ROUTING

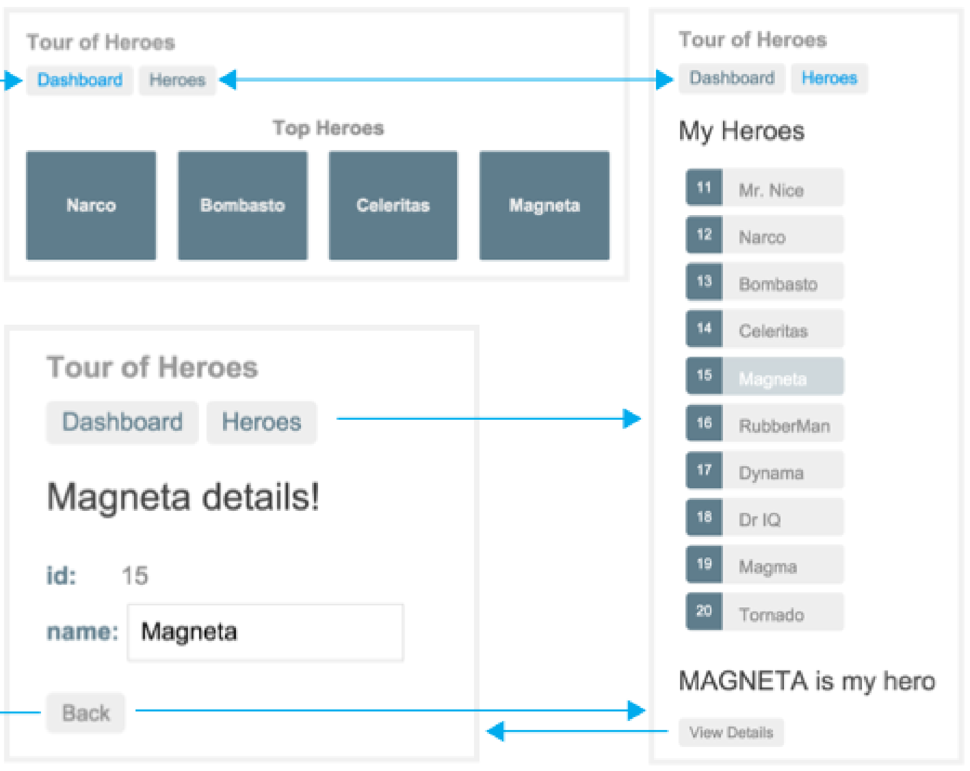
We add the Angular Component Router and learn to navigate among the views

Routing Around the App

We received new requirements for our Tour of Heroes application:

- add a *Dashboard* view.
- navigate between the *Heroes* and *Dashboard* views.
- clicking on a hero in either view navigates to a detail view of the selected hero.
- clicking a *deep link* in an email opens the detail view for a particular hero;

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



We'll add Angular's *Component Router* to our app to satisfy these requirements.

> The [Routing and Navigation]() chapter covers the router in more detail than we will in this tour.

[Run the live example]().

> To see the URL changes in the browser address bar, pop out the preview window by clicking the blue 'X' button in the upper right corner:

Where We Left Off

Before we continue with our Tour of Heroes, let's verify that we have the following structure after adding our hero service and hero detail component. If not, we'll need to go back and follow the previous chapters.

```
angular2-tour-of-heroes
  ├─ app
```

```
    ├─ app.component.ts
    ├─ hero.ts
    ├─ hero-detail.component.ts
    ├─ hero.service.ts
    ├─ main.ts
    └─ mock-heroes.ts
├─ node_modules ...
├─ typings ...
├─ index.html
├─ package.json
├─ tsconfig.json
└─ typings.json
```

Keep the app transpiling and running

Open a terminal/console window and enter the following command to start the TypeScript compiler, start the server, and watch for changes:

```
npm start
```

The application runs and updates automatically as we continue to build the Tour of Heroes.

Action plan

Here's our plan

- turn `AppComponent` into an application shell that only handles navigation.
- relocate the *Heroes* concerns within the current `AppComponent` to a separate `HeroesComponent`
- add routing
- create a new `DashboardComponent`
- tie the *Dashboard* into the navigation structure.

> *Routing* is another name for *navigation*. The *router* is the mechanism for navigating from view to view.

Splitting the *AppComponent*

Our current app loads `AppComponent` and immediately displays the list of heroes.

Our revised app should present a shell with a choice of views (*Dashboard* and *Heroes*) and then default to one of them.

