Optimizing client app size with lightweight injection tokens

This page provides a conceptual overview of a dependency injection technique that is recommended for library developers. Designing your library with *lightweight injection tokens* helps optimize the bundle size of client applications that use your library.

You can manage the dependency structure among your components and injectable services to optimize bundle size by using tree-shakable providers. This normally ensures that if a provided component or service is never actually used by the app, the compiler can eliminate its code from the bundle.

However, due to the way Angular stores injection tokens, it is possible that such an unused component or service can end up in the bundle anyway. This page describes a dependency-injection design pattern that supports proper tree-shaking by using lightweight injection tokens.

The lightweight injection token design pattern is especially important for library developers. It ensures that when an application uses only some of your library's capabilities, the unused code can be eliminated from the client's application bundle.

When an application uses your library, there might be some services that your library supplies which the client application doesn't use. In this case, the application developer should expect that service to be tree-shaken, and not contribute to the size of the compiled application. Because the application developer cannot know about or remedy a tree-shaking problem in the library, it is the responsibility of the library developer to do so. To prevent the retention of unused components, your library should use the lightweight injection token design pattern.

When tokens are retained

To better explain the condition under which token retention occurs, consider a library that provides a library-card component, which contains a body and can contain an optional header.

```
<lib-card>
  <lib-header>...</lib-header>
</lib-card>
```

In a likely implementation, the `<lib-card>` component uses `@ContentChild()` or `@ContentChildren()` to obtain `<lib-header>` and `<lib-body>`, as in the following.

```
@Component({
  selector: 'lib-header',
  ...,
```

```
})
class LibHeaderComponent {}

@Component({
  selector: 'lib-card',
  ...,
})
class LibCardComponent {
  @ContentChild(LibHeaderComponent)
  header: LibHeaderComponent|null = null;
}
```

Because `<lib-header>` is optional, the element can appear in the template in its minimal form, `<lib-card></lib-card>`. In this case, `<lib-header>` is not used and you would expect it to be tree-shaken, but that is not what happens. This is because `LibCardComponent` actually contains two references to the `LibHeaderComponent`.

`@ContentChild(LibHeaderComponent) header: LibHeaderComponent;`

- One of these reference is in the *type position*– that is, it specifies `LibHeaderComponent` as a type: `header: LibHeaderComponent;`.
- The other reference is in the *value position*– that is, LibHeaderComponent is the value of the `@ContentChild()` parameter decorator: `@ContentChild(LibHeaderComponent)`.

The compiler handles token references in these positions differently.

- The compiler erases *type position* references after conversion from TypeScript, so they have no impact on tree-shaking.
- The compiler must retain *value position* references at runtime, which prevents the component from being tree-shaken.

In the example, the compiler retains the `LibHeaderComponent` token that occurs in the value position, which prevents the referenced component from being tree-shaken, even if the application developer does not actually use `<lib-header>` anywhere. If `LibHeaderComponent` is large (code, template, and styles), including it unnecessarily can significantly increase the size of the client application.

When to use the lightweight injection token pattern

The tree-shaking problem arises when a component is used as an injection token. There are two cases when that can happen.

- The token is used in the value position of a content query.
- The token is used as a type specifier for constructor injection.

In the following example, both uses of the `OtherComponent` token cause retention of `OtherComponent` (that is, prevent it from being tree-shaken when it is not

used).

```
class MyComponent {
  constructor(@Optional() other: OtherComponent) {}

  @ContentChild(OtherComponent)
  other: OtherComponent|null;
}
```

Although tokens used only as type specifiers are removed when converted to JavaScript, all tokens used for dependency injection are needed at runtime. These effectively change `constructor(@Optional() other: OtherComponent)` to `constructor(@Optional() @Inject(OtherComponent) other)`. The token is now in a value position, and causes the tree shaker to retain the reference.

For all services, a library should use tree-shakable providers, providing dependencies at the root level rather than in component constructors.

Using lightweight injection tokens

The lightweight injection token design pattern consists of using a small abstract class as an injection token, and providing the actual implementation at a later stage. The abstract class is retained (not tree-shaken), but it is small and has no material impact on the application size.

The following example shows how this works for the `LibHeaderComponent`.

```
abstract class LibHeaderToken {}

@Component({
  selector: 'lib-header',
  providers: [
    {provide: LibHeaderToken, useExisting: LibHeaderComponent}
  ]
  ...,
})
class LibHeaderComponent extends LibHeaderToken {}

@Component({
  selector: 'lib-card',
  ...,
})
class LibCardComponent {
  @ContentChild(LibHeaderToken) header: LibHeaderToken|null = null;
}
```

In this example, the `LibCardComponent` implementation no longer refers to `LibHeaderComponent` in either the type position or the value position. This lets full tree shaking of `LibHeaderComponent` take place. The `LibHeaderToken` is

retained, but it is only a class declaration, with no concrete implementation. It is small and does not materially impact the application size when retained after compilation.

Instead, `LibHeaderComponent` itself implements the abstract `LibHeaderToken` class. You can safely use that token as the provider in the component definition, allowing Angular to correctly inject the concrete type.

To summarize, the lightweight injection token pattern consists of the following.

1. A lightweight injection token that is represented as an abstract class.
2. A component definition that implements the abstract class.
3. Injection of the lightweight pattern, using `@ContentChild()` or `@ContentChildren()`.
4. A provider in the implementation of the lightweight injection token which associates the lightweight injection token with the implementation.

Use the lightweight injection token for API definition

A component that injects a lightweight injection token might need to invoke a method in the injected class. Because the token is now an abstract class, and the injectable component implements that class, you must also declare an abstract method in the abstract lightweight injection token class. The implementation of the method (with all of its code overhead) resides in the injectable component that can be tree-shaken. This lets the parent communicate with the child (if it is present) in a type-safe manner.

For example, the `LibCardComponent` now queries `LibHeaderToken` rather than `LibHeaderComponent`. The following example shows how the pattern lets `LibCardComponent` communicate with the `LibHeaderComponent` without actually referring to `LibHeaderComponent`.

```
abstract class LibHeaderToken {
  abstract doSomething(): void;
}

@Component({
  selector: 'lib-header',
  providers: [
    {provide: LibHeaderToken, useExisting: LibHeaderComponent}
  ]
  ...,
})
class LibHeaderComponent extends LibHeaderToken {
  doSomething(): void {
    // Concrete implementation of `doSomething`
  }
}
```

```
@Component({
  selector: 'lib-card',
  ...,
})
class LibCardComponent implement AfterContentInit {
  @ContentChild(LibHeaderToken)
  header: LibHeaderToken|null = null;

  ngAfterContentInit(): void {
    this.header && this.header.doSomething();
  }
}
```

In this example the parent queries the token to obtain the child component, and stores the resulting component reference if it is present. Before calling a method in the child, the parent component checks to see if the child component is present. If the child component has been tree-shaken, there is no runtime reference to it, and no call to its method.

Naming your lightweight injection token

Lightweight injection tokens are only useful with components. The Angular style guide suggests that you name components using the "Component" suffix. The example "LibHeaderComponent" follows this convention.

To maintain the relationship between the component and its token while still distinguishing between them, the recommended style is to use the component base name with the suffix "Token" to name your lightweight injection tokens: "LibHeaderToken".