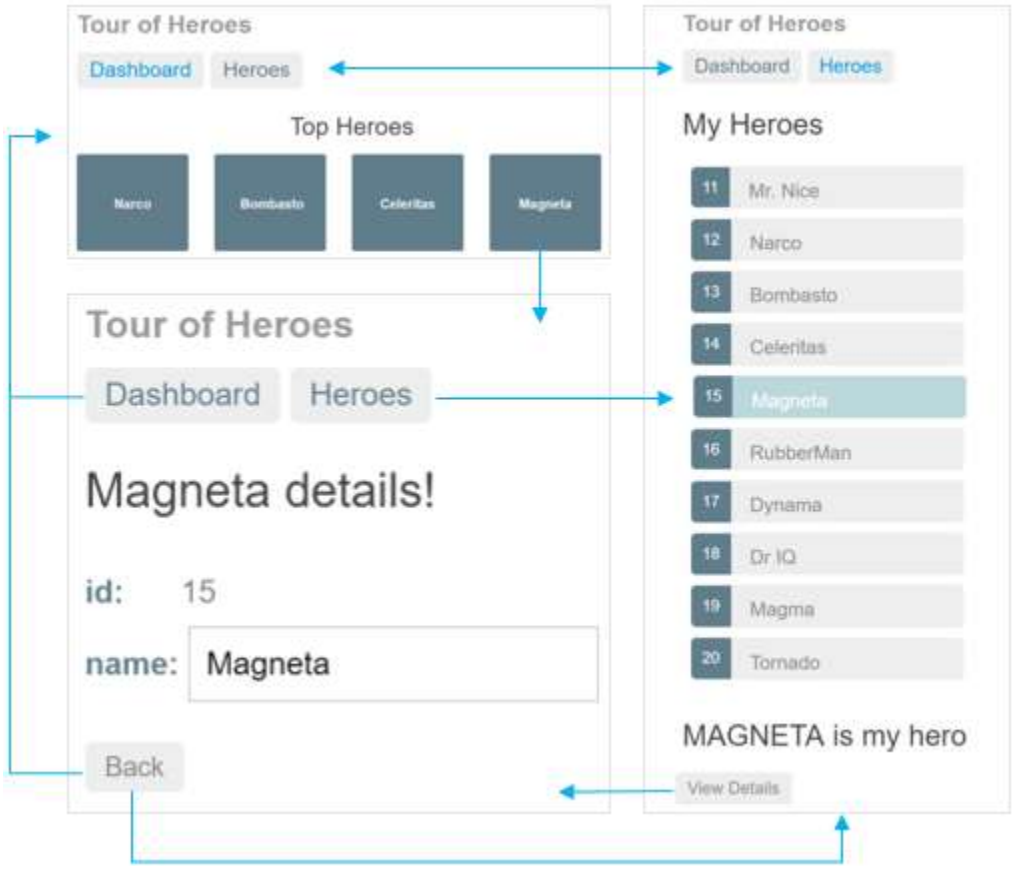
Routing

There are new requirements for the Tour of Heroes app:

- Add a Dashboard view.
- Add the ability to navigate between the Heroes and Dashboard views.
- When users click a hero name in either view, navigate to a detail view of the selected hero.
- When users click a deep link in an email, open the detail view for a particular hero.

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



Add the AppRoutingModule

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

By convention, the module class name is AppRoutingModule and it belongs in the app-routing.module.ts in the src/app folder.

Use the CLI to generate it.

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

--flat puts the file in src/app instead of its own folder.

--module=app tells the CLI to register it in the imports array of the AppModule.

The generated file looks like this:

src/app/app-routing.module.ts (generated)

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

@NgModule({
  imports: [
    CommonModule
  ],
  declarations: []
})
export class AppRoutingModule { }
```

Replace it with the following:

src/app/app-routing.module.ts (updated)

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

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

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

First, AppRoutingModule imports RouterModule and Routes so the app can have routing functionality. The next import, HeroesComponent, will give the Router somewhere to go once you configure the routes.

Notice that the CommonModule references and declarations array are unnecessary, so are no longer part of AppRoutingModule. The following sections explain the rest of the AppRoutingModule in more detail.

Routes

The next part of the file is where you configure your routes. Routes tell the Router which view to display when a user clicks a link or pastes a URL into the browser address bar.

Since AppRoutingModule already imports HeroesComponent, you can use it in the routes array:

src/app/app-routing.module.ts

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

A typical Angular Route has two properties:

- path: a string that matches the URL in the browser address bar.
- component: the component that the router should create when navigating to this route.

This tells the router to match that URL to path: 'heroes' and display the HeroesComponent when the URL is something like localhost:4200/heroes.

RouterModule.forRoot()

The @NgModule metadata initializes the router and starts it listening for browser location changes.

The following line adds the RouterModule to the AppRoutingModule imports array and configures it with the routes in one step by calling RouterModule.forRoot():

src/app/app-routing.module.ts

```
imports: [ RouterModule.forRoot(routes) ],
```

The method is called forRoot() because you configure the router at the application's root level. The forRoot() method supplies the service providers and directives needed for routing, and performs the initial navigation based on the current browser URL.

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

src/app/app-routing.module.ts (exports array)

```
exports: [ RouterModule ]
```

Add RouterOutlet

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

src/app/app.component.html (router-outlet)

```
<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.

