Angular CLI

https://cli.angular.io/

1. 설치

npm install -g angular-cli

1.1. 확인

ng help

ng build <options...>

```
  Builds your app and places it into the output path (dist/ by default).
  aliases: b
  --target (String) (Default: development)
    aliases: -t <value>, -dev (--target=development), -prod (--target=production)
  --environment (String) (Default: )
    aliases: -e <value>
  --output-path (Path) (Default: null)
    aliases: -o <value>
  --watch (Boolean) (Default: false)
    aliases: -w
  --watcher (String)
  --suppress-sizes (Boolean) (Default: false)
  --base-href (String) (Default: null)
    aliases: -bh <value>
  --aot (Boolean) (Default: false)
  --sourcemap (Boolean) (Default: true)
    aliases: -sm
```

ng completion

```
  Adds autocomplete functionality to `ng` commands and subcommands

ng doc <keyword>
  Opens the official Angular documentation for a given keyword.

ng e2e
  Run e2e tests in existing project
```

ng generate <blueprint> <options...>

```
  Generates new code from blueprints.
  aliases: g
  --dry-run (Boolean) (Default: false)
    aliases: -d
  --verbose (Boolean) (Default: false)
    aliases: -v
```

```
  --pod (Boolean) (Default: false)
    aliases: -p
  --classic (Boolean) (Default: false)
    aliases: -c
  --dummy (Boolean) (Default: false)
    aliases: -dum, -id
  --in-repo-addon (String) (Default: null)
    aliases: --in-repo <value>, -ir <value>
```

ng get

```
  Get a value from the configuration.
```

ng help

```
  Shows help for the CLI
```

ng init <glob-pattern> <options...>

```
  Creates a new angular-cli project in the current folder.
  aliases: i
  --dry-run (Boolean) (Default: false)
    aliases: -d
  --verbose (Boolean) (Default: false)
    aliases: -v
  --link-cli (Boolean) (Default: false)
    aliases: -lc
  --skip-npm (Boolean) (Default: false)
    aliases: -sn
  --skip-bower (Boolean) (Default: true)
    aliases: -sb
  --name (String) (Default: )
    aliases: -n <value>
  --source-dir (String) (Default: src)
    aliases: -sd <value>
  --style (String) (Default: css)
  --prefix (String) (Default: app)
    aliases: -p <value>
  --mobile (Boolean) (Default: false)
  --routing (Boolean) (Default: false)
  --inline-style (Boolean) (Default: false)
    aliases: -is
  --inline-template (Boolean) (Default: false)
    aliases: -it
```

ng lint

```
  Lints code in existing project
```

ng new <options...>

```
  Creates a new directory and runs ng init in it.
```

```
  --dry-run (Boolean) (Default: false)
    aliases: -d
  --verbose (Boolean) (Default: false)
    aliases: -v
  --link-cli (Boolean) (Default: false)
    aliases: -lc
  --skip-npm (Boolean) (Default: false)
    aliases: -sn
  --skip-bower (Boolean) (Default: true)
    aliases: -sb
  --skip-git (Boolean) (Default: false)
    aliases: -sg
  --directory (String)
    aliases: -dir <value>
  --source-dir (String) (Default: src)
    aliases: -sd <value>
  --style (String) (Default: css)
  --prefix (String) (Default: app)
    aliases: -p <value>
  --mobile (Boolean) (Default: false)
  --routing (Boolean) (Default: false)
  --inline-style (Boolean) (Default: false)
    aliases: -is
  --inline-template (Boolean) (Default: false)
    aliases: -it
```

ng serve <options...>

```
  Builds and serves your app, rebuilding on file changes.
  aliases: server, s
  --port (Number) (Default: 4200)
    aliases: -p <value>
  --host (String) (Default: localhost) Listens only on localhost by default
    aliases: -H <value>
  --proxy-config (Path)
    aliases: -pc <value>
  --watcher (String) (Default: events)
    aliases: -w <value>
  --live-reload (Boolean) (Default: true)
    aliases: -lr
  --live-reload-host (String) Defaults to host
    aliases: -lrh <value>
  --live-reload-base-url (String) Defaults to baseURL
    aliases: -lrbu <value>
  --live-reload-port (Number) (Defaults to port number within [49152...65535])
    aliases: -lrp <value>
  --live-reload-live-css (Boolean) (Default: true) Whether to live reload CSS (default
true)
  --target (String) (Default: development)
    aliases: -t <value>, -dev (--target=development), -prod (--target=production)
  --environment (String) (Default: )
    aliases: -e <value>
  --ssl (Boolean) (Default: false)
  --ssl-key (String) (Default: ssl/server.key)
  --ssl-cert (String) (Default: ssl/server.crt)
  --aot (Boolean) (Default: false)
  --sourcemap (Boolean) (Default: true)
    aliases: -sm
  --open (Boolean) (Default: false) Opens the url in default browser
    aliases: -o
```

ng set <options...>

```
Set a value in the configuration.
--global (Boolean) (Default: false)
  aliases: -g
```

ng test <options...>

```
Runs your app's test suite.
aliases: t
--watch (Boolean) (Default: true)
  aliases: -w
--code-coverage (Boolean) (Default: false)
  aliases: -cc
--lint (Boolean) (Default: false)
  aliases: -l
--single-run (Boolean) (Default: false)
  aliases: -sr
--browsers (String)
--colors (Boolean)
--log-level (String)
--port (Number)
--reporters (String)
--build (Boolean) (Default: true)
--sourcemap (Boolean) (Default: true)
  aliases: -sm
```

ng version <options...>

```
outputs angular-cli version
aliases: v, --version, -v
--verbose (Boolean) (Default: false)
```

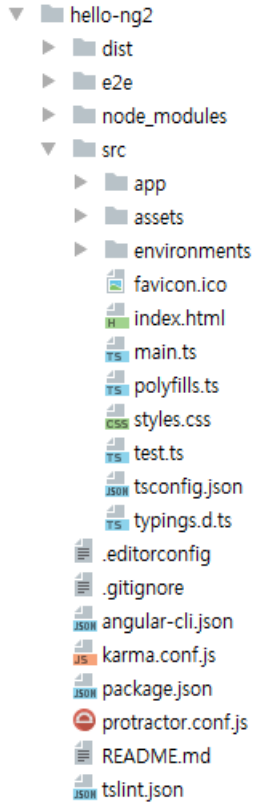
ng e2e

```
종단 테스트(end to end tests)를 수행한다.
```

2. 프로젝트 만들기

```
ng new hello-ng2
```

결과

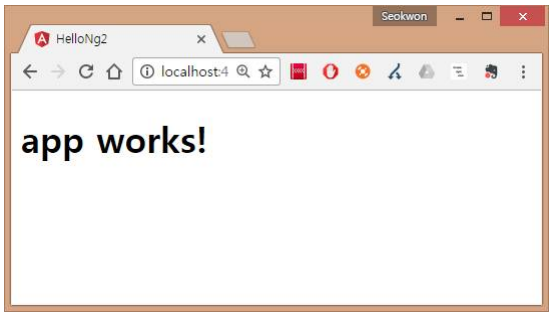


3. 개발서버 실행

```
cd hello-ng2
ng serve
```

개발서버 접속

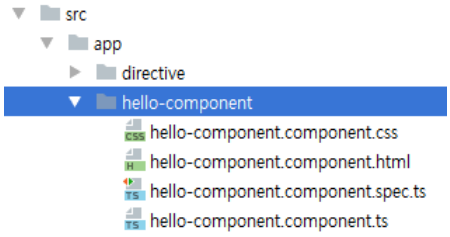
```
http://localhost:4200
```



4. 구성요소 추가

컴포넌트 추가

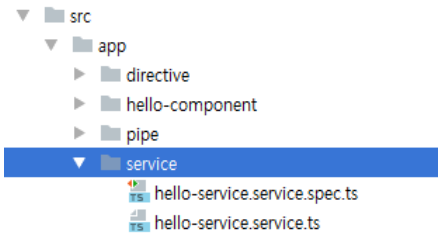
```
/
ng g component hello-component
```



프로젝트 루트 패스에서 컴포넌트 생성 명령을 실행하면 자동으로 src/app 폴더 밑으로 주어진 키워드 'hello-component'를 사용하여 폴더를 만들고 그 밑으로 컴포넌트가 생성된다. 따라서 컴포넌트의 폴더의 하부 폴더로 자식 컴포넌트를 생성하고 싶다면 해당 컴포넌트로 커서를 옮긴 후 자식 컴포넌트를 생성하는 명령을 수행해야 한다.

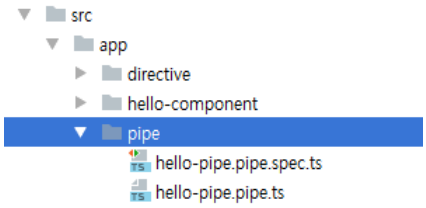
서비스 추가

```
cd src/app
mkdir service
cd service
ng g service hello-service
```



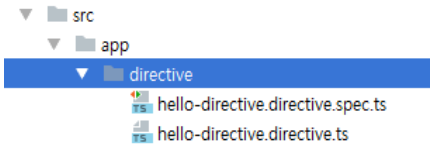
파이프 추가

```
cd src/app
mkdir pipe
cd pipe
ng g pipe hello-pipe
```



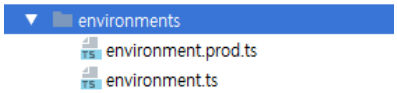
디렉티브 추가

```
cd src/app
mkdir directive
cd directive
ng g directive hello-directive
```



5. 빌드 for 개발자

빌드 설정



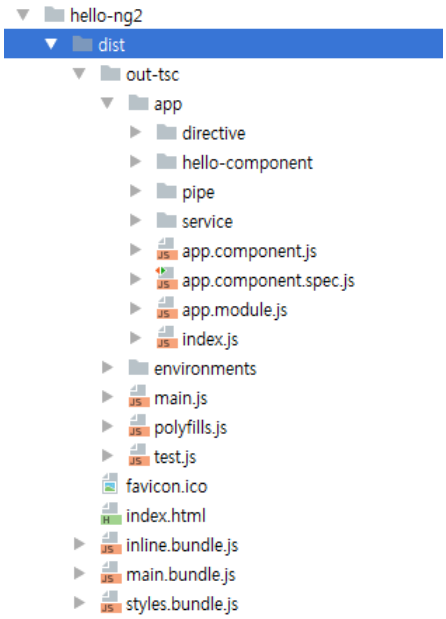
environment.ts

```
// The file contents for the current environment will overwrite these during build.
// The build system defaults to the dev environment which uses `environment.ts`,
// but if you do
// `ng build --env=prod` then `environment.prod.ts` will be used instead.
// The list of which env maps to which file can be found in `angular-cli.json`.

export const environment = {
  production: false
};
```

빌드

```
/
ng build
```

결과



6. 빌드 for 서비스(Production)

빌드 설정

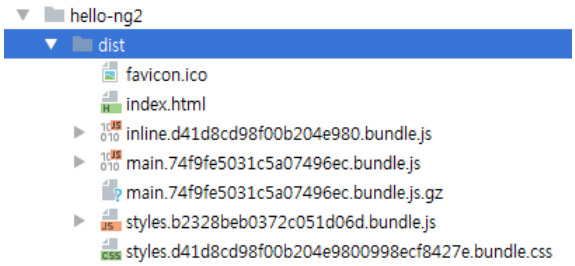
environment.prod.ts

```
export const environment = {
  production: true
};
```

빌드

```
/
ng build -prod
```

결과



추천 사이트

https://www.sitepoint.com/ultimate-angular-cli-reference/

실습 : Angular CLI

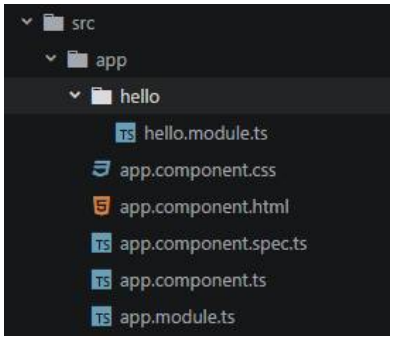
새 프로젝트 생성

```
C:\Lesson2\angular2\workspace>ng new hello-ng3
installing ng2
  create .editorconfig
  create README.md
  create src\app\app.component.css
  create src\app\app.component.html
  create src\app\app.component.spec.ts
  create src\app\app.component.ts
  create src\app\app.module.ts
  create src\assets\.gitkeep
  create src\environments\environment.prod.ts
  create src\environments\environment.ts
  create src\favicon.ico
  create src\index.html
  create src\main.ts
  create src\polyfills.ts
  create src\styles.css
  create src\test.ts
  create src\tsconfig.json
  create angular-cli.json
  create e2e\app.e2e-spec.ts
  create e2e\app.po.ts
  create e2e\tsconfig.json
  create .gitignore
  create karma.conf.js
  create package.json
  create protractor.conf.js
  create tslint.json
Successfully initialized git.
Installing packages for tooling via npm.
Installed packages for tooling via npm.
Project 'hello-ng3' successfully created.


C:\Lesson2\angular2\workspace>cd hello-ng3

C:\Lesson2\angular2\workspace\hello-ng3>dir
 C 드라이브의 볼륨에는 이름이 없습니다.
 볼륨 일련 번호: 6212-7067

 C:\Lesson2\angular2\workspace\hello-ng3 디렉터리

2017-02-23  오전 09:33    <DIR>          .
2017-02-23  오전 09:33    <DIR>          ..
2017-02-23  오전 09:32               245 .editorconfig
2017-02-23  오전 09:32               487 .gitignore
2017-02-23  오전 09:32             1,117 angular-cli.json
```

```
2017-02-23  오전 09:32    <DIR>          e2e
2017-02-23  오전 09:32             1,127 karma.conf.js
2017-02-23  오전 09:35    <DIR>          node_modules
2017-02-23  오전 09:32             1,379 package.json
2017-02-23  오전 09:32               757 protractor.conf.js
2017-02-23  오전 09:32             1,176 README.md
2017-02-23  오전 09:32    <DIR>          src
2017-02-23  오전 09:32             2,691 tslint.json
               8 개 파일               8,979 바이트
               5 개 디렉터리  395,677,585,408 바이트 남음
```

모듈 생성 / 생성된 모듈의 기동 컴포넌트 생성 및 설정

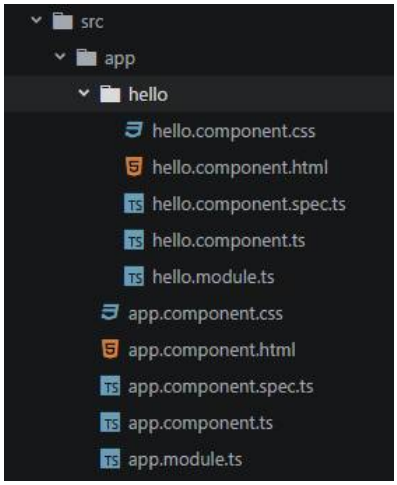
```
C:\Lesson2\angular2\workspace\hello-ng3>ng g module hello
installing module
  create src\app\hello\hello.module.ts
```



module 및 component 는 폴더가 없다면 폴더를 만들고 그 밑으로 파일을 제너레이트 한다.

service, directive, pipe 는 폴더를 만들지 않고 src/app 폴더 밑으로 파일을 제너레이트 한다.

```
C:\Lesson2\angular2\workspace\hello-ng3>ng g component hello
installing component
  create src\app\hello\hello.component.css
  create src\app\hello\hello.component.html
  create src\app\hello\hello.component.spec.ts
  create src\app\hello\hello.component.ts
update src\app\app.module.ts <-- 기본적으로 기동 모듈에 컴포넌트가 추가된다.
```



기동 모듈이 아니라 새로 만든 hello 모듈의 기동 컴포넌트로 사용하고 싶으므로 직접 설정을 변경하고 한다.

app.module.ts

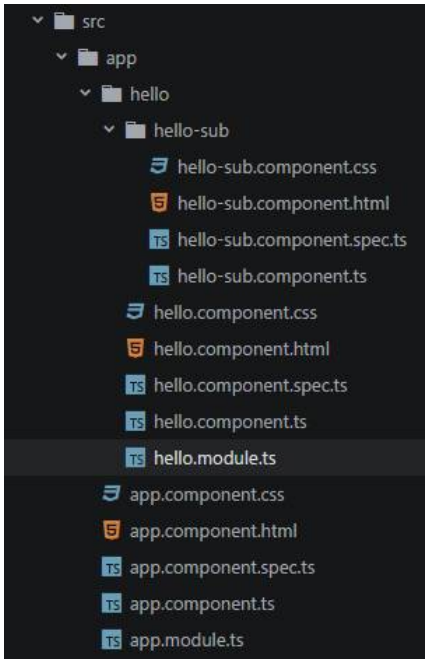
```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpModule } from '@angular/http';

import { AppComponent } from './app.component';
// import { HelloComponent } from './hello/hello.component'; // delete

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

hello.module.ts

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { HelloComponent } from './hello.component'; // add

@NgModule({
  imports: [
    CommonModule
  ],
  declarations: [
    HelloComponent, // add
  ]
})
export class HelloModule { }
```

컴포넌트 생성

```
C:\Lesson2\angular2\workspace\hello-ng3>ng g component hello/hello-sub
installing component
  create src\app\hello\hello-sub\hello-sub.component.css
  create src\app\hello\hello-sub\hello-sub.component.html
  create src\app\hello\hello-sub\hello-sub.component.spec.ts
  create src\app\hello\hello-sub\hello-sub.component.ts
  update src\app\hello\hello.module.ts
```



hello.module.ts

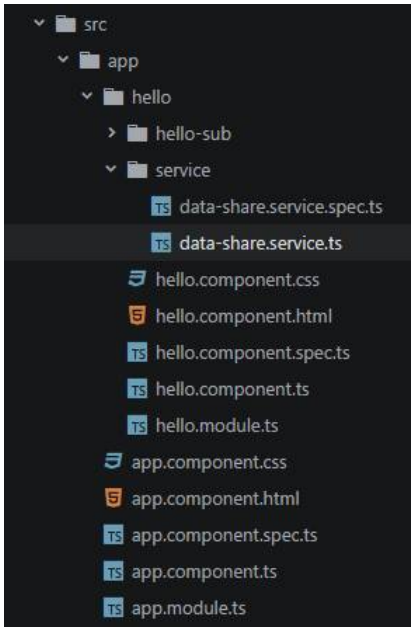
```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { HelloComponent } from './hello.component'; // add manually
import { HelloSubComponent } from './hello-sub/hello-sub.component'; // add automatically

@NgModule({
  imports: [
    CommonModule
  ],
  declarations: [
    HelloComponent, // add manually
    HelloSubComponent, // add automatically
  ]
})
export class HelloModule { }
```

서비스 생성 및 등록

서비스는 자동으로 폴더를 만들어 주지 않으므로 직접 만든다.

```
C:\Lesson2\angular2\workspace\hello-ng3>ng g service hello/service/data-share
installing service
Invalid path: "C:\Lesson2\angular2\workspace\hello-ng3\src\app\hello\service"" is not a valid path.

C:\Lesson2\angular2\workspace\hello-ng3>mkdir src/app/hello/service
명령 구문이 올바르지 않습니다.

C:\Lesson2\angular2\workspace\hello-ng3>cd src/app/hello && mkdir service && cd ../../..

C:\Lesson2\angular2\workspace\hello-ng3>
```

```
C:\Lesson2\angular2\workspace\hello-ng3>ng g service hello/service/data-share
installing service
  create src\app\hello\service\data-share.service.spec.ts
  create src\app\hello\service\data-share.service.ts
WARNING Service is generated but not provided, it must be provided to be used
```



서비스는 모듈에 자동으로 등록되지 않으므로 직접 등록한다.

