Angular
For beginners

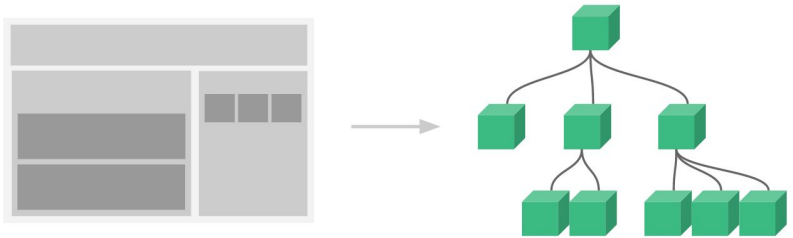
Angular

Angular is a platform and framework for building client applications in HTML and TypeScript.

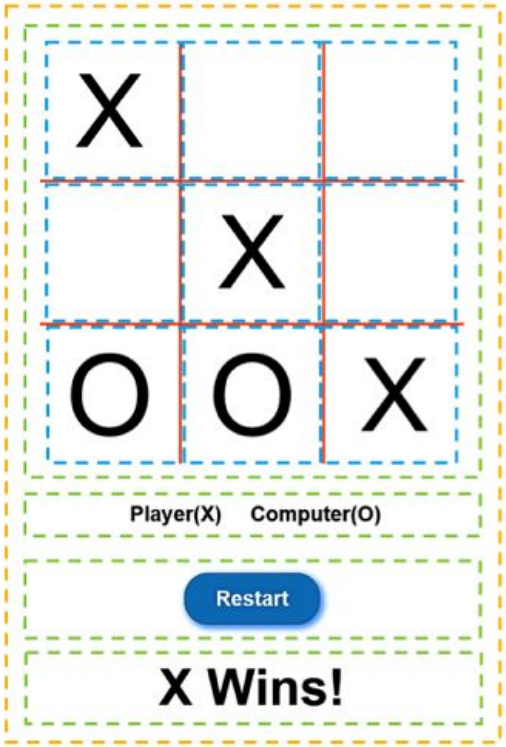
Angular is written in TypeScript.

It implements core and optional functionality

as a set of TypeScript libraries that you import into your apps.

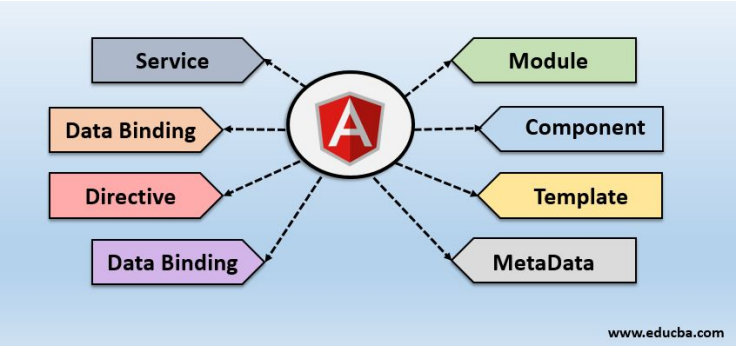


View your app as Components

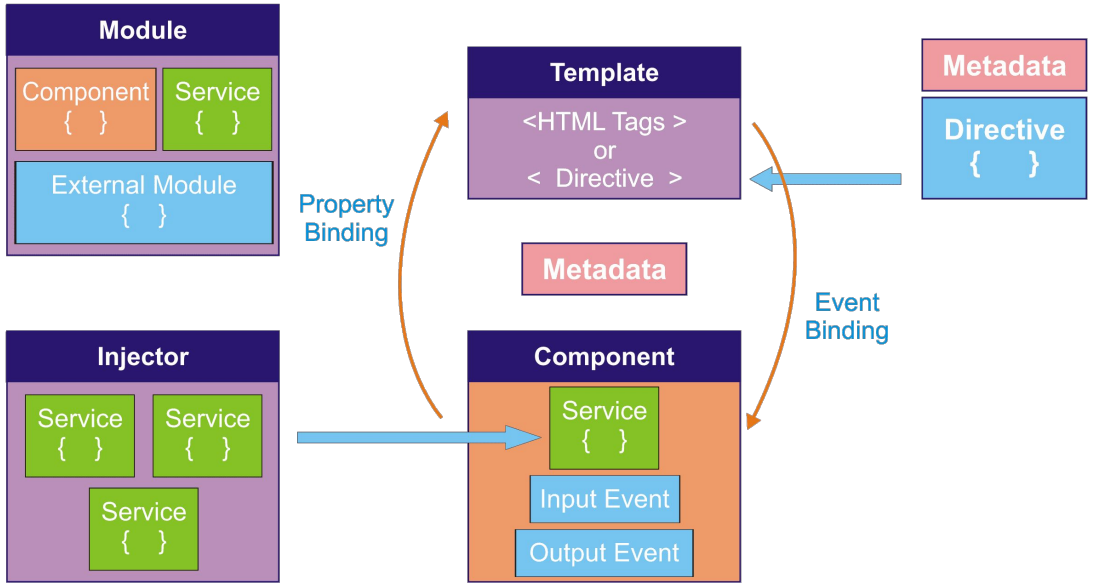


App
|-- Board
|-- Cells x 9
|-- Text
|-- RestartBtn
|-- GameStatusBar

Angular Architecture



https://www.educba.com/angular-2-architecture/



https://www.ngdevelop.tech/angular/architecture/

Let's dive in...

Prerequisites:

1. Good knowledge on JavaScript
2. Node.js
3. Visual Studio Code with the following extensions:
    a. Prettier
    b. ESLint
    c. Angular Language Service
    d. Bracket Pair Colorizer 2
    e. Nice theme (like Andromeda) :)
    f. Material Icon Theme

Angular CLI - Your best friend forever

Open CMD/Terminal and type:

npm install -g **@angular/cli**

ng new gocode-shop

Choose y for strict mode
Choose y for Angular Routing
And choose SCSS for preprocessor

code gocode-shop (To start VSCode)

Open terminal and type: **ng serve**

Open http://localhost:4200 **on browser**

Angular CLI Docs:
https://angular.io/cli

Cheetsheet:
https://gist.github.com/VaLeXaR/2d3e814e29b12809b5fd91773820cf31

Angular - Tour of Heroes

https://angular.io/tutorial

Add prior styling to our app

Use these styles on src/styles.scss:

```
/* Application-wide Styles */
h1 {
  color: #369;
  font-family: Arial, Helvetica, sans-serif;
  font-size: 250%;
}
h2, h3 {
  color: #444;
  font-family: Arial, Helvetica, sans-serif;
  font-weight: lighter;
}
body {
  margin: 2em;
}
body, input[type="text"], button {
  color: #333;
  font-family: Cambria, Georgia, serif;
}
/* everywhere else */
* {
  font-family: Arial, Helvetica, sans-serif;
}
```

Data binding on content

Use double curly braces of interpolation to display the app title.

Change the title property on **app.component.ts class**:

```
title = 'Tour of Heroes';
```

And use this title on **app.component.html**:

```
<h1>{{ title }}</h1>
```

Data binding on attribute

Angular lets us bind properties to the template easily and conveniently;

we saw that with interpolation. Now we'll see how to bind to an element's property (not to be confused with class properties).

We surround the wanted property with **square brackets** and pass it the class member:

On **app.component.html**:

```
<input [value]="title">
```

Event Binding

Event binding allows you to listen for and respond to user actions such as keystrokes, mouse movements, clicks, and touches.

On **app.component.html**:

```
<button (click)="onSave()">
```

On **app.component.ts**:

```
onSave(): void {
  alert("OUCHHH!!");
}
```

Heroes Component - Create Component

```
ng generate component heroes
```

Alias:

```
ng g c heroes
```

Template - heroes.component.html:

```
{{ hero }}
```

Script - heroes.component.html on class:

```
hero = 'Windstorm';
```

Heroes Component - Use Component

Replace all **app.component.html** content with:

```
<h1>{{title}}</h1>

<app-heroes></app-heroes>
```

Create a Hero interface

Create a Hero interface in **src/app/hero.ts**:

```
export interface Hero {
  id: number;
  name: string;
}
```

Use Hero interface

Use it on **heroes.component.ts** class:

```
import { Component, OnInit } from '@angular/core';
import { Hero } from '../hero';

@Component({
  selector: 'app-heroes',
  templateUrl: './heroes.component.html',
  styleUrls: ['./heroes.component.css']
})
export class HeroesComponent implements OnInit {
  hero: Hero = {
    id: 1,
    name: 'Windstorm'
  };

  constructor() { }

  ngOnInit() {
  }

}
```

Show it on **heroes.component.html**:

```
<h2>{{ hero.name }} Details</h2>
<div><span>id: </span>{{ hero.id }}</div>
<div><span>name: </span>{{ hero.name }}</div>
```

Format with the UppercasePipe

On **heroes.component.html**:

```
<h2>{{ hero.name | uppercase }} Details</h2>
```

Pipes are a good way to format strings, currency amounts, dates and other display data.

Angular ships with several built-in pipes and you can create your own.

DatePipe: Formats a date value according to locale rules.
UpperCasePipe: Transforms text to all upper case.
LowerCasePipe: Transforms text to all lower case.
CurrencyPipe: Transforms a number to a currency string, formatted according to locale rules.
DecimalPipe: Transforms a number into a string with a decimal point, formatted according to locale rules.
PercentPipe: Transforms a number to a percentage string, formatted according to locale rules.

Create a custom pipe:
https://angular.io/guide/pipes#creating-pipes-for-custom-data-transformations

Edit the Hero - two way data binding

On **heroes.component.html**:

```
<h2>{{hero.name | uppercase}} Details</h2>
<div><span>id: </span>{{hero.id}}</div>
<div>
  <label>name:
    <input [(ngModel)]="hero.name" placeholder="name"/>
  </label>
</div>
```

[(ngModel)] is Angular's two-way data binding syntax.

Notice that the app stopped working when you added [(ngModel)].

Template parse errors:
Can't bind to 'ngModel' since it isn't a known property of 'input'.

