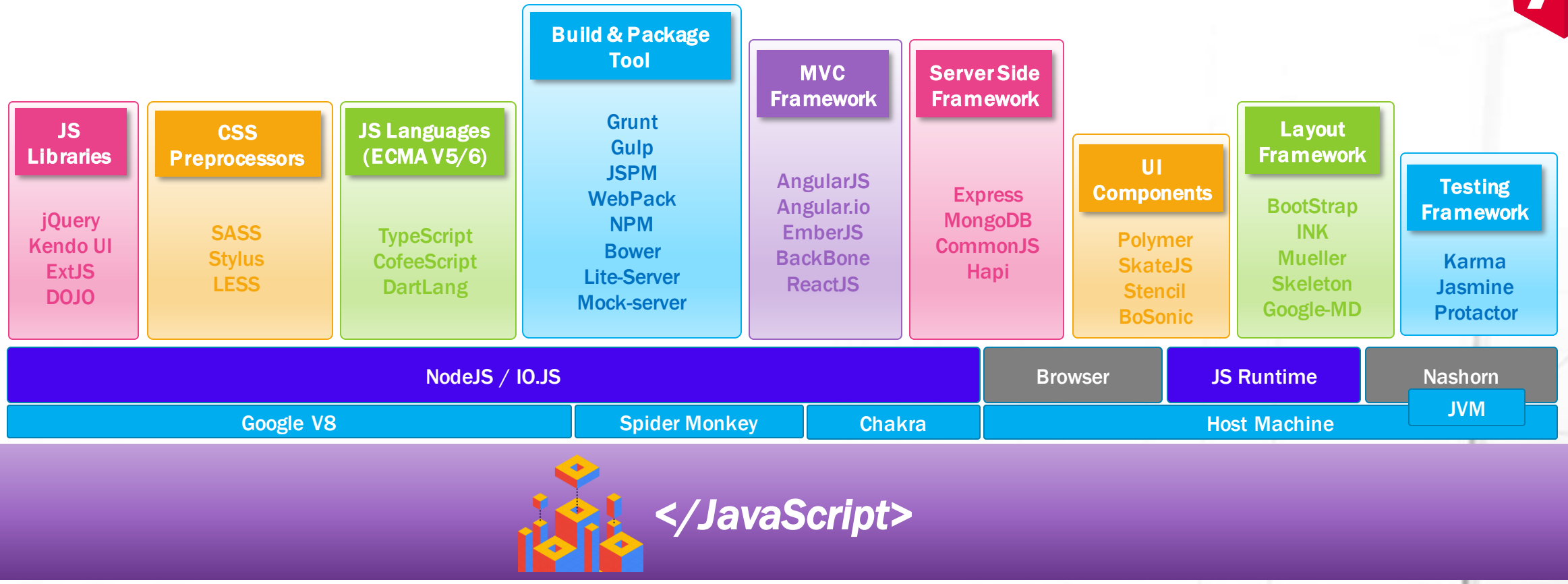
<anglular.io/>

Training Sessions

Agenda

- JavaScript Ecosystem
- Paradigm Shift
- Angular Introduction
- What is Angular
- Architecture
- Associated Frameworks
- TypeScript
- Angular Components

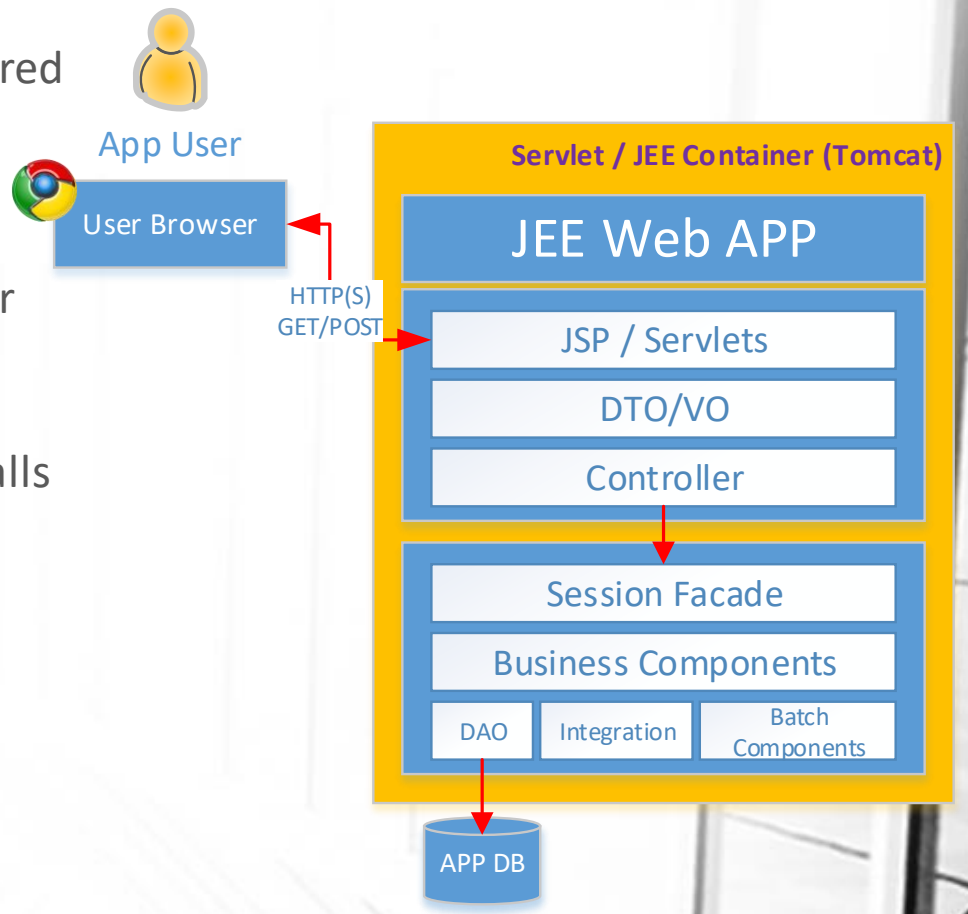
JavaScript Eco System -*



- * → indicates only the popular & widely used tools and frameworks
- Mature Eco System with enterprise grade tools and framework
- Cohesive & Self Contained Feature set
- Relatively Light weight Solution & eases various deployment level issues

The As-IS State Implementation

- Deployed as WAR/EAR, and contains both Presentation and Business logic components, unless distributed into different nodes.
- The Presentation layer would get processed in the Server Side, and rendered on the user browser.
- Upon rendering, the browser gets a chunk of payload containing HTML elements such as HTML Code, JS, CSS, and Data.
- The predominant communication between Browser and the Server is over HTTP(S), and GET and POST used.
- Uses HTTP's Session store for persisting user context data.
- The Browser is responsible for DOM manipulation, UI validations, AJAX Calls only.
- The Page Navigation, Execution is being done on the Server Side.

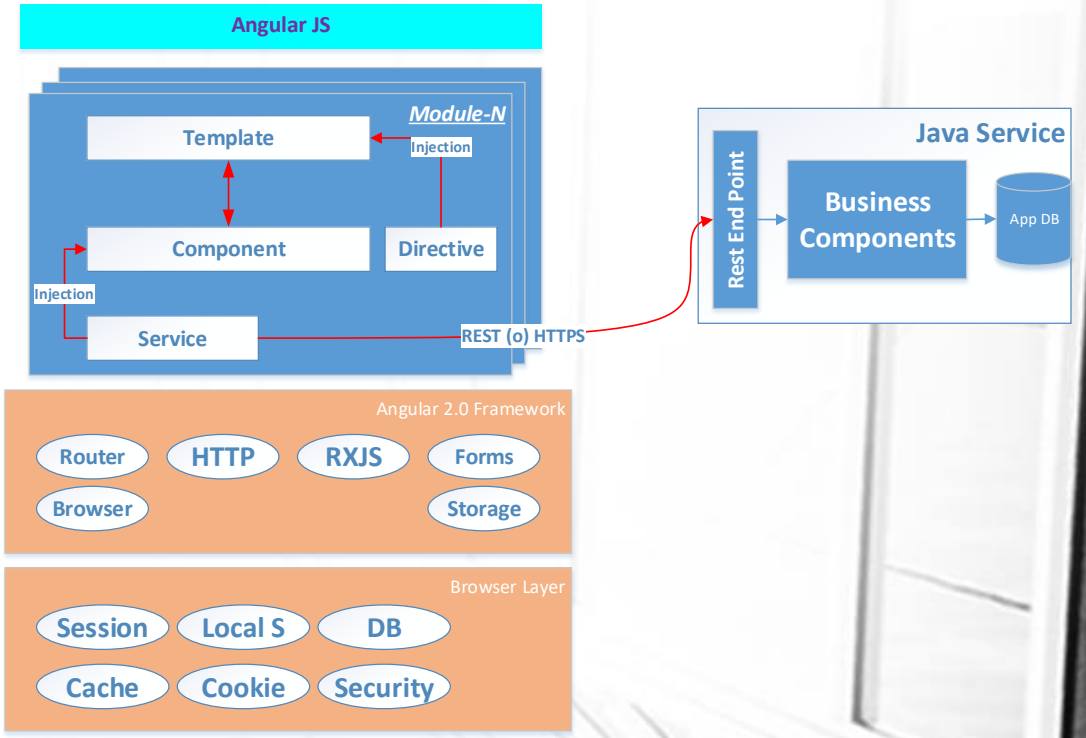


- **Pros**
  - Uni Technology Architecture
  - Easier to Maintain & Develop
  - Needs only Java Team
- **Cons**
  - Multi Channel/layout Capability. ?
  - Responsiveness ?
  - Not harnessing the Browser Capability. ?
  - Relatively Heavy Weight Solution. ?
  - Limited Scalability. ?

The Angular Implementation

- Based on SPA Concept, and entire UI layer is developed using Angular Stack
- Module Design – the functionality of the applications can be grouped as modules, and managed separately
- Angular itself architected as a collection of modules such as FormsModule, CoreModue, ServerSIdeModule etc
- Has native integration with RWD tools such as BootStrap, MediaQueries etc
- The UI Application gets downloaded for the first time, and gets executed on the browser for all its need. Ex For Routing, Validation, Rendering, DOM Manipulation, Server Side Calls, Data Binding etc.
- REST Protocol over HTTPS being used to communicate between Browser and the Server layer.
- JSON is the format of data being used.
- Uses REST Verbs such as GET,POST,UPDATE,DELETE for all its actions.

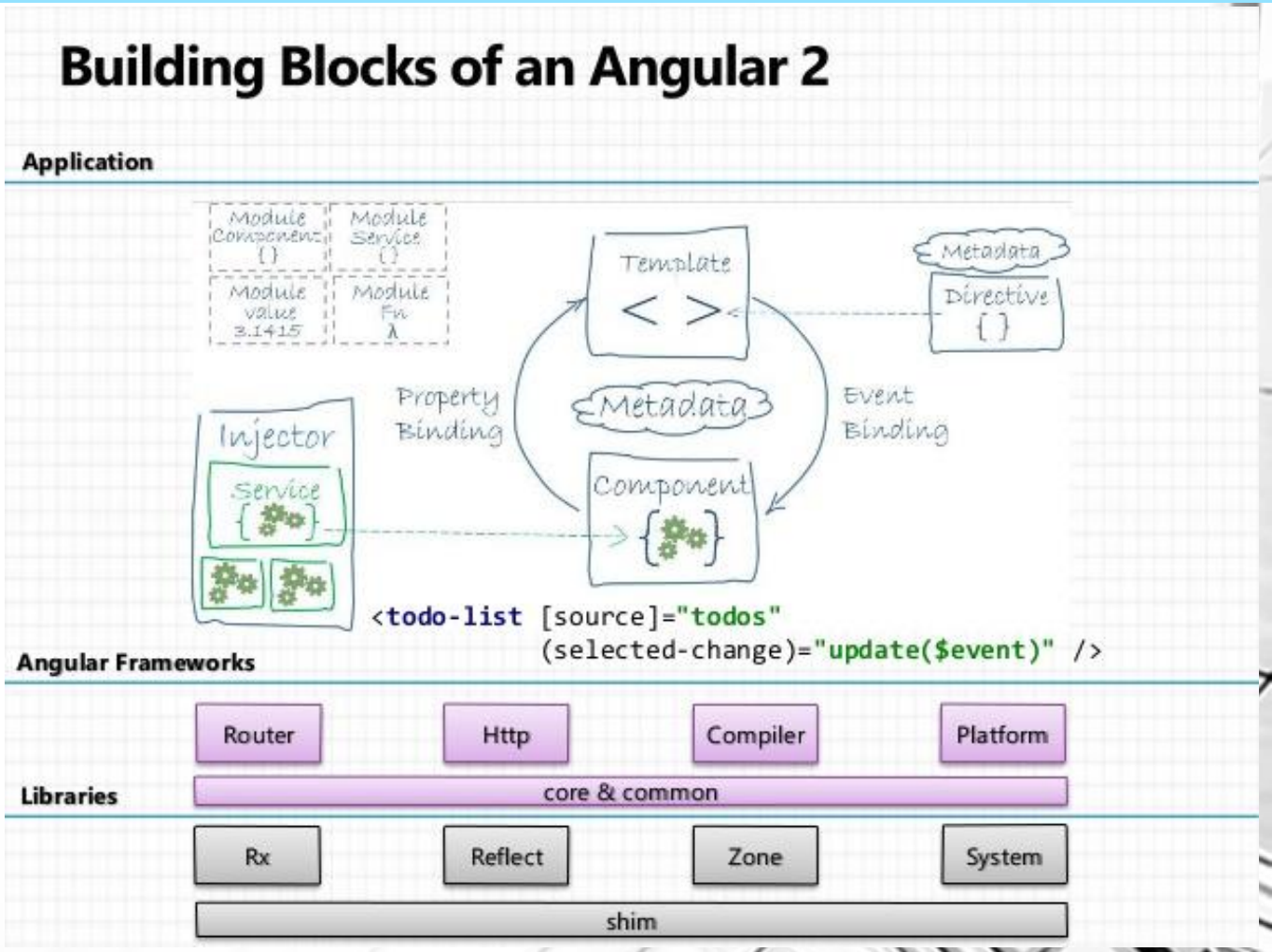
- Pros
  - Java Script based JS Framework backed by Google.
  - Faster to Develop
  - Multi Layout/Channel Capability
  - Mobile enablement is faster & easier
  - High Scalability with the user of Promise & Observable.
  - Easier to Maintain & Develop
- Cons
  - New Technology to be learnt
  - One more technology in landscape



Angular Introduction

Angular JS is an open source framework built over JavaScript. This framework was used to overcome obstacles encountered while working with Single Page applications. Also, testing was considered as a key aspect while building the framework. It was ensured that the framework could be easily tested.