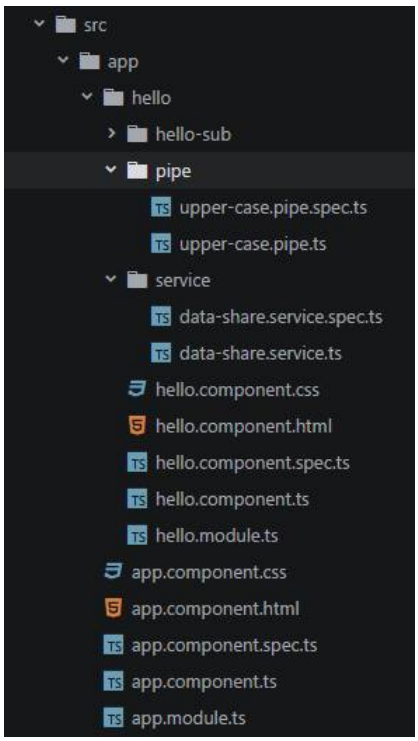
hello.module.ts

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { HelloComponent } from './hello.component'; // add manually
import { HelloSubComponent } from './hello-sub/hello-sub.component'; // add automatically

import { DataShareService } from './service/data-share.service'; // add manually

@NgModule({
  imports: [
    CommonModule
  ],
  declarations: [
    HelloComponent, // add manually
    HelloSubComponent, // add automatically
  ],
  providers: [DataShareService], // add manually
})
export class HelloModule { }
```

파이프 생성

파이프 파일을 모아 놓을 폴더를 만든 후 작업한다.

```
C:\Lesson2\angular2\workspace\hello-ng3>cd src/app/hello && mkdir pipe && cd ../../..

C:\Lesson2\angular2\workspace\hello-ng3>ng g pipe hello/pipe/upper-case
installing pipe
  create src\app\hello\pipe\upper-case.pipe.spec.ts
  create src\app\hello\pipe\upper-case.pipe.ts
update src\app\hello\hello.module.ts
```



hello.module.ts

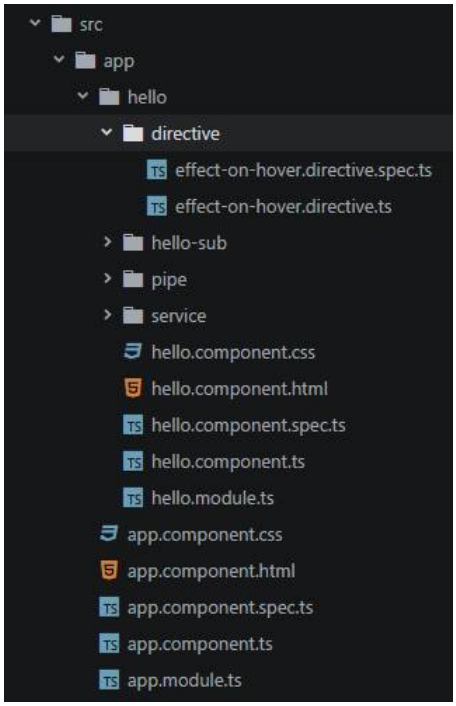
```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { HelloComponent } from './hello.component'; // add manually
import { HelloSubComponent } from './hello-sub/hello-sub.component'; // add automatically

import { DataShareService } from './service/data-share.service'; // add manually
import { UpperCasePipe } from './pipe/upper-case.pipe'; // add automatically

@NgModule({
  imports: [
    CommonModule
  ],
  declarations: [
```

```
    HelloComponent, // add manually
    HelloSubComponent, // add automatically
    UpperCasePipe, // add automatically
  ],
  providers: [DataShareService],
})
export class HelloModule { }
```

디렉티브 생성

디렉티브 파일을 모아 놓을 폴더를 만든 후 작업한다.

```
C:\Lesson2\angular2\workspace\hello-ng3>cd src/app/hello && mkdir directive && cd ../../..

C:\Lesson2\angular2\workspace\hello-ng3>ng g directive hello/directive/effect-on-hover
installing directive
  create src\app\hello\directive\effect-on-hover.directive.spec.ts
  create src\app\hello\directive\effect-on-hover.directive.ts
  update src\app\hello\hello.module.ts
```



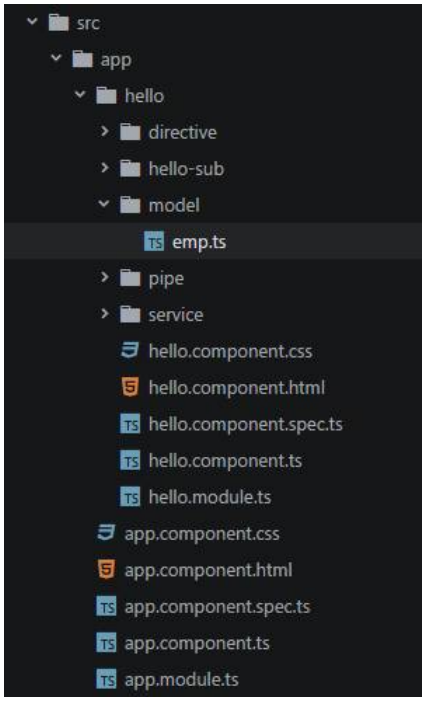
hello.module.ts

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { HelloComponent } from './hello.component'; // > add manually
import { HelloSubComponent } from './hello-sub/hello-sub.component'; // add automatically

import { DataShareService } from './service/data-share.service'; // > add manually
import { UpperCasePipe } from './pipe/upper-case.pipe'; // add automatically
import { EffectOnHoverDirective } from './directive/effect-on-hover.directive';
// add automatically

@NgModule({
  imports: [
    CommonModule
  ],
```

```
  declarations: [
    HelloComponent, // > add manually
    HelloSubComponent, // add automatically
    UpperCasePipe, // add automatically
    EffectOnHoverDirective, // add automatically
  ],
  providers: [DataShareService], // > add manually
})
export class HelloModule { }
```

클래스 생성

```
C:\Lesson2\angular2\workspace\hello-ng3>cd src/app/hello && mkdir model && cd ../../..

C:\Lesson2\angular2\workspace\hello-ng3>ng g class hello/model/emp
installing class
  create src\app\hello\model\emp.ts
```



Routing

새 프로젝트를 만든다.

HomeComponent, AboutComponent 컴포넌트를 만든다.

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 { RouterModule }   from '@angular/router';

import { AppComponent } from './app.component';
import { HomeComponent } from './home/home.component';
import { AboutComponent } from './about/about.component';

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    RouterModule.forRoot([
      {
        path: '',
        redirectTo: '/home',
        pathMatch: 'full'
      },
      {
        path: 'home',
        component: HomeComponent
      },
      {
        path: 'about',
        component: AboutComponent
      },

      {
        path: '**',
        component: HomeComponent
      },

    ])
  ],
  declarations: [
    AppComponent,
    HomeComponent,
    AboutComponent,

  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

네이게이션 링크를 사용자에게 노출 시킨다.

router-outlet 위치를 지정한다.

app.component.html

```
<h1>
  {{title}}
</h1>
<nav>
  <a routerLink="/home">Home</a>
  <a routerLink="/about">About</a>
</nav>
<div class="router-outlet">
  <router-outlet></router-outlet>
</div>
```

app.component.css

```
:host div.router-outlet {
  border: 1px dotted black;
}
```

@ViewChild

컴포넌트를 추가한다.

```
ng g component viewchild
```

viewchild.component.ts

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

@Directive({ selector: 'item' })
export class Item {
  @Input() status: boolean;
}

@Component({
  selector: 'item-component',
  template: '<button>알림창</button>'
})
export class ItemComponent {
  display(str:string){
    alert(str + 'ItemComponent 입니다');
  }
}

// <item-component (click)="display()">></item-component>
// 클릭 이벤트가 발생하면 ViewchildComponent 의 display() 함수가 호출된다.
@Component({
  selector: 'app-view-child',
  template: `
    <item status="false" *ngIf="isShow==false"></item>
    <item status="true" *ngIf="isShow==true"></item>
    <button (click)="toggle()">선택</button><br>
    isShow : {{isShow}}<br>
    status : {{status}}<br>
    <item-component (click)="display()"></item-component>`
})
export class ViewchildComponent {
  status: boolean;
  // 처음 실행할 때 isShow: boolean = true; 이므로
  // <item status="true" *ngIf="isShow==true"></item> 가 선택된다.
  // 따라서 item 디렉티브의 status 값은 true 로 초기에 설정된다.
  // 화면이 뜬 다음 토글 함수가 호출되면
  // toggle() --> !isShow 코드로 값이 반전된다.
  // 화면 갱신과정에서 *ngIf="isShow==false" 이므로
  // item 디렉티브의 status 값은 false 로 바뀐다.
  isShow: boolean = true;
```

```
  // @ViewChild 데코레이터를 사용해서 디렉티브인 Item 에 접근한다.
  // 디렉티브의 속성, 메소드 정보를 가진 객체를 세터의 파라미터로 전달 받는다.
  @ViewChild(Item)
  set item(v: Item) {
    // 디렉티브의 돔은 화면이 초기화되고 나서 접근할 수 있기 때문에 setTimeout 함수를 사용한다.
    setTimeout(() => { this.status = v.status; }, 0);
  }

  // @ViewChild(접근할 컴포넌트의 클래스명) 접근한 지시자의 돔을 가리키는 변수: 자료형;
  // @ViewChild 데코레이터를 사용해서 컴포넌트 ItemComponent 에 접근한다.
  @ViewChild(ItemComponent) itemComponent: ItemComponent;

  toggle() {
    console.log('ViewchildComponent.toggle() called');
    this.isShow = !this.isShow;
  }

  display(){
    console.log('ViewchildComponent.display() called');
  // ViewchildComponent 에서 display() 함수가 호출되면
  // ItemComponent 의 display() 함수를 호출한다.
    this.itemComponent.display('>');
  }

}
```

라우팅 설정을 업데이트 한다.

app.module.ts

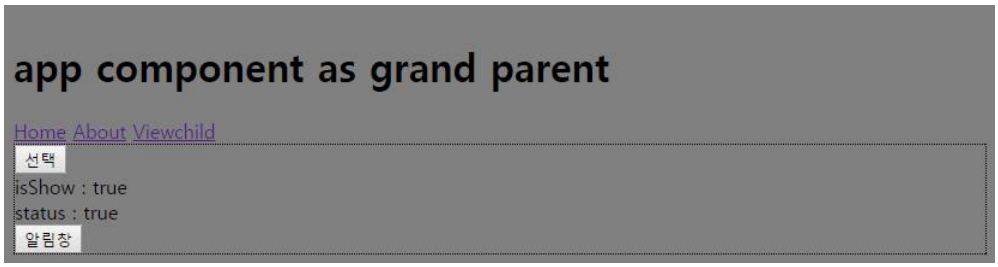
```
// import { ViewchildComponent } from './viewchild/viewchild.component'; // 아래처럼 변경
import { ViewchildComponent, Item, ItemComponent } from './viewchild/viewchild.component';

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    RouterModule.forRoot([
      {
        path: '',
        redirectTo: '/home',
        pathMatch: 'full'
      },
      {
        path: 'home',
        component: HomeComponent
      },
      {
        path: 'about',
        component: AboutComponent
      },
```

```
      {
        path: 'viewchild',
        component: ViewchildComponent
      },
      {
        path: '**',
        component: HomeComponent
      },
    ])
  ],
  declarations: [
    AppComponent,
    BookComponent,
    BookDetailComponent,
    HomeComponent,
    AboutComponent,
    // ViewchildComponent, // 아래처럼 변경
    ViewchildComponent, Item, ItemComponent
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

연동 링크를 부분을 업데이트 한다.

app.component.html

```
<nav>
  <a routerLink="/home">Home</a>
  <a routerLink="/about">About</a>
  <a routerLink="/viewchild">Viewchild</a>
</nav>
```



Communicating Between Components with Observable & Subject

http://jasonwatmore.com/post/2016/12/01/angular-2-communicating-between-components-with-observable-subject

서비스를 사용해 두 개의 컴포넌트가 대화할 수 있다.
컴포넌트 전환 시 라이프사이클 메소드 중 ngOnInit()가 기동하므로 초기화 작업으로 서비스에게서 데이터를 가져오면 된다.

그런데 두 개의 컴포넌트가 이미 모두 화면에 떠 있는 상태라면 초기화 메소드를 이용할 수 없으므로 다른 대안이 필요하다. rxjs 의 비동기 기술을 통해 처리하는 방법을 살펴보자.

A 컴포넌트 : publish → Observable ← subscribe : B 컴포넌트

Observable 객체는 EventEmitter 객체와 유사하게 작동한다. A 컴포넌트에서 이벤트가 발생하여 데이터가 변경되면 이를 Observable 에게 알린다. 그런 다음 Observable 은 구도신청을 한 B 컴포넌트에게 이를 통지하는 방식이다.

Module

https://github.com/wikibook/ng2-book.git

루트 모듈

앵귤러는 루트모듈이라는 최상위 모듈을 통해 앱을 구성한다.

앱 영역에서 사용하는 컴포넌트, 지시자, 파이프, 서비스 등과 같은 모듈을 등록하고 관리한다.

관심사에 따라 모듈을 분리하여 사용할 수 있다.

■ 핵심 모듈

항상 사용하는 모듈을 핵심 모듈로 보자. 예: 타이틀 컴포넌트

■ 특징 모듈

특정 기능을 처리하는 모듈을 분리하자. 관심사에 따라 모듈을 쪼개는 방법이다. 예: 게시판

■ 공유 모듈

여러 모듈에서 반복 사용되는 기능은 별개의 모듈로 만든다. 주로 특징 모듈에서 임포트해서 사용한다.

src/app/app.module.ts

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

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

/* 어플리케이션 라우팅 모듈 */
import { AppRoutingModule } from './app-routing.module';

/* 특징 모듈 */
import { MemberModule } from './member/member.module';
import { PlayerModule } from './player/player.module';
import { CoreModule } from './core/core.module';

/* 전역 컴포넌트 */
import { HelloComponent } from './hello/hello.component';
import { CoreTestComponent } from './core-test/core-test.component';

@NgModule({
  imports: [
    BrowserModule,CommonModule, FormsModule,
    MemberModule, PlayerModule,
    CoreModule.forRoot({nickName: 'Happy'}),
```

```
    AppRoutingModule
  ],
  providers: [],
  declarations: [
    AppComponent, IntroComponent,
    HelloComponent,
    CoreTestComponent
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

BrowserModule

브라우저 상에서 동작한다면 반드시 포함해야 한다.

컴포넌트, 지시자, 파이프 같은 구성요소를 템플릿에 표현하는 역할을 수행한다.

CommonModule

ngIf, ngFor 와 관련된 기능을 포함하고 있다.

BrowserModule 선언 시 생략할 수 있다.

FormsModule

템플릿에서 자주 사용하는 NgModel 지시자나 내장 검증기 지시자 등을 포함하고 있다.

app-routing.module

사용자 정의 라우팅 모듈이다. 루트 모듈에 추가한 라우팅 모듈은 앱수준에서 라우팅을 수행한다.

AppRoutingModule 키워드는 클래스명이다. imports 에 등록하면 라우팅등록이 완료된다.

declarations

앱 레벨에서 사용하고자 하는 컴포넌트, 지시자, 파이프를 선언한다.

bootstrap

앱 시작 컴포넌트를 등록한다.

CoreModule.*forRoot*({nickName: 'Happy'})

값 제공자에 매개변수로 값을 전달한다.

import { MemberModule } from './member/member.module';

특징모듈을 임포트한다.

src/app/app.component.ts

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

@Component({
  selector: 'app-root',
  template: `
  <div class="left-menu">
      <div class="menu">
        <a routerLink="/">
          <h1>모듈</h1>
        </a>
        <ol class="tree-list">
          <li><a routerLink="/hello" routerLinkActive="active">/hello 어플리케이션 루트모듈 라우팅 테스트</a></li>
          <li><a routerLink="/core-test" routerLinkActive="active">/core-test 핵심모듈 테스트</a></li>
          <li><a routerLink="/member-list" routerLinkActive="active">/member-list 특징모듈 테스트</a></li>
          <li><a routerLink="/player" routerLinkActive="active">/player 공유모듈 테스트</a></li>
          <li><a routerLink="/lazy/player" routerLinkActive="active">/lazyplayer/player 느린모듈 테스트</a></li>
        </ol>
    </div>
  </div>
  <div class="play-box">
    <router-outlet></router-outlet>
  </div>`
})
export class AppComponent { }
```

router-outlet

라우팅에 따라 하위 컴포넌트를 교체하는 영역이다.

src/app/app-routing.module.ts

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

import { IntroComponent } from './intro.component';
import { HelloComponent } from './hello/hello.component';
import { CoreTestComponent } from './core-test/core-test.component';

const appRoutes: Routes = [
  { path: '', component: IntroComponent },
  { path: 'hello', component: HelloComponent },
  { path: 'lazy', loadChildren: 'app/player/player.module#PlayerModule' },
  { path: 'core-test', component: CoreTestComponent }
];

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

router-outlet 에 표시할 컴포넌트를 라우팅 모듈 설정에 등록한다.

appRoutes

라우팅 설정을 담고 있다.

path 에 설정된 URL 요청을 지정된 컴포넌트로 라우팅되게 한다.

loadChildren: '모듈파일#모듈클래스'

게으른 모듈 로딩이다. 사용자가 요청하는 시점에 모듈을 비동기적으로 로딩한다.

{ path: 'lazy', loadChildren: 'app/player/player.module#PlayerModule' }

http://localhost:4200/lazy 요청이 들어오면

src/app/player/player-routing.module.ts 에 선언된 내용에 따라 처리한다.

```
  RouterModule.forChild([
    { path: '', redirectTo: 'player', pathMatch: 'full' },
    { path: 'player', component: PlayerComponent }
  ])
```

이 경우 루트패스로 접근하는 걸로 취급된다. path 가 '' 인 경우 player 요청으로 변환돼서 PlayerComponent 를 실행한다.

핵심 모듈

핵심 모듈 → 루트 모듈

src/app/core/title.component.ts

```
import { Component, Input } from '@angular/core';
import { UserService } from '../core/user.service';

@Component({
  selector: 'app-title',
  templateUrl: `
  <h1 highlight>{{title}}</h1>
  by <b>{{user}}</b>`,
})
export class TitleComponent {
  @Input() title = '';
  user = '';

  constructor(userService: UserService) {
    this.user = userService.nickName;
  }
}
```

@Input() title

<app-title [title]="타이틀"></app-title>

타이틀 컴포넌트를 다른 컴포넌트가 템플릿에서 위처럼 설정하면 title 속성을 통해 문자열을 받아 사용할 수 있다.

src/app/core/user.service.ts

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

export class UserServiceConfig {
  nickName = '';
}

@Injectable()
export class UserService {
  private _nickName = '';

  constructor( @Optional() config: UserServiceConfig) {
    if (config) { this._nickName = config.nickName; }
  }

  get nickName() {
    return this._nickName;
  }
}
```

@Optional

주입 객체가 있다면 사용하고 아니면 null 을 주입한다.

주입대상 객체가 없어서 발생하는 예외를 막는다.

get nickName()

게터메소드로 정의하면 서비스객체.nickName 과 같이 접근하여 값을 얻어갈 수 있다.

src/app/core/core.module.ts

```
import { ModuleWithProviders, NgModule, Optional, SkipSelf } from '@angular/core';
import { CommonModule } from '@angular/common';

import { TitleComponent } from './title.component';
import { UserService } from './user.service';
import { UserServiceConfig } from './user.service';

@NgModule({
  imports: [CommonModule],
  declarations: [TitleComponent],
  exports: [TitleComponent],
  providers: [UserService]
})
export class CoreModule {
  constructor( @Optional() @SkipSelf() parentModule: CoreModule) {
    if (parentModule) {
      throw new Error(
        'CoreModule 이 이미 로딩되었습니다.');
    }
  }

  static forRoot(config: UserServiceConfig): ModuleWithProviders {
    return {
      ngModule: CoreModule,
      providers: [
        { provide: UserServiceConfig, useValue: config }
      ]
    };
  }
}
```

exports: [TitleComponent]

핵심모듈을 루트모듈에 추가하기 위해서 외부로 노출한다.

constructor(@Optional() @SkipSelf() parentModule: CoreModule)

생성자에서 핵심모듈 CoreModule 을 주입받고자 시도한다. 부모 주입기에 핵심모듈이 이미 생성됐는지 검사한다.

@SkipSelf()

주입기 체계에서 자신은 건너 뛰고 자신으로부터 위에 존재하는 부모 주입기에서 CoreModule 을 찾는다.

static *forRoot*(config: UserServiceConfig): ModuleWithProviders

루트 모듈에서 CoreModule.*forRoot*({nickName: 'Happy'}) 설정으로 객체를 매개변수로 넘기면 받아서 사용한다.