The `AppComponent` should only handle navigation. Let's move the display of *Heroes* out of `AppComponent` and into its own `HeroesComponent`.

HeroesComponent

`AppComponent` is already dedicated to *Heroes*. Instead of moving anything out of `AppComponent`, we'll just rename it `HeroesComponent` and create a new `AppComponent` shell separately.

The steps are:

- rename `app.component.ts` file to `heroes.component.ts`.
- rename the `AppComponent` class to `HeroesComponent`.
- rename the selector `my-app` to `my-heroes`.

app/heroes.component.ts (renaming)

```
@Component({
  selector: 'my-heroes',
})
export class HeroesComponent implements OnInit {
}
```

Create *AppComponent*

The new `AppComponent` will be the application shell. It will have some navigation links at the top and a display area below for the pages we navigate to.

The initial steps are:

- create a new file named `app.component.ts`.
- define an `AppComponent` class.
- `export` it so we can reference it during bootstrapping in `main.ts`.
- expose an application `title` property.
- add the `@Component` metadata decorator above the class with a `my-app` selector.
- add a template with `<h1>` tags surrounding a binding to the `title` property.
- add the `<my-heroes>` tags to the template so we still see the heroes.

- add the `HeroesComponent` to the `directives` array so Angular recognizes the `<my-heroes>` tags.
- add the `HeroService` to the `providers` array because we'll need it in every other view.
- add the supporting `import` statements.

Our first draft looks like this:

app/app.component.ts (v1)

```
import { Component }       from 'angular2/core';
import { HeroService }     from './hero.service';
import { HeroesComponent } from './heroes.component';

@Component({
  selector: 'my-app',
  template: `
    <h1>{{title}}</h1>
    <my-heroes></my-heroes>
  `,
  directives: [HeroesComponent],
  providers: [
    HeroService
  ]
})
export class AppComponent {
  title = 'Tour of Heroes';
}
```

REMOVE *HEROSERVICE* FROM THE *HEROESCOMPONENT* PROVIDERS

Go back to the `HeroesComponent` and **remove the `HeroService`** from its `providers` array. We are *promoting* this service from the `HeroesComponent` to the `AppComponent`. We ***do not want two copies*** of this service at two different levels of our app.

The app still runs and still displays heroes. Our refactoring of `AppComponent` into a new `AppComponent` and a `HeroesComponent` worked! We have done no harm.

Add Routing

We're ready to take the next step. Instead of displaying heroes automatically, we'd like to show them *after* the user clicks a button. In other words, we'd like to navigate to the list of heroes.

We'll need the Angular *Component Router*.

Include the Router Library

Not all apps need routing which is why the Angular *Component Router* is in a separate, optional module library.

Our Tour of Heroes needs routing, so we load the library in the `index.html` in a script tag immediately *after* the angular script itself.

index.html (router)

```
<script src="node_modules/angular2/bundles/router.dev.js"></script>
```

While we're in `index.html`, we add `<base href="/">` at the top of the `<head>` section.

index.html (base href)

```
<head>
  <base href="/">
```

BASE HREF IS ESSENTIAL

See the *base href* section of the Router chapter to learn why this matters.

Make the router available.

The *Component Router* is a service. Like any service, we have to import it and make it available to the application by adding it to the `providers` array.

The Angular router is a combination of multiple services (`ROUTER_PROVIDERS`), multiple directives (`ROUTER_DIRECTIVES`), and a configuration decorator (`RouteConfig`). We'll import them all together:

app.component.ts (router imports)

```
import { RouteConfig, ROUTER_DIRECTIVES, ROUTER_PROVIDERS } from
'angular2/router';
```

Next we update the `directives` and `providers` metadata arrays to *include* the router assets.

app.component.ts (directives and providers)

```
directives: [ROUTER_DIRECTIVES],
providers: [
  ROUTER_PROVIDERS,
  HeroService
]
```

Notice that we also removed the `HeroesComponent` from the `directives` array. `AppComponent` no longer shows heroes; that will be the router's job. We'll soon remove `<my-heroes>` from the template too.

Add and configure the router

The `AppComponent` doesn't have a router yet. We'll use the `@RouteConfig` decorator to simultaneously (a) assign a router to the component and (b) configure that router with *routes*.

Routes tell the router which views to display when a user clicks a link or pastes a URL into the browser address bar.

Let's define our first route, a route to the `HeroesComponent`.

app.component.ts (RouteConfig for heroes)

```
@RouteConfig([
  {
    path: '/heroes',
    name: 'Heroes',
    component: HeroesComponent
  }
])
```

`@RouteConfig` takes an array of *route definitions*. We have only one route definition at the moment but rest assured, we'll add more.

