softcontext@gmail.com

Simple Website

http://onehungrymind.com/build-a-simple-website-with-angular-2/

Chapter 1

We are going to get acquainted with the brave new world of Angular 2

by building out a simple website

so we can see how the Angular 2 pieces come together.

Grab the code from the repository below and let's go!

Github

https://github.com/simpulton/angular2-website-routes

Dependencies

You must have **node** and npm installed. (via brew install node or NodeJS.org)

You must also have **typings** installed globally via "**npm i -g typings**".

Be sure that you have typings version 1.x.

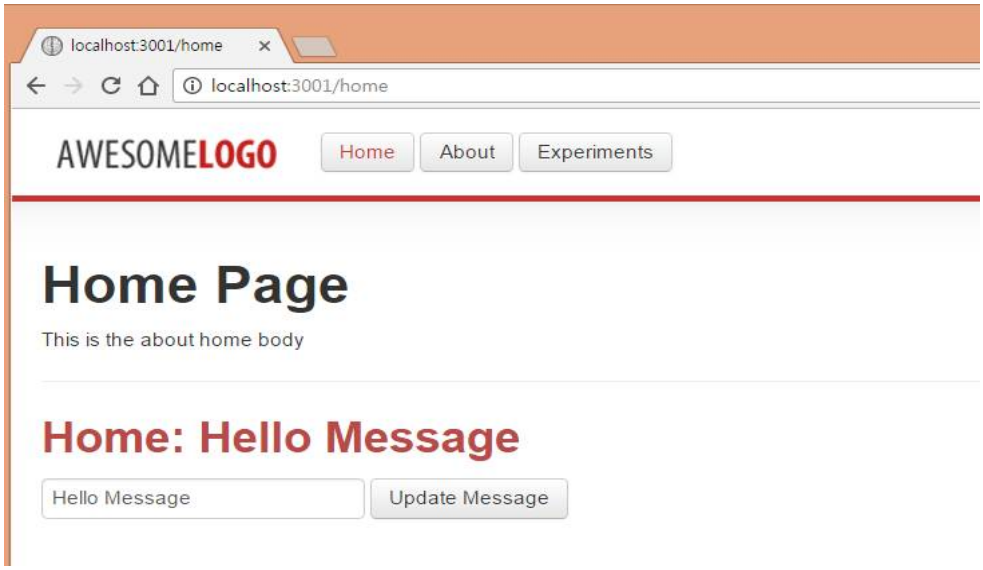
Getting Started

```
git clone https://github.com/simpulton/angular2-website-routes.git
cd angular2-website-routes
npm i
```

```
typings install
npm start
```

Check Site

Then navigate your browser to http://localhost:3001 and use the app.



Testing

The test setup includes

- webpack.test.config.js
- spec-bundle.js
- karma.conf.js

To run unit tests, execute **npm test** in your terminal.

Chapter 2

Compile and Serve

There are two main ways to compile and serve an Angular 2 application.

■ **webpack**

■ **systemjs**

I have personally fallen in love with webpack, and that is what we will use to build this app.

You can write your Angular 2 application in **ES5**, EcmaScript 2015(**ES6**) or **TypeScript**,
but the framework lends itself best to TypeScript.
Writing in TypeScript requires a bit more setup but the return on investment is tenfold
in terms of productivity and clarity in our code.

When you download the repository, the first thing you need to do is run npm i to **install the package dependencies**.

2.1. tsconfig.json

One of the package dependencies that we are going to install is typescript.

We need to tell our compiler how we want to compile our TypeScript files,

so we need to create a **tsconfig.json** file.

The file below reads pretty well, but I will call out the two most important properties

and those are "target": "ES5" and "module": "commonjs".

We are setting our ECMAScript version to ES5

and indicating that we want to generate our modules in the commonjs format.

tsconfig.json

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "sourceMap": true,
    "suppressImplicitAnyIndexErrors":true
  },
  "compileOnSave": false,
  "buildOnSave": false,
  "exclude": [
    "node_modules"
  ],
  "filesGlob": [
    "app/**/*.ts",
    "typings/index.d.ts"
  ],
  "atom": {
    "rewriteTsconfig": false
  }
}
```

2.2. package.json

We have defined how we want our TypeScript to be compiled, now we need to create a hook for the work to be done. In our packages.json file, we have defined a **task** that uses webpack-dev-server along with **webpack to bundle our TypeScript modules and other assets into a single javascript file** and then serve the app.

package.json

```
{
  "name": "fem-ng2-simple-app",
  "version": "0.0.1",
  "license": "SEE LICENSE IN LICENSE",
  "repository": {
    "type": "git",
    "url": "https://github.com/onehungrymind/fem-ng2-simple-app/"
  },
  "scripts": {
    "start": "webpack-dev-server --inline --colors --watch --display-error-
details --display-cached  --port 3001 --hot",
    "test": "karma start"
  },
  "dependencies": {
    "@angular/common": "2.0.0-rc.1",
    "@angular/compiler": "2.0.0-rc.1",
    "@angular/core": "2.0.0-rc.1",
    "@angular/platform-browser": "2.0.0-rc.1",
    "@angular/platform-browser-dynamic": "2.0.0-rc.1",
    "@angular/router": "2.0.0-rc.1",
    "es6-shim": "^0.35.0",
    "reflect-metadata": "0.1.3",
    "rxjs": "5.0.0-beta.6",
    "systemjs": "^0.19.27",
    "zone.js": "^0.6.12"
  },
  "devDependencies": {
    "awesome-typescript-loader": "^0.17.0",
    "core-js": "^2.4.0",
    "jasmine-core": "^2.4.1",
    "karma": "^0.13.22",
    "karma-coverage": "^1.0.0",
    "karma-jasmine": "^1.0.2",
    "karma-phantomjs-launcher": "^1.0.0",
    "karma-sourcemap-loader": "^0.3.7",
    "karma-spec-reporter": "0.0.26",
    "karma-webpack": "^1.7.0",
    "phantomjs-prebuilt": "^2.1.7",
    "raw-loader": "^0.5.1",
    "source-map-loader": "^0.1.5",
    "typescript": "^1.8.10",
```

```
    "webpack": "^1.13.0",
    "webpack-dev-server": "^1.14.1"
  }
}
```

We then call **npm start** when we are ready to compile and serve our application.

2.3. Bootstrapping

The first mystery of Angular 2 for me was **"How in the world do I even run the app!?"**.

The second mystery was **"Okay, so how do I bootstrap the application?!"**.

The first thing we need to do to bootstrap our application is to include the necessary resources into our **index.html** file. Because **webpack is bundling everything into a single file**, **we only need to include one bundle.js file**.

index.html

```
<!DOCTYPE html>
<html>

<head>
    <base href="/">
    <link href="//netdna.bootstrapcdn.com/twitter-
bootstrap/2.3.2/css/bootstrap-combined.min.css" rel="stylesheet">
    <link rel="icon" href="favicon.ico" title="favicon" charset="utf-8">
</head>

<body>
    <app>Loading...</app>

    <script src="bundle.js"></script>
</body>

</html>
```

Inside the **boot.ts** file, we are importing three components.

- **bootstrap**
- **ROUTER_PROVIDERS**
- **AppComponent**

We are also importing **core-js** **which is an ES6 polyfill**,
as well as **zone.js** **which Angular uses for change detection**.

We then **instantiate** our application and specify our **root level component**, **AppComponent** and **inject ROUTER_PROVIDERS** as a submodule.

boot.ts

```
import 'core-js';
import 'zone.js/dist/zone';

import {bootstrap} from '@angular/platform-browser-dynamic';
import {ROUTER_PROVIDERS} from '@angular/router';
import {AppComponent} from './app.component';

bootstrap(AppComponent, [
  ROUTER_PROVIDERS
]);
```

Back in our index.html file, our **entry point** in the markup happens at this line:

<app>Loading...</app>.

Angular has instantiated AppComponent and loaded its template into the app element.

index.html

```
<body>
    <app>Loading...</app>

    <script src="bundle.js"></script>
</body>
```

We have just covered how to compile, serve and bootstrap an Angular 2 application.

Let us unpack what a component consists of

so that we can make the connection between **how AppComponent becomes app** on our page.

2.4. The App Component

Angular **components** are really **just JavaScript classes wrapped with** love in Angular 2 **metadata**.

The first thing I do when creating a component is to **create the class**.

I will usually just stub it out because I will enhance it later.

The next thing we are going to do is to **import our dependencies**.

In this case, we just need to import Component from @angular/core.

app.component.ts

```
import {Component} from '@angular/core';
import {Routes, ROUTER_DIRECTIVES} from '@angular/router';

import {AboutComponent} from './about/about.component';
import {ExperimentsComponent} from './experiments/experiments.component';
import {HomeComponent} from './home/home.component';

import {StateService} from './common/state.service';
import {ExperimentsService} from './common/experiments.service';

@Component({
  selector: 'app',
  template: require('./app.component.html'),
  styles: [require('./app.component.css')],
  directives: [ ROUTER_DIRECTIVES ],
  providers: [StateService, ExperimentsService],
})
@Routes([
  {path: '/',            component: HomeComponent },
  {path: '/home',        component: HomeComponent },
  {path: '/about',       component: AboutComponent },
  {path: '/experiments', component: ExperimentsComponent },
  {path: '/*',          component: HomeComponent }
])
export class AppComponent {}
```

And from here, we are going to **decorate** our class by adding **@Component metadata** to tell our application how we want the AppComponent to behave.

We are **defining** the **HTML element** we want this class to target in the **selector property** as well as setting our **template** and **styles** via **templateUrl** and **styleUrls**, respectively.

We are also **injecting ROUTER_DIRECTIVES** into our component at the directives property.

app.component.ts

```
@Component({
  selector: 'app',
  template: require('./app.component.html'),
  styles: [require('./app.component.css')],
  directives: [ ROUTER_DIRECTIVES ],
  providers: [StateService, ExperimentsService],
})
export class AppComponent {}
```

This is where things get a bit fluid in the **CIDER process**(주: 컴포넌트 개발순서, 작자가 지은 말). I will often spend n iterations enhancing my components and this could go on indefinitely.

In our case, we want to add in routing to our application and so we are going to import the appropriate modules and decorate our component with **@Routes**.

To **enable routing**, we are going to **import Routes** and **ROUTER_DIRECTIVES**

as well as **AboutComponent**, **ExperimentsComponent** and **HomeComponent**.

app.component.ts

```
import {Routes, ROUTER_DIRECTIVES} from '@angular/router';

import {AboutComponent} from './about/about.component';
import {ExperimentsComponent} from './experiments/experiments.component';
import {HomeComponent} from './home/home.component';

import {StateService} from './common/state.service';
import {ExperimentsService} from './common/experiments.service';
```

We need the routing modules to enable routing and the additional components for our routing table. We are passing in an array of route definitions into **@Routes**, which tell us the path of the route and what component is going to be **mapped** to that route. We want our home route to be our **default route**; to accomplish this, we do two things. First, we add a '/' route to handle navigations to the top-level domain and reroute them to the home component. Then we will add in a '/*' route to **catch all other routes** and reroute them to the home component. We will circle back around and talk about how the templates deal with routes in a moment but the basic gist of the new component router in Angular 2 is that each route maps to a component.

Not really sure why they called it ComponentRouter though... JK!

app.component.ts

```
@Routes([
  {path: '/',              component: HomeComponent },
  {path: '/home',          component: HomeComponent },
  {path: '/about',         component: AboutComponent },
  {path: '/experiments', component: ExperimentsComponent },
  {path: '/*',            component: HomeComponent }
])
```

We are also going to import **StateService** and **ExperimentsService** into our component and then decorate our component on the **providers property** which we will cover later.

And I now present the #drumRoll AppComponent in its entirety!

app.component.ts

```
import {Component} from '@angular/core';
import {Routes, ROUTER_DIRECTIVES} from '@angular/router';
import {AboutComponent} from './about/about.component';
import {ExperimentsComponent} from './experiments/experiments.component';
import {HomeComponent} from './home/home.component';
import {StateService} from './common/state.service';
import {ExperimentsService} from './common/experiments.service';

@Component({
  selector: 'app',
  template: require('./app.component.html'),
  styles: [require('./app.component.css')],
  directives: [ ROUTER_DIRECTIVES ],
  providers: [StateService, ExperimentsService],
})
@Routes([
  {path: '/',            component: HomeComponent },
  {path: '/home',        component: HomeComponent },
  {path: '/about',       component: AboutComponent },
  {path: '/experiments', component: ExperimentsComponent },
  {path: '/*',          component: HomeComponent }
])
export class AppComponent {}
```

I really dig how everything up to this point has been **self-documenting** in its intentions and purpose. There isn't a single piece in this component that I cannot immediately figure out what its purpose is.

To complete the CIDER process, we are going to **repeat** the process on a **sub-component**. All three of our components defined in our router are pretty much the same and so we are going to focus on the HomeComponent. There is some additional functionality in the ExperimentComponent that we will cover in a later post.

2.5. The Home Component

We are going to break ground on our HomeComponent by defining the HomeComponent class. We are also defining and initializing **two properties** on our class for the component's **title** and **body**. This is a website after all!

home.component.ts

```
export class HomeComponent implements OnInit {
  title: string = 'Home Page';
  body:  string = 'This is the about home body';
}
```

We will import the appropriate dependencies.

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

We will then decorate our class and set the **selector** and **templateUrl** properties.

```
@Component({
  selector: 'home',
  template: require('./home.component.html')
})
```

We are going to use the **StateService** to store state between our routes
and so we will add that to our component.

Dependency injection within Angular 2 happens at the **constructor** and so we are going to add one to our class and inject the StateService.

Components also have lifecycle hooks that we can use to sequence units of work. In our case, we want to retrieve and set our message from StateService when our component is initialized. We will use the **ngOnInit** hook to make that call for us.

We don't have to import OnInit and extend the class, but it is a good idea semantically.

home.component.ts

```
import {Component, OnInit} from '@angular/core';
import {StateService} from '../common/state.service';

@Component({
  selector: 'home',
  template: require('./home.component.html')
})
export class HomeComponent implements OnInit {
  title: string = 'Home Page';
  body:  string = 'This is the about home body';
  message: string;

  constructor(private _stateService: StateService) { }

  ngOnInit() {
    this.message = this._stateService.getMessage();
  }

  updateMessage(m: string): void {
    this._stateService.setMessage(m);
  }
}
```

Again, incredibly self-documenting and **easy to read**.

2.6. The State Service

We are going to create the StateService class and then **expose** it as a **provider** to our application. I am not going to offer too much commentary on the code below because it is so rudimentary(기본적인) in nature. It is essentially a service that has a **getter** and a **setter** for a message property.

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

@Injectable()
export class StateService {
  private _message = 'Hello Message';

  getMessage(): string {
    return this._message;
  };

  setMessage(newMessage: string): void {
    this._message = newMessage;
  };
}
```

Things get interesting when we want to make StateService available for injection into other components. The first step is to import Injectable into our class.

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

And then we decorate it with @Injectable().

```
@Injectable()
export class StateService {
  private _message = 'Hello Message';

  getMessage(): string {
    return this._message;
  };

  setMessage(newMessage: string): void {
    this._message = newMessage;
  };
}
```

We have spent most of our time working through building our components,
so let us complete the loop by locking down our templates.

2.7. The Views

Using home.component.html as our reference point, let us take an abbreviated tour of the updated template syntax in Angular 2.

One-way data binding is defined exactly like it was in Angular 1.x through string **interpolation**.

home.component.html

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

User input events are no longer captured by adding custom Angular directives to our markup but rather through **capturing** native **DOM events** and wrapping them in **parenthesis**. We can see this in the code below as we capture the **click event** with (click)="updateMessage(message)" and call updateMessage.

```
<button type="submit" class="btn" (click)="updateMessage(message)">
        Update Message
</button>
```

Two-way data binding in Angular 2 is basically **one-way data binding applied twice**.

We talked about binding using string interpolation but we can also bind to properties using **brackets** syntax. We combine **property binding (component to view)** and **event binding (view to component)** to accomplish **two-way data binding**. The solution is surprisingly simple as we wrap our ngModel in both brackets and parenthesis to make [(ngModel)]="message".

```
<input type="text" [(ngModel)]="message" placeholder="Message">
```

For context, here is the entire home template.

home.component.html

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

<hr>
<div>
  <h2 class="text-error">Home: {{ message }}</h2>
  <form class="form-inline">
    <input type="text" [(ngModel)]="message" placeholder="Message">
    <button type="submit" class="btn" (click)="updateMessage(message)">
        Update Message
    </button>
  </form>
</div>
```

2.8. Routing Markup

We are back to where we started as we wrap up this lesson with a discussion on the template syntax for routing in app.component.html.

When we define a route, **where does the component's template actually get inserted?**
We set that insert point with **router-outlet**. It is the **new ngView**(주: 앵귤러 1 의 문법).

app.component.html

```
<div id="container">
  <router-outlet></router-outlet>
</div>
```

(주: id="container"는 부트스트랩 적용을 위한 속성)

Great! So **how do we navigate from one route to another?**
We do that with **routerLink** in the form of [routerLink]="['/home']"
as we can see in the code below.

app.component.html

```
<h1 id="logo">
  <a [routerLink]="['/Home']"></a>
</h1>

<div id="menu">
  <a [routerLink]="['/home']" class="btn">Home</a>
  <a [routerLink]="['/about']" class="btn">About</a>
  <a [routerLink]="['/experiments']" class="btn">Experiments</a>
</div>
```

(주: class="btn"은 CSS 적용을 위한 속성)

And the entire app.component.html.

```
<header id="header">
  <h1 id="logo">
    <a [routerLink]="['/Home']"></a>
  </h1>

  <div id="menu">
    <a [routerLink]="['/home']" class="btn">Home</a>
    <a [routerLink]="['/about']" class="btn">About</a>
    <a [routerLink]="['/experiments']" class="btn">Experiments</a>
  </div>
```

```
  <div class="color"></div>
  <div class="clear"></div>
</header>

<div class="shadow"></div>

<div id="container">
  <router-outlet></router-outlet>
</div>
```

2.9. Review

Before you raise your billable rates by 15% and start selling this sweet Angular 2 website, let us do a quick review of what we covered.

■ We define how we want to compile **TypeScript** in our **tsconfig.json** file.
■ We use **webpack-dev-server** to compile and serve our application.
■ We saw how to use **CIDER** to build out our AppComponent and HomeComponent
■ We learned how to created an **injectable service** using **@Injectable() metadata**.
■ We did an abbreviated tour of the Angular 2 **binding syntax**.
■ We learned how routes are added to our view with **router-outlet** and the syntax for navigating to a route using **routerLink**

Resources

Angular 2 Template Syntax
https://angular.io/docs/ts/latest/guide/template-syntax.html

tsconfig.json Documentation
http://www.typescriptlang.org/docs/handbook/tsconfig-json.html

TypeScript Documentation
https://github.com/Microsoft/TypeScript/wiki

Webpack
https://webpack.github.io/

Webpack Dev Server
https://webpack.github.io/docs/webpack-dev-server.html

CIDER

http://onehungrymind.com/cider-my-checklist-for-creating-angular-2-components/

- Create your class
- Import your dependencies
- Decorate your class
- Enhance with composition
- Repeat for sub-components

1. Create Your Class

The first thing we are going to do is create is our AppComponent class.

```
class AppComponent {
  public title = 'Tour of Heroes';
}
```

2. Import Your Dependencies

The next thing we are going to do is import our dependencies. The only dependency we have for AppComponent for now is just Component.

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

class AppComponent {
  public title = 'Tour of Heroes';
}
```

3. Decorate Your Class

Now that we have imported our basic Angular dependencies, it is time to decorate our class to do Angular things. We are going to tell Angular to take our AppComponent class and **bind it to the my-app element** with the my-template.html file.

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

class Hero {
  id: number;
  name: string;
}

@Component({
  selector: 'my-app',
  templateUrl:'my-template.html'
})
class AppComponent {
  public title = 'Tour of Heroes';
  public hero: Hero = {
    id: 1,
    name: 'Windstorm'
  };
}
```

In this example, I am using templateUrl because of formatting issues but you can also use the template property with string literals as illustrated in the tutorial.

```
<h1>{{title}}</h1>
<h2>{{hero.name}} details!</h2>
<div><label>id: </label>{{hero.id}}</div>
<div><label>name: </label>{{hero.name}}</div>
```

4. Enhance with Composition

Now that we have laid the foundation, it is time to enhance our component. We are going to take our current template and add in an input element which you can see below.

```
<h1>{{title}}</h1>
<h2>{{hero.name}} details!</h2>
<div><label>id: </label>{{hero.id}}</div>
<div>
  <label>name: </label>
  <div><input [(ng-model)]="hero.name" placeholder="name"></div>
</div>
```

We are going to import **FORM_DIRECTIVES** so that we can have **access to ng-model** in our template. This next step is very important: we add FORM_DIRECTIVES to the directives arrays.

```
import {Component, FORM_DIRECTIVES} from 'angular2/angular2';
class Hero {
  id: number;
  name: string;
}
@Component({
  selector: 'my-app',
  templateUrl:'my-template.html',
  directives: [FORM_DIRECTIVES]
})
class AppComponent {
  public title = 'Tour of Heroes';
  public hero: Hero = {
    id: 1,
    name: 'Windstorm'
  };
}
```

5. Repeat for Sub-Components

We have just enhanced our component to handle form input, but obviously we would not stop there in real-life. As our application grows in complexity, we are going to want to **abstract those units of complexity into sub-components.**

This is where the **CIDER process starts again**. For instance, you may have a form that has some complex validation that you want to isolate from the main component. You would start CIDER by first declaring a new class to encapsulate your form functionality, import your dependencies, and so on.

EXTRA: Bootstrap the Main Component

Since this is the main component, we are going to need to do one extra step to get our application running and that is to bootstrap it. We will first import the bootstrap dependency and then at the bottom of our class we **kick** things **off by calling bootstrap(AppComponent)**. You only need to do this once for your application and not for every single component.

