کامپایلر جلوتر از زمان (AOT)

برنامه‌های Angular اساساً شامل کامپوننت ها و قالب‌های HTML مربوط به آن‌ها هستند. با توجه به اینکه به صورت مستقیم مرورگر نمی‌تواند این کامپوننت ها و قالب‌های ارائه شده توسط Angular را درک کند، این برنامه‌ها قبل از اجرا شدن در یکی از مرورگرها به فرایند کامپایل نیاز دارند.

کامپایلر جلوتر از زمان (AOT) طی مرحله‌ی ساخت و قبل از آنکه مرورگر کد جاوا اسکریپت را دانلود و اجرا کند، کد تایپ اسکریپت و Angular HTML شما را به یک کد کارامد جاوا اسکریپت تبدیل می‌کند. کامپایل کردن برنامه طی فرایند ساخت باعث می‌شود که برنامه‌ی تان سریع‌تر در مرورگر رندر شود.

در این آموزش توضیح خواهیم داد که برای آنکه بتوانید با استفاده از کامپایلر AOT با بازدهی بیشتری برنامه‌های خود را کامپایل کنید، چگونه می‌توانید متادیتا را مشخص کنید و امکانات موجود در کامپایلر را به کار بگیرید.

توضیحات نویسنده‌ی این کامپایلر آقای Tobias Bosch را در رابطه با کامپایلر Angular در AngularConnect 2016 تماشا کنید.

کامپایل در Angular

Angular دو روش زیر برای کامپایل کردن برنامه‌هایتان را ارائه می‌کند:

1. درجا (JIT) که در زمان اجرا برنامه‌ی شما را در مرورگر کامپایل می‌کند.
2. جلوتر از زمان (AOT) که در زمان ساخت، برنامه‌ی شما را کامپایل می‌کند.

در زمان اجرای دستورات build-and-serve-locally و build-only در CLI عمل کامپایل به صورت پیشفرض به شکل JIT انجام می‌شود:

```
ng build
ng serve
```

برای انجام کامپایل به صورت AOT، پرچم‌های aot-- را به دستورات بالا اضافه کنید:

```
ng build --aot
ng serve –aot
```

دستور ng build به همراه پرچم prod– باعث می‌شود که کار کامپایل به صورت پیشفرض به شکل AOT انجام شود. برای جزییات بیشتر به آموزش CLI مخصوصاً عنوان build آن مراجعه کنید.

چرا باید به صورت جلوتر از زمان برنامه را کامپایل کرد؟

رندر سریع‌تر

مرورگر به کمک AOT نسخه‌ی پیش کامپایل شده‌ای از برنامه را دانلود می‌کند. سپس مرورگر برای آن که بتواند بلافاصله برنامه را رندر کند شروع به بارگیری کد قابل اجرا می‌کند؛ بدون آن که منتظر کامپایل شدن برنامه بماند.

درخواست‌های ناهمگام کمتر

این کامپایلر داخل جاوا اسکریپت برنامه دارای قالب‌های خارجی HTML و سبک‌های CSS است، به همین دلیل درخواست‌های ajax مجزا برای این سورس فایل‌ها حذف شده است.

حجم دانلود کمتر برای فریمورک Angular

در صورتی که برنامه کامپایل شده باشد دیگر نیازی به دانلود کامپایلر Angular نیست. حجم کامپایلر تقریباً نصف خود Angular است، به همین دلیل حذف کردن آن باعث می‌شود حداکثر بار برنامه تا حد زیادی کاهش یابد.

شناسایی زودتر خطاهای قالب

کامپایلر AOT طی مرحله‌ی ساخت و قبل از آن که کاربران بتوانند خطاهای مقیدسازی قالب را ببینند، این خطاها را شناسایی کرده و گزارش می‌دهند.

امنیت بیشتر

AOT قالب‌های HTML و کامپوننت ها را داخل فایل‌های جاوا اسکریپت کامپایل می‌کند؛ خیلی قبل‌تر از آن که آن‌ها به کلاینت ارائه شوند. با توجه به اینکه هیچ قالبی برای خواندن وجود ندارد و هیچ HTML سمت کلاینت خطرناک و یا ارزیابی جاوا اسکریپتی وجود ندارد، موقعیت‌های کمتری برای حملات تزریقی پیش می‌آید.

کنترل کامپایل کردن برنامه

در زمان استفاده از کامپایلر AOT می‌توانید به دو صورت زیر کامپایل کردن برنامه‌ی خود را کنترل کنید:

- با ارائه‌ی امکانات کامپایلر قالب در فایل tsconfig.json

  برای اطلاعات بیشتر به «امکانات کامپایلر قالب در Angular» مراجعه کنید.
- با مشخص کردن متادیتا در Angular

مشخص کردن متادیتا در Angular

متادیتاهای Angular به Angular می گویند که چگونه نمونه‌های کلاس‌های برنامه‌ی شما را بسازد و در زمان اجرا با آن‌ها تعامل داشته باشد. کامپایلر AOT برای تفسیر بخش‌هایی از برنامه که Angular قرار است آن‌ها را مدیریت کند، متادیتاها را استخراج می‌کند.

برای مشخص کردن متادیتاها تنها کافی است از دکوراتورها مانند Component()@ و Input()@ استفاده کنید. راه دیگر این است که این کار را به صورت ضمنی در اعلان‌های موجود در constructor کلاس‌های آذین شده (decorated) انجام دهید.

در مثال زیر شیء متادیتای Component()@ و constructor کلاس به Angular می گویند که چگونه نمونه‌ای از TypicalComponent را ایجاد کرده و نمایش دهد.

```
@Component({
 selector: 'app-typical',
 template: '<div>A typical component for {{data.name}}</div>'
)}
export class TypicalComponent {
```

```
@Input() data: TypicalData;
constructor(private someService: SomeService) { ... }
}
```

کامپایلر Angular تنها یک بار این متادیتاها را استخراج می‌کند و برای TypicalComponent یک *فاکتوری* تولید می‌کند. زمانی که کامپایلر نیاز به ایجاد نمونه‌ای از TypicalComponent داشته باشد، Angular این فاکتوری را فراخوانی می‌کند. این فاکتوری المان بصری جدیدی را تولید می‌کند، به گونه‌ای که به نمونه‌ی جدیدی از کلاس کامپوننت به همراه وابستگی تزریق شده‌ی آن مقید است.

محدودیت‌های متادیتا

متادیتاهایی را که داخل زیر مجموعه‌ای از تایپ اسکریپت می‌نویسید باید از شرایط کلی زیر پیروی کنند:

1. سینتکس عبارتی را در زیر مجموعه‌ی پشتیبانی شده از جاوا اسکریپت محدود کند.
2. تنها پس از فولد شدن کد به علامت‌های صادر شده اشاره داشته باشد.
3. تنها توابعی را فراخوانی کند که توسط کامپایلر پشتیبانی می‌شوند.
4. اعضای کلاس‌های مقید به داده و آذین شده باید عمومی باشند.

بخش‌های بعدی به تبیین این نکات می‌پردازند.

نحوه‌ی کار AOT

می‌توان AOT را به صورت دو مرحله در نظر گرفت. مرحله‌ی اول، مرحله‌ی تحلیل کد است که در آن صرفاً نمایشی از منبع را ثبت می‌کند و در مرحله‌ی دوم که مرحله‌ی تولید کد است، StaticReflector تفسیر و محدودیت‌های مکان‌هایی که تفسیر می‌کند را مدیریت می‌کند.