We'll see a solution in the next slide.

AppModule

Angular needs to know how the pieces of your application fit together and what other files and libraries the app requires. This information is called metadata.

Some of the metadata is in the **@Component** decorators that you added to your component classes. Other critical metadata is in **@NgModule** decorators.

On **app.module.ts**:

```
import { FormsModule } from '@angular/forms';

...
imports: [
  BrowserModule,
  FormsModule
],
```

Angular CLI declared **HeroesComponent** in the AppModule when it generated that component.

Create mock heroes

Create a new file /src/app/mock-heroes.ts and define a HEROES constant:

```
import { Hero } from './hero';

export const HEROES: Hero[] = [
  { id: 11, name: 'Dr Nice' },
  { id: 12, name: 'Narco' },
  { id: 13, name: 'Bombasto' },
  { id: 14, name: 'Celeritas' },
  { id: 15, name: 'Magneta' },
  { id: 16, name: 'RubberMan' },
  { id: 17, name: 'Dynama' },
  { id: 18, name: 'Dr IQ' },
  { id: 19, name: 'Magma' },
  { id: 20, name: 'Tornado' }
];
```

Displaying heroes

---

https://angular.io/api/common/NgForOf

NgForOf DIRECTIVE

A structural directive that renders a template for each item in a collection.
The directive is placed on an element, which becomes the parent of the cloned templates.

On **heroes.component.ts** file:

```
import { HEROES } from '../mock-heroes';

export class HeroesComponent implements OnInit {

  heroes = HEROES;
}
```

And loop over heroes with ***ngFor** directive, on **heroes.component.html**:

```
<h2>My Heroes</h2>
<ul class="heroes">
  <li *ngFor="let hero of heroes">
    <span class="badge">{{hero.id}}</span> {{hero.name}}
  </li>
</ul>
```

Styling heroes

On **heroes.component.css** use this gist:

https://gist.github.com/eladcandroid/aaaf73d7de2cbb38a2138c6be04b84c8

Master/Detail - Event Binding

Add a click event binding to the <li> like this:

On heroes.component.html:

```
<li *ngFor="let hero of heroes" (click)="onSelect(hero)">
```

Add the event handler on **heroes.component.ts:**

```
selectedHero: Hero;
onSelect(hero: Hero): void {
  this.selectedHero = hero;
}
```

Master/Detail - Details section

– – –

Add the following to **heroes.component.html** beneath the list section:

```
<h2>{{selectedHero.name | uppercase}} Details</h2>
<div><span>id: </span>{{selectedHero.id}}</div>
<div>
  <label>name:
    <input [(ngModel)]="selectedHero.name" placeholder="name"/>
  </label>
</div>
```

But we get an error…
HeroesComponent.html:3 ERROR TypeError: Cannot read property 'name' of undefined

Hmm… I know! Because there is no selected hero!

Hide empty details with *ngIf

Wrap the hero detail HTML in a . Add Angular's *ngIf directive to the and set it to selectedHero. On **heroes.component.html**:

```
<div *ngIf="selectedHero">

  <h2>{{selectedHero.name | uppercase}} Details</h2>
  <div><span>id: </span>{{selectedHero.id}}</div>
  <div>
    <label>name:
      <input [(ngModel)]="selectedHero.name" placeholder="name"/>
    </label>
  </div>

</div>
```

When selectedHero is undefined, the ngIf removes the hero detail from the DOM. There are no selectedHero bindings to consider.

When the user picks a hero, selectedHero has a value and ngIf puts the hero detail into the DOM.

What about else if and else?

Try this for else:

```
<div *ngIf="condition; else elseBlock">Content to render when condition is true.</div>
<ng-template #elseBlock>Content to render when condition is false.</ng-template>
```

You can also try this for if and else:

```
<div *ngIf="condition; then thenBlock else elseBlock"></div>
<ng-template #thenBlock>Content to render when condition is true.</ng-template>
<ng-template #elseBlock>Content to render when condition is false.</ng-template>
```

*ngIf Docs

Ng-Template

Article

<ng-template> is a template element that Angular uses with structural directives (*ngIf, *ngFor, [ngSwitch] and custom directives).

Angular wraps the host element (to which the directive is applied) inside <ng-template> and consumes the <ng-template> in the finished DOM by replacing it with diagnostic comments.

```
<div *ngIf="shouldSayHello" class="hello-world">Hello World</div>


<!-- Converted element -->
<ng-template [ngIf]="shouldSayHello">
  <div class="hello-world">Hello World</div>
</ng-template>
```

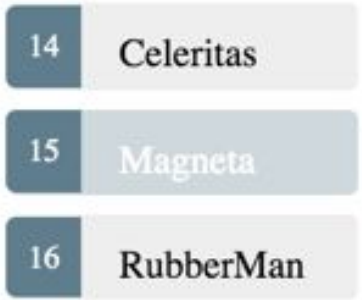
```
▼<body>
  ▼<app-root _nghost-c0 ng-version="6.1.10">
    <!--bindings={
      "ng-reflect-ng-if": "true"
    }-->
    <div _ngcontent-c0 class="hello-world">Hello World</div>
  </app-root>
```

There's also <ng-container> that it's a grouping element that doesn't interfere with styles or layout because Angular doesn't put it in the DOM

Style the selected hero

That selected hero coloring is the work of the .selected CSS class in the styles you added earlier. You just have to apply the .selected class to the <li> when the user clicks it. Add [class.selected] to **heores.component.html**:

```
<li *ngFor="let hero of heroes"
  [class.selected]="hero === selectedHero"
  (click)="onSelect(hero)">
  <span class="badge">{{hero.id}}</span> {{hero.name}}
</li>
```



NgClass Docs:
https://angular.io/api/common/NgClass

Create a feature component

You'll want to split up large components into smaller sub-components, each focused on a specific task or workflow.

The **HeroesComponent** will only present the list of heroes. The **HeroDetailComponent** will present details of a selected hero.

Use the Angular CLI to generate the component:

ng generate component **hero-detail**

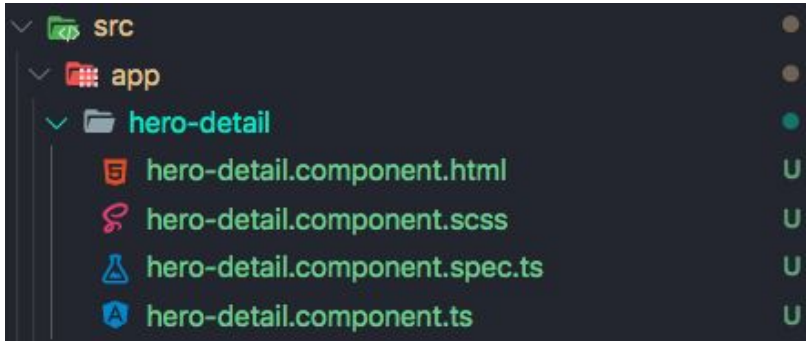
Or, in short syntax:

ng g c **hero-detail**

Create a feature component - result

Creates a directory src/app/hero-detail.

Inside that directory four files are generated:



A CSS file for the component styles.

An HTML file for the component template.

A TypeScript file with a component class named HeroDetailComponent.

A test file for the HeroDetailComponent class.

The command also adds the HeroDetailComponent as a declaration in the @NgModule decorator of the src/app/app.module.ts file.

Write the component template

Cut the HTML for the hero detail from the bottom of the HeroesComponent template and paste it over the generated boilerplate in the HeroDetailComponent template (**hero-detail.component.html**).

The pasted HTML refers to a selectedHero. The new HeroDetailComponent can present any hero, not just a selected hero. So replace "selectedHero" with "hero" everywhere in the template.

```
<div *ngIf="hero">

  <h2>{{hero.name | uppercase}} Details</h2>
  <div><span>id: </span>{{hero.id}}</div>
  <div>
    <label>name:
      <input [(ngModel)]="hero.name" placeholder="name"/>
    </label>
  </div>

</div>
```

Add the @Input() hero property

The HeroDetailComponent template binds to the component's hero property which is of type Hero.

Open the HeroDetailComponent class file (**hero-detail.component.ts**) and import the Hero symbol.

```
import { Component, OnInit, Input } from '@angular/core';
import { Hero } from '../hero';

...
export class HeroDetailComponent implements OnInit {

@Input() hero: Hero;
...
```

Update the HeroesComponent template

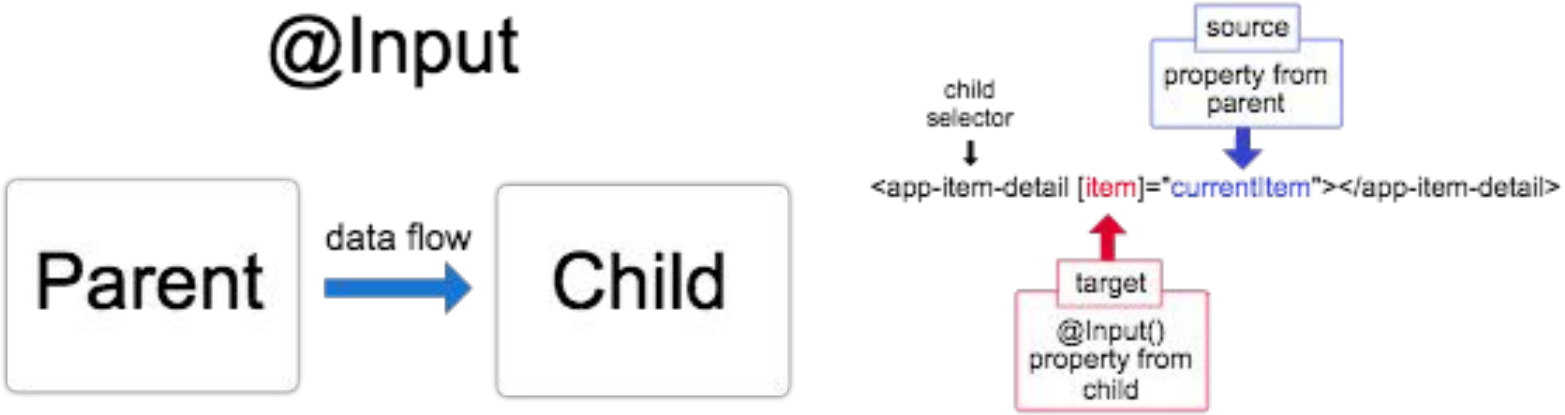
The HeroDetailComponent selector is 'app-hero-detail'. Add an <app-hero-detail> element near the bottom of the HeroesComponent template, where the hero detail view used to be.