- TypeScript based open-source front-end web application platform
- Angular 2 uses a hierarchy of components as its main architectural concept
- The desktop development is much easier when mobile performance issues are handled first
- Works well in modern browsers
- Improved dependency injection – bindings make it possible for dependencies to be named
- Reactive programming support using RxJS
- Modularity – much core functionality has moved to modules, producing a lighter, faster core
- Angular also includes the benefits of ES6:Lambdas
  - Iterators
  - For/Of loops
  - Python-style generators
  - Reflection
- TypeScrips is recommended one, optionally its supports DART,JavaScript



Angular Benefits

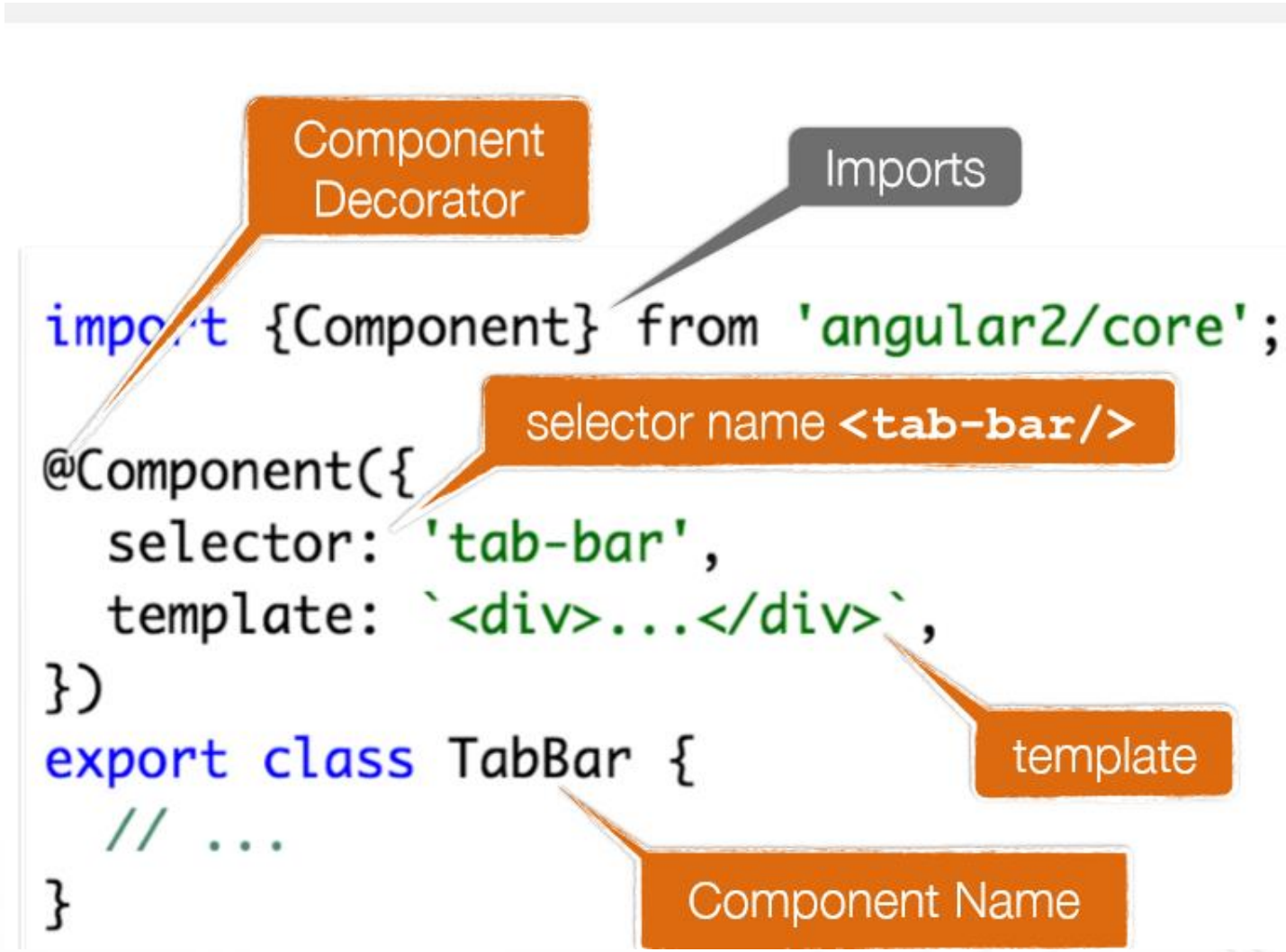
- Angular2/4 is 5x faster than AngularJS 1.x
- Loose Coupling
- Support for Web Components
- Ease in Templating, Form Field Management, Routing
- Server Side UI rendering
- Modular approach
- Native Mobile Support
- Rich Tool Set Support
  - IDE
  - Build Management
  - Development Support
  - Debugging Tools (browser based)
- Strong Static Typing
- ECMA 5/6 Compatibility
- Multi Language Support
  - TypeScript (Recommended)
  - Dart
  - JavaScript

Angular Building Blocks

- Data Binding
- Template
- Service
- Dependency Injection
- Module
- Component
- Metadata
- Directive

Pause for TypeScript

Angular 6 Components



Angular 6 Components

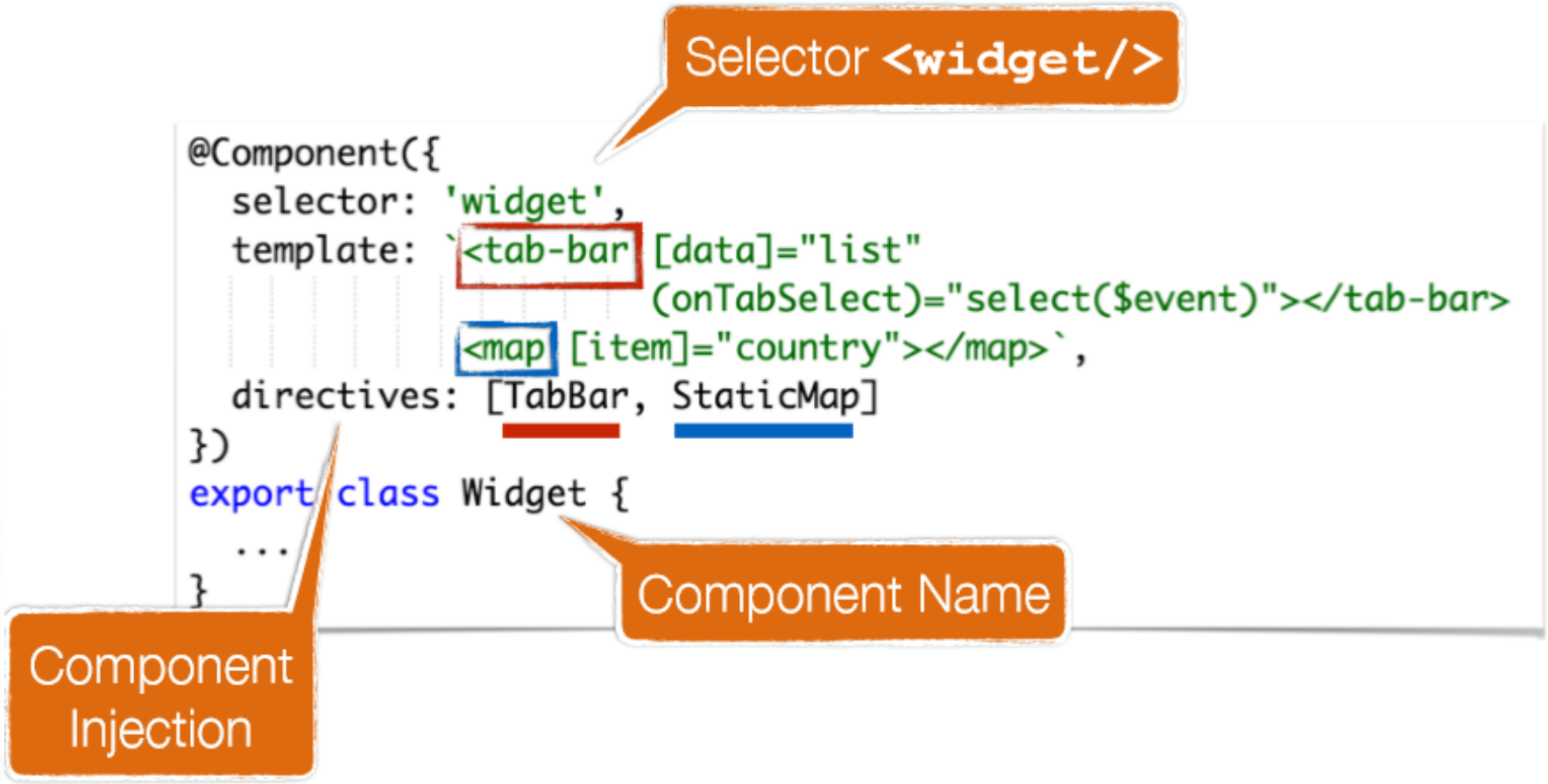

```
export  class Country {
  constructor(
    public id: string,
    public name: string,
    public coords: string) {
  }
}
```

Automatically generates class properties

Country.ts (model)

http://plnkr.co/edit/vIlrTYuOo6VJE7vMPEfQ?p=preview

Angular 6 Components



<widget/> (partial)

Angular 6 Components

```
const countries = [
  new Country( '1', 'Italy' , '42,13' ),
  new Country( '2', 'Greece', '42,25' ),
  new Country( '3', 'Usa', '40.7,-73' ),
];

@Component({
  selector: 'widget',
  template: `<tab-bar [data]="list"
                      (onTabSelect)="select($event)"></tab-bar>
             <map [item]="country"></map>`,
  directives: [TabBar, StaticMap]
})
export class Widget {
  list: Country[] = countries;
  country: Country = new Country();

  select(c:Country) {
    this.country = c;
  }
}
```

1

2

3

<widget/> (completed)

Angular 2 Components

- Modules – This is used to break up the application into logical pieces of code. Each piece of code or module is designed to perform a single task.
- Component – This can be used to bring the modules together.
- Templates – This is used to define the views of an Angular JS application.
- Metadata – This can be used to add more data to an Angular JS class.
- Service – This is used to create components which can be shared across the entire application.

Angular 2 Environment Setup

- Install Node.js® and npm if they are not already on your machine.

Verify that you are running at least node 6.9.x and npm 3.x.x by running node -v and npm -v in a terminal/console window. Older versions produce errors, but newer versions are fine.

- Install AngularJS

```
$ npm install –g @angular/cli
```

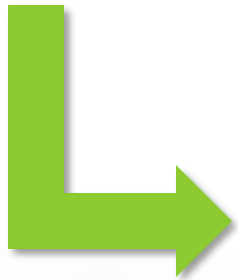
- Create a new project

```
$ ng new my-app
```

- Start the application

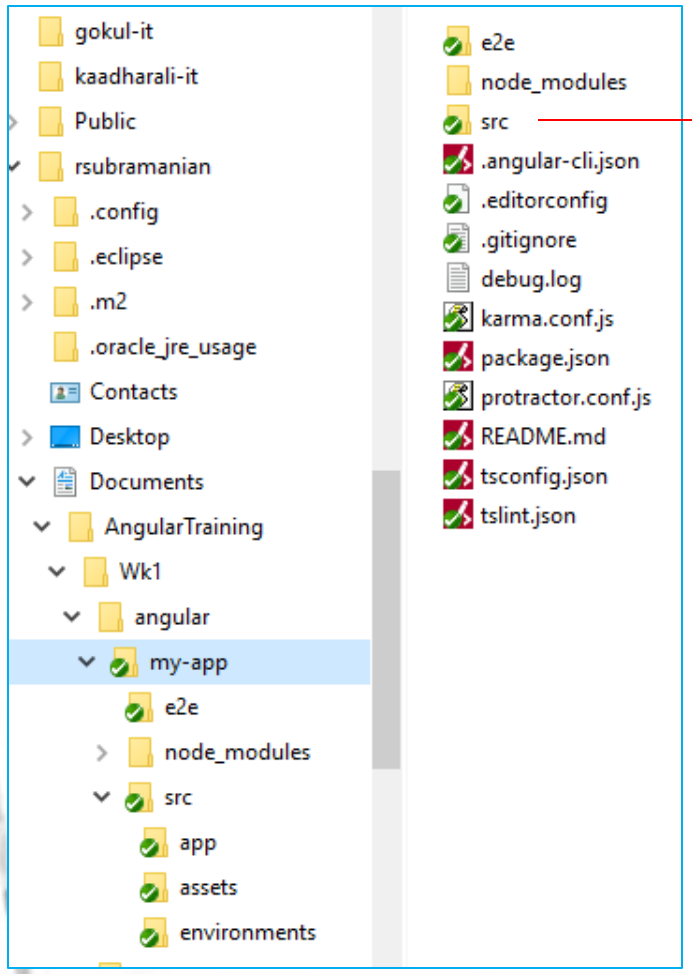
```
$ cd my-app
$ ng serve –open
```

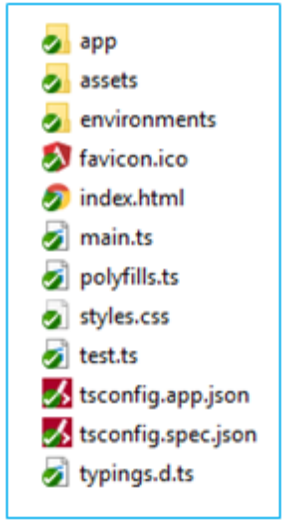
Output in the Default Browser:-



Welcome to app!!