특징 모듈

만약 루트모듈에 모든 모듈을 구성하려고 한다면 모듈 구성이 복잡해질 것이다.

따라서 루트 모듈에서 하위 모듈로 분리할 필요가 있다.

이렇게 하위로 분리하는 모듈을 특징 모듈이라 부른다.

특징 모듈은 관심사가 같은 모듈을 묶어 상위 모듈에게 제공한다.

라우팅 모듈은 일반적으로 분리해서 특징모듈로써 사용한다.

특징 모듈 라우팅 모듈 → 특징 모듈 → 루트 모듈 ← 라우팅 모듈(앱 레벨)

src/app/member/highlight.directive.ts

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

@Directive({
  selector: '[highlight]'
})
export class HighlightDirective {
  private el: HTMLElement;

  constructor(el: ElementRef) {
    this.el = el.nativeElement;
    this.el.style.fontSize = "30px";
  }
  @HostListener('mousemove') onMouseMove() {
    this.el.style.backgroundColor = "blue";
    this.el.style.color ="white";
  }
  @HostListener('mouseleave') onMouseLeave() {
    this.el.style.backgroundColor = null;
    this.el.style.color ="black";
  }
}
```

@HostListener

디렉티브가 사용되는 엘리먼트에서 발생하는 이벤트를 듣는다. 등록된 이벤트에 따라 지시자 선언 엘리먼트 영역에 마우스가 위치하면 배경색을 blue 로 바꾼다.

src/app/member/member.service.ts

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

export class Member {
  constructor(public name: string, public age: number) { }
}

const MEMBERS: Member[] = [
  new Member('유비',30),
  new Member('관우',29),
  new Member('장비',28)
];

@Injectable()
export class MemberService {

  getMembers() {
    return new Promise<Member[]>(resolve => {
      setTimeout(() => { resolve(MEMBERS); }, 500);
    });
  }

  getMember(name: string) {
    return this.getMembers()
      .then(members => members.find(member => member.name == name));
  }

}
```

src/app/member/member-list.component.ts

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

import { Member, MemberService } from './member.service';

@Component({
  selector: 'member-list',
  template: `
  {{name}}의 나이 : {{age}}<br>
  <div *ngFor='let m of members' highlight (mouseover)="setAge(m.name)">
    {{m.name}} {{m.age}}
  </div>`
})
export class MemberListComponent {
  members: Member[];
  age: number;
  name: string;

  constructor(private memberService: MemberService) {}

  ngOnInit() {
    this.memberService.getMembers().then(members => {
      this.members = members;
    });
    this.setAge("유비");
  }

  setAge(name: string){
    this.name = name;
    this.memberService.getMember(name).then(member => {
      this.age = member.age;
    });
  }
}
```

this.memberService.getMembers()

서비스로부터 데이터를 받는다.

src/app/member/member.module.ts

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

import { MemberListComponent }   from './member-list.component';
import { HighlightDirective } from './highlight.directive';
import { MemberRoutingModule }   from './member-routing.module';
import { MemberService }     from './member.service';

@NgModule({
  imports:      [ CommonModule, FormsModule, MemberRoutingModule ],
  declarations: [ MemberListComponent, HighlightDirective ],
  providers:    [ MemberService ]
})
export class MemberModule { }
```

특징 모듈은 루트 모듈의 하위로 관심사를 별도로 갖고 있다.

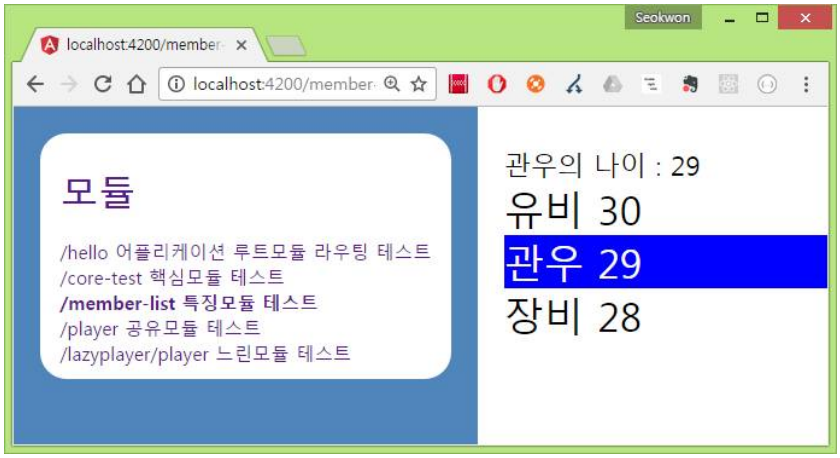
특징 모듈은 bootstrap 속성을 사용하지 않는다.

src/app/member/member-routing.module.ts

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

import { MemberListComponent }    from './member-list.component';

@NgModule({
  imports: [RouterModule.forChild([
    { path: 'member-list', component: MemberListComponent}
  ])],
  exports: [RouterModule]
})
export class MemberRoutingModule {}
```

RouterModule.forChild

루트 모듈의 라우팅 모듈에 설정하지 않고 특징 모듈의 라우팅 모듈에서 특징 모듈과 관련된 라우팅 설정을 추가한다.



공유 모듈

자주 반복적으로 사용되는 모듈이 공유 모듈 대상이다.

루트 모듈은 특징 모듈을 임포트하고 특징 모듈은 공유 모듈을 임포트한다.

공유 모듈 → 특징 모듈 → 루트 모듈

BrowserModule, FormsModule 은 일반적으로 공유 모듈로 묶고 특징 모듈에 제공해서 사용한다.

src/app/shared/highlight.directive.ts

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

@Directive({
  selector: '[highlight]'
})
export class HighlightDirective {

  private el: HTMLElement;
  constructor(el: ElementRef) {
    this.el = el.nativeElement;
    this.el.style.fontSize = "30px";
  }
  @HostListener('mousemove') onMouseMove() {
    this.el.style.backgroundColor = "red";
    this.el.style.color = "white";
  }
  @HostListener('mouseleave') onMouseLeave() {
    this.el.style.backgroundColor = null;
    this.el.style.color = "black";
  }

}
```

src/app/shared/my-upper.pipe.ts

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

@Pipe({ name: 'myupper' })
export class MyUpperPipe implements PipeTransform {
  transform(phrase: string) {
    return phrase.toUpperCase();
  }
}
```

src/app/shared/shared.module.ts

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

import { MyUpperPipe } from './my-upper.pipe';
import { HighlightDirective } from './highlight.directive';

@NgModule({
  imports: [CommonModule],
  declarations: [MyUpperPipe, HighlightDirective],
  exports: [MyUpperPipe, HighlightDirective,
    CommonModule, FormsModule]
})
export class SharedModule { }
```

src/app/player/player.module.ts

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

import { SharedModule } from '../shared/shared.module';

import { PlayerRoutingModule } from './player-routing.module';
import { PlayerComponent } from './player.component';

@NgModule({
  imports: [ PlayerRoutingModule, SharedModule],
  declarations: [PlayerComponent],
  providers: []
})
export class PlayerModule { }
```

src/app/player/player.component.ts

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

@Component({
  selector: 'player',
  template: `<div highlight>{{"player!!!"|myupper}}</div>`
})
export class PlayerComponent { }
```

highlight

해당 디렉티브는 모듈에서 임포트한 SharedModule 에서 제공한다.

myupper

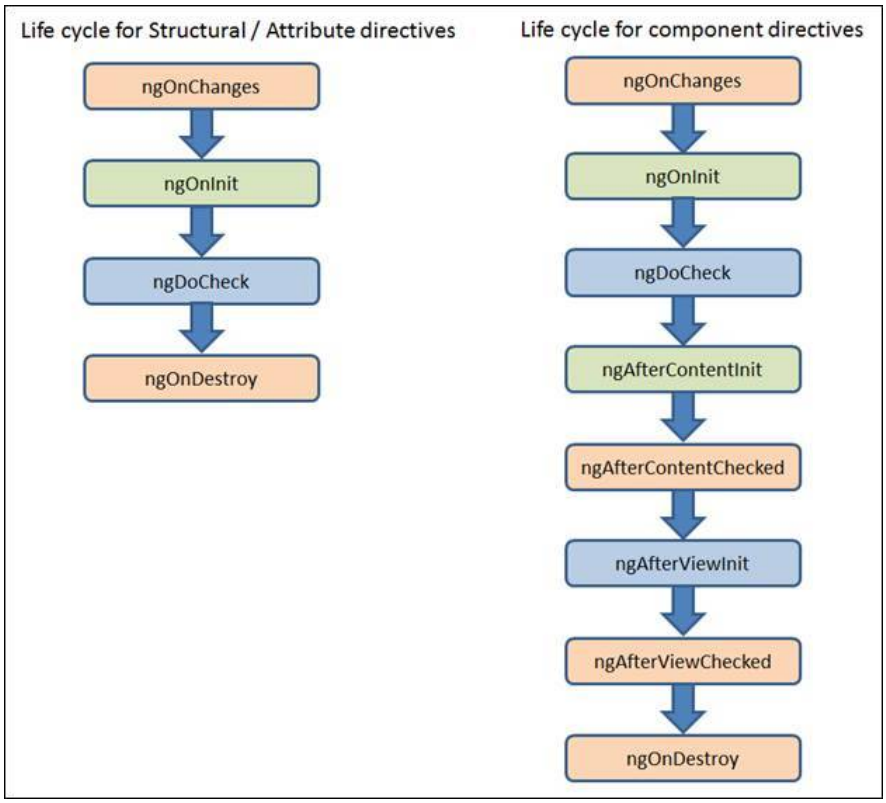
해당 파이프는 모듈에서 임포트한 SharedModule 에서 제공한다.

src/app/player/player-routing.module.ts

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

import { PlayerComponent } from './player.component';

@NgModule({
  imports: [RouterModule.forChild([
    { path: '', redirectTo: 'player', pathMatch: 'full' },
    { path: 'player', component: PlayerComponent }
  ])],
  exports: [RouterModule]
})
export class PlayerRoutingModule { }
```

Life Cycle

컴포넌트와 디렉티브는 생명주기를 갖는다.



	event	init	status	설명
ngOnChanges	v			속성 바인딩의 상태 값이 변경될 때 호출
ngOnInit		v		속성 바인딩 후에 컴포넌트와 지시자가 초기화될 때 호출
ngDoCheck			v	엘리먼트의 상태가 변할 때마다 호출
ngAfterContentInit		v		컴포넌트의 컨텐츠가 컴포넌트 뷰에 입력되고 나서 호출
ngAfterContentChecked			v	프로젝트의 외부 컨텐츠와 바인딩을 점검한 후 호출
ngAfterViewInit		v		컴포넌트의 뷰와 자식 컴포넌트 뷰가 초기화된 후 호출
ngAfterViewChecked			v	컴포넌트나 자식뷰의 바인딩을 점검한 후 호출
ngOnDestroy	v			컴포넌트나 디렉티브가 제거되기 전 실행

용어정리

```
<parent [prop]="..">
  <child name=""></child>
</parent>
```

parent 컴포넌트 입장에서 컨텐츠는 child 컴포넌트이다.

```
@Component({
  selector: 'parent',
  template: `<child id="1" *ngIf="status"></child>`
})
```

parent 컴포넌트 입장에서 뷰는 child 컴포넌트이다.

src/app/cmp-lifecycle/my-cmp.component.ts

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

@Component({
    selector: 'my-cmp',
    template: ``
})
export class MyCmp {
    @Input() value: string;
}
```

부모 컴포넌트에 포함되어 상태 값을 받아서 사용될 컴포넌트이다.

src/app/cmp-lifecycle/window.directive.ts

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

@Directive({ selector: 'window' })
export class Window {
    @Input() id: string;
}
```

src/app/cmp-lifecycle/child-cmp.component.ts

```
import {
    Component,
    Input,
    ContentChild,
    ViewChild,
    OnChanges,
    DoCheck,
    OnInit,
    AfterContentInit,
    AfterContentChecked,
    AfterViewInit,
    AfterViewChecked,
    OnDestroy
} from '@angular/core';
import {MyCmp} from './my-cmp.component';
import {Window} from './window.directive';

@Component({
    selector: 'child-cmp',
    template: `<input type="text" [(ngModel)]="message" placeholder="바인딩 있음"> <input
type="text" placeholder="바인딩 없음">
    <window id="1" *ngIf="shouldShow"></window>
    <window id="2" *ngIf="!shouldShow"></window>
    <button (click)="toggle()">View 상태변경</button>`
})
export class ChildCmp implements OnChanges, DoCheck, OnInit, AfterContentInit,
AfterContentChecked, AfterViewInit, AfterViewChecked, OnDestroy {
    message: string = "";
    oldMessage: string = "";

    constructor() {
        console.log("\n1-컴포넌트 : constructor()");
    }

    @Input()
    set prop(name: string) {
        console.log("@Input prop() 세터 메서드 호출");
    }

    ngOnChanges() {
        console.log("2--컴포넌트 : ngOnChanges()");
    }

    ngOnInit() {
        console.log("3---컴포넌트 : ngOnInit()");
    }

    ngDoCheck() {
        console.log("4----컴포넌트 : ngDoCheck()");
        if (this.message != this.oldMessage) {
            console.log("4----컴포넌트 : ngDoCheck()에서 변화감지");
            this.oldMessage = this.message;
        }
    }

    @ContentChild(MyCmp) myCmp: MyCmp;
```

```
    oldMyCmpId: any;

    ngAfterContentInit() {
        console.log("5-----컴포넌트 : ngAfterContentInit()");
        this.oldMyCmpId = this.myCmp.value;

    }

    ngAfterContentChecked() {
        console.log("6------컴포넌트 : ngAfterContentChecked()");

        if (this.oldMyCmpId != this.myCmp.value) {
            console.log("6------컴포넌트 : ngAfterContentChecked()에서 변화감지");
            this.oldMyCmpId = this.myCmp.value;
        }
    }


    shouldShow = true;

    toggle() {
        this.shouldShow = !this.shouldShow;
    }

    @ViewChild(Window) window;
    oldWindowId: any;

    ngAfterViewInit() {
        console.log("7-------컴포넌트 : ngAfterViewInit()");
        this.oldWindowId = this.window.id;
    }

    ngAfterViewChecked() {
        console.log("8--------컴포넌트 : ngAfterViewChecked()");
        if (this.oldWindowId != this.window.id) {
            console.log("8--------컴포넌트 : ngAfterViewChecked()에서 변화감지");
            this.oldWindowId = this.window.id;
        }
    }

    ngOnDestroy() {
        console.log("9---------컴포넌트 : ngOnDestroy()");
    }
}
```

src/app/cmp-lifecycle/child-lifecycle.component.ts

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

@Component({
    selector: 'cmp-lifecycle',
    template: `
    <button (click)="toggle()">Content 상태변경</button>
    <button (click)="isShow=!isShow">
      {{isShow==false?'컴포넌트 추가':'컴포넌트 삭제'}}
    </button>
    <br>
    <child-cmp *ngIf="isShow" [prop]="shouldShow">
        <my-cmp value="{{shouldShow}}"></my-cmp>
    </child-cmp>`
})
export class CmpLifecycleComponent {
    isShow: boolean = false;

    shouldShow = true;
    toggle() { this.shouldShow = !this.shouldShow; }
}
```

테스트

Content 상태변경 버튼 클릭

toggle() { this.shouldShow = !this.shouldShow; }

클릭하면 toggle 메소드를 호출해서 값을 바꾼다.

child-cmp

값을 사용하는 컴포넌트가 현재 화면에 표시되고 있지 않기 때문에 훅 메소드를 호출하지 않는다.

컴포넌트 추가 버튼 클릭

(click)="isShow=!isShow"

값을 토글한다.

```
    <child-cmp *ngIf="isShow" [prop]="shouldShow">
       <my-cmp value="{{shouldShow}}"></my-cmp>
    </child-cmp>`
```

컴포넌트를 화면에 표시한다.

```
1-컴포넌트 : constructor()
@Input prop() 세터 메서드 호출                    child-cmp.component.ts:35
2--컴포넌트 : ngOnChanges()                      child-cmp.component.ts:39
3---컴포넌트 : ngOnInit()                        child-cmp.component.ts:43
4----컴포넌트 : ngDoCheck()                      child-cmp.component.ts:47
5-----컴포넌트 : ngAfterContentInit()            child-cmp.component.ts:58
6------컴포넌트 : ngAfterContentChecked()         child-cmp.component.ts:64
7-------컴포넌트 : ngAfterViewInit()             child-cmp.component.ts:83
8--------컴포넌트 : ngAfterViewChecked()          child-cmp.component.ts:88
4----컴포넌트 : ngDoCheck()                      child-cmp.component.ts:47
6------컴포넌트 : ngAfterContentChecked()         child-cmp.component.ts:64
8--------컴포넌트 : ngAfterViewChecked()          child-cmp.component.ts:88
```

컴포넌트 시퀀스 훅 메소드 7 개가 차례대로 실행된다.

```
@Component({
    selector: 'child-cmp',
    template: `<input type="text" [(ngModel)]="message" placeholder="바인딩 있음"> <input type="text" placeholder="바인딩 없음">
    <window id="1" *ngIf="shouldShow"></window>
    <window id="2" *ngIf="!shouldShow"></window>
    <button (click)="toggle()">View 상태변경</button>`
})
```

[(ngModel)]="message"

ngModel 바인딩을 처리하기 위해서 상태체크 훅 메소드 3 개가 실행된다.

Content 상태변경 버튼 클릭

컴포넌트가 추가된 상태에서 Content 상태변경 버튼을 클릭한다.

```
@Input prop() 세터 메서드 호출                                    child-cmp.component.ts:35
2--컴포넌트 : ngOnChanges()                                       child-cmp.component.ts:39
4----컴포넌트 : ngDoCheck()                                       child-cmp.component.ts:47
6------컴포넌트 : ngAfterContentChecked()                          child-cmp.component.ts:64
6------컴포넌트 : ngAfterContentChecked()에서 변화감지              child-cmp.component.ts:67
8--------컴포넌트 : ngAfterViewChecked()                          child-cmp.component.ts:88
```

생명주기보다 클래스가 우선적으로 처리되기 때문에 먼저 세터 메소드가 호출된다.
컴포넌트 엘리먼트 내부에 선언된 엘리먼트 속성이 변경되면 상태 체크 훅 메소드들이 실행된다.

```
@Input()
set prop(name: string) {
    console.log("@Input prop() 세터 메서드 호출");
}
```

child-cmp 의 자식 엘리먼트인 my-cmp 컴포넌트에 선언된 속성 값을 받는다.

```
@ContentChild(MyCmp) myCmp: MyCmp;
```

이전 값과 달라졌기 때문에 '변화감지' 로그가 출력된다.

```
ngAfterContentChecked() {
    console.log("6------컴포넌트 : ngAfterContentChecked()");

    if (this.oldMyCmpId != this.myCmp.value) {
        console.log("6------컴포넌트 : ngAfterContentChecked()에서 변화감지");
        this.oldMyCmpId = this.myCmp.value;
    }
}
```

아래 코드에서 [prop]="shouldShow" 속성에 선언된 상태가 변경되면 내부 엘리먼트인 my-cmp 의 값 value 가 @ContentChild 데코레이터를 통해 전달된다.

```
@Component({
    selector: 'cmp-lifecycle',
    template: `
    <button (click)="toggle()">Content 상태변경</button>
    <button     (click)="isShow=!isShow">{{isShow==false?'컴포넌트     추가':'컴포넌트     삭제
'}}</button>
    <br>
    <child-cmp *ngIf="isShow" [prop]="shouldShow">
        <my-cmp value="{{shouldShow}}"></my-cmp>
    </child-cmp>`
})
```

View 상태변경 버튼 클릭

```
4----컴포넌트 : ngDoCheck()                                    child-cmp.component.ts:47
6------컴포넌트 : ngAfterContentChecked()                      child-cmp.component.ts:64
8--------컴포넌트 : ngAfterViewChecked()                       child-cmp.component.ts:88
8--------컴포넌트 : ngAfterViewChecked()에서 변화감지           child-cmp.component.ts:90
```

토글 메소드가 호출되고 shouldShow 값이 변경된다.

```
@Component({
    selector: 'child-cmp',
    template: `<input type="text" [(ngModel)]="message" placeholder="바인딩 있음"> <input
type="text" placeholder="바인딩 없음">
    <window id="1" *ngIf="shouldShow"></window>
    <window id="2" *ngIf="!shouldShow"></window>
    <button (click)="toggle()">View 상태변경</button>`
})
```

shouldShow 가 참이면 id 는 1 인 상태를 갖는다.

```
@Directive({ selector: 'window' })
export class Window {
    @Input() id: string;
}
```

shouldShow 가 바뀌면 id 의 상태 값이 바뀐다.

window 디렉티브는 @Input 데코레이터를 통해 값을 받는다.

```
@ViewChild(Window) window;
oldWindowId: any;

// 받아들이고 바인딩이 이루어진 시점에 호출
ngAfterViewChecked() {
    console.log("8--------컴포넌트 : ngAfterViewChecked()");
    if (this.oldWindowId != this.window.id) {
        console.log("8--------컴포넌트 : ngAfterViewChecked()에서 변화감지");
        this.oldWindowId = this.window.id;
    }
}
```

@ViewChild(Window) window;

window 디렉티브가 가지고 있는 상태 값 id 를 가져올 수 있다.