مرحله‌ی 1: تحلیل

کامپایلر تایپ اسکریپت برخی از کارهای تحلیل مرحله‌ی اول را انجام می‌دهد. این کامپایلر *فایل‌های تعریف نوع* d.ts. به همراه اطلاعات نوعی که کامپایلر AOT برای تولید کد برنامه به آن‌ها نیاز دارد را منتشر می‌کند.

در همین زمان *جمع کننده‌ی* AOT متادیتاهای ثبت شده در دکوراتورهای Angular را تحلیل می‌کند و نتیجه‌ی اطلاعات متادیتا را در فایل‌های metadata.json. هر یک به صورت فایلd.ts. قرار می‌دهد.

می‌توان metadata.json. را به عنوان نموداری از ساختار کلی متادیتای یک دکوراتور در نظر گرفت که به صورت درخت سینتکس انتزاعی (AST) نمایش داده می‌شود.

schema.ts Angular فرمت JSON را به صورت مجموعه‌ای از رابط‌های تایپ اسکریپت شرح می‌دهد.

سینتکس عبارتی

جمع کننده تنها زیر مجموعه‌ی جاوا اسکریپت را می‌شناسد. برای تعریف اشیاء متادیتا می‌توانید از سینتکس های محدود زیر استفاده کنید:

Syntax	Example
Literal object	`{cherry: true, apple: true, mincemeat: false}`
Literal array	`['cherries', 'flour', 'sugar']`
Spread in literal array	`['apples', 'flour', ...the_rest]`
Calls	`bake(ingredients)`

New	`new Oven()`
Property access	`pie.slice`
Array index	`ingredients[0]`
Identity reference	`Component`
A template string	`` `pie is ${multiplier} times better than cake` ``
Literal string	`pi`
Literal number	`3.14153265`
Literal boolean	`true`

Literal null	`null`
Supported prefix operator	`!cake`
Supported binary operator	`a+b`
Conditional operator	`a ? b: c`
Parentheses	`(a+b)`

اگر در عبارتی از سینتکس پشتیبانی نشده‌ای استفاده شود، در این صورت جمع کننده گره‌ی خطایی را برای فایلmetadata.json. می‌نویسد. بعد از آن اگر کامپایلر به این بخش از متادیتا برای تولید کد برنامه نیاز داشته باشد، این خطا را گزارش می‌دهد.

اگر می‌خواهید ngc به جای آن که فایلmetadata.json. را به همراه خطاهایی ایجاد کند، خطاهای سینتکس را بلافاصله گزارش کند، strictMetadataEmit را در tsconfig تنظیم کنید.

```
"angularCompilerOptions": {
...
 "strictMetadataEmit": true
}
```

کتابخانه‌های Angular به این دلیل دارای این آپشن هستند که تضمین کنند تمامی فایل‌هایmetadata.json. بی نقص هستند. اگر می‌خواهید کتابخانه‌ای برای خود بسازید، این روش می‌تواند بهترین روش برای شما باشد.

توابع برداری ممنوع

کامپایلر AOT از عبارت‌های تابعی و توابع برداری که معروف به توابع لامبدا هستند، پشتیبانی نمی‌کند.

دکوراتور کامپوننت زیر را در نظر بگیرید.

```
@Component({
...
 providers: [{provide: server, useFactory: () => new Server()}]
})
```

جمع کننده‌ی AOT از تابع برداری () => new Server() پشتیبانی نمی‌کند. در یک عبارت متادیتا جمع کننده در مکان این تابع گره‌ی خطایی را تولید می‌کند.

در آینده اگر کامپایلر این گره را تفسیر کند، خطایی می‌دهد. به گونه‌ای که از شما دعوت می‌کند که این تابع برداری را به یک *تابع صادر شده* تبدیل کنید.

برای برطرف کردن این خطا می‌توانید به صورت زیر عمل کنید:

```
export function serverFactory() {
 return new Server();
}
@Component({
...
 providers: [{provide: server, useFactory: serverFactory}]
})
```

از ورژن 5 به بعد، کامپایلر در عین انتشار فایلjs. عملیات بازنویسی بالا را به صورت خودکار انجام می‌دهد.

فراخوان‌های محدود توابع

تا زمانی که سینتکس معتبر باشد، جمع کننده می‌تواند با دستور new فراخوان تابعی را ارائه و یا شیئی را ایجاد کند. تنها چیزی که برای جمع کننده مهم است، صحیح بودن سینتکس است.

اما حواستان باشد. کامپایلر ممکن است در آینده از تولید یک فراخوان برای یک تابع مشخص و یا از ایجاد یک شیء مشخص سر باز زند. کامپایلر تنها می‌تواند مجموعه‌ی کوچکی از توابع را فراخوانی کند و تنها برای تعداد کمی از کلاس‌های تخصیص داده شده از new استفاده کند. این توابع و کلاس‌ها در جدولی در ادامه‌ی همین صفحه آمده است.

فولدینگ

کامپایلر تنها می‌تواند ارجاع‌های به نشان‌های صادر شده را حل و فصل کند. خوشبختانه جمع کننده از طریق فولدینگ امکان استفاده‌ی محدودی از علامت‌های غیر صادر شده را فراهم می‌کند.

جمع کننده ممکن است طی جمع آوری عبارتی را ارزیابی کند و نتیجه را به جای عبارت اصلی درmetadata.json. ثبت کند.

مثلاً جمع کننده می‌تواند عبارت 4 + 3 + 2 + 1 را ارزیابی کند و جای آن را با 10 عوض کند.

به این فرآیند فولدینگ گفته می‌شود. عبارتی که بتوان به این صورت خلاصه کرد را قابل فولد شدن می گویند.

جمع کننده می‌تواند ارجاع‌های به اعلانات const در محل ماژول را ارزیابی کند و اعلانات var و let را مقداردهی اولیه کند و به طور مؤثر آن‌ها را از فایلmetadata.json. حذف کند.

تعریف کامپوننت زیر را در نظر بگیرید:

```
const template = '<div>{{hero.name}}</div>';

@Component({
 selector: 'app-hero',
 template: template
```

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

کامپایلر نمی‌تواند به ثابت template اشاره کند، زیرا این ثابت صادر شده نیست.

اما جمع کننده می‌تواند با لحاظ کردن محتویات تعریف متادیتا، این ثابت را درون این تعریف فولد کند. نحوه‌ی رفتار کامپایلر با کد زیر درست مانند کد بالا است:

```
@Component({
 selector: 'app-hero',
 template: '<div>{{hero.name}}</div>'
})
export class HeroComponent {
 @Input() hero: Hero;
}
```

حالا دیگر اشاره‌ای به template وجود ندارد و در آینده که کامپایلر بخواهد خروجی جمع کننده را درmetadata.json. تفسیر کند، با مشکلی مواجه نخواهد شد.