```
import {bootstrap, Component, FORM_DIRECTIVES} from 'angular2/angular2';

class Hero {
  id: number;
  name: string;
}

@Component({
  selector: 'my-app',
  templateUrl:'my-template.html',
  directives: [FORM_DIRECTIVES]
})
class AppComponent {
  public title = 'Tour of Heroes';
  public hero: Hero = {
```

```
      id: 1,
      name: 'Windstorm'
    };
}
bootstrap(AppComponent);
```

Outro

Hopefully, this little trick that I use makes it easier for you to make the jump from Angular 1.X to Angular 2. Check out the resources below for a great place to start applying this technique. Let me know what you think in the comments!

Tour of Heroes Tutorial

https://angular.io/docs/ts/latest/tutorial/

Egghead.io – Angular 2 Fundamentals

https://egghead.io/courses/angular-2-fundamentals

Angular Class – Angular 2 Starter

https://github.com/AngularClass/angular2-webpack-starter

CIDER Practice

1. Create Your Class

etc/etc.component.ts

```
export class EtcComponent {
  title: string = 'ETC Page';
  body: string = 'This is the about etc body';
}
```

2. Import Your Dependencies

etc/etc.component.ts

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

export class EtcComponent {
  title: string = 'ETC Page';
  body: string = 'This is the about etc body';
}
```

3. Decorate Your Class

etc/etc.component.ts

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

@Component({
  selector: 'etc',
  template: require('./etc.component.html')
})
export class EtcComponent {
  title: string = 'ETC Page';
  body: string = 'This is the about etc body';
}
```

etc/etc.component.html

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

4. Enhance with Composition

app.component.ts

```
import {Component} from '@angular/core';
import {Routes, ROUTER_DIRECTIVES} from '@angular/router';

import {AboutComponent} from './about/about.component';
import {ExperimentsComponent} from './experiments/experiments.component';
import {HomeComponent} from './home/home.component';
import {EtcComponent} from './etc/etc.component';

import {StateService} from './common/state.service';
import {ExperimentsService} from './common/experiments.service';

@Component({
  selector: 'app',
  template: require('./app.component.html'),
  styles: [require('./app.component.css')],
  directives: [ ROUTER_DIRECTIVES ],
  providers: [StateService, ExperimentsService],
})
@Routes([
  {path: '/',            component: HomeComponent },
  {path: '/home',        component: HomeComponent },
  {path: '/about',       component: AboutComponent },
  {path: '/experiments', component: ExperimentsComponent },
  {path: '/etc',         component: EtcComponent },
  {path: '/*',          component: HomeComponent }
])
export class AppComponent {}
```

app.component.html

```
<header id="header">
  <h1 id="logo">
    <a [routerLink]="['/Home']"></a>
  </h1>

  <div id="menu">
    <a [routerLink]="['/home']" class="btn">Home</a>
    <a [routerLink]="['/about']" class="btn">About</a>
    <a [routerLink]="['/experiments']" class="btn">Experiments</a>
    <a [routerLink]="['/etc']" class="btn">ETC</a>
  </div>

  <div class="color"></div>
  <div class="clear"></div>
</header>
```

```
<div class="shadow"></div>

<div id="container">
  <router-outlet></router-outlet>
</div>
```

중간 확인

5. Repeat for Sub-Components

5.1. Create Your Class

etc/etc-body/etc.body.component.ts

```
export class EtcBodyComponent {
  name: string = 'Bob';
  address: string = 'Seoul Korea';
}
```

5.2. Import Your Dependencies

etc/etc-body/etc.body.component.ts

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

export class EtcBodyComponent {
  name: string = 'Bob';
  address: string = 'Seoul Korea';
}
```

5.3. Decorate Your Class

etc/etc-body/etc.body.component.ts

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

@Component({
  selector: 'etc-body',
  template: require('./etc.body.component.html')
})
export class EtcBodyComponent {
  name: string = 'Bob';
  address: string = 'Seoul Korea';
}
```

etc/etc-body/etc.body.component.html

```
<h3>
  {{ name }}
</h3>
```

```
<span>
  {{ address }}
</span>
```

5.4. Enhance with Composition

etc.component.ts

```
import {Component} from '@angular/core';
import {EtcBodyComponent} from './etc-body/etc.body.component';

@Component({
  selector: 'etc',
  template: require('./etc.component.html'),
  directives: [EtcBodyComponent]
})
export class EtcComponent {
  title: string = 'ETC Page';
  body: string = 'This is the about etc body';
}
```

etc.component.html

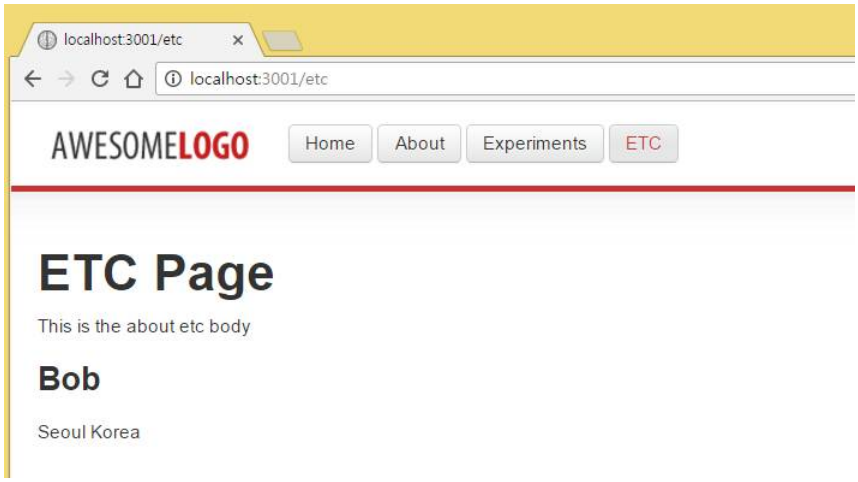
```
<h1>
  {{ title }}
</h1>
<p>
  {{ body }}
</p>
<etc-body></etc-body>
```

5.5. Repeat for Sub-Components

필요하다면 1 번부터 다시 반복한다.

결과 확인

6. 결과 확인


localhost:3001/etc
localhost:3001/etc
AWESOMELOGO
Home
About
Experiments
ETC
ETC Page
This is the about etc body
Bob
Seoul Korea

ES vs TS

es6-class.js

```
class Polygon {
  constructor(height, width) {
    let writer = 'Chris';
    this.height = height;
    this.width = width;
  }
  // 게터, 세터 메소드는 함수처럼 작성하지만 변수처럼 사용한다.
  get area() {
    return this.calcArea();
  }
  calcArea() {
    return this.height * this.width;
  }

  static help() {
    console.log('new Polygon(height, width)');
  }
}
const p = new Polygon(10, 10);
/*
constructor 함수안에서 this 로 정의한 자원은
new 키워드로 새로 만들어지는 객체에 프로퍼티로 추가된다.
*/
console.log(p); // {"height":10,"width":10}
console.log(p.area); // 100
// 게터, 세터 메소드는 ()를 사용하지 않고 변수처럼 사용한다.
console.log(p.calcArea()); // 100
/*
클래스안에 선언된 함수들은 클래스.prototype 객체에 메소드로 추가된다
*/
console.log(Object.getOwnPropertyNames(Polygon.prototype));
// [ 'constructor', 'area', 'calcArea' ]
```

```
console.log(typeof Polygon.prototype.area); // number
console.log(typeof Polygon.prototype.calcArea); // function
/*
함수앞에 static 키워드를 붙이면 함수객체 자신에 프로퍼티로 추가된다
*/
console.log(Object.getOwnPropertyNames(Polygon));
// [ 'length', 'name', 'prototype', 'help' ]
Polygon.help();
// new Polygon(height, width)
```

ts-class.ts

```
class Korean {
  _a: number = 1;
  _b: number;
  constructor(b: number) {
    this._b = b;
  }
  toString():string {
    return this._a +','+this._b;
  }
  get a():number {
    return this._a;
  }
  set a(a:number) {
    this._a = a;
  }
  static print(target):void {
    console.log('{a='+target.a+'}');
  }
}

var k: Korean = new Korean(2);

console.log(k);
// console.log(k.__proto__);
console.log(k.toString());
```

```
console.log(k.a);
k.a = 11;
console.log(k.a);
Korean.print(k);
```

ts-class2.ts

```
class Korean2 {
  constructor(public _a: number = 1, public _b: number = 2) {
    // this._c = 3; // 이렇게 사용하지 못한다.
  }
  toString(): string {
    return this._a + ',' + this._b;
  }
  get a(): number {
    return this._a;
  }
  set a(a: number) {
    this._a = a;
  }
  static print(target): void {
    console.log('{a=' + target.a + '}');
  }
}

var k: Korean2 = new Korean2();

console.log(k);
// console.log(k.__proto__);
console.log(k.toString());
console.log(k.a);
k.a = 11;
console.log(k.a);
Korean2.print(k);
```

class-extends.ts

```
class Animal {
  //#1: 외부에서 Animal 클래스의 객체를 생성할 수 없다. ~라는게 의도!
  protected constructor(public name: string, public leg: number) {

  }
  getLeg(): number {
    return this.leg;
  }
  //#2: 외부에서 Animal 클래스의 메소드를 접근할 수 없다. ~라는게 의도!
  protected getName(): string {
    return this.name;
  }
}

// var animal = new Animal('x', 4); // #1
// console.log(animal);
// console.log(animal.getName());

class Monkey extends Animal {
  // 생성자 생략가능
  constructor(name: string, leg: number) {
    super(name, leg);
  }
  isClimbing() {
    return true;
  }
  superGetName() {
    // #2: protected 로 선언한 자원을 클래스, 서브클래스에서는 접근 가능!
    return super.getName();
  }
}

var monkey: Monkey = new Monkey('ogong', 2);
console.log(monkey);
// Monkey { name: 'ogong', leg: 2 }
console.log(monkey.name);
console.log(monkey.leg);
```

```
// ogong
// 2
console.log(monkey.isClimbing());
console.log(monkey.superGetName());
// true
// ogong
console.log(monkey.getLeg());
// 2
// console.log(monkey.getName()); // #2
```

interface-implements.ts

```
interface Animal {
  // 변수, 메소드를 정의
  leg: number;
}

interface Bird extends Animal {
  wing: number;
  getWingNumber():number;
}

class BlueBird implements Bird {
  constructor(public leg: number, public wing: number) {}
  getWingNumber():number {
    return this.wing;
  }
}

let blueBird = new BlueBird(2, 2);
console.log(blueBird);
console.log(blueBird.getWingNumber());
```

interface-constructor.ts

```
interface Flyable {
  name: string;
}

class Flyer implements Flyable {
  constructor(public name: string) {
    console.log('Flyer.constructor(public name: string) called');
  }
}

interface Doable {
  new (Creater: string): Flyable;
}

function makeObject(Creater: Doable) {
  return new Creater('airplane');
}

console.log(makeObject(Flyer).name);
```

abstract-class.ts

```
abstract class SmallAnimals {
  abstract sound(): string;
  abstract name(): string;
  makeSound(): string {
    return `${this.name()}:${this.sound()}`;
  }
}

class Mouse extends SmallAnimals {
  sound(): string {
    return 'squeak';
  }
  name(): string {
    return 'mouse';
```

```
  }
}

var mouse = new Mouse();
console.log(mouse);
console.log(mouse.makeSound());
// Mouse {}
// mouse:squeak

// console.log(mouse.__proto__);
// console.log(mouse.__proto__.__proto__);
// Mouse {
//     constructor: [Function: Mouse],
//     sound: [Function],
//     name: [Function] }
// SmallAnimals { makeSound: [Function] }
```

Angular 2 소개

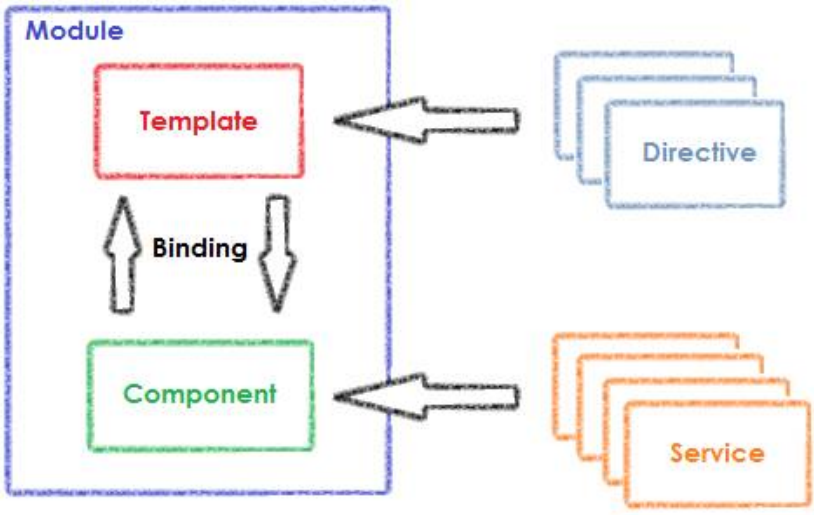
앵귤러는 구글에서 만든 자바스크립트 프레임워크다. 2010 년 10 월 정식버전을 발표하였다. 2016 년 9 월 앵귤러 2 버전을 발표하였다. 앵귤러 2 는 이전 버전의 처리속도를 개선하기 위해서 여러 단계를 거쳐 호출해야 하는 복잡한 단계를 없앴다. 더불어 자바스크립트의 장점인 높은 자율도에 의해서 발생되는 단점인 디버깅에 어려움을 해결하기 위해서 타입스크립트를 도입했다. 타입스크립트는 마이크로소프트가 2012 년 10 월 출시했다.

앵귤러를 사용해야 하는 이유

- 크로스 플랫폼 지원
- 처리속도
- 개발생산성

최근 들어 브라우저에서 동작하는 스크립트의 역할이 뷰에서 머물지 않고 컨트롤러까지 확대되었고 데이터를 자바스크립트 객체상태로 유지한다. 이에 따라 서버의 역할은 데이터를 제공하는 공급자 역할로 축소되고 있다.

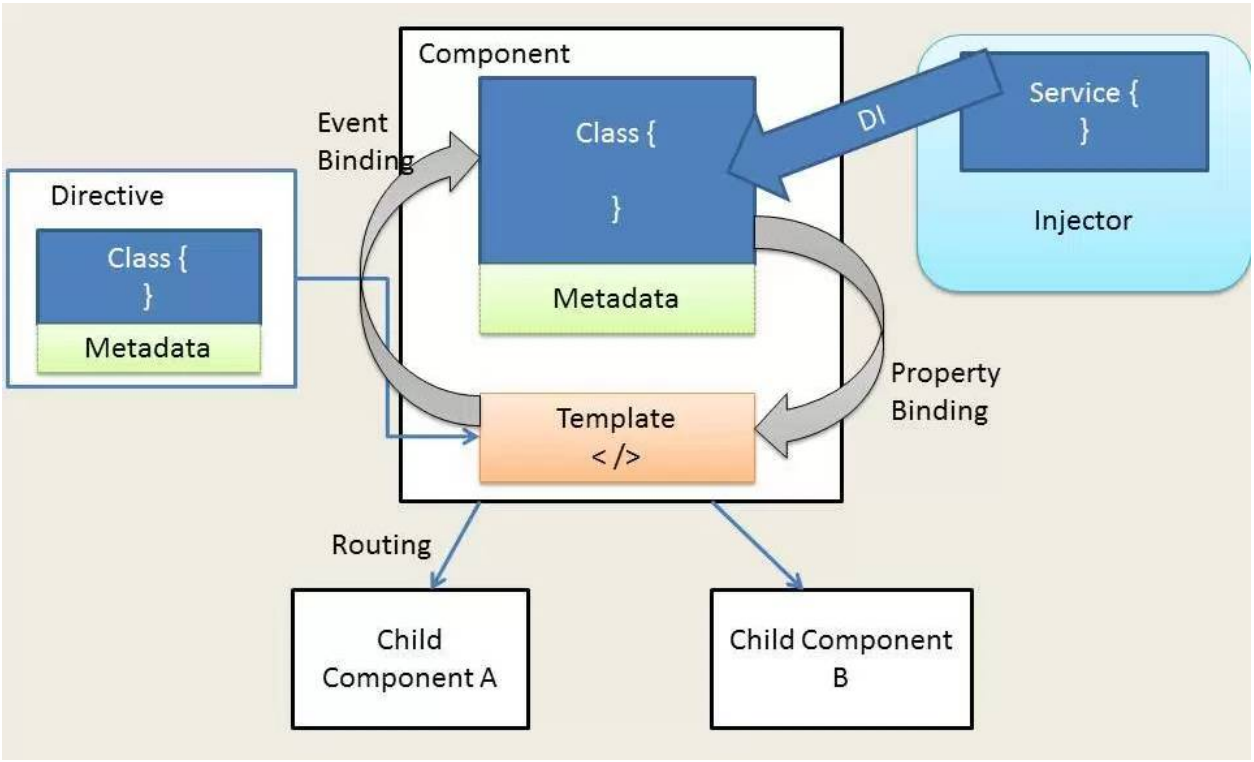
앵귤러 아키텍처



검색엔진에 컨텐츠가 노출되기 위해서는 서버에서 랜더링을 해야한다. 앵귤러 유니버셜을 이용하여 처리할 수 있다. https://universal.angular.io/

구성	책임
Module	화면의 구성요소를 묶어서 처리하는 앵귤러의 모듈시스템의 기본 단위
Component	템플릿과 바인딩으로 연결되며 필요한 로직을 처리하는 컨트롤러
Service	재사용하기 위해서 독립해서 정의하는 컴포넌트를 위한 처리 로직
Template	화면에 엘리먼트를 어떻게 배치할 것인지 정의하는 틀(엘리먼트들의 묶음)
Directive	재사용하기 위해서 독립해서 정의하는 엘리먼트를 위한 추가 기능

구성요소의 결합



Component 구성요소 3 가지

import : 외부 자원 임포트

@Component : 컴포넌트 Metadata 선언(Template 지정 필수)

class : 자원 및 처리 로직

양방향 바인딩 = 단방향 바인딩 + 단방향 바인딩

Template --(Event Binding)--> Component

Component --[Property Binding]--> Template

앵귤러 설정파일

package.json

```
{
  "name": "fem-ng2-simple-app",
  "version": "0.0.1",
  "license": "SEE LICENSE IN LICENSE",
  "repository": {
    "type": "git",
    "url": "https://github.com/onehungrymind/fem-ng2-simple-app/"
  },
  "scripts": {
    "start": "webpack-dev-server --inline --colors --watch --display-error-
details --display-cached  --port 3001 --hot",
    "test": "karma start"
  },
  "dependencies": {
    "@angular/common": "2.0.0-rc.1", // 서비스, 파이프, 디렉티브
    "@angular/compiler": "2.0.0-rc.1",
    "@angular/core": "2.0.0-rc.1",
    "@angular/platform-browser": "2.0.0-rc.1",
    "@angular/platform-browser-dynamic": "2.0.0-rc.1",
    "@angular/router": "2.0.0-rc.1",
    "es6-shim": "^0.35.0", // ES5 브라우저에서 ES6 호환성을 추가
    "reflect-metadata": "0.1.3", // 앵귤러와 타입스크립트 컴파일러와의 공유
    "rxjs": "5.0.0-beta.6", // 동기식 처리
    "systemjs": "^0.19.27", // ES6 다이나믹 모듈 로더
    "zone.js": "^0.6.12" // 변경감지
  },
  "devDependencies": {
    "awesome-typescript-loader": "^0.17.0",
    "core-js": "^2.4.0",
    "jasmine-core": "^2.4.1", // 테스트 프레임워크
    "karma": "^0.13.22",
    "karma-coverage": "^1.0.0",
    "karma-jasmine": "^1.0.2",
    "karma-phantomjs-launcher": "^1.0.0",
    "karma-sourcemap-loader": "^0.3.7",
    "karma-spec-reporter": "0.0.26",
    "karma-webpack": "^1.7.0",
    "phantomjs-prebuilt": "^2.1.7",
    "raw-loader": "^0.5.1",
    "source-map-loader": "^0.1.5",
    "typescript": "^1.8.10",
    "webpack": "^1.13.0",
    "webpack-dev-server": "^1.14.1"
  }
}
```

npm start 커맨드로 다음을 수행한다.

```
"scripts": {
    "start": "webpack-dev-server --inline --colors --watch --display-error-details --display-cached --port 3001 --hot",
    "test": "karma start"
  },
```

dependencies

운영 시 필요한 모듈을 설정한다.

devDependencies

개발 시 필요한 모듈을 설정한다.

tsconfig.json

```
{
  "compilerOptions": {
    "target": "es5", // 목표 결과 JS 의 버전
    "module": "commonjs", // 목표 모듈시스템
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "sourceMap": true,
    "suppressImplicitAnyIndexErrors":true
  },
  "compileOnSave": false,
  "buildOnSave": false,
  "exclude": [
    "node_modules"
  ],
  "filesGlob": [
    "app/**/*.ts",
    "typings/index.d.ts"
  ],
  "atom": {
    "rewriteTsconfig": false
  }
}
```

typings.json

```
{
  "globalDependencies": {
    "es6-shim": "github:DefinitelyTyped/DefinitelyTyped/es6-shim/es6-shim.d.ts#6697d6f7dadbf5773cb40ecda35a76027e0783b2",
    "hammerjs": "github:DefinitelyTyped/DefinitelyTyped/hammerjs/hammerjs.d.ts#74a4dfc1bc2dfadec47b8aae953b28546cb9c6b7",
    "jasmine": "github:DefinitelyTyped/DefinitelyTyped/jasmine/jasmine.d.ts#4b36b94d5910aa8a4d20bdcd5bd1f9ae6ad18d3c",
    "node": "github:DefinitelyTyped/DefinitelyTyped/node/node.d.ts#8cf8164641be73e8f1e652c2a5b967c7210b6729",
    "selenium-webdriver": "github:DefinitelyTyped/DefinitelyTyped/selenium-webdriver/selenium-webdriver.d.ts#a83677ed13add14c2ab06c7325d182d0ba2784ea",
    "webpack": "github:DefinitelyTyped/DefinitelyTyped/webpack/webpack.d.ts#95c02169ba8fa58ac1092422efbd2e3174a206f4"
  }
}
```

타입스크립트의 정의를 내린 d.ts 파일에 대한 설정을 관리한다.

http://wouterdekort.com/2016/04/04/typings-for-typescript/

When using TypeScript, you will **need TypeScript definition files to work with external libraries**. A lot of those definition files are available on GitHub: DefinitelyTyped. At the time of writing, there are 1708 entries which is to much to show, even for GitHub.

Third-party Declaration Files

When using a library that was originally designed for regular JavaScript, we need to apply a declaration file to make that library compatible with TypeScript. A declaration file has the extension .d.ts and contains various information about the library and its API.

TypeScript declaration files are usually written by hand, but there's a high chance that the library you need already has a .d.ts. file created by somebody else. DefinitelyTyped is the biggest public repository, containing files for over a thousand libraries.

http://definitelytyped.org/

There is also a popular Node.js module for managing TypeScript definitions called Typings.

https://github.com/typings/typings

If you still need to write a declaration file yourself, this guide will get you started.

http://www.typescriptlang.org/docs/handbook/writing-declaration-files.html

앵귤러 기동

npm start → package.json 파일안에 scripts 안에 스크립트 실행

package.json → 웹팩 기동 시 webpack.config.js 참조

```
"scripts": {
  "start": "webpack-dev-server --inline --colors --watch --display-error-
details --display-cached  --port 3001 --hot",
  "test": "karma start"
},
```

webpack.config.js → 시작 스크립트 실행(boot.ts)

```
var webpack = require('webpack'),
    path = require('path');

function root(args) {
  args = Array.prototype.slice.call(arguments, 0);
  return path.join.apply(path, [__dirname].concat(args));
}

module.exports = {
  devtool: 'source-map',
  debug: true,
  entry: './app/boot.ts',
  resolve: {
    extensions: ['', '.ts', '.js']
  },
  output: { // webpack 커맨드를 사용하면 bundle.js 파일을 build 폴더에 생성한다.
    path: './build',
    filename: 'bundle.js'
  },
  module: {
    preLoaders: [
      { test: /\.js$/, loader: 'source-map-loader', exclude:
[ root('node_modules/rxjs') ] }
    ],
    loaders: [
      { test: /\.ts$/, loader: 'awesome-typescript-loader', exclude:
[ /\.(spec|e2e)\.ts$/ ] },
      { test: /\.(html|css)$/, loader: 'raw-loader' }
    ]
  },
  devServer: {
    historyApiFallback: true
  }
};
```

boot.ts → ROUTER_PROVIDERS 를 DI 하면서 AppComponent 를 기동

```
import 'core-js';
import 'zone.js/dist/zone';

import {bootstrap} from '@angular/platform-browser-dynamic';
import {ROUTER_PROVIDERS} from '@angular/router';
import {AppComponent} from './app.component';

bootstrap(AppComponent, [
  ROUTER_PROVIDERS
]);
```

app.component.ts → 각 컴포넌트 기동 및 처리, 화면 처리

```
import {Component} from '@angular/core';
import {Routes, ROUTER_DIRECTIVES} from '@angular/router';

import {AboutComponent} from './about/about.component';
import {ExperimentsComponent} from './experiments/experiments.component';
import {HomeComponent} from './home/home.component';
import {EtcComponent} from './etc/etc.component';

import {StateService} from './common/state.service';
import {ExperimentsService} from './common/experiments.service';

@Component({
  selector: 'app',
  template: require('./app.component.html'),
  styles: [require('./app.component.css')],
  directives: [ ROUTER_DIRECTIVES ],
  providers: [StateService, ExperimentsService],
})
@Routes([
  {path: '/',            component: HomeComponent },
  {path: '/home',        component: HomeComponent },
  {path: '/about',       component: AboutComponent },
  {path: '/experiments', component: ExperimentsComponent },
  {path: '/etc',         component: EtcComponent },
  {path: '/*',          component: HomeComponent }
])
export class AppComponent {}
```

모든 JS 파일을 ES5 버전 코드로 바꾸고 하나로 묶는 작업을 거쳐 만들어지는 bundle.js 파일을 index.html 에서 사용한다.

"selector: 'app'"을 타겟(`<app>Loading...</app>`)으로 해서 app.component.html 파일을 배치한다.

index.html

```
<!DOCTYPE html>
<html>

<head>
    <base href="/">
    <link href="//netdna.bootstrapcdn.com/twitter-
bootstrap/2.3.2/css/bootstrap-combined.min.css" rel="stylesheet">
    <link rel="icon" href="favicon.ico" title="favicon" charset="utf-8">
</head>

<body>
    <app>Loading...</app>

    <script src="bundle.js"></script>
</body>

</html>
```

개발자는 확장자가 .ts 파일인 타입스크립트를 만들어 로직을 처리한다.

이는 결국 .js 파일로 바뀌어 사용된다. 더불어 빌드 시 디버깅을 위한 .map 파일도 만들어진다.

컴포넌트

컴포넌트의 분할(자식 컴포넌트 추가)

etc/etc-body/etc.body.component.ts

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

@Component({
  selector: 'etc-body',
  template: require('./etc.body.component.html')
})
export class EtcBodyComponent {
  name: string = 'Bob';
  address: string = 'Seoul Korea';
}
```

etc/etc-body/etc.body.component.html

```
<h3>
  {{ name }}
</h3>
<span>
  {{ address }}
</span>
```

etc/etc.component.ts

```
import {Component} from '@angular/core';
import {EtcBodyComponent} from './etc-body/etc.body.component';

@Component({
  selector: 'etc',
  template: require('./etc.component.html'),
  directives: [EtcBodyComponent]
})
export class EtcComponent {
  title: string = 'ETC Page';
  body: string = 'This is the about etc body';
}
```

etc/etc.component.html

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

```
<p>
  {{ body }}
</p>
<etc-body></etc-body>
```