왼쪽 입력 엘리먼트에 문자를 입력

h 가 입력되는 순간 입력 엘리먼트의 상태가 변했기 때문에 관련 훅 메소드가 호출된다.

```
4----컴포넌트 : ngDoCheck()                          child-cmp.component.ts:47
4----컴포넌트 : ngDoCheck()에서 변화감지              child-cmp.component.ts:49
6------컴포넌트 : ngAfterContentChecked()             child-cmp.component.ts:64
8--------컴포넌트 : ngAfterViewChecked()              child-cmp.component.ts:88
```

컴포넌트 삭제 버튼 클릭


```
9---------컴포넌트 : ngOnDestroy()                    child-cmp.component.ts:96
```

DI

https://github.com/m00s/angular2features

Dependency resolution only walks up the tree.

```
class Car {
  constructor(e: Engine){}
}
```

DI will start resolving Engine in the same injector where the Car binding is defined. It will **check whether that injector has the Engine binding**. If it is the case, it will return that instance. If not, the injector will ask its parent whether it has an instance of Engine. The process continues **until** either an instance of Engine has been found, or we have reached **the root of the injector tree**.

```
class Car {
  constructor(@Self() e: Engine){}
}
```

@Self decorator tells DI to look for Engine **only in the same injector where Car is defined**. So it will not walk up the tree.

```
class Car {
  constructor(@Host() e: Engine){}
}
```

The @Host decorator tells DI to look for Engine in this injector, its parent, **until it reaches a host**.

```
class Car {
  constructor(@SkipSelf() e: Engine){}
}
```

The @SkipSelf decorator tells DI to look for Engine in the whole tree starting **from the parent injector**.

앵귤러는 생성자 의존성 주입 패턴을 사용한다. 의존성 주입을 활용하면 객체 생성과 설정에 들어가는 코드를 최소화하고 컴포넌트마다 일관된 방법으로 객체를 사용할 수 있다.

컴포넌트 구성에 따라 injector tree 가 만들어진다. 주입기 트리는 컴포넌트 구성과 일치한다. 따라서 컴포넌트마다 하나의 주입기를 갖는다. 주입기 트리에서 최상위 주입기를 root injector 라고 부른다.

만약 주입 요청이 들어오고 찾는 주입 대상이 현재 주입기에 없다면 상향식으로 상위 컴포넌트에 설정한 providers 설정에서 주입 대상을 찾는다. 이러한 이유로 루트 컴포넌트에 주입 설정한 서비스의 경우 최상위 컴포넌트에 선언됐기 때문에 전역 서비스로 사용할 수 있다.

Providers

@Injector(주입할 클래스를 선택)

→ Providers(주입할 클래스를 등록)

→ Dependency Injection(생성자로 의존성을 주입 받음)

주입할 클래스는 @Injectable 데코레이터를 사용한다. 생략이 가능하다.
변수나 함수는 @Injectable 데코레이터를 사용할 수 없다.

제공자의 종류

■ Value Provider

■ Factory Provider

■ Class Provider

Value Provider

src/app/value-provider/value.provider.ts

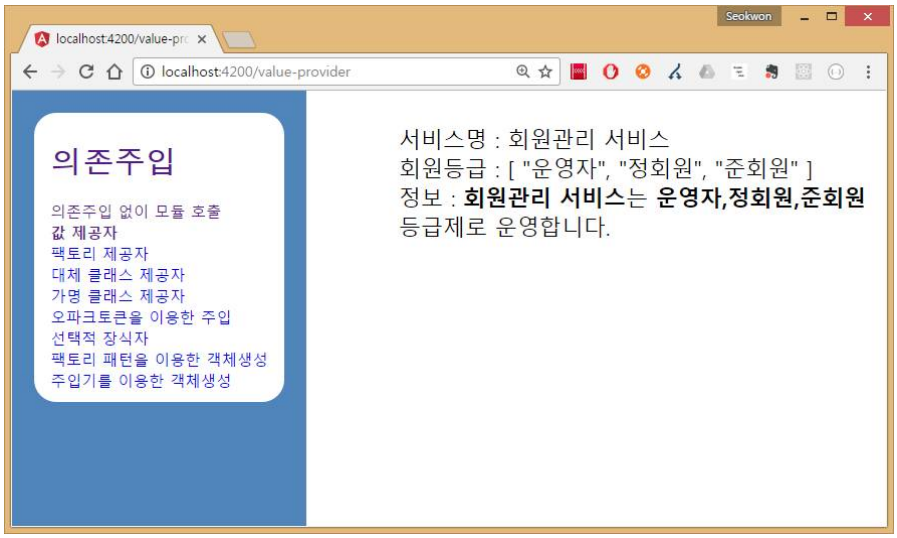
```
import {Injectable} from '@angular/core';

// 값 제공자에서 사용할 클래스
@Injectable()
export class Config {
    serviceName: string;
    grade: string[] = [];

    getInfo() {
    }
}

// 사용할 값 변수
let myConfig = {
    serviceName: '회원관리 서비스',
    grade: ['운영자', '정회원', '준회원'],
    getInfo: () => {
        return `<b>${myConfig.serviceName}</b>는 <b>${myConfig.grade}</b> 등급제로 운영합니다.`;
    }
};

// 값 제공자
export let ValueProvider = {
    provide: Config,
    useValue: myConfig
};
```

myConfig 변수는 Config 클래스형으로 정의하지 않는다. 값 제공자는 Config 클래스와 이에 대한 값을 저장하고 있는 myConfig 변수를 합해 하나의 객체로 객체로 만드는 역할을 한다.

src/app/value-provider/value-provider.component.ts

```
import {Component} from '@angular/core';
import {ValueProvider, Config} from './value.provider';

@Component({
    selector: 'app-value-provider',
    template: `<div>서비스명 : {{config.serviceName}}</div>
  <div>회원등급 : {{config.grade|json}}</div>
  <div>정보 : <span [innerHTML]="config.getInfo()"></span></div>`,
    providers: [ValueProvider]
})
export class ValueProviderComponent {
    constructor(public config: Config) {
    }
}
```

주입된 config 객체는 값 제공자인 ValueProvider 에 의해 myConfig 변수에 설정된 값으로 초기화된다.



Factory Provider

의존성 주입 시 기본적으로 싱글톤 객체가 주입된다. 때로는 싱글톤이 아닌 새롭게 생성한 객체를 의존성 주입을 통해 사용할 때가 있다.

src/app/factory-provider/car.service.ts

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

@Injectable()
export class Engine {
    public name: string;
}

@Injectable()
export class Speedmeter {
    public speed: number;
    public maxSpeed: number;
}

@Injectable()
export class CarService {
    constructor(private speedmeter: Speedmeter, public engine: Engine) {
    }
}
```

팩토리 함수에 사용할 서비스 클래스를 정의했다.

src/app/factory-provider/car.service.provider.ts

```
import {Engine, Speedmeter, CarService} from './car.service';

let carServiceFactory = (speedmeter: Speedmeter, engine: Engine) => {
    speedmeter.speed = 100;
    speedmeter.maxSpeed = 500;
    engine.name = "슈퍼엔진";
    return new CarService(speedmeter, engine);
};

export let FactoryProvider = {
    provide: CarService,
    useFactory: carServiceFactory,
    deps: [Speedmeter, Engine]
};
```

provide: CarService

최종적으로 사용할 객체의 클래스

useFactory: ***carServiceFactory***

팩토리 메소드

deps: [**Speedmeter**, **Engine**]

의존 클래스

src/app/factory-provider/factory-provider.component.ts

```
import {Component} from '@angular/core';
import {CarService} from './car.service';
import {FactoryProvider} from './car.service.provider';

@Component({
    selector: 'app-factory-provider',
    template: `
  <div>엔진이름 : {{carService.engine.name}}</div>
  <div>현재속도 : {{carService.speedmeter.speed}} km/h</div>
  <div>최대속도 : {{carService.speedmeter.maxSpeed}} km/h</div>`,
    providers: [FactoryProvider]
})
export class FactoryProviderComponent {
    constructor(public carService: CarService) {
    }
}
```

객체 설정이나 객체 생성 시 필요한 의존 객체를 컴포넌트 내부가 아닌 컴포넌트 외부에 설정했다. CarService 를 주입할 때 FactoryProvider 의 carServiceFactory 메소드가 새로 만들어 리턴하는 new CarService(speedmeter, engine) 객체를 사용한다.

Class Provider

대체 클래스 제공자

alternative class provider 는 제공할 클래스를 다른 클래스로 대체하는 의존성 주입 방법을 제공한다.

src/app/class-provider/engine.service.ts

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

@Injectable()
export class Engine {
    public name: string = "엔진";
}
```

src/app/class-provider/super-power-engine.service.ts

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

@Injectable()
export class SuperPowerEngine {
    public name: string = "슈퍼엔진";
    public description = '세계 최고의 성능을 자랑하는 슈퍼엔진';
}
```

src/app/class-provider/class-provider.component.ts

```
import {Component} from '@angular/core';
import {SuperPowerEngine as spEngine} from './super-power-engine.service';
import {Engine} from './engine.service';

@Component({
    selector: 'app-class-provider',
    template: `
  {{engine.name}}<br>
  {{spEngine.name}}<br>
  {{result}}`,
    providers: [spEngine, {provide: Engine, useClass: spEngine}]
})
export class ClassProviderComponent {
    result: string;

    constructor(public engine: Engine, public spEngine: spEngine) {
        if (engine === spEngine) {
            this.result = "두 객체는 동일 객체입니다.";
            throw new Error('Error');
        } else {
            this.result = "두 객체는 다른 객체입니다.";
        }
```

```
    }
}
```

providers: [spEngine, **{provide: Engine, useClass: spEngine}**]

provide 는 의존성 주입 시 사용할 클래스이다. 실제 클래스의 내용은 useClass 에 선언된 대체 클래스를 사용한다. providers 의 첫 번째 토큰으로 spEngine 을 설정했지만 대체 클래스 제공자는 이를 사용하지 않고 또 하나의 spEngine 객체를 만들어서 사용한다.

constructor(public engine: Engine, public spEngine: spEngine)

주입 시 engine 객체가 spEngine 객체로 대체된다.

가명 클래스 제공자

제공자에서 동일한 기능을 하는데 객체가 두 개 만들어지는 것은 바람직하지 않다.

싱글톤 객체를 사용하기 위해 가명 클래스 제공자를 사용한다.

src/app/class-provider/aliased-class-provider.component.ts

```
import { Component } from '@angular/core';
import { SuperPowerEngine as spEngine } from './super-power-engine.service';
import { Engine } from './engine.service';

@Component({
  selector: 'app-aliased-class-provider',
  template: `
  {{engine.name}}<br>
  {{spEngine.name}}<br>
  {{result}}`,
  providers: [spEngine, { provide: Engine, useExisting: spEngine }]
})
export class AliasedClassProviderComponent {
  result:string;
  constructor(public engine: Engine,public spEngine:spEngine) {
    if (engine === spEngine){
      this.result="두 객체는 동일 객체입니다.";
    }else{
      this.result="두 객체는 다른 객체입니다.";
        throw new Error('Error');
    }
  }
}
```

providers: [spEngine, **{ provide: Engine, useExisting: spEngine }**]

Engine 객체가 spEngine 객체로 대체될 때 spEngine 객체가 이미 선언돼 있다면 있는 것을 그대로 사용한다.

불투명 토큰을 이용한 제공자 설정

opaque token 은 의존성 주입 시 인터페이스를 주입받기 위해서 사용한다.

src/app/opaque-token/opaque-token.provider.ts

```
import { Inject, OpaqueToken } from '@angular/core';

export let OPAQUE_TOKEN = new OpaqueToken('OPAQUE_TOKEN');

export interface Config {
    endpointURL: string;
    PORT: string;
}

// 인터페이스 형 변수
export const MY_API_CONFIG: Config = {
    endpointURL: 'http://192.168.0.1:80/rest',
    PORT: '8000'
};

export let OpaqueTokenProvider = {
    provide: OPAQUE_TOKEN,
    useValue: MY_API_CONFIG
};
```

타입스크립트는 의존성 주입 대상 객체에 대해 인터페이스 형을 허용하지 않는다. 인터페이스 형이어도 의존성 주입이 가능하게 하려면 provide 값을 불투명 토큰 객체로 처리해야 한다.
불투명 토큰 객체가 설정되면 useValue 값이 불투명 처리된다. 그러면 인터페이스 형일지라도 의존성 주입이 가능해진다.

src/app/opaque-token/opaque-token.component.ts

```
import { Component, Inject } from '@angular/core';
import { MY_API_CONFIG, OpaqueTokenProvider } from './opaque-token.provider';

@Component({
  selector: 'app-inject-decorator',
  template: `
 <div>API URL: {{appConfig.endpointURL}}</div>
 <div>API PORT: {{appConfig.PORT}}</div>`,
 providers: [{provide:OpaqueTokenProvider, useValue:MY_API_CONFIG}]
})
export class OpaqueTokenComponent {
  constructor(@Inject(OpaqueTokenProvider) public appConfig) {}
}
```

@Inject(불투명 토큰을 설정한 제공자) public 주입 대상 변수 or 클래스

Provider 없이 객체 DI

외부에서 객체를 생성하고 가져다가 사용하는 방법을 factory pattern 이라고 한다.

팩토리 패턴에서는 상위 추상 클래스에서 객체의 결합 방법을 결정하고 하위 클래스에서는 객체 설정과 생성을 수행한다.

팩토리 패턴 이용한 객체 주입

src/app/factory/animal.ts

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

export class Config {
    public bark = '어흥';
    public name = '사자';
}

@Injectable()
export class Animal {
    constructor(public config: Config) {}
    getBark() {
        return this.config.bark;
    }
    getAnimalName() {
        return this.config.name;
    }
}
```

src/app/factory/animal-factory.ts

```
import { Animal, Config } from './animal';

//팩토리패턴
abstract class Factory {
  // 객체 생성방법 결정
  create() {
     return this.createAnimal(new Config());
  }

  abstract createAnimal(Config): Animal;
}

export class AnimalFactory extends Factory{
  // 객체 설정방법 결정
  createAnimal(config:Config){
    config.bark="야옹야옹";
    config.name="고양이";
    return new Animal(config);
  }
}
```

```
export function mainFactory() {
  let myAnimal:Animal= (new AnimalFactory()).create();
  return myAnimal;
}
```

객체 생성 방법은 Factory 추상 클래스에 정의된 create 메소드가 결정한다.

객체 설정 방법은 AnimalFactory 클래스에 정의된 createAnimal 메소드에서 결정한다.

src/app/factory/factory.component.ts

```
import { Component } from '@angular/core';
import { mainFactory } from './animal-factory';

@Component({
  selector: 'app-factory',
  template: `
  {{animal1.getAnimalName()}} : {{animal1.getBark()}}<br>
  {{animal2.getAnimalName()}} : {{animal2.getBark()}}<br>
  {{animal1 === animal2}}
  `
})
export class FactoryComponent{
  animal1= mainFactory();
  animal2= mainFactory();
}
```

주입기를 이용한 객체 생성

src/app/reflective-injector/dog.ts

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

export class Config {
    public walking = '쫄랑쫄랑';

}

interface Animal {
    getName();
}

@Injectable()
export class Dog implements Animal {
    constructor(public config: Config) { }
    walking() {
        return this.config.walking;
    }
    getName() {
        return "강아지";
    }
}

@Injectable()
export class Pet extends Dog {
    constructor(public config: Config) { super(config); }

    run() {
        return this.config.walking;
    }

    getName() {
        return "애완견";
    }
}
```

src/app/reflective-injector/animal-injector.ts

```
import { ReflectiveInjector } from '@angular/core';
import { Dog, Pet, Config } from './dog';


export function configInjector() {
  let injector: ReflectiveInjector = ReflectiveInjector.resolveAndCreate([Config]);
  return injector.get(Config);
}

export function dogInjector() {
  let injector: ReflectiveInjector = ReflectiveInjector.resolveAndCreate([Dog, Pet,
Config]);
  return injector.get(Dog);
}
```

ReflectiveInjector.***resolveAndCreate***([Config]);

토큰으로 입력된 클래스를 싱글톤 객체로 만들어주는 메소드이다.

Provider 배열을 입력받고 Injector 를 리턴한다.

ReflectiveInjector.*resolveAndCreate*([Dog, Pet, Config]);

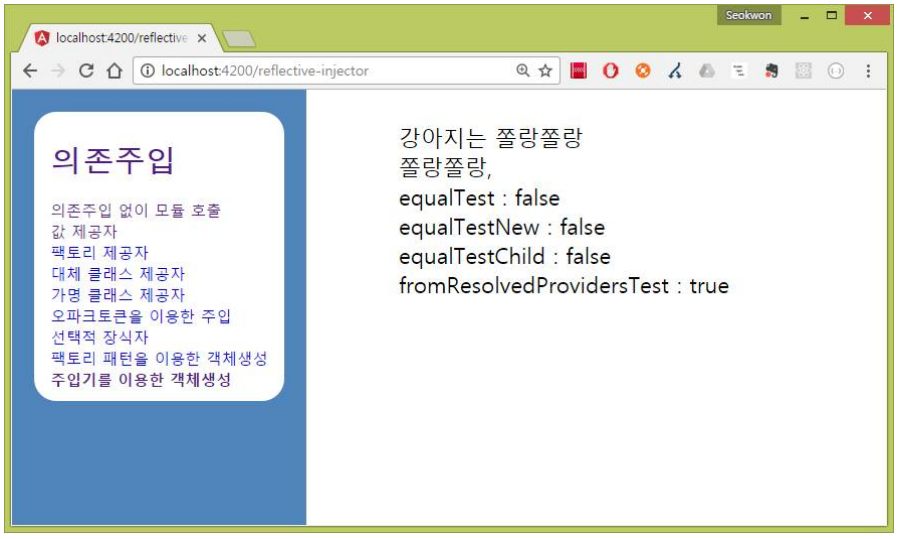
의존하는 클래스를 함께 선언해야 한다.

src/app/reflective-injector/animal-injector-test.ts

```
import { ReflectiveInjector } from '@angular/core';
import { Dog, Pet, Config } from './dog';

export function equalTest() {
    let injector: ReflectiveInjector = ReflectiveInjector.resolveAndCreate([Dog, Config]);
    let injector2: ReflectiveInjector = ReflectiveInjector.resolveAndCreate([Dog, Config]);
    let dog1 = injector.get(Dog);
    let dog2 = injector2.get(Dog);
    // 인젝터로부터 받은 두 객체가 동일한지 테스트한다.
    return dog1 === dog2;
}

export function equalTestNew() {
    // 수동으로 만든 두 객체가 동일한지 테스트한다.
    return (new Dog(new Config()) === new Dog(new Config()));
}

export function equalTestChild() {
    let injector: ReflectiveInjector = ReflectiveInjector.resolveAndCreate([Dog, Pet,
Config]);
    let injector2: ReflectiveInjector = injector.resolveAndCreateChild([Pet]);
    let pet1 = injector.get(Pet);
    let pet2 = injector2.get(Pet);
    // 부모 인젝터로 만든 객체와 자식 인젝터로 만든 객체가 같은지 테스트한다.
    return pet1 === pet2;
}

export function fromResolvedProvidersTest() {
    let providers = ReflectiveInjector.resolve([Dog, Pet, Config]);
    // 프로바이더 정보를 활용해 새로운 인젝터를 만든다.
    let injector: ReflectiveInjector = ReflectiveInjector.fromResolvedProviders(providers);
    // 인젝터에 선언한 클래스 개수가 3 개인지 테스트한다.
    return (injector.get(Pet) instanceof Pet) && providers.length == 3;
}
```

src/app/reflective-injector/reflective-injector.component.ts

