The Ahead-of-Time (AOT) Compiler

The Angular Ahead-of-Time (AOT) compiler converts your Angular HTML and TypeScript code into efficient JavaScript code during the build phase *before* the browser downloads and runs that code.

This guide explains how to build with the AOT compiler and how to write Angular metadata that AOT can compile.

Watch compiler author Tobias Bosch explain the Angular Compiler at AngularConnect 2016.

{@a overview}

Angular compilation

An Angular application consists largely of components and their HTML templates. Before the browser can render the application, the components and templates must be converted to executable JavaScript by an *Angular compiler*.

Angular offers two ways to compile your application:

1. ***Just-in-Time* (JIT)**, which compiles your app in the browser at runtime
2. ***Ahead-of-Time* (AOT)**, which compiles your app at build time.

JIT compilation is the default when you run the *build-only* or the *build-and-serve-locally* CLI commands:

ng build ng serve

{@a compile}

For AOT compilation, append the `--aot` flags to the *build-only* or the *build-and-serve-locally* CLI commands:

ng build --aot ng serve --aot

The `--prod` meta-flag compiles with AOT by default. See the [CLI documentation](https://github.com/angular/angular-cli/wiki) for details, especially the [`build` topic](https://github.com/angular/angular-cli/wiki/build).

{@a why-aot}

Why compile with AOT?

Faster rendering

With AOT, the browser downloads a pre-compiled version of the application. The browser loads executable code so it can render the application immediately, without waiting to compile the app first.

Fewer asynchronous requests

The compiler *inlines* external HTML templates and CSS style sheets within the application JavaScript, eliminating separate ajax requests for those source files.

Smaller Angular framework download size

There's no need to download the Angular compiler if the app is already compiled. The compiler is roughly half of Angular itself, so omitting it dramatically reduces the application payload.

Detect template errors earlier

The AOT compiler detects and reports template binding errors during the build step before users can see them.

Better security

AOT compiles HTML templates and components into JavaScript files long before they are served to the client. With no templates to read and no risky client-side HTML or JavaScript evaluation, there are fewer opportunities for injection attacks.

Angular Metadata and AOT

The Angular **AOT compiler** extracts and interprets **metadata** about the parts of the application that Angular is supposed to manage.

Angular metadata tells Angular how to construct instances of your application classes and interact with them at runtime.

You specify the metadata with **decorators** such as `@Component()` and `@Input()`. You also specify metadata implicitly in the constructor declarations of these decorated classes.

In the following example, the `@Component()` metadata object and the class constructor tell Angular how to create and display an instance of `TypicalComponent`.

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

The Angular compiler extracts the metadata *once* and generates a *factory* for `TypicalComponent`. When it needs to create a `TypicalComponent` instance, Angular calls the factory, which produces a new visual element, bound to a new instance of the component class with its injected dependency.

Metadata restrictions

You write metadata in a *subset* of TypeScript that must conform to the following general constraints:

1. Limit expression syntax to the supported subset of JavaScript.
2. Only reference exported symbols after code folding.
3. Only call functions supported by the compiler.
4. Decorated and data-bound class members must be public.

The next sections elaborate on these points.

How AOT works

It helps to think of the AOT compiler as having two phases: a code analysis phase in which it simply records a representation of the source; and a code generation phase in which the compiler's `StaticReflector` handles the interpretation as well as places restrictions on what it interprets.

Phase 1: analysis

The TypeScript compiler does some of the analytic work of the first phase. It emits the `.d.ts` *type definition files* with type information that the AOT compiler needs to generate application code.

At the same time, the AOT ***collector*** analyzes the metadata recorded in the Angular decorators and outputs metadata information in `.metadata.json` files, one per `.d.ts` file.

You can think of `.metadata.json` as a diagram of the overall structure of a decorator's metadata, represented as an abstract syntax tree (AST).

Angular's [schema.ts](https://github.com/angular/angular/blob/master/packages/compiler-cli/src/metadata/schema.ts) describes the JSON format as a collection of TypeScript interfaces.

{@a expression-syntax}

Expression syntax

The *collector* only understands a subset of JavaScript. Define metadata objects with the following limited syntax:

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]`
Identifier 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)`

If an expression uses unsupported syntax, the *collector* writes an error node to the `.metadata.json` file. The compiler later reports the error if it needs that piece of metadata to generate the application code.