Bind the HeroesComponent.selectedHero to the element's hero property.
Now you have this **heroes.component.html**:

```
<h2>My Heroes</h2>

<ul class="heroes">
  <li *ngFor="let hero of heroes"
    [class.selected]="hero === selectedHero"
    (click)="onSelect(hero)">
    <span class="badge">{{hero.id}}</span> {{hero.name}}
  </li>
</ul>

<app-hero-detail [hero]="selectedHero"></app-hero-detail>
```

Data Flow - From parent to child - @Input

Use the @Input() decorator in a child component or directive to let Angular know that a property in that component can receive its value from its parent component.



https://angular.io/guide/inputs-outputs#input

Use @Output property to inform the parent

Demo: https://stackblitz.com/angular/eqrajmaamqg

Add a new component, voter, with: **ng g c voter.** Add other component, votetaker, with: **ng g c votetaker**
The voter component will "tell" votetaker about its descision.

```
import { Component, EventEmitter, Input, Output }
        from '@angular/core';

@Component({
  selector: 'app-voter',
  template: `
    <h4>{{name}}</h4>
    <button (click)="vote(true)"
            [disabled]="didVote">Agree</button>
    <button (click)="vote(false)"
            [disabled]="didVote">Disagree</button>
  `
})
export class VoterComponent {
  @Input()  name: string;
  @Output() voted = new EventEmitter<boolean>();
  didVote = false;

  vote(agreed: boolean) {
    this.voted.emit(agreed);
    this.didVote = true;
  }
}
```

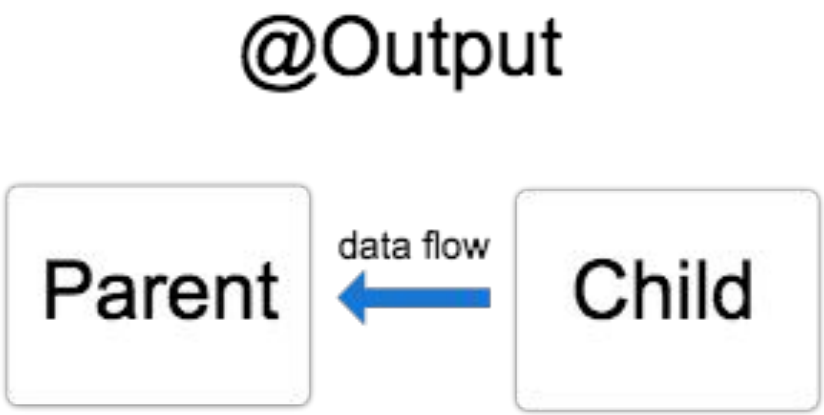
Voter Component (Child)

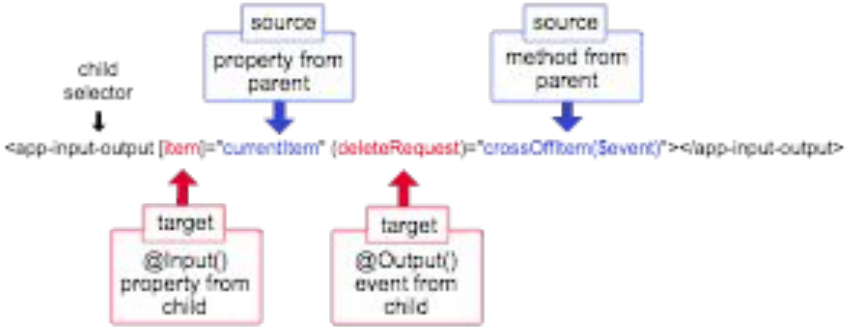
```
import { Component } from '@angular/core';

@Component({
  selector: 'app-vote-taker',
  template: `
    <h2>Should mankind colonize the Universe?</h2>
    <h3>Agree: {{agreed}}, Disagree: {{disagreed}}</h3>
    <app-voter *ngFor="let voter of voters"
      [name]="voter"
      (voted)="onVoted($event)">
    </app-voter>
  `
})
export class VoteTakerComponent {
  agreed = 0;
  disagreed = 0;
  voters = ['Narco', 'Celeritas', 'Bombasto'];

  onVoted(agreed: boolean) {
    agreed ? this.agreed++ : this.disagreed++;
  }
}
```

VoteTaker Component (Parent)

Data Flow - From child to parent - @Output

Use the @Output() decorator in a child component or directive to let Angular know that a property in that component can send its value to its parent component.





https://angular.io/guide/inputs-outputs#output

Add services

Components shouldn't fetch or save data directly and they certainly shouldn't knowingly present fake data. They should focus on presenting data and delegate data access to a service.

Add a service with Angular CLI:

```
ng generate service hero
```

Or, in a short way:

```
ng g s hero
```

The **@Injectable()** descorator marks the class as one that participates in the dependency injection system. The HeroService class is going to provide an injectable service, and it can also have its own injected dependencies. It doesn't have any dependencies yet, but it will soon.

Get hero data

The HeroService could get hero data from anywhere—a web service, local storage, or a mock data source.

Removing data access from components means you can change your mind about the implementation anytime, without touching any components. They don't know how the service works.

Import the Hero and HEROES on **hero.service.ts**, and add a getHeroes method to return the mock heroes:

```
import { Hero } from './hero';
import { HEROES } from './mock-heroes';
...

getHeroes(): Hero[] {
  return HEROES;
}
```

Update HeroesComponent

Import the HeroService instead of the HEROES import, on **heroes.component.ts**:

```
import { HeroService } from '../hero.service';
```

Replace the definition of the heroes property with a simple declaration and Inject the HeroService:

```
heroes: Hero[];
constructor(private heroService: HeroService) {}
```

The parameter simultaneously defines a private heroService property and identifies it as a HeroService injection site.

When Angular creates a HeroesComponent, the Dependency Injection system sets the heroService parameter to the singleton instance of HeroService. Now get the heroes with:

```
this.heroes = this.heroService.getHeroes();
```

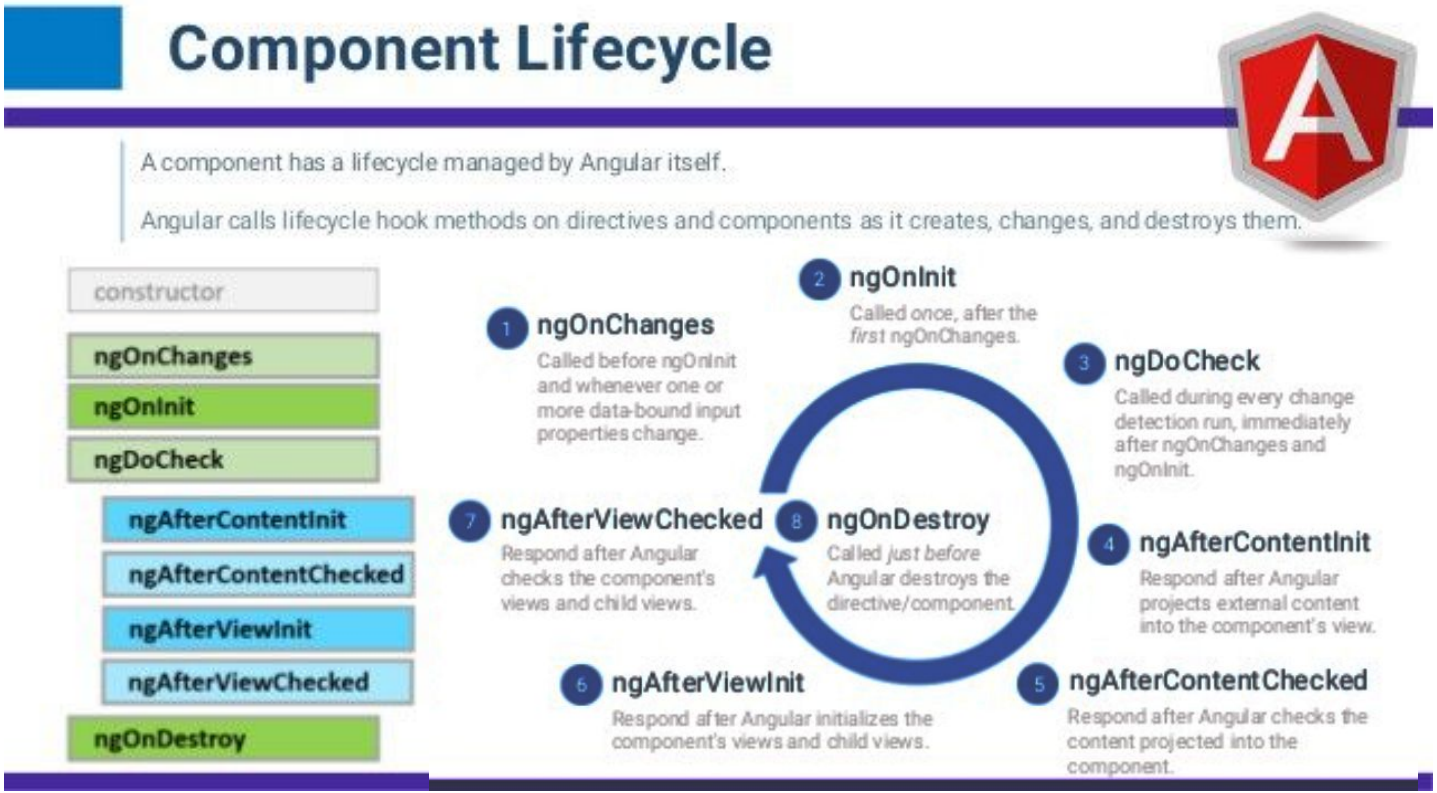
Get data on ngOnInit

Call getHeroes() inside the **ngOnInit** lifecycle hook and let Angular call ngOnInit() at an appropriate time after constructing a HeroesComponent instance. On **heroes.component.ts,** call getHeroes on the ngOnInit method:

```
ngOnInit() {
  this.getHeroes();
}
```

Your application can use lifecycle hook methods to tap into key events in the lifecycle of a component or directive in order to initialize new instances, initiate change detection when needed, respond to updates during change detection, and clean up before deletion of instances.

