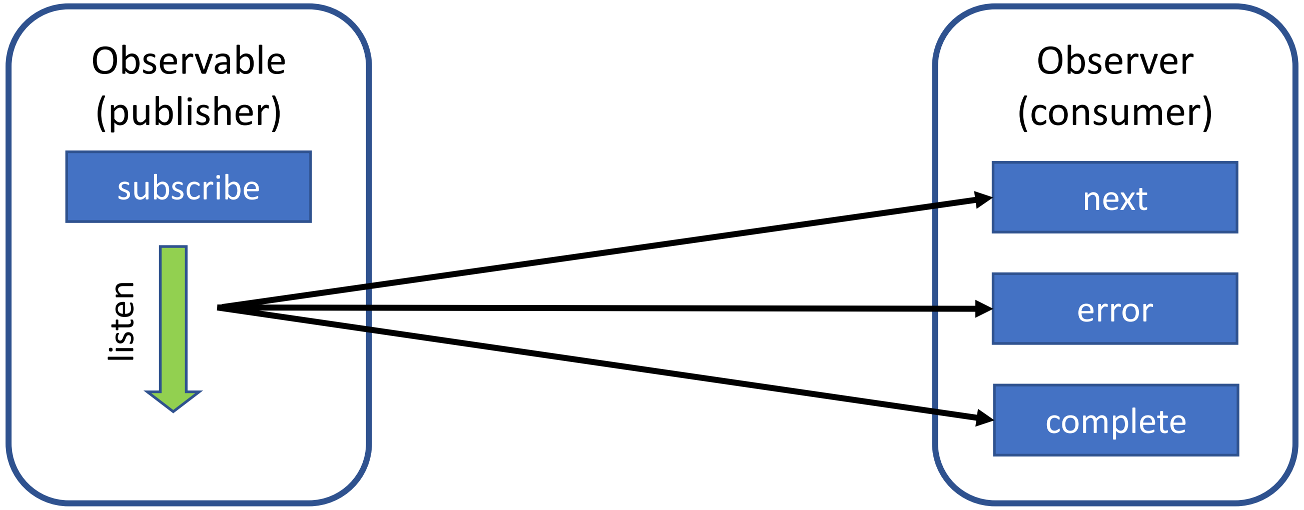
Observables

- Observables are declarative—that is, you define a function for **publishing** values, but it is not executed until an observer (consumer) **subscribes** to it
- The subscribed consumer then receives **notifications until the function completes**, or until they unsubscribe



Observers

- The **Observer** has **three handles** to use the data that it receives:
  - **next** - Required. A handler for each delivered value that's called zero or more times after execution starts
  - **error** - Optional. A handler for an error notification. An error halts execution of the observable instance
  - **complete** - Optional. A handler for the execution-complete notification. Delayed values can continue to be delivered to the next handler after execution is complete.



Observables – Simple Example

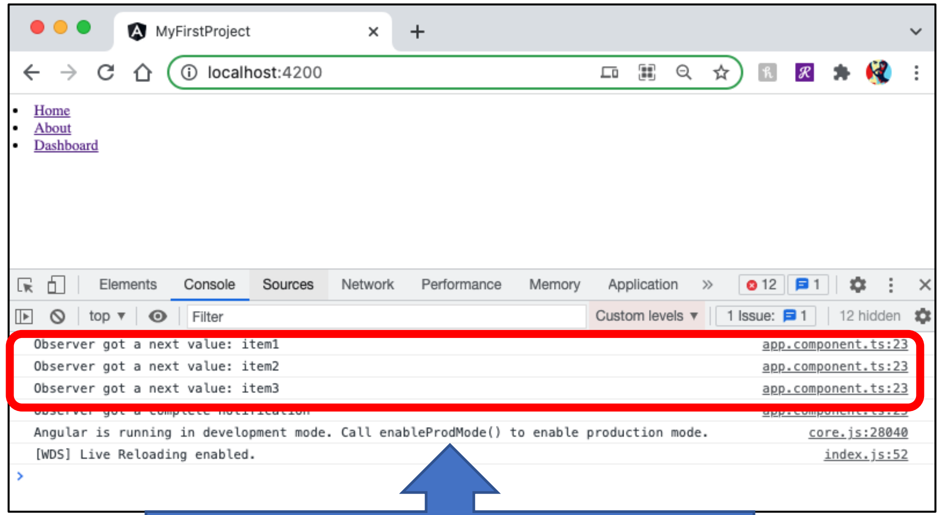
- In this example, we create a simple Observable that publishes a list of items that are subscribed to by an Observer

