

Angular (<http://angular.io>)



Learning Object(학습모듈) 및 커리큘럼



LO	커리큘럼
Angular2 소개	<ul style="list-style-type: none">- Angular2 개요- TypeScript 에센셜- Angular2 어플리케이션 작성
Angular2 주요 요소	<ul style="list-style-type: none">- 데이터 바인딩- 디렉티브- 파이프- 코드관리, 데이터모델- 데이터 바인딩, 이벤트 처리- Router
네트워크	<ul style="list-style-type: none">- 서비스- HTTP 모듈

Angular2 소개

- Angular2 개요
- TypeScript 에센셜
- Angular2 어플리케이션 작성

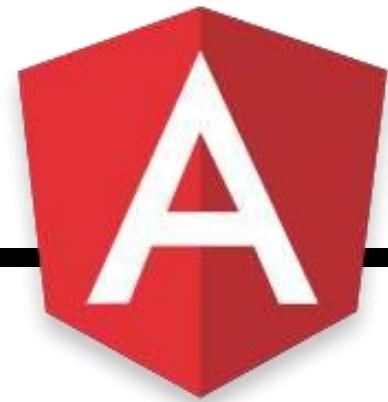
◎ 학습지식 개요/요점

Angular2 의 개념과 기본원리를 확인합니다.

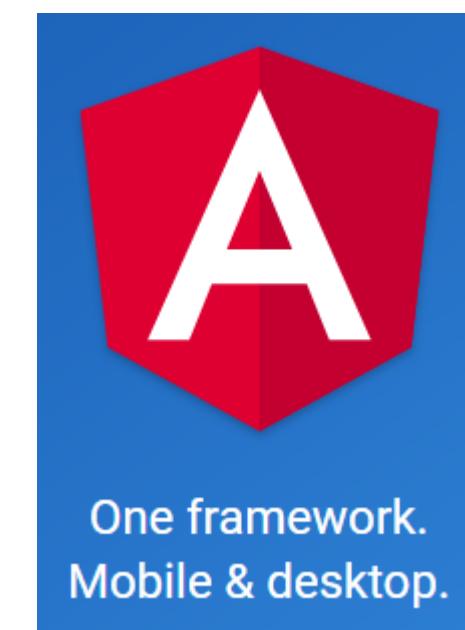
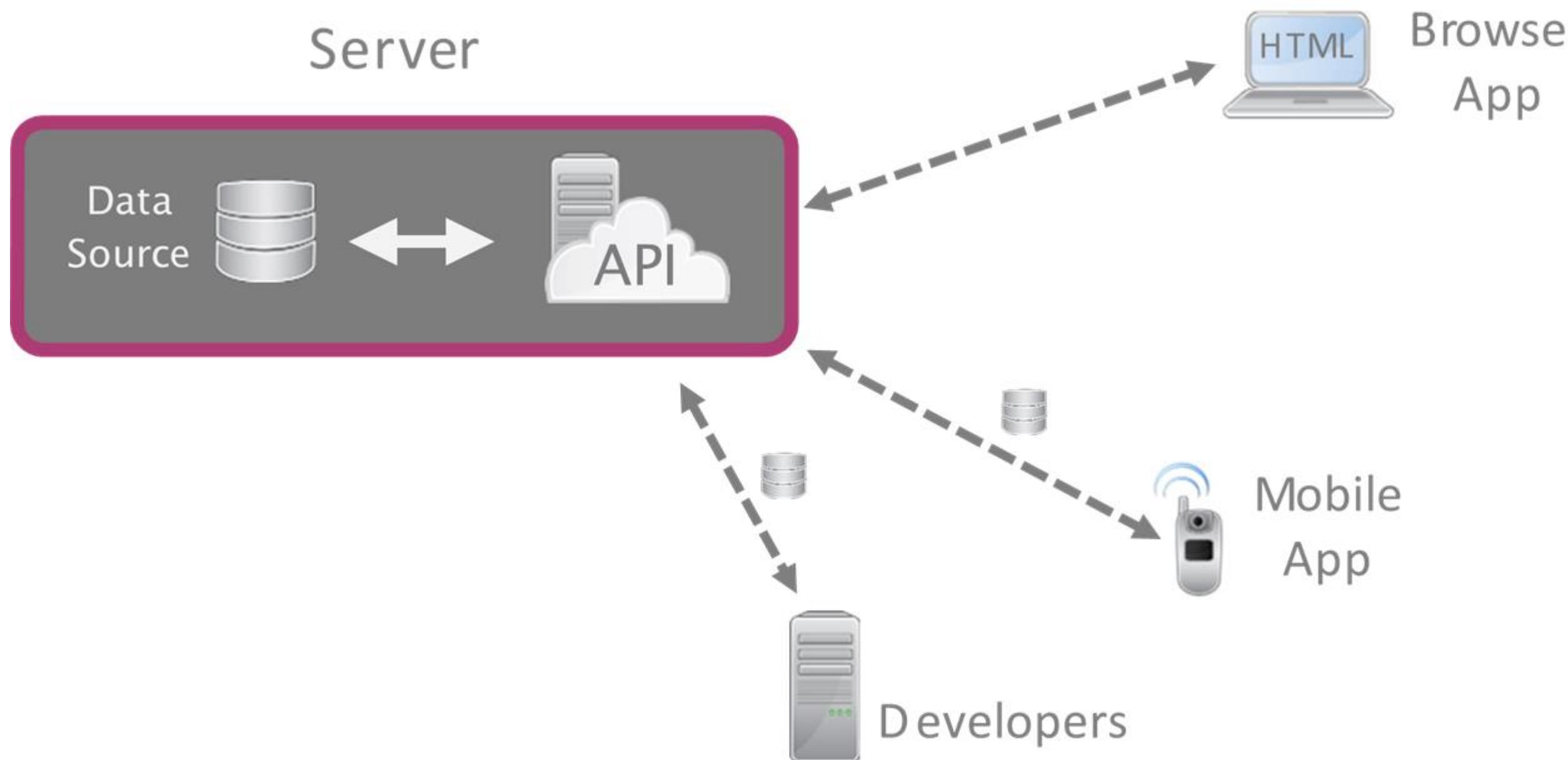
◎ 실습 예제 및 수행가이드

- Angular2 개요와 웹 컴포넌트
- TypeScript 필수 문법 이해 및 실습
- Angular2 HelloWorld app 작성

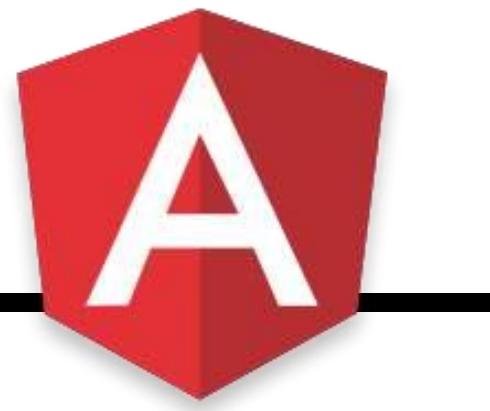
Angular2



- Google이 만든 오픈소스 다이나믹 웹 어플리케이션 프레임워크.
- 프론트엔드 코드를 단순하고 명료하게 하기위해서, HTML, JavaScript, 그리고 CSS 를 하나로 조합하는 방법을 제공.



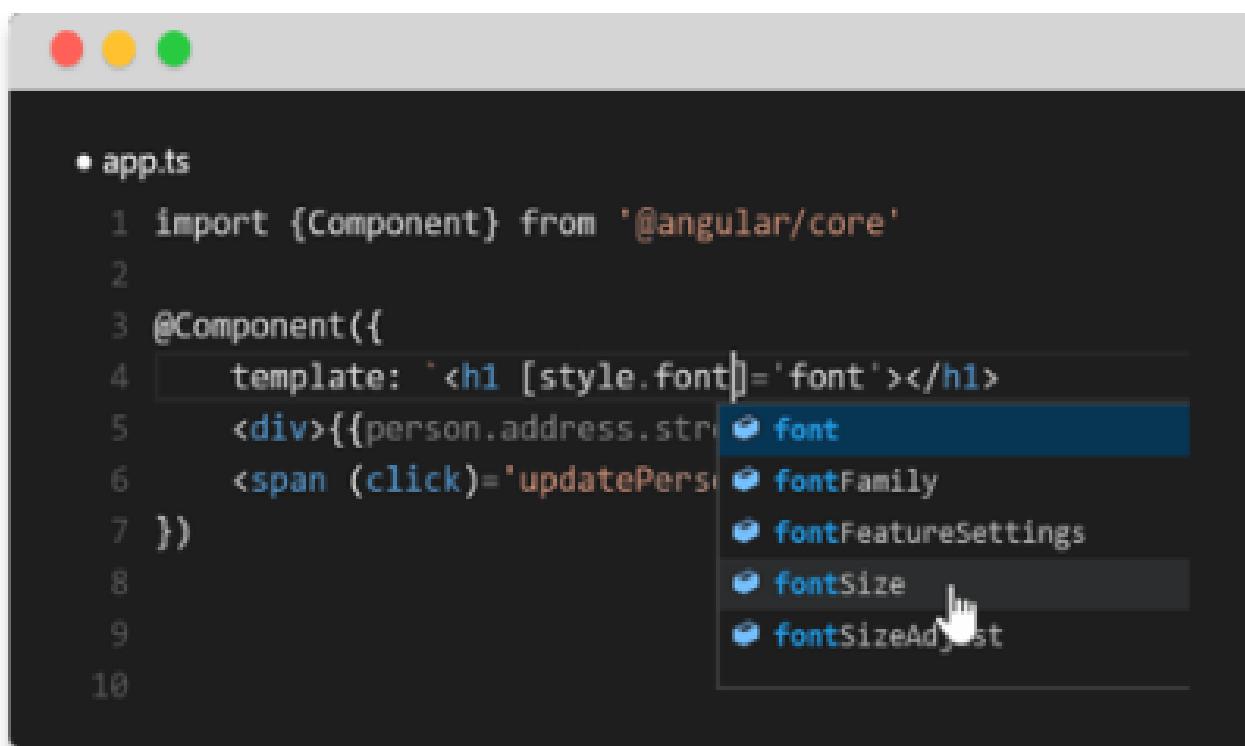
Angular2의 특징



Develop Across All Platforms



Incredible Tooling



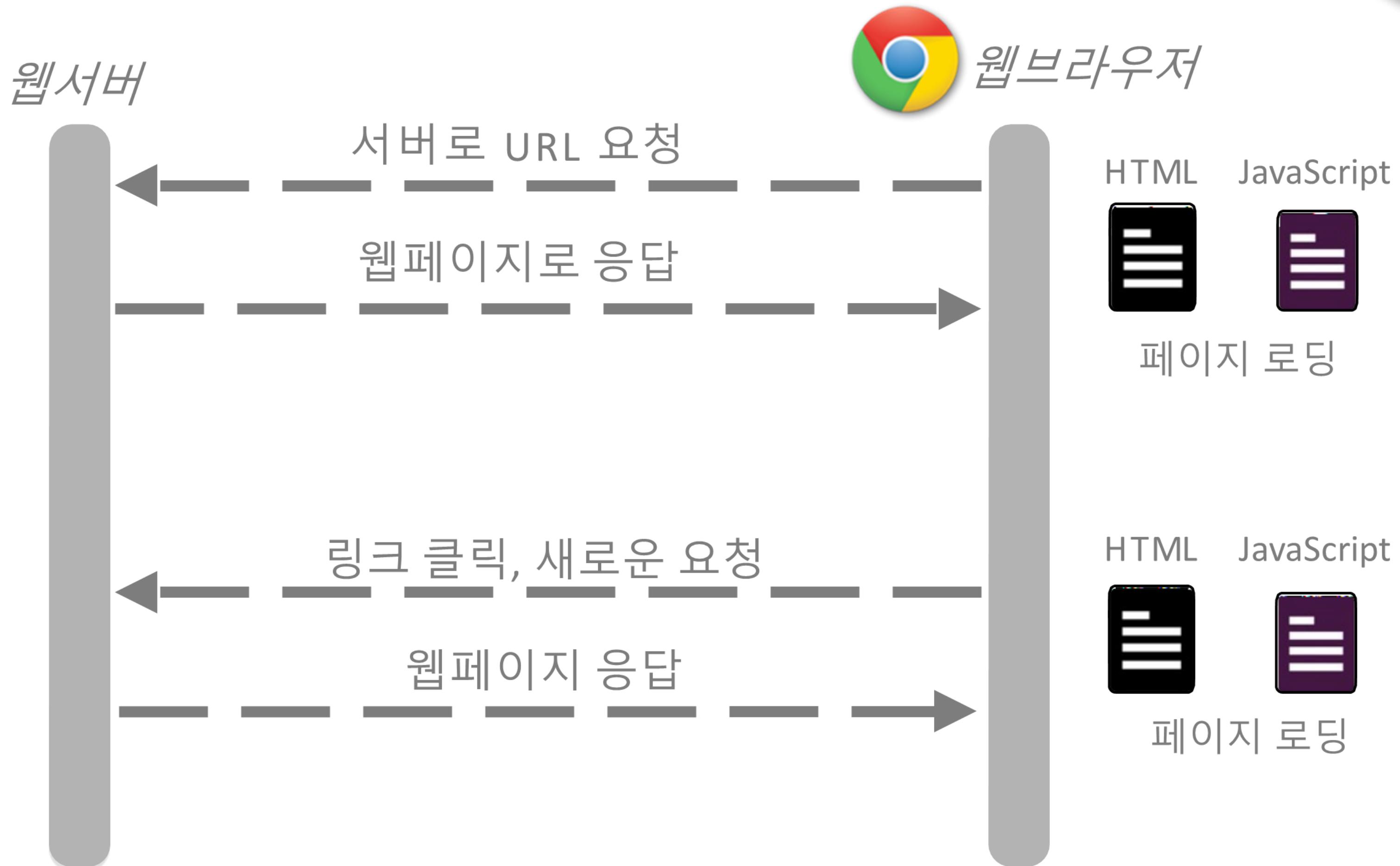
A screenshot of an IDE interface showing an Angular component definition in TypeScript. The code includes template and style declarations. A tooltip or code completion dropdown is open over the 'font' property in the template, listing several options: font, fontFamily, fontFeatureSettings, fontSize, and fontSizeAdjust. The 'fontSize' option is highlighted with a cursor.

```
• app.ts
1 import {Component} from '@angular/core'
2
3 @Component({
4   template: `<h1 [style.font]=`font`></h1>
5   <div>{{person.address.str}}</div>
6   <span (click)=`updatePerson`>${person.name}</span>
7 })
8
9
10
```

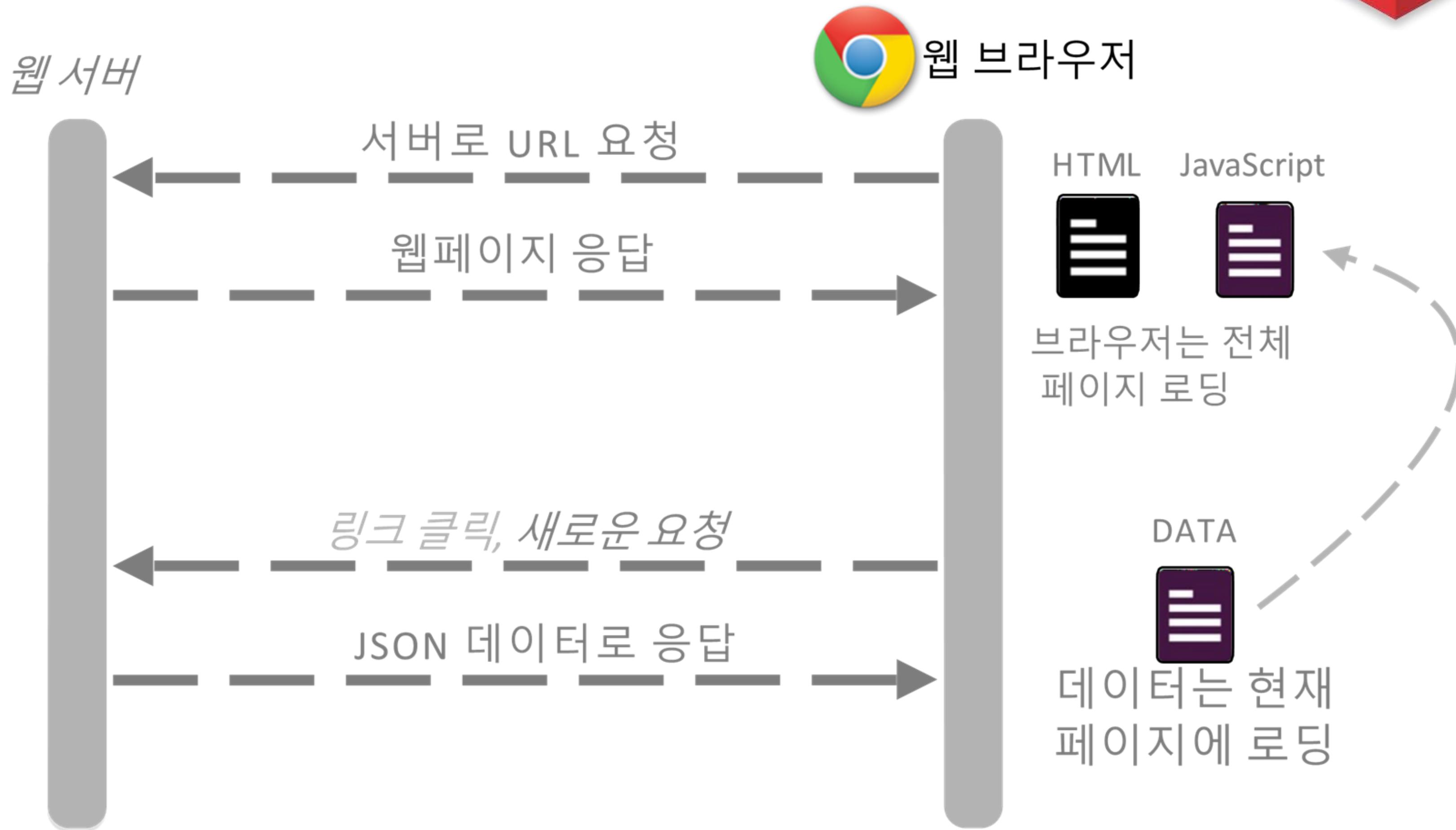
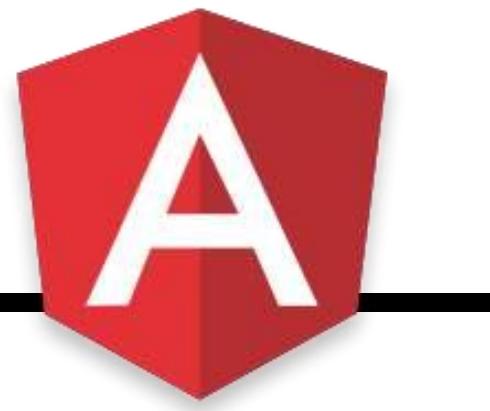
Loved by Millions



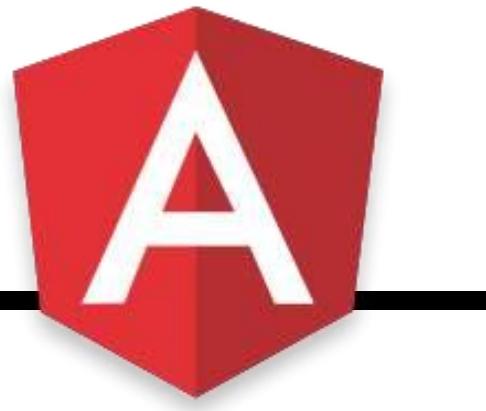
기존 웹 어플리케이션 구조



Data-Driven 웹 아키텍처

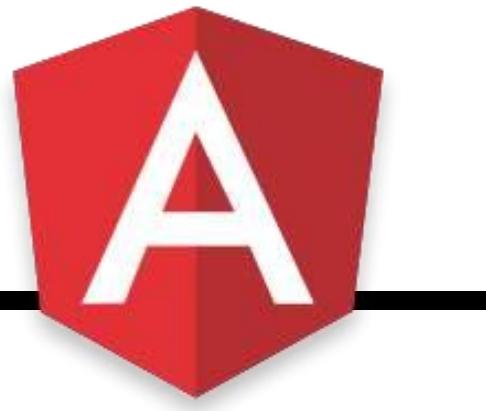


Angular의 History



- 2014년 10월
ngEurope conference에서 첫소개
- 2015년 4월 30일 (Alpha Version)
- 2015년 12월 (Beta Version)
<https://angular.io/> 에서 다운로드 받을 수 있게됨
- 2016년 5월
처음 angular2 release candidate가 되어 출시함
- 2017년 3월
v2.4.10
- 2017년 9월 현재
v4.4

Angular 1 & 2 차이점?



속도 — Angular 2 가 더 빠름.

컴포넌트 — Controller와 Scope 대신에 Component를 사용, 더 심플한 컨셉.

단순해진 디렉티브 — 사용자 정의 Directive의 작성이 단순해짐.

직관적 데이터 바인딩 — 데이터를 HTML에 바인딩 할 때나 버튼 클릭 이벤트를 부여할 때 더 직관적이고 단순한 문법을 제공.

서비스는 단순 클래스로 바뀜.

많은 사소한 개선사항이 있음.

Angular1 vs Angular2

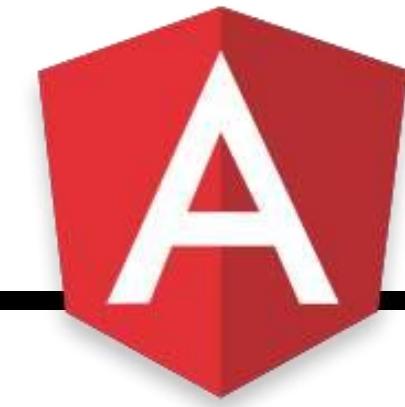


```
module.directive('myConfirmation', function() {
  return {
    scope: {},
    bindToController: {
      message: '=',
      onOk: '&'
    },
    controller: function() { },
    controllerAs: 'ctrl',
    template: `
      <div>
        {{ctrl.message}}
        <button ng-click="ctrl.onOk()">
          OK
        </button>
      </div>
    `
  }
});
```

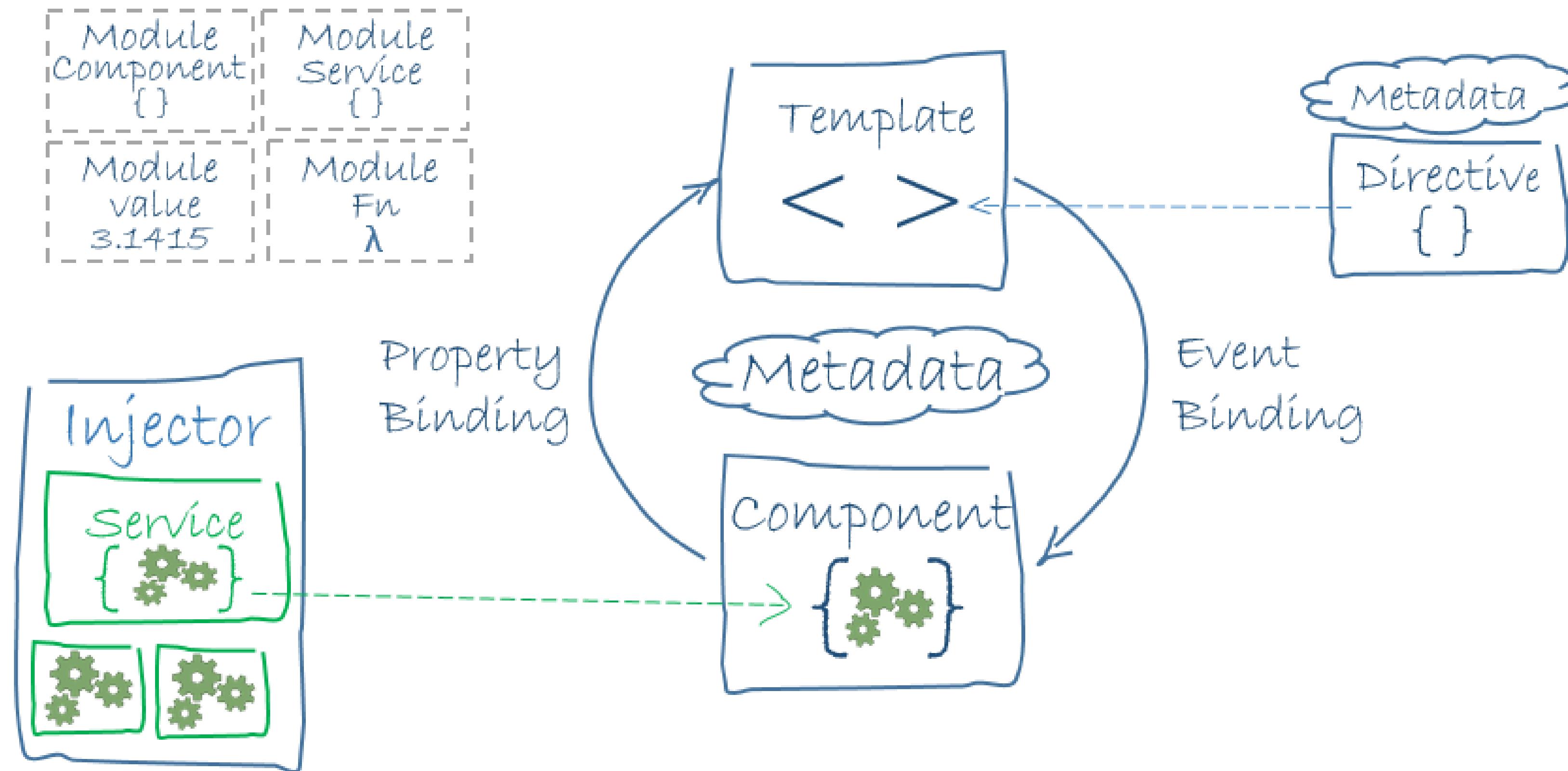
```
@Component({
  selector: 'my-confirmation',
  inputs: ['message'],
  outputs: ['ok']
})
@View({
  template: `
    <div>
      {{message}}
      <button (click)="ok()">OK</button>
    </div>
  `
})
class MyConfirmation {
  okEvents = new EventEmitter();
  ok() {
    this.okEvents.next();
  }
}
```

Reference : <http://teropa.info/blog/2015/10/18/refactoring-angular-apps-to-components.html>

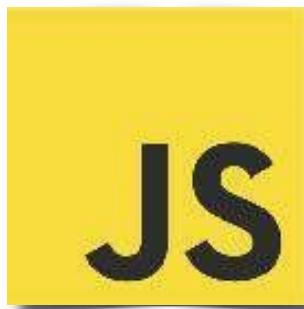
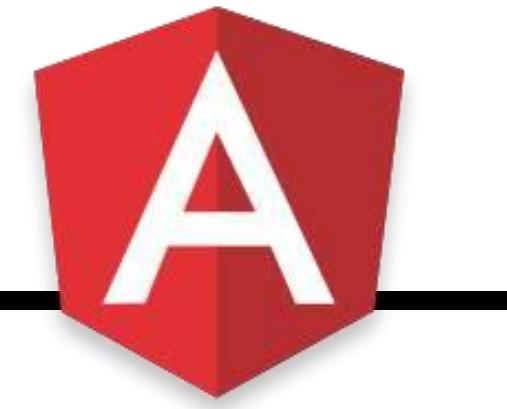
Angular의 아키텍쳐



- Component를 이용한 Template과 Service로직 관리



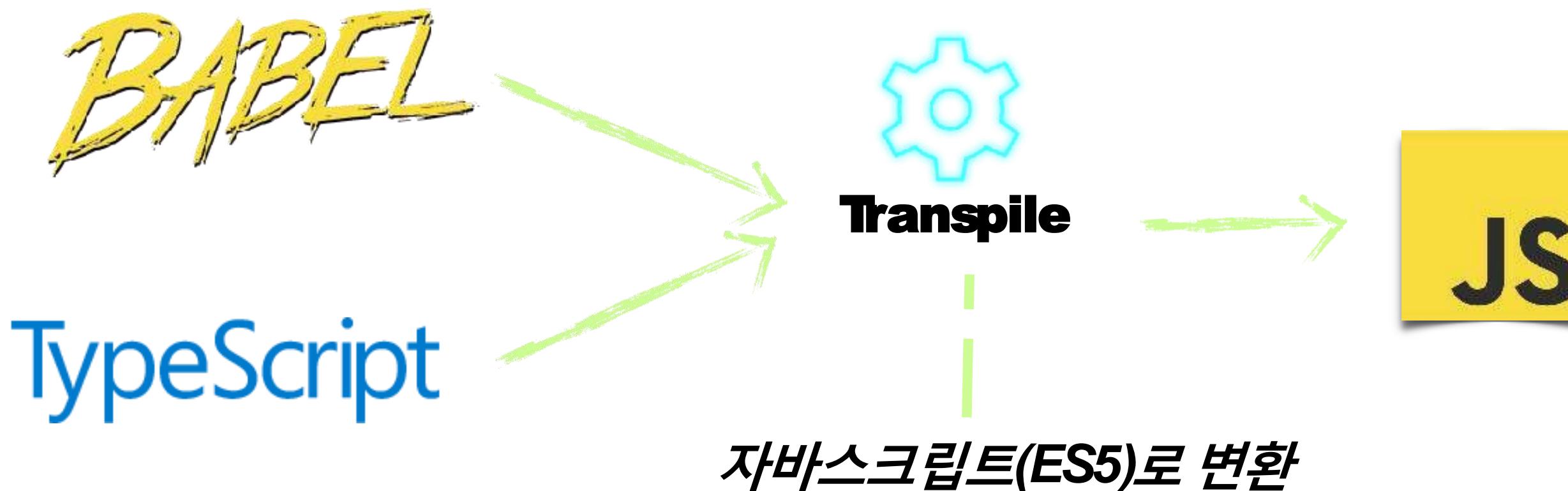
Angular2에서 사용되는 언어는?



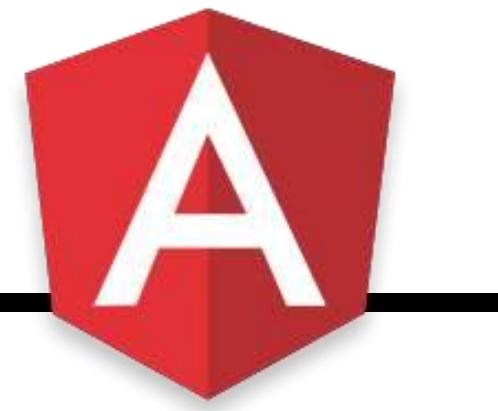
JavaScript

최신 자바스크립트 버전을 모든 브라우저가 지원하지 않음.

다음과 같은 방법으로 해결 가능:



Angular 개발을 위한 IDE



Angular IDE by Webclipse

Built first and foremost for Angular. Turnkey setup for beginners; powerful for experts.

IntelliJ IDEA

Capable and Ergonomic Java * IDE

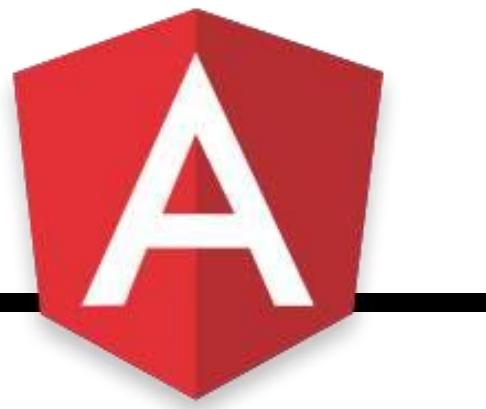
Visual Studio Code

VS Code is a Free, Lightweight Tool for Editing and Debugging Web Apps.

Webstorm

Lightweight yet powerful IDE, perfectly equipped for complex client-side development and server-side development with Node.js

Angular 개발을 위한 TOOL



Angular CLI

The official Angular CLI makes it easy to create and develop applications from initial commit to production deployment. It already follows our best practices right out of the box!

Angular Universal

Server-side Rendering for Angular 2 apps.

Augury

A Google Chrome Dev Tools extension for debugging Angular 2 applications.

Celerio Angular Quickstart

Generate an Angular 2 CRUD application from an existing database schema

Codelyzer

A set of tslint rules for static code analysis of Angular 2 TypeScript projects.