Lifecycle hooks



ngAfterContentInit vs. ngAfterViewInit

https://stackblitz.com/edit/github-zmvqb3

Observable HeroService

Observable is one of the key classes in the **RxJS** library.

In a later tutorial on HTTP, you'll learn that Angular's HttpClient methods return RxJS Observables. In this tutorial, you'll simulate getting data from the server with the RxJS of() function.

Open the HeroService file (**hero.service.ts**) and import the Observable and of symbols from RxJS.

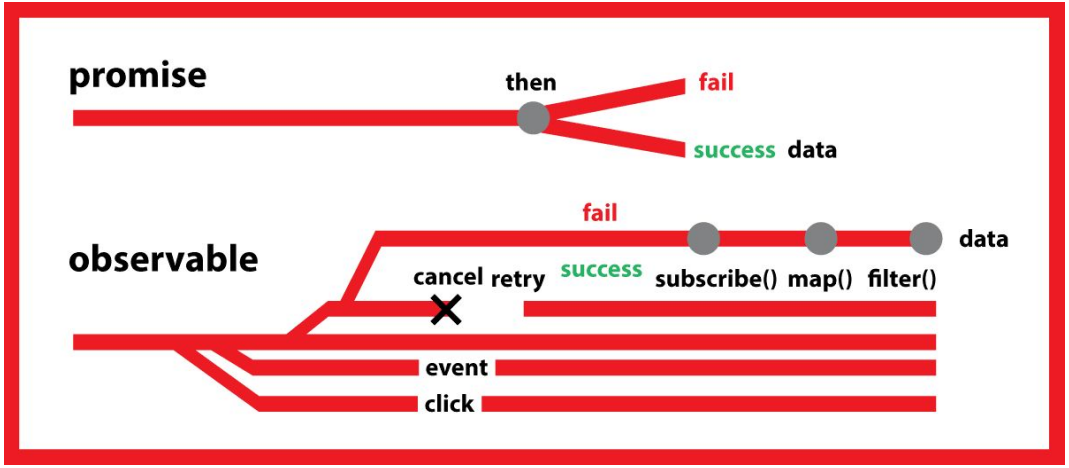
```
import { Observable, of } from 'rxjs';
```

RXJS Docs

Replace the getHeroes() method with the following:

```
getHeroes(): Observable<Hero[]> {
  return of(HEROES);
}
```

Differences between Promises and Observables



https://stackoverflow.com/a/43828666

Promise:

- having one pipeline
- usually only use with async data return
- not easy to cancel

Observable:

- are cancellable
- are re-triable by nature such as retry and retryWhen
- stream data in multiple pipelines
- having array-like operations like map, filter etc
- can be created from other sources like events
- they are functions, which could be subscribed later on

Subscribe in HeroesComponent

The HeroService.getHeroes method used to return a Hero[]. Now it returns an Observable<Hero[]>.

You'll have to adjust to that difference in HeroesComponent.

Find the getHeroes method and replace it with the following code

```
getHeroes(): void {
  this.heroService.getHeroes()
      .subscribe(heroes => this.heroes = heroes);
}
```

RXJS Docs

The new version waits for the Observable to emit the array of heroes—which could happen now or several minutes from now. The subscribe() method passes the emitted array to the callback, which sets the component's heroes property.

This asynchronous approach will work when the HeroService requests heroes from the server.

Show messages - Create MessageComponent

Use the CLI to create the MessagesComponent:

```
ng g c messages
```

Use the appropriate selector in **app.component.html**:

```
<h1>{{title}}</h1>
<app-heroes></app-heroes>
<app-messages></app-messages>
```

Create the MessageService

Use the CLI to create the MessageService in src/app

```
ng g s message
```

Open MessageService (**message.service.ts**) and replace its contents with the following:

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

@Injectable({
  providedIn: 'root',
})
export class MessageService {
  messages: string[] = [];

  add(message: string) {
    this.messages.push(message);
  }

  clear() {
    this.messages = [];
  }
}
```

Inject MessageService into the HeroService

In HeroService (**hero.service.ts**), import the MessageService and modify the constructor with a parameter that declares a private messageService property and modify the getHeroes() method to send a message when the heroes are fetched.

```
import { MessageService } from './message.service';

...

constructor(private messageService: MessageService) { }


getHeroes(): Observable<Hero[]> {
  // TODO: send the message _after_ fetching the heroes
  this.messageService.add('HeroService: fetched heroes');
  return of(HEROES);
}
```

Display the message from HeroService

Open MessagesComponent and import the MessageService. Modify the constructor with a parameter that declares a public messageService property. The messageService property must be **public** because you're going to bind to it in the template. On **messages.component.ts**:

```
import { MessageService } from '../message.service';

...

constructor(public messageService: MessageService) {}
```

Replace the **messages.component.html** template with the following:

```
<div *ngIf="messageService.messages.length">

  <h2>Messages</h2>
  <button class="clear"
          (click)="messageService.clear()">clear</button>
  <div *ngFor='let message of messageService.messages'> {{message}} </div>

</div>
```

Add additional messages to hero service

On **heroes.component.ts**, replace **onSelect** method with the following:

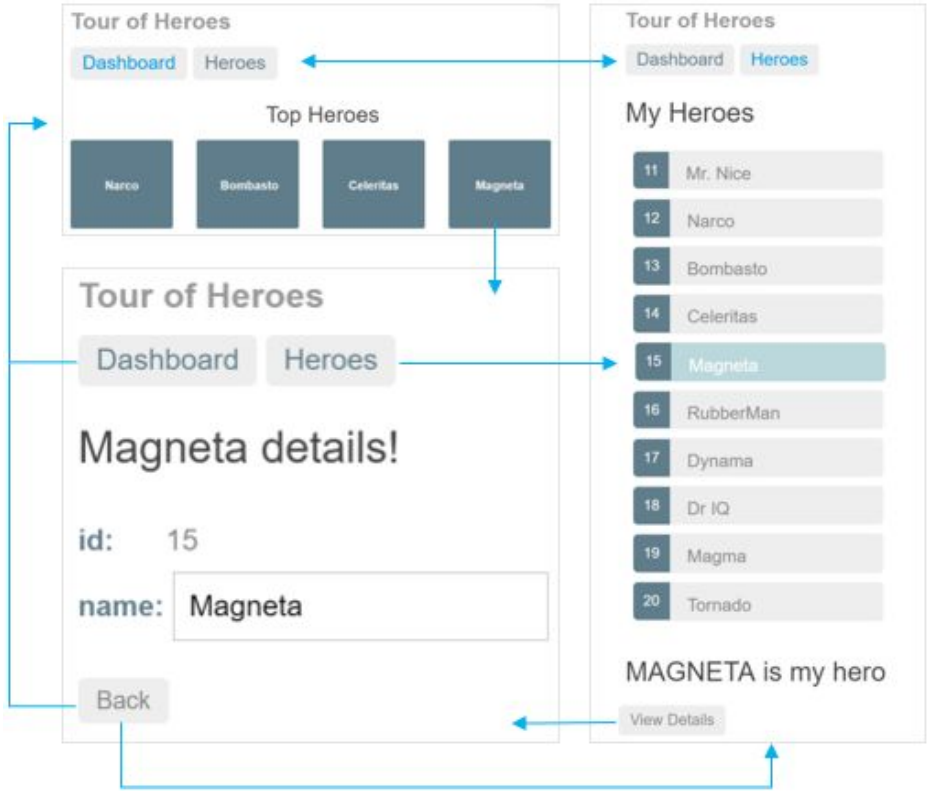
```
onSelect(hero: Hero): void {
    this.selectedHero = hero;
    this.messageService.add(`HeroesComponent: Selected hero id=${hero.id}`);
}
```

Style for MessagesComponent (**messages.component.scss**)

```
/* MessagesComponent's private CSS styles */
h2 {
  color: red;
  font-family: Arial, Helvetica, sans-serif;
  font-weight: lighter;
}

button.clear {
  font-family: Arial, sans-serif;
  color: #333;
  background-color: #eee;
  margin-bottom: 12px;
  border: none;
  padding: 5px 10px;
  border-radius: 4px;
  cursor: pointer;
}
button:hover {
  background-color: #cfd8dc;
}
```

Add in-app navigation with routing

When you're done,
users will be able
to navigate the app like this:



Add the AppRoutingModule & New route

Add the routing module with this CLI command: (If not present)

```
ng g m app-routing --flat --module=app
```

Replace routes array with:

```
const routes: Routes = [
  { path: 'heroes', component: HeroesComponent }
];
```

AppRoutingModule exports RouterModule so it will be available throughout the app.

Add RouterOutlet

Open the AppComponent template and replace the <app-heroes> element with a <router-outlet> element.