This *route definition* has three parts:

- **path**: the router matches this route's path to the URL in the browser address bar (`/heroes`).
- **name**: the official name of the route; it *must* begin with a capital letter to avoid confusion with the *path* (`Heroes`).
- **component**: the component that the router should create when navigating to this route (`HeroesComponent`).

> Learn more about defining routes with @RouteConfig in the Routing chapter.

Router Outlet

If we paste the path, `/heroes`, into the browser address bar, the router should match it to the

`'Heroes'` route and display the `HeroesComponent`. But where?

We have to ***tell it where*** by adding `<router-outlet>` marker tags to the bottom of the template. `RouterOutlet` is one of the `ROUTER_DIRECTIVES`. The router displays each component immediately below the `<router-outlet>` as we navigate through the application.

Router Links

We don't really expect users to paste a route URL into the address bar. We add an anchor tag to the template which, when clicked, triggers navigation to the `HeroesComponent`.

The revised template looks like this:

app.component.ts (template for Heroes)

```
template: `
  <h1>{{title}}</h1>
  <a [routerLink]="['Heroes']">Heroes</a>
  <router-outlet></router-outlet>
`,
```

Notice the `[routerLink]` binding in the anchor tag. We bind the `RouterLink` directive (another of the `ROUTER_DIRECTIVES`) to an array that tells the router where to navigate when the user clicks the link.

We define a *routing instruction* with a *link parameters array*. The array only has one element in our little sample, the quoted ***name* of the route** to follow. Looking back at the route configuration, we confirm that `'Heroes'` is the name of the route to the `HeroesComponent`.

> Learn about the *link parameters array* in the Routing chapter.

Refresh the browser. We see only the app title. We don't see the heroes list.

> The browser's address bar shows `/`. The route path to `HeroesComponent` is `/heroes`, not `/`. We don't have a route that matches the path `/`, so there is nothing to show. That's something we'll want to fix.

We click the "Heroes" navigation link, the browser bar updates to `/heroes`, and now we see the list of heroes. We are navigating at last!

At this stage, our `AppComponent` looks like this.

app/app.component.ts (v2)

```
import { Component } from 'angular2/core';
import { RouteConfig, ROUTER_DIRECTIVES, ROUTER_PROVIDERS } from
'angular2/router';

import { HeroService }     from './hero.service';
import { HeroesComponent } from './heroes.component';

@Component({
  selector: 'my-app',
  template: `
    <h1>{{title}}</h1>
    <a [routerLink]="['Heroes']">Heroes</a>
    <router-outlet></router-outlet>
  `,
  directives: [ROUTER_DIRECTIVES],
  providers: [
    ROUTER_PROVIDERS,
    HeroService
  ]
})
@RouteConfig([
  {
    path: '/heroes',
    name: 'Heroes',
    component: HeroesComponent
  }
])
export class AppComponent {
  title = 'Tour of Heroes';
}
```

The *AppComponent* is now attached to a router and displaying routed views. For this reason and to distinguish it from other kinds of components, we call this type of component a *Router Component*.

Add a *Dashboard*

Routing only makes sense when we have multiple views. We need another view.

Create a placeholder `DashboardComponent` that gives us something to navigate to and from.

app/dashboard.component.ts (v1)

```
import { Component } from 'angular2/core';

@Component({
  selector: 'my-dashboard',
  template: '<h3>My Dashboard</h3>'
})
```

```
export class DashboardComponent { }
```

We'll come back and make it more useful later.

Configure the dashboard route

Go back to `app.component.ts` and teach it to navigate to the dashboard.

Import the `DashboardComponent` so we can reference it in the dashboard route definition.

Add the following `'Dashboard'` route definition to the `@RouteConfig` array of definitions.

app.component.ts (Dashboard Route)

```
{
  path: '/dashboard',
  name: 'Dashboard',
  component: DashboardComponent,
  useAsDefault: true
},
```

useAsDefault

We want the app to show the dashboard when it starts and we want to see a nice URL in the browser address bar that says `/dashboard`. Remember that the browser launches with `/` in the address bar. We don't have a route for that path and we'd rather not create one.

Fortunately we can add the `useAsDefault: true` property to the *route definition* and the router will display the dashboard when the browser URL doesn't match an existing route.