برای آن که یک قدم به جلو بروید می‌توانید ثابت template را در عبارت دیگری لحاظ کنید:

```
const template = '<div>{{hero.name}}</div>';

@Component({
 selector: 'app-hero',
 template: template + '<div>{{hero.title}}</div>'
})
export class HeroComponent {
```

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

در این حالت جمع کننده این عبارت را به رشته‌ی معادل فولد شده‌ی آن خلاصه می‌کند.:

سینتکس های قابل فولد

در جدول زیر عبارت‌هایی که جمع کننده می‌تواند آن‌ها را فولد کند و یا نکند، نشان داده شده‌اند:

Syntax	Foldable
Literal object	Yes
Literal array	Yes
Spread in literal array	no
Calls	no
New	no

Property access	yes, if target is foldable
Array index	yes, if target and index are foldable
Identity reference	yes, if it is a reference to a local
A template with no substitutions	yes
A template with substitutions	yes, if the substitutions are foldable
Literal string	yes
Literal number	yes
Literal boolean	yes

Literal null	yes
Supported prefix operator	yes, if operand is foldable
Supported binary operator	yes, if both left and right are foldable
Conditional operator	yes, if condition is foldable
Parentheses	yes, if the expression is foldable

اگر عبارتی قابل فولد باشد، در این صورت جمع کننده آن را به صورت یک AST برای کامپایلر جهت حل و فصل شدن درmetadata.json. می‌نویسد.

مرحله 2: تولید کد

جمع کننده هیچ قدمی در راستای درک متادیتایی که جمع آوری می‌کند و بهmetadata.json. می‌فرستد، بر نمی‌دارد. بلکه این متادیتا را تا حد امکان به بهترین شکل ممکن نمایش می‌دهد و زمانی که ببیند سینتکس متادیتایی نقض شده است، خطایی را ثبت می‌کند.

تفسیرmetadata.json. در مرحله‌ی تولید کد وظیفه‌ی کامپایلر است.

کامپایلر تمامی سینتکس هایی که جمع کننده از آن‌ها پشتیبانی می‌کند را درک می‌کند. اما اگر معناها قوانین کامپایلر را نقض کنند، کامپایلر ممکن است متادیتاهایی که سینتکس آن‌ها درست است را رد کند.

کامپایلر تنها می‌تواند به علامت‌های استخراج شده اشاره داشته باشد.

اعضای کلاس‌های کامپوننت آذین شده باید عمومی باشند. نمی‌توان ویژگی @Input() را خصوصی و یا داخلی کرد.

ویژگی‌های مقید سازی داده‌ها نیز باید عمومی باشند.

```
// BAD CODE - title is private
@Component({
 selector: 'app-root',
 template: '<h1>{{title}}</h1>'
})
export class AppComponent {
 private title = 'My App'; // Bad
}
```

از همه مهم‌تر کامپایلر تنها کدهایی را ایجاد می‌کند که بتواند نمونه‌هایی از کلاس‌های مشخصی را ایجاد کند، از دکوراتورهای مشخصی پشتیبانی کند و توابع مشخصی را از لیست‌های زیر فراخوانی کند.

نمونه‌های جدید

کامپایلر به متادیتا این اجازه را می‌دهد که تنها نمونه‌های کلاس InjectionToken و @angular/core را ایجاد کند.

حاشیه نویسی ها و دکوراتورها

کامپایلر تنها به ازای دکوراتورهای Angular زیر از متادیتا پشتیبانی می‌کند.

Decorator	Module

Attribute	@angular/core
Component	@angular/core
ContentChild	@angular/core
ContentChildren	@angular/core
Directive	@angular/core
Host	@angular/core
HostBinding	@angular/core
HostListner	@angular/core

Inject	@angular/core
Injectable	@angular/core
Input	@angular/core
NgModule	@angular/core
Optional	@angular/core
Output	@angular/core
Pipe	@angular/core
Self	@angular/core

SkipSelf	@angular/core
ViewChild	@angular/core

توابع ماکرو و متدهای استاتیک ماکرو

کامپایلر تنها به صورت تابع و یا متدهای استاتیک که عبارتی را برگشت می‌دهند از ماکروها پشتیبانی می‌کند.

برای مثال تابع زیر را در نظر بگیرید:

```
export function wrapInArray<T>(value: T): T[] {
 return [value];
}
```

می‌توانید در تعریف یک متادیتا wrapInArray را فراخوانی کنید. زیرا آن مقداری از یک عبارت را برگشت می‌دهد که این عبارت از زیر مجموعه‌ی جاوا اسکریپت محدود کننده‌ی کامپایلر پیروی می‌کند.

همچنین می‌توانید مانند زیر از wrapInArray() استفاده کنید:

```
@NgModule({
 declarations: wrapInArray(TypicalComponent)
})
export class TypicalModule{}
```

نحوه‌ی رفتار کامپایلر با کد زیر درست مانند کد بالا است:

```
@NgModule({
```

```
 declarations: [TypicalComponent]
})
export class TypicalModule{}
```

جمع کننده برای تعیین این که چیزی صلاحیت ماکرو بودن را دارد یا ندارد، بسیار ساده عمل می‌کند. یعنی تنها کافی است شامل یک دستور return باشد تا به عنوان ماکرو پذیرفته شود.

Angular RouterModule برای کمک به اعلان مسیرهای فرزند و ریشه دو متد استاتیک ماکرو به نام‌های forRoot و forChild را صادر می‌کند. برای آن که بفهمید ماکروها چگونه می‌توانند پیکربندی NgModule های پیچیده را ساده کنند، به سورس کد این متدها مراجعه کنید.

بازنویسی متادیتا

کامپایلر به object literal هایی که شامل فیلدهای useClass, useValue, useFactory و به خصوص data هستند، رسیدگی می‌کند. این کامپایلر عبارتی را به یک متغیر export شده تبدیل می‌کند که یکی از این فیلدها را مقداردهی اولیه می‌کند. فرآیند بازنویسی این عبارت‌ها باعث می‌شود تمامی محدودیت‌های آن‌ها حذف شود زیرا کامپایلر نیازی به دانستن مقدار عبارت ندارد؛ بلکه تنها باید ارجاعی به آن مقدار را ایجاد کند.

می‌توانید کدی مانند زیر بنویسید:

```
class TypicalServer {
}
@NgModule({
 providers: [{provide: SERVER, useFactory: () => TypicalServer}]
})
export class TypicalModule{}
```

بدون بازنویسی این کد نامعتبر است زیرا لامبداها پشتیبانی نشده است و TypicalServer به حالت export شده در نیامده است.

برای انجام این کار، کامپایلر به صورت خودکار این کد را به چیزی مانند زیر بازنویسی می‌کند:

```
class TypicalServer {


}


export const θ0 = () => new TypicalServer();
@NgModule({
 providers: [{provide: SERVER, useFactory: θ0}]
})
export class TypicalModule{}
```

این کار باعث می‌شود کامپایلر بتواند در فاکتوری و بدون دانستن مقدار θ0 ارجاعی را به θ0 تولید کند.

کامپایلر این کار بازنویسی را طی انتشار فایل.js انجام می‌دهد. با این حال کامپایلر فایلd.ts. را بازنویسی نمی‌کند. به همین دلیل تایپ اسکریپت این فایل را به عنوان یک فایل export شده نمی‌شناسد. در نتیجه این فایل API، export شده‌ی ماژول ES را آلوده نمی‌کند.

خطاهای متادیتا

در ادامه می‌توانید خطاهای متادیتایی که ممکن است با آن‌ها مواجه شوید را همراه با توضیحات و راه حل‌های پیشنهادی مشاهده کنید.