If you want `ngc` to report syntax errors immediately rather than produce a `.metadata.json` file with errors, set the `strictMetadataEmit` option in `tsconfig`. ``` "angularCompilerOptions": { ... "strictMetadataEmit" : true } ``` Angular libraries have this option to ensure that all Angular `.metadata.json` files are clean and it is a best practice to do the same when building your own libraries.

{@a function-expression} {@a arrow-functions}

No arrow functions

The AOT compiler does not support function expressions and arrow functions, also called *lambda* functions.

Consider the following component decorator:

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

The AOT *collector* does not support the arrow function, `() => new Server()`, in a metadata expression. It generates an error node in place of the function.

When the compiler later interprets this node, it reports an error that invites you to turn the arrow function into an *exported function*.

You can fix the error by converting to this:

```
export function serverFactory() {
  return new Server();
}

@Component({
  ...
  providers: [{provide: server, useFactory: serverFactory}]
})
```

Beginning in version 5, the compiler automatically performs this rewritting while emitting the `.js` file.

Limited function calls

The *collector* can represent a function call or object creation with `new` as long as the syntax is valid. The *collector* only cares about proper syntax.

But beware. The compiler may later refuse to generate a call to a *particular* function or creation of a *particular* object. The compiler only supports calls to a small set of functions and will use `new` for only a few designated classes. These functions and classes are in a table of below.

Folding

{@a exported-symbols} The compiler can only resolve references to ***exported*** symbols. Fortunately, the *collector* enables limited use of non-exported symbols through *folding*.

The *collector* may be able to evaluate an expression during collection and record the result in the `.metadata.json` instead of the original expression.

For example, the *collector* can evaluate the expression `1 + 2 + 3 + 4` and replace it with the result, `10`.

This process is called *folding*. An expression that can be reduced in this manner is *foldable*.

{@a var-declaration} The collector can evaluate references to module-local `const` declarations and initialized `var` and `let` declarations, effectively removing them from the `.metadata.json` file.

Consider the following component definition:

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

@Component({
  selector: 'app-hero',
  template: template
})
export class HeroComponent {
  @Input() hero: Hero;
}
```

The compiler could not refer to the `template` constant because it isn't exported.

But the *collector* can *fold* the `template` constant into the metadata definition by inlining its contents. The effect is the same as if you had written:

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

There is no longer a reference to `template` and, therefore, nothing to trouble the compiler when it later interprets the *collector's* output in `.metadata.json`.

You can take this example a step further by including the `template` constant in another expression:

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

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

The *collector* reduces this expression to its equivalent *folded* string:

`'<div>{{hero.name}}</div><div>{{hero.title}}</div>'`.

Foldable syntax

The following table describes which expressions the *collector* can and cannot fold:

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
Identifier 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

If an expression is not foldable, the collector writes it to `.metadata.json` as an AST for the compiler to resolve.

Phase 2: code generation

The *collector* makes no attempt to understand the metadata that it collects and outputs to `.metadata.json`. It represents the metadata as best it can and records errors when it detects a metadata syntax violation.

It's the compiler's job to interpret the `.metadata.json` in the code generation phase.

The compiler understands all syntax forms that the *collector* supports, but it may reject *syntactically* correct metadata if the *semantics* violate compiler rules.

The compiler can only reference *exported symbols*.

Decorated component class members must be public. You cannot make an `@Input()` property private or internal.

Data bound properties must also be public.

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

{@a supported-functions} Most importantly, the compiler only generates code to create instances of certain classes, support certain decorators, and call certain functions from the following lists.

New instances

The compiler only allows metadata that create instances of the class `InjectionToken` from `@angular/core`.

Annotations/Decorators

The compiler only supports metadata for these Angular decorators.

Decorator	Module
`Attribute`	`@angular/core`
`Component`	`@angular/core`
`ContentChild`	`@angular/core`
`ContentChildren`	`@angular/core`
`Directive`	`@angular/core`
`Host`	`@angular/core`
`HostBinding`	`@angular/core`
`HostListener`	`@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`

Macro-functions and macro-static methods

The compiler also supports *macros* in the form of functions or static methods that return an expression.

For example, consider the following function:

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

You can call the `wrapInArray` in a metadata definition because it returns the value of an expression that conforms to the compiler's restrictive JavaScript subset.

You might use `wrapInArray()` like this:

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

The compiler treats this usage as if you had written:

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

The collector is simplistic in its determination of what qualifies as a macro function; it can only contain a single `return` statement.

The Angular `RouterModule` exports two macro static methods, `forRoot` and `forChild`, to help declare root and child routes. Review the source code for these methods to see how macros can simplify configuration of complex NgModules.

Metadata rewriting

The compiler treats object literals containing the fields `useClass`, `useValue`, `useFactory`, and `data` specially. The compiler converts the expression initializing one of these fields into an exported variable, which replaces the expression. This process of rewriting these expressions removes all the restrictions on what can be in them because the compiler doesn't need to know the expression's value—it just needs to be able to generate a reference to the value.

You might write something like:

```
class TypicalServer {

}

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

Without rewriting, this would be invalid because lambdas are not supported and `TypicalServer` is not exported.

