Ahead-of-time (AOT) compilation

An Angular application consists mainly of components and their HTML templates. Because the components and templates provided by Angular cannot be understood by the browser directly, Angular applications require a compilation process before they can run in a browser.

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. Compiling your application during the build process provides a faster rendering in the browser.

This guide explains how to specify metadata and apply available compiler options to compile your applications efficiently using the AOT compiler.

Watch Alex Rickabaugh explain the Angular compiler at AngularConnect 2019.

{@a why-aot}

Here are some reasons you might want to use 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 application 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 application 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.

{@a overview}

Choosing a compiler

Angular offers two ways to compile your application:

- ***Just-in-Time* (JIT)**, which compiles your application in the browser at runtime. This was the default until Angular 8.
- ***Ahead-of-Time* (AOT)**, which compiles your application and libraries at build time. This is the default since Angular 9.

When you run the `ng build` (build only) or `ng serve` (build and serve locally) CLI commands, the type of compilation (JIT or AOT) depends on the value of the `aot` property in your build configuration specified in `angular.json`. By default, `aot` is set to `true` for new CLI applications.

See the CLI command reference and Building and serving Angular apps for more information.

How AOT works

The Angular AOT compiler extracts **metadata** to interpret the parts of the application that Angular is supposed to manage. You can specify the metadata explicitly in **decorators** such as `@Component()` and `@Input()`, or implicitly in the constructor declarations of the decorated classes. The metadata tells Angular how to construct instances of your application classes and interact with them at runtime.

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.

Compilation phases

There are three phases of AOT compilation.

- Phase 1 is *code analysis*. In this phase, the TypeScript compiler and *AOT collector* create a representation of the source. The collector does not attempt to interpret the metadata it collects. It represents the metadata as best it can and records errors when it detects a metadata syntax violation.
- Phase 2 is *code generation*. In this phase, the compiler's `StaticReflector` interprets the metadata collected in phase 1, performs additional validation of the metadata, and throws an error if it detects a metadata restriction violation.
- Phase 3 is *template type checking*. In this optional phase, the Angular *template compiler* uses the TypeScript compiler to validate the binding expressions in templates. You can enable this phase explicitly by setting the `fullTemplateTypeCheck` configuration option; see Angular compiler options.

Metadata restrictions

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

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

For additional guidelines and instructions on preparing an application for AOT compilation, see Angular: Writing AOT-friendly applications.

Errors in AOT compilation commonly occur because of metadata that does not conform to the compiler's requirements (as described more fully below). For help in understanding and resolving these problems, see AOT

Metadata Errors.

Configuring AOT compilation

You can provide options in the TypeScript configuration file that controls the compilation process. See Angular compiler options for a complete list of available options.

Phase 1: Code 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 describes the JSON format as a collection of TypeScript interfaces.

{@a expression-syntax}

Expression syntax limitations

The AOT 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]`
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)`

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 the TypeScript configuration file.

```
 "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}]
})
```

In version 5 and later, the compiler automatically performs this rewriting while emitting the `.js` file.

{@a exported-symbols} {@a code-folding}

Code folding

The compiler can only resolve references to ***exported*** symbols. The collector, however, can evaluate an expression during collection and record the result in the `.metadata.json`, rather than the original expression. This allows you to make limited use of non-exported symbols within expressions.

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. The collector, however, can fold the `template` constant into the metadata definition by in-lining 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
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

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.

Public symbols

The compiler can only reference *exported symbols*.

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

Supported classes and functions

The collector can represent a function call or object creation with `new` as long as the syntax is valid. The compiler, however, can later refuse to generate a call to a *particular* function or creation of a *particular* object.

The compiler can only create instances of certain classes, supports only core decorators, and only supports calls to macros (functions or static methods) that return expressions.

- New instances

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

- Supported decorators

  The compiler only supports metadata for the Angular decorators in the `@angular/core` module.

- Function calls

  Factory functions must be exported, named functions. The AOT compiler does not support lambda expressions ("arrow functions") for factory functions.

{@a function-calls}

Functions and static method calls

The collector accepts any function or static method that contains a single `return` statement. The compiler, however, only 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 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.

{@a metadata-rewriting}

Metadata rewriting

The compiler treats object literals containing the fields `useClass`, `useValue`, `useFactory`, and `data` specially, converting the expression initializing one of these fields into an exported variable that 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. It does not, however, rewrite the `.d.ts` file, so TypeScript doesn't recognize it as being an export. and it does not interfere with the ES module's exported API.

{@a binding-expression-validation}

Phase 3: Template type checking

One of the Angular compiler's most helpful features is the ability to type-check expressions within templates, and catch any errors before they cause crashes at runtime. In the template type-checking phase, the Angular template compiler uses the TypeScript compiler to validate the binding expressions in templates.

Enable this phase explicitly by adding the compiler option `"fullTemplateTypeCheck"` in the `"angularCompilerOptions"` of the project's TypeScript configuration file (see Angular Compiler Options).

Template validation produces error messages when a type error is detected in a template binding expression, similar to how type errors are reported by the TypeScript compiler against code in a `.ts` file.

For example, consider the following component:

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

This produces the following error:

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

The file name reported in the error message, `my.component.ts.MyComponent.html`, is a synthetic file generated by the template compiler that holds contents of the `MyComponent` class template. The compiler never writes this file to disk. The line and column numbers are relative to the template string in the `@Component` annotation of the class, `MyComponent` in this case. If a component uses `templateUrl` instead of `template`, the errors are reported in the HTML file referenced by the `templateUrl` instead of a synthetic file.

The error location is the beginning of the text node that contains the interpolation expression with the error. If the error is in an attribute binding such as `[value]="person.address.street"`, the error location is the location of the attribute that contains the error.

The validation uses the TypeScript type checker and the options supplied to the TypeScript compiler to control how detailed the type validation is. For example, if the `strictTypeChecks` is specified, the error `my.component.ts.MyComponent.html(1,1): : Object is possibly 'undefined'` is reported as well as the above error message.

Type narrowing

The expression used in an `ngIf` directive is used to narrow type unions in the Angular template compiler, the same way the `if` expression does in TypeScript. For example, to avoid `Object is possibly 'undefined'` error in the template above, modify it to only emit the interpolation if the value of `person` is initialized as shown below:

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

Using `*ngIf` allows the TypeScript compiler to infer that the `person` used in the binding expression will never be `undefined`.

For more information about input type narrowing, see Improving template type checking for custom directives.

Non-null type assertion operator

Use the non-null type assertion operator to suppress the `Object is possibly 'undefined'` error when it is inconvenient to use `*ngIf` or when some constraint in the component ensures that the expression is always non-null when the binding expression is interpolated.

In the following example, the `person` and `address` properties are always set together, implying that `address` is always non-null if `person` is non-null. There is no convenient way to describe this constraint to TypeScript and the template compiler, but the error is suppressed in the example by using `address!.street`.

```
  @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;
    }
  }
```

The non-null assertion operator should be used sparingly as refactoring of the component might break this constraint.

In this example it is recommended to include the checking of `address` in the `*ngIf` as shown below:

```
  @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;
    }
  }
```