복수 자식 컴포넌트

etc/etc-for/etc.for.component.ts

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

@Component({
  selector: 'etc-for',
  template: require('./etc.for.component.html')
})
export class EtcForComponent {
  name: string = 'Bob';
  address: string = 'Seoul Korea';
}
```

etc/etc-for/etc.for.component.html

```
<h3>
  {{ name }}
</h3>
<span>
  {{ address }}
</span>
```

etc/etc.component.ts

```
import {Component} from '@angular/core';
import {EtcBodyComponent} from './etc-body/etc.body.component';
import {EtcForComponent} from './etc-for/etc.for.component';

@Component({
  selector: 'etc',
  template: require('./etc.component.html'),
  directives: [EtcBodyComponent, EtcForComponent]
})
export class EtcComponent {
  title: string = 'ETC Page';
  body: string = 'This is the about etc body';
}
```

etc/etc.component.html

```
<h1>
  {{ title }}
</h1>
<p>
  {{ body }}
</p>
<etc-body></etc-body>
<hr>
<etc-for></etc-for>
```

데이터 루프 처리(ngFor)

etc/etc-for/etc.for.component.ts

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

@Component({
  selector: 'etc-for',
  template: require('./etc.for.component.html')
})
export class EtcForComponent {
  name: string = 'Bob';
  address: string = 'Seoul Korea';
  colors: any = ['red', 'green', 'blue'];
  friends: any = [{name: 'Aaron', phone: '1234'}, {name: 'David', phone:
'5678'}];
}
```

etc/etc-for/etc.for.component.html

```
<h3>
  {{ name }}
</h3>
<span>
  {{ address }}
</span>
<ul>
  <template ngFor let-color [ngForOf]="colors" >
    <li>{{color}}</li>
  </template >
</ul>
<ol>
  <template ngFor let-friend [ngForOf]="friends" >
    <li>{{friend.name}},{{friend.phone}}</li>
  </template >
</ol>
```

양방향 데이터 바인딩

etc/etc-message/etc.message.component.ts

```
import {Component, OnInit} from '@angular/core';
import {StateService} from '../../common/state.service';

@Component({
  selector: 'etc-message',
  template: require('./etc.message.component.html')
})
export class EtcMessageComponent implements OnInit{
  message: string;

  constructor(public _stateService: StateService) { }

  ngOnInit() {
    this.message = this._stateService.getMessage();
  }

  updateMessage(m: string): void {
    this._stateService.setMessage(m);
  }
}
```

etc/etc-message/etc.message.component.html

```
<div>
  <h2 class="text-error">About: {{ message }}</h2>
  <form class="form-inline">
    <input type="text" [(ngModel)]="message" placeholder="Message">
    <button type="submit" class="btn" (click)="updateMessage(message)">Update
Message</button>
  </form>
</div>
```

etc/etc.component.ts

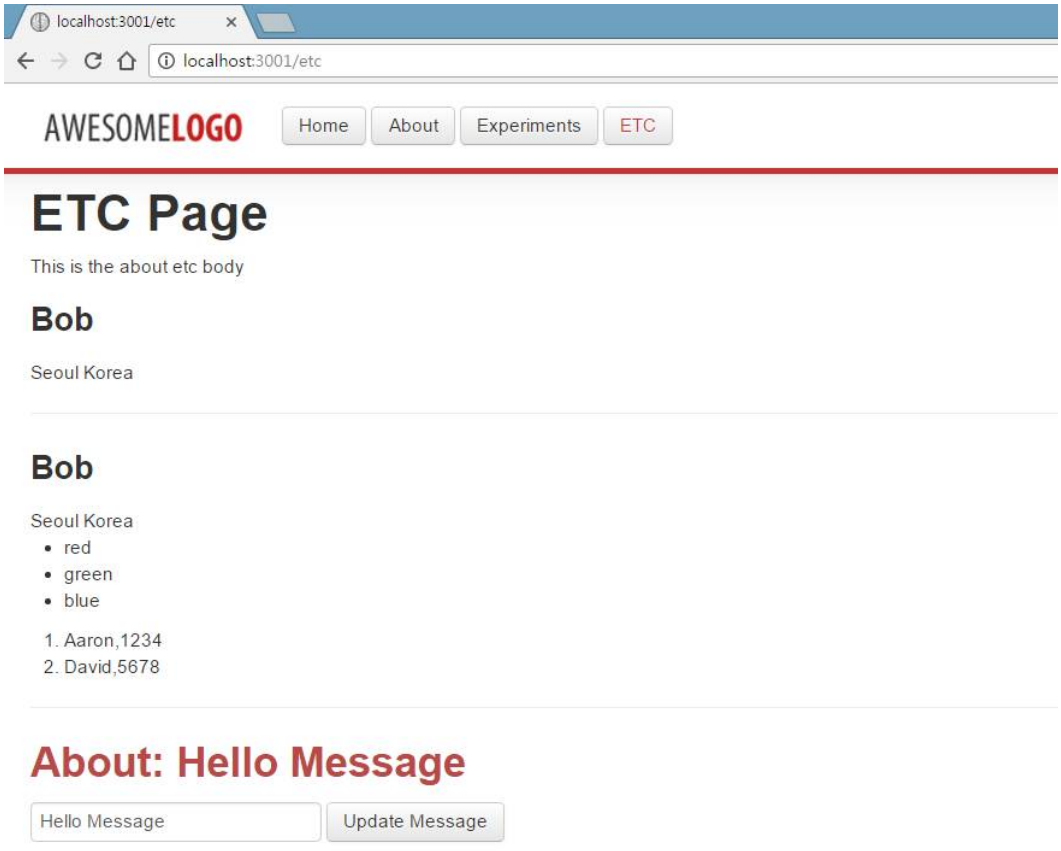
```
import {Component} from '@angular/core';
import {EtcBodyComponent} from './etc-body/etc.body.component';
import {EtcForComponent} from './etc-for/etc.for.component';
import {EtcMessageComponent} from './etc-message/etc.message.component';

@Component({
  selector: 'etc',
  template: require('./etc.component.html'),
  directives: [EtcBodyComponent, EtcForComponent, EtcMessageComponent]
})
export class EtcComponent {
  title: string = 'ETC Page';
```

```
  body: string = 'This is the about etc body';
}
```

etc/etc.component.html

```
<h1>
  {{ title }}
</h1>
<p>
  {{ body }}
</p>
<etc-body></etc-body>
<hr>
<etc-for></etc-for>
<hr>
<etc-message></etc-message>
```

결과화면



클라이언트-서버 데이터 동기화

1 단계 : 클라이언트 사이드 - 기본 틀

app/emp/emp.component.ts

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

@Component({
  selector: 'emp',
  template: require('./emp.component.html'),
})
export class EmpComponent {
  title: string = 'Employee Page';
  employees: any = [
    {id: 1, firstName: 'AAA', lastName: 'aaa'},
    {id: 2, firstName: 'BBB', lastName: 'bbb'},
    {id: 3, firstName: 'CCC', lastName: 'ccc'}];

  add(firstName: string, lastName: string): void {
    alert(firstName + ' ' + lastName);
  }

  remove(person: any): void {
    alert(JSON.stringify(person));
  }
}
```

app/emp/emp.component.html

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

<div>
  <input type="text" [(ngModel)]="firstName" placeholder="first name">
  <input type="text" [(ngModel)]="lastName" placeholder="last name">

  <button (click)="add(firstName, lastName)">Add</button>

  <div>
    <table>
      <template ngFor let-p [ngForOf]="employees">
        <tr>
          <td>{{p.id}}</td>
          <td><span>{{p.firstName}} {{p.lastName}}</span></td>
          <td><button (click)="remove(p)">Remove</button></td>
```

```
        </tr>
      </template >
    </table>
  </div>
</div>
```

app/app.component.ts

```
import {Component} from '@angular/core';
import {Routes, ROUTER_DIRECTIVES} from '@angular/router';

import {AboutComponent} from './about/about.component';
import {ExperimentsComponent} from './experiments/experiments.component';
import {HomeComponent} from './home/home.component';
import {EtcComponent} from './etc/etc.component';
import {EmpComponent} from './emp/emp.component';

import {StateService} from './common/state.service';
import {ExperimentsService} from './common/experiments.service';

@Component({
  selector: 'app',
  template: require('./app.component.html'),
  styles: [require('./app.component.css')],
  directives: [ ROUTER_DIRECTIVES ],
  providers: [StateService, ExperimentsService],
})
@Routes([
  {path: '/',            component: HomeComponent },
  {path: '/home',        component: HomeComponent },
  {path: '/about',       component: AboutComponent },
  {path: '/experiments', component: ExperimentsComponent },
  {path: '/etc',         component: EtcComponent },
  {path: '/emp',         component: EmpComponent },
  {path: '/*',          component: HomeComponent }
])
export class AppComponent {}
```

app/app.component.html

```
<header id="header">
  <h1 id="logo">
    <a [routerLink]="['/Home']"></a>
  </h1>

  <div id="menu">
    <a [routerLink]="['/home']" class="btn">Home</a>
    <a [routerLink]="['/about']" class="btn">About</a>
    <a [routerLink]="['/experiments']" class="btn">Experiments</a>
    <a [routerLink]="['/etc']" class="btn">ETC</a>
    <a [routerLink]="['/emp']" class="btn">Employee</a>
  </div>
```

```
  <div class="color"></div>
  <div class="clear"></div>
</header>

<div class="shadow"></div>

<div id="container">
  <router-outlet></router-outlet>
</div>
```

2 단계: 서버 사이드 - Restful Service

Spring STS > Spring Boot Project > angular-server

AngularServerApplication.java

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.web.servlet.config.annotation.CorsRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;

@SpringBootApplication
public class AngularServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(AngularServerApplication.class, args);
    }

    @Bean
    public WebMvcConfigurer corsConfigurer() {
        return new WebMvcConfigurerAdapter() {
```

```
            @Override
            public void addCorsMappings(CorsRegistry registry) {
                registry.addMapping("/employees/**")
                        .allowedOrigins("*")
                        .allowedMethods("POST", "GET", "PUT", "DELETE", "OPTIONS");
            }
        };
    }
}
```

Employee.java

```
package com.example.common.model;

public class Employee {
    private int id;
    private String firstName;
    private String lastName;

    public Employee() {}
    public Employee(int id, String firstName, String lastName) {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }

    public String getFirstName() {
```

```
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

EmployeeRestController.java

```
package com.example.common.controller;

import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;

import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

import com.example.common.model.Employee;

@RestController
public class EmployeeRestController {
```

```
    private List<Employee> employees = new ArrayList<Employee>();

    public EmployeeRestController() {
        employees.add(new Employee(1, "ADAM", "Sandler"));
        employees.add(new Employee(2, "BOB", "Ross"));
        employees.add(new Employee(3, "CHRIS", "Evans"));
    }

    @GetMapping("/employees")
    public Object get() {
        return employees;
    }

    @PostMapping("/employees")
    public Object post(@RequestBody Employee employee) {
        if (employees.size() > 0) {
            Comparator<Employee> comp = (e1, e2) -> Integer.compare( e1.getId(), e2.getId());
            Employee emp = employees.stream().max(comp).get();

            employee.setId(emp.getId()+1);
        } else {
            employee.setId(1);
        }

        employees.add(employee);
        return employee;
    }

    @DeleteMapping("/employees/{id}")
    public Object delete(@PathVariable int id) {
        Employee emp = employees.stream()
                .filter((e) -> e.getId() == id)
                .findAny().orElse(null);
```

```
            if (emp != null) {
                employees.remove(emp);
            }
            return employees;
    }
}
```

서버 서비스 확인

3 단계 : 클라이언트 사이드 - HTTP 로직

모듈 추가

```
npm install --save @angular/http@2.0.0-rc.1
```

app/boot.ts

```
import 'core-js';
import 'zone.js/dist/zone';

import {bootstrap} from '@angular/platform-browser-dynamic';
import {ROUTER_PROVIDERS} from '@angular/router';
import {AppComponent} from './app.component';

// Property 'map' does not exist on type 'Observable<Response>'
```

```
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/catch';
import {HTTP_PROVIDERS} from '@angular/http';

bootstrap(AppComponent, [
  ROUTER_PROVIDERS, HTTP_PROVIDERS
]);
```

app/common/http.emp.service.ts

```
import {Injectable} from '@angular/core';
import {Http, Response} from '@angular/http';
import {Headers, RequestOptions} from '@angular/http';
import {Observable} from 'rxjs/Observable';
import {Emp} from '../emp/emp';

@Injectable()
export class HttpEmpService {
  private empsUrl: string = "http://localhost:8080/employees";

  constructor(private http: Http) {}

  getEmps (): Observable<Emp[]> {
    return this.http.get(this.empsUrl)
      .map(this.extractData)
      .catch(this.handleError);
  }

  addEmp (firstName: string, lastName: string): Observable<Emp> {
    let headers = new Headers({ 'Content-Type': 'application/json' });
    let options = new RequestOptions({ headers: headers });
    let emp = { "id":0, "firstName":firstName, "lastName":lastName };

    return this.http.post(this.empsUrl, JSON.stringify(emp), options)
      .map(this.extractDataForObject)
      .catch(this.handleError);
  }

  private headers = new Headers({'Content-Type': 'application/json'});

  removeEmp (emp): Observable<Emp[]> {
    const url = `${this.empsUrl}/${emp.id}`;
    return this.http.delete(url, {headers: this.headers})
      .map(this.extractData)
      .catch(this.handleError);
  }

  private extractData(res: Response) {
    console.log('res = '+JSON.stringify(res));
    // res =
    // {"_body":"[
    //  {\"id\":1,\"firstName\":\"111\",\"lastName\":\"aaa\"},
    //  {\"id\":2,\"firstName\":\"222\",\"lastName\":\"bbb\"},
```

```
    //  {\"id\":3,\"firstName\":\"333\",\"lastName\":\"ccc\"}]",
    // "status":200,
    // "ok":true,
    // "statusText":"Ok",
    // "headers":{"Content-Type":["application/json;charset=UTF-8"]},
    // "type":2,"url":"http://localhost:8080/employees"}
    console.log(typeof res); // object

    // Property '_body' is private and only accessible
    // within class 'Response'.
    //console.log('res._body = '+JSON.stringify(res._body));

    let json = res.text();
    json = JSON.parse(json);
    return json || [];
  }

  private extractDataForObject(res: Response) {
    let json = res.text();
    json = JSON.parse(json);
    return json || {};
  }

  private handleError (error: Response | any) {
    // In a real world app, we might use a remote logging infrastructure
    let errMsg: string;
    if (error instanceof Response) {
      const body = error.json() || '';
      const err = body.error || JSON.stringify(body);
      errMsg = `${error.status} - ${error.statusText || ''} ${err}`;
    } else {
      errMsg = error.message ? error.message : error.toString();
    }
    console.error(errMsg);
    return Observable.throw(errMsg);
  }
}
```

app/emp/emp.ts

```
export class Emp {
  constructor(
        public id: number,
        public firstName: string,
        public lastName: string) {
  }
}
```

app/emp/emp.component.ts

```
import {Component, OnInit} from '@angular/core';
import {HttpEmpService} from '../common/http.emp.service';
```

```
import {Emp} from './emp';

@Component({
  selector: 'emp',
  template: require('./emp.component.html'),
  providers: [HttpEmpService]
})
export class EmpComponent implements OnInit {

  title: string = 'Employee Page';
  employees: any = [
    {id: 1, firstName: 'AAA', lastName: 'aaa'},
    {id: 2, firstName: 'BBB', lastName: 'bbb'},
    {id: 3, firstName: 'CCC', lastName: 'ccc'}];

  add(firstName: string, lastName: string): void {
    alert(firstName + ' ' + lastName);
  }

  remove(person: any): void {
    alert(JSON.stringify(person));
  }

  constructor(private httpEmpService: HttpEmpService) {
  }

  ngOnInit() {
    this.getEmps();
  }

  errorMessage: string;
  emps: Emp[];

  getEmps() {
    this.httpEmpService.getEmps()
      .subscribe(
        emps => this.emps = emps,
        error => this.errorMessage = <any>error);
  }

  addEmp(firstName: string, lastName: string) {
    if (!firstName || !lastName) {
      return;
    }
    this.httpEmpService.addEmp(firstName, lastName)
      .subscribe(
        car => this.emps.push(car),
        error => this.errorMessage = <any>error);
  }

  removeEmp(emp: any) {
    this.httpEmpService.removeEmp(emp)
      .subscribe(
        emps => this.emps = emps,
```

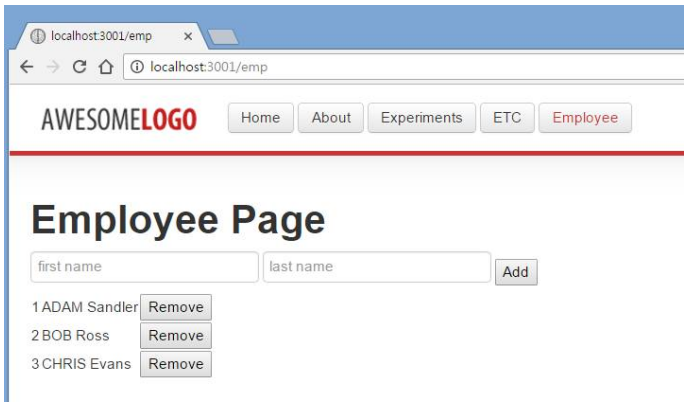
```
        error => this.errorMessage = <any>error);
  }
}
```

app/emp/emp.component.html

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

<div>
  <input type="text" [(ngModel)]="firstName" placeholder="first name">
  <input type="text" [(ngModel)]="lastName" placeholder="last name">

  <button (click)="addEmp(firstName, lastName); firstName=''; last-
Name='';">Add</button>

  <div>
    <table>
      <template ngFor let-p [ngForOf]="emps">
        <tr>
          <td>{{p.id}}</td>
          <td><span>{{p.firstName}} {{p.lastName}}</span></td>
          <td><button (click)="removeEmp(p)">Remove</button></td>
        </tr>
      </template>
    </table>
  </div>
</div>

<p class="error" *ngIf="errorMessage">{{errorMessage}}</p>
```

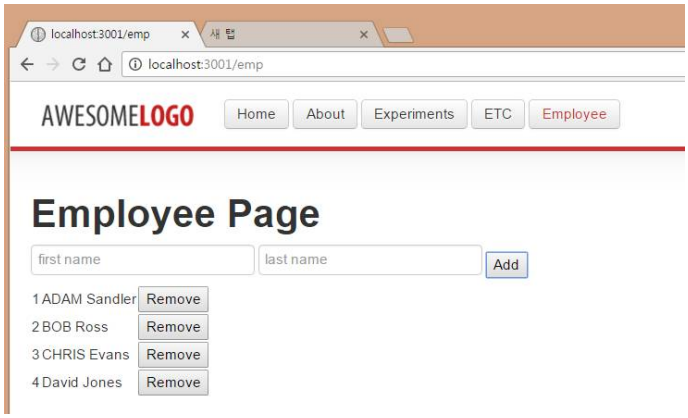
결과 확인

조회화면



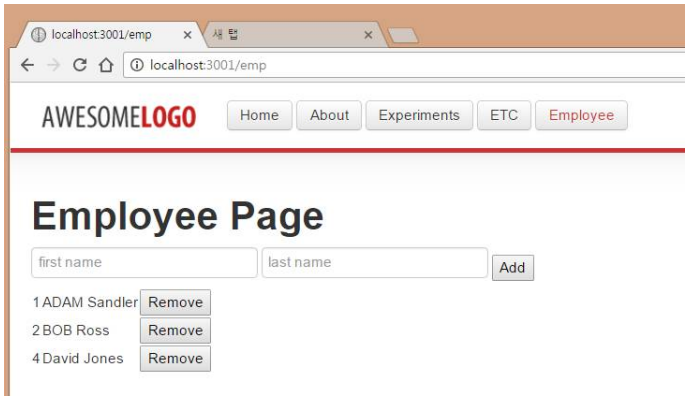
항목추가 화면

first name, last name 입력 후 Add 버튼 클릭



삭제화면

3 번 항목 오른쪽 Remove 버튼 클릭



앵귤러 아키텍처 구성요소

Controllers

The best practices for building AngularJS applications state is that the **controllers should not manipulate the DOM at all**, instead all **DOM access and manipulations should be isolated in directives**. If we have some repetitive logic between controllers, most likely we want to encapsulate it into a service and **inject this service with the dependency injection mechanism** of AngularJS in all the controllers that need that functionality.

This is where we're coming from in AngularJS 1. x. All this said, it seems that the functionality of controllers could be moved into the directive's controllers. Since directives support the dependency injection API, after receiving the user's input, we can directly delegate the execution to a specific service, already injected. This is the main reason **Angular 2** uses a different approach by **removing the ability to put controllers everywhere by using the ng-controller directive**.

앵귤러 2 에서는 컨트롤러가 사라졌다. 이를 컴포넌트의 클래스가 대신한다.