Expression form not supported

Reference to a local (non-exported) symbol

Only initialized variables and constants

Reference to a non-exported class

Reference to a non-exported function

Function calls are not supported

Destructured variable or constant not supported

Could not resolve type

Name expected

Unsupported enum member name

Tagged template expressions are not supported

Symbol reference expected

Expression form not supported

کامپایلر طی ارزیابی متادیتای Angular با عبارتی مواجه شده است که آن را درک نمی‌کند. همان طور که در مثال زیر مشاهده می‌کنید، ویژگی‌های زبانی خارج از سینتکس عبارت‌های محدود کامپایلر می‌توانند چنین خطایی را ایجاد کنند.

```
// ERROR
export class Fooish { ... }
...
const prop = typeof Fooish; // typeof is not valid in metadata
...
// bracket notation is not valid in metadata
{ provide: 'token', useValue: { [prop]: 'value' } };
...
```

در برنامه‌های دارای کدهای معمولی می‌توانید از typeof و براکت استفاده کنید. فقط نباید از این ویژگی‌ها درون عبارت‌هایی استفاده کنید که کارشان تعریف کردن متادیتای Angular است.

برای جلوگیری از بروز این خطا در زمان نوشتن متادیتا در Angular از سینتکس عبارت‌های محدود کامپایلر تبعیت کنید و نسبت به ویژگی‌های جدید و یا غیرمعمول تایپ اسکریپت آگاه باشید.

Reference to a local (non-exported) symbol

در صورت مشاهده‌ی خطای Reference to a local (non-exported) symbol 'symbol name' کافی است سمبل را به حالت export شده در آورید.

کامپایلر با ارجاعی به یکی از سمبل‌ها مواجه شده است که این سمبل به صورت محلی تعریف شده است. مشکل این سمبل این است که به صورت export شده در نیامده و یا مقداردهی اولیه نشده است.

در ادامه مثالی از این مشکل را مشاهده می‌کنید که در آن از provider استفاده شده است.

```
// ERROR
let foo: number; // neither exported nor initialized
@Component({
 selector: 'my-component',
 template: ...,
 providers: [
 { provide: Foo, useValue: foo }
 ]
})
export class MyComponent{}
```

کامپایلر فاکتوری کامپوننت را در ماژول مجزایی تولید می‌کند. به گونه‌ای که این فاکتوری شامل کد ارائه دهنده‌ی useValue است. *آن* ماژول فاکتوری برای دسترسی به متغیر محلی (export نشده) foo نمی‌تواند به *این* ماژول منبع دست پیدا کند.

این مشکل را می‌توانید با دادن مقدار اولیه‌ای به foo حل کنید.

```
let foo = 42; // initialized
```

کامپایلر درست مانند زمانی که کد زیر را نوشته باشید، این عبارت را در ارائه دهنده fold می‌کند.

```
providers: [
 { provide: Foo, useValue: 42 }
]
```

به عنوان جایگزین، برای حل این مشکل می‌توانید با این انتظار foo را export کنید که foo درست در زمان اجرا و در زمانی که شما واقعاً مقدار آن را می دانید، تخصیص داده شود.

```
// CORRECTED
export let foo: number; // exported
@Component({
 selector: 'my-component',
 template: ...,
 providers: [
 { provide: Foo, useValue: foo }
 ]
})
export class MyComponent{}
```

اضافه کردن export اغلب برای متغیرهایی جواب می‌دهد که در متادیتاهایی مانند providers و animations به آن‌ها اشاره شده باشد؛ زیرا کامپایلر می‌تواند ارجاعات متغیرهای export شده را در این عبارت‌ها تولید کند. کامپایلر نیازی به مقادیر این متغیرها ندارد.

اگر کامپایلر جهت تولید کد به مقدار واقعی متغیر نیاز داشته باشد دیگر امکان اضافه کردن export وجود ندارد. مثلاً، این کار برای ویژگی template جواب نمی‌دهد.

```
// ERROR
export let someTemplate: string; // exported but not initialized
@Component({
 selector: 'my-component',
 template: someTemplate
})
export class MyComponent{}
```

کامپایلر برای تولید فاکتوری کامپوننت همین الان به مقدار ویژگی template نیاز دارد. ارجاع متغیر به خودی خود کافی نیست. استفاده از export قبل از اعلان صرفاً باعث تولید خطای جدید "Only initialized variables and constants can be referenced" می‌شود.

Only initialized variables and constants

تنها به متغیرها و ثابت‌هایی که دارای مقدار اولیه هستند می‌توان اشاره کرد زیرا کامپایلر قالب به مقدار این متغیرها نیاز دارد.

کامپایلر ارجاعی به یکی از متغیرهای export شده و یا یک فیلد استاتیک پیدا کرده است که مقدار اولیه ندارد. کامپایلر برای تولید کد به مقدار این متغیر نیاز دارد.

در مثال زیر سعی شده است که ویژگی template کامپوننت بر روی مقدار متغیر someTemplate اکسپورت شده که اعلان شده اما تخصیص نیافته است تنظیم شود.

```
// ERROR
export let someTemplate: string;
@Component({
 selector: 'my-component',
 template: someTemplate
})
export class MyComponent{}
```

اگر someTemplate را از ماژول دیگری import کنید و فراموش کنید که در آنجا به آن مقدار اولیه بدهید باز هم با این خطا مواجه می‌شوید.

```
// ERROR - not initialized there either
import { someTemplate } from './config';
@Component({
 selector: 'my-component',
 template: someTemplate
})
export class MyComponent{}
```

کامپایلر برای دریافت اطلاعات قالب نمی‌تواند تا زمان اجرا صبر کند. حتماً باید به صورت استاتیکی مقدار متغیر someTemplate را از سورس کد به دست آورد تا بتواند فاکتوری کامپوننتی را تولید کند که شامل دستورالعمل‌های مورد نیاز برای ساختن المان بر اساس قالب است.

برای برطرف کردن این خطا مقدار اولیه‌ی متغیر را در یک بند initializer و در همان خط ارائه کنید.

```
// CORRECTED
export let someTemplate = '<h1>Greetings from Angular</h1>';
@Component({
 selector: 'my-component',
 template: someTemplate
})
export class MyComponent{}
```

Reference to a non-exported class

اشاره به یکی از کلاس‌های export نشده. در صورت مواجه شدن با این خطا کلاس را export کنید.

متادیتا به کلاسی اشاره کرده است که export نشده است.

برای مثال، ممکن است کلاسی را تعریف کرده باشید و در یکی از آرایه‌های ارائه کننده از آن به عنوان یک توکن تزریقی استفاده کرده باشید. اما فراموش کرده‌اید که آن کلاس را export کنید.

```
// ERROR
abstract class MyStrategy { }
…
 providers: [
 { provide: MyStrategy, useValue: … }
 ]
…
```

Angular در یک ماژول مجزا، یک فاکتوری کلاس را تولید می‌کند و آن فاکتوری می‌تواند تنها به کلاس‌های export شده دسترسی داشته باشد. برای برطرف کردن این خطا کلاس اشاره شده را export کنید.

```
// CORRECTED
export abstract class MyStrategy { }
…
```

```
providers: [
{ provide: MyStrategy, useValue: ... }
]
...
```

Reference to a non-exported function

متادیتا به تابع export نشده‌ای اشاره کرده است.