Finally, add a dashboard navigation link to the template, just above the *Heroes* link.

app.component.ts (template)

```
template: `
  <h1>{{title}}</h1>
  <nav>
    <a [routerLink]="['Dashboard']">Dashboard</a>
    <a [routerLink]="['Heroes']">Heroes</a>
  </nav>
  <router-outlet></router-outlet>
`,
```

We nestled the two links within `<nav>` tags. They don't do anything yet but they'll be convenient when we style the links a little later in the chapter.

Refresh the browser. The app displays the dashboard and we can navigate between the dashboard and the heroes.

Dashboard Top Heroes

Let’s spice up the dashboard by displaying the top four heroes at a glance.

Replace the `template` metadata with a `templateUrl` property that points to a new template file.

app/dashboard.component.ts (templateUrl)

```
templateUrl: 'app/dashboard.component.html',
```

We specify the path *all the way back to the application root*. Angular doesn't support module-relative paths.

Create that file with these contents:

dashboard.component.html

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

We use `*ngFor` once again to iterate over a list of heroes and display their names. We added extra `<div>` elements to help with styling later in this chapter.

There's a `(click)` binding to a `gotoDetail` method we haven't written yet and we're displaying a list of heroes that we don't have. We have work to do, starting with those heroes.

Share the *HeroService*

We'd like to re-use the `HeroService` to populate the component's `heroes` array.

Recall earlier in the chapter that we removed the `HeroService` from the `providers` array of the `HeroesComponent` and added it to the `providers` array of the top level `AppComponent`.

That move created a singleton `HeroService` instance, available to *all* components of the application. We'll inject and use it here in the `DashboardComponent` .

Get heroes

Open the `dashboard.component.ts` and add the requisite `import` statements.

app/dashboard.component.ts (imports)

```
import { Component, OnInit } from 'angular2/core';

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

We need `OnInit` interface because we'll initialize the heroes in the `ngOnInit` method as we've done before. We need the `Hero` and `HeroService` symbols in order to reference those types.

Now implement the `DashboardComponent` class like this:

app/dashboard.component.ts (class)

```
export class DashboardComponent implements OnInit {

  heroes: Hero[] = [];

  constructor(private _heroService: HeroService) { }

  ngOnInit() {
    this._heroService.getHeroes()
      .then(heroes => this.heroes = heroes.slice(1,5));
  }

  gotoDetail(){ /* not implemented yet */}
}
```

We saw this kind of logic before in the `HeroesComponent`.

- create a `heroes` array property
- inject the `HeroService` in the constructor and hold it in a private `_heroService` field.
- call the service to get heroes inside the Angular `ngOnInit` lifecycle hook.

The noteworthy differences: we cherry-pick four heroes (2nd, 3rd, 4th, and 5th) with *slice* and stub the `gotoDetail` method until we're ready to implement it.

Refresh the browser and see four heroes in the new dashboard.

Navigate to Hero Details

Although we display the details of a selected hero at the bottom of the `HeroesComponent`, we don't yet *navigate* to the `HeroDetailComponent` in the three ways specified in our requirements:

1. from the *Dashboard* to a selected hero.
2. from the *Heroes* list to a selected hero.
3. from a "deep link" URL pasted into the browser address bar.

Adding a `'HeroDetail'` route seem an obvious place to start.

Routing to a hero detail

We'll add a route to the `HeroDetailComponent` in the `AppComponent` where our other routes are configured.

The new route is a bit unusual in that we must tell the `HeroDetailComponent` *which hero to show*. We didn't have to tell the `HeroesComponent` or the `DashboardComponent` anything.

At the moment the parent `HeroesComponent` sets the component's `hero` property to a hero object with a binding like this.

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

That clearly won't work in any of our routing scenarios. Certainly not the last one; we can't embed an entire hero object in the URL! Nor would we want to.

Parameterized route

We *can* add the hero's `id` to the URL. When routing to the hero whose `id` is 11, we could expect to see an URL such as this:

```
/detail/11
```

The `/detail/` part of that URL is constant. The trailing numeric `id` part changes from hero to hero. We need to represent that variable part of the route with a *parameter* (or *token*) that stands for the hero's `id`.

Configure a Route with a Parameter

Here's the *route definition* we'll use.

app/app.component.ts (Route to HeroDetailComponent)

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

The colon (:) in the path indicates that `:id` is a placeholder to be filled with a specific hero `id` when navigating to the `HeroDetailComponent`.

> Of course we have to import the `HeroDetailComponent` before we create this route:
>
> ```
> import { HeroDetailComponent } from './hero-detail.component';
> ```

We're finished with the `AppComponent`.

We won't add a `'Hero Detail'` link to the template because users don't click a navigation *link* to view a particular hero. They click a *hero* whether that hero is displayed on the dashboard or in the heroes list.

We'll get to those *hero* clicks later in the chapter. There's no point in working on them until the `HeroDetailComponent` is ready to be navigated *to*.

That will require an `HeroDetailComponent` overhaul.

Revise the *HeroDetailComponent*

Before we rewrite the `HeroDetailComponent`, let's remember what it looks like now:

app/hero-detail.component.ts (current)