```
import { Component } from '@angular/core';
import { dogInjector, configInjector, } from './animal-injector';
import { equalTest, equalTestNew, equalTestChild, fromResolvedProvidersTest } from
'./animal-injector-test';

@Component({
  selector: 'app-reflective-injector',
  template: `
  {{dog.getName()}}는 {{dog.walking()}} <br>
  {{config.walking}}, <br>
  equalTest : {{equalTest}}<br>
  equalTestNew : {{equalTestNew}}<br>
  equalTestChild : {{equalTestChild}}<br>
  fromResolvedProvidersTest : {{fromResolvedProvidersTest}}
  `
})
export class ReflectiveInjectorComponent {
  dog = dogInjector();
  config = configInjector();

  equalTest = equalTest();
  equalTestNew = equalTestNew();
  equalTestChild = equalTestChild();
  fromResolvedProvidersTest = fromResolvedProvidersTest();
}
```



인젝터가 만드는 객체는 싱글톤이 아닌 새로운 객체다.

Binding

바인딩은 클래스와 템플릿을 연결하는 기능이다.

클래스의 apple 변수에 들어 있는 값을 입력 엘리먼트로 바인딩하려면 다음과 같이 설정한다.

단방향 바인딩

```
template: `<input type="text" [value]="apple">`
```

양방향 바인딩

```
template: `<input type="text" [(ngModel)]="apple">`
```

단방향 바인딩

src/app/interpolation/interpolation.component.ts

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

@Component({
  selector: 'app-interpolation',
  template: `
  {{1 + 1}} <br>
  {{hello()+" world!"}} <br>
  {{hello()==="hello"?"YES":"NO"}} <br>
  {{basket.items[0]}} <br>
  {{goodbye}} <br>

  <input type="text" value="{{myclass}}">
  <button class="{{myclass}}">{{myclass}}</button>`,
  styles: [`
  .my-italic { font-style: italic; }`]
})
export class InterpolationComponent {

  hello() { return "hello"; }

  basket = {
    items: ['apple', 'grapee', 'orange']
  };

  goodbye: string;
  constructor() {
    let x: string = '굿';
    let y: string = '바이';
    this.goodbye = `${x + y}`;
  }
```

```
  myclass = "my-italic";

}
```

src/app/oneway-expression/oneway-expression.component.ts

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

@Component({

  selector: 'app-oneway-expression',
  template: `
  {{greeting}}<br>
  <input type="text" [value]="greeting">
  <input type="text" bind-value="greeting">
  <input type="text" [attr.value]="greeting">`
})
export class OnewayExpressionComponent {

  greeting: string="hello";

}
```

src/app/oneway-statement/contact.component.ts

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

@Component({
    selector: 'contactinput',
    template: `
    <div><input #contactname type="text" placeholder="이름" /></div>
    <div><input #contacttele type="text" placeholder="전화번호" /></div>
    <button (click)="handleClick(contactname.value, contacttele.value)">저장</button>`
})

export class ContactComponent {

    @Output() save: EventEmitter<any> = new EventEmitter();

    public handleClick(name: string, tele: string): void {
        this.save.next({name: name, telephone: tele});
    }
}
```

저장버튼을 클릭하면 handleClick 메소드가 동작하고 내부에 EventEmitter 로 정의한 save 변수를 통해 연락처 컴포넌트를 호출한 contactinput 의 커스텀 속성인 save 를 통해 this.save.next 에 입력한 정보가 입력한 정보가 $event 변수로 전달된다.

src/app/oneway-statement/my-click.directive.ts

```
import { Directive, ElementRef, EventEmitter, Output } from '@angular/core';

@Directive({selector: '[myClick]'})
export class MyClickDirective {

  toggle = false;

  @Output('myClick') clicks = new EventEmitter<string>();

  constructor(el: ElementRef) {
    el.nativeElement.addEventListener('click', (event: Event) => {
        this.toggle = !this.toggle;
        this.clicks.emit(this.toggle ? '선택했습니다.' : '해제했습니다.');
    });
  }
}
```

myClick 지시자를 사용하는 엘리먼트에서 클릭 이벤트가 발생하면 클릭 이벤트에 설정된 내용이 처리되고 this.clicks.emit 을 통해 지시자에서 컴포넌트로 값이 전달된다.

src/app/oneway-statement/oneway-statement.component.ts

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

@Component({
  selector: 'app-oneway-statement',
  template: `
  <h3>{{msg}}</h3>
  <button (click)="greetings('안녕하세요');">안녕하세요</button>
  <button on-click="greetings('환영합니다');">환영합니다</button><br><br>

  <h3>{{msg2}}</h3>
  <contactinput (save)="saveContact($event)"></contactinput><br><br>

  <h3>{{clicked}}</h3>
  <button (myClick)="clicked=$event">클릭</button>`
})
export class OnewayStatementComponent {

  public msg: string = "버튼을 선택해주세요";
  public msg2: string = "이름과 이메일을 입력해주세요";

  clicked: any = "버튼을 선택해주세요";

  greetings(msg: string) {
    this.msg = msg;
  }

  private saveContact(contact) {
    this.msg2 = JSON.stringify(contact);
  }

}
```

(click)="greetings('안녕하세요');"

이벤트 바인딩을 통해 처리한다. 이벤트 바인딩 시 사용할 수 있는 이벤트는 웹 표준 이벤트를 따른다.

http://developer.mozilla.org/en-US/docs/Web/Events

@Output() save: EventEmitter<any> = new EventEmitter();

@Output 장식자를 통해 이벤트 명령식인 saveContact($event)를 실행한다.

@Output('myClick') clicks = new EventEmitter<string>();

@Output 장식자를 통해 이벤트 명령식인 clicked=$event 를 실행한다.

양방향 바인딩

src/app/twoway-ngmodel/twoway-ngmodel.component.ts

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

@Component({
  selector: 'app-twoway-ngmodel',
  template: `
  <select [(ngModel)]="city">
        <option *ngFor="let obj of cities" [value]="obj.eng">{{obj.han}}</option>
  </select>
  <select (change)="city=$event.target.value">
         <option       *ngFor="let       obj       of       cities"       [value]="obj.eng"
[selected]="city==obj.eng?true:null">
         {{obj.han}}
         </option>
  </select><br>

  <input [(ngModel)]="city">
  <input [value]="city" (input)="city=$event.target.value" ><br>

  <span *ngFor="let obj of cities">
         <input              type="radio"              [checked]="(obj.eng==city?true:null)"
(click)="city=$event.target.value" [value]="obj.eng" name="city">{{obj.han}}
  </span>`,
  styles: [`section{margin-bottom:20px;}`]
})
export class TwowayNgmodelComponent {

  city: string = "seoul";
  cities: Object[] = [
    { han: "서울", eng: "seoul" },
    { han: "대전", eng: "daejeon" },
    { han: "대구", eng: "daegu" },
    { han: "부산", eng: "pusan" }
  ];

}
```

속성 지시자

src/app/ngclass/ngclass.component.ts

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

@Component({
  selector: 'app-ngclass',
  template: `
  <button class="button" [ngClass]="{active: isActive}"
  (click)="isActive=!isActive;">{{isActive?'활성화':'비 활성화'}}</button><br>

  <button [ngClass]="myclass">버튼 1</button>
  <button [ngClass]="'active'">버튼 2</button>
  <button bind-ngClass="myclass">버튼 3</button><br>

  <button [attr.class]="myclass">버튼 4</button>
  <button [class.active]="true">버튼 5</button>
  `,
  styles: [`
  button {
      width: 100px; padding: 10px;
      margin-bottom: 10px;
      text-align:center;
      border: 1px dotted #666;
  }
  button.active {
      background-color: #CFD7EB; border: 1px solid #666;
  }`]
})
export class NgclassComponent {

  public isActive: boolean = false;
  myclass: string="active";

}
```

[ngClass]="{active: isActive}"

CSS 의 클래스 이름을 더하거나 해제하는 속성 지시자이다.

active 는 클래스명이다.

attr.class

attr 속성을 통해 클래스를 추가한다.

class.active

class 속성을 통해 active 클래스를 추가한다.

src/app/ngstyle/ngstyle.component.ts

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

@Component({
  selector: 'app-ngstyle',
  template: `
  <div>
    <input type="text" [value]="text" [(ngModel)]="text">
    <select [value]="style" [(ngModel)]="style">
      <option value="normal">반듯한 글자</option>
      <option value="italic">기울어진 글자</option>
    </select>
    <label>볼드체<input type="checkbox" (change)="changeWeight($event)"></label>
  </div>
  <h2 [ngStyle]="{'font-style': style, 'font-weight': weight}">{{text}}</h2>
  <h1 [style.font-style]="style" [style.font-weight]="weight">{{text}}</h1>`})
export class NgstyleComponent {

  text = '안녕하세요';
  weight = 'normal';
  style = 'normal';

  changeWeight($event: any) {
    this.weight = $event.target.checked ? 'bold' : 'normal';
  }

}
```

ngStyle 은 CSS 클래스가 아닌 CSS 스타일 속성과 값을 이용해 설정한다.

(change)="changeWeight($event)"

이벤트 방식으로 함수를 호출해 함수 내부에서 스타일을 결정한다.

이벤트가 발생한 시점에 대한 상태 정보를 전달하기 위해 함수로 $event 매개변수를 전달한다.

[(ngModel)]="style"

양방향 바인딩.

구조 지시자

src/app/ngif/ngif.component.ts

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

@Component({
  selector: 'app-ngif',
  template: `
  <select [(ngModel)]="gender">
      <option value=1>남자</option>
      <option value=2>여자</option>
  </select>

  <h2 *ngIf="gender == 1">남자가 선택됨</h2>
  <h2 *ngIf="gender == 2">여자가 선택됨</h2>

  <h3>
    <span template="ngIf gender == 1">남자가 선택됨</span>
    <span template="ngIf gender == 2">여자가 선택됨</span>
  </h3>

  <h4>
    <template [ngIf]="gender == 1">남자가 선택됨</template>
    <template [ngIf]="gender == 2">여자가 선택됨</template>
  </h4>`
})
export class NgifComponent{

  gender = 1;

}
```

src/app/ngswitch/ngswitch.component.ts

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

@Component({
  selector: 'app-ngswitch',
  template: `
  <div>
  <button (click)="grade='admin'"
  [style.font-weight]="grade=='admin'?'bold':'normal'">운영자</button>

  <button (click)="grade='member'"
  [style.font-weight]="grade=='member'?'bold':'normal'">회원</button>

  <button (click)="grade='guest'"
  [style.font-weight]="grade=='guest'?'bold':'normal'">손님</button>
  </div><br>

  <p *ngIf="grade=='null'">회원등급을 선택해주세요</p>
  <p *ngIf="grade!='null'">{{grade}} 등급이 선택됨</p>

  <div [ngSwitch]="grade">
    <h3 *ngSwitchCase="'null'">회원등급을 선택해주세요</h3>
    <h3 *ngSwitchCase="'admin'">운영자가 선택됨</h3>
    <h3 *ngSwitchCase="'member'">회원이 선택됨</h3>
    <h3 *ngSwitchDefault>손님 선택</h3>
  </div>

  <p [ngSwitch]="grade">
    <template ngSwitchCase="null">회원등급을 선택해주세요</template>
    <template [ngSwitchCase]="'admin'">운영자가 선택됨</template>
    <template [ngSwitchCase]="'member'">회원이 선택됨</template>
    <template ngSwitchDefault>손님 선택</template>
  </p>`
})
export class NgswitchComponent {

  grade: any;
  constructor() {
    this.grade = 'null';
  }

}
```

src/app/ng-for/ng-for.component.ts

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

@Component({
  selector: 'app-ngstyle',
  template: `
  <h3>한국에서 가장 높은 산 TOP3</h3>
  <div *ngFor="let item of items; let i = index">
    {{ i+1 }} | {{item.title}} | {{item.height | number}}m
  </div><br>`
})
export class NgForComponent {

  items: Object[] = [];

  constructor() {
     this.items.push({'title': '한라산','height':'1950'});
     this.items.push({'title': '지리산','height':'1915'});
     this.items.push({'title': '설악산','height':'1707.9'});
  }

}
```

템플릿 태그

템플릿 참조변수

템프릿 내에서 정의하고 템플릿 내부에서만 사용할 수 있다.

간단한 상태 전달이 필요한 경우에 이용한다.

src/app/template-reference-variables/template-reference-variables.component.ts

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

@Component({
  selector: 'app-template-reference-variables',
  template: `
  <input #num1 type="number" value="{{init.num1}}" (input)="0">  +
  <input #num2 type="number" value="{{init.num2}}" (input)="0">
  = {{num1.valueAsNumber + num2.valueAsNumber}}<br>

  <input ref-number1 type="number" value="{{init.num1}}" (input)="0">  +
  <input ref-number2 type="number" value="{{init.num2}}" (input)="0">
  = {{number1.valueAsNumber + number2.valueAsNumber}}`
})
export class TemplateReferenceVariablesComponent {

  private get init(): any {
    return { num1: 10, num2: 20 }
  }

}
```

valueAsNumber

HTML5 의 입력 엘리먼트의 속성으로 숫자일 경우 출력하고 그렇지 않으면 NaN 값을 출력한다.

Sanitization

새니티제이션은 잠재적인 위험요소를 제거하는 기능으로써 악의적인 의도가 있는 스크립트 유입을 막아준다. 별도 설정을 하지 않아도 템플릿에서 속성 바인딩 시 새니티제이션이 적용된다. 때로는 새니티제이션이 적용되지 않도록 설정을 할 필요가 있다.

src/app/trusturl/trusturl.component.ts

```
import { Component } from '@angular/core';
import { SafeUrl, DomSanitizer } from '@angular/platform-browser';

@Component({
  selector: 'app-trusturl',
  template: `
  신뢰할 수 없는 URL : <a [href]="riskyURL">연결</a><br>
  신뢰할 수 있는 URL : <a [href]="trustURL">연결</a><br>
  신뢰할 수 없는 URL : <a [href]="'https://angular.io'">연결</a><br>
  신뢰할 수 있는 URL : <a [href]="transToTrustURL('https://angular.io')">연결</a>`
})
export class TrusturlComponent {

  riskyURL: String;
  trustURL: SafeUrl;

  constructor(private _sanitizer: DomSanitizer) {
    this.riskyURL = "javascript:alert('hello')";
    this.trustURL = this._sanitizer.bypassSecurityTrustUrl("javascript:alert('hello');");
  }

  transToTrustURL(url: string){
    return this._sanitizer.bypassSecurityTrustUrl(url);
  }
}
```

src/app/trusthtml/trusthtml.component.ts

```
import { Component } from '@angular/core';
import { DomSanitizer, SafeHtml } from '@angular/platform-browser';

@Component({
  selector: 'app-trusthtml',
  template: `
  <h3>신뢰할 수 없는 HTML</h3>
  <div [innerHtml]="notTrustHtml"></div>

  <h3>신뢰할 수 있는 HTML</h3>
  <div [innerHtml]="trustHtml"></div><br>`
})
export class TrusthtmlComponent {

  trustHtml: SafeHtml;
  notTrustHtml:   string   =   `<script>function   hello(){   alert("헬로우   메서드   알림
창!"); }</script>
  <style>button{font-size:20px;padding:10px;font-style:italic;}</style>
  <button onclick="hello()">메서드 호출 by onclick</button>
  <button (click)="hello()">메서드 호출 by (click)</button><br><br>

  <button onclick="javascript:alert('헬로우 알림창!')">Hello 알림창 띄우기</button>`;

  constructor(private _sanitizer: DomSanitizer) {
    this.trustHtml = this._sanitizer.bypassSecurityTrustHtml(this.notTrustHtml);
  }

  hello(){ alert('hello'); }

}
```

src/app/trustresourceurl/trustresourceurl.component.ts

```
import { Component } from '@angular/core';
import { DomSanitizer, SafeResourceUrl } from '@angular/platform-browser';

@Component({

  selector: 'app-trustresourceurl',
  template: `<iframe width="250px" height="150px" [src]="trustResourceURL"></iframe>`
})
export class TrustresourceurlComponent {

  trustResourceURL: SafeResourceUrl;

  constructor(private _sanitizer: DomSanitizer) {
    let url="https://angular.io";
    this.trustResourceURL = this._sanitizer.bypassSecurityTrustResourceUrl(url);
  }

}
```

src/app/truststyle/truststyle.component.ts

```
import { Component, Pipe, PipeTransform } from '@angular/core';
import { DomSanitizer, SafeStyle } from '@angular/platform-browser';


@Pipe({name: 'safe'})
export class Safe implements PipeTransform  {
  constructor(private sanitizer:DomSanitizer){
    this.sanitizer = sanitizer;
  }

  transform(style) {
    return this.sanitizer.bypassSecurityTrustStyle(style);
  }
}

@Component({
  selector: 'app-truststyle',
  template: `
  <img [style.background-image]="styleURL">
  <img [style.background-image]="styleURL | safe">
  <img [style.background-image]="trustStyle">
  <img [style.background-image]="'url(' + imageURL + ')\'">
  `,
  styles: [`img{height:128px;width:128px;}`]
})
export class TruststyleComponent {

  imageURL: string = '/images/car.png';
  styleURL:                                    string                                    =
"url(\'https://angular.io/resources/images/logos/angular2/angular.svg\')";
  trustStyle: SafeStyle;

  constructor(private _sanitizer: DomSanitizer) {
    this.trustStyle = this._sanitizer.bypassSecurityTrustStyle(this.styleURL);
  }

}
```

<img [style.background-image]="'url(' + imageURL + ')₩'">

원격이미지를 사용하기 때문에 가능하다.

Router

라우터는 사용자가 요청한 URL 에 따라 해당하는 컴포넌트를 표시하는 역할을 수행한다.
컴포넌트의 출력 영역은 <router-outlet></router-outlet>으로 정의한다.

라우터 아울렛은 루트 컴포넌트나 특징 컴포넌트에 설정한다.

루트 컴포넌트	특징 컴포넌트	자식 컴포넌트
/root	/root/children	/root/children/child1

HTML 은 앵커태그의 href 속성으로 화면전환을 한다. 이 속성은 화면 전체를 로딩하기 때문에 사용하지 않고 대신 routerLink 디렉티브를 사용하여 컴포넌트 간 라우팅이 일어나게 한다.
<a routerLink="/root">루트 컴포넌트

주소에 / 를 붙이면 절대주소로 취급한다. 도메인 다음부터 설정된 주소를 사용한다.
주소에 ../ 를 붙이면 상대주소로 취급한다. 현재 주소에서 한단계 위로 이동한 다음 설정된 주소를 사용한다.