Lite-server

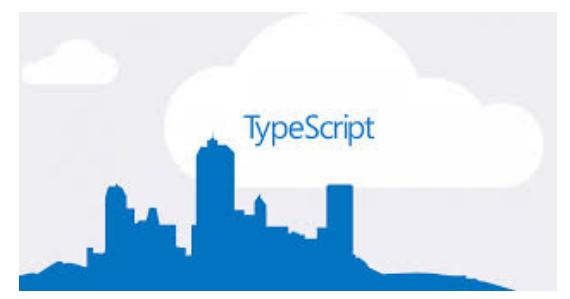
Lightweight development only node server

TypeScript



<http://www.typescriptlang.org>

ECMA 스크립트에 대하여



- ECMA 스크립트는 ECMA International의 표준

최초 ECMA 스크립트는 브라우저 언어인 Javascript와 Jscript간 차이를 줄이기 위한 공통 스펙 제안으로 출발 (1997, ECMA-262)

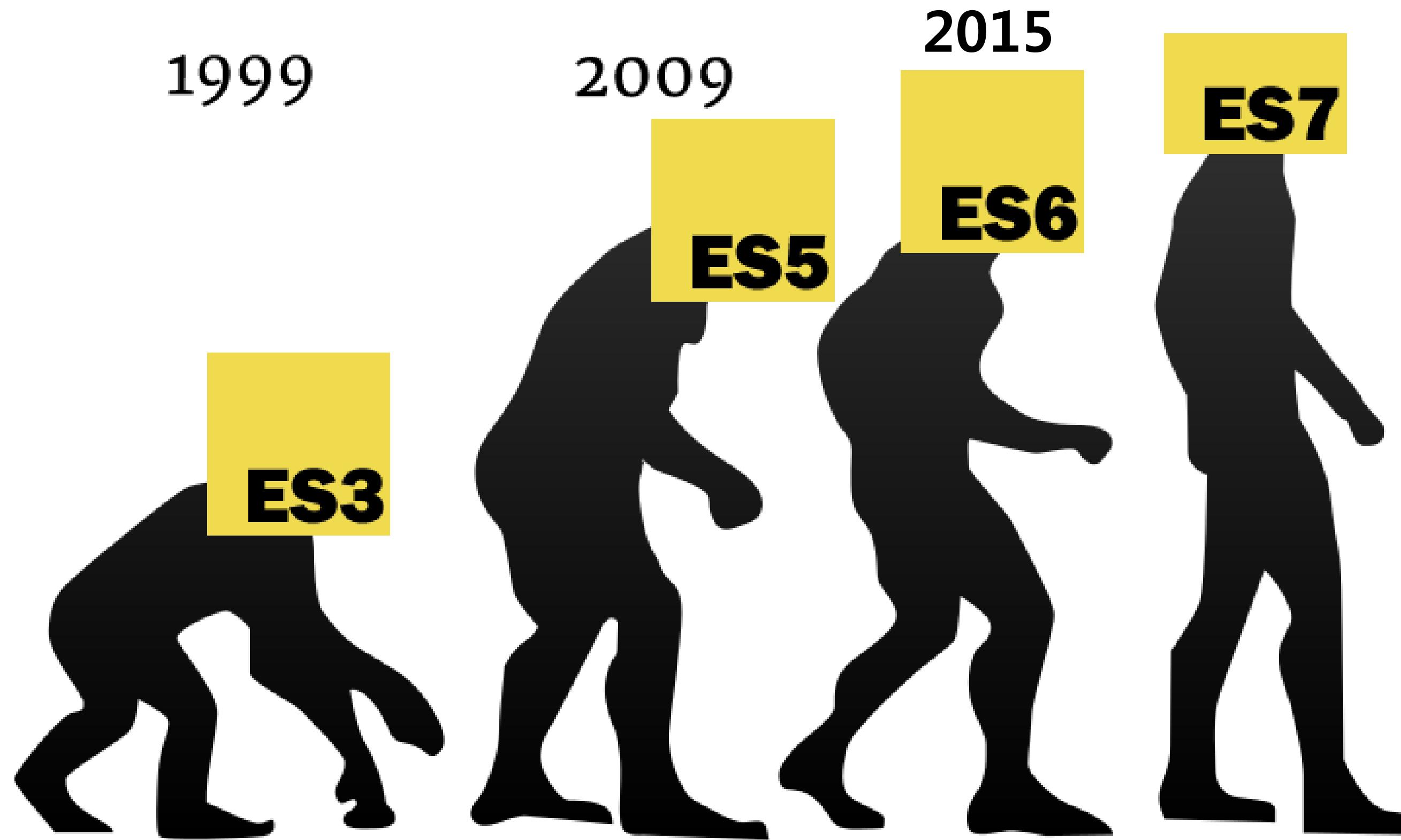
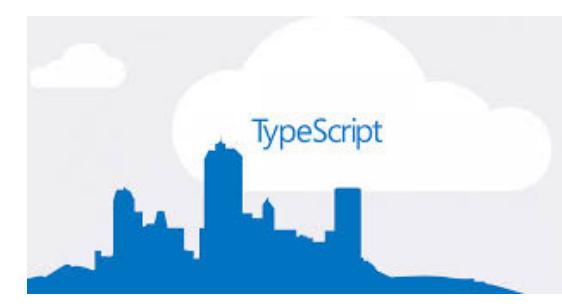
- ECMA International은 전세계적인 표준기관

유럽 컴퓨터 제조협회로부터 기원함

ECMA(European Computer Manufacturers Association)

C#, JSON, Dart을 포함한 많은 언어 표준을 관리함

ECMA Script(ES) 히스토리



TypeScript : www.typescriptlang.org



The screenshot shows the official TypeScript website at <https://www.typescriptlang.org>. The page features a dark blue header with the "TypeScript" logo and navigation links for Documentation, Samples, Download, Connect, and Playground. A banner at the top right encourages users to "Fork me on GitHub". The main content area has a large, stylized blue-toned illustration of a city skyline with a Ferris wheel and the Space Needle. Below the illustration, the word "TypeScript" is prominently displayed in white, followed by the tagline "JavaScript that scales." in orange. A descriptive paragraph explains that TypeScript is a typed superset of JavaScript that compiles to plain JavaScript, supporting "Any browser. Any host. Any OS. Open source." Two yellow buttons labeled "Download" and "Documentation" are visible. At the bottom, three icons represent the language's features: a brace icon for "Starts and ends with JavaScript", a gear icon for "Strong tools for large apps", and a pen icon for "State of the art JavaScript".

TypeScript - JavaScript the x https://www.typescriptlang.org

Documentation Samples Download Connect Playground

TypeScript 1.8 is now available. Download our latest version today!

Fork me on GitHub

TypeScript

JavaScript that scales.

TypeScript is a typed superset of JavaScript that compiles to plain JavaScript.

Any browser. Any host. Any OS. Open source.

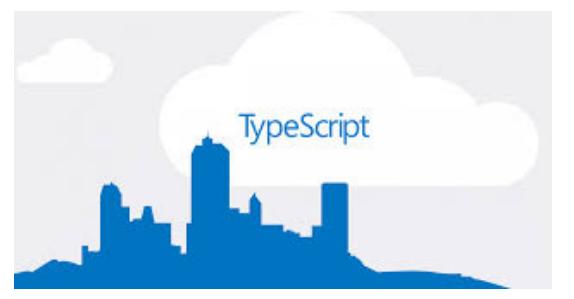
Download Documentation

{JS}

Starts and ends with JavaScript

Strong tools for large apps

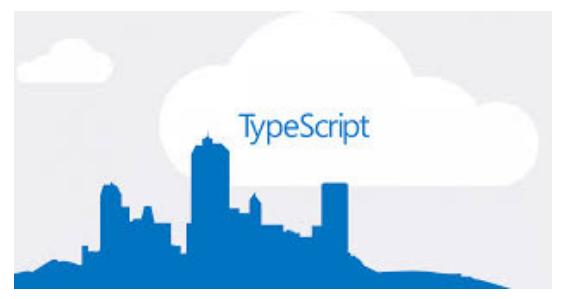
State of the art JavaScript



TypeScript의 역사

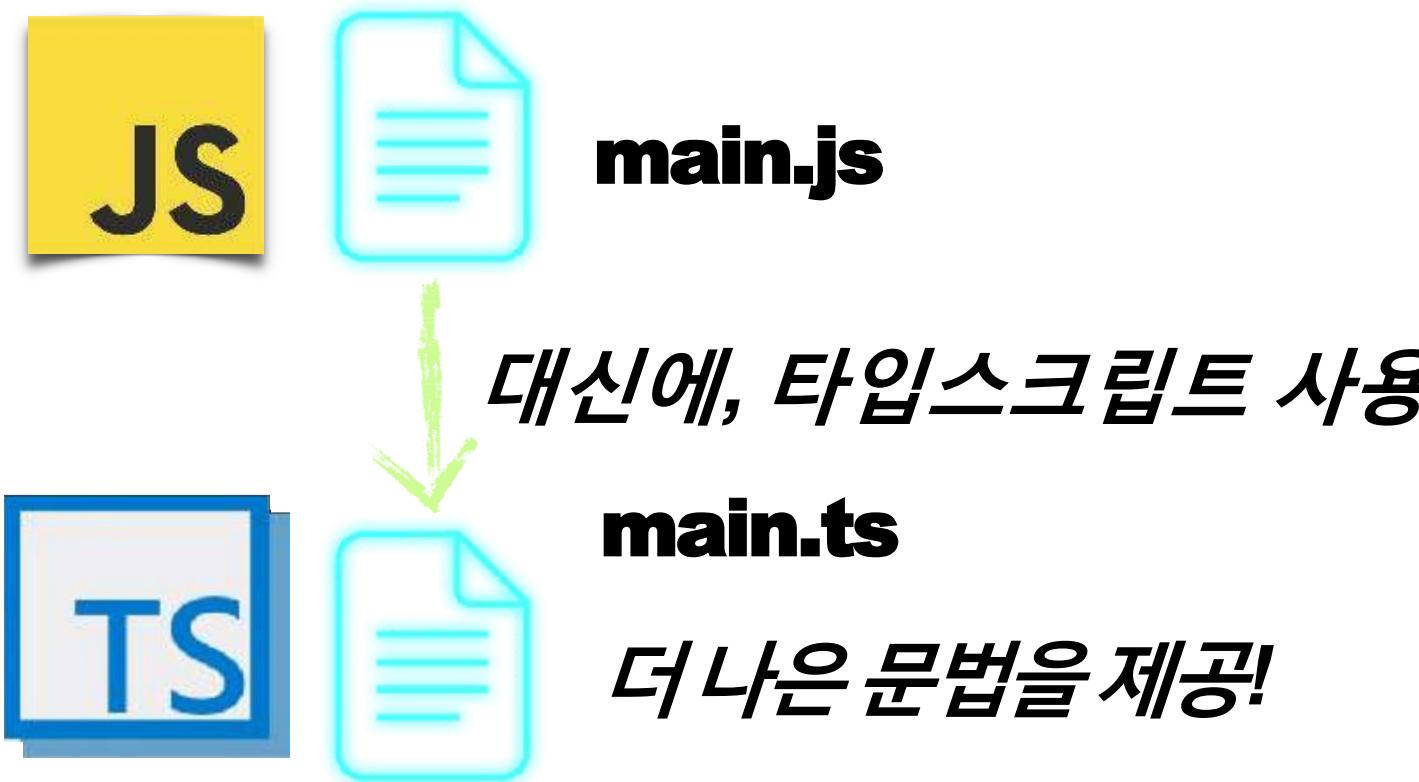
- 2012년 10월 첫 타입스크립트 버전 0.8 발표
- 2013년 6월 18일 타입스크립트 버전 0.9 발표
- 2014년 2월 25일 Visual Studio 2013 빌트인 지원
- 2014년 4월 2일 타입스크립트 1.0 발표
- 2014년 7월 타입스크립트 컴파일러 발표, Github 이전
- 2016년 5월 타입스크립트 2.0 발표
- 2017년 현재 타입스크립트 2.5

TypeScript의 개요

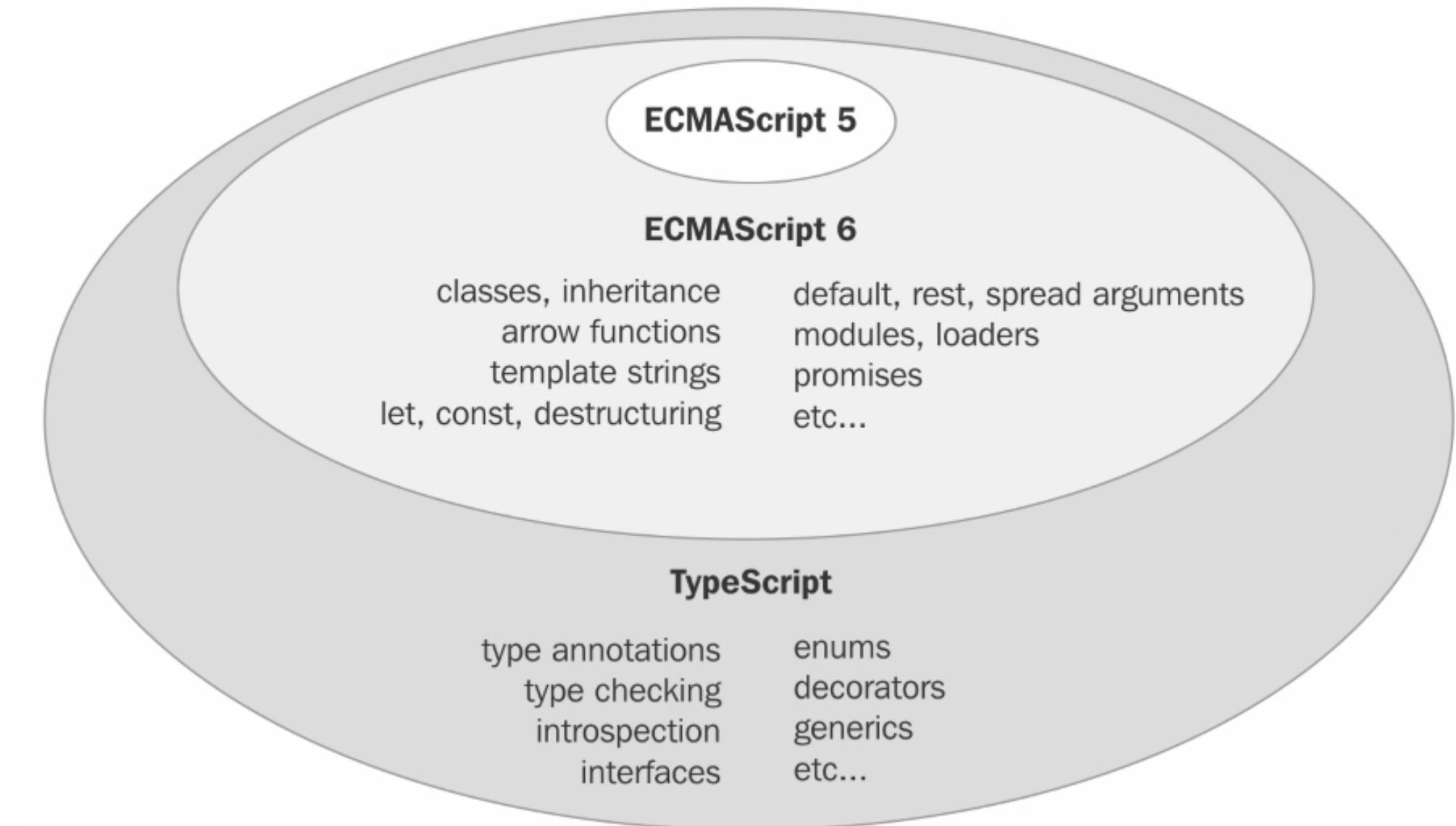


TypeScript는 마이크로소프트(Anders Hejlsberg)에서 개발한 자바스크립트의 확장된 언어이며, ES2015의 기능에 추가적으로 엄격한 타입체크와 객체지향적 기능 등을 추가한 자바스크립트의 Superset이다.

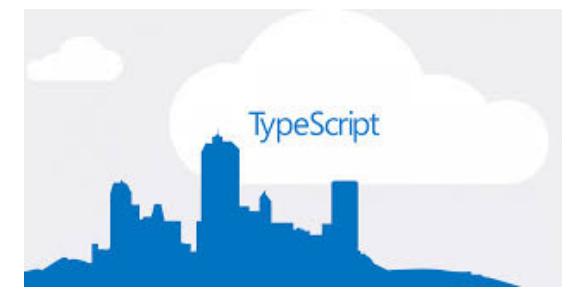
Angular 2 소스는 TypeScript로 개발되었다.



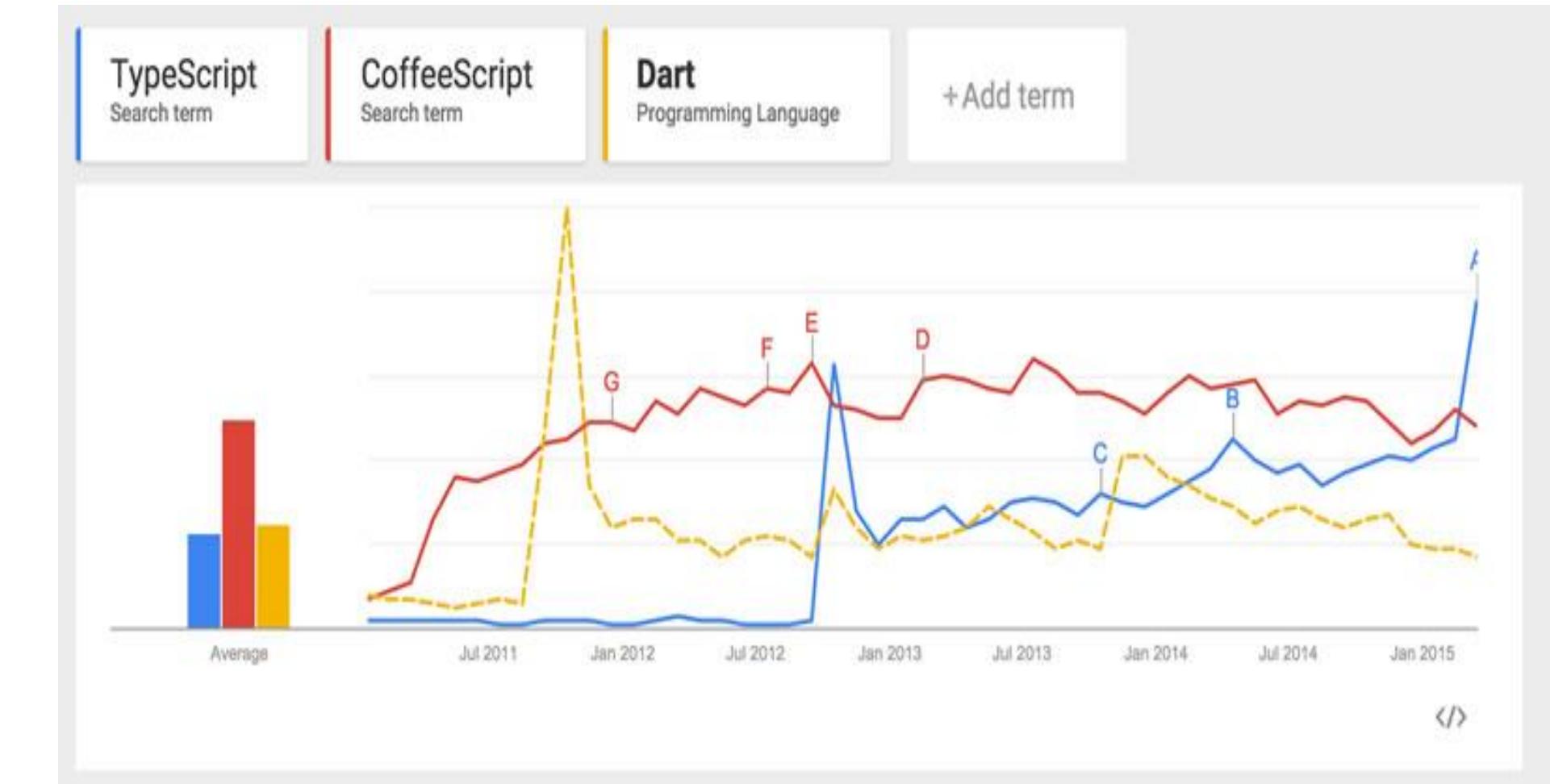
<http://www.typescriptlang.org>



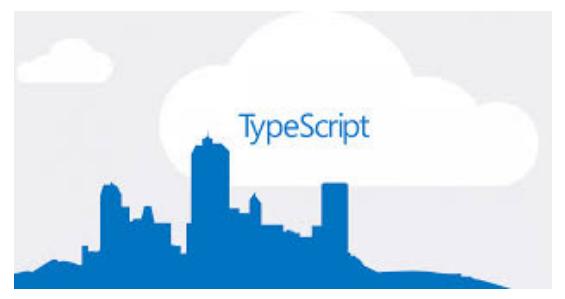
TypeScript의 개요



- 자바스크립트 → **ES2015(ES6)** → **ES2016(ES7)** → **TypeScript**
- 타입스크립트는 자바스크립트의 미래(?)이다.
- 자바스크립트(**ES6, ES7**) 기능에 타입스크립트의 필요한 기능을 추가하면 된다. (새로운 언어가 아님)
 - 클래스 관련기능
 - 정적 타이핑 (**static typing**)



TypeScript의 차별점



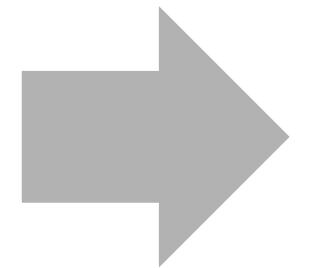
- 명시적인 자료형 선언 가능

JS

```
var a="10";
var b=10;
var sum=a+b;
console.log(sum);
```

결과 : 1010

해석: 20이 출력되길 기대했지만, 자바스크립트의 암묵적(implicit) 형변환은 예측할 수 없는 오류를 만들어 냈.



TS

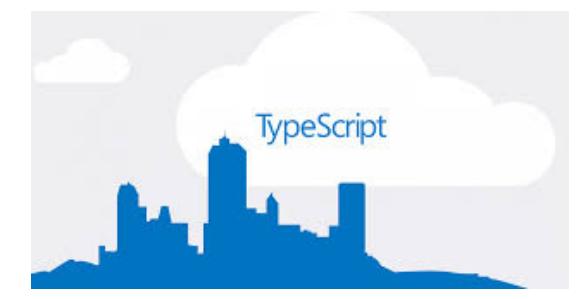
```
function add_error(){
    let a: string = "10";
    let b: number = 10;

    let sum: number = a + b;
    //error!
    console.log(sum);
}

add_error();
```

결과 : 타입이 다른 경우 더하기에
러가 발생함! (오류 가능성 사전제거)

TypeScript의 차별점



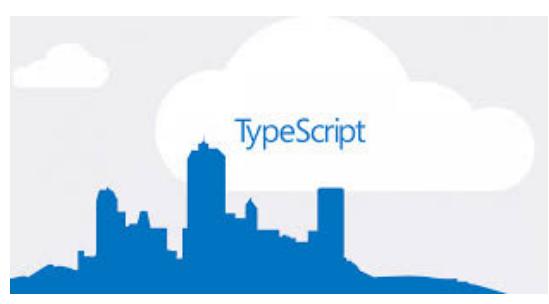
- 명시적인 자료형 선언 가능

```
TS  
function add(){  
    let a: number = 10;  
    let b: number = 10;  
    let sum: number = a + b;  
    console.log(sum);  
}  
  
add();
```

결과 : 20

- 명시적인 자료형 선언으로 가독성이 향상됨
 - 자료형을 명시적으로 정의함으로써 오류를 사전에 감지함
- 예 : 자료형이 다르면 비교나 할당이 불가능함

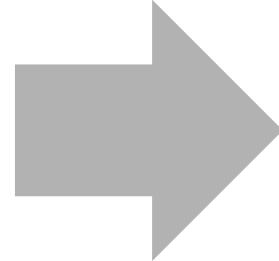
TypeScript의 차별점



- 객체지향 프로그래밍 지원

JS

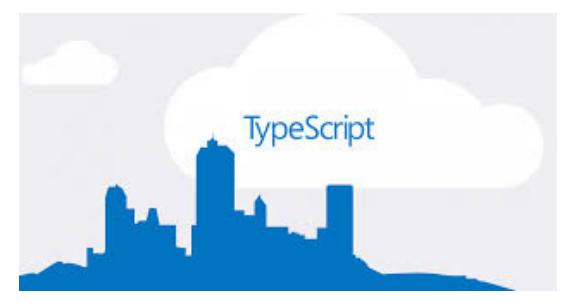
```
var car = (function(){  
    function car(){};  
    car.prototype.getNumTier=function(){};  
    ...  
}());
```



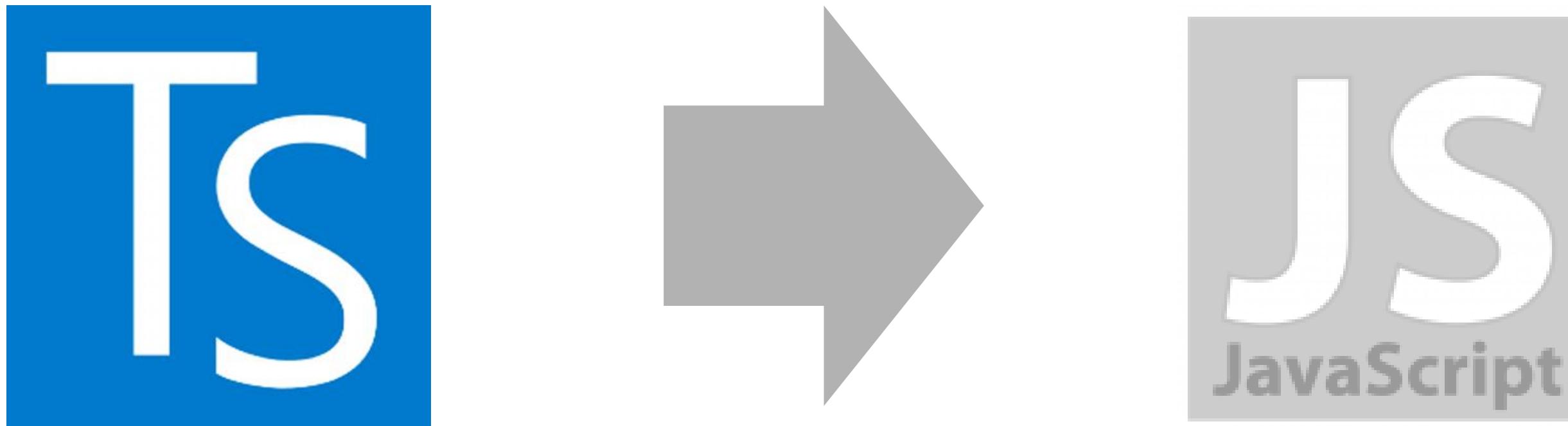
TS

```
class car {  
    numTier: number;  
    constructor(){  
        getNumTier(){  
            ...  
    }  
}
```

트랜스파일러 : tsc

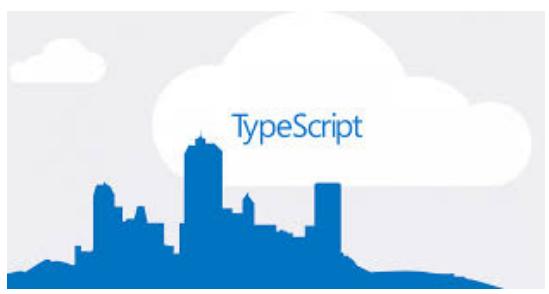


- TSC는 타입스크립트를 자바스크립트로 변환(transpiling)해주는 도구이다.



트랜스파일링

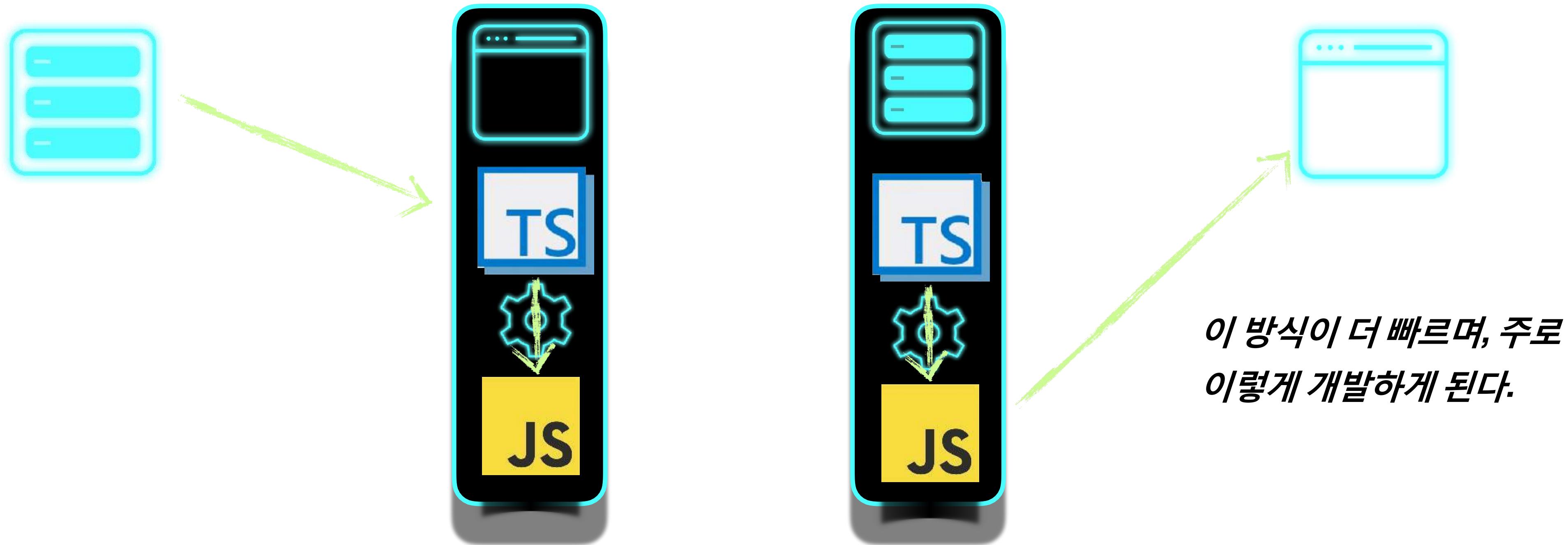
TypeScript : 트랜스파일 위치



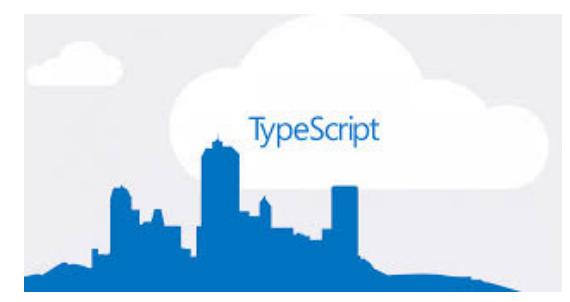
브라우저는 타입스크립트를 해석할 수 없으며, 자바스크립트로 변환하여 브라우저에서 처리되어야 한다. 다음 두 가지 방식이 사용되고 있다.

브라우저에서 자바스크립트로 변환

자바스크립트로 변환 후 브라우저로 로딩



TypeScript : Playground



Screenshot of the TypeScript Playground interface.

The browser title bar shows two tabs: "TypeScript - JavaScript" and "Playground · TypeScript". The address bar displays the URL: www.typescriptlang.org/play/index.html.

The main navigation menu includes: TypeScript, Documentation, Samples, Download, Connect, Playground, and a "Fork me on GitHub" button.

A message at the top states: "TypeScript 2.2 is now available. Download our latest version today!"

The playground interface has two code editors:

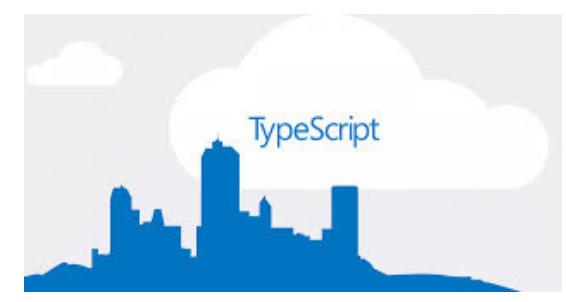
- Left Editor (TypeScript tab):**

```
1 let myAdd2 = function(x: number, y: number)
2
3 let myAdd3: (baseValue:number, increment:number) => number
4   function(x: number, y: number): number
5
6 let myAdd: (x: number, y: number) => number
7   function(x: number, y: number): number
8 console.log(myAdd(10, 20));
9
```
- Right Editor (JavaScript tab):**

```
1 var myAdd2 = function (x, y) { return x + y }
2 var myAdd3 = function (x, y) { return x + y }
3 var myAdd = function (x, y) { return x + y };
4 console.log(myAdd(10, 20));
5
```

Below the editors are buttons for "Select...", "TypeScript", "Share", "Options", "Run", and "JavaScript".

TypeScript : Download



Get TypeScript

Node.js

The command-line TypeScript compiler can be installed as a Node.js package.

INSTALL

```
npm install -g typescript
```

COMPILE

```
tsc helloworld.ts
```

Visual Studio



Visual Studio 2017



Visual Studio 2015



Visual Studio Code

And More...



Sublime Text



Emacs



WebStorm



Atom

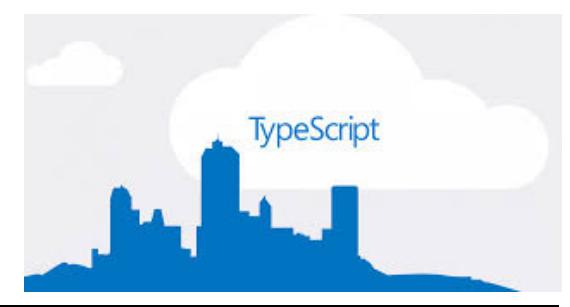


Eclipse



Vim

TypeScript 구성요소



Strongly
Typed

Classes

Interfaces

Generics

Modules

Type
Definitions

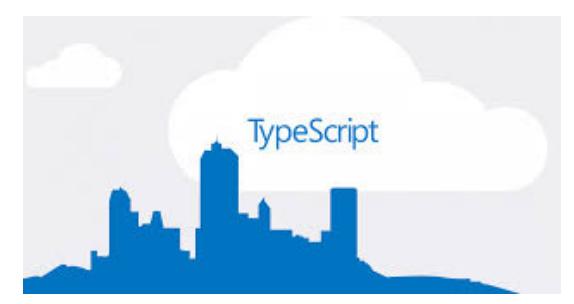
Compiles to
JavaScript

EcmaScript 6
Features

TypeScript : 타입 시스템



- **String**
- **Number**
- **Boolean**
- **Array**
- **Dynamic Typing**
- **Enum**
- **Void**



TypeScript : 클래스

- class 키워드를 이용하여, 클래스 정의

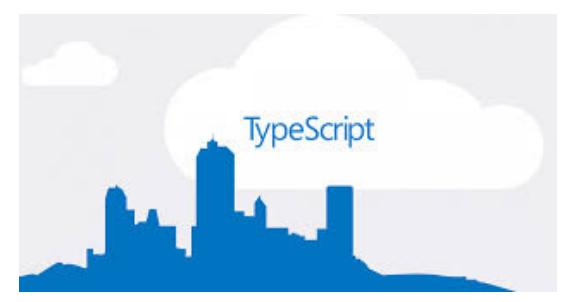
```
class car {  
    numTier: number;  
    carName: string;  
  
    constructor(  
        carName: string, numTier: number){  
            this.carName = carName;  
            this.numTier = numTier;  
    }  
    getNumTier(){  
        return this.numTier;  
    }  
    getCarName(){  
        return this.carName;  
    }  
}
```

```
let myCar = new car("해피카",4);  
console.log(myCar.getCarName()+"의 타이어  
개수는 "+myCar.getNumTier()+"개 입니다.");
```

[결과]

해피카의 타이어 개수는 4개 입니다.