برای مثال، ممکن است ویژگی useFactory ارائه دهنده‌ای را بر روی یکی از توابعی تنظیم کرده باشید که به صورت محلی تعریف شده است، اما فراموش کرده باشید که آن را export کنید.

```
// ERROR
function myStrategy() { ... }
...
providers: [
{ provide: MyStrategy, useFactory: myStrategy }
]
...
```

Angular در یک ماژول مجزا، یک فاکتوری کلاس را تولید می‌کند و آن فاکتوری می‌تواند تنها به توابع export شده دسترسی داشته باشد. برای برطرف کردن این خطا این تابع را export کنید.

```
// CORRECTED
export function myStrategy() { ... }
...
providers: [
{ provide: MyStrategy, useFactory: myStrategy }
]
...
```

Function calls are not supported

از فراخوان‌های تابع پشتیبانی نمی‌شود. در صورت برخورد با این خطا جای تابع یا لامبدا را با یک ارجاع به یکی از توابع export شده عوض کنید.

کامپایلر در حال حاضر از عبارت‌های تابع و یا توابع لامبدا پشتیبانی نمی‌کند. برای مثال، نمی‌توانید useFactory ارائه دهنده‌ای را بر روی یک تابع بی نام و یا تابع برداری تنظیم کنید. مانند زیر:

```
// ERROR
...
 providers: [
 { provide: MyStrategy, useFactory: function() { ... } },
 { provide: OtherStrategy, useFactory: () => { ... } }
 ]
...
```

همچنین اگر تابع یا متدی را در یک useValue ارائه دهنده فراخوانی کرده باشید، باز هم با این خطا مواجه می‌شوید.

```
// ERROR
import { calculateValue } from './utilities';
...
 providers: [
 { provide: SomeValue, useValue: calculateValue() }
 ]
...
```

برای برطرف کردن این خطا تابعی را از ماژول export کنید و در عوض به تابع موجود در یک ارائه دهنده‌ی useFactory اشاره کنید.

```
// CORRECTED
import { calculateValue } from './utilities';
export function myStrategy() { ... }
export function otherStrategy() { ... }
```

```
export function someValueFactory() {
 return calculateValue();
}
…
 providers: [
 { provide: MyStrategy, useFactory: myStrategy },
 { provide: OtherStrategy, useFactory: otherStrategy },
 { provide: SomeValue, useFactory: someValueFactory }
 ]
…
```

Destructured variable or constant not supported

اشاره به یک متغیر و یا ثابت تجزیه شده و export شده توسط کامپایلر قالب پشتیبانی نمی‌شود. برای جلوگیری از تجزیه شدن، ساده سازی را در نظر بگیرید.

مثلاً شما نمی‌توانید چیزی مانند زیر را بنویسید:

```
// ERROR
import { configuration } from './configuration';

// destructured assignment to foo and bar
const {foo, bar} = configuration;
…
 providers: [
 {provide: Foo, useValue: foo},
 {provide: Bar, useValue: bar},
 ]
…
```

جهت تصحیح این خطا، به مقادیر تجزیه نشده اشاره کنید.

```
// ERROR
import { configuration } from './configuration';

// destructured assignment to foo and bar
const {foo, bar} = configuration;
…
 providers: [
 {provide: Foo, useValue: foo},
 {provide: Bar, useValue: bar},
 ]
…
```

Could not resolve type

کامپایلر با نوعی از داده مواجه شده است که نمی‌داند کدام ماژول این نوع را export کرده است.

در صورتی که به یک نوع پیرامونی (ambient) اشاره کنید این اتفاق ممکن است رخ دهد. مثلاً، نوع window یک نوع پیرامونی است که در فایل سراسریd.ts. اعلان شده است.

اگر در constructor کامپوننت به این نوع اشاره کنید به خطایی برخورد می‌کنید که کامپایلر باید به صورت ایستا آن را تحلیل کند.

```
// ERROR
@Component({ })
export class MyComponent {
 constructor (private win: Window) { … }
}
```

تایپ اسکریپت نوع‌های پیرامونی را می‌شناسد تا شما مجبور به import کردن آن‌ها نباشید. کامپایلر Angular نوعی را نمی‌شناسد که شما آن‌ها export یا import نکنید.

در این حالت، کامپایلر نمی‌داند که چگونه چیزی را با توکن Window تزریق کند.

در عبارت‌های متادیتا به نوع‌های پیرامونی اشاره نکنید.

اگر باید نمونه‌ای از یک نوع پیرامونی را تزریق کنید، می‌توانید با زیرکی در 4 مرحله‌ی زیر این مشکل را حل کنید:

1. برای نمونه‌ی این نوع پیرامونی یک توکن تزریقی ایجاد کنید.
2. یک تابع فاکتوری ایجاد کنید به گونه‌ای که این نمونه را برگشت دهد.
3. به کمک این تابع یک ارائه دهنده‌ی useFactory اضافه کنید.
4. برای تزریق نمونه از Inject@ استفاده کنید.

در ادامه مثال گویایی در این باره ذکر شده است.

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

export const WINDOW = new InjectionToken('Window');
export function _window() { return window; }

@Component({
...
 providers: [
 { provide: WINDOW, useFactory: _window }
 ]
})
export class MyComponent {
 constructor (@Inject(WINDOW) private win: Window) { ... }
}
```

بعد از انجام این کار نوع Window موجود در constructor دیگر برای کامپایلر مشکل ساز نخواهد بود، زیرا کامپایلر برای تولید کد تزریقی از (WINDOW)Inject@ استفاده می‌کند.

به طور مشابه Angular به توکن DOCUMENT رسیدگی می‌کند. تا شما بتوانید شیء document مرورگر را تزریق کنید (یا انتزاعی از آن، که این امر بستگی به پلتفرمی دارد که برنامه در آن اجرا می‌شود).

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

```
import { DOCUMENT } from '@angular/platform-browser';
@Component({ … })
export class MyComponent {
 constructor (@Inject(DOCUMENT) private doc: Document) { … }
}
```

Name expected

کامپایلر در عبارتی که در حال ارزیابی آن است، انتظار دارد اسمی را ببیند. این اتفاق زمانی رخ می‌دهد که شما به عنوان اسم ویژگی از یک عدد استفاده کنید. مانند مثال زیر:

```
// ERROR
provider: [{ provide: Foo, useValue: { 0: 'test' } }]
```

اسم این ویژگی را به چیزی غیر از عدد تغییر دهید.

```
// CORRECTED
provider: [{ provide: Foo, useValue: { '0': 'test' } }]
```

Unsupported enum member name

کامپایلر نتوانسته است مقدار عضو enum که شما در متادیتا به آن اشاره کرده‌اید را تشخیص دهد.

کامپایلر می‌تواند مقادیر enum ساده را بشناسد، اما در تشخیص مقادیر پیچیده‌ای مانند مقادیری که از ویژگی‌های محاسبه شده حاصل شده‌اند، ناتوان است.