```
import {Component} from 'angular2/core';
import {Hero} from './hero';

```

```
@Component({
  selector: 'my-hero-detail',
  template: `
    <div *ngIf="hero">
      <h2>{{hero.name}} details</h2>
      <div>
        <label>id: </label>{{hero.id}}
      </div>
      <div>
        <label>name: </label>
        <input [(ngModel)]="hero.name" placeholder="name"/>
      </div>
    </div>
  `,
  inputs: ['hero']
})
export class HeroDetailComponent {
  hero: Hero;
}
```

The template won't change. We'll display a hero the same way. The big changes are driven by how we get the hero.

We will no longer receive the hero in a parent component property binding. The new `HeroDetailComponent` should take the `id` parameter from the router's `RouteParams` service and use the `HeroService` to fetch the hero with that `id` from storage.

We need an import statement to reference the `RouteParams`.

```
import {RouteParams} from 'angular2/router';
```

We import the `HeroService` so we can fetch a hero`.

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

We import the `OnInit` interface because we'll call the `HeroService` inside the `ngOnInit` component lifecycle hook.

```
import { Component, OnInit } from 'angular2/core';
```

We inject the both the `RouteParams` service and the `HeroService` into the constructor as we've done before, making private variables for both:

app/hero-detail.component.ts (constructor)

```
constructor(
  private _heroService: HeroService,
  private _routeParams: RouteParams) {
}
```

Inside the `ngOnInit` lifecycle hook, extract the `id` parameter value from the `RouteParams` service and use the `HeroService` to fetch the hero with that `id`.

app/hero-detail.component.ts (ngOnInit)

```
  ngOnInit() {
    let id = +this._routeParams.get('id');
    this._heroService.getHero(id)
      .then(hero => this.hero = hero);
  }
```

Notice how we extract the `id` by calling the `RouteParams.get` method.

```
let id = +this._routeParams.get('id');
```

The hero `id` is a number. Route parameters are *always strings*. So we convert the route parameter value to a number with the JavaScript (+) operator.

Add *HeroService.getHero*

The problem with this bit of code is that `HeroService` doesn't have a `getHero` method! We better fix that quickly before someone notices that we broke the app.

Open `HeroService` and add the `getHero` method. It's trivial given that we're still faking data access:

app/hero.service.ts (getHero)

```
getHero(id: number) {
  return Promise.resolve(HEROES).then(
    heroes => heroes.filter(hero => hero.id === id)[0]
  );
}
```

Return to the `HeroDetailComponent` to clean up loose ends.

Find our way back

We can navigate *to* the `HeroDetailComponent` in several ways. How do we navigate somewhere else when we're done?

The user could click one of the two links in the `AppComponent`. Or click the browser's back button. We'll add a third option, a `goBack` method that navigates backward one step in the browser's history stack

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

```
goBack() {
  window.history.back();
}
```

> Going back too far could take us out of the application. That's acceptable in a demo. We'd guard against it in a real application, perhaps with the *routerCanDeactivate* hook.

Then we wire this method with an event binding to a *Back* button that we add to the bottom of the component template.

```
<button (click)="goBack()">Back</button>
```

Modifing the template to add this button spurs us to take one more incremental improvement and migrate the template to its own file called `hero-detail.component.html`

app/hero-detail.component.html

```
<div *ngIf="hero">
  <h2>{{hero.name}} details!</h2>
  <div>
    <label>id: </label>{{hero.id}}</div>
  <div>
    <label>name: </label>
    <input [(ngModel)]="hero.name" placeholder="name" />
  </div>
  <button (click)="goBack()">Back</button>
</div>
```

We update the component metadata with a `templateUrl` pointing to the template file that we just created.

app/hero-detail.component.ts (templateUrl)

```
templateUrl: 'app/hero-detail.component.html',
```

Here's the (nearly) finished `HeroDetailComponent`:

app/hero-detail.component.ts (latest)

```
import { Component, OnInit } from 'angular2/core';
import {RouteParams} from 'angular2/router';