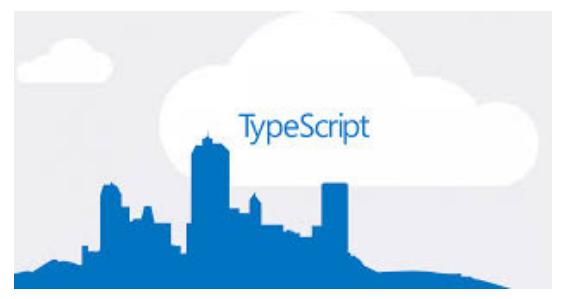
TypeScript : 상속



- extends 키워드를 이용하여 부모 클래스를 상속

```
class HappyCar {  
    numTier: number;  
    carName: string;  
    speed: number;  
  
    constructor(carName: string, numTier: number){  
        this.carName = carName;  
        this.numTier = numTier;  
    }  
  
    setSpeed(speed:number){  
        this.speed=speed;  
    }  
  
    getSpeed(){  
        return this.speed;  
    }  
}
```

```
class bus extends HappyCar {  
    constructor(carName: string, numTier: number) {  
        super(carName,numTier);  
    }  
    setSpeed(speed = 0) {  
        super.setSpeed(speed);  
    }  
}  
  
class truck extends HappyCar {  
    constructor(carName: string, numTier: number) {  
        super(carName,numTier);  
    }  
    setSpeed(speed = 0) {  
        super.setSpeed(speed);  
    }  
}
```



TypeScript : 상속

- 인스턴스 생성 후 테스트

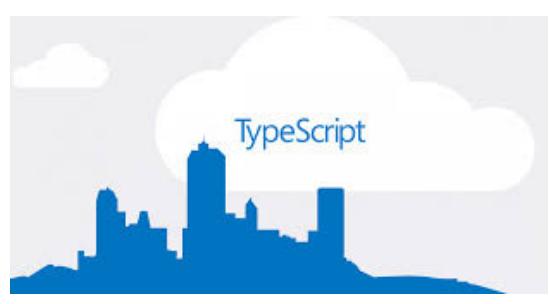
```
let myBus = new bus("myBus",6);
let myTruck: HappyCar = new truck("myTruck",10);

myBus.setSpeed(100);
myTruck.setSpeed(120);
console.log("현재 버스속도 : "+myBus.getSpeed());
console.log("현재 트럭속도 : "+myTruck.getSpeed());
```

[결과]

현재 버스속도 : 100
현재 트럭속도 : 120

TypeScript : 인터페이스

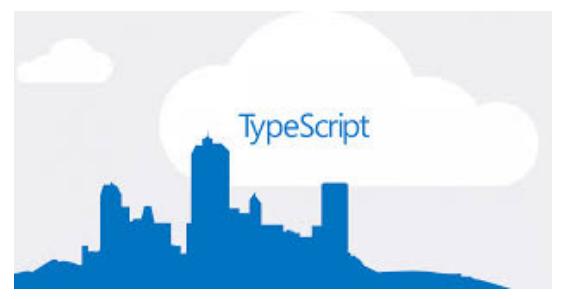


- Interface에 선언된 변수나 메서드에 대한 사용을 강제함

```
interface AddressInterface {  
    addressBookName:string;  
    setBookName(addressBookName: string);  
    getBookName();  
}  
  
class AddressBook implements AddressInterface {  
  
    addressBookName:string;  
    setBookName(addressBookName: string) {  
        this.addressBookName = addressBookName;  
    }  
    getBookName(){  
        return this.addressBookName;  
    }  
    constructor() { }  
}
```

```
let myAddressBook = new AddressBook();  
myAddressBook.setBookName("나의 주소록");  
console.log(myAddressBook.getBookName());
```

[결과]
나의 주소록



TypeScript : module

- ECMAScript 2015에서의 module 개념, export 와 import

Validation.ts

```
export interface StringValidator {  
    isAcceptable(s: string): boolean;  
}
```

ZipCodeValidator.ts

```
import { StringValidator } from "./Validation";  
  
export const numberRegexp = /^[0-9]+$/;  
export class ZipCodeValidator implements StringValidator {  
    isAcceptable(s: string) {  
        return s.length === 5 && numberRegexp.test(s);  
    }  
}
```

Test.ts

```
import { ZipCodeValidator } from "./ZipCodeValidator";  
let myValidator = new ZipCodeValidator();
```

Hello Angular2

Angular2 앱 작성

1. Configuration Files

1. 먼저 프로젝트 directory를 만든다

```
> mkdir angular2-helloworld  
> cd angular2-helloworld
```

2. 프로젝트 configuration 파일들 작성

- **package.json** – Angular2 프로젝트를 위한 정의와 dependency 를 설정한 파일이다. dependency들은 Angular2 에 필요 한 JS 들이 정의 되어 있다.
- **tsconfig.json** – Angular2를 coding 할 때 사용하는 TypeScript의 compiler가 어떻게 JavaScript를 만들 것인가를 정의한 파일 이다.
- **typings.json** – TypeScript compiler가 기본적으로 인식하지 못하는 library들에 대한 추가적인 정의를 제공하는 파일이다.
- **systemjs.config.js** – 처음 Angular2 application이 시작할 때 필요한 packages 를 load 하는 정보가 정의 되어 있다. systemjs.config.js 는 처음 application이 실행 될 때 필요한 패키지 들과 라이브러리 들을 사용 할 수 있게 load 하는 기능을 한다

1.Configuration Files

2.1 package.json : npm 의존성 모듈 관리 파일이다. angular2를 위한 dependency들.

```
package.json
  "scripts": {
    "tsc": "tsc",
    "tsc:w": "tsc -w",
    "typings": "typings"
  },
  "author": "Brient Oh",
  "license": "ISC",
  "dependencies": {
    "@angular/common": "2.0.0",
    "@angular/compiler": "2.0.0",
    "@angular/core": "2.0.0",
    "@angular/forms": "2.0.0",
    "@angular/http": "2.0.0",
    "@angular/platform-browser": "2.0.0",
    "@angular/platform-browser-dynamic": "2.0.0",
    "@angular/router": "3.0.0",
    "@angular/upgrade": "2.0.0",
    "systemjs": "0.19.27",
    "core-js": "^2.4.1",
    "reflect-metadata": "^0.1.3",
    "rxjs": "5.0.0-beta.12",
    "zone.js": "^0.6.23",
    "angular2-in-memory-web-api": "0.0.20",
    "bootstrap": "^3.3.6"
  },
  "devDependencies": {
    "concurrently": "^2.0.0",
    "lite-server": "^2.2.2",
    "typescript": "^2.0.2",
    "typings": "^1.3.2"
  }
}
```

1.Configuration Files

2.2 tsconfig.json : Typescript 컴파일러의 설정파일.

```
tsconfig.json
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "moduleResolution": "node",
    "sourceMap": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "removeComments": false,
    "noImplicitAny": false
  }
}
```

2.3 typings.json : Typescript 컴파일러가 인식할 수 없는 추가적인 라이브러리 파일 정의.

```
typings.json
{
  "globalDependencies": {
    "core-js": "registry:dt/core-js#0.0.0+20160725163759",
    "jasmine": "registry:dt/jasmine#2.2.0+20160621224255",
    "node": "registry:dt/node#6.0.0+20160909174046"
  }
}
```

1.Configuration Files

2.4 system.config.js : 어플리케이션 모듈을 찾기 위한 module loader 정보 제공

```
systemjs.config.js
(function(global) {

    // path
    var paths = {
        'npm': 'node_modules/'
    };

    var map = {
        // our app root
        'app': 'app',

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

        // other libraries
        'angular2-in-memory-web-api': 'npm:angular2-in-memory-web-api',
        'rxjs': 'npm:rxjs'
    };

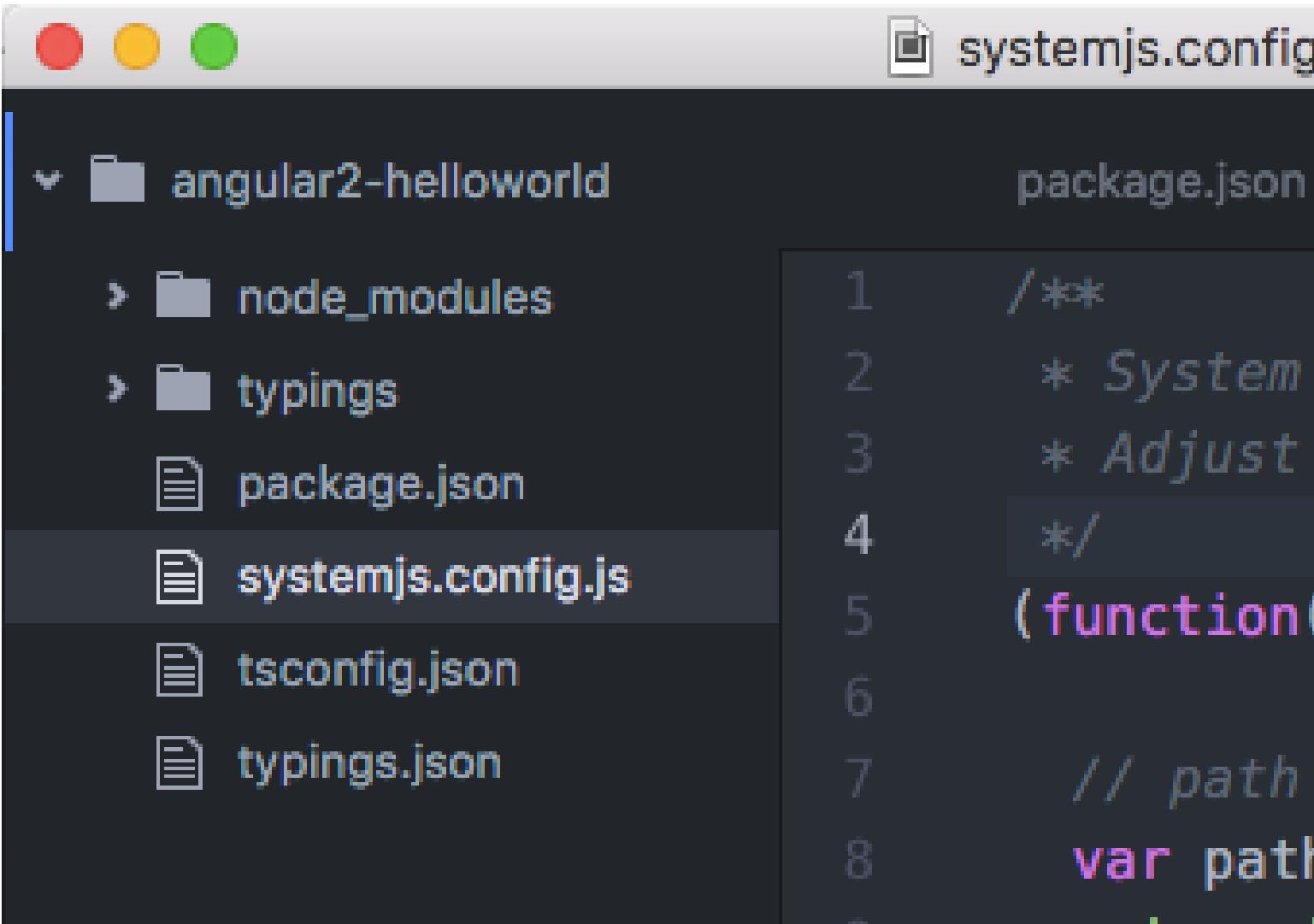
    var packages = {
        'app': { main: './main.js', defaultExtension: 'js' },
        'rxjs': { defaultExtension: 'js' },
        'angular2-in-memory-web-api': { main: './index.js', defaultExtension: 'js' },
    };

    var config = {
```

2.라이브러리 설치

> npm install

package.json과 typings.json 에 정의 되어 있던 dependency들을 다운받기 시작을 한다.
node_modules 와 typings 폴더가 생성되고 그 아래 많은 package 와 library들이 다운로드 됨



```
systemjs.config.js
package.json
angular2-helloworld
  node_modules
  typings
  package.json
  systemjs.config.js
  tsconfig.json
  typings.json
```

```
1  /**
2   * System
3   * Adjust
4   */
5  (function(
6
7    // path
8    var path
```

> root 폴더 아래에 app 폴더를 만든다.

3. App 작성 : index.html

index.html HTML

```
<!DOCTYPE html>
<html>

  <head>
    <!-- All the Angular 2 libraries needed -->
  </head>

  <body>
    <my-app>Loading App ...</my-app>
  </body>

</html>
```

여기에 **Angular2** 앱이 로딩 된다.

주로 앱 이름의 태그가 온다.
예를 들어: <racing-app>

브라우저에 앱이 로딩될 때 까지 다음 메시지가 화면에 출력된다.



3. App 작성 : index.html

index.html

```
<!DOCTYPE html>
<html>

<head>
    <!-- All the Angular 2 libraries needed -->
    <script>
        ...
        System.import('app')
            .catch(function(err) { console.error(err); })
    </script>

</head>

<body>
    <my-app>Loading App . . .</my-app>
</body>

</html>
```

SystemJS는 다른 라이브러리를 **import** 해
줄 수 있는 자바스크립트 라이브러리이다.

: SystemJS 모듈을 사용해서 자바스크립트 파일을 로딩

여기서 app을 로딩한다.



app/main.ts

에러 메시지는 브라우저 콘솔로 출력된다.

3. App 작성 : main.ts

app/main.ts

TypeScript

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

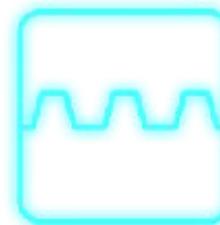
Angular 2 라이브러리

import

ES2015 가능: 모듈을 지원, 함수나 객체 또는 변수를 **import** 하여 사용

Component

Function we will use to create our first Component.



Component는 Angular 2 애플리케이션의 기본 구성 단위이다.
Component는 화면을 구성하는 단위이기도 하다.

3. App 작성 : main.ts

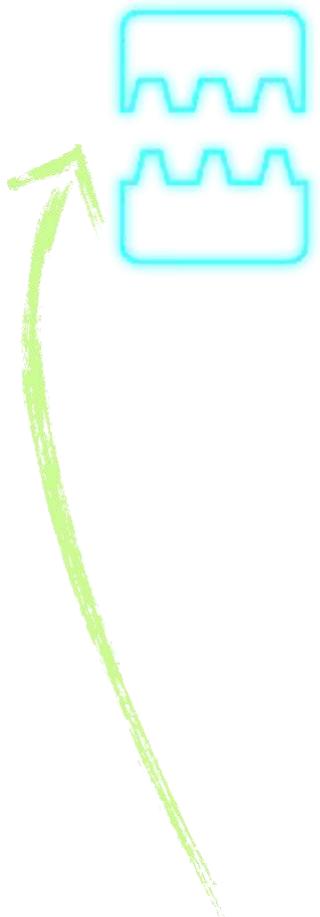
Component Decorating

app/main.ts **TypeScript**

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

Component Decorator 코드가 위치.

class AppComponent { }
```



Decorator

클래스와 한 쌍이 되어 클래스 앞에 선언된다.

Decorator 는 **클래스**를 **Component**로 바꿔준다.

3. App 작성 : main.ts

app/main.ts

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

@Component({
  selector: 'my-app',
  template: '<h1>Hello {{name}} </h1>'
})
class AppComponent {
  name='Angular';
}
```



TypeScript

index.html

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

메타데이터(metadata)
라고도 함

@Component

Component 클래스와 한 쌍을 이룬다.

Decorator는 타입스크립트와 EcmaScript7의 기능

selector

Component가 로딩 될 HTML Element

template

selector 내부에 로딩될 컨텐츠

3. App 작성 : main.ts

Root Angular Module 선언

: module은 Angular에서 어플리케이션을 구성하는 방법이다.

: 모든 Angular app은 반드시 앱을 실행 시키기 위해 “root module”을 가져야 한다.

app/main.ts **TypeScript**

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

@Component({
  selector: 'my-app',
  template: '<h1>Hello {{name}}</h1>'
})

class AppComponent {
  name='Angular';
}

@NgModule({
  declarations: [ AppComponent ]
})
class AppModule { }
```

module 내의 모든 컴포넌트
목록이 위치한다.

3. App 작성 : main.ts

Application 실행을 위한 의존성 추가

app/main.ts

TypeScript

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

...

BrowserModule

웹브라우저에서 실행하기 위한 모듈.

platformBrowserDynamic

웹브라우저 렌더링에 필요한 모듈.

위 두 가지는 **Angular app 실행에 모두 필요**

3. App 작성 : main.ts

Component 시작하기

app/main.ts

TypeScript

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

...

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

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

브라우저에서 앱을 로딩하기 위한 모듈

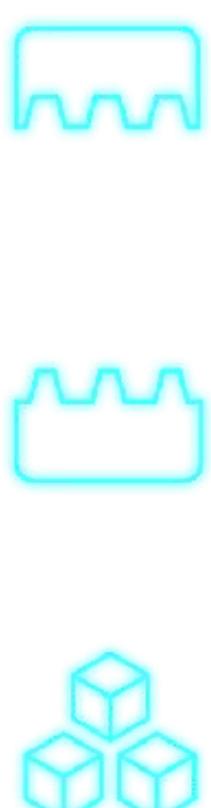
루트 컴포넌트

AppModule로 어플리케이션 로딩



4. App 작성 완료 및 실행

> npm start

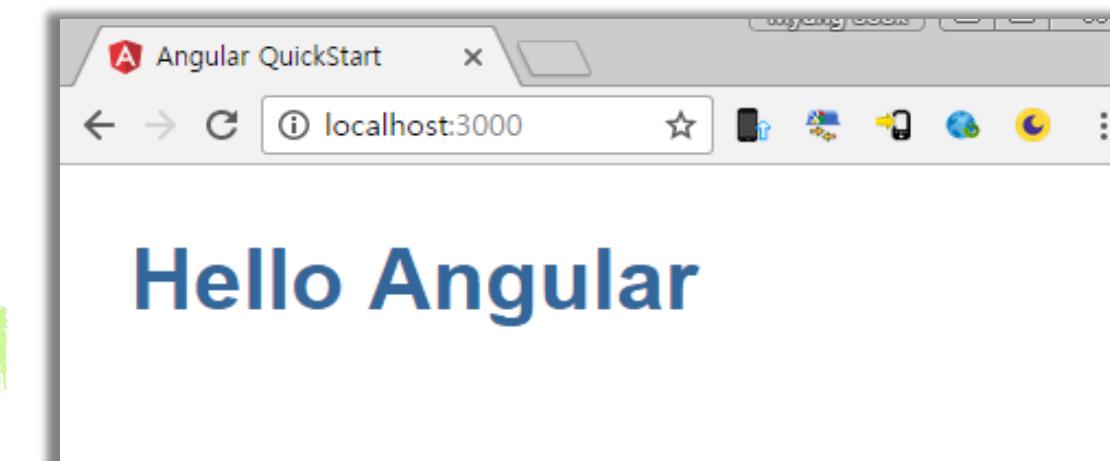


app/main.ts **TypeScript**

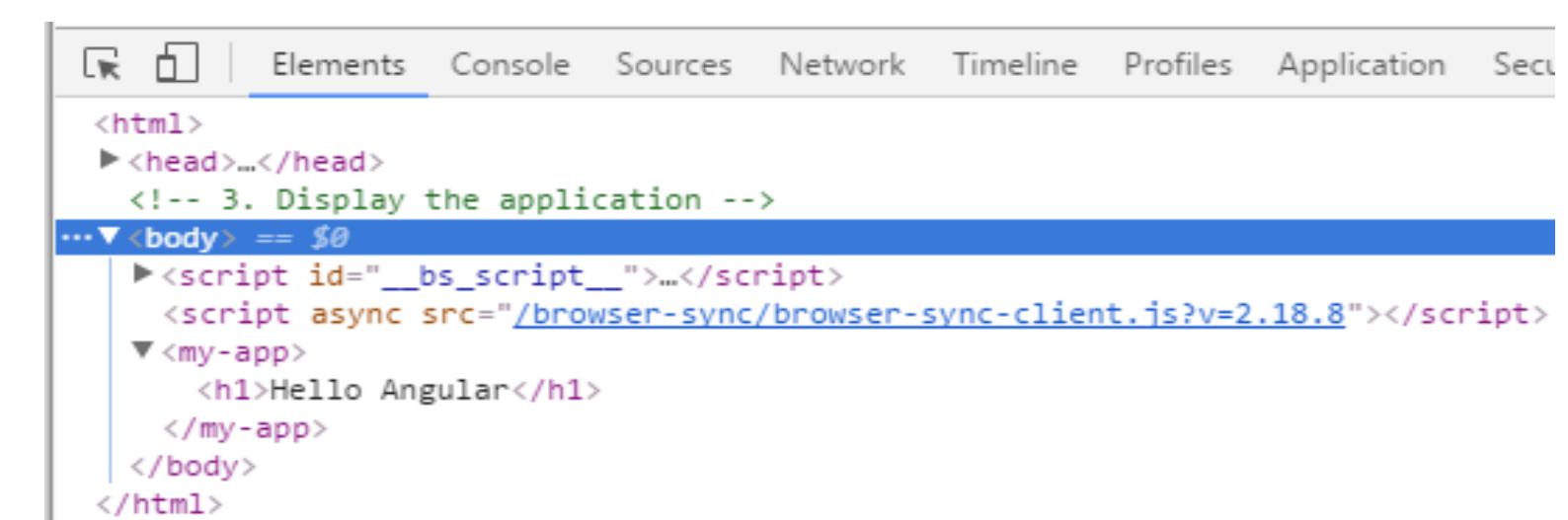
```
...  
@Component({  
  selector: 'my-app',  
  template: '<h1>Hello {{name}} </h1>'  
})  
class AppComponent {  
  name='Angular';  
}  
  
@NgModule({  
  declarations: [ AppComponent ],  
  imports: [ BrowserModule ],  
  bootstrap: [ AppComponent ]  
})  
class AppModule { }  
  
platformBrowserDynamic()  
  .bootstrapModule(AppModule);
```

index.html

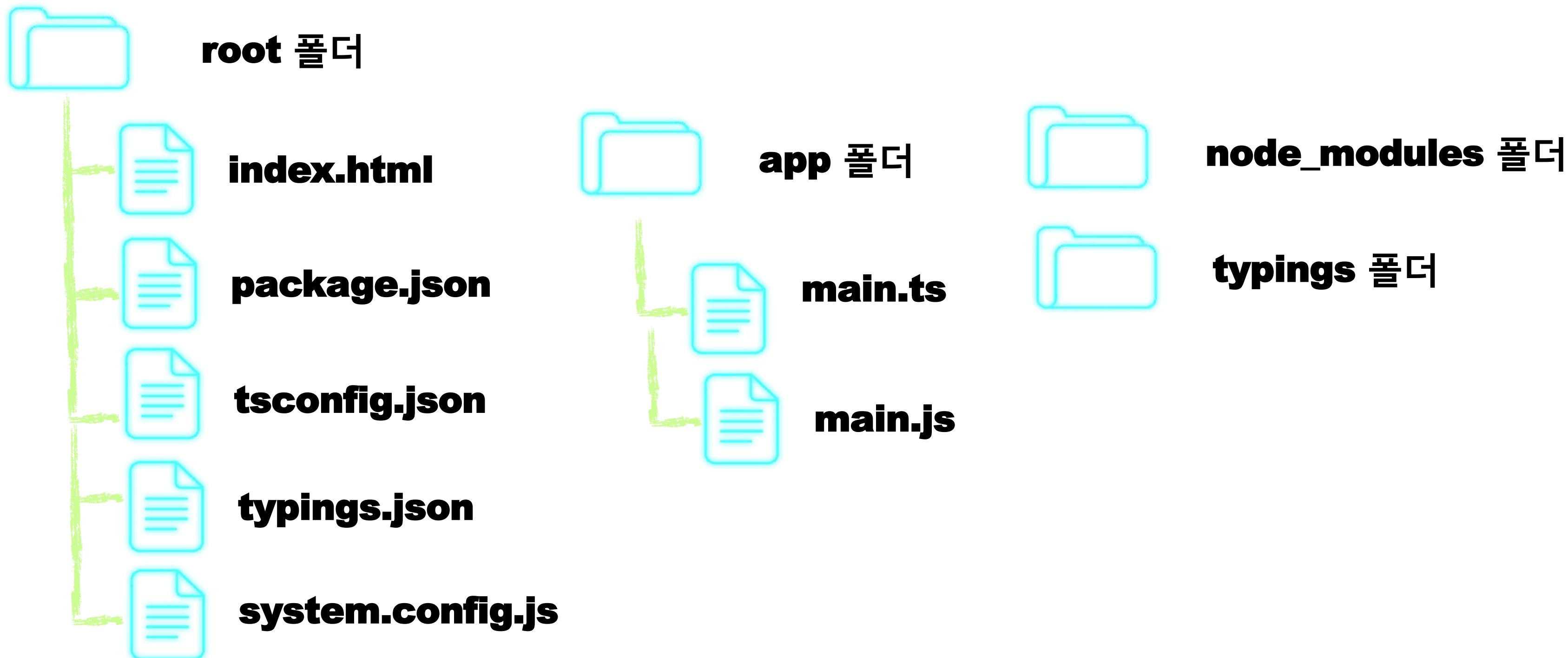
```
<body>  
  <my-app>Loading App ...</my-app>  
</body>
```



Viewing the Source



5. App의 구조



지금까지의 내용

- Angular는 다이나믹 웹 어플리케이션을 위한 프레임워크이다.
- Angular 개발을 위해 Typescript를 사용한다. Typescript 는 Javascript로 변환 된다.
- Angular 앱의 기본 building block은 Component이다.
- HTML 내에서 Component 를 보여주기 위해서는 사용자 정의 HTML 태그를 사용해야 한다.
- Typescript 클래스를 Component 로 바꿔주는 것은 Decorator이다.

◎ 학습지식 개요/요점

AngularJS 의 컴포넌트와 어플리케이션 작성 방법을 실습합니다.

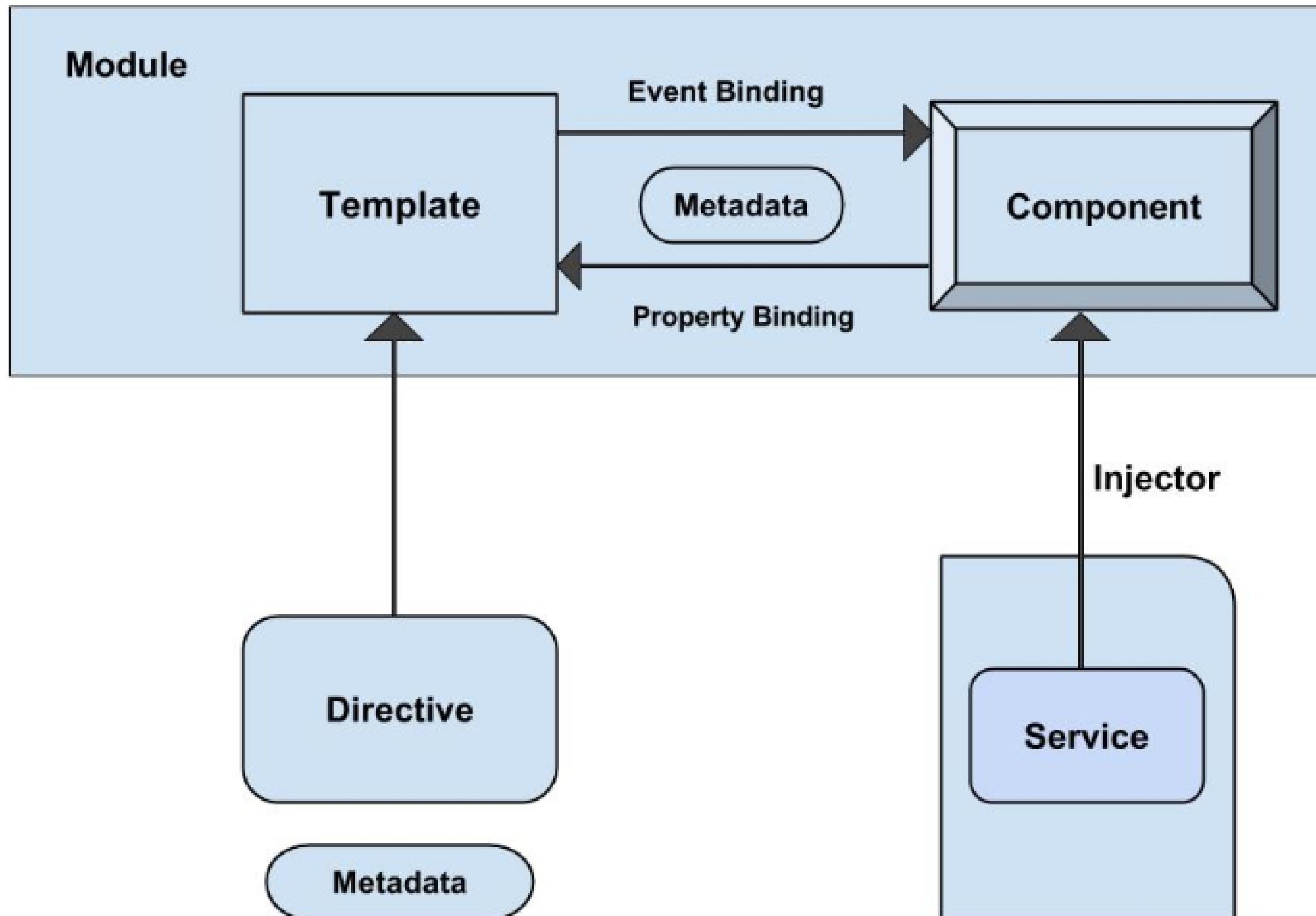
◎ 실습 예제 및 수행가이드

- Component의 개념
- Interpolation
- Directive의 개념과 컴포넌트
- Pipe 사용과 Custom Pipe
- Data Model을 활용
- Data Binding의 종류와 Event 처리

Angular2 Component

Component 개념과
Interpolation, {{name}}

Angular[®] Architecture



1. Component : 컴포넌트 중심의 개발

- Component 정의

기능 명세에 의한 개발

명세 따른 배포, 조립 가능한 독립 구성단위

컴포넌트는 인터페이스만을 통해서 접근 (예 : WSDL)

- Component 중심 개발의 특징

관심사 분리의 개발 방법으로 컴포넌트간 연결이 느슨하다(loosely coupled).

컴포넌트 재활용에 초점을 맞춘다.

1.Component : Angular2의 컴포넌트

- Front-End에서의 컴포넌트

custom element(HTML5)로 볼 수 있다

Angular2에서 컴포넌트는 특정 Element를 의미한다.

사용예 : <my-component></my-component>

- Angular 1의 컴포넌트

directives, controllers, Scope에 의해 구현

컴포넌트에 해당하는 directive는 Angular1의 구성요소의 일부였다.

1.Component : Angular2의 컴포넌트

- Component 중심의 Angular2

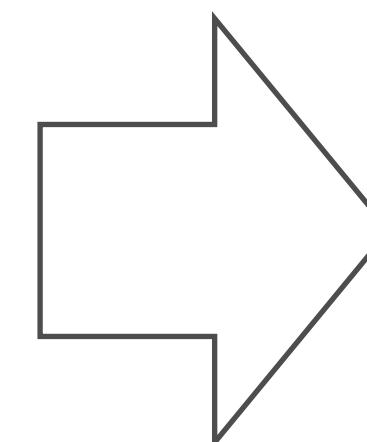
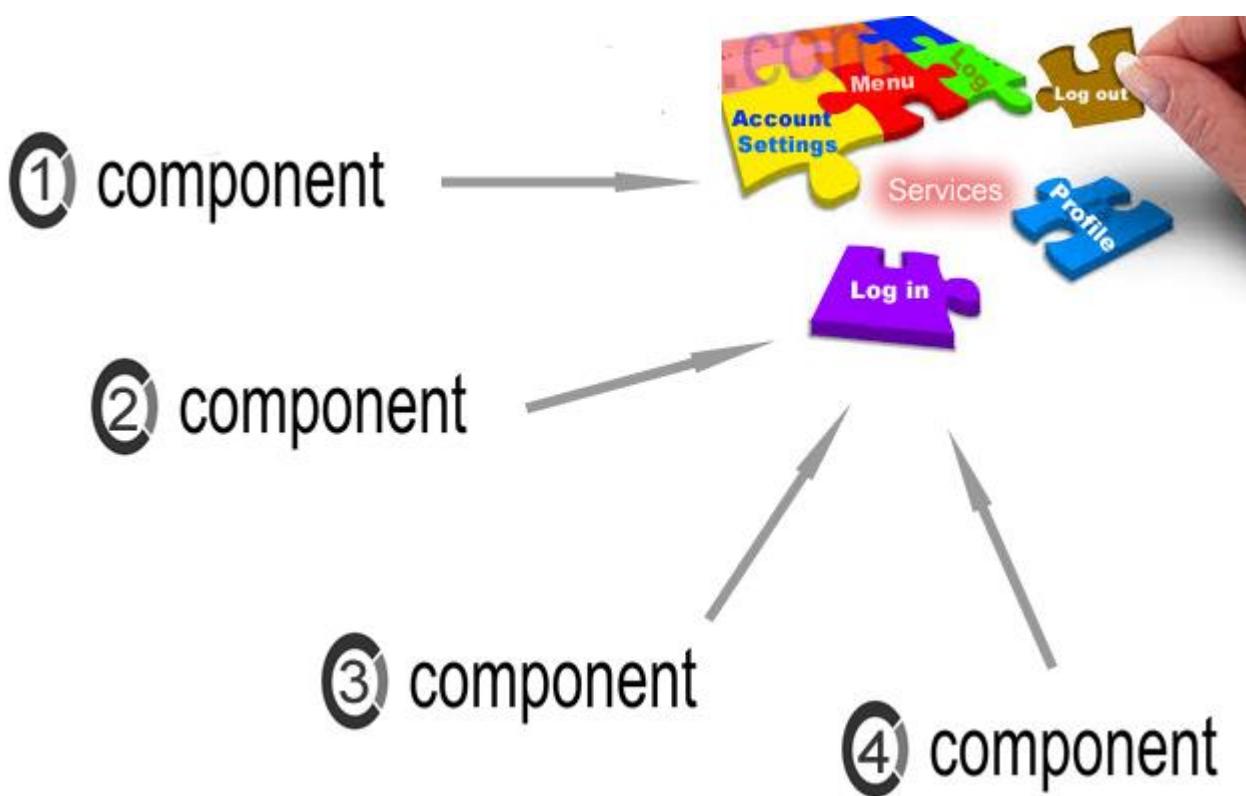
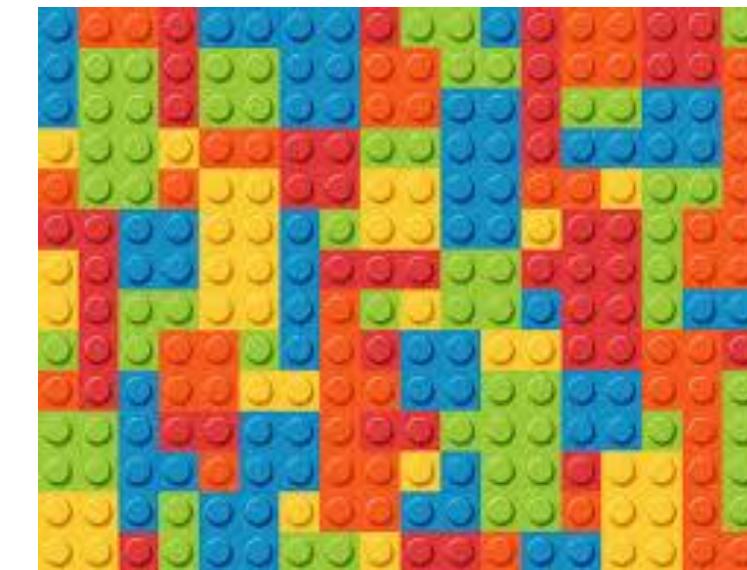
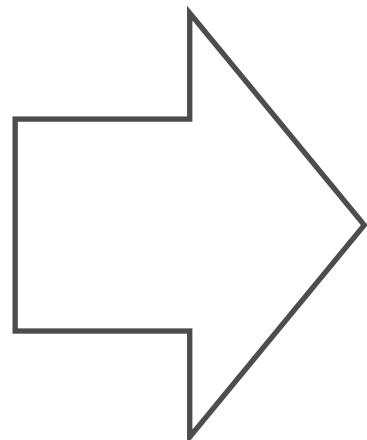
Angular2는 컴포넌트 중심으로 개발을 진행한다.

컴포넌트는 Template과 Logic을 포함한다.

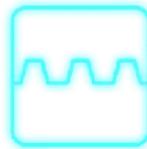
component는 하나의 독립적인 모듈결합을 가진다.

component는 다른 component나 service를 의존주입(Dependency Injection) 받을 수 있다.

1.Component : Angular2를 활용한 컴포넌트 중심의 개발



1.Component : Component 구성 예



component는 angular2 app의 구성 요소(Building blocks)

각 component는 서로 포함될 수 있다.



각 **component**는 다음을 포함한다.

class file

html file

css file

1.Component : Ultra Racing App 구현

Angular QuickStart

localhost:3000

Ultra Racing

There are 9 total parts in stock



SUPER TIRES
These tires are the very best
5 in Stock

\$4.99 0 - 0 +



REINFORCED SHOCKS
Shocks made from kryptonite
4 in Stock

\$9.99 0 - 0 +

Component Tree Router Tree NgModules

AppComponent

CarPart Component

 input

 input

 input

2. Interpolation :

컴포넌트 클래스에서 **HTML**에게 데이터를 어떻게 전달할 수 있을까?

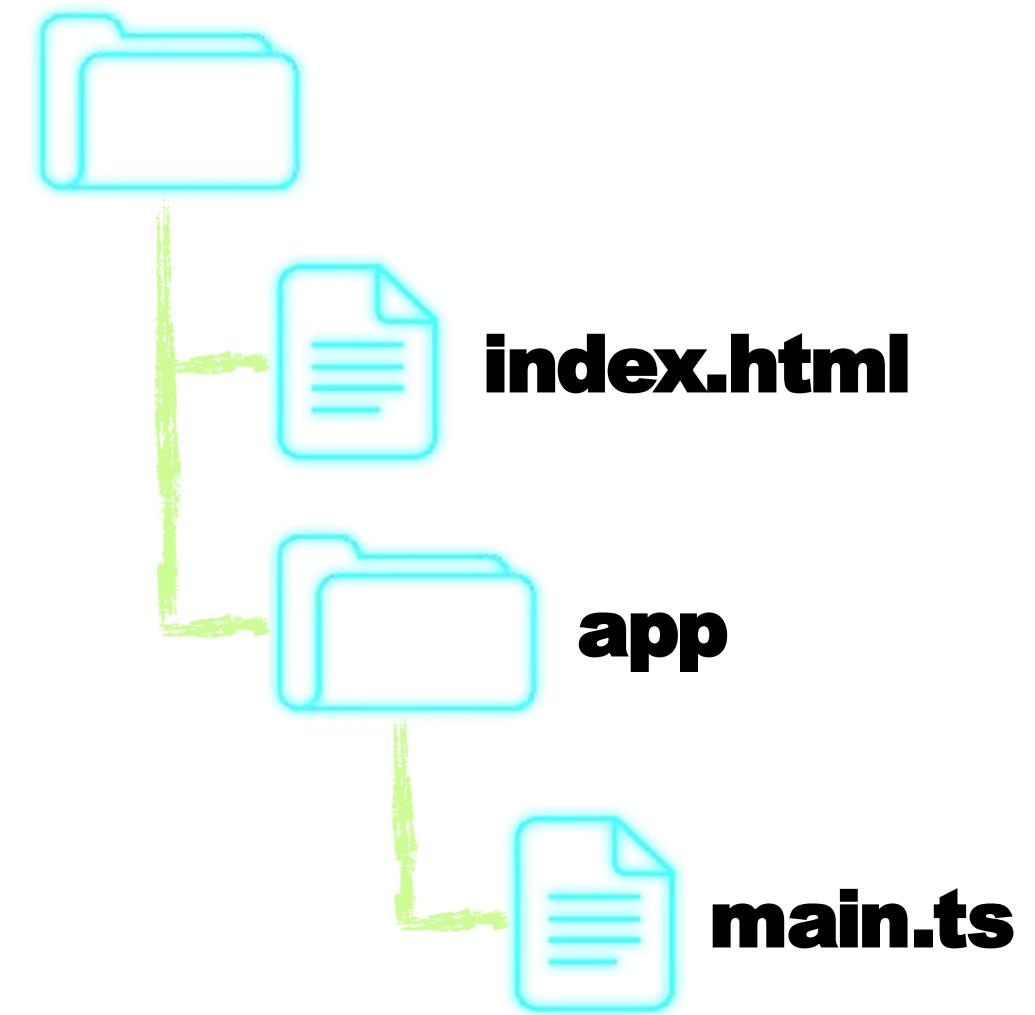
app/main.ts

TypeScript