```
@Component({
    selector: 'hello-world',
    template: '< h1 > Hello, {{ target}}! </ h1 >' })
class HelloWorld {
    target: string;
    constructor() { this.target = 'world'; }
}
```

Scope

The **data-binding in AngularJS is achieved using the scope object**. We can **attach properties to it**(스코프) and explicitly declare in the template that we want to bind to these properties (one or two-way). Although the idea of the scope seems clear, **the scope has two more responsibilities**, **including event dispatching** and **the change detection-related behavior**. Angular beginners have a hard time understanding what scope really is and how it should be used. AngularJS

1.2 introduced something called **controller as syntax**. It allows us **to add properties to the current context inside the given controller (this)**, instead of explicitly injecting the scope object and later adding properties to it. This simplified syntax can be demonstrated from the following snippet:

Angular 2 took this even further by **removing the scope object**. All the expressions are evaluated in the context of given UI component. **Removing the entire scope** API introduces higher simplicity; we don't need to explicitly inject it anymore and we **add properties to the UI components** to which we can later bind. This API feels much simpler and more natural.

앵귤러 2 에서는 스코프가 사라졌다. 컴포넌트 클래스 별로 자원을 관리한다. 컴포넌트들 끼리의 자원공유는 서비스를 사용한다.

Injection

Maybe the first framework on the market that included inversion of control (IoC) through dependency injection (DI) in the JavaScript world was AngularJS 1. x. **DI** provides a number of **benefits**, such as easier **testability, better code organization and modularization, and simplicity**. Although the DI in 1. x does an amazing job, Angular 2 takes this even further. Since Angular 2 is on top of the latest web standards, it **uses the ECMAScript 2016(ES7) decorators' syntax** for annotating the code for using DI. Decorators are quite **similar to** the decorators in Python or **annotations in Java**. They allow us to decorate the behavior of a given object by **using reflection**. Since decorators are not yet standardized and supported by major browsers, their usage requires an intermediate transpilation step; however, if you don't want to take it, you can directly write a little bit more verbose code with ECMAScript 5 syntax and achieve the same semantics.

앵귤러 2 는 데코레이터(결국 자바의 애노테이션과 같음)를 사용하여 선언적으로 DI 를 설정한다.
데코레이터는 ES7 에서 제안하므로 현재 브라우저가 지원하는 ES5 로에 트랜스파일링이 필요하다.

Server-side rendering

Another typical use case for the server-side rendering is for building **Search Engine Optimization (SEO)**-friendly applications. There were a couple of hacks used in the past for making the AngularJS 1. x applications indexable by the search engines. One such practice, for instance, is traversal

of the application with a headless browser, which executes the scripts on each page and caches the rendered output into HTML files, making it accessible by the search engines.

The decoupling of Angular 2 with the DOM allows us to run our Angular 2 applications outside the context of the browser.

화면구성을 클라이언트에서 행하게 되면 서치엔진에 컨텐츠가 노출되지 않는다. 앵귤러 2 를 서버사이드에서 처리해야 서치엔진에 컨텐츠가 노출된다. 앵귤러 2 를 서포트하는 유니버셜 앵귤러를 사용하여 이를 처리할 수 있다.

TypeScript

The Angular core team decided to use TypeScript because of the better tooling possible with it and the **compile-time type checking**, which help us be more productive and less error-prone. As the preceding figure shows, **TypeScript is a superset of ECMAScript**; it introduces explicit type annotations and a compiler. The TypeScript language is compiled to plain JavaScript, supported by today's browsers. Since version 1.6, TypeScript implements the ECMAScript 2016 decorators, which makes it the perfect choice for Angular 2.

타입스크립트는 ECMAScript 의 슈퍼셋이다. 타입스크립트를 사용하면 컴파일타임에 타입체크가 가능해서 현저히 에러를 줄일 수 있다. 또한 IDE 툴을 사용하는 경우 타입정보를 바탕으로 개발자의 코드작성을 도울 수 있다.

Templates

Templates are one of the key features in AngularJS 1. x. They **are simple HTML** and do not require any intermediate processing and compilation, unlike most template engines such as mustache. Templates in AngularJS combine simplicity with power by allowing us to **extend HTML by creating an internal Domain Specific Language (DSL)** inside it, **with custom elements and attributes**.

For example, let's say we built a directive and we want to allow the user to pass a property to it by using an attribute. In AngularJS 1. x, we can approach this in three different ways:

```
< user name =" literal" > </ user >
< user name =" expression" > </ user >
< user name ="{{ interpolate}}" > </ user >
```

ng-for

For instance, if we want to iterate over a list of users and display their names in AngularJS 1. x, we can use:

```
< div ng-for =" user in users" >{{ user.name}} </ div >
```

Although this syntax looks intuitive to us, it allows limited tooling support.
However, Angular 2 approached this differently by bringing a little bit more explicit syntax with richer semantics:

```
< template ngFor let-user [ngForOf] =" users" > {{ user.name}}</ template >
```

However, this syntax is too verbose for typing. Developers can use the following syntax, which later gets translated to the more verbose one:

```
< li *ngFor =" let user of users" > {{ user.name}} </ li >
```

Detection Mechanisms

As a result, Angular 2 has two built-in change detection mechanisms:

Dynamic change detection:

This is **similar to the change detection mechanism used by AngularJS 1.x**. It is used in systems with disallowed eval(), such as CSP and Chrome extensions.

JIT change detection:

This generates the code that performs the change detection run-time, allowing the JavaScript virtual machine to perform further code optimizations.
성능을 향상시키기 위한 일종의 Lazy Loading 이다.

Directive

The main purpose of Angular 2' s directives is to attach behavior to the DOM by extending it with custom logic defined in an ES2015(ES6) class. We can think of these classes **as controllers** associated to the directives, and think of their **constructors as similar to the linking function** of the directives from AngularJS 1. x. However, the new directives have limited configurability. They **do not allow for the definition of a template**, which makes most of the already known properties for defining directives unnecessary.

Components VS Directive

Components

■ To register a component we use @Component meta-data annotation.

■ **Component is a directive which uses shadow DOM** to create encapsulated visual behavior called components. Components are typically used to create UI widgets.

■ Component is used to break up the application into smaller components.

■ Only one component can be present per DOM element.

■ @View decorator or templateurl template are mandatory in the component.

Directive

■ To register directives we use @Directive meta-data annotation.

■ Directive is used to **add behavior to an existing DOM element**.

■ Directive is use to design re-usable components.

■ Many directives can be used per DOM element.

■ **Directive doesn't use View**.

@NgModule

https://scotch.io/bar-talk/getting-to-know-angular-2s-module-ngmodule

Angular 2 has re-introduced the module concept since RC 5 onwards (and now Final Release!). Why do we say reintroduced instead of just introduced? Modules existed in Angular 1!

The initial plan for Angular 2 was to **drop the angular.module()** and **use ES6 modules instead**. Throwback to the day when Igor Minar and Tobias Bosch announced the killing of ng.module() during ng-europe conference, Oct 2014.

앵귤러 1 의 angular.module 은 앵귤러 2 에서 사라졌다. 대신 ES6 가 제안하는 모듈시스템을 사용한다.

Module in Angular 1

You may skip this part if you are totally new to Angular.

An Angular 1 module is a collection of:

■ services

■ directives

■ controllers

■ filters

■ configuration information.

■

You syntax will look something like this:

```
// ng1-my-app.js


...
// Creating a new module
var myModule = angular.module('myModule', []);
```

```
// Defining a new module
myModule
    .value('appName', 'MyCoolApp');
    .controller('controllerName', ControllerFunction)
    .service('serviceName', ServiceFunction)
    .filter('filterName', FilterFunction);
...

// Create a new module that uses myModule
var anotherModule = angular.module('anotherModule', ['myModule']);
```

Module in Angular 2

Fast forward to Angular 2. **Angular Module is now called @NgModule**.

Question 1 **Why do we need @NgModule?**
Short Answer: Angular Modules help **to organize** an application **into cohesive blocks** of functionalities and extend it with capabilities from external libraries.

For longer, deeper and more technical answer, check out the explanation in the Angular blog or this document - Understanding @NgModule.

Question 2 But... **Don't we have Javascript Modules already?**
We can already import and export each JavaScript file as a module, right?

Take a deep breath, long answer ahead

In Javascript Modules, every file is one module.
In Angular 2, one component is normally a file (if you follow the style guide).

Take for example, you have 10 components in your project, and that translates into 10 files.
Now, given that you have another two new components - namely component A and B.
These **2 components depend on the 10 components** we just described above.

Prior to RC 5, this is how you do it.

```
// a.component.ts

// import all 10 components here
import { No1Component, No2Component, ... } from './components';

@Component({
  selector: 'a-cmp',
  templateUrl: 'a-cmp.component.html',
  directives:[No1Component, No2Component, ...] // inject all 10 components here
})
export class AComponent {
}
```

Then, **repeat the same process for component B**. Now, if we were to create many more components that behave like component A and B, it surely will become cumbersome to keep including these 10 components into every other component. It is certainly not a fun thing to do! However, you can improve the code by creating barrel. Lets look at a more refactored code:

```
// barrel.ts

// import all 10 components here
import { No1Component, No2Component, ... } from './components';

// export an array of all 10 components
export const groupedComponents = [No1Component, No2Component, ...];
```

Then change your component A code, to import the barrel to Component A

```
// a.component.ts

import { groupedComponents } from './barrel';
```

```
@Component({
   selector: 'a-cmp',
   templateUrl: 'a-cmp.component.html',
   directives:[groupedComponents] // inject all grouped components here
})
export class AComponent {
}
```

The code sure does looks neater now

but you **still have to include groupedComponents in every component** that you need it.

With @NgModule, what you need to do is this

```
// a.component.ts

@Component({
   selector: 'a-cmp',
   templateUrl: 'a-cmp.component.html'
   // no more directives, no more importing component to component
})
export class AComponent {
}
```

Yes, no more importing components / directives / pipes to each of your components (in our case, component A and B). **Instead, we import all to @NgModule** and all components / directives / pipes will be **available throughout the components under the module**.

Every Angular app has at least one module, the root module,
conventionally named AppModule.

Here is how a typical AppModule will look like:

```
// app.module.ts
```

```
@NgModule({
    declarations: [ // put all your components / directives / pipes here
        AppComponent, // the root component
        No1Component, No2Component, ... // e.g. put all 10 components here
        AComponent, BComponent, // e.g. put component A and B here
        NiceDirective,
        AwesomePipe,
    ],
    imports: [ // put all your modules here
        BrowserModule, // Angular 2 out of the box modules
        TranslateModule, // some third party modules / libraries
    ],
    providers: [ // put all your services here
        AwesomeService,
    ],
    bootstrap: [ // The main components to be bootstrapped in main.ts file, normally one only
        AppComponent
    ]
})
export class AppModule { }
```

Later on, you can bootstrap your application using this app.module.ts.

```
₩₩ main.ts

...
platformBrowserDynamic().bootstrapModule(AppModule);
...
```

There are 3 more less commonly used @NgModule properties (exports, schemas, entryComponents) which I didn't include for the sake of simplicity.

exports : 다른 모듈에서 사용할 수 있도록 정의한 서브셋이다.
imports : 다른 모듈에서 exports 로 선언한 서브셋을 가져온다.

@NgModule 로 선언하여 특정 범주의 기능들을 묶어서 관리한다. 모듈을 사용함으로써 각 그룹별 관리가 가능해지고 그룹내에서 공동으로 사용하는 기능을 추가한다.

main.ts : 사용하고 싶은 라이브러리의 특정 폴리필 기능을 설정한다. 기동 모듈을 설정한다.
app.module.ts : 전역적으로 사용할 기능을 설정한다. 기동 컴포넌트를 설정한다.

@angular/common : pipe, ngIf, ngFor, ngSwitch
@angualr/core : 핵심모듈
@angular/forms : 폼 모듈
@angular/http : HTTP 모듈
@angular/platform-browser : 브라우저 모듈, 새니타이저
@angular/router : 라우터 모듈
@angular/testing : 테스트 모듈

Module 연습

작업공간으로 커서를 이동한다.

angula-cli 를 이용하여 프로젝트를 만든다.

```
ng new hello-ng2
cd hello-ng2
ng g component book-component
```

다음 작업은 수동으로 진행한다.

cust/cust.module.ts

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

import { CustComponent }   from './cust.component'
import { CustService }   from './cust.service'

@NgModule({
  imports:      [ CommonModule, FormsModule ],
  declarations: [ CustComponent ],
  exports:      [ CustComponent ],
  providers:    [ CustService ]
})
export class CustModule { }
```

cust.module.ts 모듈에서 CustComponent 컴포넌트를 익스포트한다.

cust/cust.component.ts

```
import {Component} from '@angular/core'
import { CustService }    from './cust.service'

@Component({
    selector: 'cust',
    templateUrl: './app/cust/cust.component.html',
    styleUrls:["./assets/stylesheets/cust.css"]
})
export class CustComponent {
    name:string
    age:number
    constructor(private custService: CustService) {
        var obj = custService.getCust()
```

```
        this.name = obj.name
        this.age = obj.age
    }
}
```

cust/cust.component.html

```
<h2>구매고객</h2>
<p class="font-color">{{name}}</p>
<p class="font-color">{{age}}</p>
```

cust/cust.service.ts

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

@Injectable()
export class CustService {
  getCust() {
    return {
        name:'홍길동',
        age:21
    }
  }
}
```

app.module.ts

```
import {BrowserModule} from '@angular/platform-browser';
import {NgModule} from '@angular/core';
import {FormsModule} from '@angular/forms';
import {HttpModule} from '@angular/http';

import {CustModule} from './cust/cust.module'

import {AppComponent} from './app.component';
import {HelloComponentComponent} from './hello-component/hello-
component.component';
import { BookComponentComponent } from './book-component/book-
component.component';

@NgModule({
  declarations: [
    AppComponent,
    HelloComponentComponent,
    BookComponentComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    CustModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {
}
```

BrowserModule : 브라우저와 연동하기위한 필수모듈. 컴포넌트, 지시자, 파이프 같은 구성 요소를 템플릿에 나타나게 한다. 브라우저 모듈은 CommonModule 을 재노출하여 브라우저 모듈 설정만으로 CommonModule 을 선언한 것과 같다.

```
import { CustModule } from './cust/cust.module'
```

커스텀 모듈을 임포트한다. 이제 CustComponent 의 셀렉터 cust 를 사용할 수 있다.

```
bootstrap:    [ AppComponent ]
```

기동 컴포넌트는 AppComponent 이다.

app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
```

```
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'app works!';
}
```

app.module.ts 공통모듈 설정에서

import { BookComponent } from './book.component'

위 선언으로 모듈내 어디서나 BookComponent 컴포넌트의 셀렉터 my-book 을 사용할 수 있다.

app.component.html

```
<h1>
  {{title}}
</h1>
<hr/>
<h1>Book Component</h1>
<app-book-component></app-book-component>
```

angula-cli 를 통해 만든 컴포넌트를 수정한다.

book-component.component.ts

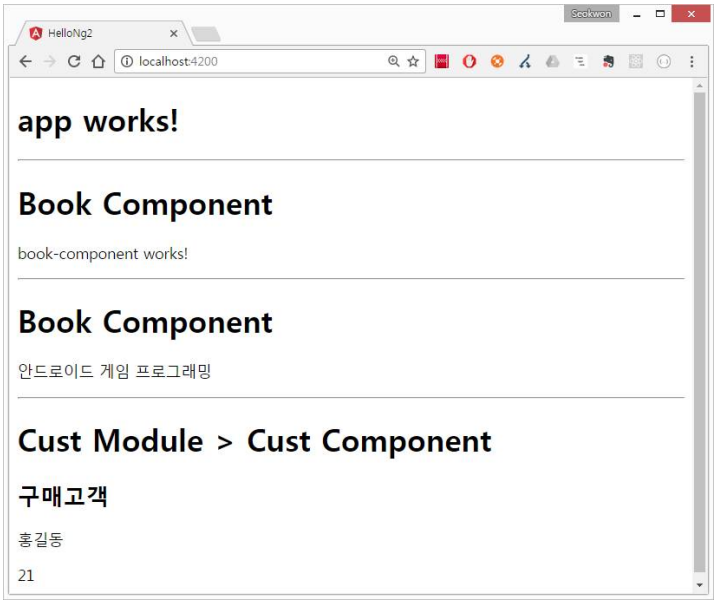
```
import {Component, OnInit} from '@angular/core';

@Component({
  selector: 'app-book-component',
  templateUrl: './book-component.component.html',
  styleUrls: ['./book-component.component.css']
})
export class BookComponentComponent implements OnInit {
  name = "안드로이드 게임 프로그래밍";

  constructor() {
  }

  ngOnInit() {
  }

}
```

book.component.html

```
<p>
  book-component works!
</p>
<hr/>
<h1>Book Component</h1>
```

```
<p>{{name}}</p>
<hr/>
<h1>Cust Module > Cust Component</h1>
<cust></cust>
```

cust.module.ts 에서 노출한 cust.component.ts 를 셀렉터 cust 로 사용한다.



앵귤러 살펴보기

hello-angular2

main.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app.module';

const platform = platformBrowserDynamic();
platform.bootstrapModule(AppModule);
```

app.module.ts

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent }   from './app.component';

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

app.component.ts

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

@Component({
  selector: 'my-app',
  template: '<h1>My First Angular App</h1>'
})
export class AppComponent { }
```

index.html

```
<html>
  <head>
    <title>Angular QuickStart</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet" href="styles.css">
    <!-- 1. Load libraries -->
```

```
    <!-- Polyfill(s) for older browsers -->
    <script src="node_modules/core-js/client/shim.min.js"></script>
    <script src="node_modules/zone.js/dist/zone.js"></script>
    <script src="node_modules/reflect-metadata/Reflect.js"></script>
    <script src="node_modules/systemjs/dist/system.src.js"></script>
    <!-- 2. Configure SystemJS -->
    <script src="systemjs.config.js"></script>
    <script>
      System.import('app').catch(function(err){ console.error(err); });
    </script>
  </head>
  <!-- 3. Display the application -->
  <body>
    <my-app>Loading...</my-app>
  </body>
</html>
```

book-promise-http

main.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';
const platform = platformBrowserDynamic();
platform.bootstrapModule(AppModule);
```

ng g component book

app.module.ts

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule }   from '@angular/forms';
import { HttpModule }    from '@angular/http';
import { AppComponent }   from './app.component';
import { BookComponent } from './book.component';
import './rxjs-extentions';

@NgModule({
  imports:      [ BrowserModule, HttpModule ],
  declarations: [ AppComponent, BookComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

app.component.ts

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

@Component({
  selector: 'my-app',
  template: `<my-book></my-book>`
})
export class AppComponent { }
```

ng g class book/book

book.ts

```
export class Book {
    id:string;
    name:string;
    price:number;
    date:string;
    img:string;
}
```

ng g service book/book

book.service.ts

```
import { Injectable } from '@angular/core';
import {Book} from './book';
import {Http} from "@angular/http";

import 'rxjs/add/operator/toPromise';

@Injectable()
export class BookService {
    constructor(private http:Http){

    }
    getBooks(): Promise<Book[]>{
        return this.http.get('./server/book.json')
                .toPromise()
                .then(res=>res.json().info.books);
    }
}
```

`this.http.get('./server/book.json')`

JSON 파일을 읽는다.

.toPromise()

프라미스객체로 전환한다. 이 메소드를 사용하기 위해서 rxjs 가 필요하다.

`import './rxjs-extentions';`

book.component.ts

```
import {Component} from '@angular/core';
import {BookService} from './book.service'
import {Book} from './book';

@Component({
    selector: 'my-book',
    templateUrl: './app/book.component.html',
    styleUrls:["./assets/stylesheets/book.css"],
    providers:[BookService]
})
export class BookComponent {
    books:Book[];   //Promise<any>;
    constructor(private bookService:BookService){
    }
    ngOnInit(){
        this.bookService.getBooks()
            .then(books=>this.books = books);
    }
}
```

this.bookService.getBooks() 메소드가 프라미스객체를 리턴한다.

ngOnInit() 컴포넌트의 초기화 작업 시 필요한 값을 구한다.

src/server/book.json

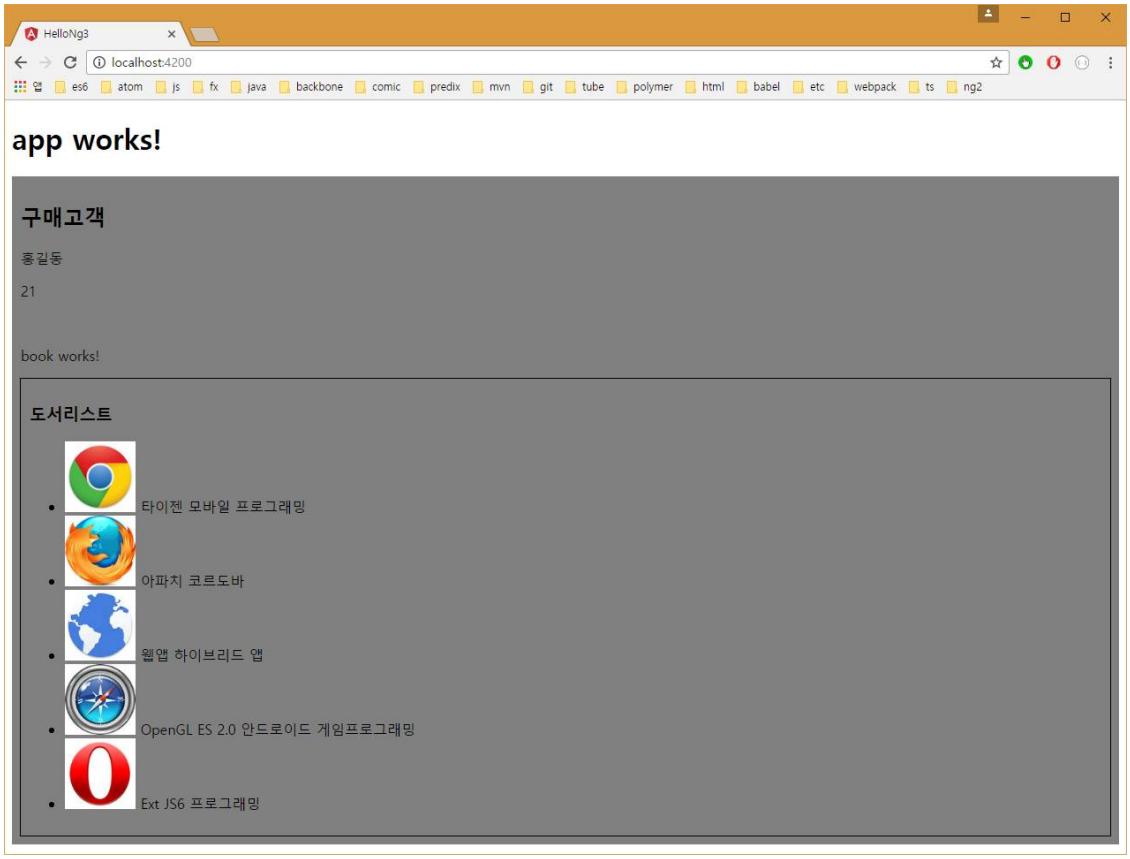
```
{
    "success":true,
    "info":{
        "books":[
            {"id": "001", "name":"타이젠 모바일 프로그래밍", "price":33000, "date":"20150115",
"img":"img001.jpg" },
            {"id": "002", "name":"아파치 코르도바", "price":33000, "date":"20150131",
"img":"img002.jpg" },
            {"id": "003", "name":"웹앱 하이브리드 앱", "price":33000, "date":"20150320",
"img":"img003.jpg" },
            {"id": "004", "name":"OpenGL ES 2.0 안드로이드 게임프로그래밍", "price":33000,
"date":"20150601", "img":"img004.jpg" },
            {"id": "005", "name":"Ext JS6 프로그래밍", "price":33000, "date":"20150115",
"img":"img005.jpg" }
        ]
    },
    "msg":"정상"
}
```

book.component.html

```
<h3>도서리스트</h3>
<ul >
    <li *ngFor="let book of books" class="font-red">
        <img src="./assets/image/{{book.img}}" height="80"> {{book.name}}
    </li>
</ul>
```

src/assets/image 폴더를 만들고 그 밑으로 img001.jpg~img005.jpg 이미지 5 개를 배치한다.

확인



book-pipe-basic

main.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';
const platform = platformBrowserDynamic();
platform.bootstrapModule(AppModule);
```

app.module.ts

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent }   from './app.component';
import { BookComponent } from './book.component';
@NgModule({
  imports:      [ BrowserModule ],
  declarations: [ AppComponent, BookComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

app.component.ts

```
import {Component} from '@angular/core';
@Component({
    selector: 'my-app',
    template:`
        <h3>문자열</h3>
        string: {{str}} <br>
        string | uppercase: {{str|uppercase}} <br>
        string | lowercase: {{str|lowercase}} <br>
        <h3>슬라이스</h3>
        slice: {{str}} <br>
        slice | slice:0:3: {{str|slice:0:3}} <br>
        slice | slice:3:5: {{str|slice:3:5}} <br>
        <h3>숫자</h3>
        number: {{num}} <br>
        number | number:'.5-10': {{num|number:'.5-10'}} <br>
        number | number:'2.1-6': {{num|number:'2.1-6'}} <br>
        number | number:'10.0-3': {{num|number:'10.0-3'}} <br>
        <h3>퍼센트</h3>
        percent | percent: {{num|percent}} <br>
        percent | percent:'2.1-6': {{num|percent:'2.1-6'}} <br>
        <h3>날짜</h3>
        date: {{date}} <br>
        date | date: {{date|date}} <br>
        date | date:'fullDate': {{date|date:"fullDate"}} <br>
        date | date:'yyyy/MM/dd': {{date|date:"yyyy/MM/dd"}} <br>
```

```
        <h3>통화</h3>
        money:{{money|currency}}<br>
        money:{{money|currency:'USD'}}<br>
        money:{{money|currency:'USD':true}}<br>
        money:{{money|currency:'KRW'}}<br>
        money:{{money|currency:'KRW':true}}<br>
        <h3>JSON</h3>
        json:{{json}}<br>
        json:{{json|json}}<br>
    `
})
export class AppComponent {
   str = "abcdEFG";
   num = 123456.123456;
   date = new Date(2017, 1, 1);
   money = 1000000;
   json = { info:{ name:'사용자 1', age:20}, list:[{ name:'사용자 1',
age:20},{ name:'사용자 2', age:20}]};
}
```

파이프는 컨텐츠를 화면에 표시하기 직전에 적용하는 데이터가공 기능이다.

book-pipe-custom

main.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';
const platform = platformBrowserDynamic();
platform.bootstrapModule(AppModule);
```

app.module.ts

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent }   from './app.component';
import { BookComponent } from './book.component';
import { MyDatePipe } from './mydate.pipe';
@NgModule({
  imports:      [ BrowserModule ],
  declarations: [ AppComponent, BookComponent, MyDatePipe ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

화면과 관련된 컴포넌트, 디렉티브, 파이프는 declarations 에 선언한다.

app.component.ts

```
import {Component} from '@angular/core';
@Component({
    selector: 'my-app',
    template:`
        <h3>커스텀</h3>
        날짜변환 : {{str|mydate:'/'}} <br>
        날짜변환 : {{str|mydate:'-'}} <br>
    `
})
export class AppComponent {
   str = "20161020";
}
```

mydate.pipe.ts

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

@Pipe({name: 'mydate'})
export class MyDatePipe implements PipeTransform {
    transform(value: string, exponent: string): string {
        if(value.length ==10){
```