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

@Component({
  selector: 'my-hero-detail',
  templateUrl: 'app/hero-detail.component.html',
})
export class HeroDetailComponent implements OnInit {
  hero: Hero;

  constructor(
    private _heroService: HeroService,
    private _routeParams: RouteParams) {
  }

  ngOnInit() {
    let id = +this._routeParams.get('id');
    this._heroService.getHero(id)
      .then(hero => this.hero = hero);
  }

  goBack() {
    window.history.back();
  }
}
```

Select a *Dashboard* Hero

When a user selects a hero in the dashboard, the app should navigate to the `HeroDetailComponent` to view and edit the selected hero..

In the dashboard template we bound each hero's click event to the `gotoDetail` method, passing along the selected `hero` entity.

app/dashboard.component.html (click binding)

```
<div *ngFor="#hero of heroes" (click)="gotoDetail(hero)" class="col-1-4" >
```

We stubbed the `gotoDetail` method when we rewrote the `DashboardComponent`. Now we give it a real implementation.

app/dashboard.component.ts (gotoDetail)

```
gotoDetail(hero: Hero) {
  let link = ['HeroDetail', { id: hero.id }];
  this._router.navigate(link);
}
```

The `gotoDetail` method navigates in two steps:

1. set a route *link parameters array*
2. pass the array to the router's navigate method.

We wrote *link parameters arrays* in the `AppComponent` for the navigation links. Those arrays had only one element, the name of the destination route.

This array has two elements, the ***name*** of the destination route and a ***route parameter object*** with an `id` field set to the value of the selected hero's `id`.

The two array items align with the ***name*** and ***:id*** token in the parameterized `HeroDetail` route configuration we added to `AppComponent` earlier in the chapter.

app/app.component.ts (hero detail route)

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

The `DashboardComponent` doesn't have the router yet. We obtain it in the usual way: `import` the `router` reference and inject it in the constructor (along with the `HeroService`):

app/dashboard.component.ts (excerpts)

```
import { Router } from 'angular2/router';
```

```
constructor(
  private _router: Router,
  private _heroService: HeroService) {
}
```

Refresh the browser and select a hero from the dashboard; the app should navigate directly to that hero's details.

Select a Hero in the *HeroesComponent*

We'll do something similar in the `HeroesComponent`.

That component's current template exhibits a "master/detail" style with the list of heroes at the top and details of the selected hero below.

app/heroes.component.ts (current template)

```
template:`
  <h1>{{title}}</h1>
  <h2>My Heroes</h2>
  <ul class="heroes">
    <li *ngFor="#hero of heroes"
      [class.selected]="hero === selectedHero"
      (click)="onSelect(hero)">
      <span class="badge">{{hero.id}}</span> {{hero.name}}
    </li>
  </ul>
  <my-hero-detail [hero]="selectedHero"></my-hero-detail>
`,
```

Delete the last line of the template with the `<my-hero-detail>` tags.

We'll no longer show the full `HeroDetailComponent` here. We're going to display the hero detail on its own page and route to it as we did in the dashboard.

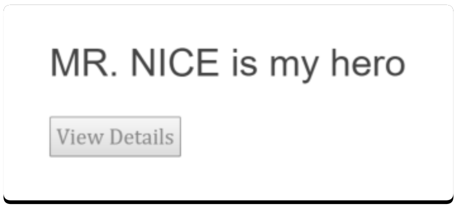
But we'll throw in a small twist for variety. When the user selects a hero from the list, we *won't* go to the detail page. We'll show a *mini-detail* on *this* page instead and make the user click a button to navigate to the *full detail* page.

Add the *mini-detail*

Add the following HTML fragment at the bottom of the template where the `<my-hero-detail>` used to be:

```
<div *ngIf="selectedHero">
  <h2>
    {{selectedHero.name | uppercase}} is my hero
  </h2>
  <button (click)="gotoDetail()">View Details</button>
</div>
```

After clicking a hero, the user should see something like this below the hero list:


MR. NICE is my hero

Format with the *UpperCasePipe*

Notice that the hero's name is displayed in CAPITAL LETTERS. That's the effect of the `UpperCasePipe` that we slipped into the interpolation binding. Look for it right after the pipe operator, (|).