```
...  
  
@Component ({  
  selector: 'my-app',  
  template: '<h1>???</h1>'  
})  
class AppComponent {  
  title = 'Ultra Racing';  
}  
...  

```

어플리케이션 구조:



TypeScript 클래스 내부에서는 변수를 선언하기 위해
var이나 let을 사용하지 않는다.

2. Interpolation : {{ }}

이중 브레이스 '{{ }}' 사용하여 컴포넌트 속성을 로딩 — **interpolation** 이라 함.

app/main.ts

TypeScript

```
...  
@Component({  
  selector: 'my-app',  
  template: '<h1>{{title}}</h1>'  
})  
class AppComponent {  
  title = 'Ultra Racing';  
}  
...
```



2. Interpolation : {{ }}

자바스크립트 객체를 화면에 출력하는 방법?

app/main.ts

TypeScript

```
...
@Component({
  selector: 'my-app',
  template: '<h1>{{title}}</h1>'
})
class AppComponent {
  title = 'Ultra Racing';
  carPart = {
    "id": 1,
    "name": "Super Tires",
    "description": "These tires are the very best",
    "inStock": 5
  };
}

...
```



2. Interpolation : Back Tick (`)을 이용한 템플리팅

외따옴표(')를 대신해서 백틱을 사용.

app/main.ts

TypeScript

```
...
@Component({
  selector: 'my-app',
  template: `<h1>{{title}}</h1>
              <h2>{{carPart.name}}</h2>
              <p>{{carPart.description}}</p>
              <p>{{carPart.inStock}} in Stock</p>`
})
class AppComponent {
  title = 'Ultra Racing';
  carPart = {
    "id": 1,
    "name": "Super Tires",
    "description": "These tires are the very best",
    "inStock": 5
  };
}
```

Single Quote



Back Tick



백틱은 멀티라인을 허용.

ES2015의 또 다른 기능이다.

Ultra Racing

Super Tires

These tires are the very best

5 in Stock

Structural Directives

`*ngIf, *ngFor, *ngSwitch`

Structural Directives

- 디렉티브(앵귤러에서) HTML을 dynamic하게 표현하는 방법
- 컴포넌트도 디렉티브의 일종 (템플리트를 포함하고 있는 디렉티브)
- 구조적 디렉티브(Structural Directive)

`*ngFor`

배열 Element 반복.

`*ngIf,`

컨텐츠를 조건에 맞게 보여줌.

`*ngSwitch`

Structural Directives : *ngFor

main.ts

TypeScript

```
...
@Component({
  selector: 'my-app',
  template: `<h1>{{title}}</h1>
<ul>
  <li *ngFor="let carPart of carParts">
    <h2>{{carPart.name}}</h2>
    <p>{{carPart.description}}</p>
    <p>{{carPart.inStock}} in Stock</p>
  </li>
</ul>`})
class AppComponent {
  title = 'Ultra Racing';
  carParts = [ {
    "id": 1,
    "name": "Super Tires",
    "description": "These tires are the very best",
    "inStock": 5
  }, { ... }, { ... } ];
```



*ngFor : structural directive.

carPart : 지역변수.

carParts : 배열로써 엘레먼트를 반복.

세 번의 루프를 수행:

각 엘레먼트: carPart.

Ultra Racing

• Super Tires

These tires are the very best

5 in Stock

• Reinforced Shocks

Shocks made from kryptonite

4 in Stock

• Padded Seats

Super soft seats for a smooth ride

0 in Stock

Structural Directives : *ngIf

부품의 재고가 없을 때 “Out of Stock”로 표시되어야 함

main.ts

TypeScript

```
...
@Component ({
  selector: 'my-app',
  template: `<h1>{{title}}</h1>
<ul>
  <li *ngFor="let carPart of carParts">
    <h2>{{carPart.name}}</h2>
    <p>{{carPart.description}}</p>
    <p *ngIf="carPart.inStock > 0">{{carPart.inStock}} in Stock</p>
    <p *ngIf="carPart.inStock === 0">Out of Stock</p>
  </li>
</ul>`)}
class AppComponent {
  ...
}
```



조건이 참이면 표시.

Ultra Racing

• Super Tires

These tires are the very best

5 in Stock

• Reinforced Shocks

Shocks made from kryptonite

4 in Stock

• Padded Seats

Super soft seats for a smooth ride

Out of Stock

Pipes & Methods

Pipe의 사용,
method 작성

Pipe

- Template에서 출력을 변경하기 위해서 pipe를 사용할 수 있다.
- Pipe는 데이터를 입력 받아 특정 포맷으로 출력해 주는 기능.

API Reference (v4.0.0)

TYPE: [P Pipe](#)

STATUS: ALL

Filter

common

[P AsyncPipe](#)

[P DecimalPipe](#)

[P JsonPipe](#)

[P SlicePipe](#)

[P CurrencyPipe](#)

[P I18nPluralPipe](#)

[P LowerCasePipe](#)

[P TitleCasePipe](#)

[P DatePipe](#)

[P I18nSelectPipe](#)

[P PercentPipe](#)

[P UpperCasePipe](#)

Pipe :UpperCasePipe

부품 이름을 모두 대문자로 변환하는 기능 구현

main.ts

TypeScript

```
...
@Component ({
  selector: 'my-app',
  template: `<h1>{{title}}</h1>
    <ul>
      <li *ngFor="let carPart of carParts">
        <h2>{{carPart.name | uppercase}}</h2>
        <p>{{carPart.description}}</p>
        <p *ngIf="carPart.inStock > 0">{{carPart.inStock}} in Stock</p>
        <p *ngIf="carPart.inStock === 0">Out of Stock</p>
      </li>
    </ul>`}

  class AppComponent {
    ...
  }
}
```

Ultra Racing

SUPER TIRES

These tires are the very best

5 in Stock

REINFORCED SHOCKS

Shocks made from kryptonite

4 in Stock

PADDED SEATS

Super soft seats for a smooth ride

Out of Stock

유닉스의 파이프 개념
과 유사

Pipe : Currency Pipe

통화 포맷을 위해, ISO 4217 통화 코드 사용.

Ex: USD, EUR, or KRW

main.ts

TypeScript

```
...
@Component ({
  selector: 'my-app',
  template: `<h1>{{title}}</h1>
    <ul>
      <li *ngFor="let carPart of carParts" >
        <h2>{{carPart.name | uppercase }}</h2>
        <p>{{carPart.description}}</p>
        <p>{{carPart.price | currency:'EUR'}}</p>
        <p *ngIf="carPart.inStock > 0">{{carPart.inStock}} in Stock</p>
        <p *ngIf="carPart.inStock === 0">Out of Stock</p>
      </li>
    <ul>
  ) )
class AppComponent {
  ...
}
```



Ultra Racing

• SUPER TIRES

These tires are the very best

EUR4.99

5 in Stock

• REINFORCED SHOCKS

Shocks made from kryptonite

EUR9.99

4 in Stock

• PADDED SEATS

Super soft seats for a smooth ride

EUR24.99

Out of Stock

Pipe : Currency Pipe

두 번째 파라미터를 통해 통화 기호를 사용할 수 있다.

main.ts

```
...
@Component({
  selector: 'my-app',
  template: `<h1>{{title}}</h1>


- <h2>{{carPart.name | uppercase}}</h2>
      <p>{{carPart.description}}</p>
      <p>{{carPart.price | currency:'EUR':true}}</p>
      <p *ngIf="carPart.inStock > 0">{{carPart.inStock}} in Stock</p>
      <p *ngIf="carPart.inStock === 0">Out of Stock</p>
    </li>


`)
class AppComponent {
  ...
}
```

TypeScript

파라미터 구분자로
콜론(:)을 사용.

Ultra Racing

• SUPER TIRES

These tires are the very best

€4.99

5 in Stock

• REINFORCED SHOCKS

Shocks made from kryptonite

€9.99

4 in Stock

• PADDED SEATS

Super soft seats for a smooth ride

€24.99

Out of Stock

Pipe : 여러가지 pipe

lowercase

소문자로 변환

사용자 정의 파이프 작성 가능!

date

원하는 형태의 날짜 포맷으로 변환

number

숫자 관련 포맷

decimal

큰 숫자 관련 포맷

replace

문자열에서 특정 문자로 치환

slice

배열이나 문자열을 자르는 기능

json

입력을 JSON 으로된 스트링으로 바꿔준다.

디버깅 용으로 사용

Pipe : 사용자 정의 Pipe 작성

- @Pipe - 메타데이터를 사용
- PipeTransform 인터페이스의 transform 메소드를 구현
- @angular/core 모듈의 Pipe를 import

```
import { Pipe, PipeTransform } from '@angular/core';
/*
 * Raise the value exponentially
 * Takes an exponent argument that defaults to 1.
 * Usage:
 *   value | exponentialStrength:exponent
 * Example:
 *   {{ 2 | exponentialStrength:10}}
 *   formats to: 1024
 */
@Pipe({name: 'exponentialStrength'})
export class ExponentialStrengthPipe implements PipeTransform {
  transform(value: number, exponent: string): number {
    let exp = parseFloat(exponent);
    return Math.pow(value, isNaN(exp) ? 1 : exp);
  }
}
```

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

method 사용 : 전체 부품 갯수 조회

재고 부품의 전체 갯수를 출력



main.ts

```
...
@Component ({
  selector: 'my-app',
  template: `<h1>{{title}}</h1>
    <ul>
      <li *ngFor="let carPart of carParts">
        ...
      </li>
    </ul>
  </div>`}
```

TypeScript



Ultra Racing

- **SUPER TIRES**

These tires are the very best

€4.99

5 in Stock

- **REINFORCED SHOCKS**

Shocks made from kryptonite

€9.99

4 in Stock

- **PADDED SEATS**

Super soft seats for a smooth ride

€24.99

Out of Stock

method 사용 : Template 설정

클래스에 메소드를 추가하여 간단하게 10을 리턴하는 코드

main.ts

TypeScript



```
...
@Component ({
  selector: 'my-app',
  template: `<h1>{{title}}</h1>
    <p>There are {{totalCarParts()}} total parts in stock.</p>
  ...
})
class AppComponent {
  title = 'Ultra Racing';
  carParts = [...];

  totalCarParts() { ←
    return 10;
  }
}
```



Ultra Racing

There are 10 total parts in stock.

- SUPER TIRES

These tires are the very best

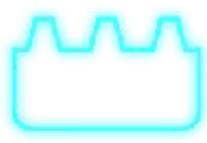
€4.99

타입스크립트 클래스에서는
“function,” 키워드를 사용하
지 않는다.

ES2015 기능이 TypeScript에서
사용됨

method 사용 : 메소드 구현

ES2015의 기능인 for of 루프 사용.



main.ts

```
class AppComponent {  
  title = 'Ultra Racing';  
  carParts = [...];  
  
  totalCarParts() {  
    let sum = 0;  
  
    for (let carPart of this.carParts) {  
      sum += carPart.inStock;  
    }  
  
    return sum;  
  }  
}
```

ES2015

TypeScript

Ultra Racing

There are 9 total parts in stock.

- SUPER TIRES

These tires are the very best

€4.99

5 in Stock

- REINFORCED SHOCKS

Shocks made from kryptonite

€9.99

4 in Stock

- PADDED SEATS

Super soft seats for a smooth ride

€24.99

Out of Stock

method 사용 : 합계 단순화 코드 (lambda 식)

```
totalCarParts() {  
    let sum = 0;  
  
    for (let carPart of this.carParts) {  
        sum += carPart.inStock;  
    }  
  
    return sum;  
}
```

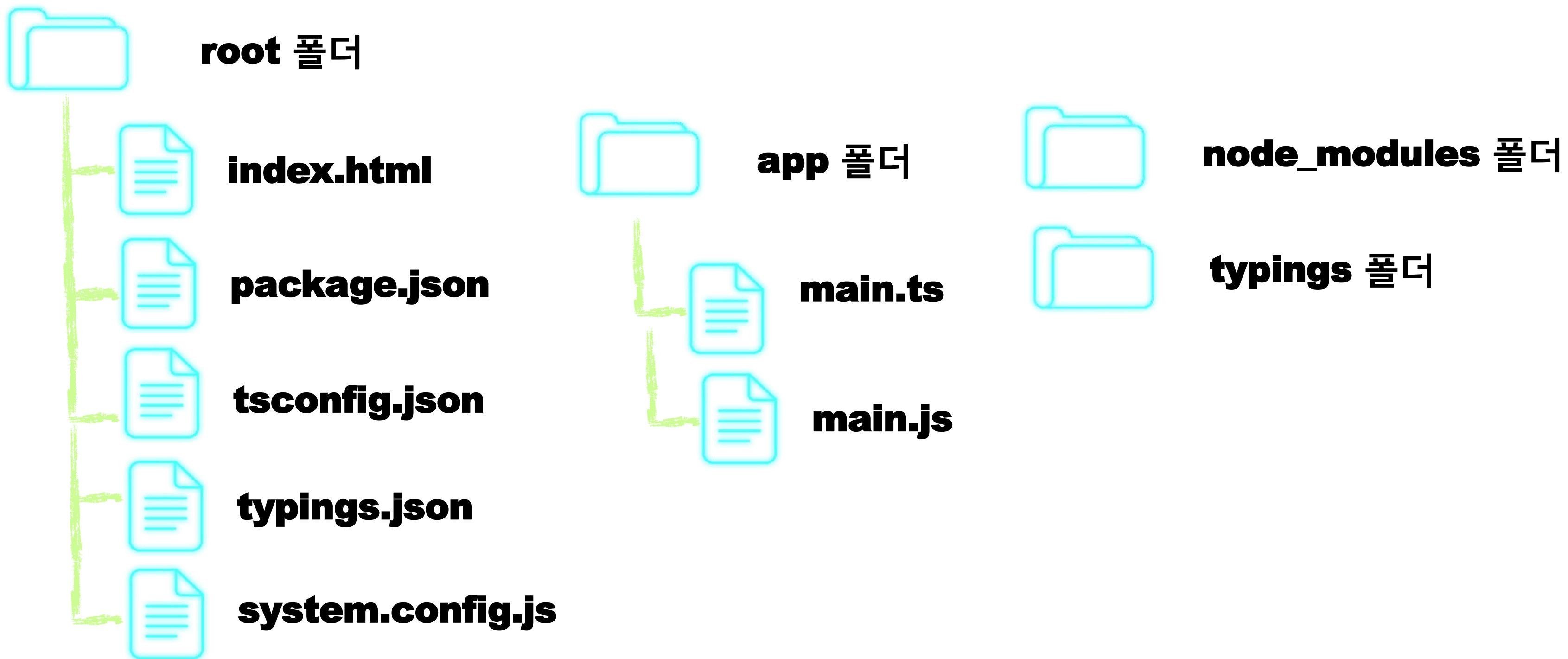
```
totalCarParts() {  
    return this.carParts.reduce(function(prev, current) { return prev + current.inStock; }, 0 );  
}
```

```
totalCarParts() {  
    return this.carParts.reduce((prev, current) => prev + current.inStock, 0 );  
}
```

 화살표 함수(ES2015)

Pipe와 method의 사용

- 템플리트에서 출력을 변경하기 위해서 pipe를 사용할 수 있다.
- 컴포넌트에서 메소드를 사용하여 로직을 수행할 수 있다.



Splitting to Two Components

Component 분리 해보기

Component 분리

지금까지는 하나의 파일 : main.ts에서 개발했지만, 확장성을 위해 다음과 같이 파일을 분리하여 관리 한다.



main.ts

main.ts 파일을 세 개의 파일로 분리



main.ts

최초의 Component 로딩



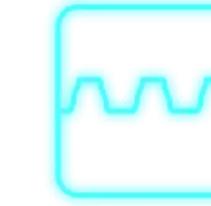
app.component.ts

페이지 header Component 포함



car-parts.component.ts

자동차부품 목록 포함



두 개의 Component로 분리

main.ts 분리

최초 로딩될 Component 가 위치

main.ts

TypeScript

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

@Component({
  selector: 'my-app',
  template: `<h1>{{title}}</h1>
  ...
})

class AppComponent {
  title = 'Ultra Racing';
  carParts = [ ... ];
  totalCarParts() { ... };
}

...
```



다른 곳으로 분리시켜야 할 코드

app.component.ts

대부분 코드를 app.component.ts 로 이동한다.

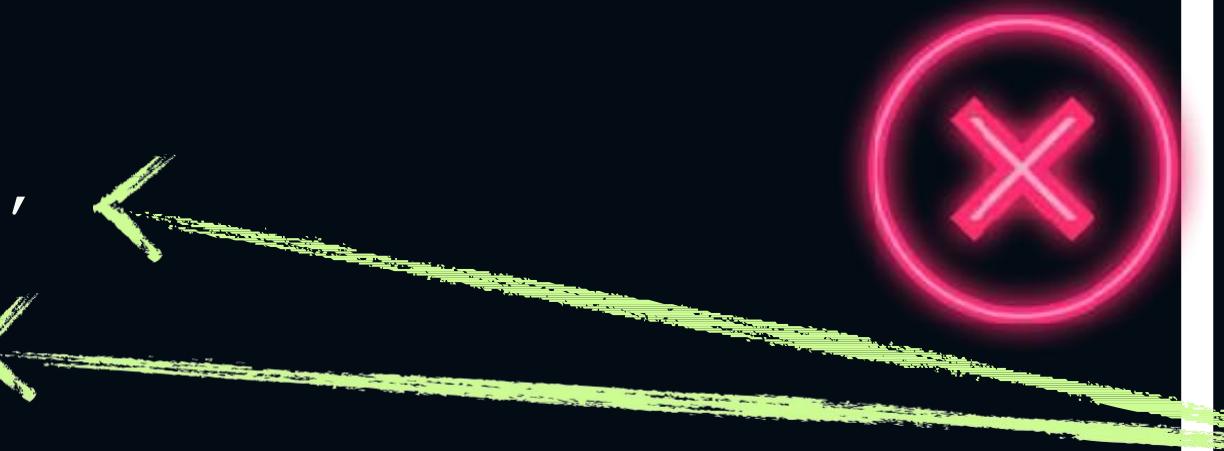
main.ts

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

TypeScript

app.component.ts

```
import { Component } from '@angular/core';  
  
@Component({  
  selector: 'my-app',  
  template: `<h1>{{title}}</h1>  
  ...  
})  
class AppComponent {  
  title = 'Ultra Racing';  
  carParts = [...];  
  totalCarParts() { ... };  
}
```



파일이 분리되면 다른 파일에 있는 클래스에 접근할 수 없다.

다른 파일에 접근하려면 자바스크립트 모듈 시스템
을 사용해야 한다. (ECMA6)

export & import

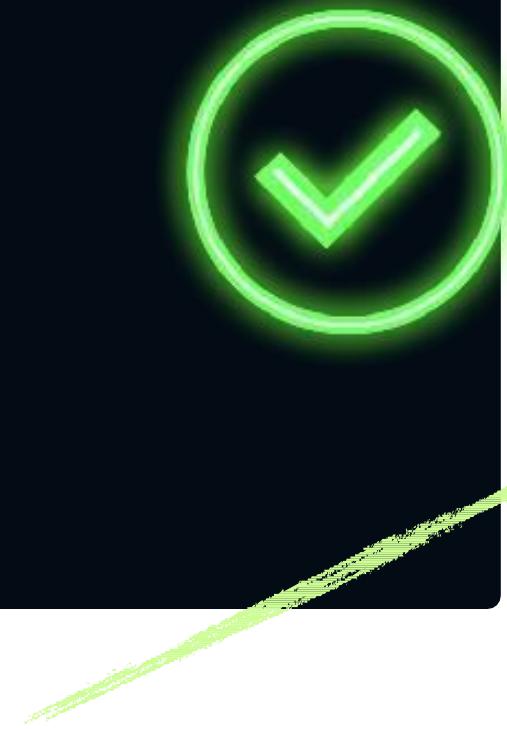
ES2015(ECMA6) 기능인 **export**와 **import**를 사용해야 한다.

main.ts

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

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

TypeScript



- 1) 외부로 노출하고 싶은 클래스를 export
- 2) main.ts 파일에서 import 한다.

app.component.ts

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

@Component({
  selector: 'my-app',
  template: `<h1>{{title}}</h1>
  ...
})

export class AppComponent {
  title = 'Ultra Racing';
  carParts = [...];
  totalCarParts() { ... };
}
```

TypeScript

이름은 서로 동일해야 함

컴포넌트를 하나 더 분리한다

car-parts.component.ts 생성.



Ultra Racing

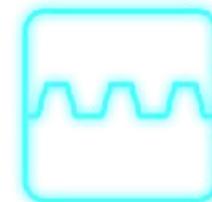
There are 9 total parts in stock.

- **SUPER TIRES**

These tires are the very best

€4.99

5 in Stock



- **REINFORCED SHOCKS**

Shocks made from kryptonite

€9.99

4 in Stock

- **PADDED SEATS**

Super soft seats for a smooth ride

€24.99

Out of Stock

두 개의 컴포넌트로 분리

app.component.ts에서 차부품 목록 관련 코드를 제거.

app.component.ts

TypeScript

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

@Component({
  selector: 'my-app',
  template: `<h1>{{title}}</h1>
    <p>There are {{totalCarParts()}} total parts in stock.</p>
    <ul>...</ul>`
})
export class AppComponent {
  title = 'Ultra Racing';
  carParts = [...];
  totalCarParts() { ... };
}
```

Car Parts Component 분리

car-parts.component.ts TypeScript

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

@Component({
  selector: 'car-parts',
  template: `
    <p>There are {{totalCarParts()}} total parts in stock.</p>
    <ul>...</ul>
  `
})
export class CarPartsComponent {
  carParts = [...];
  totalCarParts() { ... };
}
```

새로운 셀렉터 생성!

새로운 **component**를 사용하겠다고 **module**에 등록을 해야 함!

import CarPartsComponent

main.ts

TypeScript

```
import { AppComponent } from './app.component';
import { CarPartsComponent } from './car-parts.component';

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

car-parts.component.ts

TypeScript

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

@Component({
  selector: 'car-parts',
  template: `...`
})
export class CarPartsComponent {
  carParts = [...];
  totalCarParts() { ... };
}
```

main.ts 파일에 두 개의 컴포넌트가 등록된다.

- main.ts 파일에서 새로운 컴포넌트를 import 해야 함
- module declarations 배열에 CarPartsComponent를 등록

새로운 컴포넌트 완성

app.component.ts

TypeScript

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

@Component ({
  selector: 'my-app',
  template: `<h1>{{title}}</h1>
    <car-parts></car-parts>`  
  ← 새로운 셀렉터
})
export class AppComponent {
  title = 'Ultra Racing';
}
```



<car-parts> 셀렉터로 CarPartsComponent를 App에
서 보여준다.

Ultra Racing

There are 9 total parts in stock.