```
<h1>{{title}}</h1>
<router-outlet></router-outlet>
<app-messages></app-messages>
```

The AppComponent template no longer needs <app-heroes> because the app will only display the HeroesComponent when the user navigates to it.

Append /heroes to the URL in the browser address bar. You should see the familiar heroes master/detail view.

Add a navigation link

Add a <nav> element and, within that, an anchor element that, when clicked, triggers navigation to the HeroesComponent. On **app.component.html**:

```
<h1>{{title}}</h1>
<nav>
  <a routerLink="/heroes">Heroes</a>
</nav>
<router-outlet></router-outlet>
<app-messages></app-messages>
```

Append /heroes to the URL in the browser address bar. You should see the familiar heroes master/detail view.

Add navigation styling

app.component.scss

```scss
/* AppComponent's private CSS styles */
h1 {
  font-size: 1.2em;
  margin-bottom: 0;
}
nav a {
  padding: 5px 10px;
  text-decoration: none;
  margin-top: 10px;
  display: inline-block;
  background-color: #eee;
  border-radius: 4px;
}
nav a:visited, a:link {
  color: #334953;
}
nav a:hover {
  color: #039be5;
  background-color: #cfd8dc;
}
nav a.active {
  color: #039be5;
}
```

Add a dashboard view

Routing makes more sense when there are multiple views. So far there's only the heroes view. Add a DashboardComponent using the CLI:

```
ng g c dashboard
```

Replace the default file content in these three files as follows:

https://gist.github.com/eladcandroid/9e0790e1358352fce8f19e500c1cd02b

Add the dashboard route

Import the **DashboardComponent** in the **app-routing-module.ts** file and add an appropriate route:

```
import { DashboardComponent } from './dashboard/dashboard.component';

...

{ path: 'dashboard', component: DashboardComponent },
```

Add a default route

When the app starts, the browser's address bar points to the web site's root. That doesn't match any existing route so the router doesn't navigate anywhere. The space below the <router-outlet> is blank.

To make the app navigate to the dashboard automatically, add the following route to the routes array.

```
{ path: '', redirectTo: '/dashboard', pathMatch: 'full' },
```

Add dashboard link to the shell

The user should be able to navigate back and forth between the DashboardComponent and the HeroesComponent by clicking links in the navigation area near the top of the page.

Add a dashboard navigation link to the AppComponent shell template, just above the Heroes link, on **app.component.html**:

```
<h1>{{title}}</h1>
<nav>
  <a routerLink="/dashboard">Dashboard</a>
  <a routerLink="/heroes">Heroes</a>
</nav>
<router-outlet></router-outlet>
<app-messages></app-messages>
```

Add a hero detail route

Open the HeroesComponent template (heroes/**heroes.component.html**) and delete the **<app-hero-detail>** element from the bottom.

A URL like **~/detail/11** would be a good URL for navigating to the Hero Detail view of the hero whose id is 11.

Open **app-routing.module.ts** and import HeroDetailComponent.

```
import { HeroDetailComponent } from './hero-detail/hero-detail.component';

...

{ path: 'detail/:id', component: HeroDetailComponent },
```

DashboardComponent hero links

Now that the router has a route to HeroDetailComponent, fix the dashboard hero links to navigate via the parameterized dashboard route.

On **dashboard.component.html**:

```
<a *ngFor="let hero of heroes" class="col-1-4"
    routerLink="/detail/{{hero.id}}">
  <div class="module hero">
    <h4>{{hero.name}}</h4>
  </div>
</a>
```

HeroesComponent hero links

Change **heroes.component.html** from click method to routerLink:

```
<ul class="heroes">
  <li *ngFor="let hero of heroes">
    <a routerLink="/detail/{{hero.id}}">
      <span class="badge">{{hero.id}}</span> {{hero.name}}
    </a>
  </li>
</ul>
```

And here's your new styling:
https://gist.github.com/eladcandroid/06f970db59d2a7194def47aa5c5e716e

Routable HeroDetailComponent

Now the router creates the HeroDetailComponent in response to a URL such as ~/detail/11.

The HeroDetailComponent needs a new way to obtain the hero-to-display. This section explains the following:

- Get the route that created it
- Extract the id from the route
- Acquire the hero with that id from the server via the HeroService

Routable HeroDetailComponent

On **hero-detail.component.ts**, add the appropriate imports and inject the ActivatedRoute, HeroService, and Location services into the constructor, saving their values in private fields:

```
import { ActivatedRoute } from '@angular/router';
import { Location } from '@angular/common';

import { HeroService } from '../hero.service';


...

constructor(
  private route: ActivatedRoute,
  private heroService: HeroService,
  private location: Location
) {}
```

The **ActivatedRoute** holds information about the route to this instance of the **HeroDetailComponent.**

This component is interested in the route's parameters extracted from the URL.
The "id" parameter is the id of the hero to display.

Routable HeroDetailComponent - Get data

On **hero-detail.component.ts**,
get the Hero data on ngOnInit lifecycle:

```
ngOnInit(): void {
  this.getHero();
}

getHero(): void {
  const id = +this.route.snapshot.paramMap.get('id');
  this.heroService.getHero(id)
    .subscribe(hero => this.hero = hero);
}
```

The route.snapshot is a static image of the route information shortly after the component was created.

The paramMap is a dictionary of route parameter values extracted from the URL. The "id" key returns the id of the hero to fetch.

Route parameters are always strings. The JavaScript (+) operator converts the string to a number, which is what a hero id should be.

The browser refreshes and the app crashes with a compiler error. HeroService doesn't have a getHero() method. Add it now.

Add HeroService.getHero()

Open **hero.service.ts** and add the following **getHero()** method with the id after the getHeroes() method:

```
getHero(id: number): Observable<Hero> {
  // TODO: send the message _after_ fetching the hero
  this.messageService.add(`HeroService: fetched hero id=${id}`);
  return of(HEROES.find(hero => hero.id === id));
}
```

If you paste **localhost:4200/detail/11** in the browser address bar, the router navigates to the detail view for the hero with id: 11, "Dr Nice".

Find the way back

Add a go back button to the bottom of the component template and bind it to the component's goBack() method. On **hero-detail.component.html**:

```
<button (click)="goBack()">go back</button>
```

Add a goBack() method to the component class that navigates backward one step in the browser's history stack using the Location service that you injected previously. On **hero-detail.component.ts**:

```
goBack(): void {
  this.location.back();
}
```

To add a 404 page - Read this: https://angular.io/guide/router-tutorial#adding-a-404-page

Enable HTTP Services

HttpClient is Angular's mechanism for communicating with a remote server over HTTP.

Make HttpClient available everywhere in the app in two steps. First, add it to the root AppModule by importing it, on **app.module.ts**:

```
import { HttpClientModule } from '@angular/common/http';


@NgModule({
  imports: [
    HttpClientModule,
  ],
})
```

Simulate a data server

This tutorial sample mimics communication with a remote data server by using the **In-memory Web API module**.

After installing the module, the app will make requests to and receive responses from the HttpClient without knowing that the In-memory Web API is intercepting those requests, applying them to an in-memory data store, and returning simulated responses.

By using the In-memory Web API, you won't have to set up a server to learn about HttpClient.

```
npm install angular-in-memory-web-api
```

Import In-memory Web API module

On **app.module.ts:**

```
import { HttpClientInMemoryWebApiModule } from 'angular-in-memory-web-api';
import { InMemoryDataService } from './in-memory-data.service';


imports: [ …

HttpClientInMemoryWebApiModule.forRoot(
  InMemoryDataService, { dataEncapsulation: false }
)
```

We'll create the service we just imported..

Import In-memory Web API module

Generate the class **src/app/in-memory-data.service.ts** with the following command:

```
ng g s InMemoryData
```

Import In-memory Web API module

Replace the default contents of **in-memory-data.service.ts** with the following:

```
import { Injectable } from '@angular/core';
import { InMemoryDbService } from 'angular-in-memory-web-api';
import { Hero } from './hero';

@Injectable({
  providedIn: 'root',
})
export class InMemoryDataService implements InMemoryDbService {
  createDb() {
    const heroes = [
{ id: 11, name: 'Dr Nice' },
      { id: 12, name: 'Narco' },
      { id: 13, name: 'Bombasto' },
      { id: 14, name: 'Celeritas' },
      { id: 15, name: 'Magneta' },
      { id: 16, name: 'RubberMan' },
      { id: 17, name: 'Dynama' },
      { id: 18, name: 'Dr IQ' },
      { id: 19, name: 'Magma' },
      { id: 20, name: 'Tornado' }
    ];
    return {heroes};
  }

  // Overrides the genId method
  // to ensure that a hero always has an id.
  // If the heroes array is empty,
  // the method below returns the initial number (11).
  // if the heroes array is not empty,
  // the method below returns the highest
  // hero id + 1.
  genId(heroes: Hero[]): number {
    return heroes.length > 0 ?
      Math.max(...heroes.map(hero => hero.id)) + 1 : 11;
  }
}
```

Heores and HTTP

In the HeroService (**hero.service.ts**):

1. import HttpClient and HttpHeaders.

2. Inject HttpClient into the constructor in a private property called http.

3. Notice that you keep injecting the MessageService but since you'll call it so frequently, wrap it in a private log() method

4. Define the heroesUrl of the form :base/:collectionName with the address of the heroes resource on the server. Here base is the resource to which requests are made, and collectionName is the heroes data object in the in-memory-data-service.ts.

Heores and HTTP

```
import { HttpClient, HttpHeaders } from '@angular/common/http';

...

constructor(
  private http: HttpClient,
  private messageService: MessageService) { }


/** Log a HeroService message with the MessageService */
private log(message: string) {
  this.messageService.add(`HeroService: ${message}`);
}

private heroesUrl = 'api/heroes';


/** GET heroes from the server */
getHeroes(): Observable<Hero[]> {
  return this.http.get<Hero[]>(this.heroesUrl)
}
```

Error handling

Things go wrong, especially when you're getting data from a remote server.

The HeroService.getHeroes() method should catch errors and do something appropriate.

To catch errors, you "pipe" the observable result from http.get() through an RxJS catchError() operator.

```
import { catchError, map, tap } from 'rxjs/operators';

...

getHeroes(): Observable<Hero[]> {
  return this.http.get<Hero[]>(this.heroesUrl)
    .pipe(
      catchError(this.handleError<Hero[]>('getHeroes', []))
    );
}
```

Error handling

The following handleError() will be shared by many HeroService methods so it's generalized to meet their different needs.

```
private handleError<T>(operation = 'operation', result?: T) {
  return (error: any): Observable<T> => {

    // TODO: send the error to remote logging infrastructure
    console.error(error); // log to console instead

    // TODO: better job of transforming error for user consumption
    this.log(`${operation} failed: ${error.message}`);

    // Let the app keep running by returning an empty result.
    return of(result as T);
  };
}
```

Tap into the Observable

The HeroService methods will **tap** into the flow of observable values and send a message, via the log() method, to the message area at the bottom of the page.