```
// ERROR
enum Colors {
 Red = 1,
 White,
 Blue = "Blue".length // computed
}
```

…

```
providers: [
{ provide: BaseColor, useValue: Colors.White } // ok
{ provide: DangerColor, useValue: Colors.Red }// ok
{ provide: StrongColor, useValue: Colors.Blue }// bad
]
...
```

از اشاره به enum ها به کمک مقداردهنده های اولیه‌ی پیچیده و ویژگی‌های محاسبه شده پرهیز کنید.

Tagged template expressions are not supported

عبارت‌های برچسب دار قالب در متادیتا پشتیبانی نمی‌شوند.

کامپایلر در جاوا اسکریپت ES2015 با یک عبارت برچسب دار قالب مانند زیر مواجه شده است:

```
// ERROR
const expression = 'funky';
const raw = String.raw`A tagged template ${expression} string`;
...
template: '<div>' + raw + '</div>'
...
```

String.raw() یک تابع برچسبی است که به صورت بومی در جاوا اسکریپت ES2015 وجود دارد.

کامپایلر AOT از عبارت‌های برچسب دار قالب پشتیبانی نمی‌کند؛ به همین دلیل در عبارت‌های متادیتا از آن‌ها استفاده نکنید.

Symbol reference expected

در محل مشخص شده‌ی پیام خطا، کامپایلر انتظار دارد ارجاعی به یک سمبل را ببیند.

این خطا در صورتی رخ می‌دهد که در بند extends یک کلاس از یک عبارت استفاده کنید.

مرحله 3: ارزیابی عبارت مقیدسازی

در مرحله‌ی ارزیابی، کامپایلر قالب Angular برای ارزیابی عبارت‌های مقیدسازی در قالب‌ها از کامپایلر تایپ اسکریپت استفاده می‌کند. برای فعالسازی مستقیم این مرحله، آپشن کامپایلر "fullTemplateTypeCheck" را در "angularCompilerOptions" مربوط به tsconfig.json پروژه اضافه کنید (به آپشن های کامپایلر Angular مراجعه کنید).

ارزیابی قالب زمانی که خطایی نوعی را در یک عبارت مقیدسازی قالب شناسایی کند، پیام خطا می‌دهد. این کار شبیه به گزارش خطاهای نوع است که توسط کامپایلر تایپ اسکریپت در برابر کد موجود در یک فایلts. انجام می‌شود.

برای مثال، کامپوننت زیر را در نظر بگیرید:

```
@Component({
 selector: 'my-component',
 template: '{{person.addresss.street}}'
})
class MyComponent {
 person?: Person;
}
```

این کد باعث ایجاد خطای زیر می‌شود:

```
my.component.ts.MyComponent.html(1,1): : Property 'addresss' does not exist on type 'Person'. Did you mean 'address?'
```

اسم فایل گزارش شده در پیام خطا (my.component.ts.MyComponent.html) یک فایل ساختگی است که توسط کامپایلر قالب تولید شده است و محتویات داخل قالب کلاس MyComponent را در خود نگهداری می‌کند. کامپایلر هیچ گاه این فایل را داخل حافظه‌ی دیسک نمی‌نویسد. تعداد ستون‌ها و خطها متناسب با رشته‌ی قالب موجود در حاشیه نویسی کلاس (در این حالت کلاس MyComponent) است. اگر کامپوننتی به جای template از templateUrl استفاده کند، در این صورت خطاها در فایل HTML ای گزارش می‌شوند که به جای یک فایل ساختگی توسط templateUrl به آن اشاره می‌شود.

مکان خطا نقطه‌ی آغازین گره‌ی متنی است که شامل عبارت اینترپولاسیون به همراه این خطا است. اگر این خطا مانند "[value]="person.address.street یک attribute binding باشد، در این صورت مکان خطا مکان همان صفتی است که شامل این خطا است.

ارزیابی برای کنترل جزییات ارزیابی نوع از بررسی کننده‌ی نوع تایپ اسکریپت و آپشن های موجود در کامپایلر تایپ اسکریپت استفاده می‌کند. مثلاً اگر strictTypeChecks مشخص شده باشد، خطای my.component.ts.MyComponent.html(1,1)::: Object is possibly 'undefined' به همراه پیام خطای بالا گزارش می‌شود.

Type narrowing (محدود کردن نوع)

عبارت استفاده شده در دستورالعمل ngIf جهت محدود کردن اتحادیه انواع موجود در کامپایلر قالب Angular کاربرد دارد. همین کار را عبارت if در تایپ اسکریپت انجام می‌دهد. برای مثال برای جلوگیری از بروز خطای Object is possibly 'undefined' در قالب بالا، آن را به گونه‌ای اصلاح کنید که در صورتی که فقط مقدار person مانند زیر مقداردهی اولیه شده باشد، اینترپولاسیون را منتشر کند:

```
@Component({
 selector: 'my-component',
 template: '<span *ngIf="person"> {{person.addresss.street}} </span>'
})
class MyComponent {
 person?: Person;
}
```

استفاده از ngIf* باعث می‌شود که کامپایلر تایپ اسکریپت به این نتیجه برسد که person استفاده شده در عبارت مقید سازی هرگز undefined نخواهد بود.

دستورالعمل‌های اختصاصی شبه ngIf

دستورالعمل‌هایی که مانند nglf* رفتار می‌کنند می‌توانند با لحاظ کردن یک نشانگر عضو استاتیک اعلان کنند که می‌خواهند به شیوه‌ی یکسانی به آن‌ها رسیدگی شود. این نشانگر عضو استاتیک، سیگنالی به کامپایلر قالب است تا به آن‌ها مانند nglf* رسیدگی کند. این عضو استاتیک برای nglf* در زیر آمده است:

```
public static ngIfUseIfTypeGuard: void;
```

این کد اعلان می‌کند که به ویژگی ورودی nglf مربوط به دستورالعمل Nglf باید به عنوان محافظی برای استفاده از این قالب رسیدگی شود. به این معنی که تنها در صورتی قالب نمونه سازی می‌شود که ویژگی ورودی nglf برابر با true باشد.

عملگر اعلان نوع غیرتهی

اگر استفاده از nglf* آسان نباشد و یا اگر قیود موجود در کامپوننت تضمین کنند در صورت interpolated بودن مقیدسازی عبارت، این عبارت همیشه غیرتهی است، در این صورت استفاده از این عملگر برای سرکوب کردن خطای Object is possibly 'undefined' توصیه می‌شود.

در مثال زیر ویژگی‌های address و person همیشه در کنار یکدیگر تنظیم می‌شوند. به این معنی که address همیشه غیرتهی است اگر person غیرتهی باشد. هیچ روش ساده‌ای جهت توصیف این قید برای تایپ اسکریپت و کامپایلر قالب وجود ندارد، اما خطای موجود در این مثال با استفاده از address!.street سرکوب می‌شود.

```
@Component({
 selector: 'my-component',
 template: '<span *ngIf="person"> {{person.name}} lives on {{address!.street}} </span>'
})
class MyComponent {
 person?: Person;
 address?: Address;
 setData(person: Person, address: Address) {
 this.person = person;
 this.address = address;
```

```
}
}
```

با توجه به اینکه بازسازی کامپوننت ممکن است باعث شکست این قید شود، بهتر است که از عملگر اعلان غیرتهی کمتر استفاده شود.

در مثال بالا توصیه می‌شود که مانند زیر چک کردن address در nglf* لحاظ شود:

```
@Component({
 selector: 'my-component',
 template: '<span *ngIf="person && address"> {{person.name}} lives on {{address.street}} </span>'
})
class MyComponent {
 person?: Person;
 address?: Address;

 setData(person: Person, address: Address) {
 this.person = person;
 this.address = address;
 }
}
```

غیرفعال کردن بررسی نوع با استفاده از $any()

جهت غیرفعال کردن بررسی یک عبارت مقیدسازی، می‌توانید این عبارت را در یک فراخوان $any() cast pseudofunction محصور کنید. کامپایلر به صورت یک cast برای نوع any با آن رفتار می‌کند، درست مانند زمانی که از کست as any یا <any> در تایپ اسکریپت استفاده می‌شود.

در مثال زیر خطای Property addresss does not exist با کست کردن person در نوع any سرکوب می‌شود.