```
            return value;
        }
        if(value.length == 8){
            return value.substring(0,4) + exponent +
                value.substring(4,6) +  exponent +
                value.substring(6,8);
        }
        else {
            return value;
        }
    }
}
```

book-inout-input

main.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';
const platform = platformBrowserDynamic();
platform.bootstrapModule(AppModule);
```

app.module.ts

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent }   from './app.component';
import { BookComponent } from './book.component';
import { BookImageComponent } from './book-image.component';
@NgModule({
  imports:      [ BrowserModule ],
  declarations: [ AppComponent, BookComponent, BookImageComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

app.component.ts

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

@Component({
  selector: 'my-app',
  template: `<my-book></my-book>`
})
export class AppComponent { }
```

book.component.ts

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

@Component({
    selector: 'my-book',
    templateUrl: './app/book.component.html',
    styleUrls:["./assets/stylesheets/book.css"]
})
export class BookComponent {
    books = [
       {id:'001', name:'타이젠 모바일 프로그래밍', price:31500, date:'20150115',
img:'img001.jpg'},
       {id:'002', name:'아파치 코르도바', price:31500, date:'20150131', img:'img002.jpg'},
       {id:'003', name:'웹앱 하이브리드 앱', price:31500, date:'20150320',
img:'img003.jpg'},
       {id:'004', name:'OpenGL ES 2.0 안드로이드 게임프로그래밍', price:31500,
```

```
date:'20150601', img:'img004.jpg'},
       {id:'005', name:'Ext JS6 프로그래밍', price:31500, date:'20151005',
img:'img005.jpg'},
    ]
}
```

book.component.html

```
<h3>도서리스트</h3>
<ul>
    <li *ngFor="let book of books">
        <book-image [path]="book.img" [title]="book.name" [width]="60"
[height]="'80'"></book-image>
    <li>
</ul>
```

book-image.component.ts

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

@Component({
    selector: 'book-image',
    templateUrl: './app/book-image.component.html'
})
export class BookImageComponent {
    @Input() title = "제목";
    @Input() path = "./assets/image/img001.jpg";
    @Input() width = "100";
    @Input() height = "150";
}
```

부모 컴포넌트로부터 파라미터를 전달받을 때 @Input 데코레이터를 사용한다.

book-image.component.html

```
<img src="./assets/image/{{path}}" width="{{width}}" height="{{height}}">
{{title}} <br>
```

book-inout-output

main.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';
const platform = platformBrowserDynamic();
platform.bootstrapModule(AppModule);
```

app.module.ts

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent }   from './app.component';
import { BookComponent } from './book.component';
import { BookImageComponent } from './book-image.component';
@NgModule({
  imports:      [ BrowserModule ],
  declarations: [ AppComponent, BookComponent, BookImageComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

app.component.ts

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

@Component({
  selector: 'my-app',
  template: `<my-book></my-book>`
})
export class AppComponent { }
```

book.component.ts

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

@Component({
    selector: 'my-book',
    templateUrl: './app/book.component.html',
    styleUrls:["./assets/stylesheets/book.css"]
})
export class BookComponent {
    books = [
       {id:'001', name:'타이젠 모바일 프로그래밍', price:31500, date:'20150115',
img:'img001.jpg'},
       {id:'002', name:'아파치 코르도바', price:31500, date:'20150131', img:'img002.jpg'},
       {id:'003', name:'웹앱 하이브리드 앱', price:31500, date:'20150320',
img:'img003.jpg'},
       {id:'004', name:'OpenGL ES 2.0 안드로이드 게임프로그래밍', price:31500,
```

```
date:'20150601', img:'img004.jpg'},
       {id:'005', name:'Ext JS6 프로그래밍', price:31500, date:'20151005',
img:'img005.jpg'},
    ]

    bookName:''
    onSelectBook(name){
        this.bookName = name;
    }
}
```

book.component.html

```
<h3>도서리스트</h3>
<ul *ngFor="let book of books">
    <book-image [path]="book.img" [title]="book.name" [width]="60"
[height]="'80'" (selectBook)="onSelectBook(book.name)"></book-image>
</ul>
<br>
선택된 도서: <input type="text" name="selectedBook" value="{{bookName}}">
```

@Output() selectBook 이벤트에미터객체가 이벤트를 방출하므로 (selectBook) 이벤트로 받는다.

book-image.component.ts

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

@Component({
    selector: 'book-image',
    templateUrl: './app/book-image.component.html'
})
export class BookImageComponent {
    @Input() title="제목";
    @Input() path = "./assets/image/img001.jpg";
    @Input() width = "100";
    @Input() height = "150";

    @Output() selectBook = new EventEmitter<string>();
    onClick(name){
        this.selectBook.emit(name);
    }
}
```

book-image.component.html

```
<img src="./assets/image/{{path}}" width="{{width}}" height="{{height}}"><a
(click)="onClick(title)"> {{title}} </a><br>
```

book-form-basic

main.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';
const platform = platformBrowserDynamic();
platform.bootstrapModule(AppModule);
```

app.module.ts

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule }   from '@angular/forms';
import { AppComponent }   from './app.component';
import { BookComponent } from './book.component';
import {} from '@angular/forms';

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

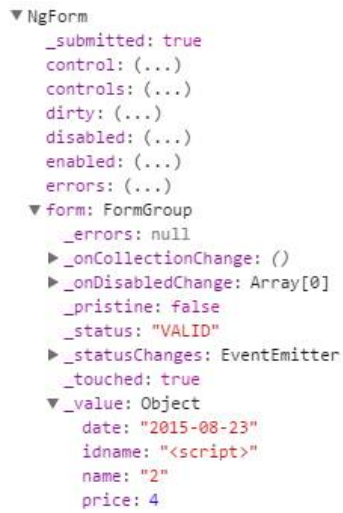
app.componennt.ts

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

@Component({
  selector: 'my-app',
  template: `<my-book></my-book>`
})
export class AppComponent { }
```

book.component.ts

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

@Component({
    selector: 'my-book',
    templateUrl: './app/book.component.html',
    styleUrls:["./assets/stylesheets/book.css"]
})
export class BookComponent {
    onSubmit(f){
        console.log(f);
        console.log(JSON.stringify(f.form._value));
         // {"idname":"1","name":"2","price":3}
```

```
        debugger;
    }
}
```



book.component.html

#변수 : 템플릿내에서 사용하는 참조

ngModel : 클래스에서 사용할 대상, 서밋 할 경우 값을 전달한다.

#f="ngForm" : 서밋 될 경우 ngForm 으로 선언한 폼을 파라미터로 전달한다. ngForm 객체에서 ngModel 로 선언한 값을 읽을 수 있다.

```
<h3>도서등록</h3>
<form #f="ngForm" (ngSubmit)="onSubmit(f)">
  <label for="id" style="width:100px;display:inline-block">아이디: </label>
  <input type="text" name="idname" ngModel #id required><br>
  <label for="name" style="width:100px;display:inline-block">도서명: </label>
  <input type="text" name="name" ngModel #name required><br>
  <label for="price" style="width:100px;display:inline-block">가격: </label>
  <input type="number" name="price" ngModel #price required><br>
  <label for="date" style="width:100px;display:inline-block">날짜: </label>
  <input type="date" name="date" #date><br>
  <br>
  <button type="submit">등록</button>
  <br>
  <br>
  <div style="background-color:gray">
    <div [innerText]="'bookId:' + id.value"></div>
    <div [innerText]="'name:' + name.value"></div>
    <div [innerText]="'price:' + price.value"></div>
    <div [innerText]="'date:' + date.value"></div>
  </div>
</form>
```

book-form-valid

app.module.ts

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule }   from '@angular/forms';
import { AppComponent }   from './app.component';
import { BookComponent } from './book.component';

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

app.component.ts

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

@Component({
  selector: 'my-app',
  template: `<my-book></my-book>`
})
export class AppComponent { }
```

book.component.ts

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

@Component({
    selector: 'my-book',
    templateUrl: './app/book.component.html',
    styleUrls:["./assets/stylesheets/book.css"]
})
export class BookComponent {
    onSubmit(f){
        console.log(f);
        debugger; // 디버거를 실행하면 프로파게이션을 막고 있다.
    }
}
```

book.component.html

[hidden]="*id*.valid" : id 가 유효하면 true 로 hidden 한다. *ngIf="!id.valid" 와 같다.

처음에 작동하지 않기위하여 보통 *ngIf="id.touched"와 함께 사용한다.

```
<h3>도서등록</h3>
<form #f="ngForm" (ngSubmit)="onSubmit(f)">
    <label for="id" style="width:100px;display:inline-block" >아이디: </label>
    <input type="text" name="id" ngModel #id="ngModel" required><br>
    <div [hidden]="id.valid" style="background-color:red;color:white">
        아이디를 입력하세요.
    </div>
    <label for="name" style="width:100px;display:inline-block" >도서명:
</label>
    <input type="text" name="name" ngModel #name="ngModel" required><br>
    <div [hidden]="name.valid" style="background-color:red;color:white">
        도서명을 입력하세요.
    </div>
    <label for="price" style="width:100px;display:inline-block" >가격:
</label>
    <input type="number" name="price" ngModel #price="ngModel" required><br>
    <div [hidden]="price.valid" style="background-color:red;color:white">
        가격을 입력하세요.
    </div>
    <label for="date" style="width:100px;display:inline-block" >날짜: </label>
    <input type="date" name="date" ngModel #date="ngModel" required><br>
    <div [hidden]="date.valid" style="background-color:red;color:white">
        날짜를 입력하세요.
    </div>
    <br>
    <button type="submit">등록</button><br>
    <br>
    <div style="background-color:gray">
    <div [innerText]="'id:' + id.value "></div>
    <div [innerText]="'name:' + name.value"></div>
    <div [innerText]="'price:' + price.value"></div>
    <div [innerText]="'date:' + date.value"></div>
</div>
```

book-form-control

app.module.ts

ReactiveFormsModule : 템플릿과 클래스간 엘리먼트를 공유하기 위해 필요하다.

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule, ReactiveFormsModule }   from '@angular/forms';
import { AppComponent }   from './app.component';
import { BookComponent } from './book.component';
 import {} from '@angular/forms';

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

book.component.ts

FormControl(초기값, 유효성 설정) : 초기값과 필수항목여부를 클래스에 정의한다.

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

@Component({
    selector: 'my-book',
    templateUrl: './app/book.component.html',
    styleUrls:["./assets/stylesheets/book.css"]
})
export class BookComponent {
    bookForm:FormGroup = new FormGroup({
        id: new FormControl('004', Validators.required),
        name: new FormControl('안드로이드 게임 프로그래밍', Validators.required),
        price:new FormControl('', Validators.required),
        date:new FormControl('', Validators.required)
    })

    constructor(private _formBuilder:FormBuilder){
        /*
        this.bookForm= _formBuilder.group({
            id: new FormControl('111', Validators.required)
            // name: new FormControl('', Validators.required),
            //price:new FormControl('', Validators.required),
            //date:new FormControl('', Validators.required)
        });
        */
    }
```

```
    ngOnInit(){

    }
    onSubmit(f){
        console.log(f);
        console.log(this.bookForm);
        debugger;
    }
}
```

```
id: new FormControl('004', Validators.required)
```

초기값으로 004 를 사용한다. 필수입력항목으로 설정한다.

book.component.html

템플릿에서도 formGroup 을 추가한다. ngModel 대신 formControlName 로 맵핑한다.

*ngIf="bookForm.controls['id'].touched", [hidden]="bookForm.controls['id'].valid" 으로 설정한다.

```
<h3>도서등록</h3>
<form [formGroup]="bookForm" #f (ngSubmit)="onSubmit(f)">
    <label for="id" style="width:100px;display:inline-block" >아이디: </label>
    <input type="text" name="id" formControlName="id" ><br>
    <div *ngIf="bookForm.controls['id'].touched"  [hid-
den]="bookForm.controls['id'].valid" style="background-color:red;color:white">
        아이디를 입력하세요.
    </div>
    <label for="name" style="width:100px;display:inline-block" >도서명:
</label>
    <input type="text" name="name" formControlName="name" ><br>
    <div *ngIf="bookForm.controls['name'].touched"  [hid-
den]="bookForm.controls['name'].valid" style="background-
color:red;color:white">
        도서명을 입력하세요.
    </div>
    <label for="price" style="width:100px;display:inline-block" >가격:
</label>
    <input type="number" name="price" formControlName="price"><br>
    <div *ngIf="bookForm.controls['price'].touched"  [hid-
den]="bookForm.controls['price'].valid" style="background-
color:red;color:white">
        가격을 입력하세요.
    </div>
    <label for="date" style="width:100px;display:inline-block" >날짜: </label>
    <input type="date" name="date" formControlName="date"><br>
    <div *ngIf="bookForm.controls['date'].touched"  [hid-
den]="bookForm.controls['date'].valid" style="background-
color:red;color:white">
        날짜를 입력하세요.
    </div>
```

```
    <br>
    <button type="submit">등록</button><br>
    <br>
    <div style="background-color:gray">
        <div [innerText]="'id:' + bookForm.controls['id'].value "></div>
        <div [innerText]="'name:' + bookForm.controls['name'].value"></div>
        <div [innerText]="'price:' + bookForm.controls['price'].value"></div>
        <div [innerText]="'date:' + bookForm.controls['date'].value"></div>
    </div>
</form>
```

위 코드에서 하단 부분을 다음과 같이 바꾸고 다시 테스트 해 보자

```
<button type="submit" [disabled]="!bookForm.valid">등록</button><br>
<br>
<div style="background-color:gray">
    <div [innerText]="'id:' + bookForm.controls['id'].value "></div>
    <div [innerText]="'name:' + bookForm.controls['name'].value"></div>
    <div [innerText]="'price:' + bookForm.controls['price'].value"></div>
    <div [innerText]="'date:' + bookForm.controls['date'].value"></div>
    <div>bookForm.value : {{bookForm.value | json}}</div>
    <div>bookForm.status : {{bookForm.status}}</div>
    <div>bookForm.valid : {{bookForm.valid}}</div>
</div>
```

폼 작성 정보가 유효하지 않으면 등록 버튼이 비활성화 상태가 된다.

book-form-formbuilder

app.module.ts

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule, ReactiveFormsModule }   from '@angular/forms';
import { AppComponent }   from './app.component';
import { BookComponent } from './book.component';
import {} from '@angular/forms';

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

app.component.ts

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

@Component({
  selector: 'my-app',
  template: `<my-book></my-book>`
})
export class AppComponent { }
```

book.component.ts

FormBuilder 는 FormGroup 과 유사하다.

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

@Component({
  selector: 'my-book',
  templateUrl: './app/book.component.html',
  styleUrls: ["./assets/stylesheets/book.css"]
})
export class BookComponent {
  bookForm: FormGroup;

  constructor(private _formBuilder: FormBuilder) {
    this.bookForm = _formBuilder.group({
      id: new FormControl('004', Validators.required),
      name: new FormControl('안드로이드 게임 프로그래밍', Validators.required),
      price: new FormControl('', Validators.required),
      date: new FormControl('', Validators.required)
```

```
    });
  }

  onSubmit(f) {
    console.log(f);
  }
}
```

Validators.required 대신에 Validators.compose 메소드를 사용하여 조건을 복합적으로 적용할 수 있다.

- Validators.compose([Validators.required, Validators.pattern('[a-zA-Z]{3}')])
- Validators.compose([Validators.minLength(3), Validators.maxLength(5))

book-directive-property

A @Component requires a view whereas a @Directive does not.

A component creates it's own view with attached behaviour.

Directives add behaviour to an existing DOM element.

app.module.ts

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule, ReactiveFormsModule }   from '@angular/forms';
import { AppComponent }   from './app.component';
import { BookComponent } from './book.component';
import { TextSizeDirective } from './text-size.directive'

@NgModule({
  imports:      [ BrowserModule, FormsModule, ReactiveFormsModule ],
  declarations: [ AppComponent, BookComponent, TextSizeDirective ],
  bootstrap:    [ AppComponent]
})
export class AppModule { }
```

text-size.directive.ts

Renderer 객체를 사용하여 화면에 출력한다. ElementRef 모듈은 DOM 에 직접 접근하여 처리한다. Renderer 방식은 관련된 부분에만 접근하므로 보다 안전하고 좋은 방법이다.

selector 로 정의한 이름을 속성명으로 사용한다.

```
import {Directive, ElementRef, Renderer} from '@angular/core'

@Directive({
    selector:'[text-size]'
})
export class TextSizeDirective {
    constructor(private el:ElementRef, private renderer:Renderer){
        this.renderer.setElementAttribute(this.el.nativeElement, 'size',
'10');
    }
}
```

selector 를 정의할 때 대괄호를 생략할 수 있다. 대괄호를 붙이면 속성 바인딩 시 값을 할당받을 수 있다. 속성 값을 받아들이려면 @Input 데코레이터를 사용한다.

```
@Input('text-size') textSize: number;
```

app.component.ts

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

@Component({
  selector: 'my-app',
  template: `<my-book></my-book>`
})
export class AppComponent { }
```

book.component.ts

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

@Component({
    selector: 'my-book',
    templateUrl: './app/book.component.html'
})

export class BookComponent {

}
```

book.component.html

```
<h1>도서</h1>
<p>{{name}}</p>
<input type="text" text-size value="웹앱 하이브리드앱"><br><br>
<input type="text" value="아파치 코르도바"><br>
```

book-directive-event

app.module.ts

```
import { NgModule }      from '@angular/core'
import { BrowserModule } from '@angular/platform-browser'
import { FormsModule, ReactiveFormsModule }   from '@angular/forms'
import { AppComponent }   from './app.component'
import { BookComponent } from './book.component'
import {TextColorDirective} from './text-color.directive'

@NgModule({
  imports:      [ BrowserModule, FormsModule, ReactiveFormsModule ],
  declarations: [ AppComponent, BookComponent, TextColorDirective ],
  bootstrap:    [ AppComponent]
})
export class AppModule { }
```

text-color.directive.ts

@HostListener 를 사용하여 이벤트를 정의한다.

ElementRef : 엘리먼트를 참조하기 위해 생성자의 파라미터로 DI 받는다.

```
import {Directive, ElementRef, Renderer, HostListener} from '@angular/core'

@Directive({
    selector:'[text-color]',
})
export class TextColorDirective {
    constructor(private el:ElementRef, private renderer:Renderer){
    }
    @HostListener('focus')  onFocus(){
        this.renderer.setElementStyle(
            this.el.nativeElement,
            'background',
            'yellow');
    }
    @HostListener('blur') onBlur(){
        this.renderer.setElementStyle(
            this.el.nativeElement,
            'background',
            'white');
        console.log(this.el);
    }
}
```

디렉티브 데코레이터에서 host 설정을 통해 이벤트 연동을 정의할 수도 있다.

```
host: {
      '(focus)' : 'onFocus()',
      '(blur)':'onBlur()'
}
```

app.component.ts

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

@Component({
  selector: 'my-app',
  template: `<my-book></my-book>`
})
export class AppComponent { }
```

book-component.ts

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

@Component({
    selector: 'my-book',
    templateUrl: './app/book.component.html',
    encapsulation: ViewEncapsulation.Native
})
export class BookComponent {
}
```

ViewEncapsulation

http://blog.thoughtram.io/angular/2015/06/29/shadow-dom-strategies-in-angular2.html

`encapsulation: ViewEncapsulation.Native`

If we run our code in the browser, we see that no styles are written to the document head anymore. However, styles do now end up in the component's template inside the shadow root.

book.component.html

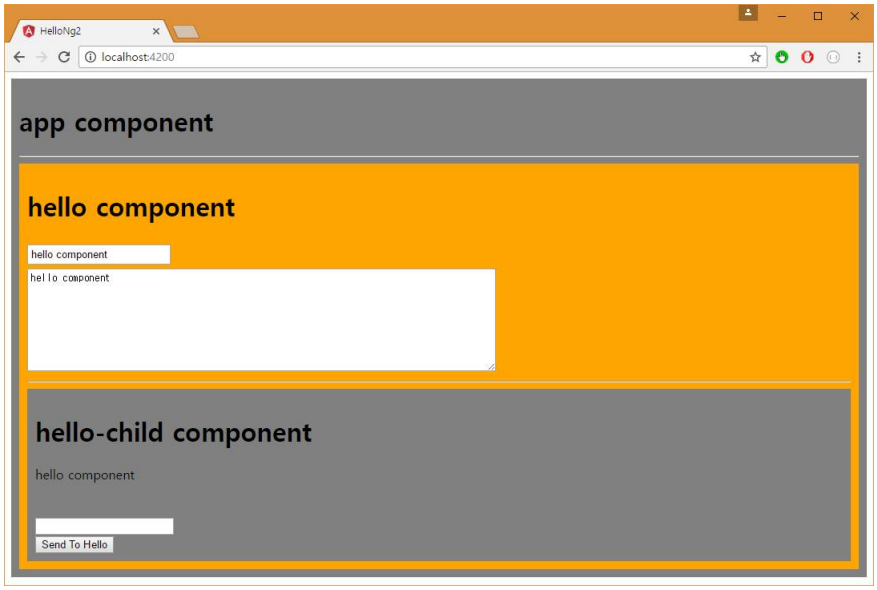
```
<h1>Book</h1>
<p>{{name}}</p>
<input type="text" size="30" text-color value="웹앱 하이브리드앱"><br><br>
<input type="text" size="30" text-color value="아파치 코르도바"><br>
```

실습 1 : Components Composition

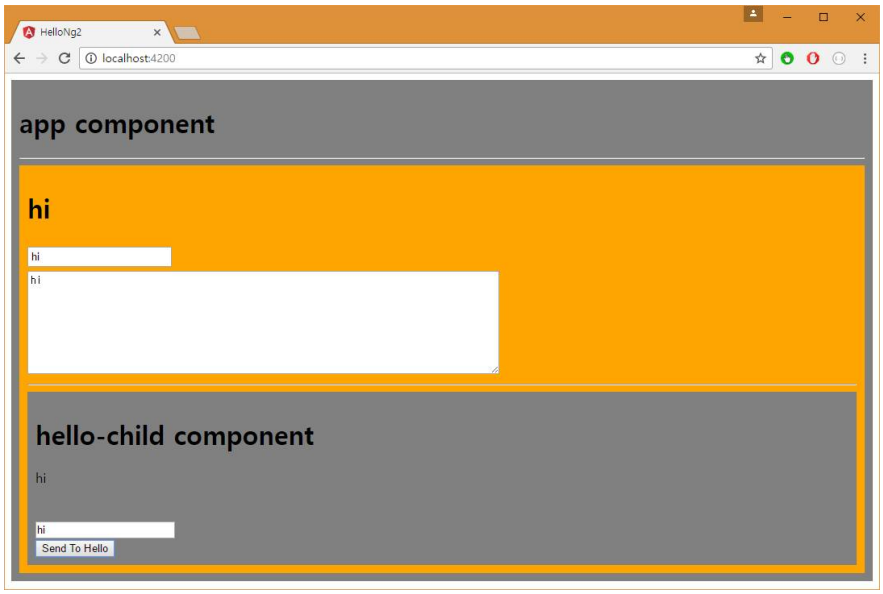
다음 화면을 참고하여 컴포넌트를 구성합니다.

app.component.ts : 기동 컴포넌트, Grand Parent
    hello.component.ts : Parent
        hello.child.component.ts : Child

Send data from Parent → Child



Send data from Child → Parent



실습 2 : Registration and Login

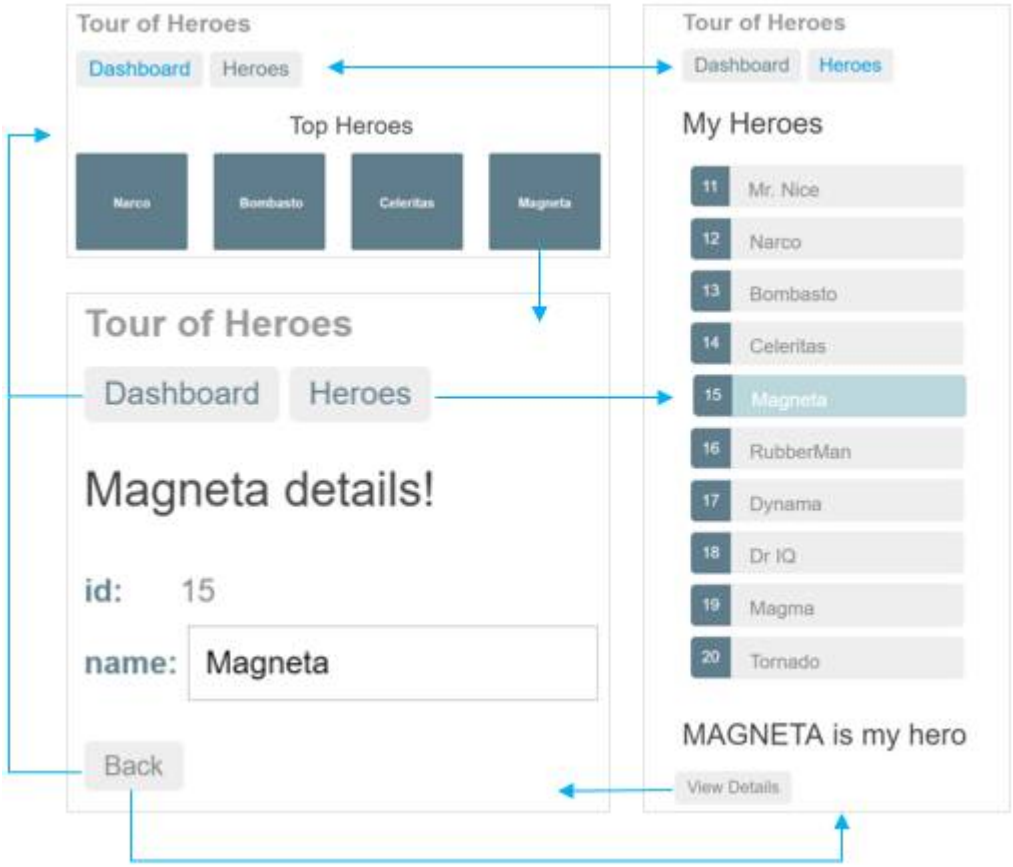
http://jasonwatmore.com/post/2016/09/29/angular-2-user-registration-and-login-example-tutorial

```
git clone https://github.com/cornflourblue/angular2-registration-login-example.git
cd angular2-registration-login-example
npm install
```

실습 3 : Tour of Heroes Tutorial

https://angular.io/docs/ts/latest/tutorial/

앱 디자인 미리보기



설정 샘플 프로젝트 살펴보기

https://angular.io/docs/ts/latest/guide/setup.html

```
git clone https://github.com/angular/quickstart.git quickstart
cd quickstart
npm install
npm start
```

Chapter 1. 간단한 앱

angular-tour-of-heroes 프로젝트 구조

```
angular-tour-of-heroes
    app
        app.component.ts
        app.module.ts
        main.ts
    node_modules
    index.html
    package.json
    styles.css
    systemjs.config.js
    tsconfig.json
```

프로젝트 설정파일

앞서 작업한 quickstart 프로젝트에서 설정파일을 복사해서 사용한다.