```
getHeroes(): Observable<Hero[]> {
  return this.http.get<Hero[]>(this.heroesUrl)
    .pipe(
      tap(_ => this.log('fetched heroes')),
      catchError(this.handleError<Hero[]>('getHeroes', []))
    );
}
```

Get hero by id

Most web APIs support a get by id request in the form :baseURL/:id.

Here, the base URL is the heroesURL defined in the Heroes and HTTP section (api/heroes) and id is the number of the hero that you want to retrieve. For example, api/heroes/11.

Update the HeroService getHero() method with the following to make that request:

```
/** GET hero by id. Will 404 if id not found */
getHero(id: number): Observable<Hero> {
  const url = `${this.heroesUrl}/${id}`;
  return this.http.get<Hero>(url).pipe(
    tap(_ => this.log(`fetched hero id=${id}`)),
    catchError(this.handleError<Hero>(`getHero id=${id}`))
  );
}
```

Update heroes

Edit a hero's name in the hero detail view. As you type, the hero name updates the heading at the top of the page. But when you click the "go back button", the changes are lost.

If you want changes to persist, you must write them back to the server.

At the end of the hero detail template, add a save button with a click event binding that invokes a new component method named save().

On **hero-detail.component.html**:

```
<button (click)="save()">save</button>
```

On **hero-detail.component.ts**:

```
save(): void {
  this.heroService.updateHero(this.hero)
    .subscribe(() => this.goBack());
}
```

Add HeroService.updateHero()

The overall structure of the updateHero() method is similar to that of getHeroes(), but it uses http.put() to persist the changed hero on the server. Add the following to the HeroService (**hero.service.ts**):

```
/** PUT: update the hero on the server */
updateHero(hero: Hero): Observable<any> {
  return this.http.put(this.heroesUrl, hero, this.httpOptions).pipe(
    tap(_ => this.log(`updated hero id=${hero.id}`)),
    catchError(this.handleError<any>('updateHero'))
  );
}
```

The heroes web API expects a special header in HTTP save requests.
That header is in the httpOptions constant defined in the HeroService.
Add the following to the HeroService class:

```
httpOptions = {
  headers: new HttpHeaders({ 'Content-Type': 'application/json' })
};
```

Add a new hero - Template

To add a hero, this app only needs the hero's name. You can use an <input> element paired with an add button.

Insert the following into the HeroesComponent template (**heroes.component.html**), just after the heading:

```
<div>
  <label>Hero name:
    <input #heroName />
  </label>
  <!-- (click) passes input value to add() and then clears the input -->
  <button (click)="add(heroName.value); heroName.value=''">
    add
  </button>
</div>
```

Add a new hero - Logic

In response to a click event, call the component's click handler, add(), and then clear the input field so that it's ready for another name. Add the following to the HeroesComponent class (**heroes.component.ts**):

```
add(name: string): void {
  name = name.trim();
  if (!name) { return; }
  this.heroService.addHero({ name } as Hero)
    .subscribe(hero => {
      this.heroes.push(hero);
    });
}
```

Add a new hero - Service

Add the following addHero() method to the HeroService class (**hero.service.ts**):

```
/** POST: add a new hero to the server */
addHero(hero: Hero): Observable<Hero> {
  return this.http.post<Hero>(this.heroesUrl, hero, this.httpOptions).pipe(
    tap((newHero: Hero) => this.log(`added hero w/ id=${newHero.id}`)),
    catchError(this.handleError<Hero>('addHero'))
  );
}
```

Delete a new hero - Template

Add the following button element to the HeroesComponent template (**heroes.component.html**), after the hero name in the repeated <li> element.

```
<button class="delete" title="delete hero"
  (click)="delete(hero)">x</button>
```

The HTML for the list of heroes should look like this:

```
<ul class="heroes">
  <li *ngFor="let hero of heroes">
    <a routerLink="/detail/{{hero.id}}">
      <span class="badge">{{hero.id}}</span> {{hero.name}}
    </a>
    <button class="delete" title="delete hero"
      (click)="delete(hero)">x</button>
  </li>
</ul>
```

Delete a new hero - Styling

heroes.component.scss

```scss
button {
  background-color: #eee;
  border: none;
  padding: 5px 10px;
  border-radius: 4px;
  cursor: pointer;
  font-family: Arial, sans-serif;
}

button:hover {
  background-color: #cfd8dc;
}

button.delete {
  position: relative;
  left: 194px;
  top: -32px;
  background-color: gray !important;
  color: white;
}
```

Delete a new hero - Logic

Add the delete() handler to the component class (**heroes.component.ts**):

```
delete(hero: Hero): void {
  this.heroes = this.heroes.filter(h => h !== hero);
  this.heroService.deleteHero(hero).subscribe();
}
```

Delete a new hero - Service

Next, add a deleteHero() method to HeroService (**hero.service.ts**) like this.

```
/** DELETE: delete the hero from the server */
deleteHero(hero: Hero | number): Observable<Hero> {
  const id = typeof hero === 'number' ? hero : hero.id;
  const url = `${this.heroesUrl}/${id}`;

  return this.http.delete<Hero>(url, this.httpOptions).pipe(
    tap(_ => this.log(`deleted hero id=${id}`)),
    catchError(this.handleError<Hero>('deleteHero'))
  );
}
```

HeroService.searchHeroes()

Start by adding a searchHeroes() method to the HeroService (**hero.service.ts**)

```
/* GET heroes whose name contains search term */
searchHeroes(term: string): Observable<Hero[]> {
  if (!term.trim()) {
    // if not search term, return empty hero array.
    return of([]);
  }
  return this.http.get<Hero[]>(`${this.heroesUrl}/?name=${term}`).pipe(
    tap(x => x.length ?
       this.log(`found heroes matching "${term}"`) :
       this.log(`no heroes matching "${term}"`)),
    catchError(this.handleError<Hero[]>('searchHeroes', []))
  );


}
```

Add search to the Dashboard

Open the DashboardComponent template (**dashboard.component.html**) and add the hero search element, <app-hero-search>, to the bottom of the markup.

```
<h3>Top Heroes</h3>
<div class="grid grid-pad">
  <a *ngFor="let hero of heroes" class="col-1-4"
      routerLink="/detail/{{hero.id}}">
    <div class="module hero">
      <h4>{{hero.name}}</h4>
    </div>
  </a>
</div>

<app-hero-search></app-hero-search>
```

Create HeroSearchComponent

Create a HeroSearchComponent with the CLI.

```
ng g c hero-search
```

HeroSearchComponent - Template

Replace the generated HeroSearchComponent template (**hero-search.component.html**) with an <input> and a list of matching search results, as follows.

```
<div id="search-component">
  <h4><label for="search-box">Hero Search</label></h4>

  <input #searchBox id="search-box" (input)="search(searchBox.value)" />

  <ul class="search-result">
    <li *ngFor="let hero of heroes$ | async" >
      <a routerLink="/detail/{{hero.id}}">
        {{hero.name}}
      </a>
    </li>
  </ul>
</div>
```

The *ngFor repeats hero objects. Notice that the *ngFor iterates over a list called heroes$, not heroes. The $ is a convention that indicates heroes$ is an Observable, not an array.

Since *ngFor can't do anything with an Observable, use the **pipe character (|) followed by async**. This identifies Angular's AsyncPipe and subscribes to an Observable automatically so you won't have to do so in the component class.

HeroSearchComponent - Styling

Use this styling for hero-search.component.scss:

```
/* HeroSearch private styles */
.search-result li {
  border-bottom: 1px solid gray;
  border-left: 1px solid gray;
  border-right: 1px solid gray;
  width: 195px;
  height: 16px;
  padding: 5px;
  background-color: white;
  cursor: pointer;
  list-style-type: none;
}

.search-result li:hover {
  background-color: #607D8B;
}
```

```
.search-result li a {
  color: #888;
  display: block;
  text-decoration: none;
}

.search-result li a:hover {
  color: white;
}
.search-result li a:active {
  color: white;
}
#search-box {
  width: 200px;
  height: 20px;
}


ul.search-result {
  margin-top: 0;
  padding-left: 0;
}
```

HeroSearchComponent - Logic

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

import { Observable, Subject } from 'rxjs';

import {
   debounceTime, distinctUntilChanged, switchMap
 } from 'rxjs/operators';

import { Hero } from '../hero';
import { HeroService } from '../hero.service';

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

Passing a new search term directly to the `searchHeroes()` after every user keystroke would create an excessive amount of HTTP requests, taxing server resources and burning through data plans.

- **`debounceTime(300)`** waits until the flow of new string events pauses for 300 milliseconds before passing along the latest string. You'll never make requests more frequently than 300ms.
- **`distinctUntilChanged()`** ensures that a request is sent only if the filter text changed.
- **`switchMap()`** calls the search service for each search term that makes it through `debounce()` and `distinctUntilChanged()`. It cancels and discards previous search observables, returning only the latest search service observable.

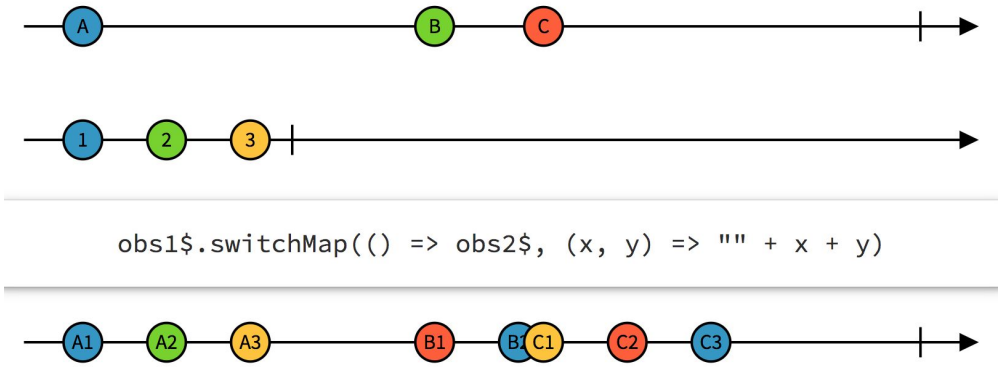
```
  heroes$: Observable<Hero[]>;
  private searchTerms = new Subject<string>();

  constructor(private heroService: HeroService) {}

  // Push a search term into the observable stream.
  search(term: string): void {
    this.searchTerms.next(term);
  }

  ngOnInit(): void {
    this.heroes$ = this.searchTerms.pipe(
      // wait 300ms after each keystroke before considering the term
      debounceTime(300),