Angular 2 Project Folder Setup





- Project Structure create by ng package
- Project is compatable with IDE such as NodePad++, Atom, Sublime etc

Points to Note:-

- Package.json → Has details about 3rd Party Packasges, and Custom Scripts
- tsconfig.json → TypeScript compiler configuration for your IDE
- tslint.json → Linting configuration for TSLint together with Codelyzer (when ng lint)
- karma.conf.js → Unit test configuration for the Karma test runner, used when running ng test
- E2e/ → which has e2e test cases.
- node_modules → place where all dependent packages are stored in a recursive fashion.
- environments/* → For maintaining env specific configurations.
- assets/* → A folder where you can put images and anything else to be copied wholesale when you build your application.
- index.html → main html file where all js/css files are linked (at runtime)
- main.ts → entry point for the application

Angular 2 Sample Code Base

app.module.ts

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
```

a

index.html

```
<!doctype html>
<html lang="en"><head>
<title>MyApp</title>
 <base href="/">
<meta name="viewport" content="width=device-width,
initial-scale=1"> <link rel="icon" type="image/x-icon"
href="favicon.ico">
</head>
<body>
<app-root></app-root>
</body>
</html>
```

b

app.component.ts

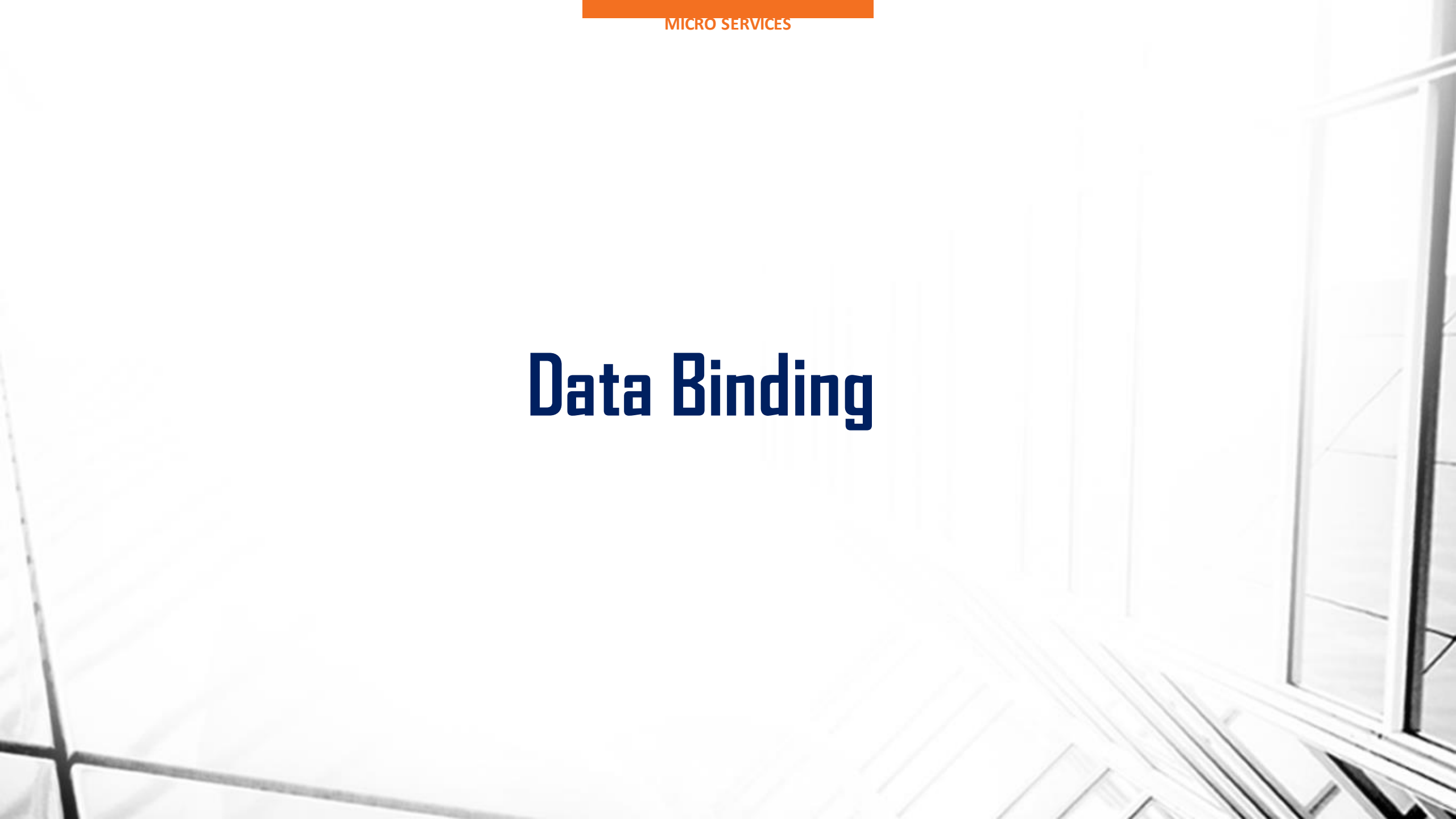
```
import { Component } from '@angular/core';

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

export class AppComponent {
    title = 'app';
}
```

c

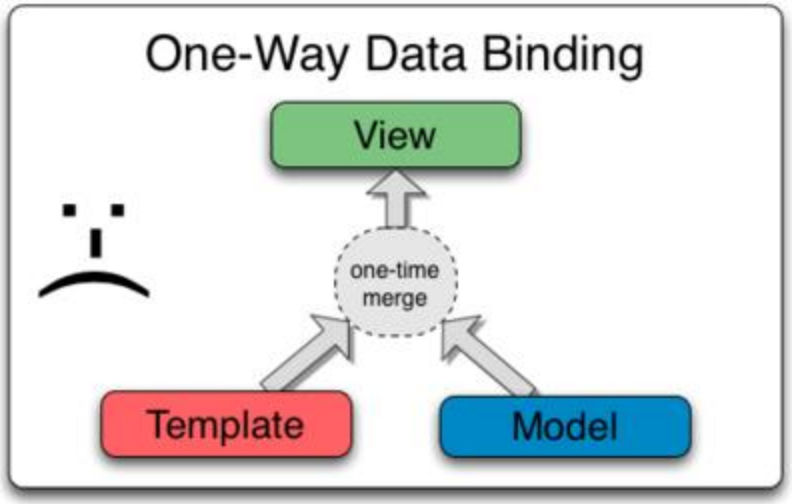
app.component.html

```
<h1>    Welcome to {{title}}!!  </h1>
<h2>Here are some links to help you start:
</h2>
```

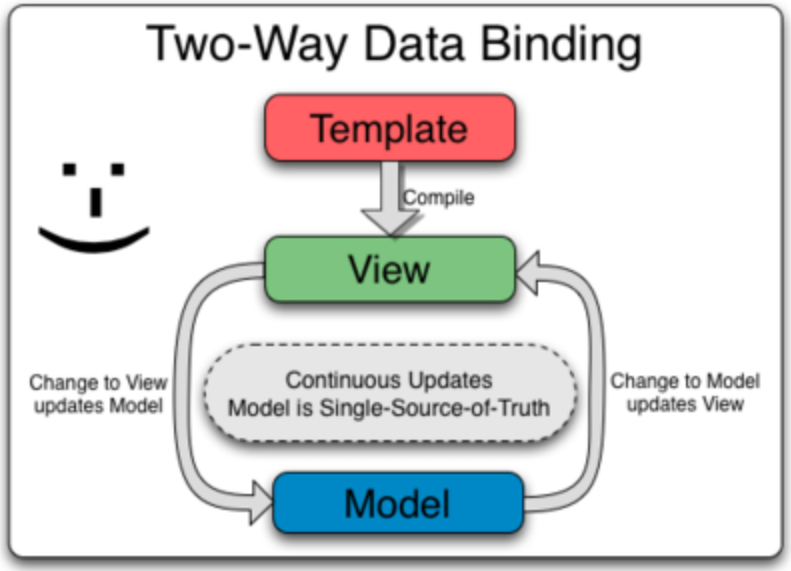
Data Binding



Data Binding / interpolation – One Way



Data Binding / interpolation – 2 Way Binding



Data Binding / interpolation

….the insertion of something of a different nature into something else.

Data Binding / interpolation

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

@Component({
  selector: 'app-root',
  template: `
    <h1>{{title}}</h1>
    <h2>My favorite hero is: {{myHero}}</h2>
    `
})
export class AppComponent {
  title = 'Tour of Heroes';
  myHero = 'Windstorm';
}
```

Data Binding / interpolation – via Constructor

```
export class AppCtorComponent {
  title: string;
  myHero: string;

  constructor() {
    this.title = 'Tour of Heroes';
    this.myHero = 'Windstorm';
  }
}
```

Data Binding / interpolation – 2 Way Binding

```
<input [(ngModel)]="username">

<p>Hello {{username}}!</p>
```

OR

```
<input [value]="username" (input)="username = $event.target.value">

<p>Hello {{username}}!</p>
```

- [value]=”username” - Binds the expression username to the input element’s value property
- (input)=”expression” - Is a declarative way of binding an expression to the input element’s input event (yes there’s such event)
- username = $event.target.value - The expression that gets executed when the input event is fired
- $event - Is an expression exposed in event bindings by Angular, which has the value of the event’s payload

Data Binding / interpolation – 2 Way Binding

```
<input [ngModel]="username" (ngModelChange)="username = $event">

<p>Hello {{username}}!</p>
```

- The property binding [ngModel] takes care of updating the underlying input DOM element.
- The event binding (ngModelChange) notifies the outside world when there was a change in the DOM.
- ngModelChange takes care of extracting target.value from the inner $event payload

Data Binding / interpolation – Custom 2 Way Binding

app/counter.component.ts

```
export class CounterComponent {
  @Input() count = 0;
  @Output() countChange = EventEmitter<number>();

  increment() {
    this.count++;
    this.countChange.emit(this.count);
  }
}
```

app/counter.component.html

```
<div>
  <p>
     Count: {{ count }} -
    <button (click)="increment()">Increment</button>
  </p>
</div>
```

Data Binding / interpolation – NGFor*

```
export class AppComponent {
  title = 'Tour of Heroes';
  heroes = ['Windstorm', 'Bombasto', 'Magneta', 'Tornado'];
  myHero = this.heroes[0];
}
```

```
template: `
  <h1>{{title}}</h1>
  <h2>My favorite hero is: {{myHero}}</h2>
  <p>Heroes:</p>
  <ul>
    <li *ngFor="let hero of heroes">
      {{ hero }}
    </li>
  </ul>
`
```

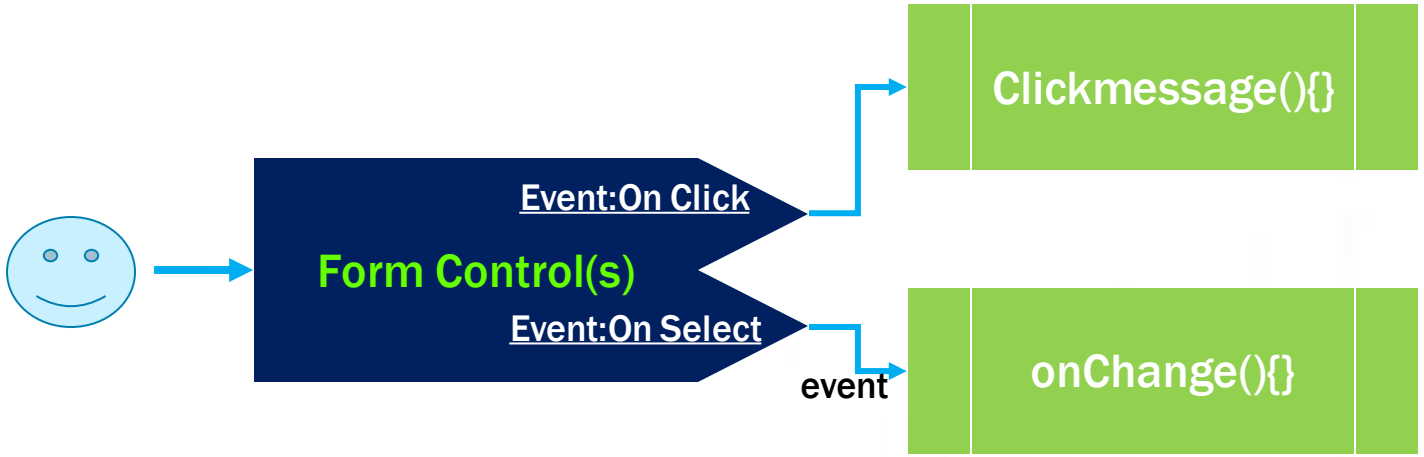
Data Binding / interpolation – NgIf*

```
<p *ngIf="heroes.length > 3">There are many heroes!</p>
```


LIVE DEMO

Event Handling

Event handling




```
<button (click)="onClickMe()">Click me!</button>

{{clickMessage}}
```

```
export class ClickMeComponent {
  clickMessage = '';

  onClickMe() {
    this.clickMessage = 'You are my hero!';
  }
}
```

Event handling

```
<input (keyup)="onKey($event)">

<p>{{values}}</p>
```

```
export class KeyUpComponent_v1 {
  values = '';

  onKey(event: any) { // without type info
    this.values += event.target.value + ' | ';
  }
}
```

The properties of an $event object vary depending on the type of DOM event. For example, a mouse event includes different information than an input box editing event.