- package.json
- systemjs.config.js
- tsconfig.json

디펜던시 설치

```
npm install
```

package.json

```
{
  "name": "angular-quickstart",
  "version": "1.0.0",
  "description": "QuickStart package.json from the documentation, supplemented
with testing support",
  "scripts": {
    "start": "tsc && concurrently \"tsc -w\" \"lite-server\" ",
    "e2e": "tsc && concurrently \"http-server -s\" \"protractor protrac-
tor.config.js\" --kill-others --success first",
    "lint": "tslint ./app/**/*.ts -t verbose",
    "lite": "lite-server",
    "pree2e": "webdriver-manager update",
    "test": "tsc && concurrently \"tsc -w\" \"karma start karma.conf.js\"",
    "test-once": "tsc && karma start karma.conf.js --single-run",
    "tsc": "tsc",
    "tsc:w": "tsc -w"
  },
  "keywords": [],
  "author": "",
  "license": "MIT",
  "dependencies": {
    "@angular/common": "~2.3.0",
    "@angular/compiler": "~2.3.0",
    "@angular/core": "~2.3.0",
    "@angular/forms": "~2.3.0",
    "@angular/http": "~2.3.0",
    "@angular/platform-browser": "~2.3.0",
    "@angular/platform-browser-dynamic": "~2.3.0",
    "@angular/router": "~3.3.0",

    "angular-in-memory-web-api": "~0.2.0",
    "systemjs": "0.19.40",
    "core-js": "^2.4.1",
    "reflect-metadata": "^0.1.8",
    "rxjs": "5.0.0-rc.4",
    "zone.js": "^0.7.2"
  },
  "devDependencies": {
    "concurrently": "^3.1.0",
    "lite-server": "^2.2.2",
    "typescript": "~2.0.10",

    "canonical-path": "0.0.2",
    "http-server": "^0.9.0",
    "tslint": "^3.15.1",
    "lodash": "^4.16.4",
    "jasmine-core": "~2.4.1",
    "karma": "^1.3.0",
    "karma-chrome-launcher": "^2.0.0",
    "karma-cli": "^1.0.1",
    "karma-jasmine": "^1.0.2",
    "karma-jasmine-html-reporter": "^0.2.2",
```

```
    "protractor": "~4.0.13",
    "rimraf": "^2.5.4",

    "@types/node": "^6.0.46",
    "@types/jasmine": "^2.5.36",
    "@types/selenium-webdriver": "^2.53.33"
  },
  "repository": {}
}
```

systemjs.config.js

```
/**
 * System configuration for Angular samples
 * Adjust as necessary for your application needs.
 */
(function (global) {
  System.config({
    paths: {
      // paths serve as alias
      'npm:': 'node_modules/'
    },
    // map tells the System loader where to look for things
    map: {
      // our app is within the app folder
      app: 'app',

      // angular bundles
      '@angular/core': 'npm:@angular/core/bundles/core.umd.js',
      '@angular/common': 'npm:@angular/common/bundles/common.umd.js',
      '@angular/compiler': 'npm:@angular/compiler/bundles/compiler.umd.js',
      '@angular/platform-browser': 'npm:@angular/platform-
browser/bundles/platform-browser.umd.js',
      '@angular/platform-browser-dynamic': 'npm:@angular/platform-browser-
dynamic/bundles/platform-browser-dynamic.umd.js',
      '@angular/http': 'npm:@angular/http/bundles/http.umd.js',
      '@angular/router': 'npm:@angular/router/bundles/router.umd.js',
      '@angular/forms': 'npm:@angular/forms/bundles/forms.umd.js',

      // other libraries
      'rxjs':                      'npm:rxjs',
      'angular-in-memory-web-api': 'npm:angular-in-memory-web-api/bundles/in-
memory-web-api.umd.js'
    },
    // packages tells the System loader how to load when no filename and/or no
extension
    packages: {
      app: {
        main: './main.js',
        defaultExtension: 'js'
```

```
      },
      rxjs: {
        defaultExtension: 'js'
      }
    }
  });
})(this);
```

tsconfig.json

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "moduleResolution": "node",
    "sourceMap": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "lib": [ "es2015", "dom" ],
    "noImplicitAny": true,
    "suppressImplicitAnyIndexErrors": true
  }
}
```

파일 생성

https://angular.io/resources/live-examples/toh-1/ts/eplnkr.html

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Angular Tour of Heroes</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet" href="styles.css">

    <!-- Polyfills for older browsers -->
    <script src="https://unpkg.com/core-js/client/shim.min.js"></script>

    <script src="https://unpkg.com/zone.js@0.7.2?main=browser"></script>
    <script src="https://unpkg.com/reflect-metadata@0.1.8"></script>
    <script src="https://unpkg.com/systemjs@0.19.39/dist/system.src.js"></script>

    <script
src="https://cdn.rawgit.com/angular/angular.io/b3c65a9/public/docs/_examples/_boilerplate/systemjs.config.web.js"></script>
    <script>
      System.import('app').catch(function(err){ console.error(err); });
    </script>
  </head>

  <body>
    <my-app>Loading...</my-app>
  </body>
</html>
```

style.css

```
/* Master 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[text], button {
   color: #888;
   font-family: Cambria, Georgia;
}
a {
   cursor: pointer;
   cursor: hand;
}
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;
}

/* Navigation link styles */
nav a {
    padding: 5px 10px;
    text-decoration: none;
    margin-right: 10px;
    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.active {
    color: #039be5;
}

/* items class */
.items {
    margin: 0 0 2em 0;
    list-style-type: none;
```

```
    padding: 0;
    width: 24em;
}
.items li {
    cursor: pointer;
    position: relative;
    left: 0;
    background-color: #EEE;
    margin: .5em;
    padding: .3em 0;
    height: 1.6em;
    border-radius: 4px;
}
.items li:hover {
    color: #607D8B;
    background-color: #DDD;
    left: .1em;
}
.items li.selected {
    background-color: #CFD8DC;
    color: white;
}
.items li.selected:hover {
    background-color: #BBD8DC;
}
.items .text {
    position: relative;
    top: -3px;
}
.items .badge {
    display: inline-block;
    font-size: small;
    color: white;
```

```
  padding: 0.8em 0.7em 0 0.7em;
  background-color: #607D8B;
  line-height: 1em;
  position: relative;
  left: -1px;
  top: -4px;
  height: 1.8em;
  margin-right: .8em;
  border-radius: 4px 0 0 4px;
}
/* everywhere else */
* {
  font-family: Arial, Helvetica, sans-serif;
}
```

app/main.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';

platformBrowserDynamic().bootstrapModule(AppModule);
```

app/app.module.ts

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule }   from '@angular/forms';

import { AppComponent }  from './app.component';

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

Before we can use two-way data binding for form inputs, we need to import the **FormsModule** package in our Angular module. We add it to the NgModule decorator's **imports** array. This array contains the **list of external modules** used by our application. Now we have included the forms package which includes **ngModel**.

Let's update the template to use the ngModel built-in directive for two-way binding.

app/app.component.ts

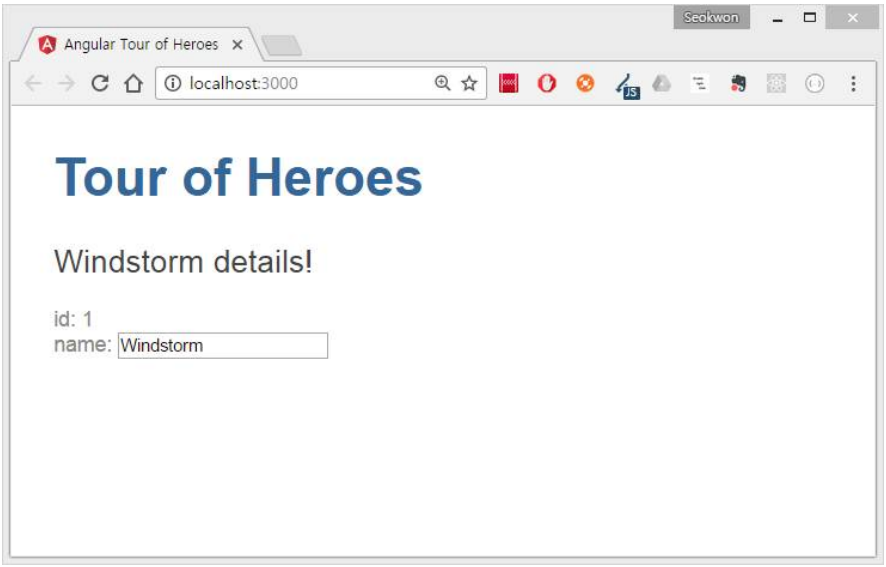
```
import { Component } from '@angular/core';


export class Hero {
  id: number;
  name: string;
}


@Component({
  selector: 'my-app',
  template: `
    <h1>{{title}}</h1>
    <h2>{{hero.name}} details!</h2>
    <div><label>id: </label>{{hero.id}}</div>
    <div>
      <label>name: </label>
      <input [(ngModel)]="hero.name" placeholder="name">
    </div>
    `
})
export class AppComponent {
  title = 'Tour of Heroes';
  hero: Hero = {
    id: 1,
    name: 'Windstorm'
  };
}
```

테스트 1

온라인 확인 > https://angular.io/resources/live-examples/toh-1/ts/eplnkr.html

로컬 확인

```
npm start
```



name 정보를 바꾸면 <h2> 태그안에 값도 갱신된다.

Chapter 2. 업그레이드: Master/Detail

angular-tour-of-heroes 프로젝트를 복사하여 angular-tour-of-heroes2 프로젝트를 만든다.

app/app.component.ts

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

export class Hero {
  id: number;
  name: string;
}

const HEROES: Hero[] = [
  { id: 11, name: 'Mr. 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' }
];

@Component({
  selector: 'my-app',
  template: `
    <h1>{{title}}</h1>
    <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>
    <div *ngIf="selectedHero">
      <h2>{{selectedHero.name}} details!</h2>
      <div><label>id: </label>{{selectedHero.id}}</div>
      <div>
        <label>name: </label>
        <input [(ngModel)]="selectedHero.name" placeholder="name"/>
      </div>
    </div>
  `,
  styles: [`
    .selected {
      background-color: #CFD8DC !important;
      color: white;
    }
```

```
    .heroes {
      margin: 0 0 2em 0;
      list-style-type: none;
      padding: 0;
      width: 15em;
    }
    .heroes li {
      cursor: pointer;
      position: relative;
      left: 0;
      background-color: #EEE;
      margin: .5em;
      padding: .3em 0;
      height: 1.6em;
      border-radius: 4px;
    }
    .heroes li.selected:hover {
      background-color: #BBD8DC !important;
      color: white;
    }
    .heroes li:hover {
      color: #607D8B;
      background-color: #DDD;
      left: .1em;
    }
    .heroes .text {
      position: relative;
      top: -3px;
    }
    .heroes .badge {
      display: inline-block;
      font-size: small;
      color: white;
      padding: 0.8em 0.7em 0 0.7em;
      background-color: #607D8B;
      line-height: 1em;
      position: relative;
      left: -1px;
      top: -4px;
      height: 1.8em;
      margin-right: .8em;
      border-radius: 4px 0 0 4px;
    }
  `]
})
export class AppComponent {
  title = 'Tour of Heroes';
  heroes = HEROES;
  selectedHero: Hero;

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

Displaying heroes in a template

The HEROES array is of type Hero, the class defined in part one, to create an array of heroes.

The leading asterisk (*) in front of ngFor is a critical part of this syntax.

■ The (*) prefix to ngFor indicates

that the <li> element and its children constitute a master template.

■ The ngFor directive iterates over the heroes array

returned by the AppComponent.heroes property and stamps out instances of this template.

■ The quoted text assigned to ngFor means

“take each hero in the heroes array, store it in the local hero variable,

and make it available to the corresponding template instance”.

■ The let keyword before "hero" identifies hero as a template input variable.

We can reference this variable within the template to access a hero’s properties.

Selecting a Hero

We have a list of heroes and we have a single hero displayed in our app. The list and the single hero are not connected in any way. We want the **user to select a hero from our list**, and have the **selected hero appear in the details view**. This UI pattern is widely known as "master-detail". In our case, the **master is the heroes list** and the **detail is the selected hero**.

Let’s connect the master to the detail through a selectedHero component property bound to a click event.

```
(click)="onSelect(hero)"
```

The parenthesis identify the <li> element’s click event as the target. The expression to the right of the equal sign **calls the AppComponent method**, **onSelect()**, **passing the template input variable hero as an argument**. That’s the same hero variable we defined previously in the ngFor.

Add the click handler

```
selectedHero: Hero;

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

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

Hide the empty detail with ngIf

We wrap the HTML hero detail content of our template with a . Then we add the **ngIf built-in directive** and set it to the selectedHero property of our component.

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


</div>
```

When there is no selectedHero, the ngIf directive removes the hero detail HTML from the DOM. There will be no hero detail elements and no bindings to worry about.

When the user picks a hero, selectedHero becomes "truthy" and ngIf puts the hero detail content into the DOM and evaluates the nested bindings.

Styling the selection

We'll add a property binding on class for the selected class to the template. We'll set this to an **expression** that compares the current selectedHero to the hero.

The **key is the name of the CSS class** (selected). The value is true if the two heroes match and false otherwise. We're saying "**apply the selected class if the heroes match**, remove it if they don't".

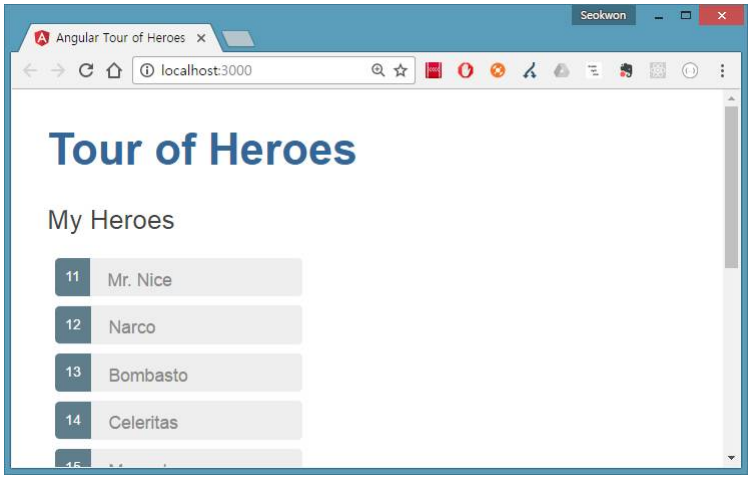
```
[class.selected]="hero === selectedHero"
```

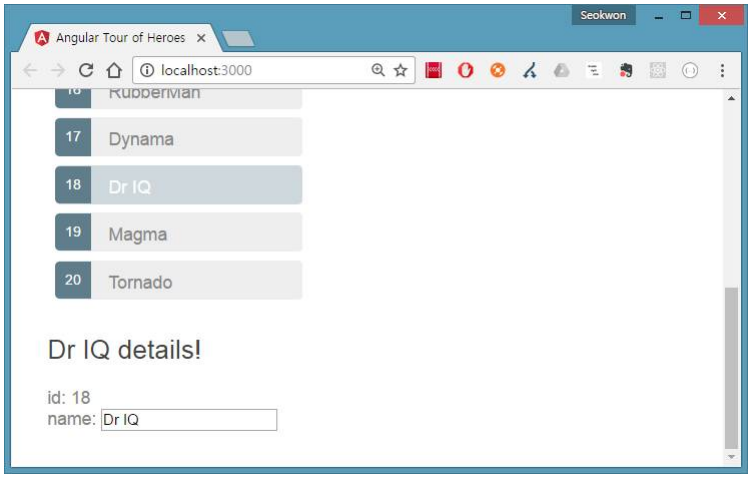
Notice in the template that the class.selected is surrounded in **square brackets** ([]). This is the **syntax for a property binding**, a binding in which data flows one way from the data source (the expression hero === selectedHero) to a property of class.

테스트 2

온라인 확인 > https://angular.io/resources/live-examples/toh-2/ts/eplnkr.html

로컬 확인





Chapter 3. 업그레이드: MULTIPLE COMPONENTS

angular-tour-of-heroes2 프로젝트를 복사하여 angular-tour-of-heroes3 프로젝트를 만든다.

Our app is growing. Use cases are flowing in for reusing components, passing data to components, and creating more reusable assets. Let's separate the heroes list from the hero details and **make the details component reusable**.

Making a Hero Detail Component

Our heroes list and our hero details are in the same component in the same file.
They're small now but each could grow. We are sure to receive new requirements for one and not the other. Yet every change puts both components at risk and doubles the testing burden without benefit. If we had to **reuse the hero details elsewhere in our app**, the heroes list would tag along for the ride.

Our current component violates the **Single Responsibility Principle**. It's only a tutorial but we can still do things right — especially if doing them right is easy and we learn how to build Angular apps in the process.

hero-detail.component.ts

Add a new file named hero-detail.component.ts **to the app folder** and create HeroDetailComponent as follows.

```
import { Component, Input } from '@angular/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>
```

```
  `
})
export class HeroDetailComponent {
  @Input()
  hero: Hero;
}
```

Naming conventions

We like to identify at a glance which classes are components and which files contain components.

Notice that we have an **AppComponent** in a file named **app.component.ts**
and our new **HeroDetailComponent** is in a file named **hero-detail.component.ts**.

All of our component names end in "Component".
All of our component file names end in ".component".

We spell our file names in **lower dash case** (AKA **kebab-case**) so we don't worry about case sensitivity on the server or in source control.

When we finish here, we'll import it into AppComponent and create a corresponding <my-hero-detail> element.

We previously bound to the selectedHero.name property of the AppComponent. Our HeroDetailComponent will have a hero property, not a selectedHero property. So we replace selectedHero with hero everywhere in our new template. That's our only change.

Let's add that hero property we were talking about to the component class.

THE HERO PROPERTY IS AN INPUT

The HeroDetailComponent must be told what hero to display. Who will tell it?
The parent AppComponent!

The AppComponent knows which hero to show: the hero that the user selected from the list.
The user's selection is in its selectedHero property.

We will soon update the AppComponent template so that it binds its selectedHero property to the hero property of our HeroDetailComponent. The binding might look like this:

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

Notice that the **hero property is the target of a property binding** — it's in square brackets to the left of the (=).

Angular insists that we **declare a target property to be an input property**.
If we don't, Angular rejects the binding and throws an error.

There are a couple of ways we can declare that hero is an input. We'll do it the way we prefer, by annotating the hero property with the @Input decorator that we imported earlier.

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

hero.ts

We declared the hero property as type Hero but our Hero class is over in the app.component.ts file. We have **two components**, each in their own file, that **need to reference the Hero class**.

We solve the problem by relocating the Hero class from app.component.ts to its own hero.ts file.

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

We **export the Hero class** from hero.ts because we'll **need to reference** it in both component files. Add the following import statement near the top of both **app.component.ts** and **hero-detail.component.ts**.

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

app.module.ts

We return to the AppModule, the application's root module, and teach it to use the HeroDetail-Component.

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule }   from '@angular/forms';

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

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

Then we add HeroDetailComponent to the **NgModule decorator's declarations array**. This array contains the list of all **components**, **pipes**, and **directives** that we created and that belong in our application's module.

Now that the application knows about our HeroDetailComponent.

app.component.ts

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

import { Hero } from './hero';

const HEROES: Hero[] = [
  { id: 11, name: 'Mr. 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' }
];

@Component({
  selector: 'my-app',
  template: `
    <h1>{{title}}</h1>
    <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>
    <my-hero-detail [hero]="selectedHero"></my-hero-detail>
  `,
  styles: [`
    .selected {
      background-color: #CFD8DC !important;
      color: white;
    }
    .heroes {
      margin: 0 0 2em 0;
      list-style-type: none;
      padding: 0;
      width: 15em;
    }
    .heroes li {
      cursor: pointer;
      position: relative;
      left: 0;
      background-color: #EEE;
      margin: .5em;
      padding: .3em 0;
      height: 1.6em;
      border-radius: 4px;
    }
```

```
    .heroes li.selected:hover {
      background-color: #BBD8DC !important;
      color: white;
    }
    .heroes li:hover {
      color: #607D8B;
      background-color: #DDD;
      left: .1em;
    }
    .heroes .text {
      position: relative;
      top: -3px;
    }
    .heroes .badge {
      display: inline-block;
      font-size: small;
      color: white;
      padding: 0.8em 0.7em 0 0.7em;
      background-color: #607D8B;
      line-height: 1em;
      position: relative;
      left: -1px;
      top: -4px;
      height: 1.8em;
      margin-right: .8em;
      border-radius: 4px 0 0 4px;
    }
  `]
})
export class AppComponent {
  title = 'Tour of Heroes';
  heroes = HEROES;
  selectedHero: Hero;

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

In the AppComponent template where we removed the Hero Detail content and add an element tag that represents the HeroDetailComponent.

The two components won't coordinate until we bind the selectedHero property of the AppComponent to the HeroDetailComponent element's hero property like this:

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

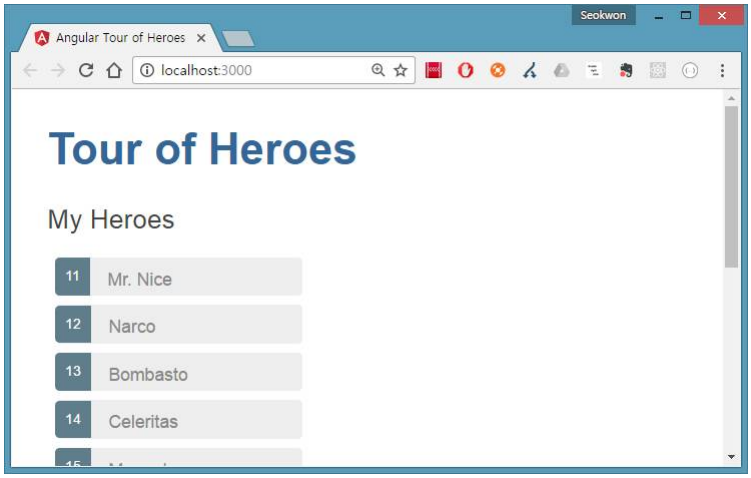
Thanks to the binding, the HeroDetailComponent should receive the hero from the AppComponent and display that hero's detail beneath the list. The detail should update every time the user picks a new hero.

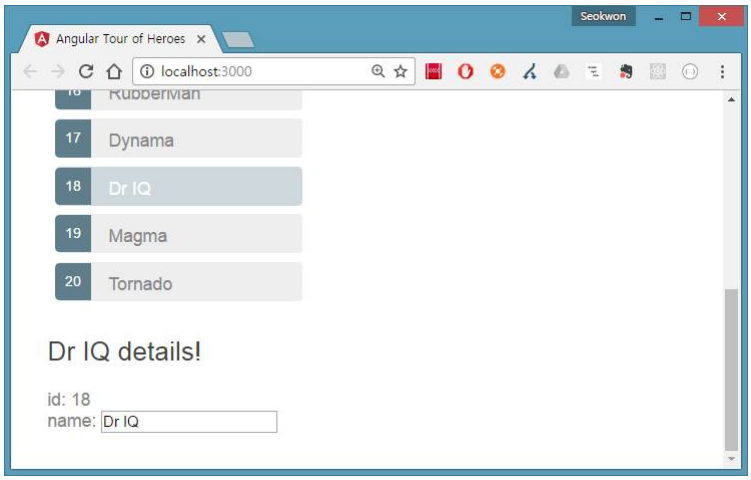
테스트 3

온라인 확인 > https://angular.io/resources/live-examples/toh-3/ts/eplnkr.html

What's fundamentally new is that we can use this HeroDetailComponent to show hero details anywhere in the app. We've created our first reusable component!

로컬 확인





Chapter 4. 업그레이드: Services

Multiple components will need **access to hero data** and we don't want to copy and paste the same code over and over. Instead, we'll **create a single reusable data service** and learn to **inject it in the components** that need it.

Refactoring data access to a separate service keeps the component lean and focused on supporting the view. It also makes it easier to unit test the component with a mock service.

Because data services are invariably asynchronous, we'll finish the chapter with a **Promise-based version** of the data service.

Creating a Hero Service

Our stakeholders have shared their larger vision for our app. They tell us they want to **show the heroes in various ways on different pages**. We already can select a hero from a list. Soon we'll add a **dashboard with the top performing heroes** and create a **separate view for editing hero details**.

All three views need hero data.

■ **list (done)**

■ **dashboard**

■ **editing**

At the moment the AppComponent defines mock heroes for display.

We have at least two objections.

■ First, defining heroes is not the component's job.

■ Second, we can't easily share that list of heroes with other components and views.

We can **refactor this hero data acquisition business to a single service** that provides heroes, and **share that service with all components** that need heroes.

hero.service.ts

Create a file in the app folder called hero.service.ts.

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

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

@Injectable()
export class HeroService {
  getHeroes(): Promise<Hero[]> {
    return Promise.resolve(HEROES);
  }
  // See the "Take it slow" appendix
  getHeroesSlowly(): Promise<Hero[]> {
    return new Promise<Hero[]>(resolve =>
      setTimeout(resolve, 2000)) // delay 2 seconds
      .then(() => this.getHeroes());
  }
}
```

We've adopted a convention in which we spell the name of a service in **lowercase** followed by **.service**. If the service name were multi-word, we'd spell the base filename in **lower dash-case**. The **SpecialSuperHeroService** would be defined in the **special-super-hero.service.ts** file.