클래스에서 코드적으로 라우팅 요청을 할 수 있다.

this._router.navigateByUrl("/root/children");
this._router.navigate(["root", "children"]);
let url = this._router.createUrlTree(["root","children"]); this._router.navigateByUrl(url);
let url = this._router.createUrlTree([{segmentPath: "children"}]); this._router.navigateByUrl(url);

src/app/app.routing.ts

```
import { ModuleWithProviders } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { LocationStrategy, HashLocationStrategy } from '@angular/common';

import { loginRoutes, authProviders } from './login.routing';
import { CanDeactivateGuard } from './can-deactivate-guard.service';
import { AuthGuard } from './auth-guard.service';

import { IntroComponent } from './intro.component';
import { NotFoundComponent } from './not-found.component';
import { HelloComponent } from './hello/hello.component';
import { FirstPageComponent } from './pages/first-page.component';
import { SecondPageComponent } from './pages/second-page.component';
import { ThirdPageComponent } from './pages/third-page.component';
import { RouterLinkTestComponent } from './router-link-test/router-link-test.component';
import { HrefTestComponent } from './router-link-test/href-test.component';
import { LoginComponent } from './login.component';
import { AdminComponent } from './admin/admin.component';

// Config
let hashLocationStrategy: boolean = false;

const helloRoutes: Routes = [
  { path: '', component: IntroComponent },
  { path: 'hello', component: HelloComponent },
  { path: 'router-link-test', component: RouterLinkTestComponent },
  { path: 'href-test', component: HrefTestComponent },
  { path: 'pages/first-page', component: FirstPageComponent },
  { path: 'pages/second-page', component: SecondPageComponent },
  { path: 'pages/third-page', component: ThirdPageComponent },
  { path: 'login', component: LoginComponent },
  { path: 'admin', component: AdminComponent }
];

const lazyRoutes: Routes = [
  {
    path: 'lazy',
    loadChildren: 'app/player/player.module#PlayerModule',
    canLoad: [AuthGuard]
  }
];

const appRoutes: Routes = [
  ...loginRoutes,
  ...lazyRoutes,
  ...helloRoutes,
  { path: '**', component: NotFoundComponent }
];

export const appRoutingProviders: any[] = [
  authProviders,
  CanDeactivateGuard
];

if(hashLocationStrategy){
  appRoutingProviders.push({provide: LocationStrategy, useClass: HashLocationStrategy});
}

export const AppRoutingModule: ModuleWithProviders = RouterModule.forRoot(appRoutes);
```

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

Routes 는 라우터 설정의 구조를 정의한 인터페이스 모듈이다.

RouterModule 은 디렉티브나 컴포넌트를 포함해 모듈을 만들 때 사용하는 모듈이다.

```
const helloRoutes: Routes = [
  { path: '', component: IntroComponent },
  { path: 'hello', component: HelloComponent },
  { path: 'router-link-test', component: RouterLinkTestComponent },
  { path: 'href-test', component: HrefTestComponent },
  { path: 'pages/first-page', component: FirstPageComponent },
  { path: 'pages/second-page', component: SecondPageComponent },
  { path: 'pages/third-page', component: ThirdPageComponent },
  { path: 'login', component: LoginComponent },
  { path: 'admin', component: AdminComponent }
];
```

path 가 비어 있는 경우 기본 경로 / 로 취급하여 IntroComponent 를 연결한다.

```
import { loginRoutes, authProviders } from './login.routing';
import { NotFoundComponent } from './not-found.component';

const appRoutes: Routes = [
  ...loginRoutes,
  ...lazyRoutes,
  ...helloRoutes,
  { path: '**', component: NotFoundComponent }
];
```

스프레드 연산자를 사용하여 라우터 설정변수를 Routes 배열에 추가한다.

** 로 404 Page Not Found 시 연동할 컴포넌트를 지정한다.

```
import { ModuleWithProviders } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
export const AppRoutingModule: ModuleWithProviders = RouterModule.forRoot(appRoutes);
```

RouterModule.forRoot 메소드로 앱 단위의 라우터 모듈을 만든다.

AppRoutingModule 을 루트 모듈에서 사용한다.

```
import { loginRoutes, authProviders } from './login.routing';
import { CanDeactivateGuard } from './can-deactivate-guard.service';

export const appRoutingProviders: any[] = [
  authProviders,
  CanDeactivateGuard
];
```

라우터 인증관 관련된 서비스를 설정한다.

src/app/app.module.ts

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

/* application router settings */
import { AppRoutingModule, appRoutingProviders } from './app.routing';

/* feature module */
import { MemberModule } from './member/member.module';
import { PlayerModule } from './player/player.module';
import { ChildrenModule } from './children/children.module';

/* global components */
import { AppComponent } from './app.component';
import { IntroComponent } from './intro.component';
import { LoginComponent } from './login.component';
import { NotFoundComponent } from './not-found.component';

import { HelloComponent } from './hello/hello.component';
import { RouterLinkTestComponent } from './router-link-test/router-link-test.component';
import { HrefTestComponent } from './router-link-test/href-test.component';

import { FirstPageComponent } from './pages/first-page.component';
import { SecondPageComponent } from './pages/second-page.component';
import { ThirdPageComponent } from './pages/third-page.component';

import { AdminComponent } from './admin/admin.component';

@NgModule({
  imports: [
    BrowserModule, CommonModule, FormsModule,
    AppRoutingModule,
    MemberModule, PlayerModule, ChildrenModule
  ],
  providers: [appRoutingProviders],
  declarations: [
    AppComponent, IntroComponent, HelloComponent,
    RouterLinkTestComponent,
    HrefTestComponent,
    FirstPageComponent, SecondPageComponent, ThirdPageComponent,
    LoginComponent,
    AdminComponent,
    NotFoundComponent
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

기동 컴포넌트인 AppComponent 가 `<router-outlet></router-outlet>`을 포함한다.

해시 기반 주소로 변경

앵귤러 2 는 기본적으로 앵귤러 1 에서 사용하던 # 해시태그를 이용하지 않는다.

그러나 때로는 해시 주소 기반으로 사용하고 싶을 수도 있다.

app.routing.ts

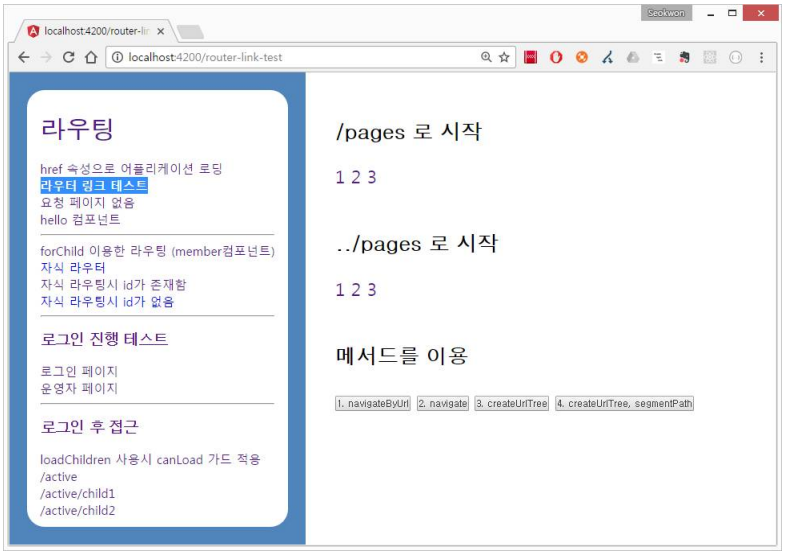
```
import { LocationStrategy, HashLocationStrategy } from '@angular/common';

import { loginRoutes, authProviders } from './login.routing';
import { CanDeactivateGuard } from './can-deactivate-guard.service';

let hashLocationStrategy: boolean = true;

export const appRoutingProviders: any[] = [
  authProviders,
  CanDeactivateGuard
];

if(hashLocationStrategy){
  appRoutingProviders.push({provide: LocationStrategy, useClass: HashLocationStrategy});
}
```

app.module.ts

```
import { AppRoutingModule, appRoutingProviders } from './app.routing';

@NgModule({
  imports: [
    BrowserModule, CommonModule, FormsModule,
    AppRoutingModule,
    MemberModule, PlayerModule, ChildrenModule
  ],
  providers: [appRoutingProviders],
  declarations: [
    AppComponent, IntroComponent, HelloComponent,
    RouterLinkTestComponent,
    HrefTestComponent,
    FirstPageComponent, SecondPageComponent, ThirdPageComponent,
    LoginComponent,
    AdminComponent,
    NotFoundComponent
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

http://localhost:4200/hello 주소 대신 http://localhost:4200/#/hello 주소를 사용한다.

연결순서 : http://localhost:4200/router-link-test



app.component.ts

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

@Component({
  selector: 'app-root',
  template: `
  <div class="left-menu">
      <div class="menu">
        <a routerLink="/">
          <h1>라우팅</h1>
        </a>
        <ol class="tree-list">
          <li><a routerLink="/href-test">href 속성으로 어플리케이션 로딩</a></li>
          <li><a routerLink="/router-link-test">라우터 링크 테스트</a></li>
          <li><a routerLink="/!@#">요청 페이지 없음</a></li>
          <li><a routerLink="/hello">hello 컴포넌트</a></li>
          <hr>
        </ol>
    </div>
  </div>
  <div class="play-box">
    <router-outlet></router-outlet>
  </div>`
})
export class AppComponent {
}
```

routerLink 는 URL 을 라우터에게 전달한다.

라우터는 전달받은 정보에 해당하는 라우터 연결이 있으면 해당 컴포넌트를 호출한다.

app.routing.ts

```
const helloRoutes: Routes = [
  { path: '', component: IntroComponent },
  { path: 'hello', component: HelloComponent },
  { path: 'router-link-test', component: RouterLinkTestComponent },
  { path: 'href-test', component: HrefTestComponent },
  { path: 'pages/first-page', component: FirstPageComponent },
  { path: 'pages/second-page', component: SecondPageComponent },
  { path: 'pages/third-page', component: ThirdPageComponent },
  { path: 'login', component: LoginComponent },
  { path: 'admin', component: AdminComponent }
];
```

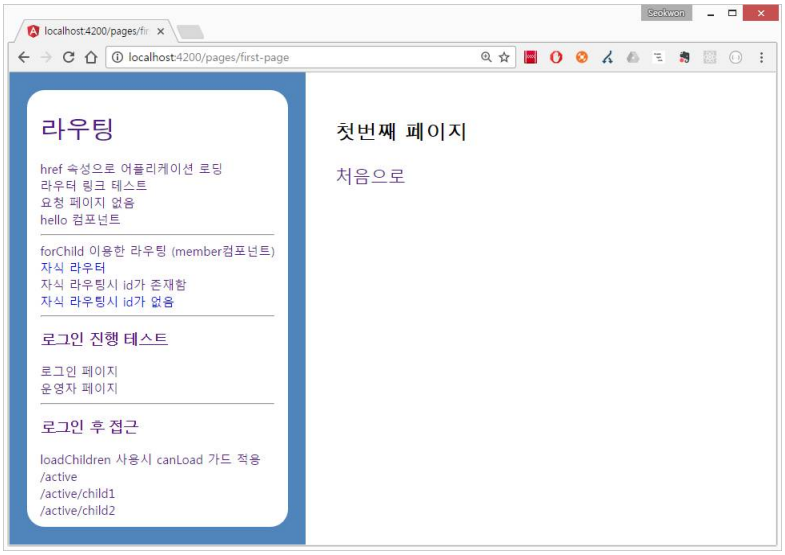
설정에 따라 RouterLinkTestComponent 컴포넌트가 기동한다.

router-link-test.component.ts

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

@Component({
  selector: 'router-link-test',
  template: `
  <h3>/pages 로 시작</h3>
  <a routerLink="/pages/first-page">1</a>
  <a routerLink="/pages/second-page">2</a>
  <a routerLink="/pages/third-page">3</a> <br><br>

  <h3>../pages 로 시작</h3>
  <a routerLink="../pages/first-page">1</a>
  <a routerLink="../pages/second-page">2</a>
  <a routerLink="../pages/third-page">3</a> <br><br>

  <h3>메서드를 이용</h3>
  <button (click)="one()">1. navigateByUrl</button>
  <button (click)="two()">2. navigate</button>
  <button (click)="three()">3. createUrlTree</button>
  <button (click)="four()">4. createUrlTree, segmentPath</button>
  `
})
export class RouterLinkTestComponent {
  constructor(public _router: Router) { }
  one() {
    this._router.navigateByUrl("/pages/first-page");
  }

  two() {
    this._router.navigate(['pages', 'second-page']);
  }

  three() {
    let url = this._router.createUrlTree(['pages', 'third-page']);
    this._router.navigateByUrl(url);
  }

  // FIXME: bug is here
  four() {
```

```
    let url = this._router.createUrlTree([{segmentPath: 'third-page'}]);
    this._router.navigateByUrl(url);
  }

}
```

연결순서: http://localhost:4200/pages/first-page



router-link-test.component.ts

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

@Component({
  selector: 'router-link-test',
  template: `
  <h3>/pages 로 시작</h3>
  <a routerLink="/pages/first-page">1</a>
  <a routerLink="/pages/second-page">2</a>
  <a routerLink="/pages/third-page">3</a> <br><br>

  <h3>../pages 로 시작</h3>
  <a routerLink="../pages/first-page">1</a>
  <a routerLink="../pages/second-page">2</a>
  <a routerLink="../pages/third-page">3</a> <br><br>

  <h3>메서드를 이용</h3>
  <button (click)="one()">1. navigateByUrl</button>
  <button (click)="two()">2. navigate</button>
  <button (click)="three()">3. createUrlTree</button>
  <button (click)="four()">4. createUrlTree, segmentPath</button>
  `
})
export class RouterLinkTestComponent {
  constructor(public _router: Router) { }
  one() {
    this._router.navigateByUrl("/pages/first-page");
  }

  two() {
    this._router.navigate(['pages', 'second-page']);
  }
```

```
  three() {
    let url = this._router.createUrlTree(['pages', 'third-page']);
    this._router.navigateByUrl(url);
  }

  // FIXME: bug is here
  four() {
    let url = this._router.createUrlTree([{segmentPath: 'third-page'}]);
    this._router.navigateByUrl(url);
  }

}
```

app.routing.ts

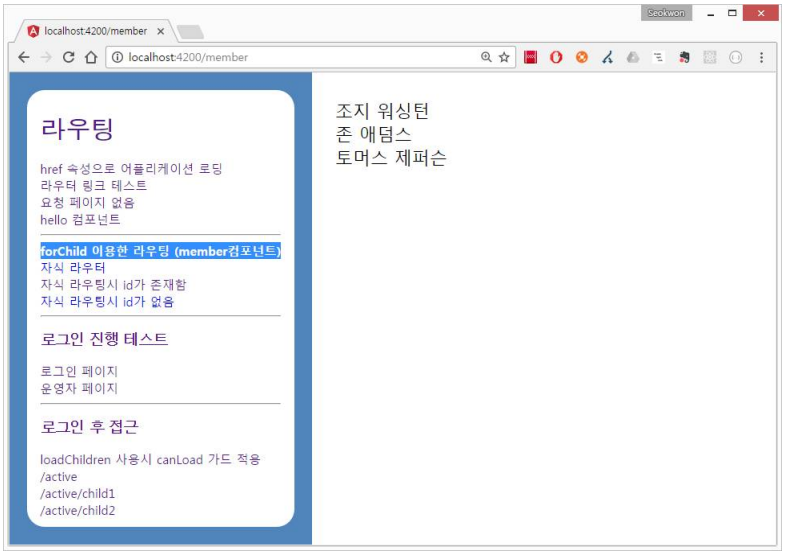
```
const helloRoutes: Routes = [
  { path: '', component: IntroComponent },
  { path: 'hello', component: HelloComponent },
  { path: 'router-link-test', component: RouterLinkTestComponent },
  { path: 'href-test', component: HrefTestComponent },
  { path: 'pages/first-page', component: FirstPageComponent },
  { path: 'pages/second-page', component: SecondPageComponent },
  { path: 'pages/third-page', component: ThirdPageComponent },
  { path: 'login', component: LoginComponent },
  { path: 'admin', component: AdminComponent }
];
```

설정에 따라 FirstPageComponent 컴포넌트가 기동한다.

first-page.component.ts

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

@Component({
  selector: 'first-page',
  template: `<h3>첫번째 페이지</h3>
  <a routerLink="/router-link-test" routerLinkActive="active">처음으로</a>`
})
export class FirstPageComponent { }
```

연결순서 : http://localhost:4200/member



app.module.ts

```
/* feature module */
import {MemberModule} from './member/member.module';
import {PlayerModule} from './player/player.module';
import {ChildrenModule} from './children/children.module';

@NgModule({
  imports: [
    BrowserModule, CommonModule, FormsModule,
    AppRoutingModule,
    MemberModule, PlayerModule, ChildrenModule
  ],
  …
  bootstrap: [AppComponent]
})
export class AppModule {}
```

member.module.ts

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

import {MemberComponent}   from './member.component';
import {MemberRoutingModule}   from './member-routing.module';

@NgModule({
  imports: [CommonModule, FormsModule, MemberRoutingModule],
  declarations: [MemberComponent],
  providers: []
})
export class MemberModule {}
```

member-routing.module.ts

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

import {MemberComponent}    from './member.component';

@NgModule({
  imports: [RouterModule.forChild([
    {path: 'member', component: MemberComponent}
  ])],
  exports: [RouterModule]
})
export class MemberRoutingModule {}
```

member-routing.module.ts → member.module.ts → app.module.ts 설정으로

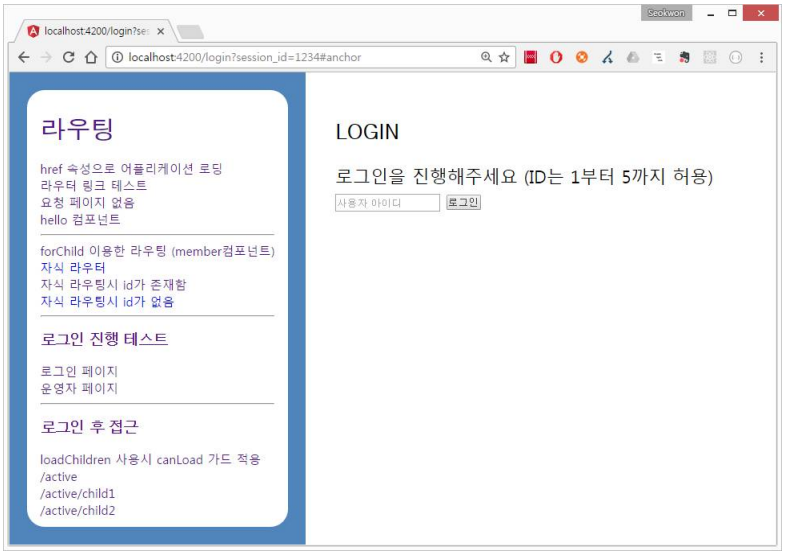
http://localhost:4200/member 주소로 접근 시 MemberComponent 가 기동한다.

member.component.ts

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

@Component({
  selector: 'member',
  template: `
  <div *ngFor='let m of member'>{{m}}</div>`
})
export class MemberComponent {
  member: string[] = ['조지 워싱턴', '존 애덤스', '토머스 제퍼슨'];
}
```

연결순서 : http://localhost:4200/children

http://localhost:4200/login?session_id=1234#anchor 주소로 리다이렉트 된다.



app.module.ts

```
/* feature module */
import {MemberModule} from './member/member.module';
import {PlayerModule} from './player/player.module';
import {ChildrenModule} from './children/children.module';

@NgModule({
  imports: [
    BrowserModule, CommonModule, FormsModule,
    AppRoutingModule,
    MemberModule, PlayerModule, ChildrenModule
  ],
  …
  bootstrap: [AppComponent]
})
export class AppModule {}
```

children.module.ts

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

import {ChildrenRoutingModule} from './children-routing.module';
import {ChildrenComponent} from './children.component';
import {Child1Component} from './child1.component';
import {Child2Component} from './child2.component';
import {Child3Component} from './child3.component';

import {ChildrenResolve} from './children-resolve.service';
import {ChildrenService} from './children.service';

@NgModule({
```

```
  imports: [ChildrenRoutingModule, FormsModule],
  declarations: [ChildrenComponent, Child1Component, Child2Component, Child3Component],
  providers: [ChildrenResolve, ChildrenService]
})
export class ChildrenModule {}
```

children-routing.module.ts

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

import {ChildrenComponent} from './children.component';
import {Child1Component} from './child1.component';
import {Child2Component} from './child2.component';
import {Child3Component} from './child3.component';

import {CanDeactivateGuard} from '../can-deactivate-guard.service';
import {ChildrenResolve} from './children-resolve.service';
import {AuthGuard} from '../auth-guard.service';

@NgModule({
  imports: [RouterModule.forChild([
    {
      path: 'children', component: ChildrenComponent,
      children: [{
        path: '',
        component: Child1Component,
        children: [
          {
            path: '',
            canActivate: [AuthGuard],
            component: Child2Component
          },
          {
            path: ':id',
            component: Child3Component,
            canDeactivate: [CanDeactivateGuard],
            resolve: {
              childrenResolve: ChildrenResolve
            }
          }
        ]
      }]
    },
    {
      path: 'active', component: ChildrenComponent,
      children: [{
        path: '',
        canActivateChild: [AuthGuard],
        children: [
          {path: 'child1', component: Child1Component},
          {path: 'child2', component: Child3Component},
          {path: '', component: Child1Component}
        ]
      }]
    }
  ])],
  exports: [RouterModule]
})
export class ChildrenRoutingModule {}
```

ChildrenComponent 가 <router-outlet></router-outlet>을 갖고 있다.

http://localhost:4200/children 주소로 접근 시 그 영역에 Child1Component 를 배치한다.

children.component.ts

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

@Component({
  selector: 'children',
  template: `
  <div>자식 라우터 표시</div>
  <router-outlet></router-outlet>
  `
})
export class ChildrenComponent {}
```

Child1Component 가 <router-outlet></router-outlet>을 갖고 있다.

child1.component.ts

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

@Component({
  selector: 'child1',
  template: `
  <h3>자식 1</h3>
  <router-outlet></router-outlet>`
})
export class Child1Component {}
```

http://localhost:4200/children 주소로 접근 시 그 영역에 Child2Component 를 배치한다.