- **SUPER TIRES**

These tires are the very best

€4.99

5 in Stock

- **REINFORCED SHOCKS**

Shocks made from kryptonite

€9.99

4 in Stock

- **PADDED SEATS**

Super soft seats for a smooth ride

€24.99

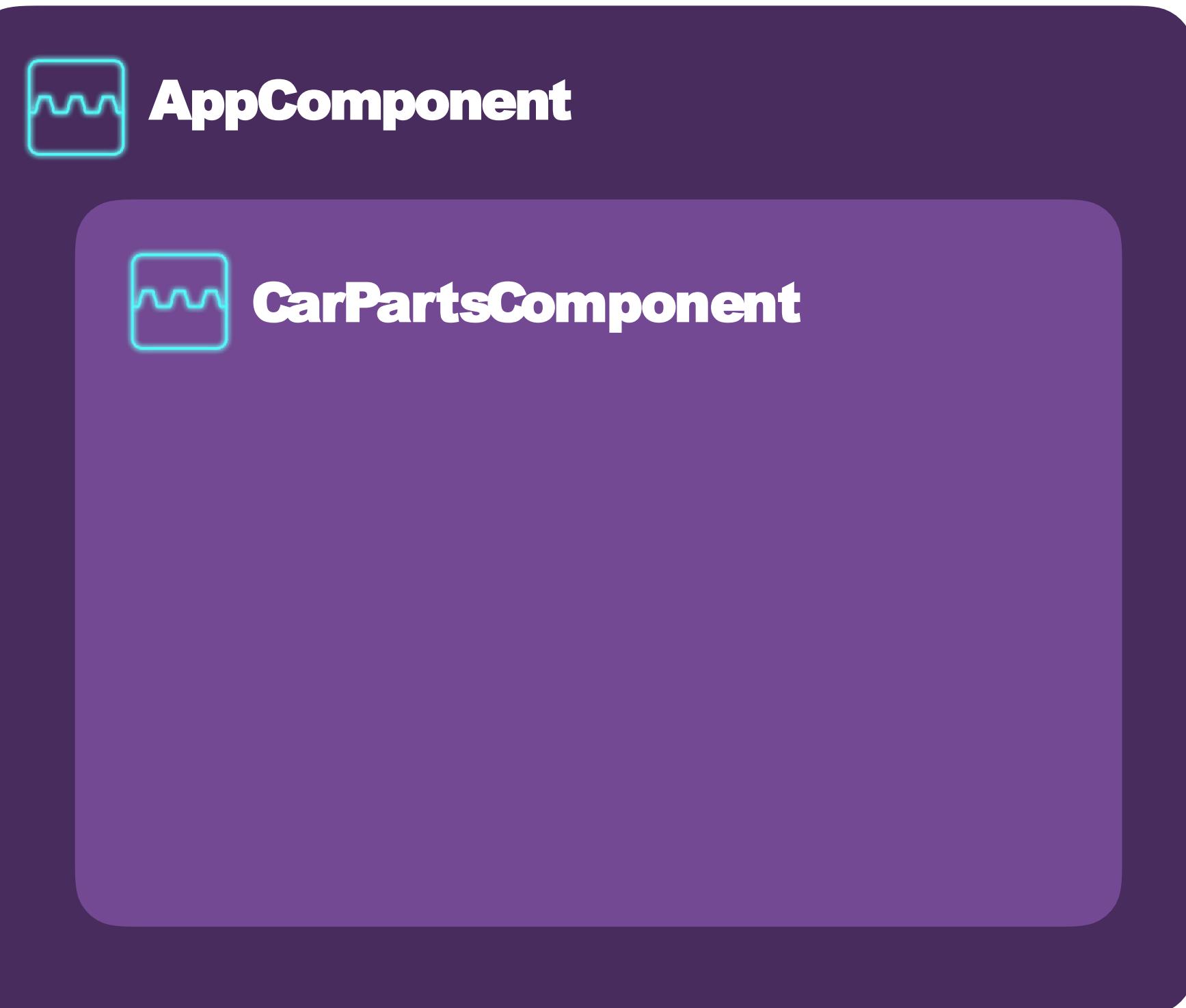
Out of Stock

최종 분리된 모습



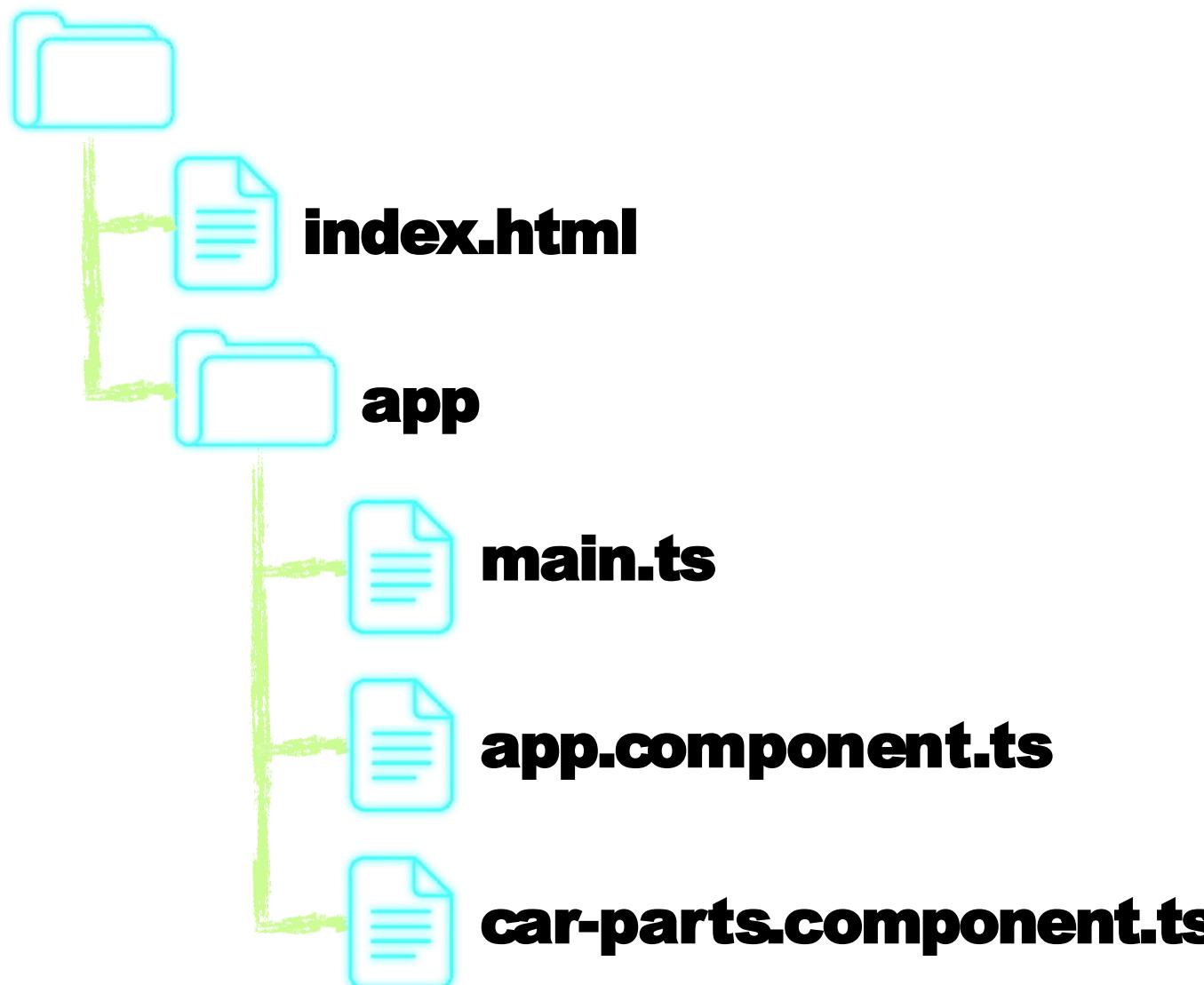
컴포넌트는 재사용 할 수 있어야 한다. 다른 어플리케이션에서도 재사용이 가능하다.

두 컴포넌트 구조



이번 장 정리

- main.ts: 모든 컴포넌트가 등록되고 어플리케이션을 시작하는 곳이다.
- 클래스를 import하기 위해서 export 키워드를 사용해 export 해준다.
- 컴포넌트는 어플리케이션의 주요 구성 요소이다.



Angular2 스타일 가이드 : <https://angular.io/docs/ts/latest/guide/style-guide.html>

Component HTML & CSS

html와 css 적용

컴포넌트에 CSS 적용

car-parts.component.ts

TypeScript

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

@Component({
  selector: 'car-parts',
  template: `...
    <p>{{carPart.description}}</p>
    <p>{{carPart.price | currency:'EUR':true}}</p>
    ...
  `
})
export class CarPartsComponent {
  ...
}
```



HTML 템플리트와 함께 CSS도 포함할 수 있다.

Style 배열 추가

car-parts.component.ts

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

@Component({
  selector: 'car-parts',
  template: `...
    <p class="description">{{carPart.description}}</p>
    <p class="price">{{carPart.price | currency:'EUR':true}}</p>
    ...
  `,
  styles: [
    '.description {
      color: #444;
      font-size: small;
    }
    .price {
      font-weight: bold;
    }
  ]
})
export class CarPartsComponent {
```

TypeScript



CSS 클래스

```
@Component({
  selector: 'car-parts',
  template: `...
    <p class="description">{{carPart.description}}</p>
    <p class="price">{{carPart.price | currency:'EUR':true}}</p>
    ...
  `,
```

```
  styles: [
    '.description {
      color: #444;
      font-size: small;
    }
    .price {
      font-weight: bold;
    }
  ]
})
```



배열로 정의

Ultra Racing

There are 9 total parts in stock.

- SUPER TIRES

These tires are the very best

€4.99

5 in Stock

- REINFORCED SHOCKS

Shocks made from kryptonite

€9.99

4 in Stock

- PADDED SEATS

Super soft seats for a smooth ride

€24.99

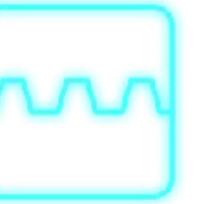
Out of Stock

HTML & CSS 분리

컴포넌트에서 **HTML**과 **CSS**를 분리.



car-parts.component.ts



HTML과 **CSS**를 각 파일로 분리.



car-parts.component.html



car-parts.component.css

HTML & CSS 분리

car-parts.component.ts

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

@Component({
  selector: 'car-parts',
  templateUrl: 'app/car-parts.component.html',
  styleUrls: ['app/car-parts.component.css']
})
export class CarPartsComponent {
  ...
}
```

TypeScript

car-parts.component.html

```
<p>There are {{totalCarParts()}} total parts in stock.</p>
<ul>...</ul>
```

HTML

car-parts.component.css

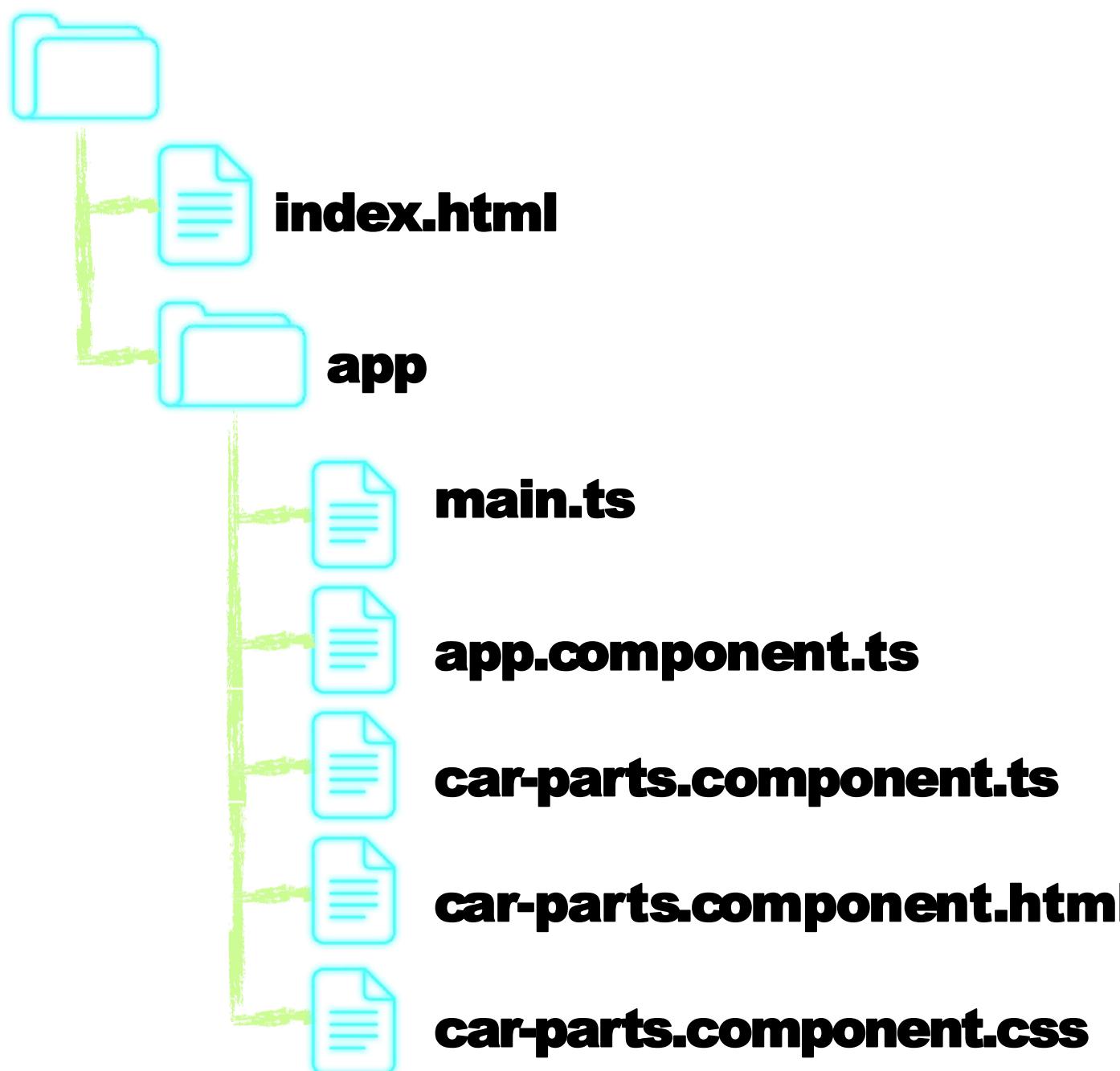
```
.description {
  color: #444;
  font-size: small;
}
.price {
  font-weight: bold;
}
```

CSS

컴포넌트 메타데이터 내부에서 **HTML**과 **CSS**를 외부 파일로 분리해서 참조한다.

이번 장 정리

- HTML 템플릿과 마찬가지로 CSS도 컴포넌트 메타데이터에 포함할 수 있다.
- CSS는 자동으로 컴포넌트 스코프로 설정된다.
- HTML과 CSS는 각각 파일로 분리될 수 있다.



Models & Mocks

Model 클래스 작성

Mock Data 작성

Model class 생성

타입스크립트의 장점인 객체지향적 기능을 활용: 모델 클래스 생성

기본적인 자바스크립트 객체

car-part.ts

TypeScript

```
export class CarPart {  
    id: number;  
    name: string;  
    description: string;  
    inStock: number;  
    price: number;  
}
```

각 속성에 타입이 추가
타입스크립트의 특성

정적인 타입 (static type) : 컴파일 타임에 변수의 타입을 체크해서
더욱 정확한 값을 세팅하고 사용할 수 있게 해준다.

Model class 적용

car-parts.component.ts

TypeScript

```
import { Component } from '@angular/core';
...
})
export class CarPartsComponent {
  carParts = [ {
    "id": 1,
    "name": "Super Tires",
    "description": "These tires are the very best",
    "inStock": 5,
    "price": 4.99
  }, { ... }, { ... } ];
...
}
```



모델 객체를 도입

car-part.ts

TypeScript

```
export class CarPart {
  id: number;
  name: string;
  description: string;
  inStock: number;
  price: number;
}
```

Model class 적용 : model class인 CarPart 정의하고 import

car-parts.component.ts

TypeScript

```
import { Component } from '@angular/core';
import { CarPart } from './car-part'; ←

...
)

export class CarPartsComponent {
  carParts: CarPart[] = [{
    "id": 1,
    "name": "Super Tires",
    "description": "These tires are the very best",
    "inStock": 5,
    "price": 4.99
  }, { ... }, { ... }];
  ...
}
```



CarPart
클래스 배열 타입

import CarPart **model 클래스**

car-part.ts

TypeScript

```
export class CarPart {
  id: number;
  name: string;
  description: string;
  inStock: number;
  price: number;
}
```

Model class 적용된 모습

car-parts.component.ts

```
...
carParts: CarPart[] = [ {
  "id": 1,
  "name": "Super Tires",
  "description": "These tires are the very best",
  "inStock": 5,
  "price": 4.99
}, { ... }, { ... } ];
```

TypeScript

car-part.ts

```
export class CarPart {
  id: number;
  name: string;
  description: string;
  inStock: number;
  price: number;
}
```

TypeScript

car-parts.component.html

HTML

```
<p>There are {{totalCarParts()}} total parts in stock.</p>
<ul>
  <li *ngFor="let carPart of carParts">
    <h2>{{carPart.name | uppercase}}</h2>
    <p class="description">{{carPart.description}}</p>
    <p class="price">{{carPart.price | currency:'EUR':true }}</p>
    <p *ngIf="carPart.inStock > 0">{{carPart.inStock}} in Stock</p>
    <p *ngIf="carPart.inStock === 0">Out of Stock</p>
  </li>
</ul>
```

Ultra Racing

There are 9 total parts in stock.

• SUPER TIRES

These tires are the very best

€4.99

5 in Stock

Mock Data 생성 : json 포맷

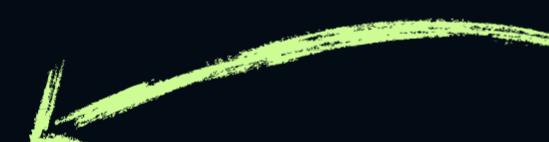
웹 서비스(**API**)를 통해 데이터를 받아오더라도, **mock 데이터(fake data)**를 통해 테스트 하는 것은 좋은 개발 방법이다.

car-parts.component.ts

TypeScript

```
import { Component } from '@angular/core';
import { CarPart } from './car-part';

...
}

export class CarPartsComponent {
  carParts: CarPart[] = [
    {
      "id": 1,
      "name": "Super Tires", 
      "description": "These tires are the very best",
      "inStock": 5,
      "price": 4.99
    }, { ... }, { ... }];
  ...
}
```



mocks.ts

TypeScript

새로운 파일을 만들어 Mock 데이터를 옮긴다.

Mock Data 사용

car-parts.component.ts

TypeScript

```
import { Component } from '@angular/core';
import { CarPart } from './car-part';
import { CARPARTS } from './mocks'; ←

...
})

export class CarPartsComponent {
  carParts: CarPart[];
}

ngOnInit() {
  this.carParts = CARPARTS;
}
...
```



→

mocks.ts

TypeScript

```
import { CarPart } from './car-part';

export const CARPARTS: CarPart[] = [
  {
    "id": 1,
    "name": "Super Tires",
    "description": "These tires are the very best",
    "inStock": 5,
    "price": 4.99
  }, { ... }, { ... }];

```

let 대신에 const를 사용: ES2015 기능으로
CARPARTS 변수는 재 할당 될수 없다.

ngOnInit 함수는 컴포넌트가 초기화 된 직후 호출된다. **Property**를 초기화 하기에 적합.

생성자(**constructor**)에서 초기화 할 수도 있지만, 테스트가 힘들다는 단점이 있다.

Component Architecture



app/car-parts.component.ts

```
import { Component } from '@angular/core';
import { CarPart } from './car-part';
import { CARPARTS } from './mocks';

@Component({
  selector: 'car-parts',
  templateUrl: 'app/car-parts.component.html',
  styleUrls: ['app/car-parts.component.css']
})
export class CarPartsComponent {
  carParts: CarPart[];  
...
```

이번 장 정리

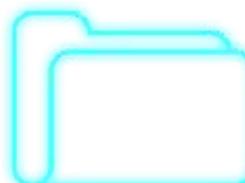
- Typescript에서는 Model을 정의하기 위해 클래스를 사용할 수 있다.
- Typescript는 클래스 속성의 타입을 정의함으로써, 컴파일타임에 데이터 타입을 체크할 수 있게 해준다. 이는 보다 더 나은 코드를 작성할 수 있게 해준다.
- Mock 데이터 파일을 모델과 Component와는 다르게 별도의 파일로 분리해서 개발 시 활용한다.

Property & CSS class Binding

Property Binding
CSS style Binding

디자인 추가

디자이너로 부터 전달받은 **html** 과 **CSS**를 **Angular**에 적용한다.



Raw HTML & CSS From Designer



index.html



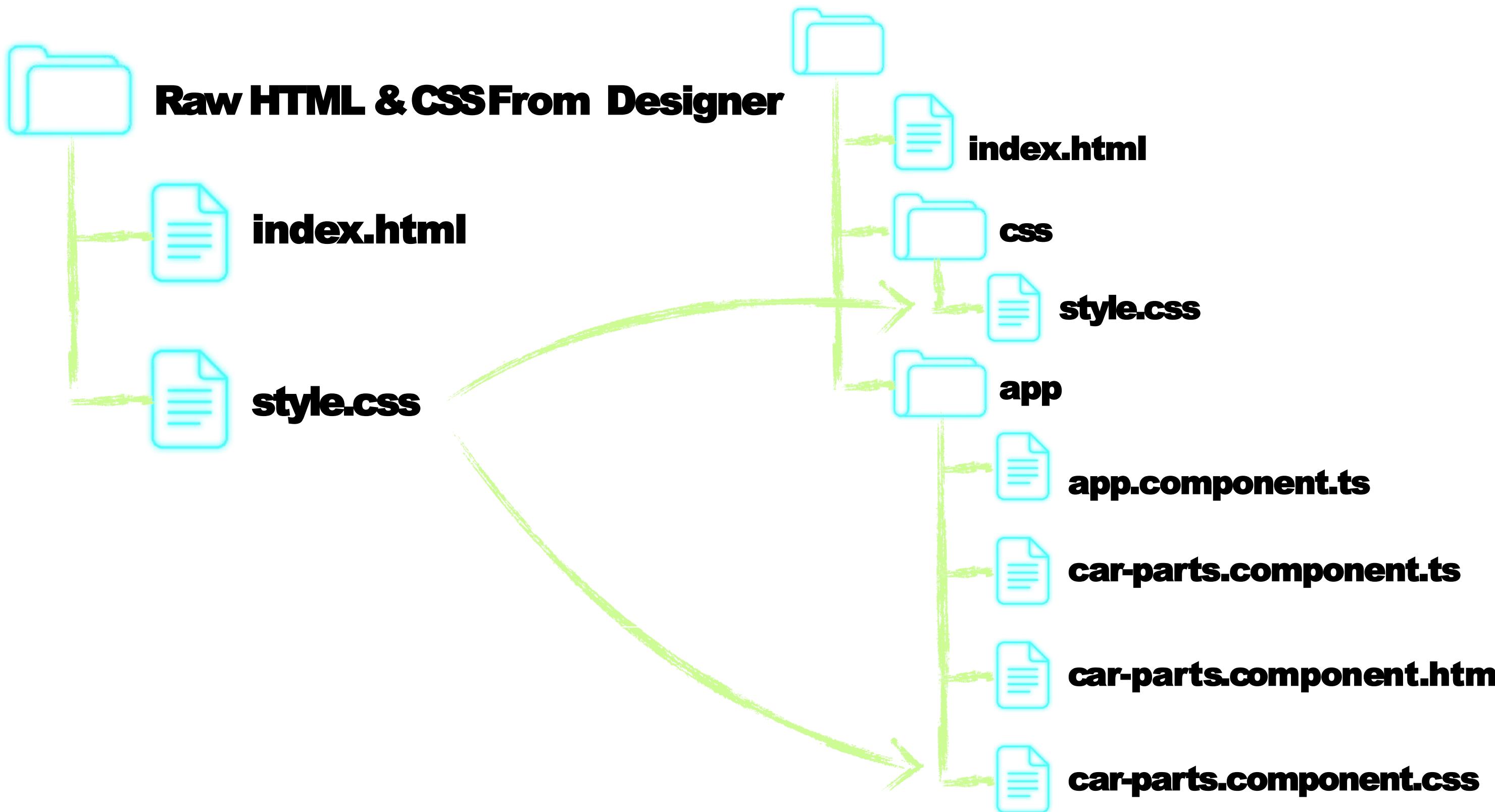
style.css

The screenshot shows a dark-themed e-commerce application interface. At the top, the text "ULTRA RACING" is displayed in a large, glowing blue font, accompanied by a lightning bolt icon. Below this, a message states "There are 9 total parts in stock." The main content area features two product cards. The first card for "SUPER TIRES" displays an image of a tire, the product name, a description ("These tires are the very best."), the current stock level ("5 in stock"), the price ("€4.99"), and a quantity selector with a value of "0". The second card for "PADDED SEATS" shows a seat, the product name, a description ("These tires are the very best."), the price ("€9.99"), and a quantity selector with a value of "0". Both cards include a "GET IT NOW" button at the bottom.

0/미지와 수량을 후에 추가

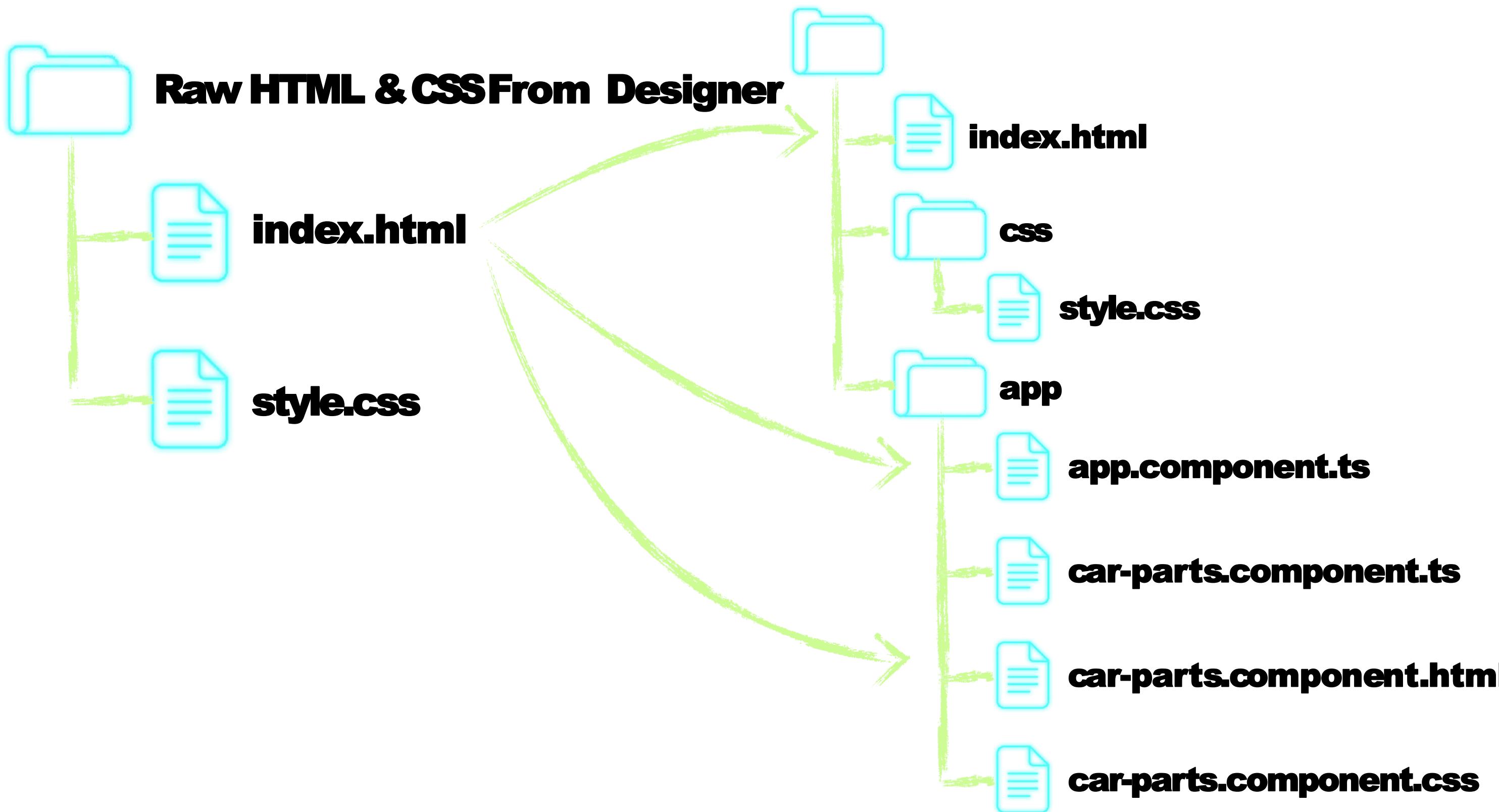
CSS 이동

전체화면에 적용되는 CSS는 공통 폴더에 추가하고 특정 컴포넌트에만 적용되는 CSS는 해당 컴포넌트 폴더의 CSS에 추가한다.



HTML 분리

HTML 파일도 분리해서 추가한다.



현재까지의 앱

이미지와 버튼을 추가한다.

ULTRA RACING



There are 9 total parts in stock.

SUPER TIRES

These tires are the very best

€4.99

5 in Stock

REINFORCED SHOCKS

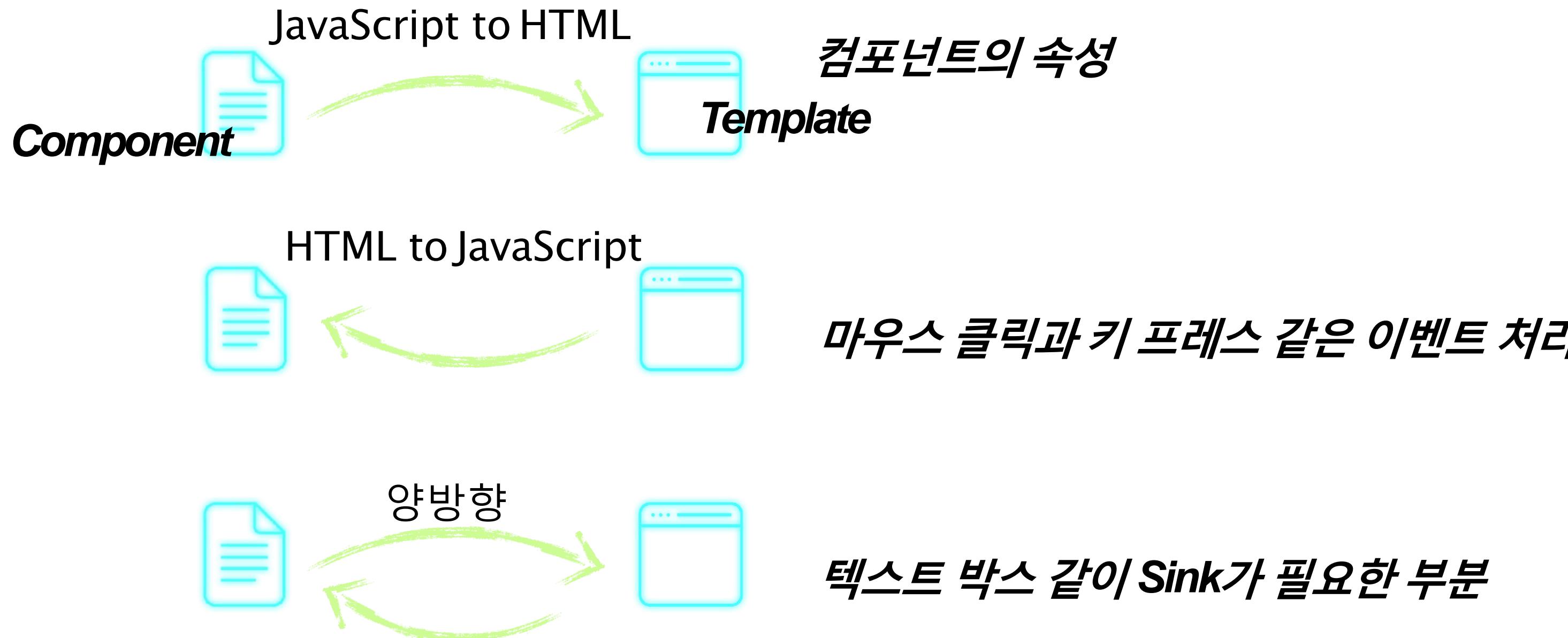
Shocks made from kryptonite

€9.99

4 in Stock

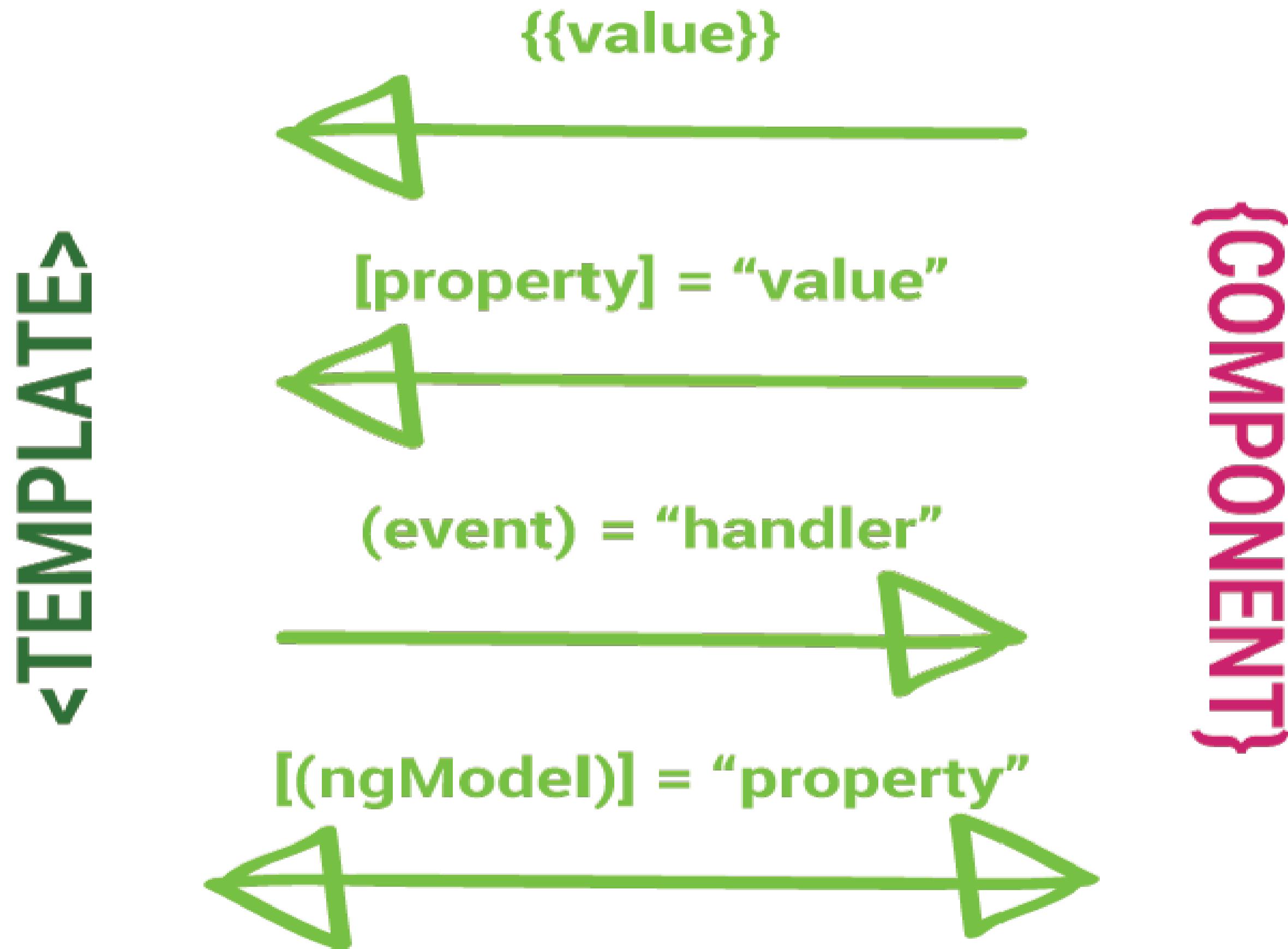
Data Binding :

angular와 같은 자바스크립트 프레임워크에서는 **HTML**과 코드를 추상화한다.



Note: 타입스크립트는 결국 자바스크립트로 변환되어, 여기서 자바스크립트라고 말한다.

Data Binding :



Data Binding : JavaScript to HTML {{ }}

컴포넌트의 모든 데이터는 **Interpolation**을 통해 **HTML**로 전달된다.

car-parts.component.html

TypeScript