```
@Component({
selector: 'my-component',
```

```
 template: '{{$any(person).addresss.street}}'
})
class MyComponent {
 person?: Person;
}
```

آپشن های کامپایلر قالب Angular

این آپشن ها به عنوان اعضای شیء "angularCompilerOptions" در فایل tsconfig.json مشخص می‌شود.

این آپشن ها را به همراه آپشن های ارائه شده به کامپایلر تایپ اسکریپت مانند زیر مشخص کنید:

```
{
 "compilerOptions": {
 "experimentalDecorators": true,
…
 },
 "angularCompilerOptions": {
 "fullTemplateTypeCheck": true,
 "preserveWhitespaces": true,
…
 }
}
```

در بخش زیر به این آپشن ها پرداخته شده است.

enableResourceInlining

این آپشن به کامپایلر می‌گوید تا در تمامی دکوراتورهای Component@ با محتویات بیان شده در ویژگی‌های styles و template جای ویژگی templateUrl را با styleUrls عوض کند. در صورتی که این آپشن فعال شود، بارگیری templateUrl یا styleUrls در خروجیjs. مربوط به ngc، دیگر با سرعت کند گذشته انجام نخواهد شد.

skipMetadataEmit

این آپشن به کامپایلر می‌گوید که فایل‌های metadata.json. را تولید نکند. این آپشن به صورت پیش فرض false است.

فایل‌های metadata.json. شامل اطلاعاتی هستند که کامپایلر قالب از یک فایل ts. به آن‌ها نیاز دارد و این اطلاعات در فایل d.ts. تولید شده توسط کامپایلر تایپ اسکریپت موجود نیستند. از جمله مواردی که این اطلاعات شامل می‌شوند، می‌توان به محتوای حاشیه نویسی ها (مانند قالب کامپوننت) اشاره کرد که تایپ اسکریپت آن را در فایل js. و نه در فایل d.ts. منتشر می‌کند.

اگر از آپشن outFile-- تایپ اسکریپت استفاده می‌کنید، می‌توانید این آپشن را برابر با true قرار دهید. زیرا فایل‌های متادیتا برای این سبک از خروجی تایپ اسکریپت معتبر نیستند. استفاده از outFile-- در Angular توصیه نمی‌شود، در عوض از یک بسته‌ی نرم افزاری مانند webpack استفاده کنید.

اگر از چکیده‌های فاکتوری استفاده می‌کنید، باز هم می‌توانید مقدار آن را برابر با true قرار دهید زیرا این چکیده‌ها شامل یک نسخه کپی از اطلاعاتی هستند که داخل فایل metadata.json. قرار دارند.

strictMetadataEmit

این آپشن به کامپایلر قالب می‌گوید خطایی را به فایل metadata.json. گزارش دهد به شرط آن که "skipMetadataEmit" برابر با false باشد. این آپشن به صورت پیش فرض false است و از آن در صورتی می‌توان استفاده کرد که "skipMetadataEmit" برابر با false و "skipTemplateCodeGen" برابر با true باشد.

هدف این آپشن ارزیابی فایل‌های metadata.json. برای دسته بندی به کمک یک بسته‌ی npm است. کار ارزیابی به صورت صریح انجام می‌شود و می‌تواند خطاها را برای متادیتایی منتشر کند که این متادیتا اگر در کامپایلر قالب استفاده شود هرگز خطایی را تولید نخواهد کرد. اگر می‌خواهید خطای منتشر شده توسط این آپشن را برای یک سمبل export شده از بین ببرید، می‌توانید dynamic@ را در توضیح سمبل لحاظ کنید.

مشکلی ندارد که فایل‌های metadata.json. شامل خطا باشند. کامپایلر قالب این خطاها را گزارش می‌کند اگر از متادیتا در تعیین محتوای حاشیه نویسی استفاده شود. جمع کننده‌ی متادیتا نمی‌تواند سمبل‌هایی را پیش بینی کند که برای استفاده در یک حاشیه نویسی طراحی شده‌اند. به همین دلیل در متادیتای مربوط به سمبل‌های export شده، گره‌های خطا را به صورت انحصاری لحاظ می‌کند. سپس اگر از همین سمبل‌ها استفاده شده باشد کامپایلر قالب می‌تواند برای گزارش دادن یک خطا از این گره‌ها استفاده کند. اگر کلاینت کتابخانه‌ای بخواهد در یک حاشیه نویسی از یک سمبل استفاده کند، کامپایلر قالب در حالت معمولی این کار را گزارش نمی‌دهد، مگر آن که کلاینت از این سمبل استفاده کند. با استفاده از این آپشن می‌توان طی مرحله‌ی ساخت این کتابخانه این خطاها را شناسایی کرد و برای مثال جهت تولید خود کتابخانه‌های Angular از این آپشن استفاده کرد.

skipTemplateCodegen

این آپشن به کامپایلر می‌گوید که انتشار فایل‌های ngfactory.js. وngstyle.jsg. را متوقف کند. پس از تنظیم شدن این آپشن، بخش بزرگی از کامپایلر قالب خاموش می‌شود و گزارش عیوب قالب نیز غیر فعال می‌شود. به کمک این آپشن می‌توان به کامپایلر قالب گفت که فایل‌های metadata.json. را تولید کند تا این فایل‌ها به کمک بسته‌ی npm توزیع شوند و در عین حال از تولید فایل‌های ngfactory.js. وngstyle.jsg. جلوگیری کرد زیرا نمی‌توان این فایل‌ها را به کمک npm توزیع کرد.

strictInjectionParameters

اگر این آپشن بر روی true تنظیم شود، به کامپایلر گفته می‌شود که خطای یکی از پارامترهای ارائه شده را گزارش دهد. که نوع تزریق این پارامتر را نمی‌توان تعیین کرد. اگر این آپشن موجود نباشد و یا false باشد، در این صورت پارامترهای constructor کلاس‌هایی که با Injectable@ علامت گذاری شده‌اند و نمی‌توان نوع آن‌ها را تعیین کرد، هشدار می‌دهند.

نکته: توصیه می‌شود که این آپشن را به صورت صریح بر روی true قرار دهید تا به صورت پیش فرض در آینده نیز برابر با true باشد.

flatModuleOutFile

در صورتی که این آپشن برابر با true باشد، به کامپایلر قالب می‌گوید که شاخص ماژول مسطحی از یک اسم فایل مشخص و متادیتای ماژول مسطح متناظر با آن را تولید کند. از این آپشن زمانی استفاده کنید که می‌خواهید ماژول‌های مسطحی که مشابه @angular/core و @angular/common بسته بندی شده‌اند را تولید کنید. زمانی که از این آپشن استفاده می‌شود، package.json مربوط به کتابخانه به جای فایل شاخص کتابخانه باید به شاخص ماژول مسطح تولید شده اشاره کند. به کمک این آپشن تنها یک عدد فایلmetadata.json. تولید می‌شود، که این فایل شامل تمامی متادیتاهای ضروری برای سمبل‌های export شده از شاخص کتابخانه است. از شاخص ماژول مسطح در فایل‌هایngfactory.js. تولید شده، جهت import سمبل‌هایی که شامل API عمومی از شاخص کتابخانه به همراه سمبل‌های داخلی پوشیده شده هستند، استفاده می‌شود.