이 때 AuthGuard 가 작동하여 접근 여부를 결정한다.

child2.component.ts

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

@Component({
  selector: 'child2',
  template: `
  <h3>자식 2</h3>`
})
export class Child2Component {}
```

http://localhost:4200/children/:id 주소로 접근 시 그 영역에 Child3Component 를 배치한다.

이 때 ChildrenResolve 가드가 작동하여 라우트 데이터를 Child3Component 에 제공한다.

child3.component.ts

```
import {Component} from '@angular/core';
import {Children} from './children.service';
import {ActivatedRoute} from '@angular/router';

@Component({
  selector: 'child3',
  template: `
  <h3>자식 3</h3>
  <input type="text" [(ngModel)]="editName"> {{editName}}
  `
})
export class Child3Component {
  children: Children;
  editName: string;

  constructor(private route: ActivatedRoute) {}

  ngOnInit() {
    this.route.data.forEach((data: {childrenResolve: Children}) => {
      this.editName = data.childrenResolve.name;
      this.children = data.childrenResolve;
    });
  }
}
```

AuthGuard

auth-guard.service.ts

```
import {Injectable} from '@angular/core';
import {
  CanActivate, CanActivateChild, CanLoad,
  Router, NavigationExtras,
  ActivatedRouteSnapshot, RouterStateSnapshot, Route
} from '@angular/router';

import {AuthService} from './auth.service';

@Injectable()
export class AuthGuard implements CanActivate, CanActivateChild, CanLoad {
  constructor(private authService: AuthService, private router: Router) {}

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean {
    let url: string = state.url;
    return this.checkLogin(url);
  }

  canActivateChild(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean {
    return this.canActivate(route, state);
  }

  canLoad(route: Route): boolean {
    let url = `/${route.path}`;
    if (window.confirm("자식 라우트가 모두 로드 되었습니다. 진행하시겠습니까?")) {
      return this.checkLogin(url);
    } else {
      return false;
    }
  }

  checkLogin(url: string): boolean {
    if (this.authService.isLogin) {
      return true;
    }
    this.authService.redirectUrl = url;
    let sessionId = 1234;

    let navigationExtras: NavigationExtras = {
      queryParams: {'session_id': sessionId},
      fragment: 'anchor'
    };

    this.router.navigate(['/login'], navigationExtras);
    return false;
  }
}
```

this.authService.**isLogin**

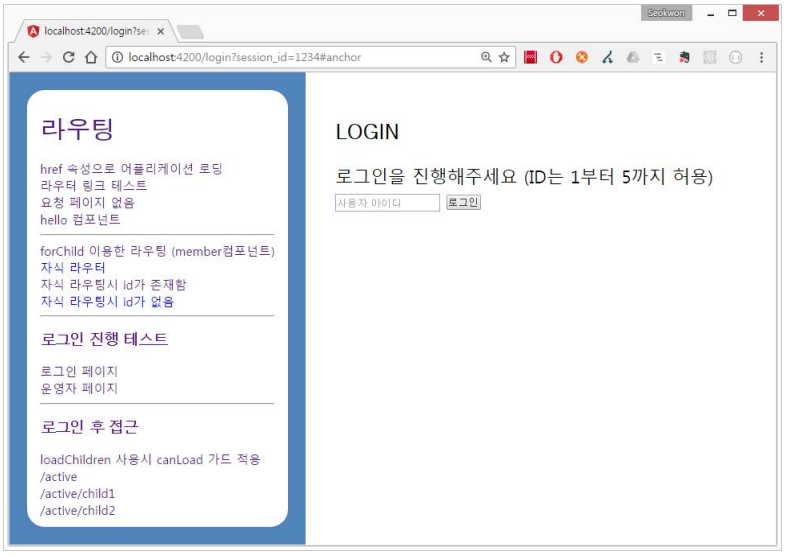
결과가 false 이므로 다음 주소로 리다이렉트 된다.

http://localhost:4200/login?session_id=1234#anchor

auth.service.ts

```
import {Injectable} from '@angular/core';
import {Observable} from 'rxjs/Observable';
import 'rxjs/add/observable/of';
import 'rxjs/add/operator/do';
import 'rxjs/add/operator/delay';

export class User {
  constructor(public id: number, public name: string) {}
}

const USERS = [
  new User(1, '첫번째 사용자'),
  new User(2, '두번째 사용자'),
  new User(3, '세번째 사용자')
];

export let userPromise = Promise.resolve(USERS);

@Injectable()
export class AuthService {
  isLogin: boolean = false;
  redirectUrl: string;
  userId: string;

  checkId(userId: string): Promise<User> {
    return userPromise
      .then(children => children.find(children => children.id === +userId));
  }

  login(userId: string): Observable<boolean> {
    return Observable.of(true).delay(500)
      .do(val => this.isLogin = true).do(val => this.userId = userId);
  }

  logout(): void {
    this.isLogin = false;
  }
}
```

연결순서 : http://localhost:4200/login?session_id=1234#anchor



app.routing.ts

```
const helloRoutes: Routes = [
  { path: '', component: IntroComponent },
  { path: 'hello', component: HelloComponent },
  { path: 'router-link-test', component: RouterLinkTestComponent },
  { path: 'href-test', component: HrefTestComponent },
  { path: 'pages/first-page', component: FirstPageComponent },
  { path: 'pages/second-page', component: SecondPageComponent },
  { path: 'pages/third-page', component: ThirdPageComponent },
  { path: 'login', component: LoginComponent },
  { path: 'admin', component: AdminComponent }
];
```

login.component.ts

```
import {Component} from '@angular/core';
import {Router, NavigationExtras} from '@angular/router';
import {AuthService} from './auth.service';

@Component({
  template: `
    <h3>LOGIN</h3> {{message}}
    <p>
      <input type="text" [(ngModel)]="userId"
         placeholder="사용자 아이디" *ngIf="!authService.isLogin">
      <button (click)="login()"  *ngIf="!authService.isLogin">로그인</button>
      <button (click)="logout()" *ngIf="authService.isLogin">로그아웃</button>
    </p>`
})
export class LoginComponent {
  message: string;
  userId: string;
```

```
  constructor(public authService: AuthService, public router: Router) {
    this.setMessage();
  }

  setMessage() {
    this.message = (this.authService.isLogin ?
      this.authService.userId + '님 로그인이 되었습니다.' :
      '로그인을 진행해주세요 (ID는 1부터 5까지 허용)');
  }

  private doLogin() {
    this.setMessage();
    if (this.authService.isLogin) {
      let redirect = this.authService.redirectUrl ?
         this.authService.redirectUrl : '/admin';

      let navigationExtras: NavigationExtras = {
        preserveQueryParams: true,
        preserveFragment: true
      };
      this.router.navigate([redirect], navigationExtras);
    } else {
      alert('로그인을 할 수 없습니다.');
    }
  }

  login() {
    if (!this.userId) {
      alert('id를 입력해주세요');
      return;
    }
    this.message = '로그인을 진행해주세요';

    return this.authService.checkId(this.userId).then(children => {
      if (children) {
        this.authService.login(this.userId).subscribe(() => this.doLogin());
      } else {
        alert('아이디가 없습니다');
      }
      this.setMessage();
    });
  }

  logout() {
    this.authService.logout();
    this.setMessage();
  }
}
```

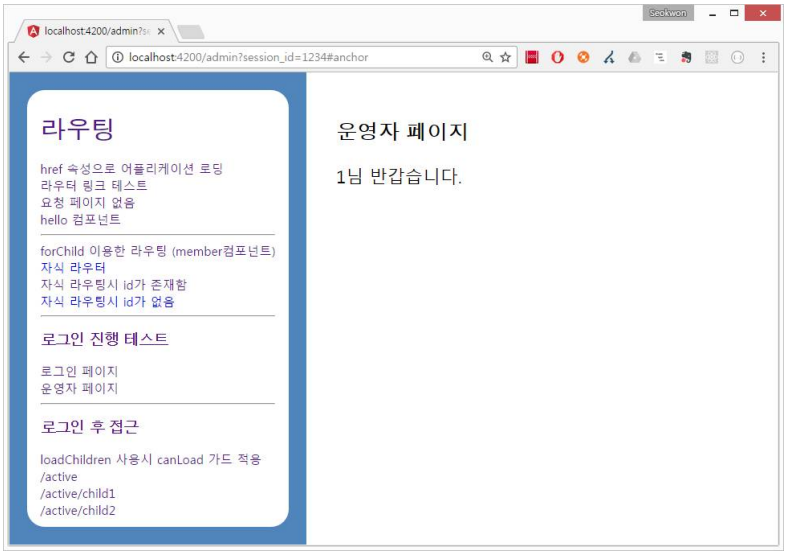
auth.service.ts

```
import {Injectable} from '@angular/core';
import {Observable} from 'rxjs/Observable';
import 'rxjs/add/observable/of';
import 'rxjs/add/operator/do';
import 'rxjs/add/operator/delay';

export class User {
```

```
  constructor(public id: number, public name: string) {}
}

const USERS = [
  new User(1, '첫번째 사용자'),
  new User(2, '두번째 사용자'),
  new User(3, '세번째 사용자')
];

export let userPromise = Promise.resolve(USERS);

@Injectable()
export class AuthService {
  isLogin: boolean = false;
  redirectUrl: string;
  userId: string;

  checkId(userId: string): Promise<User> {
    return userPromise
      .then(children => children.find(children => children.id === +userId));
  }

  login(userId: string): Observable<boolean> {
    return Observable.of(true).delay(500)
      .do(val => this.isLogin = true).do(val => this.userId = userId);
  }

  logout(): void {
    this.isLogin = false;
  }
}
```

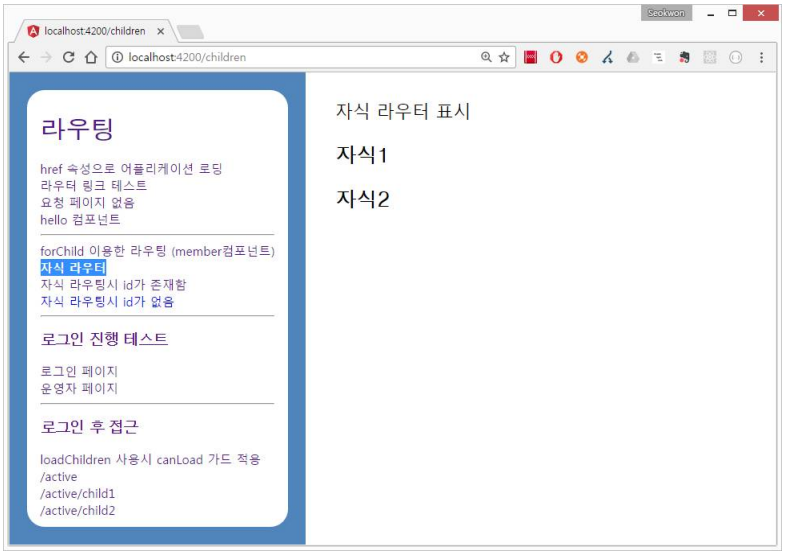
Observable.*of*

Creates an Observable that emits some values you specify as arguments

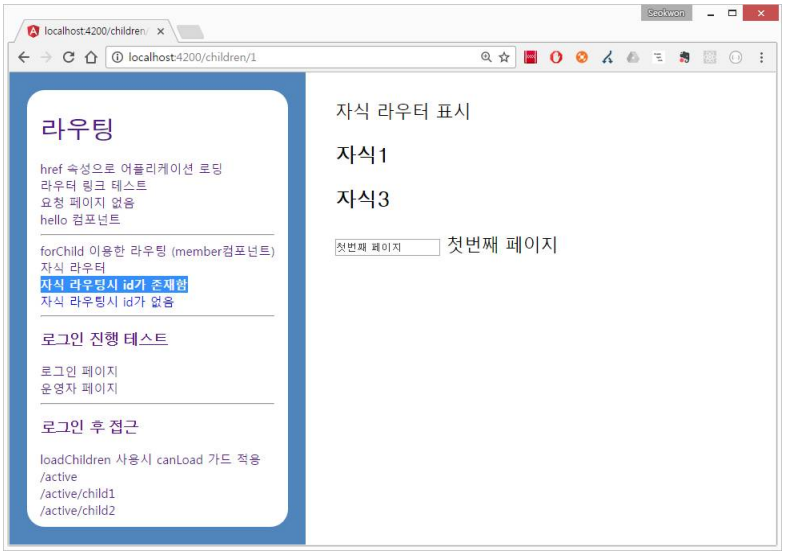


연결순서 : http://localhost:4200/children

로그인을 한 후 위 주소로 접속하면 원하는 화면을 볼 수 있다.



연결순서 : http://localhost:4200/children/1



children-routing.module.ts

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

import {ChildrenComponent} from './children.component';
import {Child1Component} from './child1.component';
import {Child2Component} from './child2.component';
import {Child3Component} from './child3.component';

import {CanDeactivateGuard} from '../can-deactivate-guard.service';
import {ChildrenResolve} from './children-resolve.service';
import {AuthGuard} from '../auth-guard.service';

@NgModule({
  imports: [RouterModule.forChild([
    {
      path: 'children', component: ChildrenComponent,
      children: [{
        path: '',
        component: Child1Component,
        children: [
          {
            path: '',
            canActivate: [AuthGuard],
            component: Child2Component
          },
          {
            path: ':id',
            component: Child3Component,
            canDeactivate: [CanDeactivateGuard],
            resolve: {
              childrenResolve: ChildrenResolve
            }
          }
        ]
      }]
```

```
    },
    {
      path: 'active', component: ChildrenComponent,
      children: [{
        path: '',
        canActivateChild: [AuthGuard],
        children: [
          {path: 'child1', component: Child1Component},
          {path: 'child2', component: Child3Component},
          {path: '', component: Child1Component}
        ]
      }]
    }
  ])],
  exports: [RouterModule]
})
export class ChildrenRoutingModule {}
```

childrent.component.ts

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

@Component({
  selector: 'children',
  template: `
  <div>자식 라우터 표시</div>
  <router-outlet></router-outlet>
  `
})
export class ChildrenComponent {}
```

child1.component.ts

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

@Component({
  selector: 'child1',
  template: `
  <h3>자식 1</h3>
  <router-outlet></router-outlet>`
})
export class Child1Component {}
```

children-resolve.service.ts

```
import {Injectable} from '@angular/core';
import {Router, Resolve, ActivatedRouteSnapshot} from '@angular/router';

import {Children, ChildrenService} from './children.service';

@Injectable()
export class ChildrenResolve implements Resolve<Children> {

  constructor(private cs: ChildrenService, private router: Router) {}

  resolve(route: ActivatedRouteSnapshot): Promise<Children> | boolean {
    let id = +route.params['id'];

    return this.cs.findById(id).then(children => {
      if (children) {
        return children;
      } else {
        this.router.navigate(['/not-found']);
        return false;
      }
    });
  }
}
```

children.service.ts

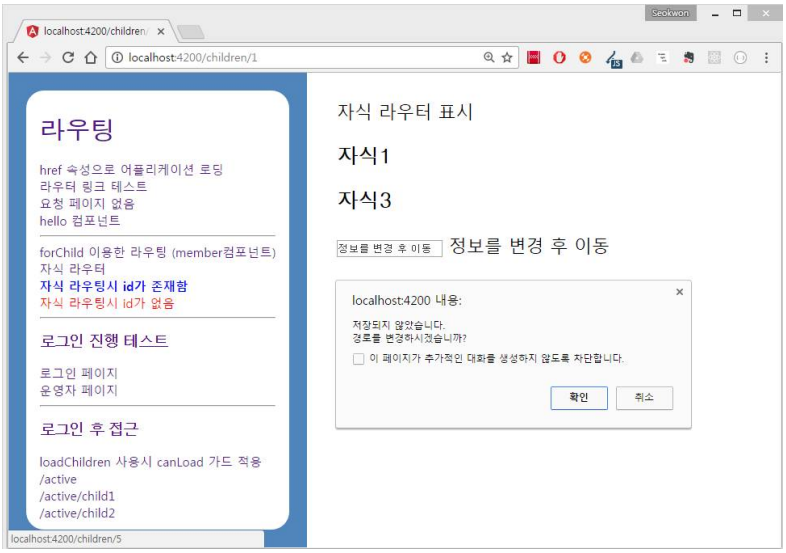
```
export class Children {
  constructor(public id: number, public name: string) {}
}

const CHILDREN = [
  new Children(1, '첫번째 페이지'),
  new Children(2, '두번째 페이지'),
  new Children(3, '세번째 페이지')
];

export let childrenPromise = Promise.resolve(CHILDREN);

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

@Injectable()
export class ChildrenService {
  static nextId = 10;

  findById(id: number | string) {
    return childrenPromise
      .then(children => children.find(children => children.id === +id));
  }
}
```

child3.component.ts

```
import {Component} from '@angular/core';
import {Children} from './children.service';
import {ActivatedRoute} from '@angular/router';

@Component({
  selector: 'child3',
  template: `
  <h3>자식 3</h3>
  <input type="text" [(ngModel)]="editName"> {{editName}}
  `
})
export class Child3Component {
  children: Children;
  editName: string;

  constructor(private route: ActivatedRoute) {}

  ngOnInit() {
    console.log('this.route.data = '+JSON.stringify(this.route.data));
    // {
    //   "_isScalar":false,
    //   "observers":[],
    //   "closed":false,
    //   "isStopped":false,
    //   "hasError":false,
    //   "thrownError":null,
    //   "_value":{
    //   "childrenResolve":{
    //       "id":1,
    //       "name":"첫번째 페이지"
    //     }
    //   }
    // }

    this.route.data.forEach((data: {childrenResolve: Children}) => {
      this.editName = data.childrenResolve.name;
      this.children = data.childrenResolve;
    });
  }
}
```

연결순서 : /children/1 → /children/5

input 상자에 정보를 변경 후 다른 경로로 이동하려 한다.