Event handling – with Proper Data Type

```
<input (keyup)="onKey($event)">

<p>{{values}}</p>
```

```
export class KeyUpComponent_v1 {
  values = '';


  onKey(event: KeyboardEvent) { // with type info
    this.values += (<HTMLInputElement>event.target).value + ' | ';
  }
}
```

The $event is now a specific KeyboardEvent.

Event handling – with “Template Reference Variable”

```
@Component({
  selector: 'app-key-up2',
  template: `
    <input #box (keyup)="onKey(box.value)">
    <p>{{values}}</p>
  `
})
export class KeyUpComponent_v2 {
  values = '';
  onKey(value: string) {
    this.values += value + ' | ';
  }
}
```

The template reference variable named box, declared on the <input> element, refers to the <input> element itself. The code uses the box variable to get the input element's value and display it with interpolation between <p> tags.

Event handling – with “Template Reference Variable” (Key Event Filter)

```
@Component({
  selector: 'app-key-up3',
  template: `
    <input #box (keyup.enter)="onEnter(box.value)">
    <p>{{value}}</p>
  `
})
export class KeyUpComponent_v3 {
  value = '';
  onEnter(value: string) { this.value = value; }
}
```

Event handling – with “Template Reference Variable” (Multi Event Handling)

```
@Component({
  selector: 'app-key-up4',
  template: `
    <input #box
      (keyup.enter)="update(box.value)"
      (blur)="update(box.value)">
    <p>{{value}}</p>
  `
})
export class KeyUpComponent_v4 {
  value = '';
  update(value: string) { this.value = value; }
}
```

Event handling – with “Template Reference Variable” - Observations

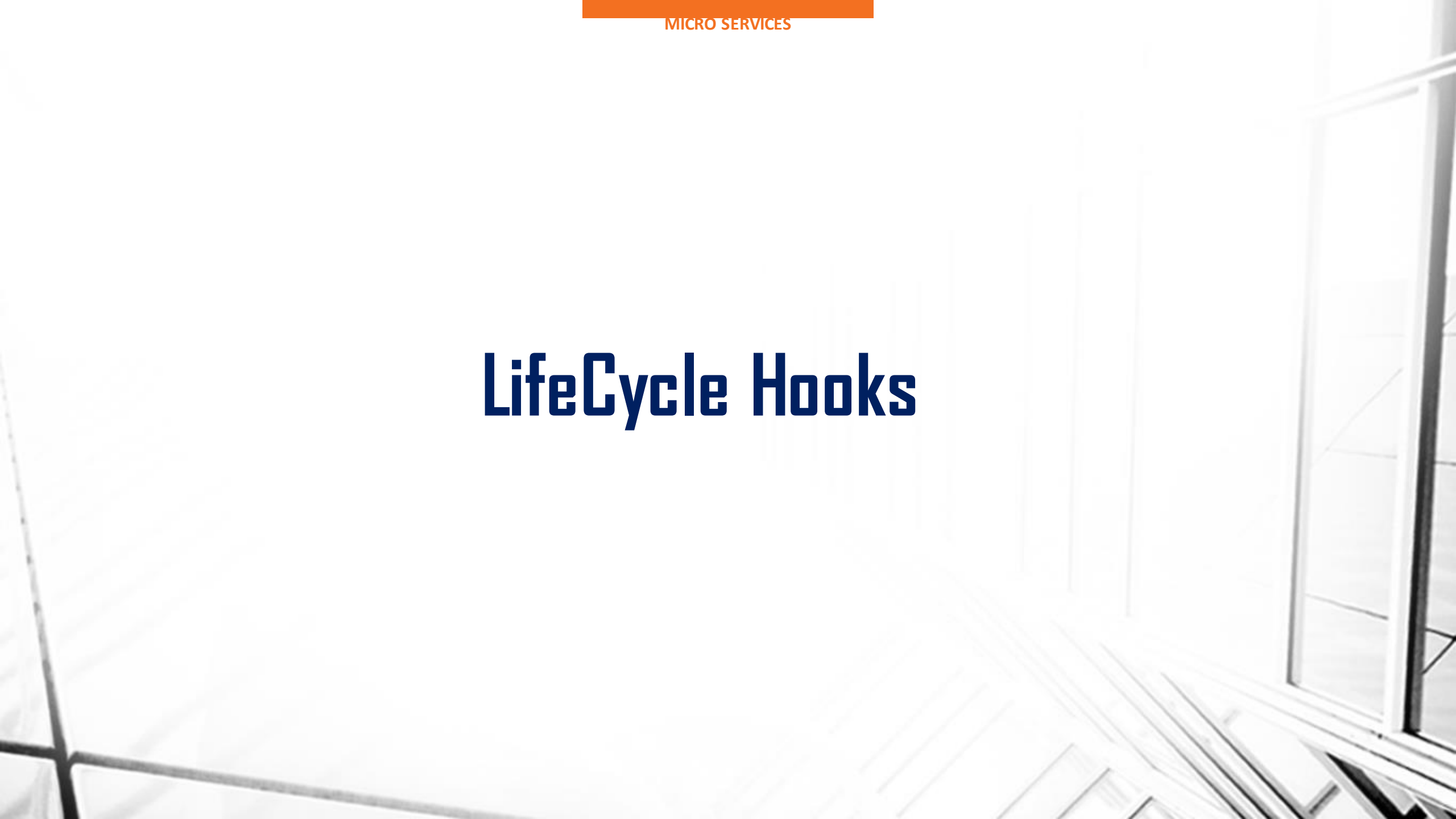
Use template variables to refer to elements

Pass values, not elements

Keep template statements simple


LIVE DEMO

LifeCycle Hooks



Life Cycle Hooks

- Angular creates it, renders it, creates and renders its children, checks it when its data-bound properties change, and destroys it before removing it from the DOM.

- Angular offers lifecycle hooks that provide visibility into these key life moments and the ability to act when they occur.

Life Cycle Hooks

```
export class PeekABoo implements  OnInit {
  constructor(private logger:  LoggerService) { }

  // implement  OnInit's  `ngOnInit`  method
  ngOnInit()  { this.logIt(`OnInit`);  }

  logIt(msg:  string) {
    this.logger.log(`#${nextId++}  ${msg}`);
  }
}
```

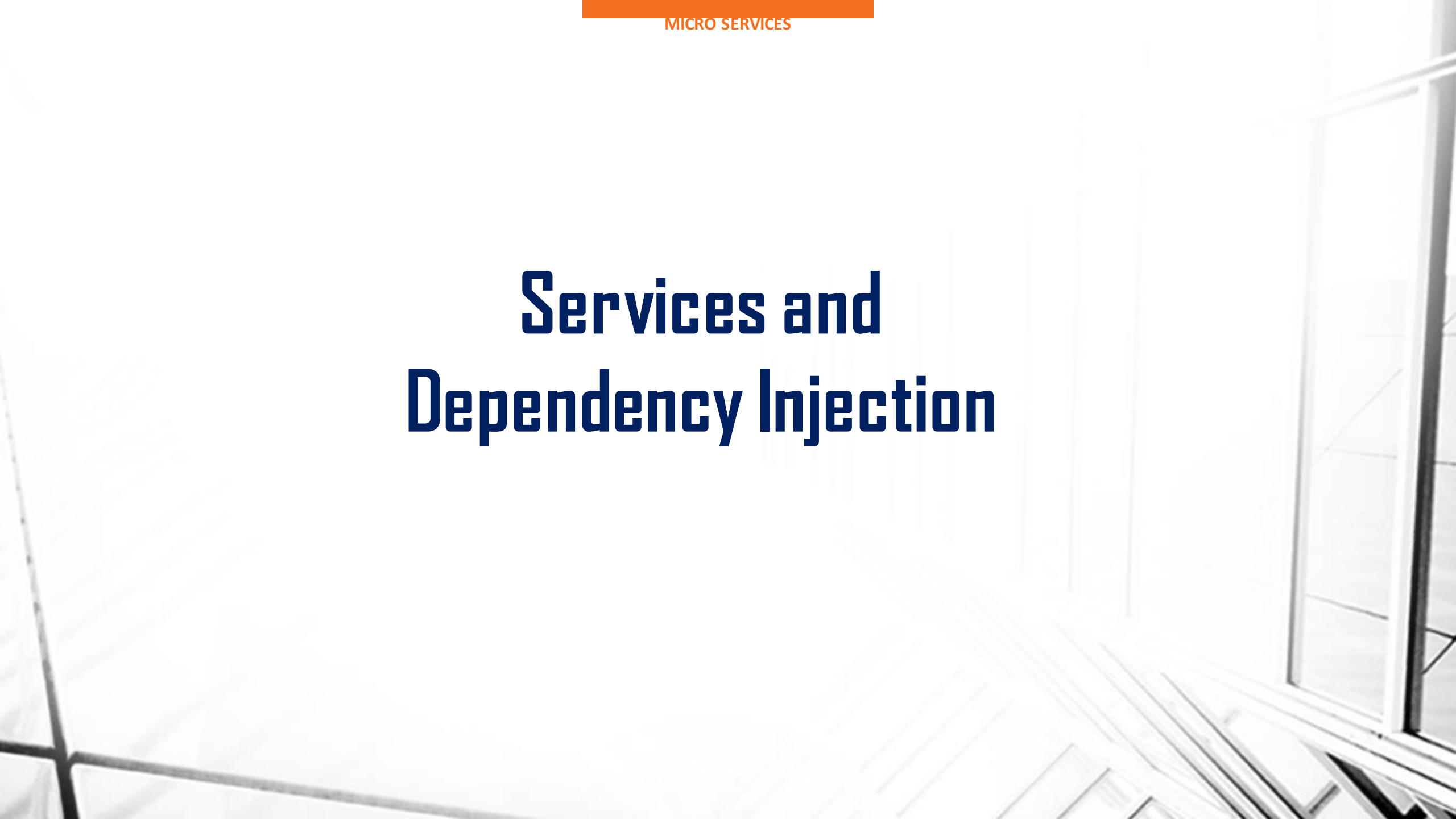
- No directive or component will implement all of the lifecycle hooks.
- Angular only calls a directive/component hook method *if it is defined*.

Life Cycle Hooks

ngOnChanges()	Respond when Angular (re)sets data-bound input properties.
ngOnInit()	Initialize the directive/component after Angular first displays the data-bound properties and sets the directive/component's input properties.
ngDoCheck()	Detect and act upon changes that Angular can't or won't detect on its own. Called during every change detection run, immediately after ngOnChanges() and ngOnInit().
ngOnDestroy()	Cleanup just before Angular destroys the directive/component. Unsubscribe Observables and detach event handlers to avoid memory leaks. Called *just before* Angular destroys the directive/component.
Other Hooks :- • ngAfterContentInit() • ngAfterContentChecked() • ngAfterViewInit() • ngAfterViewChecked()	


LIVE DEMO

Services and Dependency Injection

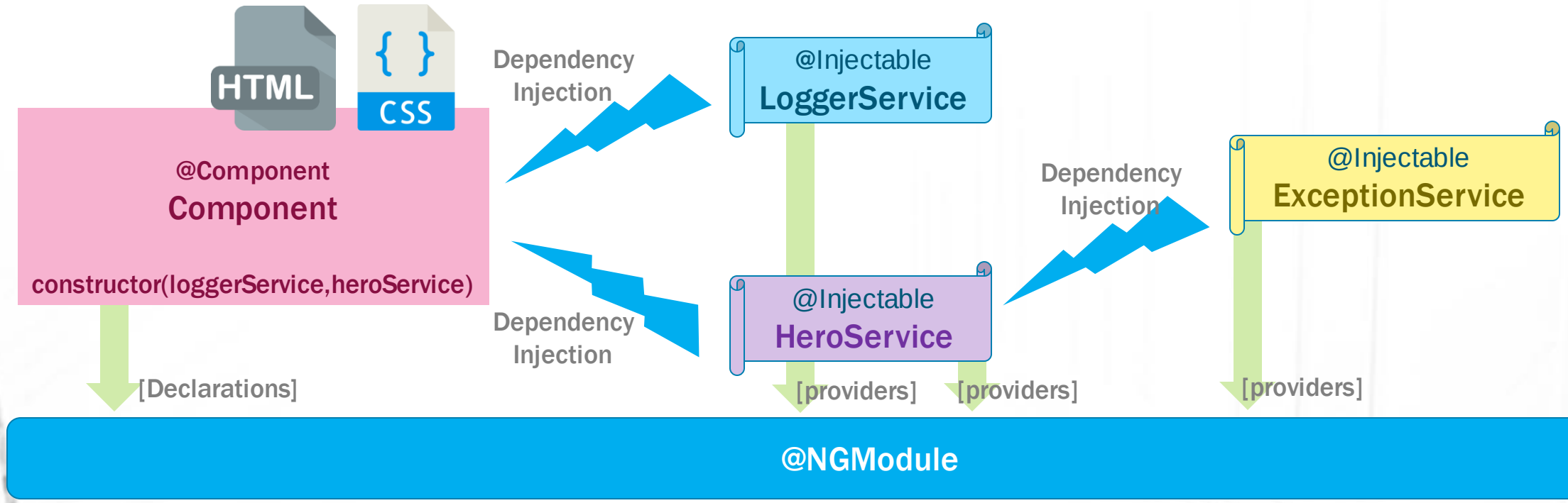


Services and DI

Service is a broad category encompassing any value, function, or feature that an app needs.