We name the class HeroService and export it for others to import.

Injectable Services

Notice that we imported the Angular Injectable function and applied that function as an @Injectable() decorator.

TypeScript sees the @Injectable() decorator and **emits metadata about our service**, metadata that Angular may need to inject other dependencies into this service.

The HeroService doesn't have any dependencies at the moment. Add the decorator anyway. It is a "best practice" to apply the @Injectable() decorator from the start both for consistency and for future-proofing.

The consumer of our service doesn't know how the service gets the data. Our HeroService could get Hero data from anywhere. It could get the data from a web service or local storage or from a mock data source.

That's the beauty of removing data access from the component. We can change our minds about the implementation as often as we like, for whatever reason, without touching any of the components that need heroes.

In the HeroService we **import the mock HEROES** and **return it from the getHeroes method**.

```
  getHeroes(): Promise<Hero[]> {
    return Promise.resolve(HEROES);
  }
```

mock-heroes.ts

We already have mock Hero data sitting in the AppComponent. It doesn't belong there. It doesn't belong here either. We'll move the mock data to its own file.

Cut the HEROES array from app.component.ts and paste it to a new file in the app folder named mock-heroes.ts. We copy the import {Hero} ... statement as well because the **heroes array uses the Hero class**.

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

export const HEROES: Hero[] = [
  {id: 11, name: 'Mr. 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'}
];
```

Meanwhile, back in app.component.ts where we cut away the HEROES array, we leave behind an uninitialized heroes property:

app.component.ts

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

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

@Component({
  selector: 'my-app',
  template: `
    <h1>{{title}}</h1>
    <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>
    <my-hero-detail [hero]="selectedHero"></my-hero-detail>
  `,
  styles: [`
    .selected {
      background-color: #CFD8DC !important;
      color: white;
    }
    .heroes {
      margin: 0 0 2em 0;
      list-style-type: none;
      padding: 0;
      width: 15em;
    }
    .heroes li {
      cursor: pointer;
      position: relative;
      left: 0;
      background-color: #EEE;
      margin: .5em;
      padding: .3em 0;
      height: 1.6em;
      border-radius: 4px;
    }
    .heroes li.selected:hover {
      background-color: #BBD8DC !important;
      color: white;
    }
    .heroes li:hover {
      color: #607D8B;
      background-color: #DDD;
      left: .1em;
    }
    .heroes .text {
      position: relative;
      top: -3px;
```

```
    }
    .heroes .badge {
      display: inline-block;
      font-size: small;
      color: white;
      padding: 0.8em 0.7em 0 0.7em;
      background-color: #607D8B;
      line-height: 1em;
      position: relative;
      left: -1px;
      top: -4px;
      height: 1.8em;
      margin-right: .8em;
      border-radius: 4px 0 0 4px;
    }
  `],
  providers: [HeroService]
})
export class AppComponent implements OnInit {
  title = 'Tour of Heroes';
  heroes: Hero[];
  selectedHero: Hero;

  constructor(private heroService: HeroService) { }

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

  ngOnInit(): void {
    this.getHeroes();
  }

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

heroService = new HeroService(); // don't do this

If we ever change the HeroService constructor, we'll have to find every place we create the service and fix it. Running around patching code is error prone and adds to the test burden.

We create a new service each time we use new. What if the service should cache heroes and share that cache with others? We couldn't do that.

We're locking the AppComponent into a specific implementation of the HeroService. It will be hard to switch implementations for different scenarios. Can we operate offline? Will we need different mocked versions under test? Not easy.

It is so ridiculously easy to avoid these problems that there is no excuse for doing it wrong.

Inject the HeroService

Two lines replace the one line that created with new:

■ We add a constructor that also defines a private property.

■ We add to the component's providers metadata.

Here's the constructor:

```
constructor(private heroService: HeroService) { }
```

The constructor itself does nothing. **The parameter simultaneously defines a private heroService property and identifies it as a HeroService injection site**.

Now **Angular will know to supply an instance of the HeroService** when it creates a new AppComponent.

The **injector does not know yet how to create a HeroService**.

If we ran our code now, Angular would fail with an error: EXCEPTION: **No provider for HeroService!** (AppComponent -> HeroService)

We have to **teach the injector how to make a HeroService by registering a HeroService provider**. Do that by adding the following providers array property to the bottom of the component metadata in the @Component call.

```
@Component({
    providers: [HeroService]
})
```

The **providers** array **tells Angular to create a fresh instance of the HeroService when it creates a new AppComponent**. The AppComponent can use that service to get heroes and so can every child component of its component tree.

getHeroes in the AppComponent

We've got the service in a heroService private variable. Let's use it.

We pause to think. We can call the service and get the data in one line.

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

We don't really need a dedicated method to wrap one line. We write it anyway:

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

The ngOnInit Lifecycle Hook

AppComponent should fetch and display heroes without a fuss. Where do we call the getHeroes method? In a constructor? We do not!

Years of experience and bitter tears have taught us to keep complex logic out of the constructor, especially anything that might call a server as a data access method is sure to do.

The **constructor is for simple initializations like wiring constructor parameters to properties**.

https://angular.io/docs/ts/latest/guide/lifecycle-hooks.html

app.component.ts

Here's the essential outline for the OnInit interface

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

export class AppComponent implements OnInit {
  ngOnInit(): void {

  }
}
```

We write an ngOnInit method with our initialization logic inside and leave it to Angular to call it at the right time. In our case, we initialize by calling getHeroes.

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

Async Services and Promises

Our HeroService returns a list of mock heroes immediately. Its getHeroes signature is synchronous

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

Ask for heroes and they are there in the returned result.

Someday we're going to **get heroes from a remote server**. We don't call http yet, but we aspire to in later chapters.

When we do, we'll have to wait for the server to respond and **we won't be able to block the UI** while we wait, even if we want to (which we shouldn't) because the browser won't block.

We'll have to **use some kind of asynchronous technique** and that will change the signature of our getHeroes method.

The Hero Service makes a Promise

A Promise is ... well it's a promise to call us back later when the results are ready. We ask an asynchronous service to do some work and give it a callback function. It does that work (somewhere) and eventually it calls our function with the results of the work or an error.

hero.service.ts

Update the HeroService with this Promise-returning getHeroes method:

```
getHeroes(): Promise<Hero[]> {
  return Promise.resolve(HEROES);
}
```

We're still mocking the data. We're simulating the behavior of an ultra-fast, zero-latency server, by returning an immediately resolved Promise with our mock heroes as the result.

app.component.ts

We have to change our implementation to act on the Promise when it resolves. When the Promise resolves successfully, then we will have heroes to display.

We pass our callback function as an argument to the Promise's then method:

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

테스트 4

온라인 확인 > https://angular.io/resources/live-examples/toh-4/ts/eplnkr.html

로컬 결과 화면 전과 동.

Chapter 5. 업그레이드: Routing

We received new requirements for our Tour of Heroes application:

■ Add a Dashboard view.

Dashboard 뷰 생성.

■ Navigate between the Heroes and Dashboard views.

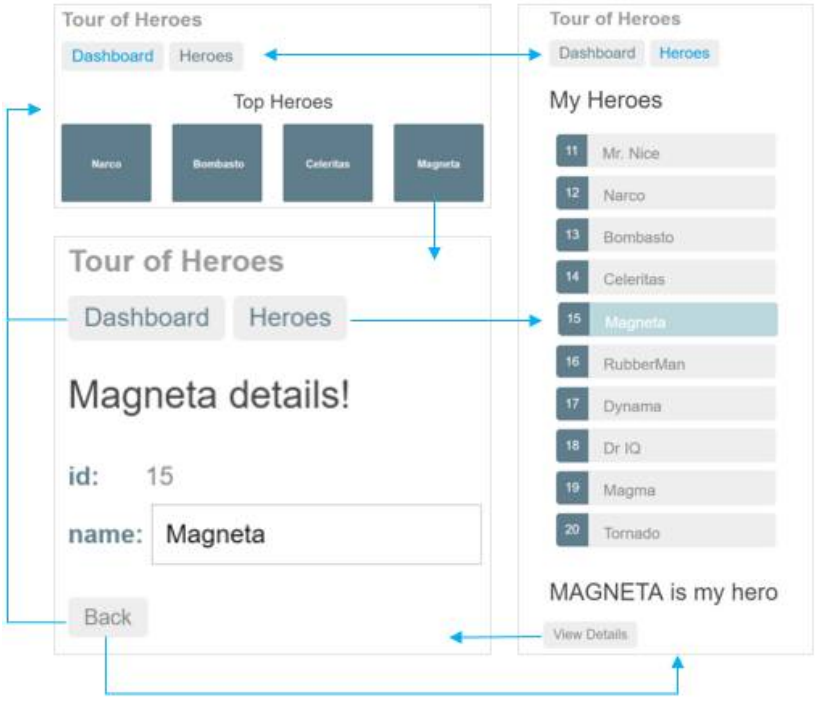
Dashboard 뷰 ←→ Heroes 뷰(버튼 2 개 제공)

■ Clicking on a hero in either view navigates to a detail view of the selected hero.

Dashboard 뷰, Heroes 뷰에서 영웅을 선택하면 detail view 로 이동

■ 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:



main.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';

platformBrowserDynamic().bootstrapModule(AppModule);
```

app.module.ts

```
import { NgModule }       from '@angular/core';
import { BrowserModule }  from '@angular/platform-browser';
import { FormsModule }    from '@angular/forms';

import { AppComponent }         from './app.component';
import { DashboardComponent }   from './dashboard.component';
import { HeroDetailComponent }  from './hero-detail.component';
import { HeroesComponent }      from './heroes.component';

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

import { AppRoutingModule }     from './app-routing.module';

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    AppRoutingModule
  ],
  declarations: [
    AppComponent,
    DashboardComponent,
    HeroDetailComponent,
    HeroesComponent
  ],
  providers: [ HeroService ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

Add **HeroesComponent** to the **declarations** array of AppModule so Angular recognizes the **<my-heroes>** tags.

Add **HeroService** to the **providers** array of AppModule because we'll need it in every other view. (Remove HeroService from the HeroesComponent providers array since it has been promoted.)

A **singleton HeroService instance**, available to all components of the application.

Angular will inject HeroService.

app-routing.module.ts

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

import { DashboardComponent }   from './dashboard.component';
import { HeroesComponent }      from './heroes.component';
import { HeroDetailComponent }  from './hero-detail.component';

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

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

The Angular router is an external, optional Angular NgModule called **RouterModule**.

The router is a **combination of multiple provided services** (RouterModule), **multiple directives** (RouterOutlet, RouterLink, RouterLinkActive), and a **configuration** (Routes).

The Routes are an array of route definitions. This route definition has the following parts:

■ **path**: the router matches this route's path to the URL in the browser address bar (heroes).

■ **component**: the component that the router should create when navigating to this route (HeroesComponent).

We use the **forRoot** method because we're **providing a configured router at the root of the application**. The forRoot method **gives us the Router service providers and directives** needed for routing, and **performs the initial navigation based on the current browser URL**.

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 a **<router-outlet>** element to the bottom of the template. **RouterOutlet is one of the directives provided by the RouterModule**. The router displays each component immediately below the <router-outlet> as we navigate through the application.

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. We can use a **redirectTo** route to make this happen.

A redirect route requires a **pathMatch** property to tell the router **how to match a URL to the path** of a route.

Technically, **pathMatch = 'full'** results in a route hit when the remaining, unmatched segments of the URL match ''. In our example, the redirect is at the top level of the route configuration tree so **the remaining URL and the entire URL are the same thing**.

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.

```
{
  path: 'detail/:id',
  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.

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.

app.component.ts

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

@Component({
  moduleId: module.id,
  selector: 'my-app',
  template: `
    <h1>{{title}}</h1>
    <nav>
      <a routerLink="/dashboard" routerLinkActive="active">Dashboard</a>
      <a routerLink="/heroes" routerLinkActive="active">Heroes</a>
    </nav>
    <router-outlet></router-outlet>
  `,
  styleUrls: ['app.component.css'],
})
export class AppComponent {
  title = 'Tour of Heroes';
}
```

The **AppComponent should only handle navigation**.

The AppComponent will be the application shell.

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.

It will have some navigation links at the top

```
    <nav>
      <a routerLink="/dashboard" routerLinkActive="active">Dashboard</a>
      <a routerLink="/heroes" routerLinkActive="active">Heroes</a>
    </nav>
```

and a display area below for the pages we navigate to.

```
<router-outlet></router-outlet>
```

Notice the routerLink binding in the anchor tag. We bind the RouterLink directive (another of the RouterModule directives) to a string that tells the router where to navigate when the user clicks the link.

The routerLinkActive directive

The Angular Router provides a **routerLinkActive** directive we can use to **add a (CSS) class to the HTML navigation element** whose route matches the active route.

We nested the two links within **<nav>** tags. They don't do anything yet but they'll be convenient when we style the links.

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.

dashboard.component.ts

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

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

@Component({
  moduleId: module.id,
  selector: 'my-dashboard',
  templateUrl: 'dashboard.component.html',
  styleUrls: [ 'dashboard.component.css' ]
})
export class DashboardComponent implements OnInit {

  heroes: Hero[] = [];

  constructor(private heroService: HeroService) { }

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

Set the moduleId property to **module.id for module-relative loading of the templateUrl**.

Set the moduleId property to **module.id so that templateUrl and styleUrls are relative to the component**.

In this dashboard we cherry-pick four heroes (2nd, 3rd, 4th, and 5th) with the Array.**slice** method.

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:

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

hero-detail.component.ts

```
import 'rxjs/add/operator/switchMap';
import { Component, OnInit }      from '@angular/core';
import { ActivatedRoute, Params } from '@angular/router';
import { Location }               from '@angular/common';

import { Hero }         from './hero';
import { HeroService }  from './hero.service';
@Component({
  moduleId: module.id,
  selector: 'my-hero-detail',
  templateUrl: 'hero-detail.component.html',
  styleUrls: [ 'hero-detail.component.css' ]
})
export class HeroDetailComponent implements OnInit {
  hero: Hero;

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

  ngOnInit(): void {
    this.route.params
      .switchMap((params: Params) => this.heroService.getHero(+params['id']))
      .subscribe(hero => this.hero = hero);
  }

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

Import the **switchMap** operator to use later with the route parameters Observable.

We will **no longer receive the hero in a parent component** property binding.

The new HeroDetailComponent **should take the id parameter** **from the params observable in the ActivatedRoute service** and **use the HeroService to fetch the hero with that id**.

Inside the ngOnInit lifecycle hook, we use the params observable to **extract the id parameter value from the ActivatedRoute service** and **use the HeroService to fetch the hero with that id**.

If the user re-navigates to this component while a getHero request is still inflight, switchMap cancels that old request before calling HeroService.getHero again.

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**.

We'll add a third option, a **goBack** method that **navigates backward** one step in the **browser's history stack** using the **Location service** we injected previously.

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.
https://angular.io/docs/ts/latest/api/router/index/CanDeactivate-interface.html

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.

heroes.component.ts

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

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

@Component({
  moduleId: module.id,
  selector: 'my-heroes',
  templateUrl: 'heroes.component.html',
  styleUrls: [ 'heroes.component.css' ]
})
export class HeroesComponent implements OnInit {
  heroes: Hero[];
  selectedHero: Hero;

  constructor(
    private router: Router,
    private heroService: HeroService) { }

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

  ngOnInit(): void {
    this.getHeroes();
  }

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

  gotoDetail(): void {
    this.router.navigate(['/detail', this.selectedHero.id]);
  }
}
```

Seperate the template contents into a new heroes.component.html file.

Seperate the styles contents into a new heroes.component.css file.

Set the component metadata's templateUrl and styleUrls properties to refer to both files.

Set the **moduleId** property to module.id so that templateUrl and styleUrls are relative to the component.

The HeroesComponent navigates to the HeroesDetailComponent in response to a button click. The button's click event is bound to a **gotoDetail** method that navigates imperatively by telling the router where to go.

This approach requires some changes to the component class:

■ Import the router from the Angular router library

■ Inject the router in the constructor (along with the HeroService)

■ Implement gotoDetail by calling the **router.navigate** method

Note that we're passing a two-element link parameters array — a **path** and the route **parameter** — to the router.navigate method just as we did in the **[routerLink] binding** back in the **dasboard.component.html**.

hero.service.ts

```
import { Hero } from './hero';
import { HEROES } from './mock-heroes';
import { Injectable } from '@angular/core';

@Injectable()
export class HeroService {
  getHeroes(): Promise<Hero[]> {
    return Promise.resolve(HEROES);
  }

  getHeroesSlowly(): Promise<Hero[]> {
    return new Promise<Hero[]>(resolve =>
      setTimeout(resolve, 2000)) // delay 2 seconds
      .then(() => this.getHeroes());
  }

  getHero(id: number): Promise<Hero> {
    return this.getHeroes()
      .then(heroes => heroes.find(hero => hero.id === id));
  }
}
```

dashboard.component.html

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

We use ***ngFor** once again **to iterate over a list of heroes** and display their names. We added extra elements to help with styling.

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

Although the dashboard heroes are presented as button-like blocks, they should **behave like anchor tags**. When hovering over a hero block, the **target URL should display** in the browser status bar and the user should be able to copy the link or open the hero detail view in a new tab.

To achieve this effect, reopen the dashboard.component.html and replace the repeated <div *ngFor...> tags with **<a>** tags. The opening <a> tag looks like this:

```
<a *ngFor="let hero of heroes"  [routerLink]="['/detail', hero.id]"
class="col-1-4">
```

[routerLink] binding

Top level navigation in the AppComponent template has router links set to fixed paths of the destination routes, "/dashboard" and "/heroes".

This time, we're **binding to an expression** containing a link parameters array. The array has two elements, the **path of the destination** route and a route **parameter** set to the value of the current **hero's id**.

The **two array items align with the path and :id token** in the parameterized hero detail route definition we added to **app-routing.module.ts**.

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

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

We are keeping the "master/detail" style but shrinking the detail to a "mini", read-only version. When the user selects a hero from the list, we don't go to the detail page. We show a mini-detail on this page instead and make the user click a button to navigate to the full detail page.

Format with the uppercase pipe

Notice that the hero's name is displayed in CAPITAL LETTERS. That's the effect of the **uppercase pipe** 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.

https://angular.io/docs/ts/latest/guide/pipes.html

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.

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>
```

테스트 5

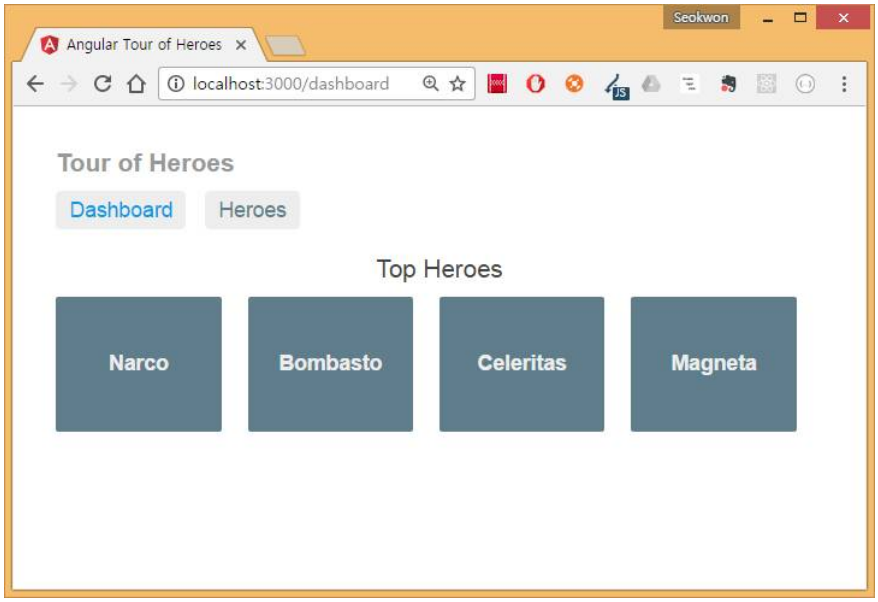
온라인 확인 > https://angular.io/resources/live-examples/toh-5/ts/eplnkr.html

로컬 확인

app.component.ts

nav

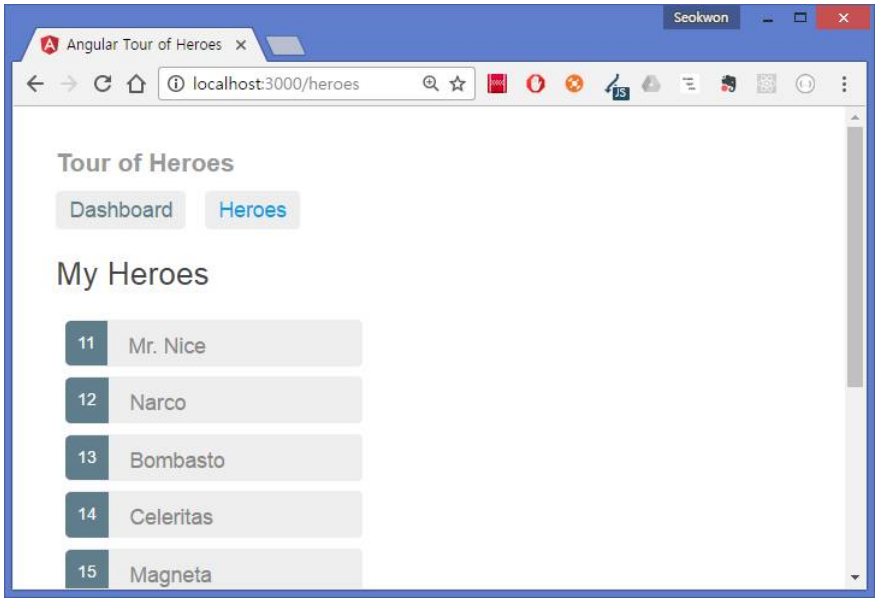
router-outlet : **dashboard.component.html** ←→ heroes.component.html



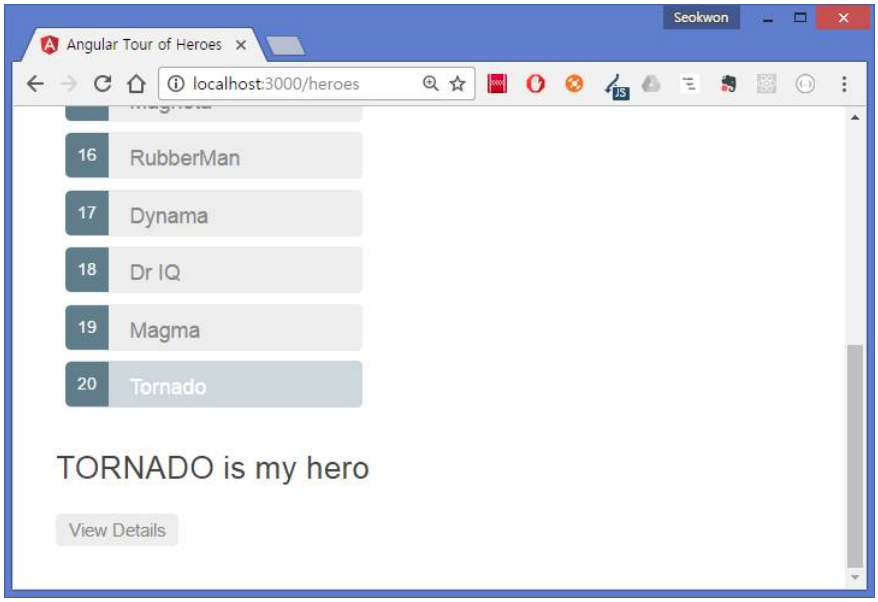
app.component.ts

nav

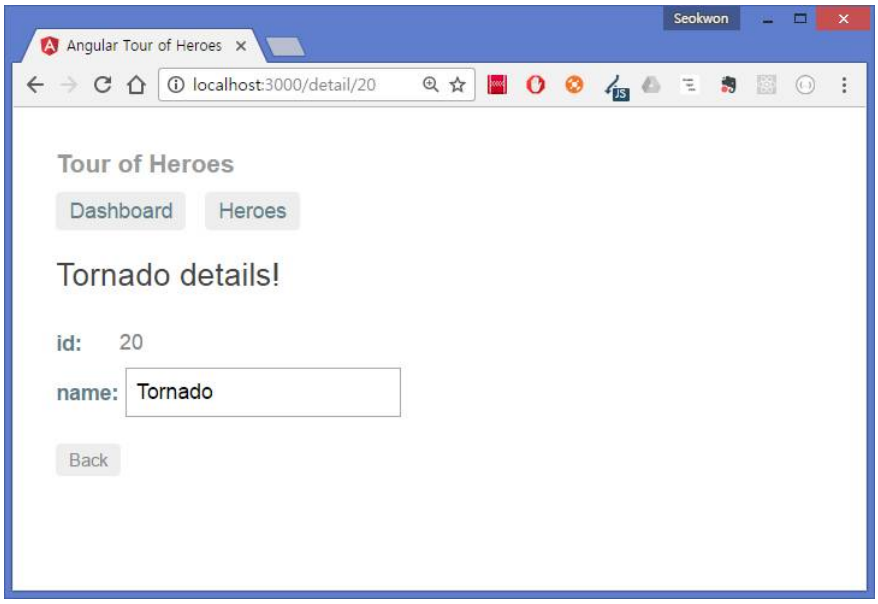
router-outlet : dashboard.component.html ←→ **heroes.component.html**



heroes.component.html > **div *ngIf="selectedHero"**



hero-detail.component.html



Chapter 6. 업그레이드: HTTP

Our stakeholders appreciate our progress. Now they want to **get the hero data from a server**, let users **add, edit, and delete heroes**, and save these changes back to the server.

https://angular.io/resources/live-examples/toh-6/ts/eplnkr.html

Providing HTTP Services

The HttpModule is not a core Angular module. It's Angular's **optional** approach to web access and it exists as a separate add-on module called **@angular/http**, shipped in a separate script file as part of the Angular npm package.

Fortunately we're ready to import from **@angular/http** because **systemjs.config** configured **SystemJS to load that library when we need it**.

systemjs.config.js