```
<li class="card" *ngFor="let carPart of carParts" >
  <div class="panel-body">
    <table class="product-info">
      <tr>
        <td>
          <h2>{{carPart.name | uppercase}}</h2>
          <p class="description">{{carPart.description}}</p>
          <p class="inventory" *ngIf="carPart.inStock > 0">{{carPart.inStock}} in Stock</p>
          <p class="inventory" *ngIf="carPart.inStock === 0">Out of Stock</p>
        </td>
        <td class="price">{{carPart.price | currency:'EUR':true }}</td>
      </tr>
    </table>
  </div>
</li>
```

자바스크립트 코드가 HTML에 바인딩 될 때, 컴포넌트의 속성
(변수)이 HTML에 포함되어 화면에 출력된다.

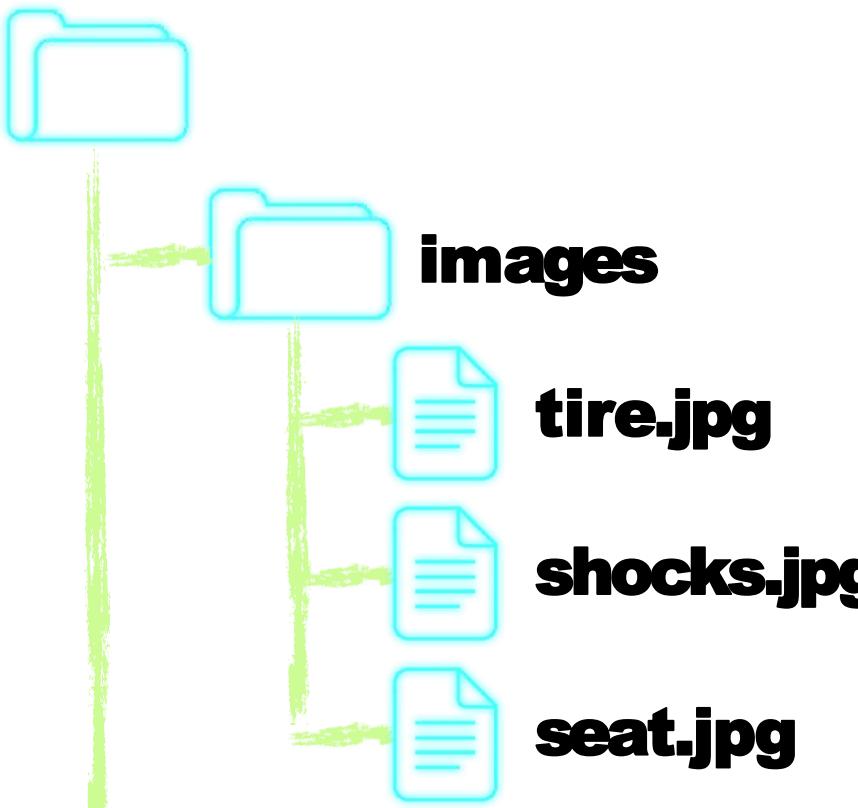
Data Binding : property binding

model에 image 속성을 추가하고, 이미지파일을 폴더에 추가하고, 그리고 mock 데이터에 파일의 경로 추가.

car-part.ts

```
export class CarPart {  
    id: number;  
    name: string;  
    description: string;  
    inStock: number;  
    price: number;  
    image: string;  
}
```

TypeScript



mocks.ts

```
import { CarPart } from './car-part';  
  
export let CARPARTS: CarPart[] = [  
    {  
        "id": 1,  
        "name": "Super Tires",  
        "description": "These tires are the very best",  
        "inStock": 5,  
        "price": 4.99,  
        "image": "/images/tire.jpg"  
    },  
    {  
        "id": 2,  
        "name": "Reinforced Shocks",  
        "description": "Shocks made from kryptonite",  
        "inStock": 4,  
        "price": 9.99,  
        "image": "/images/shocks.jpg"  
    },  
    { ... } ];
```

TypeScript

Data Binding : property binding

컴포넌트 속성을 DOM 엘리먼트에 바인딩 할 때 사용되는 문법을 확인한다.

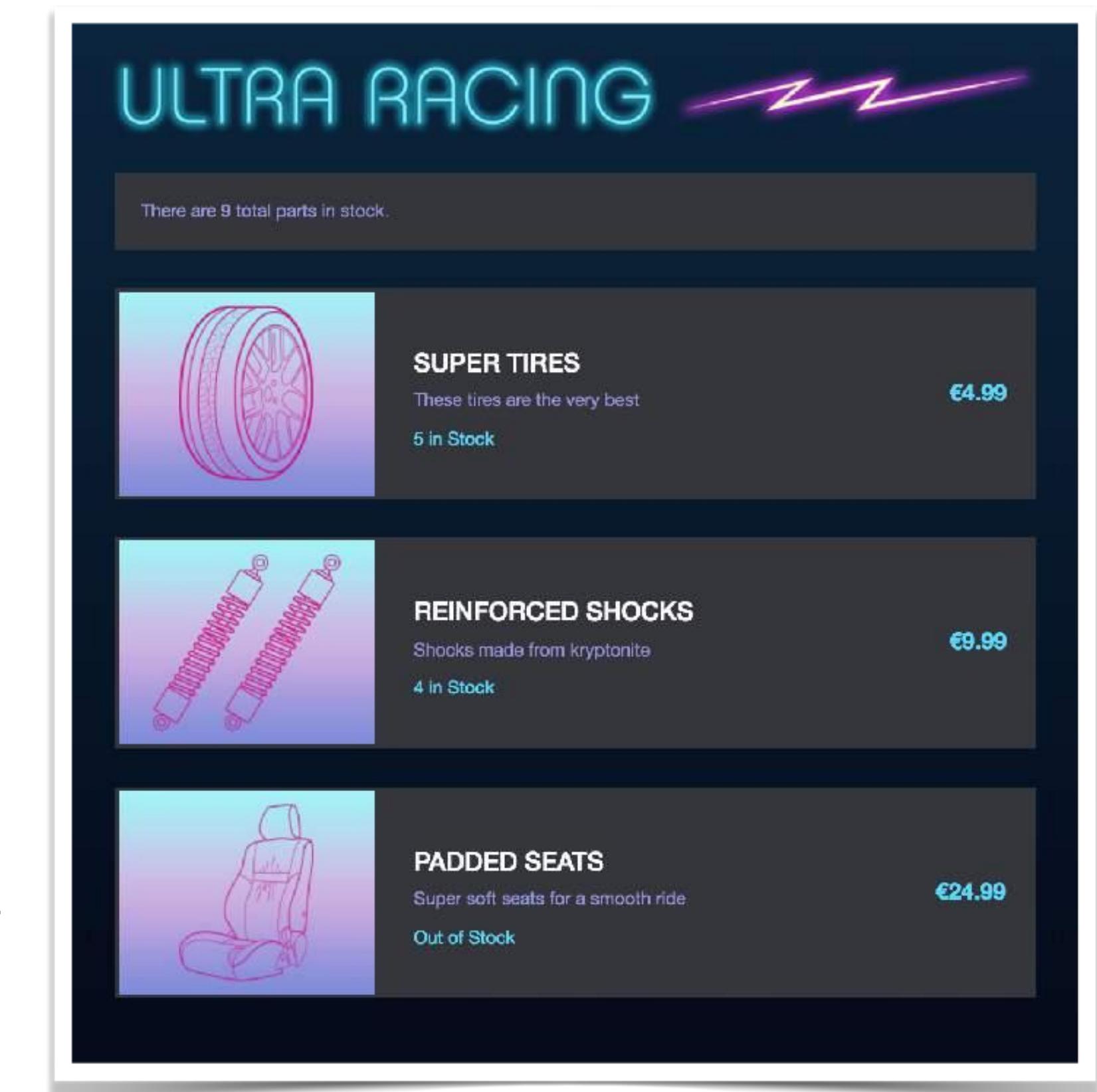
car-parts.component.html **TypeScript**

```
<li class="card" *ngFor="let carPart of carParts">
  <div class="panel-body">
    <div class="photo">
      <img [src] = "carPart.image">
    </div>
    <table class="product-info">
      <tr>
        <td>
          <h2>{{carPart.name | uppercase}}</h2>
          ...
        </td>
      </tr>
    </table>
  </div>
</li>
```

대괄호(square brackets)를 사용

angular에서 대괄호의 의미는 DOM 엘리먼트 속성을 컴포넌트 속성에 바인딩한다는 것을 의미한다.

컴포넌트 속성이 변하면 즉시 업데이트 된다.



Property Binding 예제

모든 DOM 엘리먼트에 바인딩 가능하다.

```
<div hidden>secret</div>
```

```
<button disabled>Purchase</button>
```

```
<img alt="Image Description">
```

이미지 예

```
<img [src] = "carPart.image">
```

```
<div [hidden] = "!user.isAdmin" >secret</div>
```

```
<button [disabled] = "isDisabled" >Purchase</button>
```

```
<img [alt] = "image.description">
```

Data Binding : css style property binding

featured style 부여

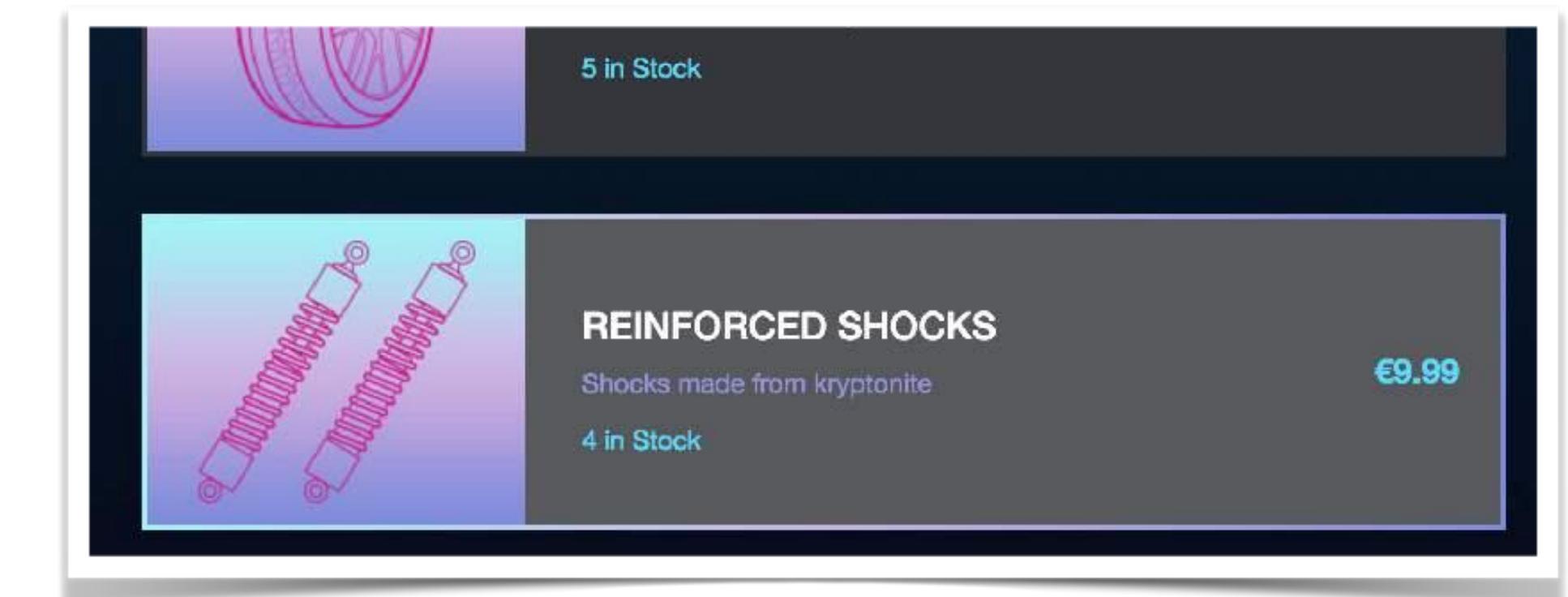
car-parts.component.css

CSS

```
...
.featured {
  background: #57595D;
  -webkit-border-image: -webkit-linear-gradient(right, #818fd8 0%, #cbb4e2 50%, #a6f2f5 100%);
  -o-border-image: linear-gradient(to left, #818fd8 0%, #cbb4e2 50%, #a6f2f5 100%);
  border-image: linear-gradient(to left, #818fd8 0%, #cbb4e2 50%, #a6f2f5 100%);
  border-image-slice: 1;
}
```

featured 클래스 속성은 밝은 배경과 테두리를 제공한다.

CSS 클래스 속성 부여하는 방법에 대해서 알아보자.



Data Binding : featured Property & Data 추가

car-part.ts

```
export class CarPart {  
    ...  
    image: string;  
    featured: boolean;  
}
```

TypeScript

mocks.ts

```
export let CARPARTS: CarPart[] = [ {  
    "id": 1,  
    ...  
    "featured": false  
},  
{  
    "id": 2,  
    ...  
    "featured": true  
},  
{  
    "id": 3,  
    ...  
    "featured": false  
} ];
```

TypeScript

Data Binding : css style property binding

클래스 속성을 부여하기 위한 문법

car-parts.component.html

TypeScript

```
<ul>
  <li class="card" *ngFor="let carPart of carParts" [class.featured]="carPart.featured">
    <div class="panel-body">
      ...
    </div>
  </li>
</ul>
```

carPart.featured 값이 true 이면 featured CSS 클래스가 적용됨.

carPart.featured 값이 false 이면 featured CSS 클래스가 제거됨.

```
<div [class.name]="property">
```



이번 장 정리

- Property Binding을 통해 Component Property와 DOM Element를 바인딩 할 수 있다.
- Component Property 값을 업데이트 하면 DOM property 속성도 즉시 업데이트 된다.
그러나 반대는 성립하지 않는다. - “one-way binding.”
- Component Property 값이 true이면 CSS 클래스가 적용된다. 이것을 css class 바인딩 이라고 한다.

이벤트 바인딩

One Way Binding

Data Binding 종류

속성 바인딩

클래스바인딩

JavaScript to HTML

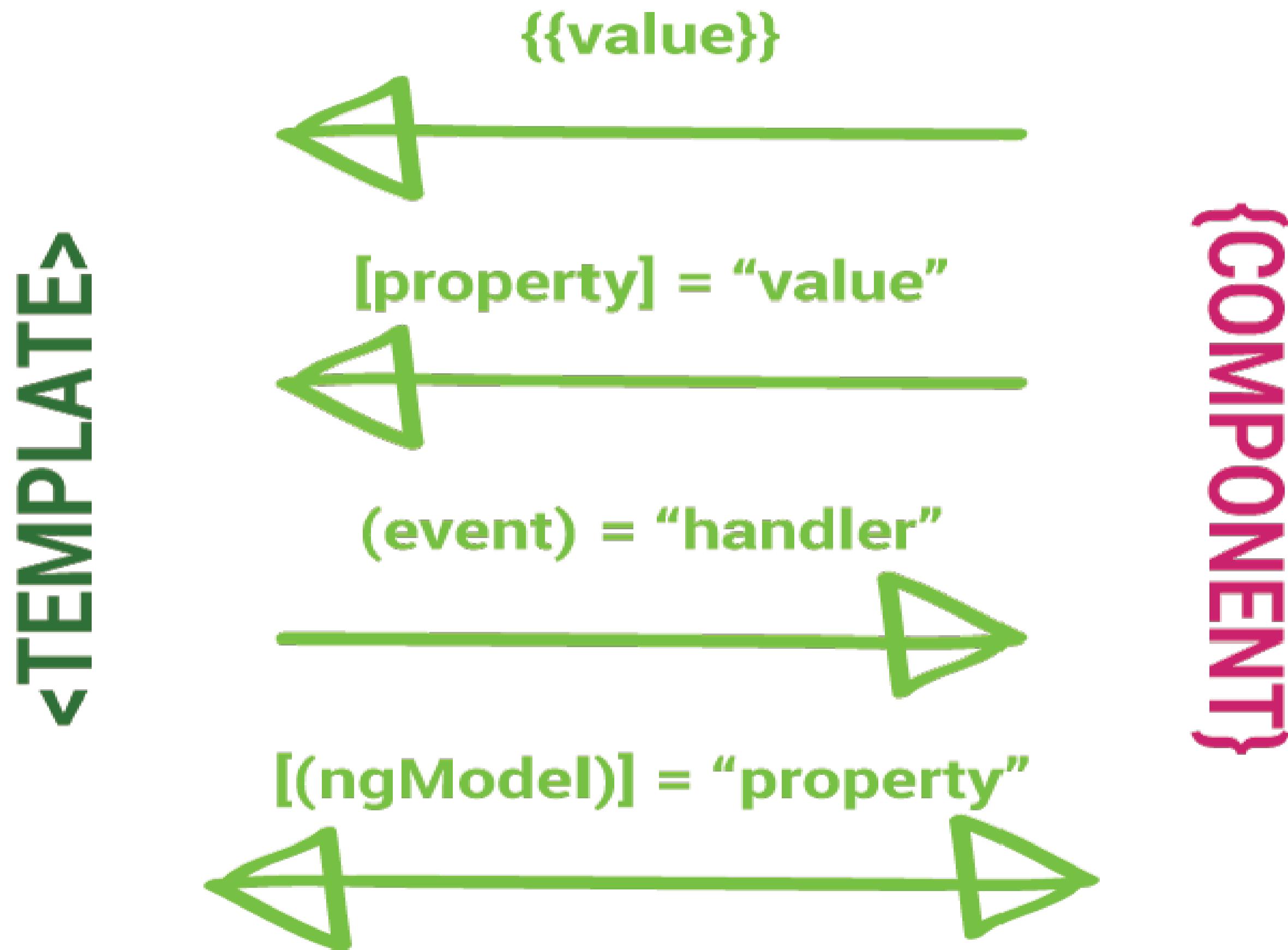


이벤트 바인딩

마우스 클릭이나 키 바인딩 등



Data Binding 종류



Event 처리 : event 추가

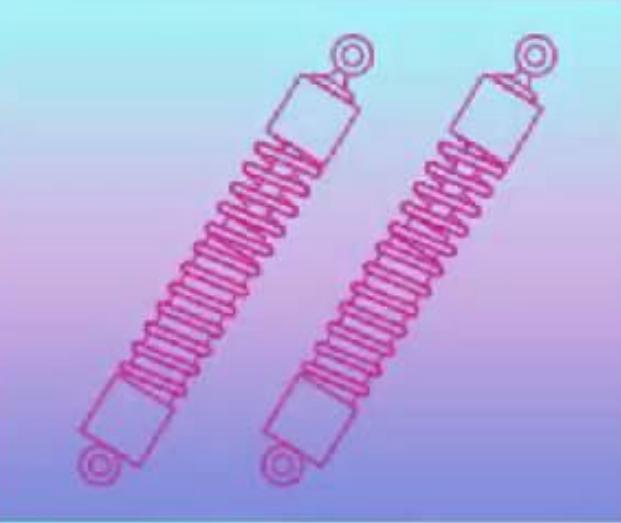
ULTRA RACING

There are 9 total parts in stock.



SUPER TIRES
These tires are the very best
5 in Stock

€4.99 - 0 +



REINFORCED SHOCKS
Shocks made from kryptonite
4 in Stock

€9.99 - 0 +

Event 처리 : Quantity Property & Data 추가

car-part.ts **model** 모델 클래스에 **quantity** 속성과 **mock** 데이터에 추가

car-part.ts

```
export class CarPart {  
  ...  
  image: string;  
  featured: boolean;  
  quantity: number;  
}
```

TypeScript

mocks.ts

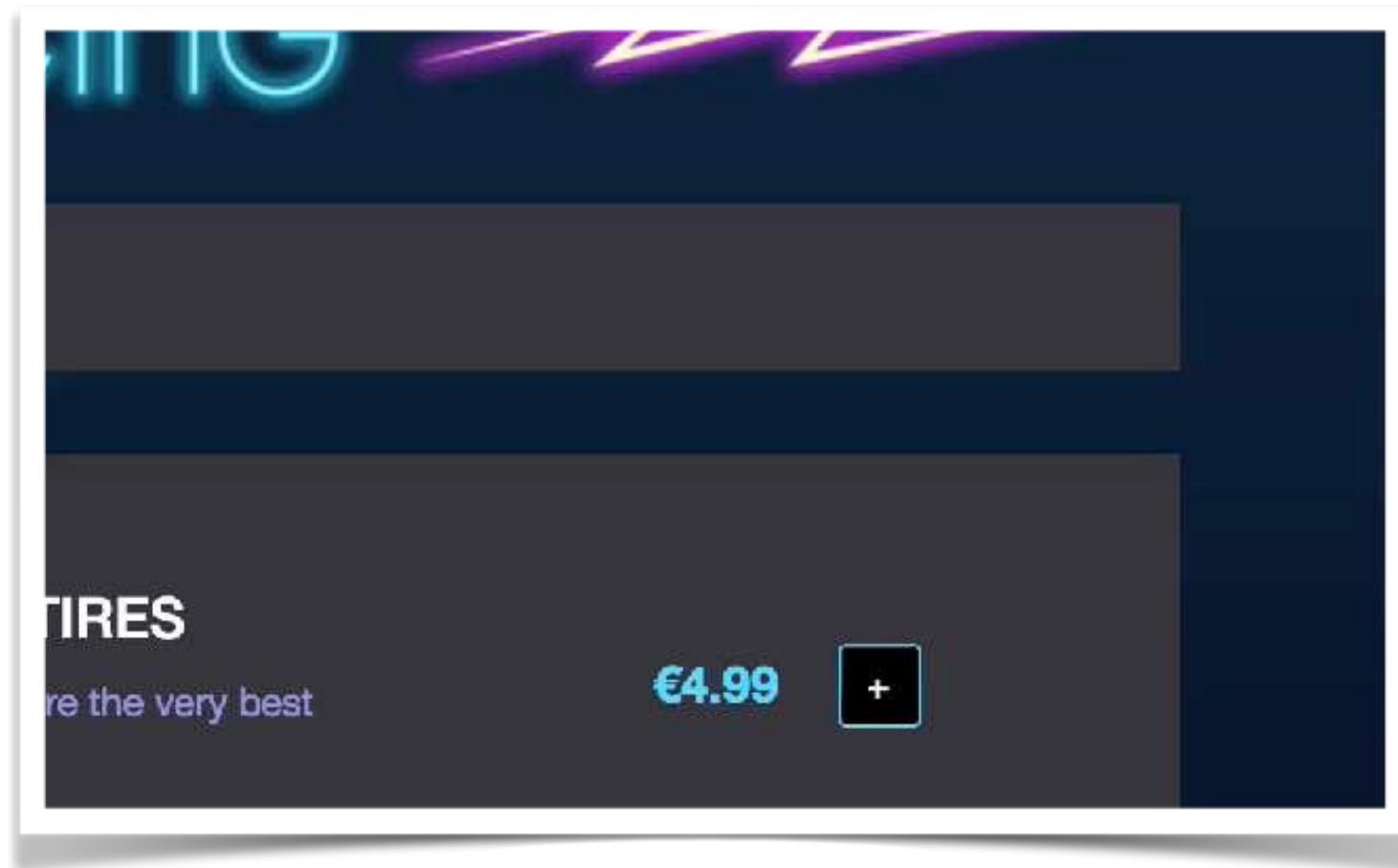
```
export let CARPARTS: CarPart[] = [ {  
  "id": 1,  
  ...  
  "featured": false,  
  "quantity": 0  
}, { ... }, { ... } ];
```

TypeScript

Event 처리 : 간단한 버튼 추가

car-parts.component.ts **TypeScript**

```
...  
export class CarPartsComponent {  
    ...  
  
    upQuantity() {  
        alert("You Called upQuantity");  
    }  
}
```



템플리트로 부터 이벤트를 처리하기 위해 이벤트 이름을 괄호로 감싸고 이벤트를 처리할 함수를 적어준다.

car-parts.component.html **HTML**

```
...  
<td class="price">{{carPart.price | currency:'EUR':true }}</td>  
<td>  
    <div class="select-quantity">  
        <button class="increase" (click)="upQuantity()">+</button>  
    </div>  
</td>
```

Event 처리 : 이벤트 동작 시키기

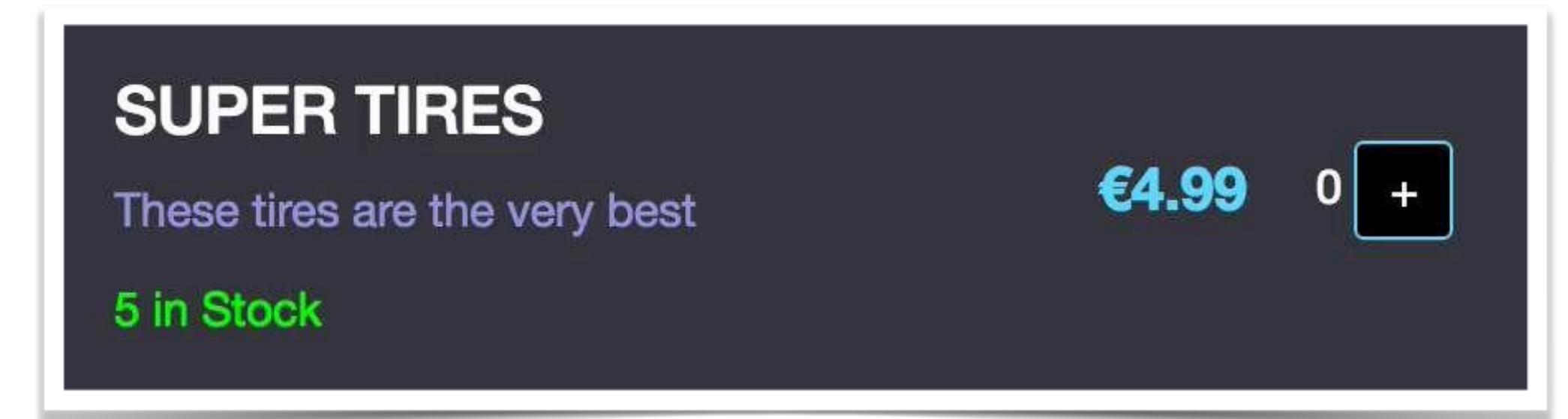
carPart.quantity 사용

car-parts.component.html **HTML**

```
<td class="price">{{carPart.price | currency:'EUR':true }}</td>
<td>
  <div class="select-quantity">
    {{carPart.quantity}}
    <button class="increase" (click)="upQuantity(carPart)">+</button>
```

car-parts.component.ts **TypeScript**

```
export class CarPartsComponent {
  ...
  upQuantity(carPart) {
    carPart.quantity++;
  }
}
```



수량은 재고 이내로 제한해야 함

Event 처리 : 수량 증가하기

car-parts.component.html

HTML

```
<td class="price">{ {carPart.price | currency:'EUR':true } }</td>
<td>
  <div class="select-quantity">
    { {carPart.quantity} }
    <button class="increase" (click)="upQuantity(carPart)">+</button>
```

car-parts.component.ts

TypeScript

```
export class CarPartsComponent {
  ...
  upQuantity(carPart) {
    if (carPart.quantity < carPart.inStock) carPart.quantity++;
  }
}
```

Event 처리 : 수량 감소 버튼 추가

수량을 줄이는 버튼을 추가 - 0 이하로는 조절하지 못한다.

car-parts.component.html

HTML

```
<td class="price">{{carPart.price | currency:'EUR':true }}</td>
<td>
  <div class="select-quantity">
    <button class="decrease" (click)="downQuantity(carPart)">-</button>
    {{carPart.quantity}}
    <button class="increase" (click)="upQuantity(carPart)">+</button>
```

car-parts.component.ts

TypeScript

```
export class CarPartsComponent {
  ...
  downQuantity(carPart) {
    if (carPart.quantity != 0) carPart.quantity--;
  }
}
```



현재 수량이 0이 아닐 때만 수량 감소 가능

Event 처리 : 기타 이벤트

표준 **DOM** 이벤트를 괄호에 감싸서 사용.
일반적으로 **on** 접두어를 생략하여 사용한다.

```
<div (mouseover)="call () ">
```

```
<input (blur)="call () ">
```

```
<input (focus)="call () ">
```

```
<input type="text" (keydown)="call () ">
```

```
<form (submit)="call () ">
```

Event 처리 : \$event 객체

이벤트가 발생한 **target** 객체에 접근하려면 **\$event** 객체를 사용하면 된다.

```
<input type="text" (keydown)="showKey($event)">
```

```
showKey(event) {  
    alert(event.keyCode);  
}
```

\$event 객체를
메소드 내부로
전달 할 수 있다.

```
<h2 (mouseover)="getCoord($event)">Hover Me</h2>
```

```
getCoord(event) {  
    console.log(event.clientX + ", " + event.clientY);  
}
```

이번 장 정리

- 이벤트 바인딩은 모든 DOM 이벤트와 연결될 수 있고, 해당 이벤트가 발생되면 컴포넌트의 이벤트 핸들러 메소드가 호출된다.
- 이벤트를 Listen하기 위해서 접두어 “on” 을 빼고 이벤트 이름을 괄호로 감싼다.
- 이벤트 객체에 접근할 필요가 있을 때 \$event 객체를 컴포넌트 메소드 파라미터로 전달해서 사용한다.

Two-Way Binding

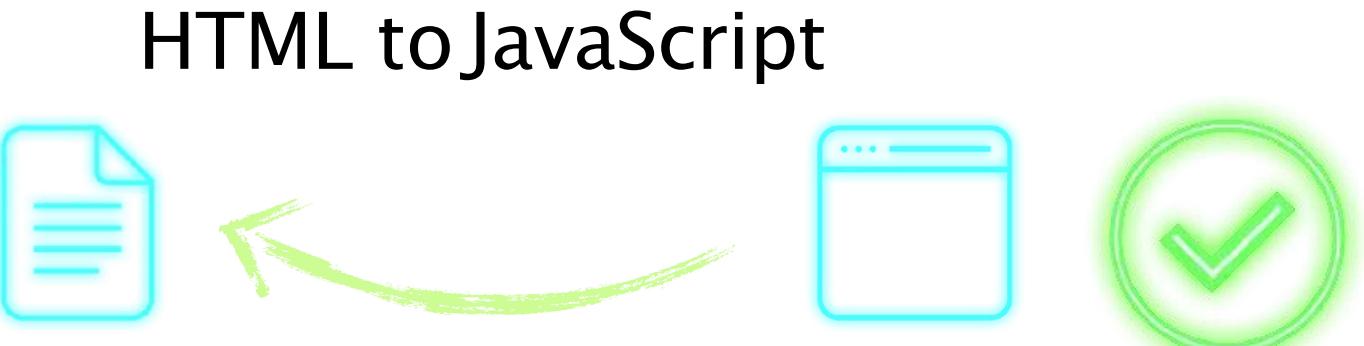
two way 바인딩
property binding + event binding

Data Binding 종류

Property Binding
Class Binding



Event Binding



Two way Binding



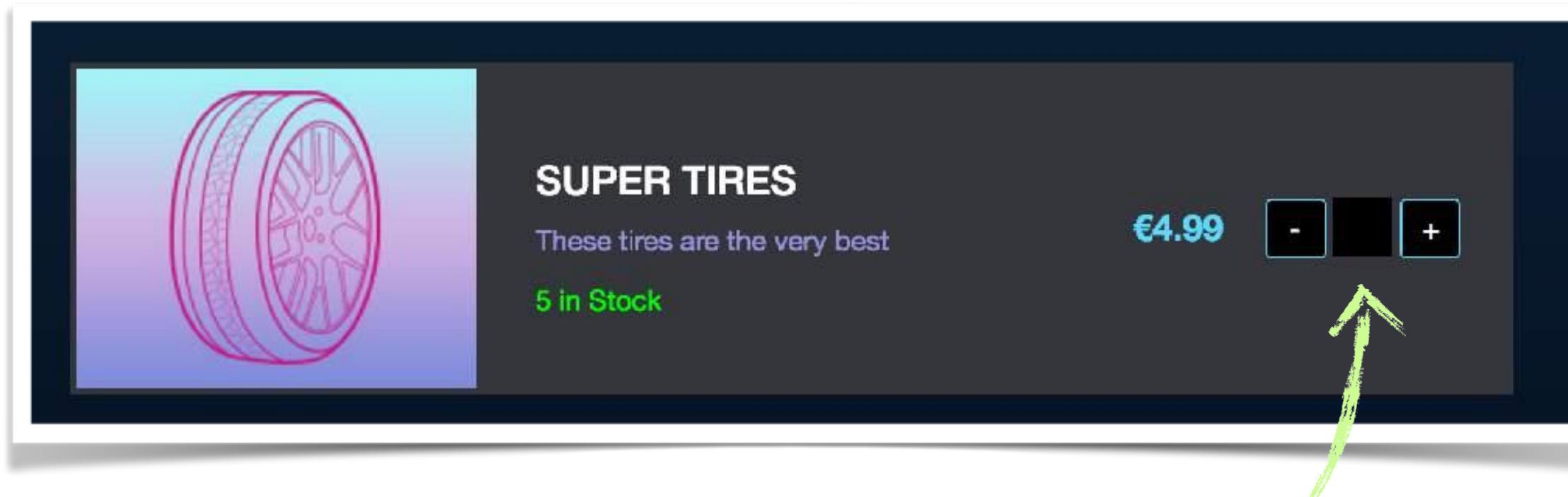
*property binding과 event binding의
Combination (조합)*

Two way binding : Input Field 추가

Input Field를 이용해서 수량 조절

car-parts.component.html HTML