To allow this, the compiler automatically rewrites this to something like:

```
class TypicalServer {

}

export const ɵ0 = () => new TypicalServer();

@NgModule({
  providers: [{provide: SERVER, useFactory: ɵ0}]
})
export class TypicalModule {}
```

This allows the compiler to generate a reference to `ɵ0` in the factory without having to know what the value of `ɵ0` contains.

The compiler does the rewriting during the emit of the `.js` file. This doesn't rewrite the `.d.ts` file, however, so TypeScript doesn't recognize it as being an export. Thus, it does not pollute the ES module's exported API.

Metadata Errors

The following are metadata errors you may encounter, with explanations and suggested corrections.

Expression form not supported

The compiler encountered an expression it didn't understand while evalutating Angular metadata.

Language features outside of the compiler's restricted expression syntax can produce this error, as seen in the following example:

```
// 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' } };
  ...
```

You can use `typeof` and bracket notation in normal application code. You just can't use those features within expressions that define Angular metadata.

Avoid this error by sticking to the compiler's restricted expression syntax when writing Angular metadata and be wary of new or unusual TypeScript features.

{@a reference-to-a-local-symbol}

Reference to a local (non-exported) symbol

Reference to a local (non-exported) symbol 'symbol name'. Consider exporting the symbol.

The compiler encountered a referenced to a locally defined symbol that either wasn't exported or wasn't initialized.

Here's a `provider` example of the problem.

```
// ERROR
let foo: number; // neither exported nor initialized

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

The compiler generates the component factory, which includes the `useValue` provider code, in a separate module. *That* factory module can't reach back to *this* source module to access the local (non-exported) `foo` variable.

You could fix the problem by initializing `foo`.

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

The compiler will fold the expression into the provider as if you had written this.

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

Alternatively, you can fix it by exporting `foo` with the expectation that `foo` will be assigned at runtime when you actually know its value.

```
// CORRECTED
export let foo: number; // exported

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

Adding `export` often works for variables referenced in metadata such as `providers` and `animations` because the compiler can generate *references* to the exported variables in these expressions. It doesn't need the *values* of those variables.

Adding `export` doesn't work when the compiler needs the *actual value* in order to generate code. For example, it doesn't work for the `template` property.

```
// ERROR
export let someTemplate: string; // exported but not initialized

@Component({
  selector: 'my-component',
  template: someTemplate
})
export class MyComponent {}
```

The compiler needs the value of the `template` property *right now* to generate the component factory. The variable reference alone is insufficient. Prefixing the declaration with

`export` merely produces a new error, "`Only initialized variables and constants can be referenced`".

{@a only-initialized-variables}

Only initialized variables and constants

Only initialized variables and constants can be referenced because the value of this variable is needed by the template compiler.

The compiler found a reference to an exported variable or static field that wasn't initialized. It needs the value of that variable to generate code.

The following example tries to set the component's `template` property to the value of the exported `someTemplate` variable which is declared but *unassigned*.

```
// ERROR
export let someTemplate: string;

@Component({
  selector: 'my-component',
  template: someTemplate
})
export class MyComponent {}
```

You'd also get this error if you imported `someTemplate` from some other module and neglected to initialize it there.

```
// ERROR - not initialized there either
import { someTemplate } from './config';

@Component({
  selector: 'my-component',
  template: someTemplate
})
export class MyComponent {}
```

The compiler cannot wait until runtime to get the template information. It must statically derive the value of the `someTemplate` variable from the source code so that it can generate the component factory, which includes instructions for building the element based on the template.

To correct this error, provide the initial value of the variable in an initializer clause *on the same line*.

```
// CORRECTED
export let someTemplate = '<h1>Greetings from Angular</h1>';

@Component({
  selector: 'my-component',
  template: someTemplate
})
export class MyComponent {}
```

Reference to a non-exported class

Reference to a non-exported class . Consider exporting the class.

Metadata referenced a class that wasn't exported.