به صورت پیش فرض فایل ts. ای که در فیلد files وجود دارد، شاخص کتابخانه در نظر گرفته می‌شود. اگر بیش از یک فایلts. مشخص شود، برای انتخاب فایل مورد استفاده می‌توان از libraryIndex استفاده کرد. اگر بیش از یک فایلts. وجود داشته باشد اما بدون استفاده از libraryIndex باشند، خطایی رخ می‌دهد. در همان مکانی که فایلd.ts. شاخص کتابخانه وجود دارد، یکjs. وd.ts. شاخص ماژول مسطح همراه با اسم معلوم flatModuleOutFile ایجاد می‌شود. مثلاً اگر کتابخانه‌ای به عنوان شاخص کتابخانه‌ی ماژول از فایل public_api.ts استفاده کند، در این صورت فیلد tsconfig.json files به صورت ["public_api.ts"] می‌شود. بعد از این کار آپشن های flatModuleOutFile را می‌توان مثلاً بر روی "index.js" تنظیم کرد. این کار باعث می‌شود فایل‌های index.d.ts و index.metadata.json تولید شوند. فیلد module مربوط به package.json به صورت "index.js" و فیلد typings به صورت "index.d.ts" می‌شود.

flatModuleId

این آپشن شناسه‌ی ماژول ترجیحی جهت import کردن یک ماژول مسطح را مشخص می‌کند. ارجاعاتی که توسط کامپایلر قالب تولید شده‌اند، در زمان import سمبل‌های حاصل از ماژول مسطح، از اسم این ماژول استفاده می‌کنند. این کار تنها زمانی معنا دارد که flatModuleOutFile نیز موجود باشد. در غیر این صورت کامپایلر این آپشن را نادیده می‌گیرد.

generateCodeForLibraries

این آپشن به کامپایلر می‌گوید تا فایل‌های فاکتوری را (.ngfactory.js و .ngstyle.js) برای فایل‌های d.ts. به همراه فایل متناظر metadata.json. تولید کند. این آپشن به صورت پیش فرض بر روی true قرار دارد و اگر false شود، فایل‌های فاکتوری تنها برای فایل‌های ts. تولید می‌شوند.

در صورت استفاده از چکیده‌های فاکتوری، این آپشن بهتر است که بر روی false تنظیم شود.

fullTemplateTypeCheck

این آپشن به کامپایلر می‌گوید تا مرحله‌ی ارزیابی عبارت مربوط به کامپایلر قالب را که برای ارزیابی عبارت‌های مقیدسازی از تایپ اسکریپت استفاده می‌کند، فعال کند.

این آپشن به صورت پیش فرض false است.

نکته: توصیه می‌شود این آپشن را بر روی true قرار دهید، زیرا در آینده به صورت پیش فرض بر روی آن تنظیم خواهد شد.

annotateForClosureCompiler

این آپشن به کامپایلر می‌گوید تا برای حاشیه نویسی جاوا اسکریپت منتشر شده به کمک کامنت های JSDoc مورد نیاز کامپایلر Closure از Tsickle استفاده کند. مقدار پیش فرض آن false است.

annotationsAs

برای کنترل نحوه‌ی انتشار حاشیه نویسی های مشخص Angular در راستای بهبود tree-shaking از این آپشن استفاده کنید. این آپشن بر حاشیه نویسی ها و دکوراتورهای غیرAngularی اثری ندارد. مقدار پیش فرض آن static fields است.

مقدار	توضیحات
decorators	دکوراتورها را در محل رها کنید. این باعث می‌شود کار کامپایلر سریع‌تر شود. تایپ اسکریپت فراخوان‌هایی را برای کمکی __decorate منتشر می‌کند. برای انعکاس زمان اجرا

	از emitDecoratorMetadata-- استفاده کنید. هرچند که کد حاصل به صورت صحیح tree-shake نخواهد شد.
staticfields	جای دکوراتورها را در کلاس با یک فیلد استاتیک عوض کنید. با این کار می‌توانید برای حذف کلاس‌های بدون استفاده از tree-shaker های پیشرفته‌ای مانند کامپایلر کلوژر استفاده کنید.

trace

این آپشن به کامپایلر می‌گوید تا هم زمان با کامپایل کردن قالب‌ها اطلاعات اضافی‌ای را چاپ کند.

enableLegacyTemplate

برای آن که بین المان‌های DOM هم نام تعارضی به وجود نیاید، از نسخه‌ی Angular 4 به بعد، المان <template> به نفع <ng-template> کنار رفت. در صورتی که این آپشن بر روی true قرار گیرد، می‌توان از المان ناکارامد <template> استفاده کرد. این آپشن به صورت پیش فرض false است. ممکن است برخی از کتابخانه‌های Angular سوم شخص به این آپشن نیاز پیدا کنند.

disableExpressionLowering

کامپایلر قالب Angular کدی که در یک حاشیه نویسی استفاده شده است و یا می‌توان از آن استفاده کرد را به گونه‌ای تبدیل می‌کند که بتوان آن کد را از ماژول‌های فاکتوری قالب import کرد. برای اطلاعات بیشتر به بازنویسی متادیتاها مراجعه کنید.

اگر این آپشن false شود، این کار بازنویسی غیرفعال می‌شود و باید این کار را به صورت دستی انجام داد.

disableTypeScriptVersionCheck

اگر مقدار این آپشن برابر با true باشد، به کامپایلر می‌گوید که نسخه‌ی تایپ اسکریپت را چک نکند. در این حالت کامپایلر از بررسی نسخه صرف نظر کرده و اگر با نسخه‌ی پشتیبانی نشده‌ای از تایپ اسکریپت مواجه شود، خطا

نمی‌دهد. قرار دادن این آپشن بر روی true توصیه نمی‌شود، زیرا نسخه‌های پشتیبانی نشده‌ی تایپ اسکریپت ممکن است رفتار تعریف نشده‌ای داشته باشند.

این آپشن به صورت پیش فرض false است.

preserveWhitespaces

این آپشن برای کامپایلر تصمیم می‌گیرد که گره‌های متنی خالی از قالب‌های کامپایل شده را حذف بکند یا نکند. پیش فرض این آپشن از نسخه‌ی 6 به بعد false است که باعث می‌شود ماژول‌های فاکتوری قالب کمتر انتشار یابند.

allowEmptyCodegenFiles

این آپشن به کامپایلر می‌گوید که تمامی فایل‌های قابل تولید را تولید کند، حتی اگر این فایل‌ها خالی باشند. این آپشن به صورت پیش فرض false است. قوانین نسخه‌ی Bazel از این آپشن استفاده می‌کنند و این آپشن در ساده سازی چگونگی ردیابی وابستگی‌های فایل توسط این قوانین کاربرد دارد. استفاده از این آپشن خارج از قوانین Bazel توصیه نمی‌شود.