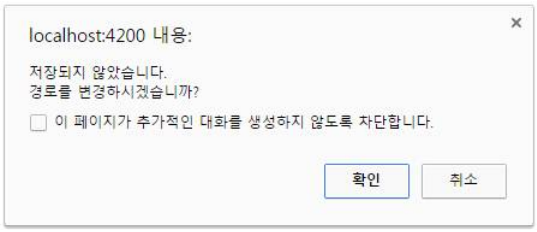
children-routing.module.ts

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

import {ChildrenComponent} from './children.component';
import {Child1Component} from './child1.component';
import {Child2Component} from './child2.component';
import {Child3Component} from './child3.component';

import {CanDeactivateGuard} from '../can-deactivate-guard.service';
import {ChildrenResolve} from './children-resolve.service';
import {AuthGuard} from '../auth-guard.service';

@NgModule({
  imports: [RouterModule.forChild([
    {
      path: 'children', component: ChildrenComponent,
      children: [{
        path: '',
        component: Child1Component,
        children: [
          {
            path: '',
            canActivate: [AuthGuard],
            component: Child2Component
          },
          {
            path: ':id',
            component: Child3Component,
            canDeactivate: [CanDeactivateGuard],
            resolve: {
              childrenResolve: ChildrenResolve
            }
          }
```

```
        ]
      }]
    },
    {
      path: 'active', component: ChildrenComponent,
      children: [{
        path: '',
        canActivateChild: [AuthGuard],
        children: [
          {path: 'child1', component: Child1Component},
          {path: 'child2', component: Child3Component},
          {path: '', component: Child1Component}
        ]
      }]
    }
  ])],
  exports: [RouterModule]
})
export class ChildrenRoutingModule {}
```

can-deactivate-guard.service

```
import {Injectable} from '@angular/core';
import {CanDeactivate, Router} from '@angular/router';
import {Observable} from 'rxjs/Observable';

export interface CanComponentDeactivate {
  canDeactivate: () => Observable<boolean> | Promise<boolean> | boolean;
}
@Injectable()
export class CanDeactivateGuard implements CanDeactivate<CanComponentDeactivate> {

  constructor(private _router: Router) {}

  canDeactivate() {
    return window.confirm("저장되지 않았습니다.\n 경로를 변경하시겠습니까?");
  }
}
```



취소를 누르면 현재 페이지가 그대로 유지된다.

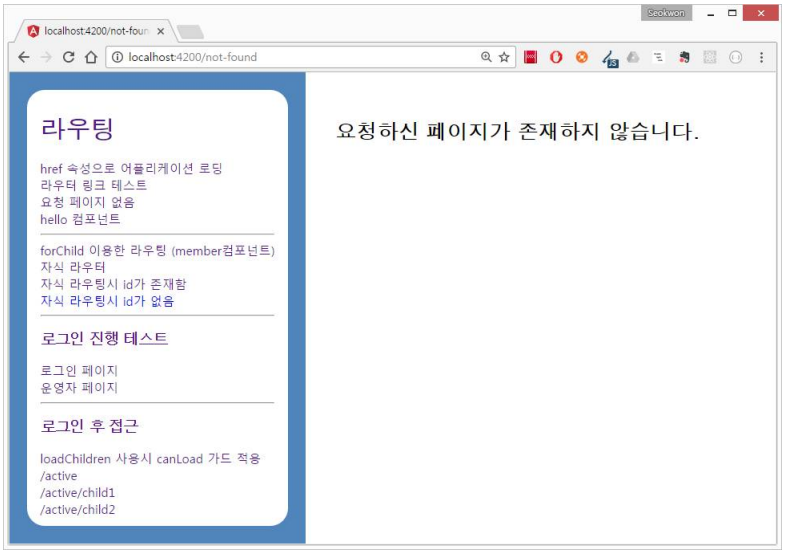
확인을 누르면 http://localhost:4200/children/5 주소로 라우팅이 요청되지만 해당 URL 에 대응하는 라우팅 정보가 없다.

app.routing.ts

```
const appRoutes: Routes = [
  ...loginRoutes,
  ...lazyRoutes,
  ...helloRoutes,
  { path: '**', component: NotFoundComponent }
];
```

not-found.component.ts

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

@Component({
  selector: 'not-found',
  template: `<h3>요청하신 페이지가 존재하지 않습니다.</h3>`
})
export class NotFoundComponent {}
```



연결순서 : http://localhost:4200/lazy/player

app.routing.ts

```
import { AuthGuard } from './auth-guard.service';

const lazyRoutes: Routes = [
  {
    path: 'lazy',
    loadChildren: 'app/player/player.module#PlayerModule',
    canLoad: [AuthGuard]
  }
];
```

auth-guard.service.ts

```
import {Injectable} from '@angular/core';
import {
  CanActivate,
  Router,
  ActivatedRouteSnapshot,
  RouterStateSnapshot,
  CanActivateChild,
  NavigationExtras,
  CanLoad,
  Route
} from '@angular/router';
import {AuthService} from './auth.service';

@Injectable()
export class AuthGuard implements CanActivate, CanActivateChild, CanLoad {
  …

  canLoad(route: Route): boolean {

    let url = `/${route.path}`;
    if (window.confirm("자식 라우트가 모두 로드 되었습니다. 진행하시겠습니까?")) {
      return this.checkLogin(url);
    } else {
      return false;
    }
  }

  checkLogin(url: string): boolean {

    if (this.authService.isLogin) {
      return true;
    }
    this.authService.redirectUrl = url;
    let sessionId = 1234;

    let navigationExtras: NavigationExtras = {
      queryParams: {'session_id': sessionId},
      fragment: 'anchor'
    };

    this.router.navigate(['/login'], navigationExtras);
    return false;
```

```
  }
}
```

로그인을 한 후 다음 처리가 진행된다.

player.module.ts

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

import {PlayerRoutingModule} from './player-routing.module';
import {PlayerComponent} from './player.component';

@NgModule({
  imports: [PlayerRoutingModule],
  declarations: [PlayerComponent],
  providers: []
})
export class PlayerModule {}
```

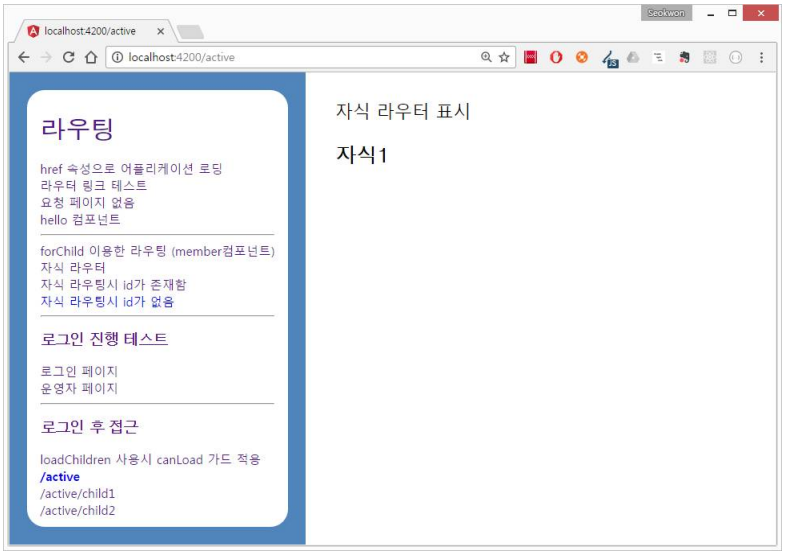
player-routing.module.ts

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

import {PlayerComponent} from './player.component';

@NgModule({
  imports: [RouterModule.forChild([
    {path: '', redirectTo: '', pathMatch: 'full'},
    {path: 'player', component: PlayerComponent}
  ])],
  exports: [RouterModule]
})
export class PlayerRoutingModule {}
```

player.component.ts

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

@Component({
  selector: 'player',
  template: `<h3>운영자만 볼 수 있는 Player 화면 입니다.</h3>`
})
export class PlayerComponent {}
```

연결순서 : http://localhost:4200/active



children-routing.module.ts

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

import {ChildrenComponent} from './children.component';
import {Child1Component} from './child1.component';
import {Child2Component} from './child2.component';
import {Child3Component} from './child3.component';

import {CanDeactivateGuard} from '../can-deactivate-guard.service';
import {ChildrenResolve} from './children-resolve.service';
import {AuthGuard} from '../auth-guard.service';

@NgModule({
  imports: [RouterModule.forChild([
    {
      path: 'children', component: ChildrenComponent,
      children: [{
        path: '',
        component: Child1Component,
        children: [
          {
            path: '',
            canActivate: [AuthGuard],
            component: Child2Component
          },
          {
            path: ':id',
            component: Child3Component,
            canDeactivate: [CanDeactivateGuard],
            resolve: {
              childrenResolve: ChildrenResolve
            }
          }
        ]
      }]
```

```
    },
    {
      path: 'active', component: ChildrenComponent,
      children: [{
        path: '',
        canActivateChild: [AuthGuard],
        children: [
          {path: 'child1', component: Child1Component},
          {path: 'child2', component: Child3Component},
          {path: '', component: Child1Component}
        ]
      }]
    }
  ])],
  exports: [RouterModule]
})
export class ChildrenRoutingModule {}
```

auth-guard.service.ts

```
import {Injectable} from '@angular/core';
import {
  CanActivate,
  Router,
  ActivatedRouteSnapshot,
  RouterStateSnapshot,
  CanActivateChild,
  NavigationExtras,
  CanLoad,
  Route
} from '@angular/router';
import {AuthService} from './auth.service';

@Injectable()
export class AuthGuard implements CanActivate, CanActivateChild, CanLoad {
  constructor(private authService: AuthService, private router: Router) {
  }

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean {
    let url: string = state.url;
    return this.checkLogin(url);
  }

  canActivateChild(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean {
    return this.canActivate(route, state);
  }

  canLoad(route: Route): boolean {

    let url = `/${route.path}`;
    if (window.confirm("자식 라우트가 모두 로드 되었습니다. 진행하시겠습니까?")) {
      return this.checkLogin(url);
    } else {
      return false;
    }
  }

  checkLogin(url: string): boolean {

    if (this.authService.isLogin) {
      return true;
    }
```

```
    this.authService.redirectUrl = url;
    let sessionId = 1234;

    let navigationExtras: NavigationExtras = {
      queryParams: {'session_id': sessionId},
      fragment: 'anchor'
    };

    this.router.navigate(['/login'], navigationExtras);
    return false;
  }
}
```

child1.component.ts

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

@Component({
  selector: 'child1',
  template: `
  <h3>자식 1</h3>
  <router-outlet></router-outlet>`
})
export class Child1Component {}
```

<router-outlet> </router-outlet>에 배정할 자식 컴포넌트를 지정하지 않았으므로 표시되지 않는다.

특징 모듈 라우터

```
□ 루트 모듈 - ❖ 루트 라우터
```

```
    □ 특징 모듈 1 - ❖ 특징 모듈 라우터 1
        ■ 컴포넌트 1
        ■ 컴포넌트 2
```

```
    □ 특징 모듈 2 - ❖ 특징 모듈 라우터 2
        ■ 컴포넌트 3
        ■ 컴포넌트 4
```

루트 모듈에서 특징 모듈의 라우터 설정 정보를 받아서 특징 모듈 간에 라우팅을 처리한다.

특징 모듈 1 은 특징 모듈 라우터 1 을 포함하고 루트 모듈은 특징 모듈 1 을 포함하는 관계이다.

예를 들어 특징 모듈 1 에 있는 컴포넌트 1 은 특징 모듈 2 에 컴포넌트 3 으로 대체될 수 있다.

src/app/member/member-routing.module.ts

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

import { MemberComponent }    from './member.component';

@NgModule({
  imports: [RouterModule.forChild([
    { path: 'member', component: MemberComponent}
  ])],
  exports: [RouterModule]
})
export class MemberRoutingModule {}
```

RouterModule.*forChild*

member 특징 모듈에 대한 라우터 정보를 설정한다.

루트 모듈에 추가되기 위해서 forChild 메소드를 사용한다.

src/app/member/member.module.ts

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

import { MemberComponent }   from './member.component';
import { MemberRoutingModule }   from './member-routing.module';

@NgModule({
  imports:      [ CommonModule, FormsModule, MemberRoutingModule ],
  declarations: [ MemberComponent ],
  providers:    [ ]
})
export class MemberModule { }
```

src/app/app.module.ts

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

/* application router settings */
import { AppRoutingModule, appRoutingProviders } from './app.routing';

/* feature module */
import { MemberModule } from './member/member.module';
import { PlayerModule } from './player/player.module';
import { ChildrenModule } from './children/children.module';
```

```
/* global components */
import { AppComponent } from './app.component';
import { IntroComponent } from './intro.component';
import { HelloComponent } from './hello/hello.component';
import { RouterLinkTestComponent } from './router-link-test/router-link-test.component';
import { HrefTestComponent } from './router-link-test/href-test.component';
import { FirstPageComponent } from './pages/first-page.component';
import { SecondPageComponent } from './pages/second-page.component';
import { ThirdPageComponent } from './pages/third-page.component';
import { LoginComponent } from './login.component';
import { AdminComponent } from './admin/admin.component';
import { NotFoundComponent } from './not-found.component';

@NgModule({
  imports: [
    BrowserModule, CommonModule, FormsModule,
    AppRoutingModule,
    MemberModule, PlayerModule, ChildrenModule
  ],
  providers: [appRoutingProviders],
  declarations: [
    AppComponent, IntroComponent, HelloComponent,
    RouterLinkTestComponent,
    HrefTestComponent,
    FirstPageComponent, SecondPageComponent, ThirdPageComponent,
    LoginComponent,
    AdminComponent,
    NotFoundComponent
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Guard

가드는 라우팅 시 접근을 제어하는 방법이다. 가드는 크게 다음과 같은 종류가 있다.

■ CanActivate

라우트 권한을 검사해 접근 허용을 결정하는 가드이다.

■ CanActivateChild

자식 라우트에 대한 접근 허용을 결정하는 가드이다.

■ CanDeactivate

다른 라우터로 변경되어 현재 라우트가 비활성화될 때 호출되는 가드이다.

■ Resolve

라우트 데이터를 가져와 컴포넌트에 제공하는 가드이다.

src/app/children/children-routing.module.ts

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

import { ChildrenComponent } from './children.component';
import { Child1Component } from './child1.component';
import { Child2Component } from './child2.component';
import { Child3Component } from './child3.component';

import { CanDeactivateGuard } from '../can-deactivate-guard.service';
import { ChildrenResolve } from './children-resolve.service';
import { AuthGuard } from '../auth-guard.service';

@NgModule({
    imports: [RouterModule.forChild([
        {
            path: 'children', component: ChildrenComponent,
            children: [{
                path: '',
                component: Child1Component,
                children: [
                    {
                        path: '',
                        canActivate: [AuthGuard],
                        component: Child2Component
                    },
                    {
                        path: ':id',
                        component: Child3Component,
                        canDeactivate: [CanDeactivateGuard],
                        resolve: {
                            childrenResolve: ChildrenResolve
                        }
                    }
                ]
            }]
        },
        {
            path: 'active', component: ChildrenComponent,
            children: [{
                path: '',
                canActivateChild: [AuthGuard],
                children: [
                    { path: 'child1', component: Child1Component },
                    { path: 'child2', component: Child3Component },
                    { path: '', component: Child1Component }
                ]
            }]
        }
    ])],
    exports: [RouterModule]
})
export class ChildrenRoutingModule { }
```

```
canDeactivate: [CanDeactivateGuard],
```

다른 컴포넌트로 변경될 때 기동한다. 예를 들어 글을 쓰다가 페이지 주소가 바뀔 때 변경사항을 저장할지 말지를 사용자에게 알리기 위한 작업을 수행할 수 있다.

```
                    resolve: {
                        childrenResolve: ChildrenResolve
                    }
```

라우트 되는 시점에 작동해서 라우트와 관련된 데이터를 구해서 컴포넌트에게 제공하는 가드이다.
ActivateRoute 라우트를 통해 라우트되는 컴포넌트에 전달된다.

```
                canActivateChild: [AuthGuard],
                children: [
                    { path: 'child1', component: Child1Component },
                    { path: 'child2', component: Child3Component },
                    { path: '', component: Child1Component }
                ]
```

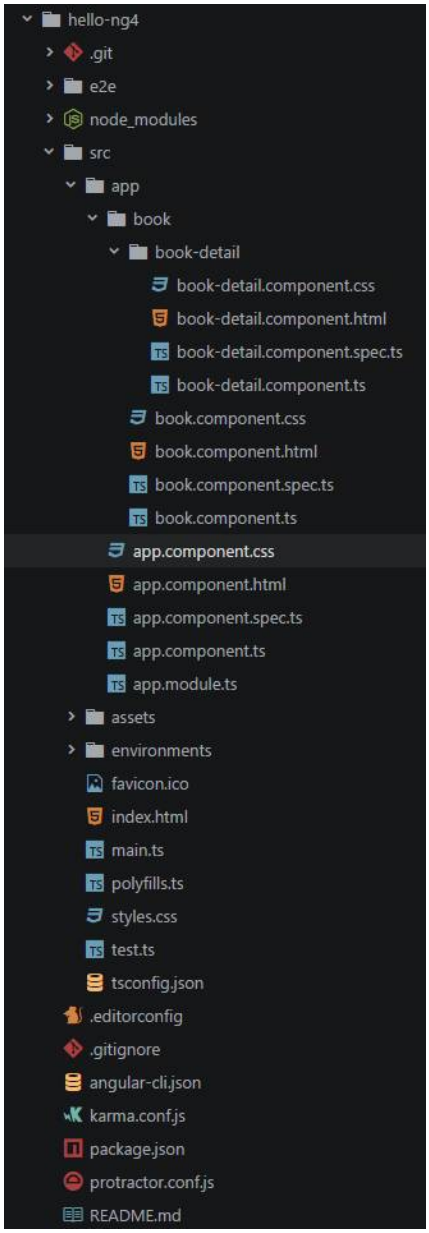
AuthGuard 는 children 프로퍼티에 정의된 라우트 정보의 접근을 허용할지 결정한다.

```
import { ChildrenComponent } from './children.component';
```

라우터 아울렛을 통해 라우팅 요청이 들어온 컴포넌트를 표시한다.
Child1Component, Child2Component, Child3Component 컴포넌트를 사용한다.

Style in Component

다음 그림을 참고하여 프로젝트를 만든다.



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 { AppComponent } from './app.component';
import { BookComponent } from './book/book.component';
import { BookDetailComponent } from './book/book-detail/book-detail.component';

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

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 component as grand parent';
}
```

app.component.html

```
<h1>
  {{title}}
</h1>
<div class="first">
  first div
</div>
<div class="second">
  second div
</div>
<div class="last">
  last div
</div>
<div class="child">
  <book class="active"></book>
</div>
```

app.component.css

```
/*컴포넌트 자신, :host 생략 불가*/

:host {
  display: block;
  padding: 10px;
  background-color: grey;
}


/*컴포넌트 자신 전체에 호버 이벤트가 발생할 때 적용*/

:host(:hover) {
  font-style: italic;
}


/*태그.클래스가 일치하는 엘리먼트에 적용, :host 생략 가능*/

:host div.first {
  color: red;
  font-size: 30px;
}


/*컴포넌트 자신 + 자식 컴포넌트들에게 적용*/

:host /deep/ div.second {
  color: blue;
  font-size: 30px;
}
```

book.component.ts

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

@Component({
  selector: 'book',
  templateUrl: './book.component.html',
  styleUrls: ['./book.component.css']
})
export class BookComponent implements OnInit {
  title: string = 'book component as parent'
  constructor() { }

  ngOnInit() { }

}
```

book.component.html

```
<h1>
  {{title}}
</h1>
<div class="first">
  first div
</div>
<div class="second">
  second div
</div>
<div class="last">
  last div
</div>
<div class="child">
  <book-detail class="active"></book-detail>
</div>
```

book.component.css

```
/*컴포넌트 자신, :host 생략 불가*/
:host {
  display: block;
  padding: 10px;
  background-color: orange;
}


/*
부모 HTML 에서 자식 컴포넌트 셀렉터(커스텀 태그) 선언 시 클래스를 설정했다면 적용
<book class="active"></book>

:host-context(.부모가 자식 셀렉터 사용 시 붙인 클래스) .현재 컴포넌트의 클래스
 */
:host-context(.active) .last{
  color: purple;
  font-size: 30px;
}
```

book-detail.component.ts

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

@Component({
  selector: 'book-detail',
  templateUrl: './book-detail.component.html',
  styleUrls: ['./book-detail.component.css']
})
export class BookDetailComponent implements OnInit {
  title: string = 'book-detail component as child'
  constructor() { }

  ngOnInit() {
  }

}
```

book-detail.component.html

```
<h1>
  {{title}}
</h1>
<div class="first">
  first div
</div>
<div class="second">
  second div
</div>
<div class="last">
  last div
</div>
<div class="child">
  this is child content <button>send</button>
</div>
```

book-detail.component.css

```
:host div {
  padding: 10px;
  background-color: yellow;
}

:host-context(.active) button {
  font-weight: bold;
  color: red;
}
```

확인


app component as grand parent
first div
second div
last div
book component as parent
first div
second div
last div
book-detail component as child
first div
second div
last div
this is child content
send

참고문헌

쉽고 빠르게 배우는 Angular 2 프로그래밍