DI framework provides declared dependencies to a class when that class is instantiated.

Services and DI



Services and DI

- A service is typically a class with a narrow, well-defined purpose. For example LoggerService, HttpClientService, DataTransformationService
- Angular creates a single, shared instance of LoggerService and injects it into any class that asks for it
- Makes the component layer more leaner and improved manageability.
- Supports Contract based Implementation using Interfaces.

Services and DI – Service Definition

src/app/logger.service.ts

```
@Injectable()
export class Logger {
  log(msg:  any)   { console.log(msg);  }
  error(msg:  any) { console.error(msg); }
  warn(msg:  any)  { console.warn(msg);  }
}
```

Services and DI – Service Injection –Option 1

src/app/hero-list.component.ts

```
@Component({
  selector:    'app-hero-list',
  templateUrl:  './hero-list.component.html',
  providers:   [ LoggerService ]
})
```

Services and DI – Service Injection –Option 2

src/app/hero-list.component.ts

```
constructor(private service: LoggerService) { }
```

Services and DI – Service Injection – to a given Module

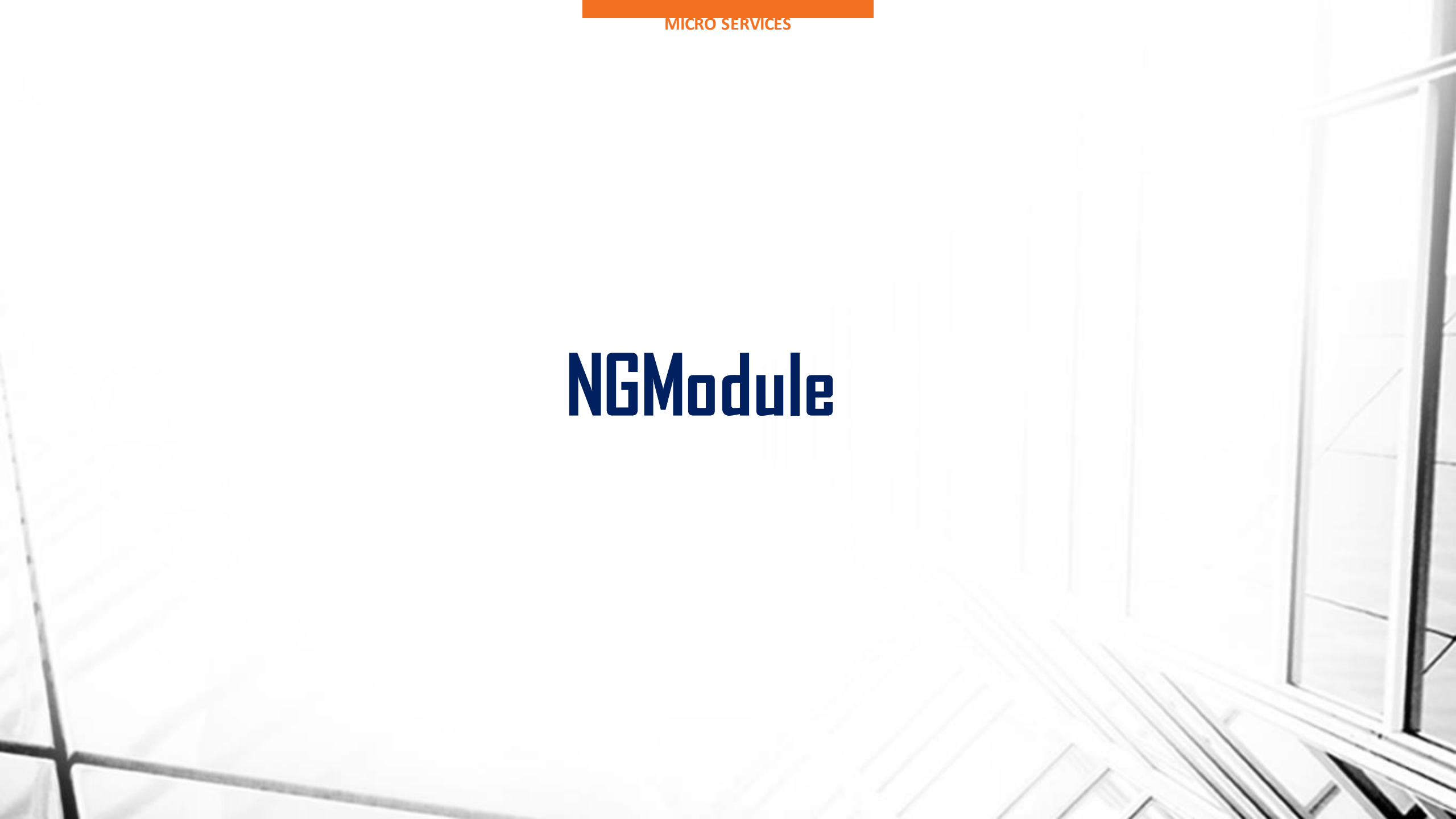
src/app/app-module.ts

```
@NgModule({
  providers: [
        BackendService,
        LoggerService
 ],
 ...
})}
```

Services and DI – Service Injection – Injectable Decorator

import { Injectable } from '@angular/core'; @Injectable({ providedIn: 'root', }) export class UserService {}	'root' → injectable will be registered as a singleton in the application, & not required to add it to the providers of the root module.
import { Injectable } from '@angular/core'; import { UserModule } from './user.module'; @Injectable({ providedIn: UserModule, }) export class UserService {}	The Service is available only the module "UserModule"
import { NgModule } from '@angular/core'; import { UserService } from './user.service'; @NgModule({ providers: [UserService], }) export class UserModule {}	The Service is available only the module "UserModule"

NGModule



NgModules

@NgModule

NgModules are containers for a cohesive block of code dedicated to an application domain, a workflow, or a closely related set of capabilities

NgModules –Meta Data →Sample Code Snippet

```
@NgModule({
  declarations: [
    AppComponent,
    ItemDirective
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule
  ],
  export:[],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

NgModules –Meta Data → Declarations

The components, directives, and pipes that belong to this NgModule.

NgModules –Meta Data →exports

The subset of declarations that should be visible and usable in the *component templates* of other NgModules.

NgModules –Meta Data →imports

Other modules whose exported classes are needed by component templates declared in *this* NgModule.

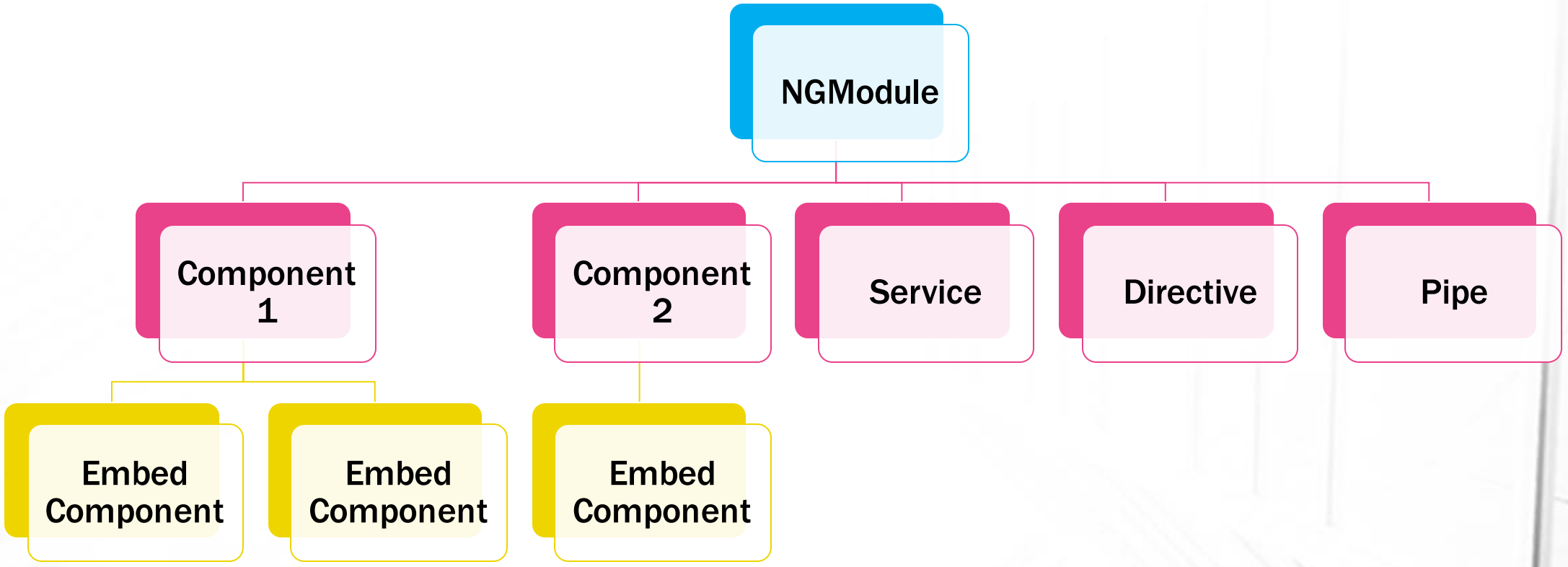
NgModules –Meta Data → Providers

Creators of services that this NgModule contributes to the global collection of services; they become accessible in all parts of the app. (You can also specify providers at the component level, which is often preferred.)

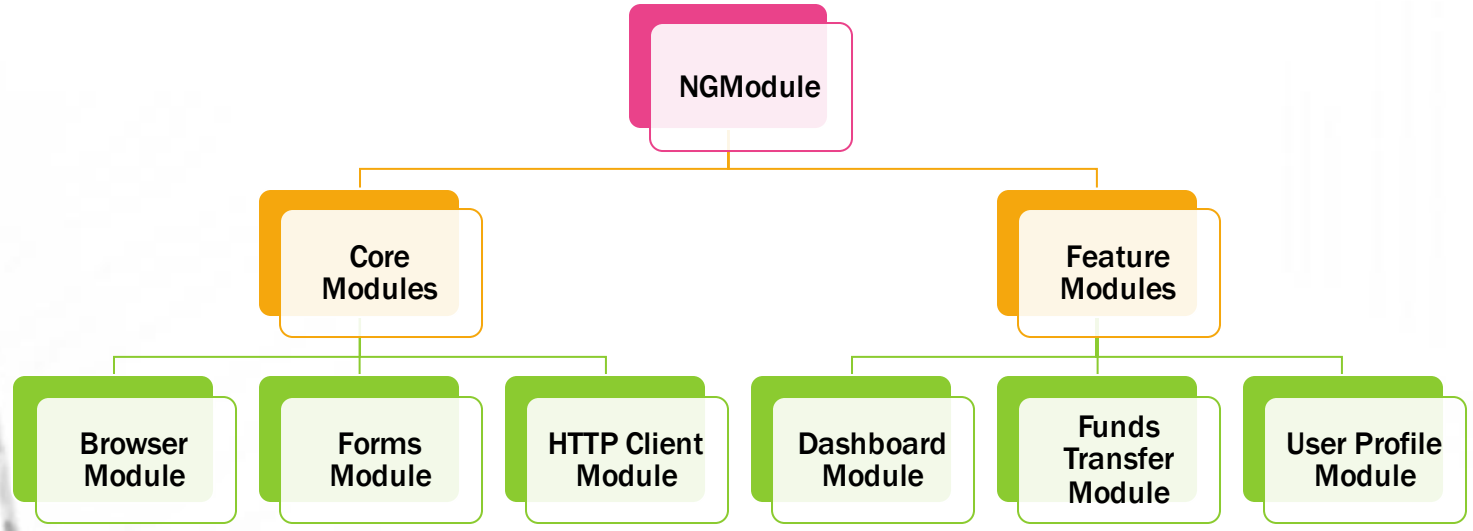
NgModules –Meta Data → bootstrap

The main application view, called the root component, which hosts all other app views. Only the root NgModule should set the bootstrap property.

NgModules – Hierarchy



NgModules – Importing other Modules (Feature Modules)




```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule,
    CustomerDashboardModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

NgModules – Importing other Modules (Feature Modules)

```
$> ng generate module CustomerDashboard
```

Template

Template

HTML is the language of the Angular template

manages what the user sees and can do

Template

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'My First Angular App!';
}
```

Template Instances

Interpolation in line with HTML elements

```
<h3> {{title}}
<img src="{{heroImageUrl}}" style="height:30px">
</h3>
```

Evaluation