```
{{selectedHero.name | uppercase}} is my hero
```

Pipes are a good way to format strings, currency amounts, dates and other display data. Angular ships with several pipes and we can write our own.

> Learn about pipes in the Pipes chapter.

Move content out of the component file

We are not done. We still have to update the component class to support navigation to the `HeroDetailComponent` when the user clicks the *View Details* button.

This component file is really big. Most of it is either template or CSS styles. It's difficult to find the component logic amidst the noise of HTML and CSS.

Let's migrate the template and the styles to their own files before we make any more changes:

1. *Cut-and-paste* the template contents into a new `heroes.component.html` file.
2. *Cut-and-paste* the styles contents into a new `heroes.component.css` file.
3. *Set* the component metadata's `templateUrl` and `styleUrls` properties to refer to both files.

The revised component data looks like this:

app/heroes.component.ts (revised metadata)

```
@Component({
  selector: 'my-heroes',
  templateUrl: 'app/heroes.component.html',
  styleUrls:  ['app/heroes.component.css'],
  directives: [HeroDetailComponent]
})
```

Now we can see what's going on as we update the component class along the same lines as the dashboard:

1. Import the `router`
2. Inject the `router` in the constructor (along with the `HeroService`)
3. Implement the `gotoDetail` method by calling the `router.navigate` method with a two-part 'HeroDetail' *link parameters array*.

Here's the revised component class:

app/heroes.component.ts (class)

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

  constructor(
    private _router: Router,
    private _heroService: HeroService) { }

  getHeroes() {
    this._heroService.getHeroes().then(heroes => this.heroes = heroes);
  }

  ngOnInit() {
    this.getHeroes();
  }

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

  gotoDetail() {
    this._router.navigate(['HeroDetail', { id: this.selectedHero.id }]);
  }
}
```

Refresh the browser and start clicking. We can navigate around the app, from the dashboard to hero details and back, for heroes list to the mini-detail to the hero details and back to the heroes again. We can jump back and forth between the dashboard and the heroes.

We've met all of the navigational requirements that propelled this chapter.

Styling the App

The app is functional but pretty ugly. Our creative designer team provided some CSS files to make it look better.

A Dashboard with Style

The designers think we should display the dashboard heroes in a row of rectangles. They've given us ~60 lines of CSS for this purpose including some simple media queries for responsive design.

If we paste these ~60 lines into the component `styles` metadata, they'll completely obscure the component logic. Let's not do that. It's easier to edit CSS in a separate `*.css` file anyway.

Add a `dashboard.component.css` file to the `app` folder and reference that file in the component metadata's `styleUrls` array property like this:

app/dashboard.component.ts (styleUrls)

```
styleUrls: ['app/dashboard.component.css']
```

> The `styleUrls` property is an array of style file names (with paths). We could list multiple style files from different locations if we needed them. As with `templateUrl`, we must specify the path *all the way back to the application root.*

Stylish Hero Details

The designers also gave us CSS styles specifically for the `HeroDetailComponent`.

Add a `hero-detail.component.css` to the `app` folder and refer to that file inside the `styleUrls` array as we did for `DashboardComponent`.

Here's the content for the aforementioned component CSS files.

```
label {
  display: inline-block;
  width: 3em;
  margin: .5em 0;
  color: #607D8B;
  font-weight: bold;
}
input {
  height: 2em;
  font-size: 1em;
  padding-left: .4em;
}
button {
  margin-top: 20px;
  font-family: Arial;
  background-color: #eee;
  border: none;
  padding: 5px 10px;
  border-radius: 4px;
  cursor: pointer; cursor: hand;
}
button:hover {
  background-color: #cfd8dc;
}
button:disabled {
  background-color: #eee;
  color: #ccc;
  cursor: auto;
}
```

Style the Navigation Links

The designers gave us CSS to make the navigation links in our `AppComponent` look more like selectable buttons. We cooperated by surrounding those links in `<nav>` tags.

Add a `app.component.css` file to the `app` folder with the following content.

app/app.component.css (Navigation Styles)

```
h1 {
  font-size: 1.2em;
  color: #999;
  margin-bottom: 0;
}
h2 {
  font-size: 2em;
  margin-top: 0;
  padding-top: 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: #607D8B;
}
nav a:hover {
  color: #039be5;
  background-color: #CFD8DC;
}
nav a.router-link-active {
  color: #039be5;
}
```

The *router-link-active* class

The Angular Router adds the `router-link-active` class to the HTML navigation element whose route matches the active route. All we have to do is define the style for it. Sweet!

Set the `AppComponent`'s `styleUrls` property to this CSS file.

app/app.component.ts (styleUrls)

```
styleUrls: ['app/app.component.css'],
```

Global application styles

When we add styles to a component, we're keeping everything a component needs — HTML, the CSS, the code — together in one convenient place. It's pretty easy to package it all up and re-use the component somewhere else.