```
<td class="price">{{carPart.price | currency:'EUR':true }}</td>
<td>
  <div class="select-quantity">
    <button class="decrease" (click)="downQuantity(carPart)">-</button>
    <input class="number" type="text">
    <button class="increase" (click)="upQuantity(carPart)">+</button>
  </div>
</td>
```



여기에 수량을 입력하거나 버튼을 이용하여 수량을 조절한다.

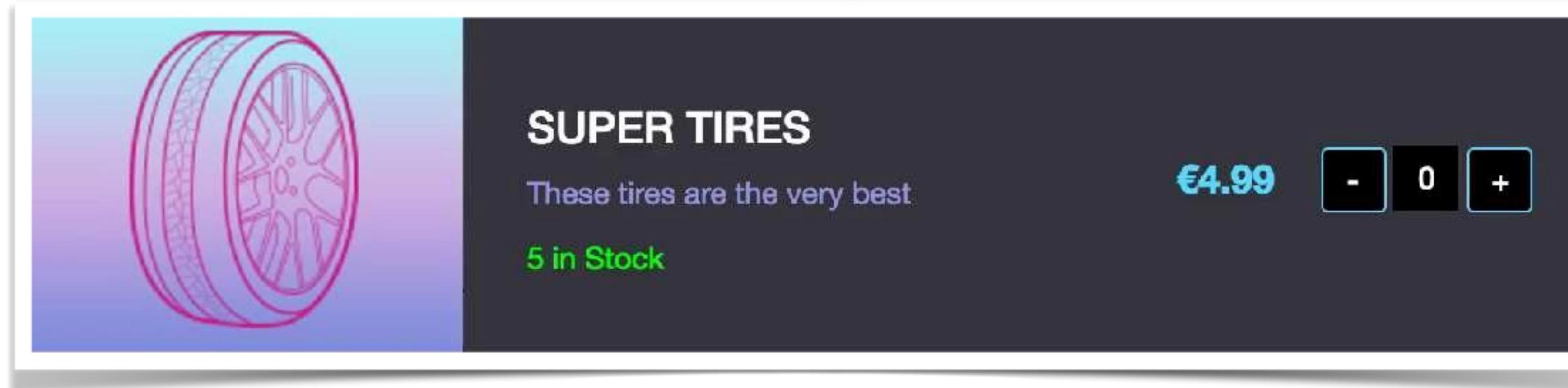
Two way binding :

Input Field 의 입력 이벤트를 감지

car-parts.component.html

HTML

```
<td class="price">{{carPart.price | currency:'EUR':true }}</td>
<td>
  <div class="select-quantity">
    <button class="decrease" (click)="downQuantity(carPart)">-</button>
    <input class="number" type="text" [value]="carPart.quantity"
           (input)="carPart.quantity = $event.target.value" >
    <button class="increase" (click)="upQuantity(carPart)">+</button>
  </div>
</td>
```



양방향 데이터 바인딩이 이루어짐

잘 동작되지만, 좀 더 간단한 방법을 제공한다.

Two way binding : FormsModule 임포트

FormsModule : 폼과 관련된 추가적인 기능을 제공

main.ts TypeScript

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

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

...

FormsModule 임포트

전체 앱에 폼과 관련된 기능을 제공

Two way binding : ngModel 사용

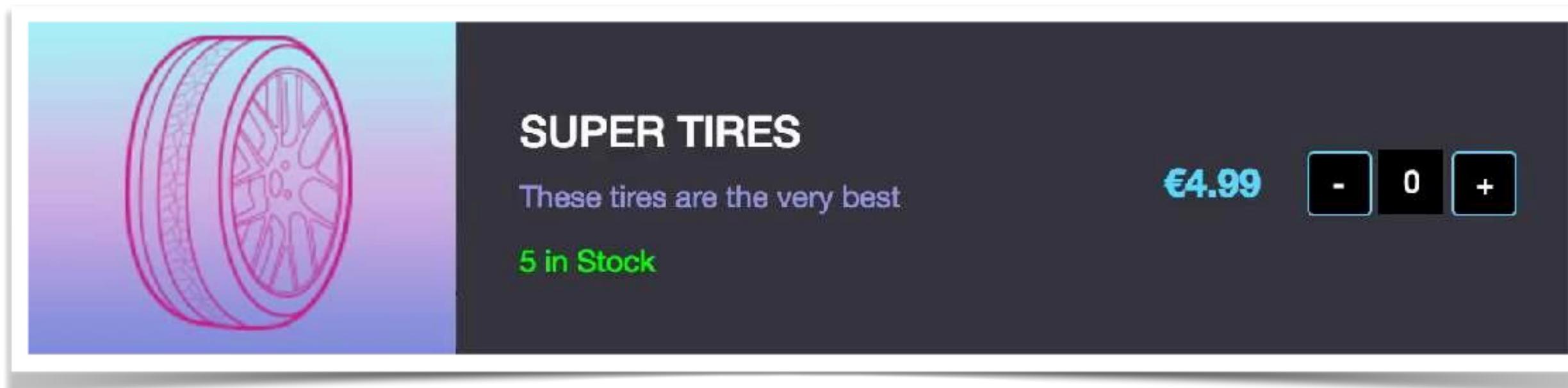
ngModel : **two way** 데이터 바인딩을 지원

car-parts.component.html

HTML

```
<td class="price">{{carPart.price | currency:'EUR':true }}</td>
<td>
  <div class="select-quantity">
    <button class="decrease" (click)="downQuantity(carPart)">-</button>
    <input class="number" type="text" [(ngModel)]="carPart.quantity">
    <button class="increase" (click)="upQuantity(carPart)">+</button>
```

대괄호 속에 괄호가 들어가는 문법에 주의! [()]



이 문법은 “banana in a box”라고 불린다.

Two way binding : ngModel 문법

```
[ (ngModel) ]=<component property>
```

component properties:

```
[ (ngModel) ]="user.age"
```



```
[ (ngModel) ]="firstName"
```



에러 발생:

```
[ (ngModel) ]="fullName()"
```



이번 장 정리

- [(ngModel)] 문법은 컴포넌트 속성과 폼 엘레먼트의 양방향 바인딩을 지원한다.
- 양방향 바인딩이란 property binding과 event binding의 조합을 말하며, Javascript 컴포넌트 내에서의 property가 변경되거나 HTML 페이지 내의 값이 변경되면 즉시 동기화 됨을 의미한다.

◎ 학습지식 개요/요점

서비스 모듈을 별도로 분리하고 http 를 통해 웹에서 데이터를 얻어 옵니다.

◎ 실습 예제 및 수행가이드

- 서비스 객체의 분리와 의존성 주입을 실습
- HTTP 모듈을 활용해 서버를 연동하여 데이터 전송

Services

서비스 클래스 작성
DI (Dependency Injection)

Mock Data 다시보기

car-parts.component.ts

```
import { Component } from '@angular/core';
import { CarPart } from './car-part';
import { CARPARTS } from './mocks';  
    ←  
...  
})  
export class CarPartsComponent {  
  carParts: CarPart[];  
  
  ngOnInit() {  
    this.carParts = CARPARTS;  
  }  
  ...  
}
```

TypeScript

mocks.ts

```
import { CarPart } from './car-part';  
  
export const CARPARTS: CarPart[] = [  
  {  
    "id": 1,  
    "name": "Super Tires",  
    "description": "These tires are the very best",  
    "inStock": 5,  
    "price": 4.99  
  }, { ... }, { ... }];
```

TypeScript

현재까지는 CarParts 데이터인 **mock** 파일을 직접 사용했다. 이것은 좋은 방법이 아니며, 데이터를 제공 해주는 별도의 객체(서비스)를 사용하는 것이 좋다.

Service 객체를 사용해야 되는 이유

- 데이터가 필요한 모든 파일에 mock 데이터를 import 해야된다. 데이터가 변경되면 데이터를 사용하는 모든 파일을 한꺼번에 변경해야 한다.
- 실제 데이터와 mock 데이터를 스위칭 하기가 어렵다.
- 데이터를 직접 호출하여, 공급 해주는 Service 객체가 필요하다.

car-parts.component.ts

TypeScript

```
import { Component } from '@angular/core';
import { CarPart } from './car-part';
import { CARPARTS } from './mocks';

...
}

export class CarPartsComponent {
  carParts: CarPart[];

  ngOnInit() {
    this.carParts = CARPARTS;
  }

  ...
}
```

Service 객체



car-parts.component.ts



데이터 요청을 서비스 객체에 함



racing-data.service.ts



실제 데이터를 가져옴



mocks.ts

먼저, 간단한 서비스를 만들고 dependency injection 기능을 추가

Service 객체

racing-data.service.ts

TypeScript

```
import { CARPARTS } from './mocks';

export class RacingDataService {
  getCarParts() {
    return CARPARTS;
  }
}
```



데이터를 분리



RacingDataService 클래스로 서비스를 객체를 생성.



데이터가 필요할 때마다 새로운 RacingDataService 객체를 생성해야 한다.



실제 데이터와 테스트용 데이터의 교체가 어렵다.

car-parts.component.ts

TypeScript

```
import { RacingDataService } from './racing-data.service';
...
export class CarPartsComponent {
  carParts: CarPart[];

  ngOnInit() {
    let racingDataService = new RacingDataService();
    this.carParts = racingDataService.getCarParts();
  }
}
```



Dependency Injection 란?

의존성 주입(Dependency Injection, DI)

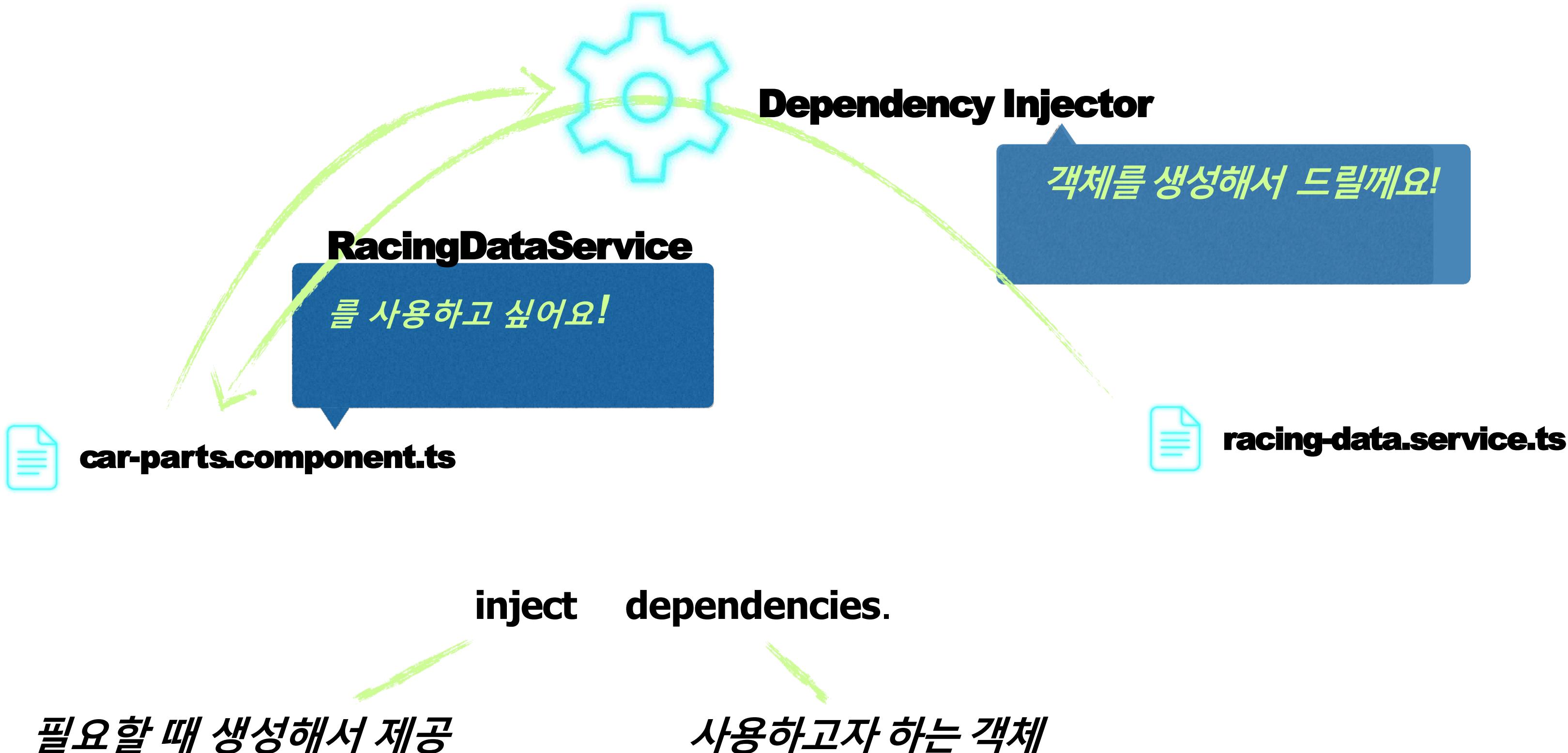
- 프로그래밍에서 구성요소간의 의존 관계가 소스코드 내부가 아닌 외부의 설정파일 등을 통해 정의되게 하는 디자인 패턴 중의 하나이다.

의존성 주입(Dependency Injection, DI)의 장점

- ① 의존 관계 설정이 컴파일시점이 아닌 실행시점에 이루어져 모듈들간의 결합도를 낮출 수 있다.
- ② 코드 재사용을 높여서 작성된 모듈을 여러 곳에서 소스코드의 수정 없이 사용할 수 있다.
- ③ **mock** 객체 등을 이용한 단위 테스트의 편의성을 높여준다.

angular와 Dependency Injection 란?

angular2 app을 실행하면 **Dependency Injector**가 생성된다.
Injector는 자동으로 객체를 생성해서 필요한 곳에 공급한다.



Dependency Injection 사용 방법

컴포넌트 “**providers**”에 등록되어 사용.

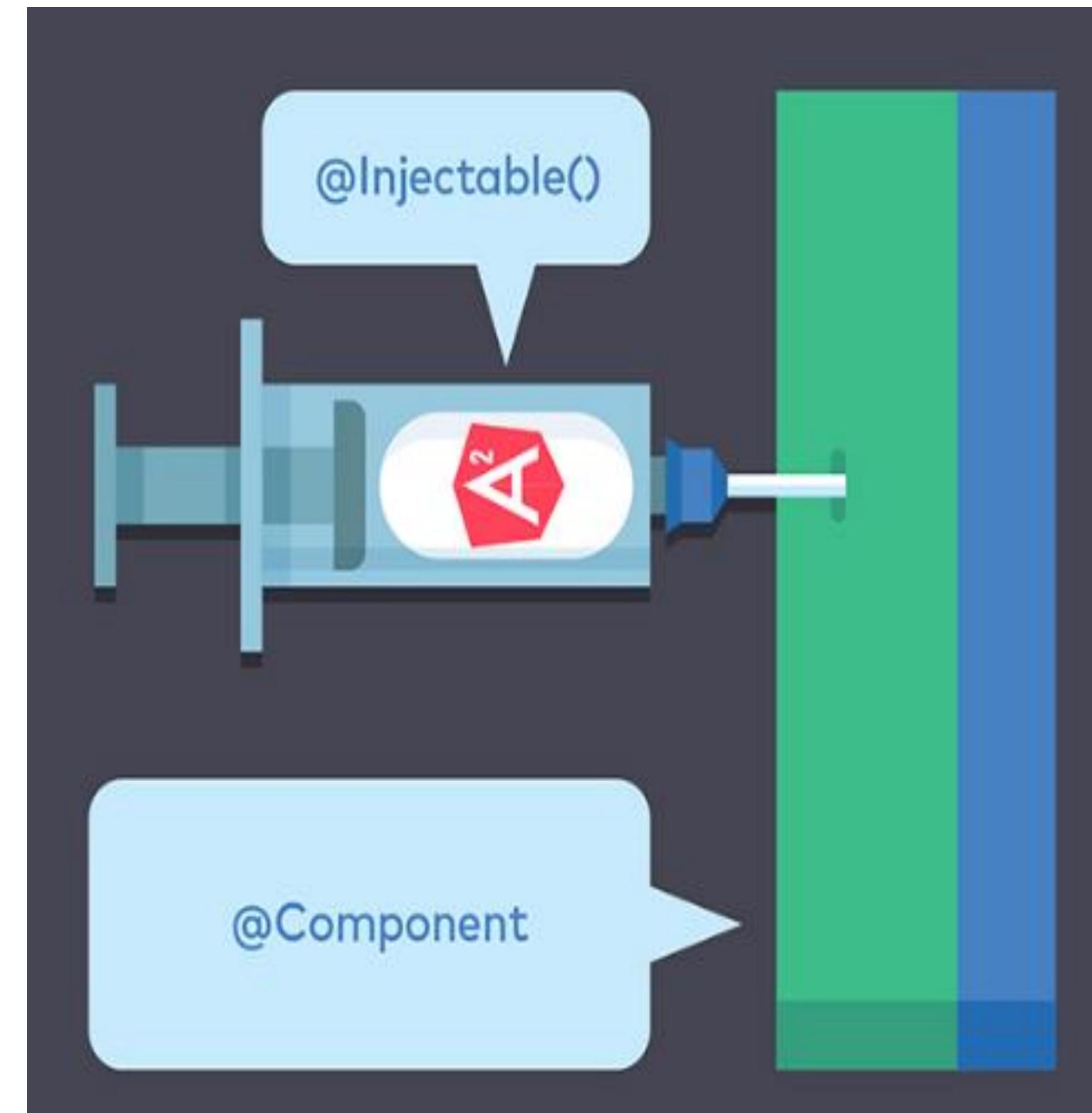
서비스 객체 — 주로 *Injector*로 사용된다.

Dependency Injector



RacingDataService 사용하기 위한 세 가지 절차 :

1. RacingDataService에 **@Injectable** 데코레이터를 추가.
2. app 모듈에 providers로 등록.
3. car-parts.component.ts에 의존성을 주입.



Step 1 : Injectable Decorator 추가

서비스 클래스를 **dependency Injector**로 사용하기 위해 **@Injectable** 데코레이터를 사용

racing-data.service.ts **TypeScript**

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

@Injectable()
export class RacingDataService {
  getCarParts() {
    return CARPARTS;
  }
}
```

괄호를 붙여야 함!



Step 2: app 모듈에 providers 등록

AppModule 의 **providers** 에 등록.

providers: 주로 싱글턴으로 생성되어 사용되며, 다른 객체들에게 의존성 주입으로 제공되는 목록

main.ts

TypeScript

```
...
import { RacingDataService } from './racing-data.service';

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

...
```

Step 3 : 의존성 주입(Injecting Dependency)

car-parts.component.ts **TypeScript**

```
...
import { RacingDataService } from './racing-data.service';

@Component({ ... })
export class CarPartsComponent {
  carParts: CarPart[];

  constructor(private racingDataService: RacingDataService) { }
}
```



타입스크립트의 파라미터로 전달하면 자동으로 객체가 생성된다.

-> 생성된 자바스크립트:

```
function CarPartsComponent(racingDataService) {
  this.racingDataService = racingDataService;
```

서비스를 통해 carParts 데이터 전달 받기

car-parts.component.ts

TypeScript

```
...
import { RacingDataService } from './racing-data.service';

@Component({ ... })
export class CarPartsComponent {
  carParts: CarPart[];

  constructor(private racingDataService: RacingDataService) { }

  ngOnInit() {
    this.carParts = this.racingDataService.getCarParts();
  }
}
```

확장성이 좋아지고 테스트가 더욱 쉬워졌다.



Scalable



Testable

이번 장 정리

- 서비스 객체는 App을 조금 더 구조적으로 확장성있게 해주는 역할을 한다. 그리고 데이터 액세스 메소드를 제공한다.
- 서비스 객체는 의존성 주입(DI)을 통해 필요한 클래스에 제공한다.
- 클래스의 생성자에서 의존성 주입 (DI) 을 수행한다.

http 모듈

웹에 연결하여 데이터를 가져오자!

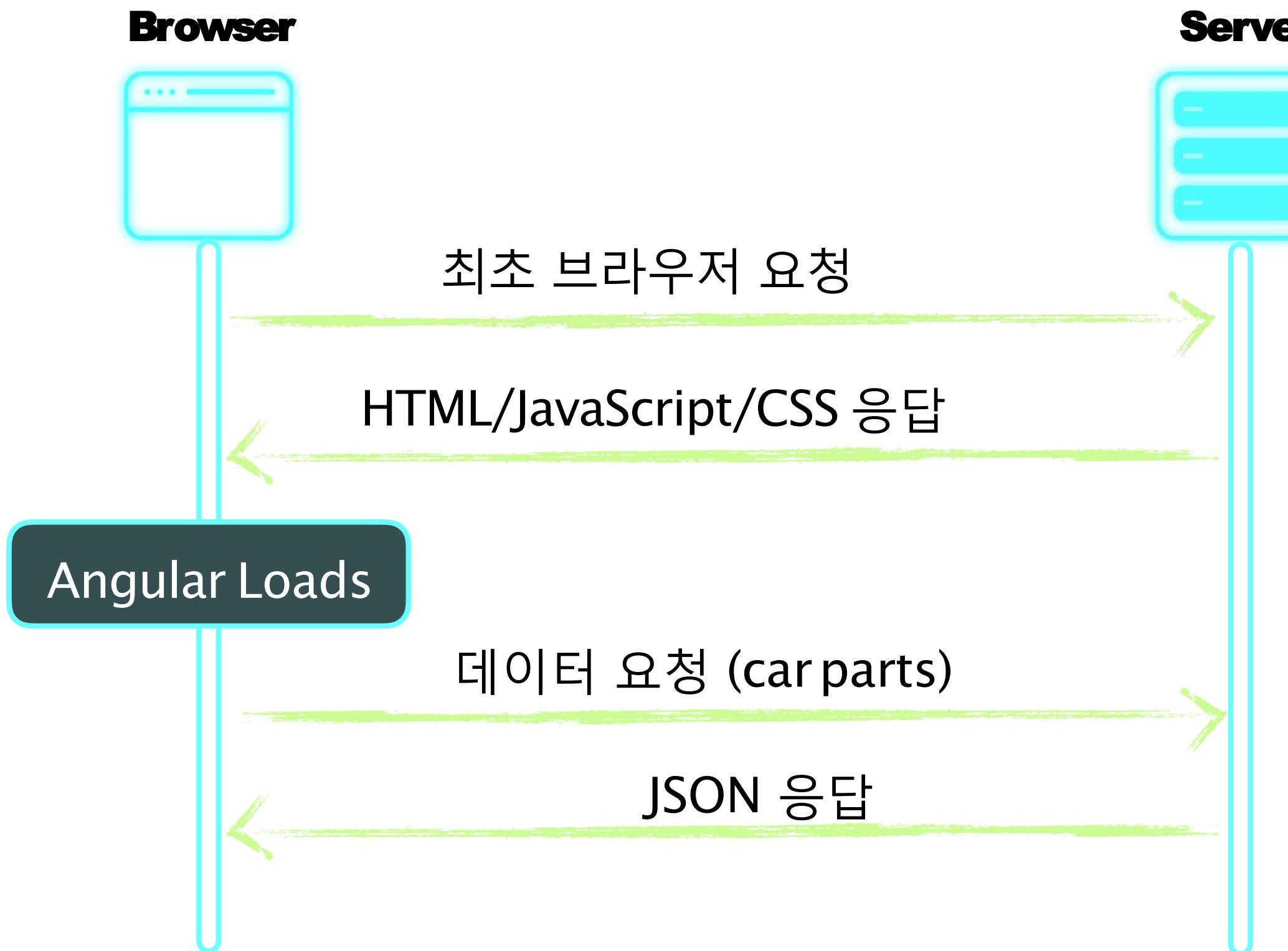
인터넷으로 데이터 요청!

데이터를 **mock** 데이터로 가져오는 것 대신에 인터넷으로 통해 데이터를 가져오는 방법에 대해서 알아보자.



서버에 데이터 요청

앵귤러 앱이 로딩되면 최초로 페이지가 로딩된다. 다음 부터는 **Ajax**로 서버요청을 해서 데이터만 가져와서 로딩한다.



HTTP 라이브러리 사용 절차

1. car part JSON 파일을 작성



car-parts.json

2. Http 요청을 위해 필요한 라이브러리를 추가 (http & RxJS)



app.component.ts

3. http 서비스를 사용하기 위해 module import



racing-data.service.ts

4. http 를 사용하기 위해서 서비스에 의존성 주입(주로 생성자를 통해서 주입)



car-parts.component.ts

5. 요청에 따라 응답을 받아 처리

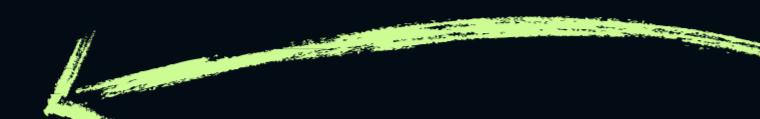
Step 1 : JSON 파일 작성

서비스 요청으로 전달 되는 **JSON** 파일을 생성해야 함

car-parts.json **JSON**

```
{  
  "data": [  
    {  
      "id": 1,  
      "name": "Super Tires",  
      "description": "These tires are the very best",  
      "inStock": 5,  
      "price": 4.99,  
      "image": "/images/tire.jpg",  
      "featured": false,  
      "quantity": 0  
    },  
    { ... },  
    { ... }  
  ]  
}
```

기존 carpart 배열 데이터를 JSON 형태로 변환

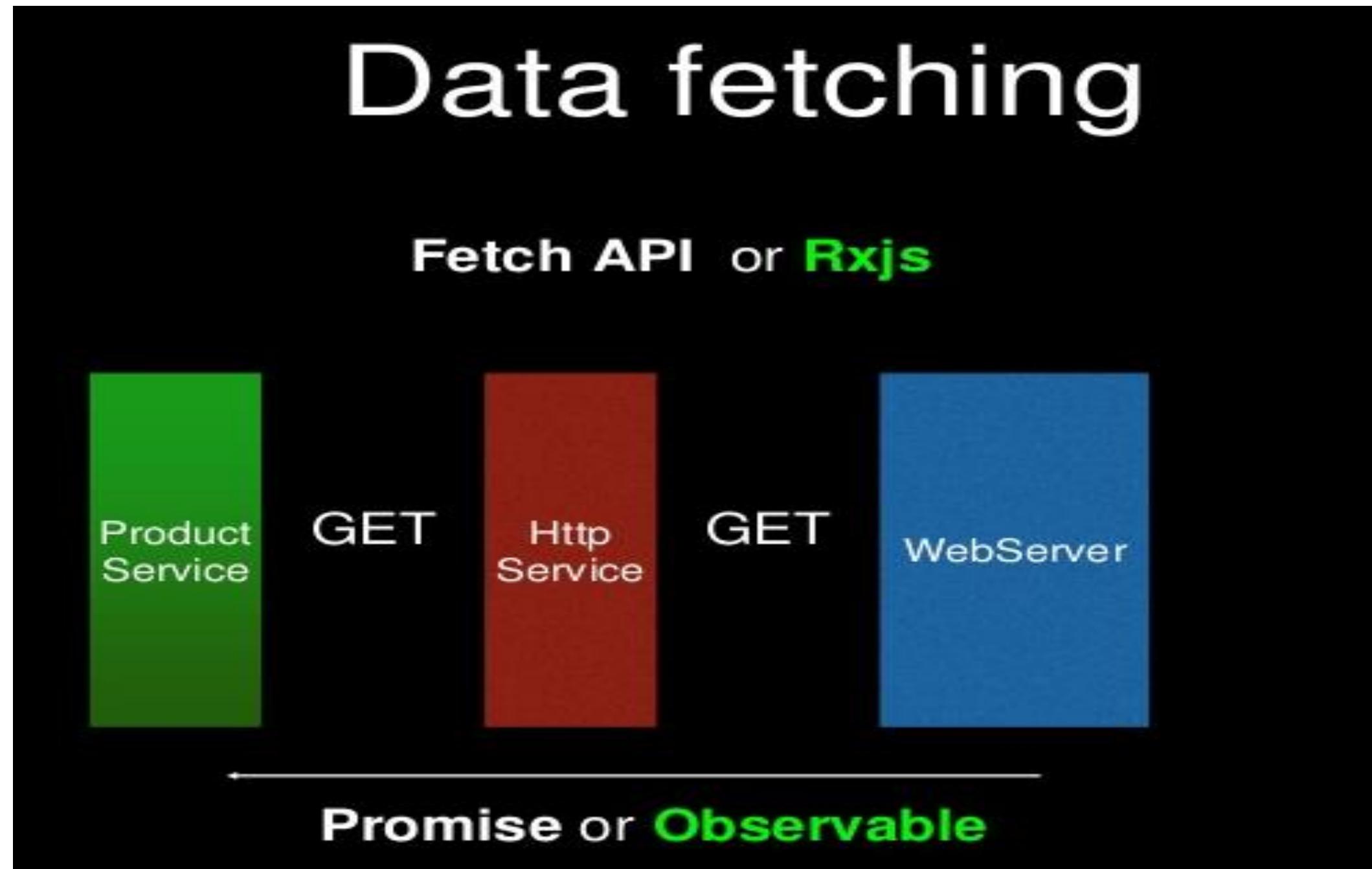


Step 2 : HTTP 와 RxJS 라이브러리 포함하기

HTTP 라이브러리는 원격서버에 접속하여 데이터를 주고 받기위한 라이브러리이다.

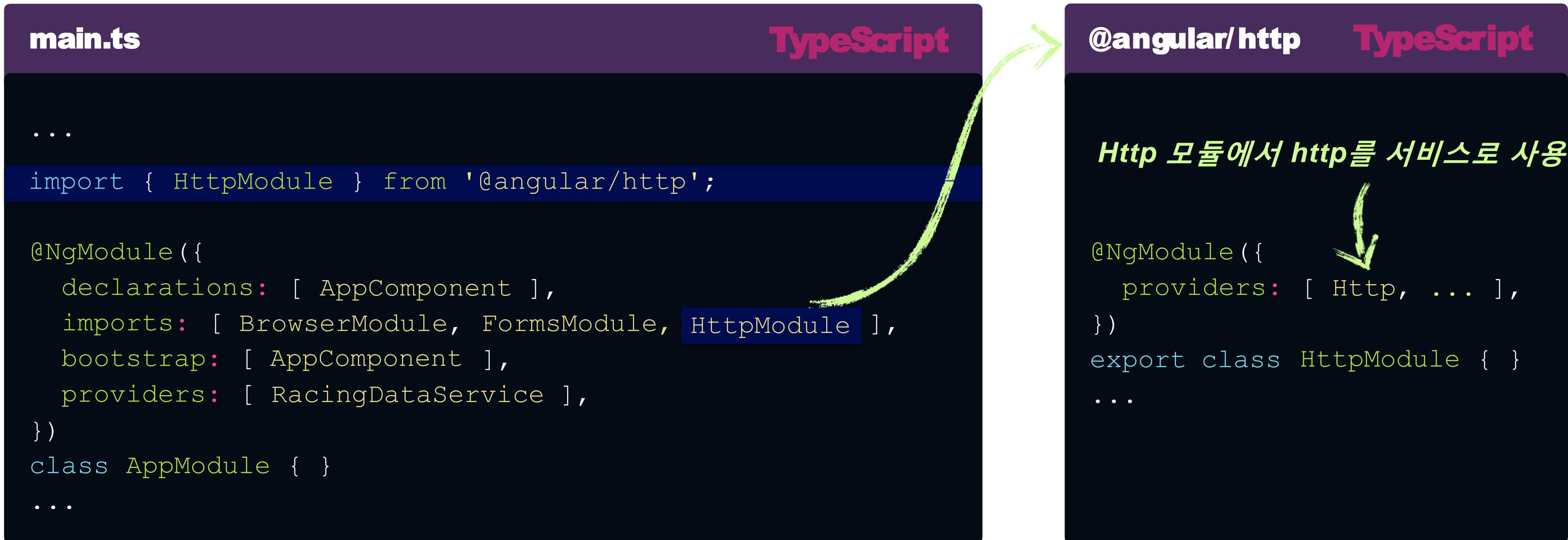
RxJS 라이브러리는 **Reactive Extensions** 의 약자이고, **http** 요청에 대한 여러가지 기능을 제공한다.

Angular **http**는 **RxJS**의 **Observable**을 사용한다.



Step 3 : HTTP 서비스를 사용하기 위해 module import

Http 서비스를 사용하기 위해 main.ts에서 import



이제 HTTP 라이브러리를 필요한 곳에서 사용할 수 있다.

Step 4 : HTTP 의존성 주입과 사용

http 를 객체를 사용해서 인터넷으로 요청 수행

racing-data.service.ts **TypeScript**

```
import { Injectable } from '@angular/core';
import { CarPart } from './car-part';
import { Http } from '@angular/http';
import 'rxjs/add/operator/map';

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

    getCarParts() {
        return this.http.get('app/car-parts.json')
            .map(response => <CarPart[]>response.json().data);
    }
}
```

HTTP 의존성 주입

get 요청 처리 자세히 알아보기

http의 get 요청은 observable 타입의 객체를 리턴한다.

Observable 객체는 http 요청에 대한 다양한 기능을 제공 — 그 중 하나는 리턴값을 배열로 돌려준다.

```
getCarParts () {  
    return this.http.get('app/car-parts.json')  
        .map(response =>  
            <CarPart []>  
            response.json()  
            .data);  
}
```

반환되는 데이터가
저장되는 변수

반환되는 데이터 타입이
CarPart 배열

응답을 JSON 형태로 변환

JSON에서 반환받을
데이터

- Reactive 프로그래밍 참조

<http://mobicon.tistory.com/467>

<https://medium.com/@andrestaltz/2-minute-introduction-to-rx-24c8ca793877#.kqx80ohbj>

Step 5: 응답을 처리

요청을 통해 **observable** 타입의 객체를 응답
서버로 부터 전달된 데이터 스트림을 처리

car-parts.component.ts **TypeScript**

```
...
export class CarPartsComponent {

    constructor(private racingDataService: RacingDataService) { }

    ngOnInit() {
        this.racingDataService.getCarParts()
            .subscribe(carParts => this.carParts = carParts);
    }
    ...
}
```



서버로 부터 전달된 carParts 데이터 스트림을 로컬 carParts 배열에 저장

Undefined carParts

carParts 가 배열 인지를 체크하고 코드를 진행

 car-parts.component.ts TypeScript

```
totalCarParts() {
  let sum = 0;

  if (Array.isArray(this.carParts)) {
    for (let carPart of this.carParts) {
      sum += carPart.inStock;
    }
  }
  return sum;
}
```

페이지가 최초로 Load 되었을 때는 데이터를 아직 가져오지 못해 carParts 가 **undefined** 상태.

이번 장 정리

- Angular 앱은 초기화되고 실행된 이후 서비스를 사용해서 데이터를 로딩 한다.
- HTTP 라이브러리는 의존성 주입을 통해 사용한다.
- http 호출은 promise가 아니라 observable 객체를 반환한다. observable 객체는 배열과 매우 유사하다.