```
<p>The sum of 1 + 1 is {{1 + 1}}</p>
```

Evaluation + Invoke Methods

```
<p>The sum of 1 + 1 is not {{1 + 1 + getVal()}}</p>
```

Template Instances

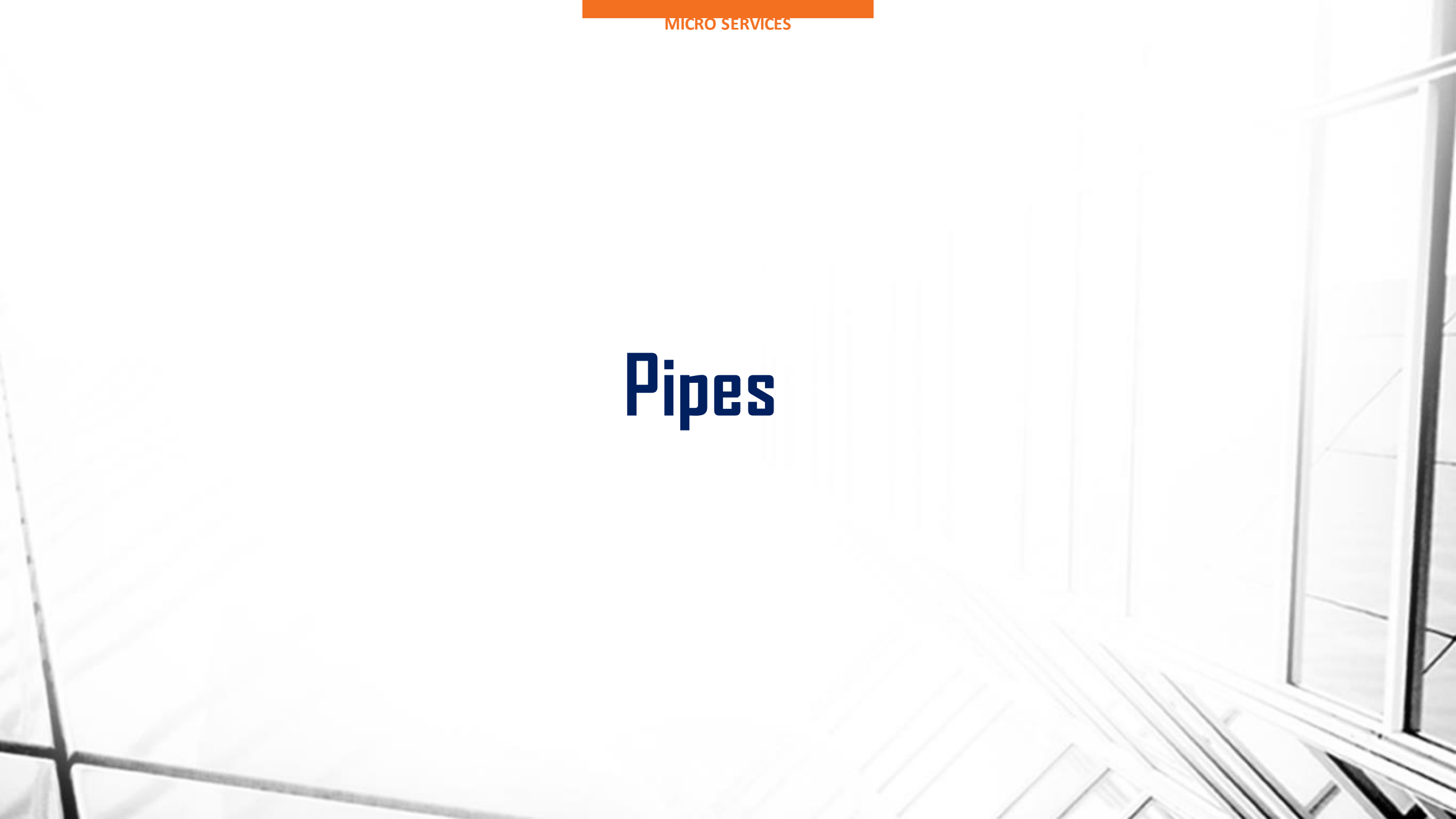
Interpolation with Iteration

```
<div *ngFor="let hero of heroes">{{hero.name}}</div>
<input #heroInput> {{heroInput.value}}
```

Event Handing with Iteration

```
<button (click)="onSave($event)">Save</button>
<button *ngFor="let hero of heroes" (click)="deleteHero(hero)">{{hero.name}}</button>
<form #heroForm (ngSubmit)="onSubmit(heroForm)"> ... </form>
```

Pipes



Pipes

a way to write display-value transformations that you can declare in your HTML.

Places of Usage:-

Data Transformation
Data Filtering
In line formatting
Custom Localization

Pipes – Built In

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

@Component({
  selector: 'app-hero-birthday',
  template: `<p>The hero's birthday is {{ birthday | date }}</p>`
})
export class HeroBirthdayComponent {
  birthday = new Date(1988, 3, 15); // April 15, 1988
}
```

Pipes – Built In with Parametrization

```
<p>The hero's birthday is {{ birthday | date:"MM/dd/yy" }} </p>
```

Pipes – Built In with Parametrization(w) expression

```
template: `
  <p>The hero's birthday is {{ birthday | date:format }}</p>
  <button (click)="toggleFormat()">Toggle Format</button>
```

```
export class HeroBirthday2Component {
  birthday = new Date(1988, 3, 15); // April 15, 1988
  toggle = true; // start with true == shortDate

  get format()   { return this.toggle ? 'shortDate' : 'fullDate'; }
  toggleFormat() { this.toggle = !this.toggle; }
}
```

Pipes – Chaining

```
The chained hero's birthday is
{{ birthday | date | uppercase}}
```

```
The chained hero's birthday is
{{  birthday | date:'fullDate' | uppercase}}
```

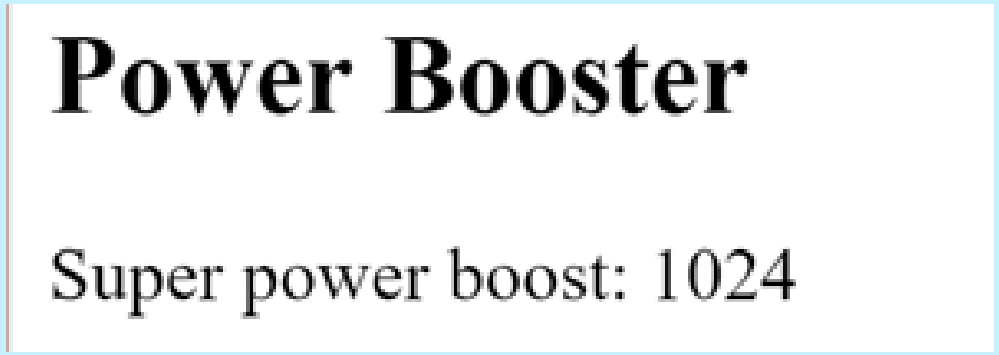
Pipes – Custom Developed Pipes

Pipe Definition:-

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({name: 'exponentialStrength'})
export class ExponentialStrengthPipe implements PipeTransform {
  transform(value: number, exponent: string): number {
    let exp = parseFloat(exponent);
    return Math.pow(value, isNaN(exp) ? 1 : exp);
  }
}
```

Pipe Usage:-

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

@Component({
  selector: 'app-power-booster',
  template: `
    <h2>Power Booster</h2>
    <p>Super power boost: {{2 | exponentialStrength: 10}}</p>`
})
export class PowerBoosterComponent { }
```



Pipes – Pure vs Im-Pure Pipes

Pure	Impure
Default Setting	
Executes when change to a primitive input value (String, Number, Boolean, Symbol) or a changed object reference	Executes an *impure pipe* during every component change detection cycle. An impure pipe is called often, as often as every keystroke or mouse-move.
Faster & generally preferred	

```
@Pipe({
  name: 'flyingHeroesImpure',
  pure: false
})
export class FlyingHeroesImpurePipe extends FlyingHeroesPipe {}
```

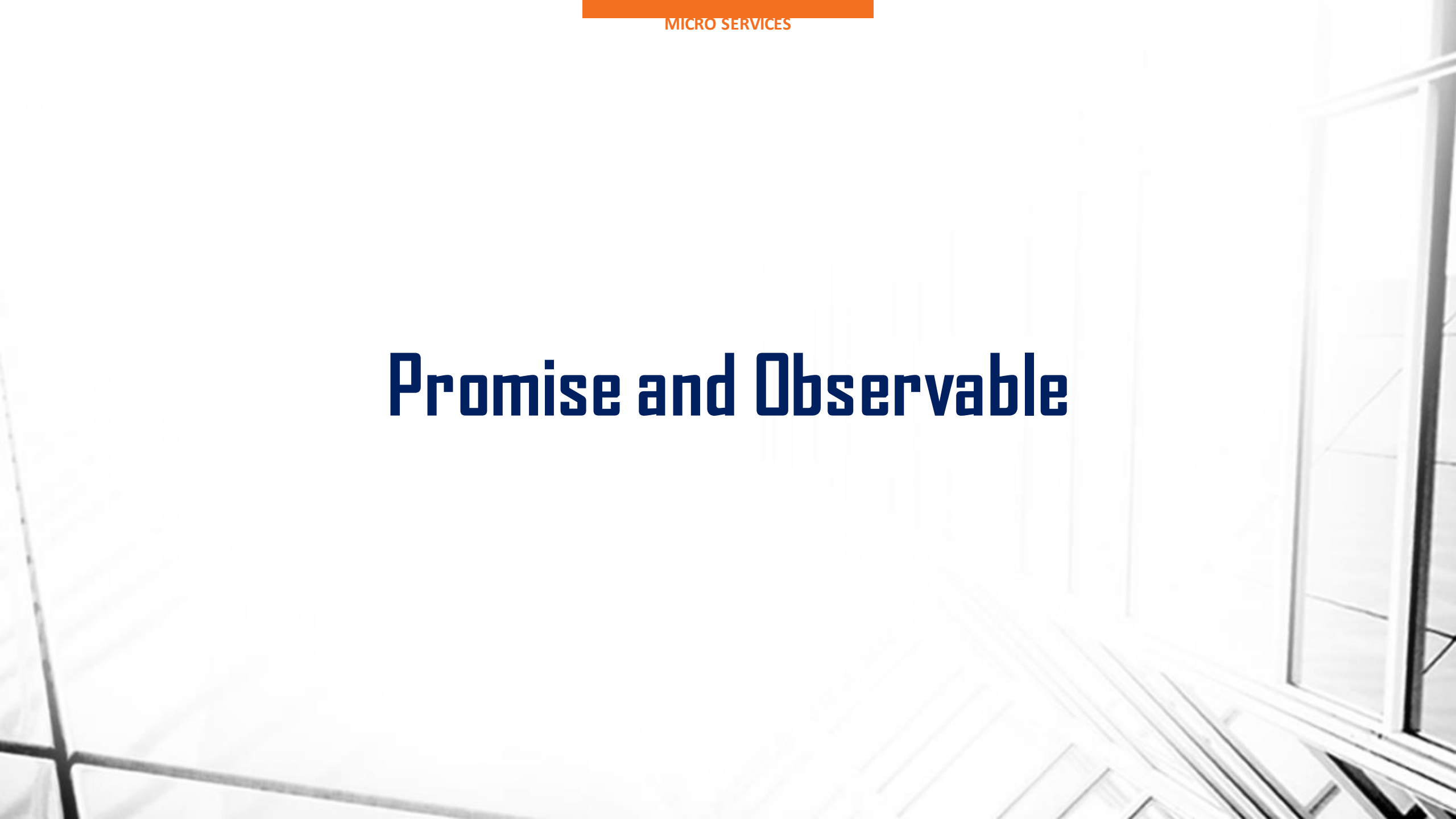
Async Pipes – Im-Pure Pipes

```
@Component({
  selector: 'app-hero-message',
  template: `
    <h2>Async Hero Message and AsyncPipe</h2>
    <p>Message: {{ message$ | async }}</p>
    <button (click)="resend()">Resend</button>`,
})
export class HeroAsyncMessageComponent {
  message$: Observable<string>;

  private messages = [
    'You are my hero!',
    'You are the best hero!',
    'Will you be my hero?'
  ];
  constructor() { this.resend(); }
  resend() {
    this.message$ = interval(500).pipe(
      map(i => this.messages[i]),
      take(this.messages.length)
    );
  }
}
```

The AsyncPipe accepts a Promise or Observable as input and subscribes to the input automatically, eventually returning the emitted values

- The AsyncPipe is also stateful.
- The pipe maintains a subscription to the input Observable and keeps delivering values from that Observable as they arrive

Promise and Observable



Promise & Observable

Promise, in Javascript, is a concept which allows the callee function to send back a promise (sort of assurance) to the caller function that it would, for sure, send back a resolution, be it a success or a failure at a little later point of time.

The caller believes the callee if a promise is sent back, and, proceeds ahead with the program execution.

Why Promise..in a nutshell

Allows asynchronous operations to happen; Caller can proceed ahead with program execution if callee function returned a promise object.

Callbacks

```
function doAsyncTask(cb) {
  setTimeout(() => {
    console.log("Async Task Calling Callback");
    cb();
  }, 1000);
}

doAsyncTask(() => console.log("Callback  Called"));
```

The doAsyncTask function when called kicks of an asynchronous task and returns immediately.

Creating a Promise

```
var promise = new Promise((resolve, reject) => {
})
```

We pass to Promise an inner function that takes two arguments (resolve, reject).