We can also create styles at the *application level* outside of any component.

Our designers provided some basic styles to apply to elements across the entire app. Add the following to a new file named `styles.css` in the root folder.

styles.css (App Styles)

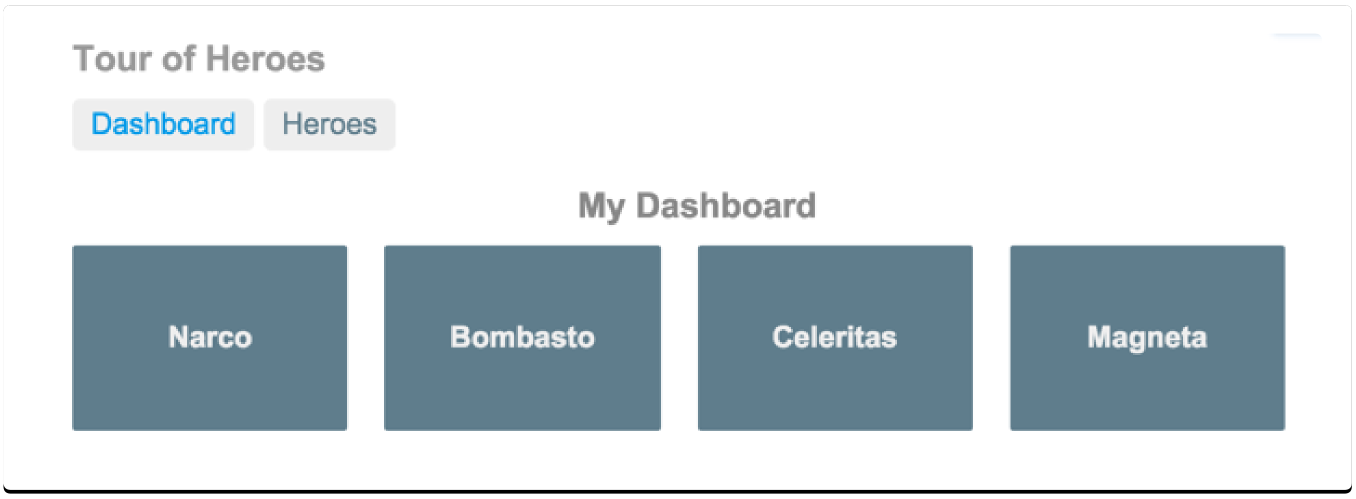
```
h2 {
  color: #444;
  font-family: Arial, Helvetica, sans-serif;
  font-weight: lighter;
}
body {
  margin: 2em;
}
body, input[text], button {
  color: #888;
  font-family: Cambria, Georgia;
}
button {
  font-family: Arial;
  background-color: #eee;
  border: none;
  padding: 5px 10px;
  border-radius: 4px;
  cursor: pointer;
  cursor: hand;
}
button:hover {
  background-color: #cfd8dc;
}
button:disabled {
  background-color: #eee;
  color: #aaa;
  cursor: auto;
}
/* everywhere else */
* {
  font-family: Arial, Helvetica, sans-serif;
}
```

Reference this stylesheet within the `index.html` in the traditional manner.

index.html (link ref)

```
<link rel="stylesheet" href="styles.css">
```

Look at the app now. Our dashboard, heroes, and navigation links are styling!



Application structure and code

Review the sample source code in the live example for this chapter. Verify that we have the following structure:

```
angular2-tour-of-heroes
  app
    app.component.ts
    app.component.css
    dashboard.component.css
    dashboard.component.html
    dashboard.component.ts
    hero.ts
    hero-detail.component.css
    hero-detail.component.html
    hero-detail.component.ts
    hero.service.ts
    heroes.component.css
    heroes.component.html
    heroes.component.ts
    main.ts
    mock-heroes.ts
  node_modules ...
  typings ...
  index.html
```

```
├─ package.json
├─ styles.css
├─ tsconfig.json
└─ typings.json
```

Recap

The Road Behind

We travelled a great distance in this chapter

- We added the Angular *Component Router* to navigate among different components.
- We learned how to create router links to represent navigation menu items
- We used router parameters to navigate to the details of user selected hero
- We shared the `HeroService` among multiple components
- We moved HTML and CSS out of the component file and into their own files.
- We added the `uppercase` pipe to format data

The Road Ahead

We have much of the foundation we need to build an application. We're still missing a key piece: remote data access.

In a forthcoming tutorial chapter, we'll replace our mock data with data retrieved from a server using http.