      // ignore new term if same as previous term
      distinctUntilChanged(),

      // switch to new search observable each time the term changes
      switchMap((term: string) => this.heroService.searchHeroes(term)),
    );
  }
}
```

HeroSearchComponent - switchMap

With the switchMap operator, every qualifying key event can trigger an HttpClient.get() method call. Even with a 300ms pause between requests, you could have multiple HTTP requests in flight and they may not return in the order sent.

switchMap() preserves the original request order while returning only the observable from the most recent HTTP method call. Results from prior calls are canceled and discarded.

Note that canceling a previous searchHeroes() Observable doesn't actually abort a pending HTTP request. Unwanted results are simply discarded before they reach your application code.



Forms in Angular - Two approaches

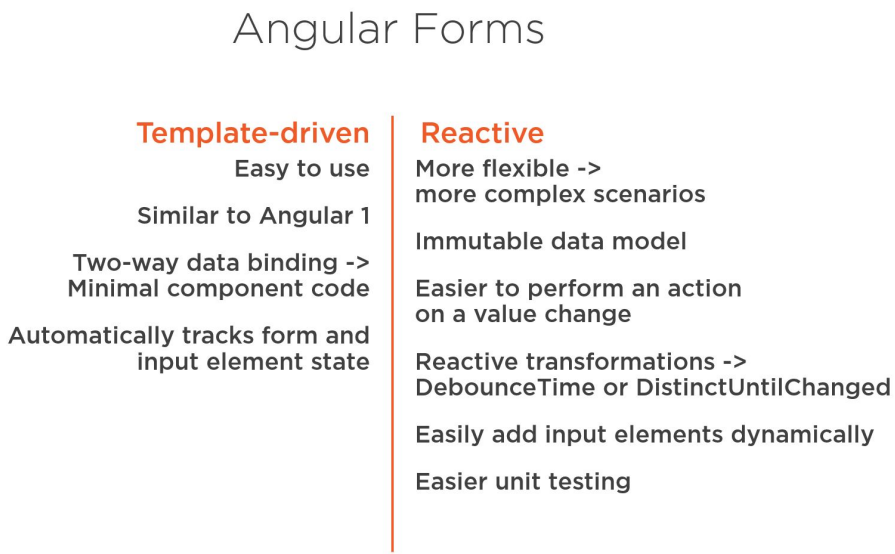
Template-driven forms rely on directives in the template to create and manipulate the underlying object model.

They are useful for adding a simple form to an app, such as an email list signup form. They're easy to add to an app, but they don't scale as well as reactive forms.

Reactive forms provide direct, explicit access to the underlying forms object model.

Compared to template-driven forms, they are more robust: they're more scalable, reusable, and testable.

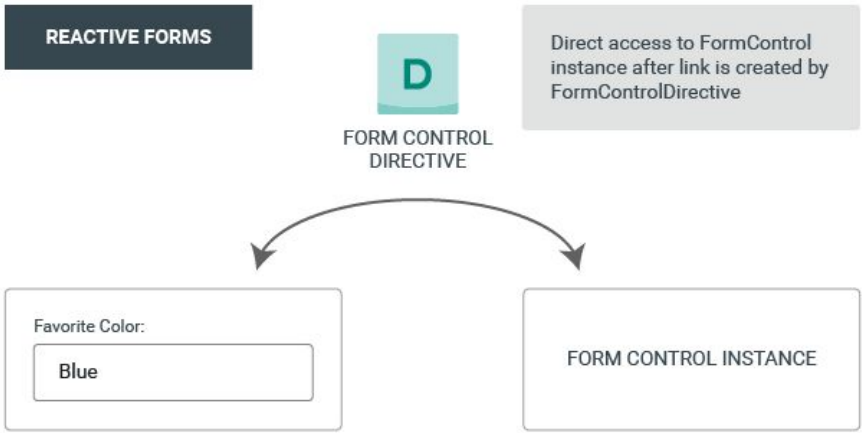
If forms are a key part of your application, or you're already using reactive patterns for building your application, use reactive forms.



Setting up the form model in reactive forms

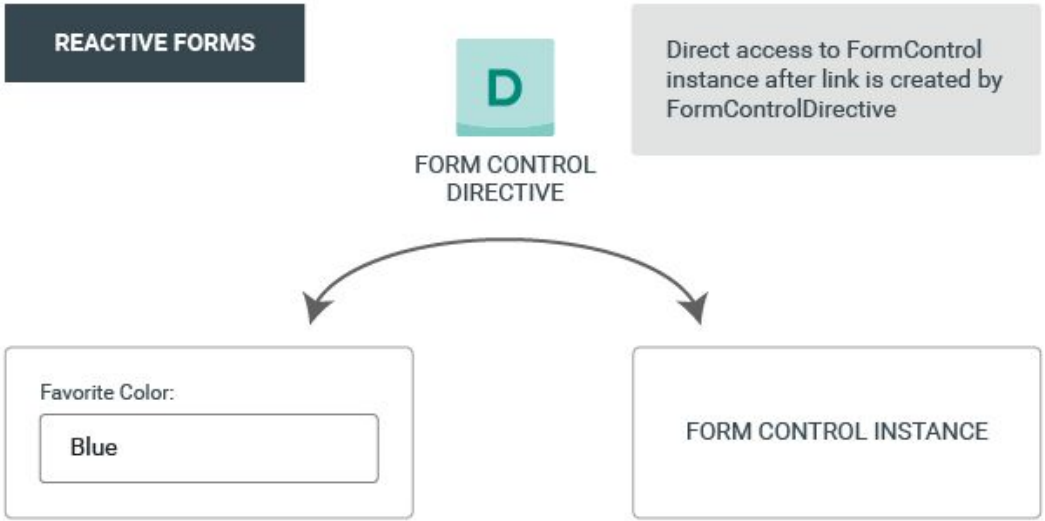
With reactive forms, you define the form model directly in the component class.

The **[formControl]** directive links the explicitly created FormControl instance to a specific form element in the view, using an internal value accessor.

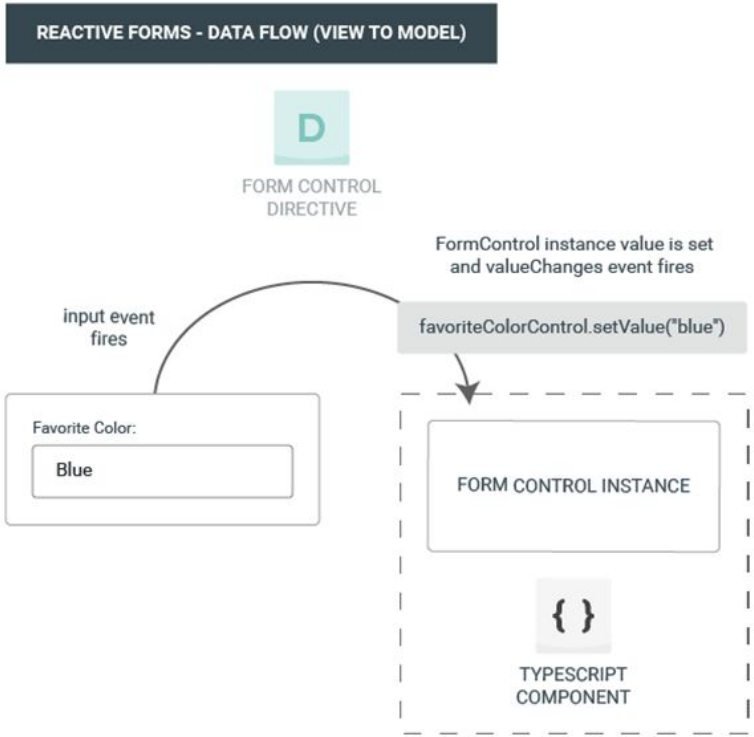



```
import { Component } from '@angular/core';
import { FormControl } from '@angular/forms';

@Component({
  selector: 'app-reactive-favorite-color',
  template: `
    Favorite Color: <input type="text"
                    [formControl]="favoriteColorControl">
  `
})
export class FavoriteColorComponent {
  favoriteColorControl = new FormControl('');
}
```

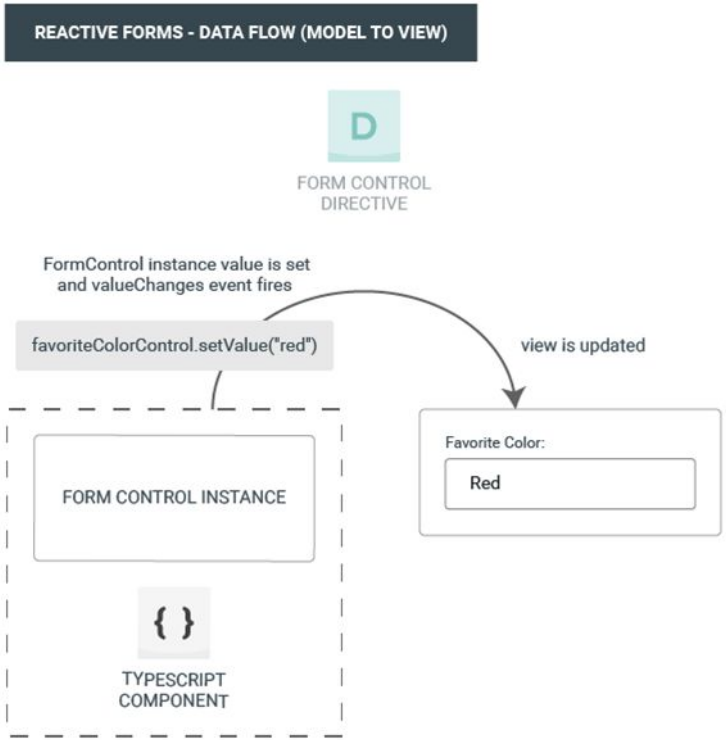
Data flow in reactive forms



Data flow in reactive forms - View to model



Data flow in reactive forms - Model to view



Adding a basic form control

To use reactive form controls, import **ReactiveFormsModule** from the **@angular/forms** package and add it to your **app.module.ts** imports array.