For example, you may have defined a class and used it as an injection token in a providers array but neglected to export that class.

```
// ERROR
abstract class MyStrategy { }

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

Angular generates a class factory in a separate module and that factory can only access exported classes. To correct this error, export the referenced class.

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

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

Reference to a non-exported function

Metadata referenced a function that wasn't exported.

For example, you may have set a providers `useFactory` property to a locally defined function that you neglected to export.

```
// ERROR
function myStrategy() { ... }

  ...
  providers: [
    { provide: MyStrategy, useFactory: myStrategy }
  ]
  ...
```

Angular generates a class factory in a separate module and that factory can only access exported functions. To correct this error, export the function.

```
// CORRECTED
export function myStrategy() { ... }

  ...
  providers: [
    { provide: MyStrategy, useFactory: myStrategy }
  ]
  ...
```

{@a function-calls-not-supported}

Function calls are not supported

Function calls are not supported. Consider replacing the function or lambda with a reference to an exported function.

The compiler does not currently support function expressions or lambda functions. For example, you cannot set a provider's `useFactory` to an anonymous function or arrow function like this.

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

You also get this error if you call a function or method in a provider's `useValue`. ``` // ERROR import { calculateValue } from './utilities';

... providers: [{ provide: SomeValue, useValue: calculateValue() }] ... ```

To correct this error, export a function from the module and refer to the function in a `useFactory` provider instead.

// 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 }] ...

{@a destructured-variable-not-supported}

Destructured variable or constant not supported

Referencing an exported destructured variable or constant is not supported by the template compiler. Consider simplifying this to avoid destructuring.

The compiler does not support references to variables assigned by destructuring.

For example, you cannot write something like this:

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

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

To correct this error, refer to non-destructured values.

// CORRECTED import { configuration } from './configuration'; ... providers: [{provide: Foo, useValue: configuration.foo}, {provide: Bar, useValue: configuration.bar},] ...

Could not resolve type

The compiler encountered a type and can't determine which module exports that type.

This can happen if you refer to an ambient type. For example, the `Window` type is an ambiant type declared in the global `.d.ts` file.

You'll get an error if you reference it in the component constructor, which the compiler must statically analyze.

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

TypeScript understands ambiant types so you don't import them. The Angular compiler does not understand a type that you neglect to export or import.

In this case, the compiler doesn't understand how to inject something with the `Window` token.

Do not refer to ambient types in metadata expressions.

If you must inject an instance of an ambiant type, you can finesse the problem in four steps:

1. Create an injection token for an instance of the ambiant type.
2. Create a factory function that returns that instance.
3. Add a `useFactory` provider with that factory function.
4. Use `@Inject` to inject the instance.

Here's an illustrative example.

// 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) { ... } }

The `Window` type in the constructor is no longer a problem for the compiler because it uses the `@Inject(WINDOW)` to generate the injection code.

Angular does something similar with the `DOCUMENT` token so you can inject the browser's `document` object (or an abstraction of it, depending upon the platform in which the application runs).

import { Inject } from '@angular/core'; import { DOCUMENT } from '@angular/platform-browser';

@Component({ ... }) export class MyComponent { constructor (@Inject(DOCUMENT) private doc: Document) { ... } }

Name expected

The compiler expected a name in an expression it was evaluating. This can happen if you use a number as a property name as in the following example.

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

Change the name of the property to something non-numeric.

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

Unsupported enum member name

Angular couldn't determine the value of the enum member that you referenced in metadata.

The compiler can understand simple enum values but not complex values such as those derived from computed properties.

// 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] ...

Avoid referring to enums with complicated initializers or computed properties.

{@a tagged-template-expressions-not-supported}

Tagged template expressions are not supported

Tagged template expressions are not supported in metadata.

The compiler encountered a JavaScript ES2015 tagged template expression such as,
`// ERROR const expression = 'funky'; const raw = String.raw`A tagged template ${expression} string`; ... template: '<div>' + raw + '</div>'`
`String.raw()` is a *tag function* native to JavaScript ES2015.

The AOT compiler does not support tagged template expressions; avoid them in metadata expressions.

Symbol reference expected

The compiler expected a reference to a symbol at the location specified in the error message.

This error can occur if you use an expression in the `extends` clause of a class.

Summary

- What the AOT compiler does and why it is important.
- Why metadata must be written in a subset of JavaScript.
- What that subset is.
- Other restrictions on metadata definition.
- Macro-functions and macro-static methods.
- Compiler errors related to metadata.