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

The RouterOutlet is one of the router directives that became available to the AppComponent because AppModule imports AppRoutingModule which exported RouterModule.

Try it

You should still be running with this CLI command.

ng serve

The browser should refresh and display the app title but not the list of heroes.

Look at the browser's address bar. The URL ends in /. The route path to HeroesComponent is /heroes.

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

Add a navigation link (routerLink)

Ideally, users should be able to click a link to navigate rather than pasting a route URL into the address bar.

Add a <nav> element and, within that, an anchor element that, when clicked, triggers navigation to the HeroesComponent. The revised AppComponent template looks like this:

src/app/app.component.html (heroes RouterLink)

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

A routerLink attribute is set to "/heroes", the string that the router matches to the route to HeroesComponent. The routerLink is the selector for the RouterLink directive that turns user clicks into router navigations. It's another of the public directives in the RouterModule.

The browser refreshes and displays the app title and heroes link, but not the heroes list.

Click the link. The address bar updates to /heroes and the list of heroes appears.

Make this and future navigation links look better by adding private CSS styles to app.component.css as listed in the final code review below.

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 generate component dashboard
```

The CLI generates the files for the DashboardComponent and declares it in AppModule.

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

src/app/dashboard/dashboard.component.html

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

src/app/dashboard/dashboard.component.ts

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

@Component({
  selector: 'app-dashboard',
  templateUrl: './dashboard.component.html',
  styleUrls: [ './dashboard.component.css' ]
})
export class DashboardComponent implements OnInit {
  heroes: Hero[] = [];

  constructor(private heroService: HeroService) { }

  ngOnInit() {
    this.getHeroes();
  }

  getHeroes(): void {
    this.heroService.getHeroes()
      .subscribe(heroes => this.heroes = heroes.slice(1, 5));
  }
}
```

src/app/dashboard/dashboard.component.css

```
/* DashboardComponent's private CSS styles */
[class*='col-'] {
  float: left;
  padding-right: 20px;
  padding-bottom: 20px;
}
[class*='col-']:last-of-type {
  padding-right: 0;
}
a {
  text-decoration: none;
}
*, *:after, *:before {
  -webkit-box-sizing: border-box;
  -moz-box-sizing: border-box;
  box-sizing: border-box;
}
h3 {
  text-align: center;
  margin-bottom: 0;
}
h4 {
  position: relative;
}
.grid {
  margin: 0;
}
.col-1-4 {
  width: 25%;
```

```
}
.module {
  padding: 20px;
  text-align: center;
  color: #eee;
  max-height: 120px;
  min-width: 120px;
  background-color: #3f525c;
  border-radius: 2px;
}
.module:hover {
  background-color: #eee;
  cursor: pointer;
  color: #607d8b;
}
.grid-pad {
  padding: 10px 0;
}
.grid-pad > [class*='col-']:last-of-type {
  padding-right: 20px;
}
@media (max-width: 600px) {
  .module {
    font-size: 10px;
    max-height: 75px; }
}
@media (max-width: 1024px) {
  .grid {
    margin: 0;
  }
  .module {
    min-width: 60px;
  }
}
```

The template presents a grid of hero name links.

- The *ngFor repeater creates as many links as are in the component's heroes array.
- The links are styled as colored blocks by the dashboard.component.css.
- The links don't go anywhere yet but they will shortly.

The class is similar to the HeroesComponent class.

- It defines a heroes array property.
- The constructor expects Angular to inject the HeroService into a private heroService property.
- The ngOnInit() lifecycle hook calls getHeroes().

This getHeroes() returns the sliced list of heroes at positions 1 and 5, returning only four of the Top Heroes (2nd, 3rd, 4th, and 5th).

src/app/dashboard/dashboard.component.ts

```
getHeroes(): void {
  this.heroService.getHeroes()
    .subscribe(heroes => this.heroes = heroes.slice(1, 5));
}
```

Add the dashboard route

To navigate to the dashboard, the router needs an appropriate route.

Import the DashboardComponent in the AppRoutingModule.

src/app/app-routing.module.ts (import DashboardComponent)

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

Add a route to the AppRoutingModule.routes array that matches a path to the DashboardComponent.

src/app/app-routing.module.ts

```
{ 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 AppRoutingModule.Routes array.

src/app/app-routing.module.ts

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

This route redirects a URL that fully matches the empty path to the route whose path is '/dashboard'.

After the browser refreshes, the router loads the DashboardComponent and the browser address bar shows the /dashboard URL.

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.

src/app/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>
```

After the browser refreshes you can navigate freely between the two views by clicking the links.

Navigating to hero details

The HeroDetailsComponent displays details of a selected hero. At the moment the HeroDetailsComponent is only visible at the bottom of the HeroesComponent

The user should be able to get to these details in three ways.