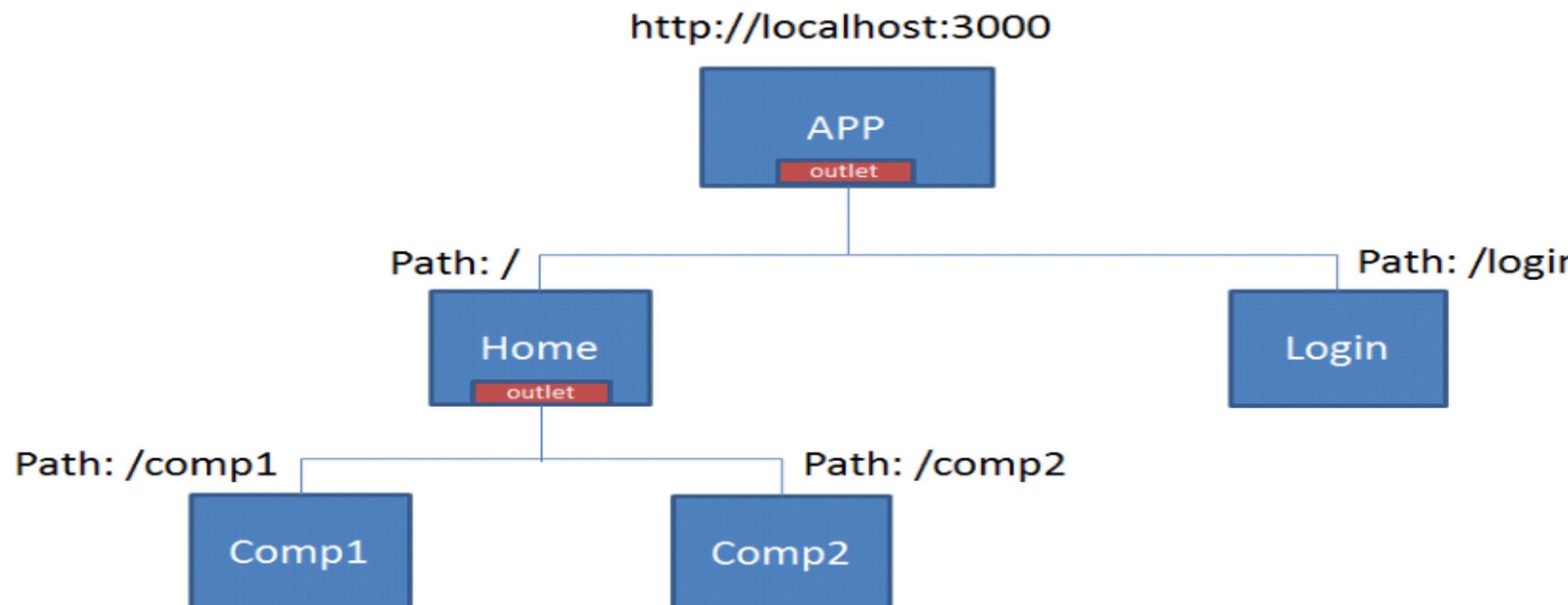
Router 모듈

routing

Router 개요

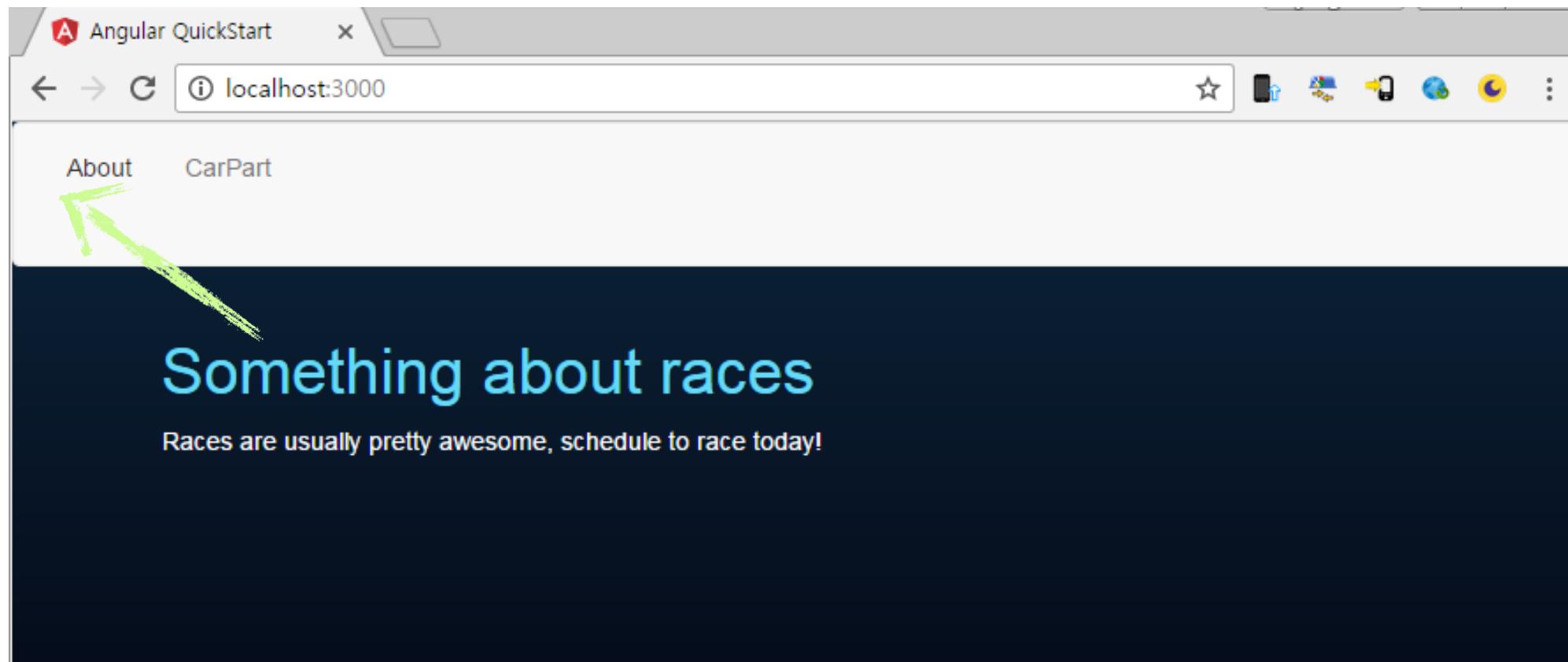
- **Router 란?**

- : URI Hash 값을 이용하여 화면의 Navigation을 지원하는 기능이다.
- : Router를 사용할 경우 구분된 여러 개의 Component를 교체하면서 화면에 보여줄 수 있다.

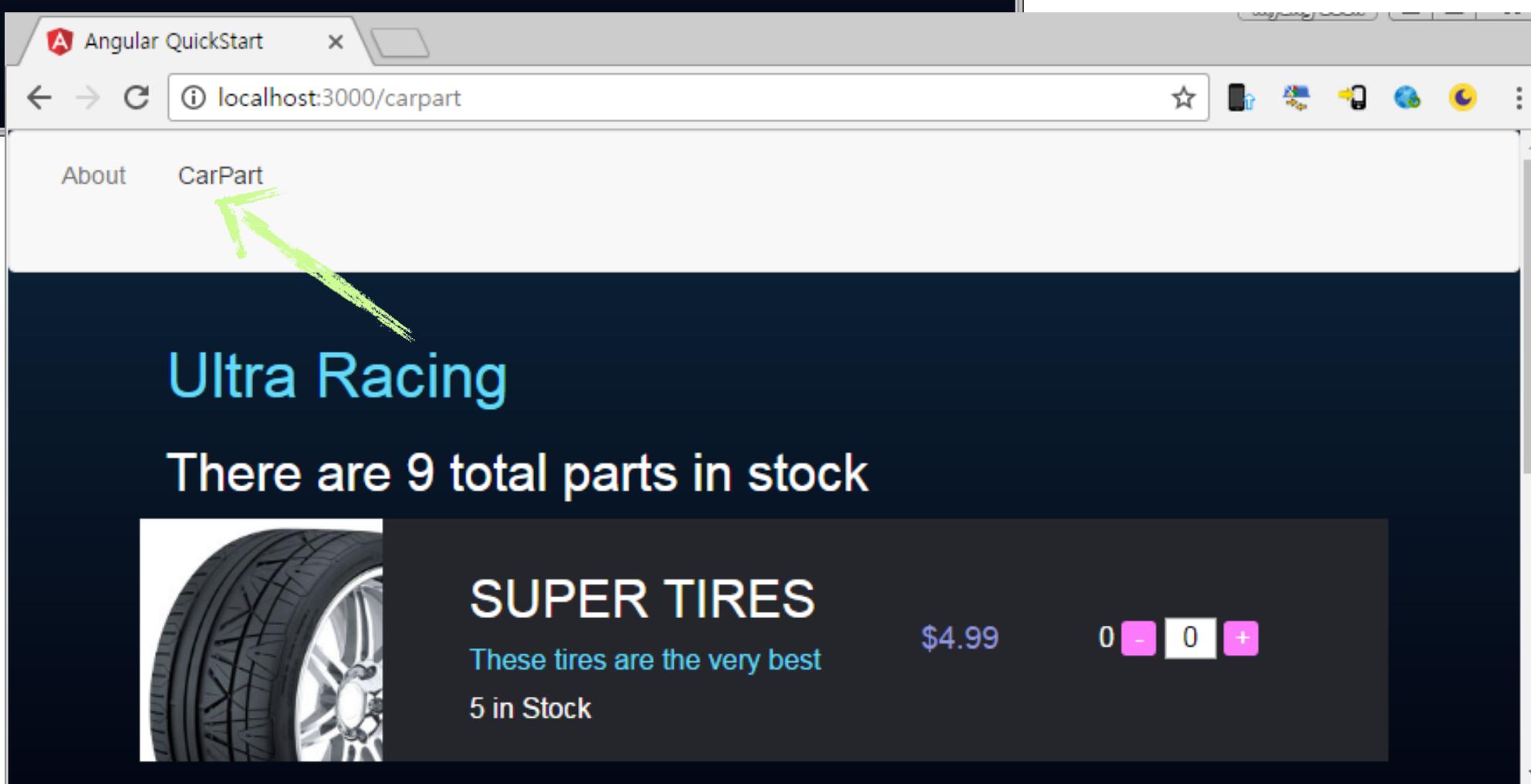


Router 적용

About 메뉴 선택



CarPart 메뉴 선택



Router 설정

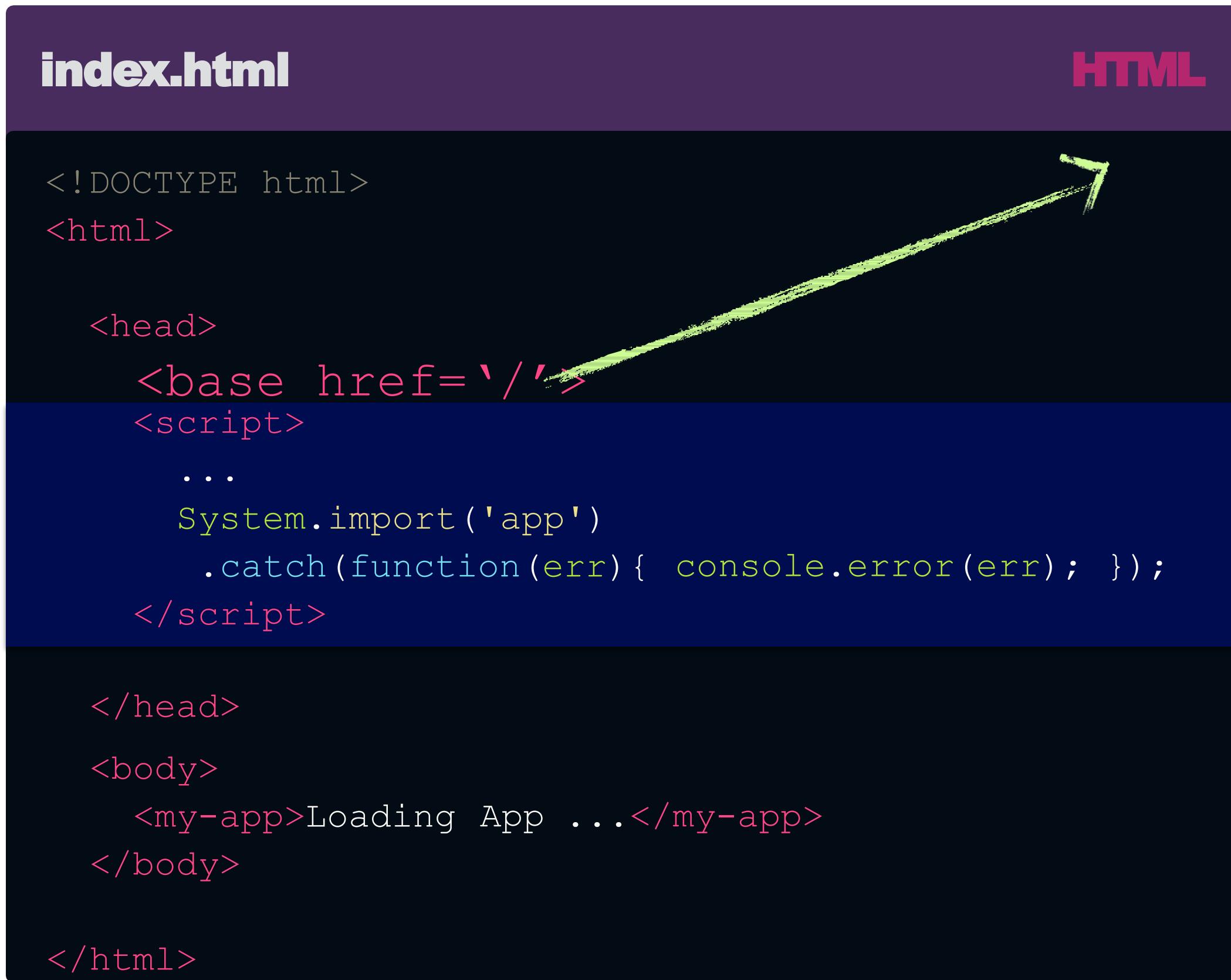
- Router 설정
 - : Router 설정은 Routes를 이용하여 분기될 Component를 설정한다.
 - : Routes에서 설정하는 Component들은 해당 module에서 <router-outlet> 위치에서 동작한다.

```
app.routes.ts x
1 import { ModuleWithProviders }      from '@angular/core';
2 import { Routes, RouterModule } from '@angular/router';
3 import { CarPartComponent } from './car-part/car-part.component';
4 import { AboutComponent } from './about.component';
5
6 export const routes: Routes = [
7   { path: '', component: AboutComponent },
8   { path: 'carpart', component: CarPartComponent }
9 ]
10
11 export const AppRoutingModule:ModuleWithProviders = RouterModule.forRoot(routes);
```

```
app.component.ts x
1 import {Component} from "@angular/core";
2
3 @Component({
4   selector: 'my-app',
5   template: `
6     <nav class="navbar navbar-default">
7       <div class="container-fluid">
8         <div class="navbar-header">
9           <ul class="nav navbar-nav navbar-right">
10             <li><a [routerLink]="">About</a></li>
11             <li><a [routerLink]="/carpart">CarPart</a></li>
12           </ul>
13         </div>
14       </div>
15     </nav>
16     <main role="main">
17       <router-outlet></router-outlet>
18     </main>
19   `
20 })
21 export class AppComponent { }
```

Step 1 : index.html 수정

상단에 <base> Tag를 추가한다.



```
index.html
```

HTML

```
<!DOCTYPE html>
<html>

  <head>
    <base href='/'>
    <script>
      ...
      System.import('app')
        .catch(function(err) { console.error(err); });
    </script>

  </head>
  <body>
    <my-app>Loading App ...</my-app>
  </body>

</html>
```

Router는 URI Hash 값을 이용하므로 root 경로를 설정하여 최상위 경로 설정을 지정할 수 있다.

Step 2 : AboutComponent 작성

추가적인 **Component** 작성

about.component.ts

TypeScript

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

@Component({
  selector: 'about',
  template: `
    <header class="container">
      <h1>Something about races</h1>

      <p>Races are usually pretty awesome, schedule to race today!</p>
    </header>
  `

})
export class AboutComponent { }
```

Step 3 : Routing Module 작성 (app-routing.module.ts)

app.routes.ts

TypeScript

```
import { NgModule }      from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { CarPartComponent } from './car-part/car-part.component';
import { AboutComponent } from './about.component';

export const routes: Routes = [
  { path: '', component: AboutComponent },
  { path: 'carpart', component: CarPartComponent }
];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Router를 구성하려면 설정을 필요함

- : 1. Router 설정을 위해 필요한 라이브러리를 import 한다.
- : 2. Router는 path와 component를 맵핑 함으로써 Hash 값과 Component가 매핑 된다.

Step 4 : main.ts 수정

main.ts

TypeScript

```
.....  
import { AboutComponent } from './about.component';  
import { AppRoutingModule } from './app.routes';  
  
@NgModule({  
  declarations: [  
    AppComponent,  
    AboutComponent,  
    CarPartComponent  
,  
  imports: [ BrowserModule,FormsModule,HttpModule,AppRoutingModule ],  
  bootstrap: [ AppComponent ],  
  providers: [ RacingDataService ]  
})  
class AppModule {}  
  
platformBrowserDynamic().bootstrapModule(AppModule);
```

1. 추가적으로 작성한 AboutComponent를 module에 선언한다.
2. AppRoutingModule을 import 한다.

Step 5 : app.component.ts 수정

app.component.ts

TypeScript

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

@Component({
  selector: 'my-app',
  template: `
    <ul class="nav navbar-nav navbar-right">
      <li><a [routerLink]="">About</a></li>
      <li><a [routerLink]="/carpart">CarPart</a></li>
    </ul>
    <main role="main">
      <router-outlet></router-outlet>
    </main>
  `
})
export class AppComponent {}
```

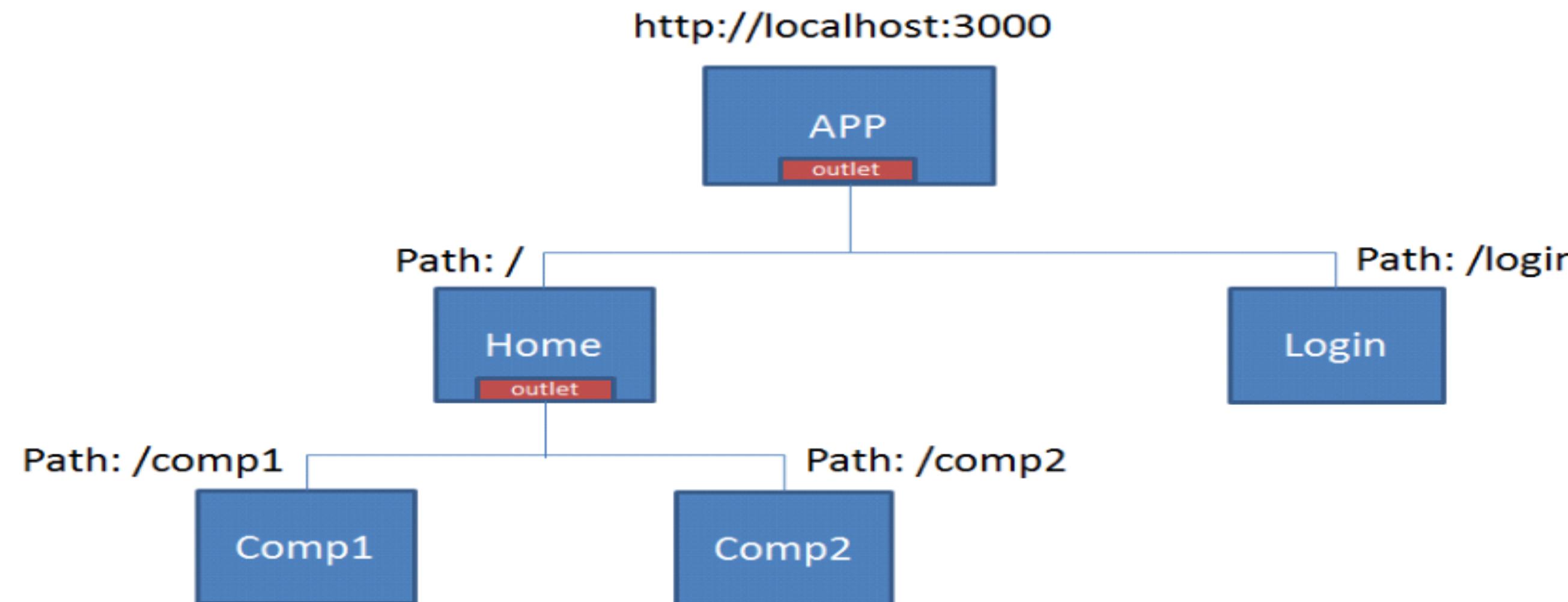
1. Click할 때 Router 설정을 위한 영역을 처리하기 위해 **[routerLink]**를 설정한다.
2. Router로 설정된 Component는 **<router-outlet></router-outlet>** 태그 안에 위치하게 된다.

Router 모듈

Child - Routing

Child Router 개요

- Route 된 페이지 내부에 다른 Component를 보여줄 때 사용함.



CarPartDetailComponent 생성

- CarPart 상세정보를 보여주기 위한 컴포넌트 생성

car-part-detail.component.ts

TypeScript

```
import { Component } from '@angular/core';
import { CarPart } from './car-part';
import { RacingDataService } from '../racing-data.service';
import { ActivatedRoute, Router} from "@angular/router";

@Component({
  selector: 'car-part-detail',
  ...
})
export class CarPartDetailComponent {
  carPart: CarPart = {
    "id": 0,
    ...
    "quantity": 0 컴포넌트 라우팅 파라미터를 받아옴 routeLink에 의해 호출될 수 있도록 함.
  };
}

constructor ( private route: ActivatedRoute, private router:Router, private racingDataService:RacingDataService ) {}
```



CarPartDetailComponent 생성

- CarPart 상세정보를 보여주기 위한 컴포넌트 생성

car-part-detail.component.ts

TypeScript

```
ngOnInit() {
  this.route.params.subscribe(params => {
    if ( params['id'] != null ) {
      this.racingDataService.getCarParts().subscribe(carParts => {
        carParts.forEach( (data) => {
          if(data.id == params['id']) {
            this.carPart = data;
          }
        });
      });
    }
  });
}
```



컴포넌트 라우팅 파라미터를 받아옴

CarPartDetailComponent 템플릿 생성

- CarPart 상세정보를 보여주기 위한 html 템플릿 생성

car-part-detail.component.html

TypeScript

```
<ul>
  <li class="card">
    <div class="panel-body">
      <div class="photo">
        <img [src]="carPart.image"/>
      </div>
      <table class="product-info">
        <tr>
          <td>
            <h2>{{ carPart.name | uppercase }}</h2>
            <p class="description">{{carPart.description}}</p>
            <p class="inventory" *ngIf="carPart.inStock > 0">{{carPart.inStock}} in Stock</p>
            <p class="inventory" *ngIf="carPart.inStock === 0">Out of Stock</p>
          </td>
          <td class="price">{{carPart.price | currency:'EUR':true}}</td>
        </tr>
      </table>
    </div>
  </li>
</ul>
```

routerLink 설정

- CarPart 상세정보를 보여주기 위한 html 템플릿 생성

car-part.component.html

TypeScript

```
<ul>
  <li class="card" *ngFor="let carPart of carParts" [class.featured]="carPart.featured">
    ...
    <h2><a [routerLink]=[ './car-part-detail', carPart.id]>{{ carPart.name | uppercase }}</a></h2>
    ...
  </li>
</ul>
<router-outlet></router-outlet>
```

[routerLink]에 URL과 Parameter 를 함께 전달하려면
"[routerLink]"="['URL', Parameter]" 로 사용한다.

car-part Router 생성

- Routing 을 위한 설정을 합니다.

car-part.routes.ts

TypeScript

```
import { ModuleWithProviders } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { CarPartsComponent } from './car-part.component';
import { CarPartDetailComponent } from './car-part-detail.component';

export const CAR_PART_ROUTES: Routes = [
  {
    path: 'carpart',
    component: CarPartsComponent,
    children: [
      { path: 'car-part-detail/:id', component: CarPartDetailComponent }
    ]
  }
];

export const CarPartRoutingModule:ModuleWithProviders = RouterModule.forChild(CAR_PART_ROUTES);
```

car-part Module 생성

- Detail 을 한 묶음으로 만들어 모듈로 관리합니다.

car-part.module.ts

TypeScript

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { CarPartRoutingModule } from './car-part.routes';
import { CarPartDetailComponent } from './car-part-detail.component';

@NgModule({
  imports: [BrowserModule, CarPartRoutingModule],
  declarations: [ CarPartDetailComponent ]
})
export class CarPartModule { }
```

CarPartModule 을 App에 추가합니다.

main.ts

TypeScript

```
...
import { CarPartModule } from "./car-part.module";

@NgModule({
  ...
  imports: [..., CarPartModule],
  ...
})
class AppModule {}

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

Angular Form

GuestBookComponent 추가

방명록을 만들기 위한 component 생성

guest-book.component.ts

TypeScript

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

@Component({
  selector: 'guest-book',
  templateUrl: 'app/guestbook/guest-book.component.html'
})
export class GuestBookComponent {

  constructor ( private route: ActivatedRoute, private router:Router ) { }

  onSubmit(form) {
    console.log(form.value);
  }
}
```

GuestBookComponent template 추가

방명록을 만들기 위한 component 생성

guest-book.component.html

TypeScript

```
<h2>Guest Book</h2>
<form #f="ngForm" (ngSubmit)="onSubmit(f)">
  <label for="id">아이디</label>
  <input type="text" name="id" ngModel #id required/><br/>

  <label for="password">비밀번호</label>
  <input type="password" name="password" ngModel #password required/><br/>

  <label for="desc">남길 말</label>
  <input type="text" name="desc" ngModel #desc required/><br/>

  <button type="submit">등록</button><br/><br/>

  <div>
    <div [innerText]="'guestBookID : ' + id.value"></div>
    <div [innerText]="'Password : ' + password.value"></div>
    <div [innerText]="'Desc : ' + desc.value"></div>
  </div>
</form>
```

#f로 ngForm 을 선언함.

현재 Template에서 #f
로 ngForm 객체에 접근
가능함

각 input 들을 ngModel
로 선언하고 폼의 속성
으로 지정한다. (#id,
#password, #desc)

GuestBookComponent Module 추가

방명록을 만들기 위한 Module 생성

guest-book.module.ts

TypeScript

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

@NgModule({
  imports: [BrowserModule, FormsModule],
  declarations: [ GuestBookComponent ]
})
export class GuestBookModule { }
```

Template에서 ngForm을 사용할 수 있도록 @angular/forms를 임포트.

방명록 주소를 Root Router에 추가

방명록을 만들기 위한 Module 생성

app.routes.ts

TypeScript

```
...
import { GuestBookComponent } from "./guest-book.component";

export const ROUTES: Routes = [
  ...
  { path: 'guestbook', component: GuestBookComponent }
];
...
...
```

Router 메뉴 추가

방명록을 만들기 위한 Module생성

app.component.ts

TypeScript

```
...
<ul class="nav navbar-nav navbar-right">
  <li>
    <a [routerLink]=[""]>About</a>
    <a [routerLink]=["/carpart"]>CarPart</a>
    <a [routerLink]=["/guestbook"]>Guestbook</a>
  </li>
</ul>
...
...
```

Angular Application 에 방명록 등록

방명록을 만들기 위한 Module생성

main.ts

TypeScript

```
...
import { GuestBookModule } from "./guest-book.module";
...
...
imports: [...GuestBookModule],
...
```

Angular Form

Form Validation Check

Form 입력값을 검증하기 위한 설정

필수 값 검증.

guest-book.component.html

TypeScript

```
...
<input id="id" type="text" name="id" ngModel #id="ngModel" required/><br/>
<div *ngIf="id.touched" [hidden]="id.valid">아이디를 입력하세요</div>

<label for="password">비밀번호</label>
<input id="password" type="password" name="password" ngModel #password="ngModel" required/><br/>
<div *ngIf="password.touched" [hidden]="password.valid">비밀번호를 입력하세요</div>

<label for="desc">남길 말</label>
<input id="desc" type="text" name="desc" ngModel #desc="ngModel" required/><br/>
<div *ngIf="desc.touched" [hidden]="desc.valid">내용을 입력하세요</div>

...
```

폼 엘리먼트들마다 .valid로 검증을 수행한다.
입력하는 값에 따라 valid가 true/false로 리턴된다.

Angular Form

Form Validation Check
Multiple Validation Check

Form 입력값을 검증하기 위한 설정

필수 값 검증과 여러가지 검증하기

guest-book.component.html

TypeScript

```
...
<input id="password"
  type="password" name="password" ngModel #password="ngModel"
  required maxlength="3" minlength="3"/><br/>
<div *ngIf="password.touched && password.errors">

  <div *ngIf="password.errors.required">
    비밀번호를 입력하세요.
  </div>
  <div *ngIf="password.errors.minLength || password.errors.maxLength">
    비밀번호를 3자리로 입력하세요.
  </div>
</div>
```

maxlength, minlength 등을 추가해 동시 검증을 수행한다.
검증에 따른 경고알림도 각각 설정한다.

Angular Form

Form Validation Check
FormGroup / FormControl

유효성 검사 로직 분리 – FormGroup / FormControl

Form 유효성 검사 로직을 컴포넌트로 분리시키기

guest-book.component.ts

TypeScript

```
...
import { FormControl, FormGroup, Validators } from '@angular/forms';
import { } from '@angular/forms';
```

```
...
export class GuestBookComponent {
```

```
    guestForm:FormGroup = new FormGroup({
        id: new FormControl('guest', Validators.required),
        password: new FormControl('', Validators.required),
        desc: new FormControl('', Validators.required)
    });
}
```

```
...
```

FormGroup을 이용해 폼에 사용될 엘리먼트들을 정의함.
동시에, FormControl로 초기값과 검증을 설정함.
FormControl(폼 초기값, 검증로직)

FormControl로 템플릿 수정하기 – FormGroup / FormControl

Form 유효성 검사 로직을 컴포넌트로 분리시키기

guest-book.component.html

TypeScript

```
<h2>Guest Book</h2>
<form [formGroup]="guestForm" #f (ngSubmit)="onSubmit(f)">
  ...
  <input id="id" type="text" name="id" formControlName="id"/><br/>
  <div *ngIf="guestForm.controls['id'].touched" [hidden]="guestForm.controls['id'].valid">아이디를 입력하세요</div>

  <label for="password">비밀번호</label>
  <input id="password" type="password" name="password" formControlName="password"/><br/>
  <div *ngIf="guestForm.controls['password'].touched" [hidden]="guestForm.controls['password'].valid">
    비밀번호를 입력하세요</div>

  <label for="desc">남길 말</label>
  <input id="desc" type="text" name="desc" formControlName="desc" /><br/>
  <div *ngIf="guestForm.controls['desc'].touched" [hidden]="guestForm.controls['desc'].valid">
    내용을 입력하세요</div>
```

FormGroup / FormControl 을 사용하면, ngModel을 사용하지 않는다.
(FormGroup 사용)

FormModule 수정 – FormGroup / FormControl

FormGroup / FormControl 을 사용하기 위한 모듈 추가 (ReactiveFormsModule)

guest-book.module.html

TypeScript

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

@NgModule({
  imports: [BrowserModule, FormsModule, ReactiveFormsModule],
  declarations: [ GuestBookComponent ]
})
export class GuestBookModule { }
```

Angular Form

Form Validation Check
Validators.compose

동시 검증 – FormGroup / FormControl - compose

guest-book.component.ts

TypeScript

```
guestForm:FormGroup = new FormGroup({
  id: new FormControl('guest', Validators.required),
  password: new FormControl('', Validators.compose([Validators.required, Validators.maxLength(20), Validators.minLength(3)])),
  desc: new FormControl('', Validators.required)
});
```

여러 검증을 사용하기 위해 Validators.compose([...]) 를 사용함.

동시 검증 – FormGroup / FormControl - compose

FormGroup / FormControl 을 사용하기 위한 모듈 추가 (ReactiveFormsModule)

guest-book.component.html

TypeScript

```
<div *ngIf="guestForm.controls['password'].touched && guestForm.controls['password'].errors"
      [hidden]="guestForm.controls['password'].valid">
  <div *ngIf="guestForm.controls['password'].errors.required">
    비밀번호를 입력하세요
  </div>
  <div *ngIf="guestForm.controls['password'].errors.minLength">
    비밀번호를 3글자 이상 입력하세요
  </div>
  <div *ngIf="guestForm.controls['password'].errors.maxLength">
    비밀번호를 20글자 이하로 입력하세요
  </div>
</div>
```

유효성 검사별로 에러메시지를 정의한다.

Angular Form

Form Validation Check
Custom Validation Function

사용자 유효성 검증

guest-book.component.ts

TypeScript

```
guestForm: FormGroup = new FormGroup({
  id: new FormControl('guest', Validators.compose([validNumber])),
});

...
}

// export class GuestBookComponent

function validNumber(c: FormControl) {
  if (c.value == "") {
    return {
      valid: false,
      errorMsg: "필수값입니다."
    };
  }

  let NUMBER_REGEXP = /^[0-9]+$/;
  return NUMBER_REGEXP.test(c.value) ? {
    valid: true,
    errorMsg: ""
  } : {
    valid: false,
    errorMsg: '숫자가 아닙니다.'
  };
}
```

사용자 정의 검증 로직을
function 으로 정의할 수 있다.

사용자 유효성 검증

guest-book.component.html

TypeScript

```
...
<label for="id">아이디</label>
<input id="id" type="text" name="id" formControlName="id"/><br/>
<div *ngIf="guestForm.controls['id'].touched" [hidden]="guestForm.controls['id'].errors.valid">
  {{guestForm.controls['id'].errors.errorMsg}}
</div>
...
...
```

사용자 검증 로직에서 사용한 속성을 이용해 검증을 처리할 수 있다.

감사합니다.