```
/**
 * System configuration for Angular samples
 * Adjust as necessary for your application needs.
 */
(function (global) {
  System.config({
    paths: {
      // paths serve as alias
      'npm:': 'node_modules/'
    },
    // map tells the System loader where to look for things
    map: {
      // our app is within the app folder
      app: 'app',

      // angular bundles
      '@angular/core': 'npm:@angular/core/bundles/core.umd.js',
      '@angular/common': 'npm:@angular/common/bundles/common.umd.js',
      '@angular/compiler': 'npm:@angular/compiler/bundles/compiler.umd.js',
      '@angular/platform-browser': 'npm:@angular/platform-
browser/bundles/platform-browser.umd.js',
      '@angular/platform-browser-dynamic': 'npm:@angular/platform-browser-
dynamic/bundles/platform-browser-dynamic.umd.js',
      '@angular/http': 'npm:@angular/http/bundles/http.umd.js',
      '@angular/router': 'npm:@angular/router/bundles/router.umd.js',
      '@angular/forms': 'npm:@angular/forms/bundles/forms.umd.js',
```

```
      // other libraries
      'rxjs':                      'npm:rxjs',
      'angular-in-memory-web-api': 'npm:angular-in-memory-web-api/bundles/in-
memory-web-api.umd.js'
    },
    // packages tells the System loader how to load when no filename and/or no
extension
    packages: {
      app: {
        main: './main.js',
        defaultExtension: 'js'
      },
      rxjs: {
        defaultExtension: 'js'
      }
    }
  });
})(this);
```

Register for HTTP services

Our app will depend upon the Angular http service which itself depends upon other supporting services. The HttpModule from @angular/http library holds providers for a complete set of HTTP services.

We **should be able to access these services from anywhere in the application**. So we register them all by adding **HttpModule** to the imports list of the **AppModule** where we **bootstrap the application and its root AppComponent**.

app.module.ts

```
import './rxjs-extensions';

import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule }   from '@angular/forms';
import { HttpModule }    from '@angular/http';

import { AppRoutingModule } from './app-routing.module';

// Imports for loading & configuring the in-memory web api
import { InMemoryWebApiModule } from 'angular-in-memory-web-api';
import { InMemoryDataService }  from './in-memory-data.service';

import { AppComponent }         from './app.component';
import { DashboardComponent }   from './dashboard.component';
import { HeroesComponent }      from './heroes.component';
import { HeroDetailComponent }  from './hero-detail.component';
import { HeroService }          from './hero.service';
import { HeroSearchComponent }  from './hero-search.component';

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    InMemoryWebApiModule.forRoot(InMemoryDataService),
    AppRoutingModule
  ],
  declarations: [
    AppComponent,
    DashboardComponent,
    HeroDetailComponent,
    HeroesComponent,
    HeroSearchComponent
  ],
  providers: [ HeroService ],
```

```
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

Simulating the web API

We **recommend registering application-wide services in the root AppModule providers**.
Our application is in the early stages of development and far from ready for production. We don't even have a web server that can handle requests for heroes. Until we do, we'll have to **fake** it.

We're going to trick the HTTP client into fetching and saving data **from a mock service, the in-memory web API**.

Here is a version of app/app.module.ts that performs this trick:

We're importing the **InMemoryWebApiModule** and adding it to the module imports. The **InMemoryWebApiModule replaces the default Http client backend** — the supporting service that talks to the remote server — with an in-memory web API alternative service.

```
InMemoryWebApiModule.forRoot(InMemoryDataService),
```

The **forRoot** configuration method takes an **InMemoryDataService** class that primes the in-memory database as follows:

in-memory-data.service.ts

```
import { InMemoryDbService } from 'angular-in-memory-web-api';
export class InMemoryDataService implements InMemoryDbService {
  createDb() {
    let heroes = [
      {id: 11, name: 'Mr. 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};
  }
}
```

This file **replaces the mock-heroes.ts** which is now safe to delete.

https://angular.io/docs/ts/latest/guide/server-communication.html#in-mem-web-api

Remember, the in-memory web API is **only useful in the early stages of development** and for demonstrations such as this Tour of Heroes. Skip it when you have a real web API server.

작업내용

rsjs-extensions.ts

```
// Observable class extensions
import 'rxjs/add/observable/of';
import 'rxjs/add/observable/throw';

// Observable operators
import 'rxjs/add/operator/catch';
import 'rxjs/add/operator/debounceTime';
import 'rxjs/add/operator/distinctUntilChanged';
import 'rxjs/add/operator/do';
import 'rxjs/add/operator/filter';
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/switchMap';
```

Import RxJS operators

The **RxJS operators are not available in Angular's base Observable implementation**. We **have to extend Observable by importing them**.

We could extend Observable with just the operators we need here by including the pertinent import statements at the top of this file.

We take a different approach in this example. We combine all of the RxJS Observable extensions that our **entire app requires into a single RxJS imports file**.

We **load them all at once by importing rxjs-extensions at the top of AppModule**.

```
import './rxjs-extensions';
```

hero.service.ts

```
import { Injectable }    from '@angular/core';
import { Headers, Http } from '@angular/http';

import 'rxjs/add/operator/toPromise';

import { Hero } from './hero';

@Injectable()
export class HeroService {

  private headers = new Headers({'Content-Type': 'application/json'});
  private heroesUrl = 'api/heroes';  // URL to web api

  constructor(private http: Http) { }

  getHeroes(): Promise<Hero[]> {
    return this.http.get(this.heroesUrl)
      .toPromise()
      .then(response => response.json().data as Hero[])
      .catch(this.handleError);
  }

  getHero(id: number): Promise<Hero> {
    const url = `${this.heroesUrl}/${id}`;
    return this.http.get(url)
      .toPromise()
      .then(response => response.json().data as Hero)
      .catch(this.handleError);
  }

  delete(id: number): Promise<void> {
    const url = `${this.heroesUrl}/${id}`;
    return this.http.delete(url, {headers: this.headers})
      .toPromise()
      .then(() => null)
      .catch(this.handleError);
  }

  create(name: string): Promise<Hero> {
    return this.http
      .post(this.heroesUrl, JSON.stringify({name: name}), {headers:
this.headers})
      .toPromise()
      .then(res => res.json().data)
      .catch(this.handleError);
  }

  update(hero: Hero): Promise<Hero> {
    const url = `${this.heroesUrl}/${hero.id}`;
    return this.http
      .put(url, JSON.stringify(hero), {headers: this.headers})
      .toPromise()
```

```
      .then(() => hero)
      .catch(this.handleError);
  }

  private handleError(error: any): Promise<any> {
    console.error('An error occurred', error); // for demo purposes only
    return Promise.reject(error.message || error);
  }
}
```

HTTP Promise

We're still returning a Promise but we're creating it differently.

The Angular **http.get** returns an **RxJS Observable**.

Observables are a powerful way to manage asynchronous data flows.

We'll learn about Observables later in this chapter.

https://angular.io/docs/ts/latest/tutorial/toh-pt6.html#observables

For now we get back on familiar ground by immediately **converting** that **Observable to a Promise using the toPromise operator**.

Unfortunately, the **Angular Observable doesn't have a toPromise operator** ... not out of the box. The Angular Observable is a bare-bones implementation.

There are scores of operators like toPromise that extend Observable with useful capabilities. If we want those capabilities, we have to **add the operators** ourselves. That's as easy as **importing them from the RxJS library** like this:

```
import 'rxjs/add/operator/toPromise';
```

Extracting the data in the then callback

In the promise's then callback we call the json method of the HTTP Response to extract the data within the response.

```
.then(response => response.json().data as Hero[])
```

That response JSON has a single data property. The **data property holds the array of heroes** that the caller really wants. So we grab that array and return it as the resolved Promise value.

> Pay close attention to the shape of the data returned by the server. This particular in-memory web API example happens to return an object with a data property. Your API might return something else. Adjust the code to match your web API.

Get hero by id

The **HeroDetailComponent** asks the **HeroService** to fetch a single hero to edit.

The HeroService currently fetches all heroes and then finds the desired hero by filtering for the one with the matching id. That's fine in a simulation. It's wasteful to ask a real server for all heroes when we only want one. Most web APIs support a get-by-id request in the form **api/hero/:id** (e.g., api/hero/11).

Update the **HeroService.getHero** method to make a get-by-id request, applying what we just learned to write getHeroes:

```
getHero(id: number): Promise<Hero> {
  const url = `${this.heroesUrl}/${id}`;
  return this.http.get(url)
    .toPromise()
    .then(response => response.json().data as Hero)
    .catch(this.handleError);
}
```

It's almost the same as getHeroes. The URL identifies which hero the server should update by encoding the hero id into the URL to match the api/hero/:id pattern.

We also adjust to the fact that the data in the response is **a single hero object** rather than an array.

Unchanged getHeroes API

Although we made significant internal changes to getHeroes() and getHero(), the public signatures did not change. We still return a Promise from both methods. We won't have to update any of the components that call them.

Hero service update method

The overall structure of the update method is similar to that of getHeroes, although we'll use an HTTP put to persist changes server-side:

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

update(hero: Hero): Promise<Hero> {
  const url = `${this.heroesUrl}/${hero.id}`;
  return this.http
    .put(url, JSON.stringify(hero), {headers: this.headers})
    .toPromise()
    .then(() => hero)
    .catch(this.handleError);
}
```

We identify which hero the server should update by encoding the hero id in the URL. The put body is the JSON string encoding of the hero, obtained by calling **JSON.stringify**. We identify the body content type (**application/json**) in the request header.

Hero service delete method

The hero service's delete method uses the delete HTTP method to remove the hero from the server:

```
delete(id: number): Promise<void> {
  const url = `${this.heroesUrl}/${id}`;
  return this.http.delete(url, {headers: this.headers})
    .toPromise()
    .then(() => null)
    .catch(this.handleError);
}
```

Observables

Each Http service method returns an Observable of HTTP Response objects.

Our HeroService **converts that Observable into a Promise** and returns the promise to the caller. In this section we learn to return the Observable directly and discuss when and why that might be a good thing to do.

Background

An observable is a stream of events that **we can process with array-like operators**.

Angular core has basic support for observables. We developers augment that support with operators and extensions from the **RxJS Observables library**. We'll see how shortly.

Recall that our HeroService quickly chained the toPromise operator to the Observable result of http.get. That operator converted the Observable into a Promise and we passed that promise back to the caller.

Converting to a promise is often a good choice. We typically ask **http.get to fetch a single chunk of data**. When we receive the data, we're done. A single result in the form of a promise is easy for the calling component to consume and it helps that promises are widely understood by JavaScript programmers.

But requests aren't always "one and done". We may start one request, then cancel it, and make a different request before the server has responded to the first request. Such a **request-cancel-new-request sequence is difficult to implement with Promises**. It's easy with Observables as we'll see.

hero-search.service.ts

```
import { Injectable }     from '@angular/core';
import { Http, Response } from '@angular/http';
import { Observable } from 'rxjs';

import { Hero }           from './hero';

@Injectable()
export class HeroSearchService {

  constructor(private http: Http) {}

  search(term: string): Observable<Hero[]> {
    return this.http
      .get(`app/heroes/?name=${term}`)
      .map((r: Response) => r.json().data as Hero[]);
  }
}
```

Search-by-name

We're going to add a hero search feature to the Tour of Heroes. As the **user types a name into a search box, we'll make repeated HTTP requests** for heroes filtered by that name.

We start by creating **HeroSearchService** that sends search queries to our server's web api.

The http.get() call in HeroSearchService is similar to the one in the HeroService, although the URL now has a query string. Another notable **difference**: we no longer call toPromise, we simply **return the observable** instead.

hero-search.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Router }            from '@angular/router';
import { Observable }        from 'rxjs/Observable';
import { Subject }           from 'rxjs/Subject';

import { HeroSearchService } from './hero-search.service';
import { Hero } from './hero';

@Component({
  moduleId: module.id,
  selector: 'hero-search',
  templateUrl: 'hero-search.component.html',
  styleUrls: [ 'hero-search.component.css' ],
  providers: [HeroSearchService]
})
export class HeroSearchComponent implements OnInit {
  heroes: Observable<Hero[]>;
  private searchTerms = new Subject<string>();

  constructor(
    private heroSearchService: HeroSearchService,
    private router: Router) {}

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

  ngOnInit(): void {
    this.heroes = this.searchTerms
      .debounceTime(300)        // wait for 300ms pause in events
      .distinctUntilChanged()   // ignore if next search term is same as pre-
vious
      .switchMap(term => term   // switch to new observable each time
        // return the http search observable
        ? this.heroSearchService.search(term)
        // or the observable of empty heroes if no search term
        : Observable.of<Hero[]>([]))
      .catch(error => {
        // TODO: real error handling
        console.log(error);
        return Observable.of<Hero[]>([]);
      });
  }

  gotoDetail(hero: Hero): void {
    let link = ['/detail', hero.id];
    this.router.navigate(link);
  }
}
```

SEARCH TERMS

Let's focus on the searchTerms:

```
private searchTerms = new Subject<string>();

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

A **Subject is a producer of an observable event stream**; searchTerms produces an Observable of strings, the filter criteria for the name search.

Each call to search puts a new string into this **subject's observable stream by calling next**.

INITIALIZE THE HEROES PROPERTY (NGONINIT)

A Subject is also an Observable. We're going to **turn the stream of search terms into a stream of Hero arrays** and **assign the result to the heroes property**.

```
heroes: Observable<Hero[]>;

ngOnInit(): void {
  this.heroes = this.searchTerms
    .debounceTime(300)        // wait for 300ms pause in events
    .distinctUntilChanged()   // ignore if next search term is same as pre-
vious
    .switchMap(term => term   // switch to new observable each time
      // return the http search observable
      ? this.heroSearchService.search(term)
      // or the observable of empty heroes if no search term
      : Observable.of<Hero[]>([]))
    .catch(error => {
      // TODO: real error handling
      console.log(error);
      return Observable.of<Hero[]>([]);
    });
}
```

If we passed every user keystroke directly to the HeroSearchService, we'd unleash a storm of HTTP requests. Bad idea. We don't want to tax our server resources and burn through our cellular network data plan.

Fortunately, we can **chain Observable operators to the string Observable** that **reduce the request flow**. We'll **make fewer calls to the HeroSearchService** and still get timely results. Here's how:

■ debounceTime(300)
waits until the flow of new string events pauses for 300 milliseconds before passing along the latest string. We'll **never make requests more frequently than 300ms**.

■ distinctUntilChanged
ensures that we only send a request if the filter text changed. **There's no point in repeating a request for the same search term**.

■ switchMap
calls our search service for each search term that **makes it through the debounce and distinctUntilChanged** gauntlet. It cancels and discards previous search observables, returning only the latest search service observable.

■ catch
intercepts a failed observable. Our simple example prints the error to the console; a real life application should do better. Then we return an observable containing an empty array to clear the search result.

hero-search.component.html

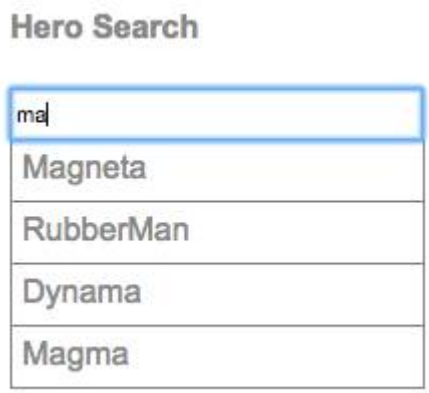
```
<div id="search-component">
  <h4>Hero Search</h4>
  <input #searchBox id="search-box" (keyup)="search(searchBox.value)" />
  <div>
    <div *ngFor="let hero of heroes | async"
         (click)="gotoDetail(hero)" class="search-result" >
      {{hero.name}}
    </div>
  </div>
</div>
```

The component template is simple — just a text box and a list of matching search results.

As the user types in the search box, a **keyup** event binding **calls the component's search method** with the new search box value.

The *ngFor repeats hero objects from the component's heroes property. No surprise there.

But, as we'll soon see, the **heroes property is now an Observable of hero arrays**, rather than just a hero array. The ***ngFor can't do anything with an Observable until we flow it through the async pipe (AsyncPipe)**. The **async pipe subscribes to the Observable and produces the array of heroes** to *ngFor.



dashboard.component.html

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

Add the search component to the dashboard

We add the hero search HTML element to the bottom of the DashboardComponent template.

heroes.component.ts

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

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

@Component({
  moduleId: module.id,
  selector: 'my-heroes',
  templateUrl: 'heroes.component.html',
  styleUrls: [ 'heroes.component.css' ]
})
export class HeroesComponent implements OnInit {
  heroes: Hero[];
  selectedHero: Hero;

  constructor(
    private heroService: HeroService,
    private router: Router) { }

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

  add(name: string): void {
    name = name.trim();
    if (!name) { return; }
    this.heroService.create(name)
      .then(hero => {
        this.heroes.push(hero);
        this.selectedHero = null;
      });
  }

  delete(hero: Hero): void {
    this.heroService
      .delete(hero.id)
      .then(() => {
        this.heroes = this.heroes.filter(h => h !== hero);
        if (this.selectedHero === hero) { this.selectedHero = null; }
      });
  }

  ngOnInit(): void {
    this.getHeroes();
  }

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

```
  gotoDetail(): void {
    this.router.navigate(['/detail', this.selectedHero.id]);
  }
}
```

```
add(name: string): void {
  name = name.trim();
  if (!name) { return; }
  this.heroService.create(name)
    .then(hero => {
      this.heroes.push(hero);
      this.selectedHero = null;
    });
}
```

When the given name is non-blank, the handler delegates creation of the named hero to the hero service, and then adds the new hero to our array.

The logic of the delete handler is a bit trickier:

```
delete(hero: Hero): void {
  this.heroService
      .delete(hero.id)
      .then(() => {
        this.heroes = this.heroes.filter(h => h !== hero);
        if (this.selectedHero === hero) { this.selectedHero = null; }
      });
}
```

Of course, we **delegate hero deletion to the hero service**, but **the component is still responsible for updating the display**: it removes the deleted hero from the array and resets the selected hero if necessary.

heroes.component.html

```
<h2>My Heroes</h2>
<div>
  <label>Hero name:</label> <input #heroName />
  <button (click)="add(heroName.value); heroName.value=''">
    Add
  </button>
</div>
<ul class="heroes">
  <li *ngFor="let hero of heroes" (click)="onSelect(hero)"
      [class.selected]="hero === selectedHero">
    <span class="badge">{{hero.id}}</span>
    <span>{{hero.name}}</span>
    <button class="delete"
            (click)="delete(hero); $event.stopPropagation()">x</button>
  </li>
</ul>
<div *ngIf="selectedHero">
  <h2>
    {{selectedHero.name | uppercase}} is my hero
  </h2>
  <button (click)="gotoDetail()">View Details</button>
</div>
```

Add a hero

To add a new hero we need to know the hero's name. Let's use an input element for that, paired with an add button.

Insert the following into the heroes component HTML, first thing after the heading:

```
<div>
  <label>Hero name:</label> <input #heroName />
  <button (click)="add(heroName.value); heroName.value=''">
    Add
  </button>
</div>
```

In response to a click event, we call the component's click handler and then **clear the input field** so that it will be ready to use for another name.

Delete a hero

Add this button element to the heroes component HTML, right after the hero name in the repeated <li> tag:

```
<button class="delete"
  (click)="delete(hero); $event.stopPropagation()">x</button>
```

In addition to calling the component's **delete** method, the delete button click handling code stops the propagation of the click event — **we don't want the <li> click handler to be triggered** because that would select the hero that we are going to delete!

hero-detail.component.ts

```
import 'rxjs/add/operator/switchMap';
import { Component, OnInit }      from '@angular/core';
import { ActivatedRoute, Params } from '@angular/router';
import { Location }               from '@angular/common';

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

@Component({
  moduleId: module.id,
  selector: 'my-hero-detail',
  templateUrl: 'hero-detail.component.html',
  styleUrls: [ 'hero-detail.component.css' ]
})
export class HeroDetailComponent implements OnInit {
  hero: Hero;

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

  ngOnInit(): void {
    this.route.params
      .switchMap((params: Params) => this.heroService.getHero(+params['id']))
      .subscribe(hero => this.hero = hero);
  }

  save(): void {
    this.heroService.update(this.hero)
      .then(() => this.goBack());
  }

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

The **save** method persists hero name changes using the **heroService.update** method and then navigates back to the previous view:

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

As we type, the hero name is updated in the view heading. But when we hit the Back button, the changes are lost!

Updates weren't lost before. What changed? When the app used a list of mock heroes, updates were applied directly to the hero objects within the single, app-wide, shared list. Now that we are fetching data from a server, if we want **changes to persist**, we'll **need to write them back to the server**.

Save hero details

Let's ensure that edits to a hero's name aren't lost. Start by adding, to the end of the hero detail template, a save button with a click event binding that invokes a new component method named save:

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

수고하셨습니다.