1. By clicking a hero in the dashboard.
2. By clicking a hero in the heroes list.
3. By pasting a "deep link" URL into the browser address bar that identifies the hero to display.

In this section, you'll enable navigation to the HeroDetailsComponent and liberate it from the HeroesComponent.

Delete hero details from HeroesComponent

When the user clicks a hero item in the HeroesComponent, the app should navigate to the HeroDetailComponent, replacing the heroes list view with the hero detail view. The heroes list view should no longer show hero details as it does now.

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

Clicking a hero item now does nothing. You'll fix that shortly after you enable routing to the HeroDetailComponent.

Add a hero detail route

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 AppRoutingModule and import HeroDetailComponent.

src/app/app-routing.module.ts (import HeroDetailComponent)

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

Then add a parameterized route to the AppRoutingModule.routes array that matches the path pattern to the hero detail view.

src/app/app-routing.module.ts

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

The colon (:) in the path indicates that :id is a placeholder for a specific hero id.

At this point, all application routes are in place.

src/app/app-routing.module.ts (all routes)

```
const routes: Routes = [
  { path: '', redirectTo: '/dashboard', pathMatch: 'full' },
  { path: 'dashboard', component: DashboardComponent },
  { path: 'detail/:id', component: HeroDetailComponent },
  { path: 'heroes', component: HeroesComponent }
];
```

DashboardComponent hero links

The DashboardComponent hero links do nothing at the moment.

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

src/app/dashboard/dashboard.component.html (hero links)

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

You're using Angular interpolation binding within the *ngFor repeater to insert the current iteration's hero.id into each routerLink.

HeroesComponent hero links

The hero items in the HeroesComponent are <li> elements whose click events are bound to the component's onSelect() method.

src/app/heroes/heroes.component.html (list with onSelect)

```
<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>
```

Strip the <li> back to just its *ngFor, wrap the badge and name in an anchor element (<a>), and add a routerLink attribute to the anchor that is the same as in the dashboard template.

src/app/heroes/heroes.component.html (list with links)

```
<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>
```

You'll have to fix the private stylesheet (heroes.component.css) to make the list look as it did before. Revised styles are in the final code review at the bottom of this guide.

Remove dead code

While the HeroesComponent class still works, the onSelect() method and selectedHero property are no longer used.

It's nice to tidy up and you'll be grateful to yourself later. Here's the class after pruning away the dead code.

src/app/heroes/heroes.component.ts (cleaned up)

```
export class HeroesComponent implements OnInit {
  heroes: Hero[];

  constructor(private heroService: HeroService) { }

  ngOnInit() {
    this.getHeroes();
  }

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

Routable HeroDetailComponent

Previously, the parent HeroesComponent set the HeroDetailComponent.hero property and the HeroDetailComponent displayed the hero.

HeroesComponent doesn't do that anymore. 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:

1. Get the route that created it
2. Extract the id from the route
3. Acquire the hero with that id from the server via the HeroService

Add the following imports:

src/app/hero-detail/hero-detail.component.ts

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

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

Inject the ActivatedRoute, HeroService, and Location services into the constructor, saving their values in private fields:

toh-pt5/src/app/hero-detail/hero-detail.component.ts

```
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.

The HeroService gets hero data from the remote server and this component will use it to get the hero-to-display.

The location is an Angular service for interacting with the browser. You'll use it later to navigate back to the view that navigated here.

Extract the id route parameter

In the ngOnInit() lifecycle hook call getHero() and define it as follows.

src/app/hero-detail/hero-detail.component.ts

```
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 HeroService and add the following getHero() method with the id after the getHeroes() method:

src/app/hero.service.ts (getHero)

```
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));
}
```

Note the backticks (`) that define a JavaScript template literal for embedding the id.

Like getHeroes(), getHero() has an asynchronous signature. It returns a mock hero as an Observable, using the RxJS of() function.

You'll be able to re-implement getHero() as a real Http request without having to change the HeroDetailComponent that calls it.

Try it

The browser refreshes and the app is working again. You can click a hero in the dashboard or in the heroes list and navigate to that hero's detail view.

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

By clicking the browser's back button, you can go back to the hero list or dashboard view, depending upon which sent you to the detail view.

It would be nice to have a button on the HeroDetail view that can do that.

Add a go back button to the bottom of the component template and bind it to the component's goBack() method.

src/app/hero-detail/hero-detail.component.html (back button)

```
<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.

src/app/hero-detail/hero-detail.component.ts (goBack)

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

Refresh the browser and start clicking. Users can navigate around the app, from the dashboard to hero details and back, from heroes list to the mini detail to the hero details and back to the heroes again.

Summary

- You added the Angular router to navigate among different components.
- You turned the AppComponent into a navigation shell with <a> links and a <router-outlet>.
- You configured the router in an AppRoutingModule
- You defined simple routes, a redirect route, and a parameterized route.
- You used the routerLink directive in anchor elements.

- You refactored a tightly-coupled master/detail view into a routed detail view.
- You used router link parameters to navigate to the detail view of a user-selected hero.
- You shared the HeroService among multiple components.