Since we are defining the function we can call these arguments whatever we want but the convention is to call them resolve and reject.

resolve and reject are in fact functions themselves.

Inside this inner function we perform our asynchronous processing and then when we are ready we call resolve(), like so:

Creating a Promise

```
var promise = new Promise((resolve, reject) => {
})
```

```
function doAsyncTask() {
  var promise = new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log("Async Work Complete");
      if (error) {
        reject();
      } else {
        resolve();
      }
    }, 1000);
  });
  return promise;
}
```

Creating a Promise Notifications

We can get notified when a promise resolves by attaching a success handler to its then function, like so

```
doAsyncTask().then(()  => console.log("Task  Complete!"));
```

Creating a Promise Notifications

then can take two arguments, the second argument is a error handler that gets called if the promise is rejected, like so:

```
doAsyncTask().then(
  () => console.log("Task Complete!"),
  () => console.log("Task Errored!"),
);
```

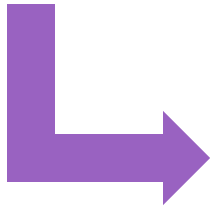
Why Promise

Allows asynchronous operations to happen; Caller can proceed ahead with program execution if callee function returned a promise object.

Why Promise. In Action

```
<div>
  <h2>Star Wars Character Search</h2>
  <input #term type="text" (keyup)="search(term.value)">
  <hr>
  <div *ngIf="results.length > 0">
    <li *ngFor="let result of results">{{ result.name }} is a character with a height of {{ result.height }}.</li>
  </div>
</div>
```

```
constructor(private http: HttpClient) { }

  private search(term) {
    console.log(term);
    this.http.get(`https://swapi.co/api/people/?search=${term}`).toPromise()
    .then((data: any) => {
      /* tslint:disable:no-console */
      console.time('request-length');
      console.log(data);
      console.timeEnd('request-length');
      this.results = data.results;
    })
    .catch(this.handleError);
  }
```

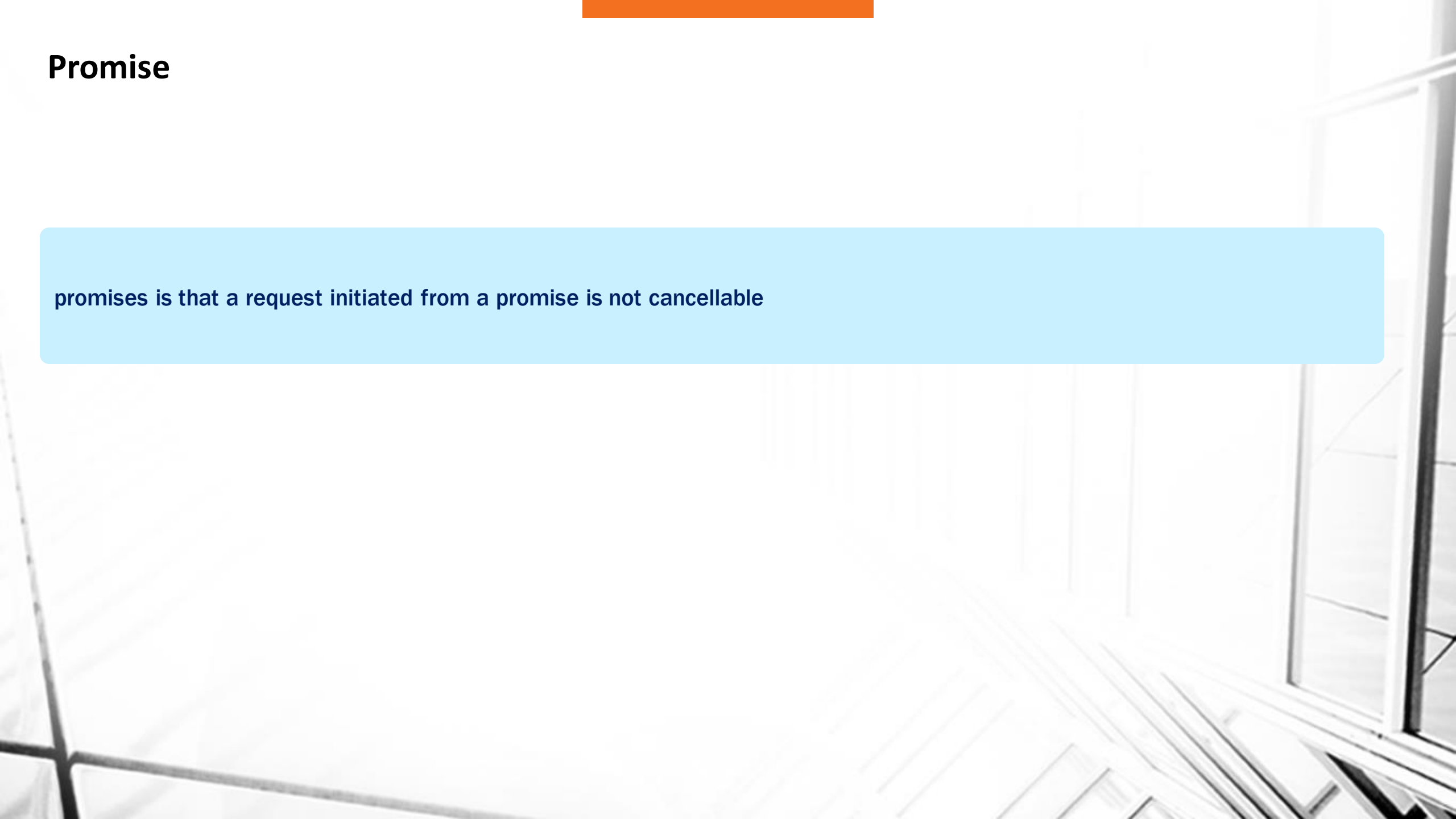


The `console` statements are left in there deliberately. This is how the browser console would look like:

```
[Log] l
[Log] lu
[Log] luk
[Log] luke
[Log] {count: 2, next: null, previous: null, results: Array}
[Debug] request-length: 0.590ms
[Log] {count: 37, next: "https://swapi.co/api/people/?page=2&search=l", previous: null, re
[Debug] request-length: 0.281ms
[Log] {count: 1, next: null, previous: null, results: Array}
[Debug] request-length: 0.377ms
[Log] {count: 1, next: null, previous: null, results: Array}
[Debug] request-length: 0.263ms
```

Promise

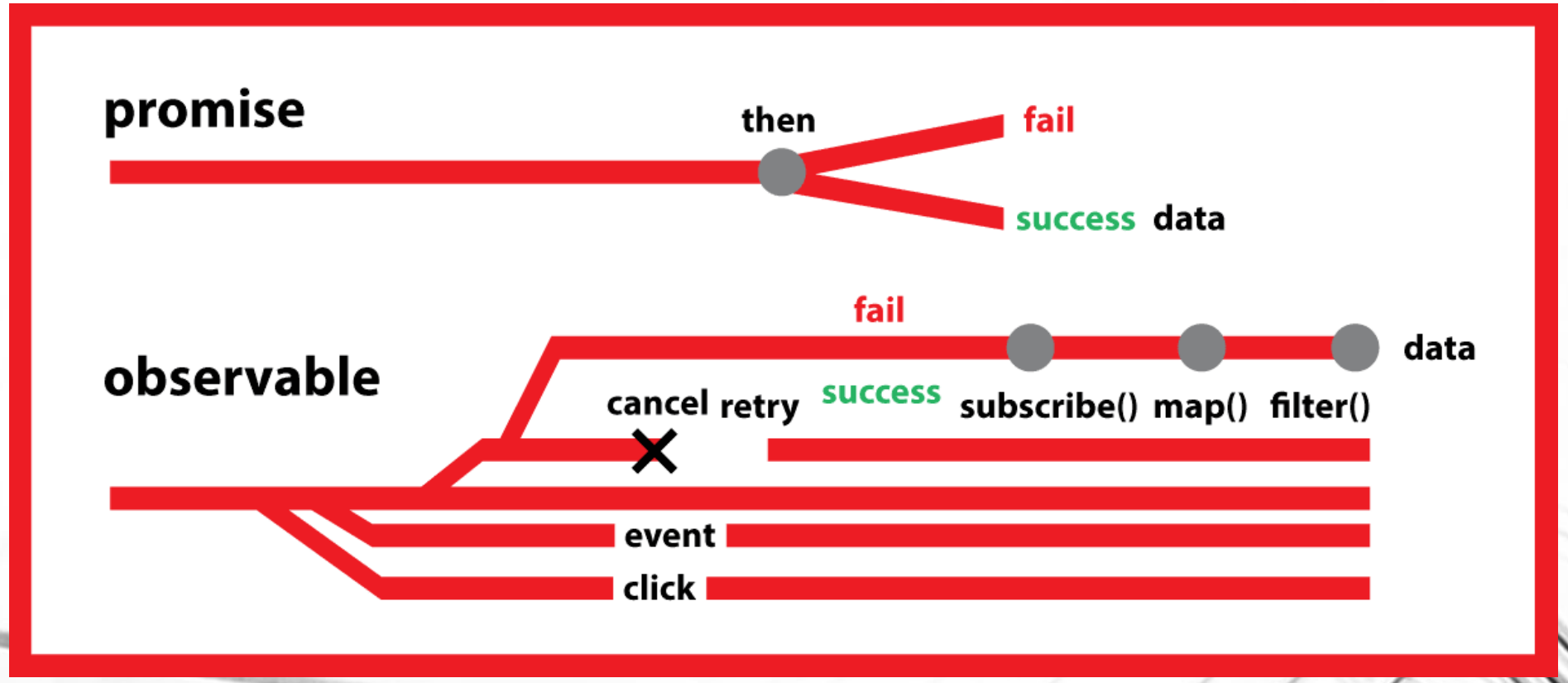
promises is that a request initiated from a promise is not cancellable



Observables

An observable is essentially a stream (a stream of events, or data) and compared to a Promise,

an Observable can be cancelled. It out of the box supports operators such as map() and filter()



Observables

```
ngOnInit() {
  this.term.valueChanges
    .debounceTime(400)
    .distinctUntilChanged()
    .subscribe(searchTerm  => {
      this.http.get(`https://swapi.co/api/people/?search=${searchTerm}`).subscribe((data:  any) => {
        /* tslint:disable:no-console  */
        console.time('request-length');
        console.log(data);
        console.timeEnd('request-length');
        this.results = data.results;
      });
    });
}
```

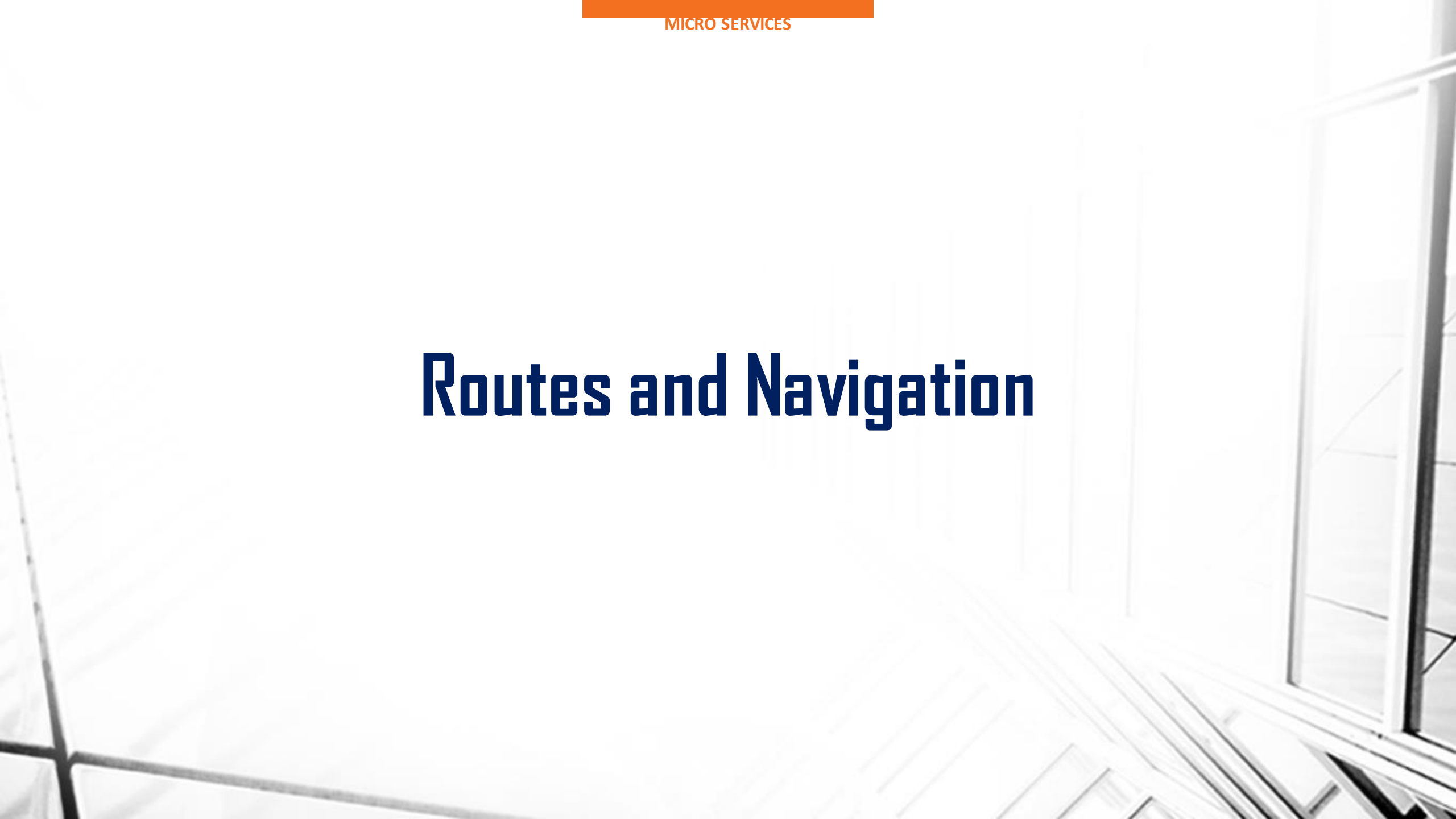
Observables

Map	**Http.get('/someUrl')** **.map(rese => res.json().data)** **.map(data => ({data: data.value*10})** **.subscribe(result => {this.view = result;});** **The first chain uses one http call, after multiple the response to 10 and assign the result to a view variable that is used to render**
SwitchMap	Http.get('/someUrl') .map(rese => res.json().data) .switchMap(data => Http.get(`/otherUrl/${data.value}`)) .subscribe(result => {this.view = result;});
ForkJoin	Observable.forkJoin(Http.get('/someUrl').map(res => res.json().data), Http.get('/otherUrl').map(res => res.json().data), (data1, data2) => ({data1, data2})).subscribe(data => {this.view = `${data.data1} - ${data.data2}`;});
Empty	Http.get('/someUrl').map(res => res.json().data) .switchMap(result => result.value >0 ? Observable.of(result) : Observable.empty()).switchMap(() => Http.get('/otherUrl').map(res => res.json().data)};
Reject	Observable.throw('Error')