app.component.ts

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

const myObservable = of("item1", "item2", "item3")

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent implements OnInit {

ngOnInit() {

  myObservable.subscribe(
    x => console.log('Observer got a next value: ' + x),
    err => console.error('Observer got an error: ' + err),
    () => console.log('Observer got a complete notification')
   );
  }
}
```

Observable object

Execute the observer object

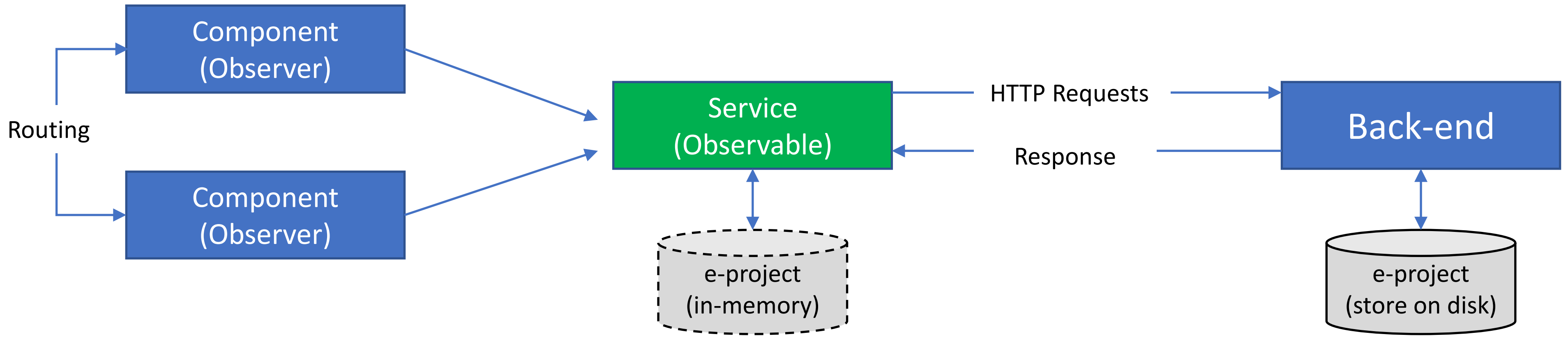


```
Observer got a next value: item1                                   app.component.ts:23
Observer got a next value: item2                                   app.component.ts:23
Observer got a next value: item3                                   app.component.ts:23
Observer got a complete notification                               app.component.ts:23
Angular is running in development mode. Call enableProdMode() to enable production mode.   core.js:28040
[WDS] Live Reloading enabled.                                      index.js:52
```

Using Chrome's developer tools, we can see our message logged to the console when the button is clicked

Observables – For your project

- Consider creating an **Observable service** which will process requests for your e-project
  - Initially the service could hold the data for your e-project items until you connect to your back-end API
- Your **other components** would **subscribe to the service** for processing requests
- When you connect your service to the back-end, your components do not have to change since the service will already be processing the requests



- Part 2 of the Hero's tutorial connects to a <u>temporary</u> in-memory data store
- Additional instructions are provided to connect the service to the back-end APIs similar to what you will do on your e-project

Angular Activity – Tour of Heroes – Part 2

- Do Activity “Tour of Heroes – Part 2”
- Complete the remaining tutorial
- Upon completion of the tutorial, you have all the necessary components to build your e-project!