```
import { ReactiveFormsModule } from '@angular/forms';

@NgModule({
  imports: [
    // other imports ...
    ReactiveFormsModule
  ],
})
export class AppModule { }
```

Generate a new FormControl

Use the CLI command ng generate to generate a component in your project to host the control.

```
ng g c NameEditor
```

To register a single form control, import the FormControl class in **name-editor.component.ts** and create a new instance of FormControl to save as a class property.

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

@Component({
  selector: 'app-name-editor',
  templateUrl: './name-editor.component.html',
  styleUrls: ['./name-editor.component.css']
})
export class NameEditorComponent {
  name = new FormControl('');
}
```

Register the control in the template

After you create the control in the component class, you must associate it with a form control element in the template.

Update **name-editor.component.html** with the form control using the formControl binding provided by FormControlDirective, which is also included in the ReactiveFormsModule.

```
<label>
  Name:
  <input type="text" [formControl]="name">
</label>
```

The form control assigned to name is displayed when the component is added to **app.component.html**.

```
<app-name-editor></app-name-editor>
```

Displaying & changing a form control value

The following example shows you how to display the current value using interpolation in the template **name-editor.component.html**.

```
<p>
  Value: {{ name.value }}
</p>
```

The following example adds a method to the component class **name-editor.component.ts** to update the value of the control to Nancy using the setValue() method.

```
updateName() {
  this.name.setValue('Nancy');
}
```

Update the template **name-editor.component.html** with a button to simulate a name update.

```
<p>
  <button (click)="updateName()">Update Name</button>
</p>
```

Grouping form controls

Reactive forms provide two ways of grouping multiple related controls into a single input form.

A form group defines a form with a fixed set of controls that you can manage together.
You can also nest form groups to create more complex forms.

A form array defines a dynamic form, where you can add and remove controls at run time.
You can also nest form arrays to create more complex forms.

Generate a ProfileEditor component:

```
ng g c ProfileEditor
```

Create a FormGroup instance

Create a property in the component class (**profile-editor.component.ts**) named profileForm and set the property to a new form group instance.
To initialize the form group, provide the constructor with an object of named keys mapped to their control.

```
import { Component } from '@angular/core';
import { FormGroup, FormControl } from '@angular/forms';

@Component({
  selector: 'app-profile-editor',
  templateUrl: './profile-editor.component.html',
  styleUrls: ['./profile-editor.component.css']
})
export class ProfileEditorComponent {
  profileForm = new FormGroup({
    firstName: new FormControl(''),
    lastName: new FormControl(''),
  });
}
```

Associate the FormGroup model and view

A form group tracks the status and changes for each of its controls, so if one of the controls changes, the parent control also emits a new status or value change.
On **profile-editor-component.html**:

```
<form [formGroup]="profileForm">

  <label>
    First Name:
    <input type="text" formControlName="firstName">
  </label>

  <label>
    Last Name:
    <input type="text" formControlName="lastName">
  </label>

</form>
```

The formControlName input provided by the FormControlName directive binds each individual input to the form control defined in FormGroup.

Save form data

The **FormGroup** directive listens for the **submit event** emitted by the form element and emits an **ngSubmit** event that you can bind to a callback function. On **profile-editor.component.html**:

```
<form [formGroup]="profileForm" (ngSubmit)="onSubmit()">
```

The onSubmit() method in the ProfileEditor (**profile-editor.component.ts**) captures the current value of profileForm. Use **EventEmitter** to keep the form encapsulated and **to provide the form value outside the component**. The following example uses console.warn to log a message to the browser console.

```
onSubmit() {
  // TODO: Use EventEmitter with form value
  console.warn(this.profileForm.value);
}
```

Note: You aren't performing any validation yet, so the button is always enabled.

Basic form validation is covered in the Validating form input section.

Use a button element to add a button to the bottom of the form on (**profile-editor.component.html**) to trigger the form submission.

```
<button type="submit" [disabled]="!profileForm.valid">Submit</button>
```

To display the ProfileEditor component that contains the form, add it to a **app.component.html**.

```
<app-profile-editor></app-profile-editor>
```

Creating nested form groups

Using a nested form group instance allows you to break large forms groups into smaller, more manageable ones.

To make more complex forms, use the following steps.

1. Create a nested group.
2. Group the nested form in the template.

Change your **profile-editor.component.ts** to the following:

```
import { Component } from '@angular/core';
import { FormGroup, FormControl } from '@angular/forms';

@Component({
  selector: 'app-profile-editor',
  templateUrl: './profile-editor.component.html',
  styleUrls: ['./profile-editor.component.css']
})
export class ProfileEditorComponent {
  profileForm = new FormGroup({
    firstName: new FormControl(''),
    lastName: new FormControl(''),
    address: new FormGroup({
      street: new FormControl(''),
      city: new FormControl(''),
      state: new FormControl(''),
      zip: new FormControl('')
    })
  });
}
```

Group the nested form in the template

After you update the model in the component class, update the template to connect the form group instance and its input elements.

On **profile-editor.component.html**:

```
<div formGroupName="address">
  <h3>Address</h3>

  <label>
    Street:
    <input type="text" formControlName="street">
  </label>

  <label>
    City:
    <input type="text" formControlName="city">
  </label>

  <label>
    State:
    <input type="text" formControlName="state">
  </label>

  <label>
    Zip Code:
    <input type="text" formControlName="zip">
  </label>
</div>
```

Updating parts of the data model

There are two ways to update the model value:

Use the **setValue()** method to set a new value for an individual control. The setValue() method strictly adheres to the structure of the form group and replaces the entire value for the control.

Use the **patchValue()** method to replace any properties defined in the object that have changed in the form model.

Add a new method on **profile-editor.component.ts**:

```
updateProfile() {
  this.profileForm.patchValue({
    firstName: 'Nancy',
    address: {
      street: '123 Drew Street'
    }
  });
}
```

Simulate an update by adding a button to **profile-editor.component.html** to update the user profile on demand.

```
<p>
  <button (click)="updateProfile()">Update Profile</button>
</p>
```

Using the FormBuilder service to generate controls

Creating form control instances manually can become repetitive when dealing with multiple forms. The FormBuilder service provides convenient methods for generating controls.

The FormBuilder service has three methods: control(), group(), and array(). These are factory methods for generating instances in your component classes including form controls, form groups, and form arrays.

Use the group method to create the profileForm controls.

Refactor your **profile-editor.component.ts**:

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

@Component({
  selector: 'app-profile-editor',
  templateUrl:
'./profile-editor.component.html',
  styleUrls:
['./profile-editor.component.css']
})
export class ProfileEditorComponent {
  profileForm = this.fb.group({
    firstName: [''],
    lastName: [''],
    address: this.fb.group({
      street: [''],
      city: [''],
      state: [''],
      zip: ['']
    }),
  });

  constructor(private fb: FormBuilder) { }
}
```

Validating form input

Read more here

Form validation is used to ensure that user input is complete and correct.

HTML5 has a set of built-in attributes that you can use for native validation, including required, minlength, and maxlength.

You can take advantage of these optional attributes on your form input elements. Add the required attribute to the firstName input element.

```
<input type="text" formControlName="firstName"
required>
```

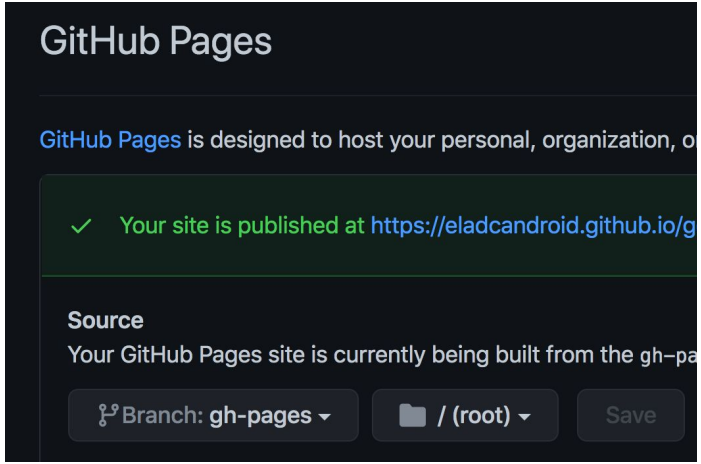
Access the current status of the form group instance through its status property.

```
<p>
  Form Status: {{ profileForm.status }}
</p>
```

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

@Component({
  selector: 'app-profile-editor',
  templateUrl:
'./profile-editor.component.html',
  styleUrls:
['./profile-editor.component.css']
})
export class ProfileEditorComponent {
  profileForm = this.fb.group({
    firstName: ['', Validators.required],
    lastName: [''],
    address: this.fb.group({
      street: [''],
      city: [''],
      state: [''],
      zip: ['']
    }),
  });

  constructor(private fb: FormBuilder) { }
}
```

Angular Deployment on Github Pages

1. Create a new repository on GitHub.
   Add all your files, commit and push.
2. Run
   ng add angular-cli-ghpages
   On your root project directory.
3. Run
   ng deploy --base-href=/gocode-shop/
   while **"gocode-shop"** is your repositry name on GitHub.
4. Go to your repository settings and choose gh-pages and root folder, then hit **Save**.
5. Change something on your master branch, **add**, **commit** and **push**.
6. Run deploy again like described on stage 3.
7. Check your repository on
   https://<username>.github.io/gocode-shop
   (Change <username> with your GitHub user.



https://github.com/angular-schule/angular-cli-ghpages