Observables

Catch	Observable.throw('Error') .catch(err => { *console.log(err);* *return* Observable.empty(); });
Finally	Http.get('/someUrl').map(res => res.json().data) .catch(err = > { console.log(err);}) .finally(()=> stopLoader());

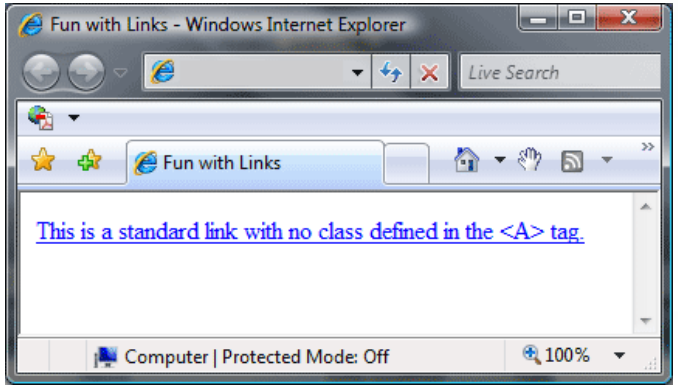
Routes and Navigation



Routes and Navigation

Angular Router enables navigation from one view to the next as users perform application tasks

URL in the address bar	Click links on the page	Click the browser's back and forward buttons



Routes and Navigation

```
const appRoutes: Routes = [
  { path: 'crisis-center', component: CrisisListComponent  },
  { path: 'hero/:id',      component: HeroDetailComponent  },
  {
    path: 'heroes',
    component: HeroListComponent,
    data: { title: 'Heroes List' }
  },
  { path: '',
    redirectTo: '/heroes',
    pathMatch: 'full'
  },
  { path: '**', component: PageNotFoundComponent  }
];
```

Order of route mapping

Pass the appRoutes to RouterModule.forRoot method in the module imports to configure the router.

```
@NgModule({
  imports: [
    RouterModule.forRoot(
      appRoutes,
      { enableTracing: true } // <-- debugging purposes only
    )
    // other imports here
  ],
  ...
})
```

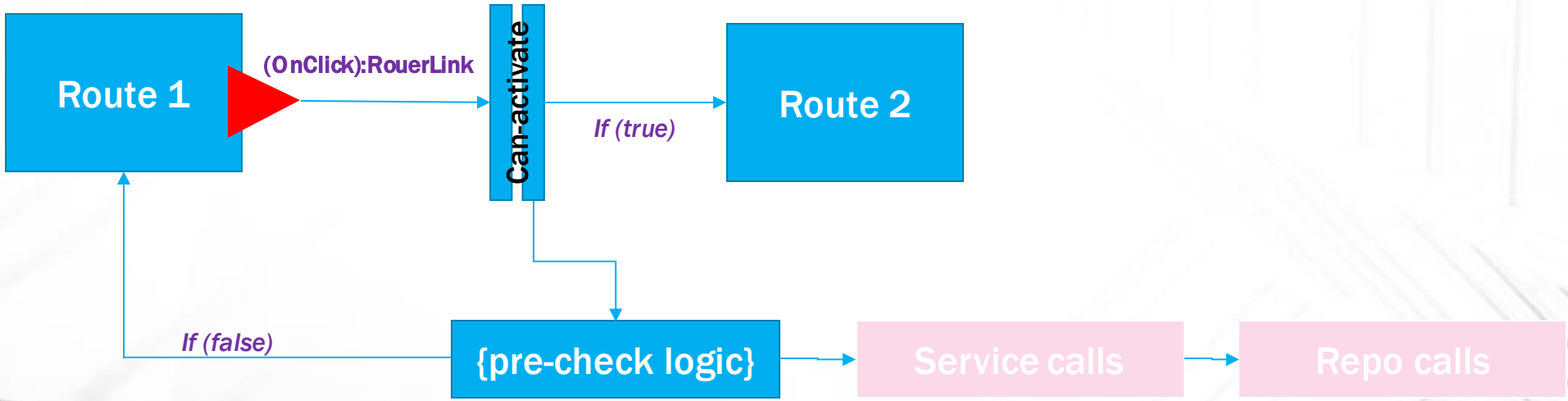
Routes – Router Link

```
<h1>Angular Router</h1>
<nav>
  <a routerLink="/crisis-center" routerLinkActive="active">Crisis Center</a>
  <a routerLink="/heroes" routerLinkActive="active">Heroes</a>
</nav>
<router-outlet></router-outlet>
```

The RouterOutlet is a directive from where the router should display the components for that outlet.

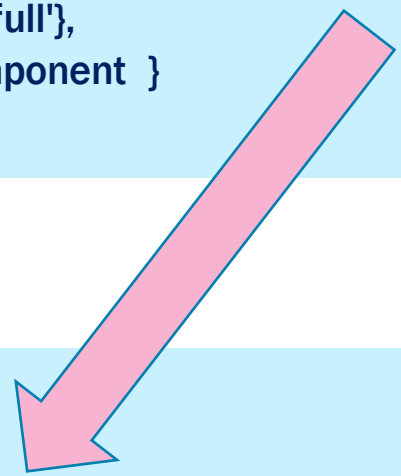
Routes – Route Guard(s)

- Angular's route guards are interfaces which can tell the router whether or not it should allow navigation to a requested route.
- Based on decision the controls goes to next route.
- This is helpful in places where following aspects are required.
  - Authorization
  - Pre Conditions Check
  - Route level logging/Auditing
  - Any further horizontal cross cutting concerns



Routes – Router Pre Verification – Can-Activate

```
const appRoutes: Routes = [
  { path: 'crisis-center', component: CrisisListComponent },
  { path: 'hero/:id',      component: HeroDetailComponent, canActivate: [AuthGuard] },
  { path: '', redirectTo: '/heroes', pathMatch: 'full'},
  { path: '**', component: PageNotFoundComponent }
];
```



```
@Injectable()
export class AuthGuard implements CanActivate {

  canActivate(next: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {
    if (this._authService.isAuthenticated()){
        return true;
    }

    // navigate to login page
    this._router.navigate(['/login']);
    // you can save redirect url so after authing we can move them back to the page they requested
    return false;
  }

}
```

Routes – Other Routes

Route Name	Purpose
CanActivate	• True→ Navigation will continue, False → Navigation will be cancelled. Or to any returned urltree object
CanActivateChild	• Same as CanActivate, but applies to Child route
CanDeactivate	• To check for user confirmation before proceeding or not with the stated route.
Resolve	• To have an intermediate code to be executed before loading the stated component
CanLoad	• To mediate navigation to a feature module loaded asynchronously

Routes – Navigating by Predefined URL(s)

```
<nav class="navbar navbar-light bg-faded">
 <a class="navbar-brand" href="/#/">iTunes Search App</a>
 <ul class="nav navbar-nav">
  <li class="nav-item active">
   <a class="nav-link" href="/#/">Home</a> (1)
  </li>
  <li class="nav-item">
   <a class="nav-link" href="/#/search">Search</a>
  </li>
 </ul>
</nav>
```

Routes – Navigating by Router API

```
class HeaderComponent  {
  constructor(private router: Router) {}

  goHome() {
    this.router.navigate(['']);
  }

  goSearch() {
    this.router.navigate(['search']);
  }
}
```

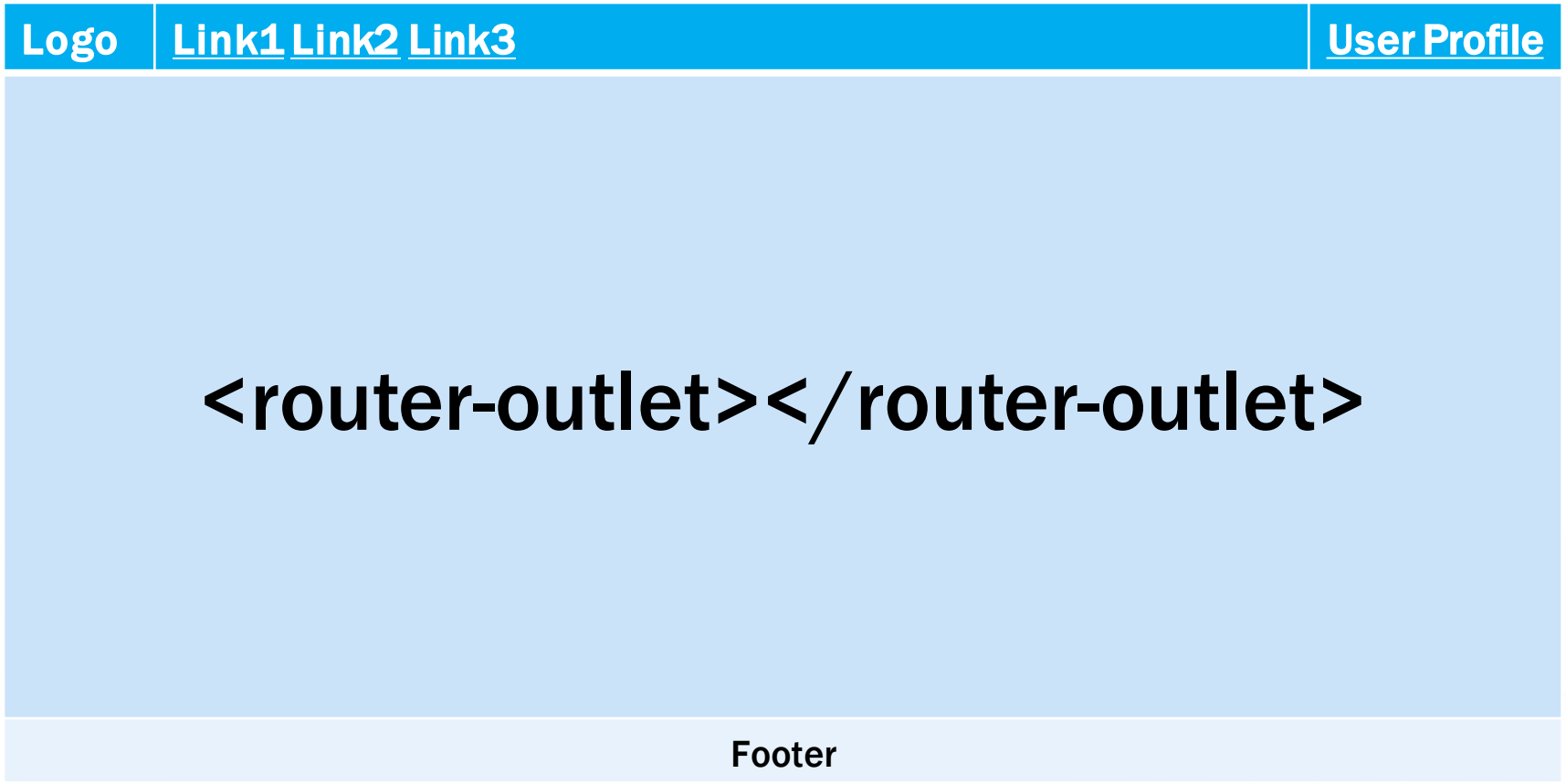
Routes – Navigating by Router API with Data

```
class HeaderComponent  {
  constructor(private router: Router) {}

  goSearch() {
    let part = "foo";
    this.router.navigate(['search',  part, 'moo']);
  }
}
```

Routes – Router Outlet

Type	Router Outlet Syntax
Default	<router-outlet></router-outlet>
Named Router Outlet	<router-outlet name="outlet1"></router-outlet>

Routes – Router Outlet – Default (Layout)



Routes – Router Outlet – Default (Layout)

Logo	Link1 Link2 Link3	User Profile
<router-outlet name='sidebar'></router-outlet>	<router-outlet name='contentarea'></router-outlet>	<router-outlet name='floatingmenu'></router-outlet>
Footer		


THANK YOU!