Missing `@Directive()`/`@Component()` decorator migration

What does this migration do?

This migration adds an empty `@Directive()` decorator to undecorated base classes that:

- use Angular features
- are extended by directives or components

For example, in the diff below, a `@Directive()` decorator is added to `BaseMenu` because `BaseMenu` uses dependency injection.

Before: “‘ts export class BaseMenu { constructor(private vcr: ViewContainerRef) {} }

@Directive({selector: ‘[settingsMenu]’}) export class SettingsMenu extends BaseMenu {} “‘

After: “‘ts @Directive() export class BaseMenu { constructor(private vcr: ViewContainerRef) {} }

@Directive({selector: ‘[settingsMenu]’}) export class SettingsMenu extends BaseMenu {} “‘

In the event that a directive or component is extended by a class without a decorator, the schematic copies any inherited directive or component metadata to the derived class.

Before:

```
@Component({
  selector: 'base-menu',
  template: '<div></div>'
})
class BaseMenu {}

export class SettingsMenu extends BaseMenu {}
```

After:

```
@Component({
  selector: 'base-menu',
  template: '<div></div>'
})
class BaseMenu {}

@Component({
  selector: 'base-menu',
  template: '<div></div>'
```

```
})
export class SettingsMenu extends BaseMenu {}
```

This schematic also decorates classes that use Angular field decorators, including: - `@Input()` - `@Output()` - `@HostBinding()` - `@HostListener()` - `@ViewChild()` / `@ViewChildren()` - `@ContentChild()` / `@ContentChildren()`

Before:

```
class Base {
  @Output()
  countChanged = new EventEmitter<number>();
}

@Directive({
  selector: '[myDir]'
})
class Dir extends Base {
}
```

After:

```
@Directive() // schematic adds @Directive()
class Base {
  @Output()
  countChanged = new EventEmitter<number>();
}

@Directive({
  selector: '[myDir]'
})
class Dir extends Base {
}
```

Why is this migration necessary?

Migrating classes that use DI

When a class has a `@Directive()` or `@Component()` decorator, the Angular compiler generates extra code to inject dependencies into the constructor. When using inheritance, Ivy needs both the parent class and the child class to apply a decorator to generate the correct code.

You can think of this change as two cases: a parent class is missing a decorator or a child class is missing a decorator. In both scenarios, Angular's runtime needs additional information from the compiler. This additional information comes from adding decorators.

Decorator missing from parent class When the decorator is missing from the parent class, the subclass will inherit a constructor from a class for which the compiler did not generate special constructor info (because it was not decorated as a directive). When Angular then tries to create the subclass, it doesn't have the correct info to create it.

In View Engine, the compiler has global knowledge, so it can look up the missing data. However, the Ivy compiler only processes each directive in isolation. This means that compilation can be faster, but the compiler can't automatically infer the same information as before. Adding the `@Directive()` explicitly provides this information.

In the future, add `@Directive()` to base classes that do not already have decorators and are extended by directives.

Decorator missing from child class When the child class is missing the decorator, the child class inherits from the parent class yet has no decorators of its own. Without a decorator, the compiler has no way of knowing that the class is a `@Directive` or `@Component`, so it doesn't generate the proper instructions for the directive.

Migrating classes that use field decorators

In ViewEngine, base classes with field decorators like `@Input()` worked even when the class did not have a `@Directive()` or `@Component()` decorator. For example:

```
class Base {
  @Input()
  foo: string;
}

@Directive(...)
class Dir extends Base {
  ngOnChanges(): void {
    // notified when bindings to [foo] are updated
  }
}
```

However, this example won't compile with Ivy because the `Base` class *requires* either a `@Directive()` or `@Component()` decorator to generate code for inputs, outputs, queries, and host bindings.

Always requiring a class decorator leads to two main benefits for Angular:

1. The previous behavior was inconsistent. Some Angular features required a decorator (dependency injection), but others did not. Now, all Angular features consistently require a class decorator.

2. Supporting undecorated classes increases the code size and complexity of Angular. Always requiring class decorators allows the framework to become smaller and simpler for all users.

What does it mean to have a `@Directive()` decorator with no metadata inside of it?

The presence of the `@Directive` decorator causes Angular to generate extra code for the affected class. If that decorator includes no properties (metadata), the directive won't be matched to elements or instantiated directly, but other classes that *extend* the directive class will inherit this generated code. You can think of this as an "abstract" directive.

Adding an abstract directive to an `NgModule` will cause an error. A directive must have a `selector` property defined in order to match some element in a template.

When do I need a `@Directive()` decorator without a selector?

If you're using dependency injection, or any Angular-specific feature, such as `@HostBinding()`, `@ViewChild()`, or `@Input()`, you need a `@Directive()` or `@Component()` decorator. The decorator lets the compiler know to generate the correct instructions to create that class and any classes that extend it. If you don't want to use that base class as a directive directly, leave the selector blank. If you do want it to be usable independently, fill in the metadata as usual.

Classes that don't use Angular features don't need an Angular decorator.

I'm a library author. Should I add the `@Directive()` decorator to base classes?

As support for selectorless decorators is introduced in Angular version 9, if you want to support Angular version 8 and earlier, you shouldn't add a selectorless `@Directive()` decorator. You can either add `@Directive()` with a selector or move the Angular-specific features to affected subclasses.

What about applications using non-migrated libraries?

The Angular compatibility compiler (`ngcc`) should automatically transform any non-migrated libraries to generate the proper code.