Angular DI: Getting to know the Ivy NodeInjector

Alexey Zuev Follow
Jan 29 · 9 min read

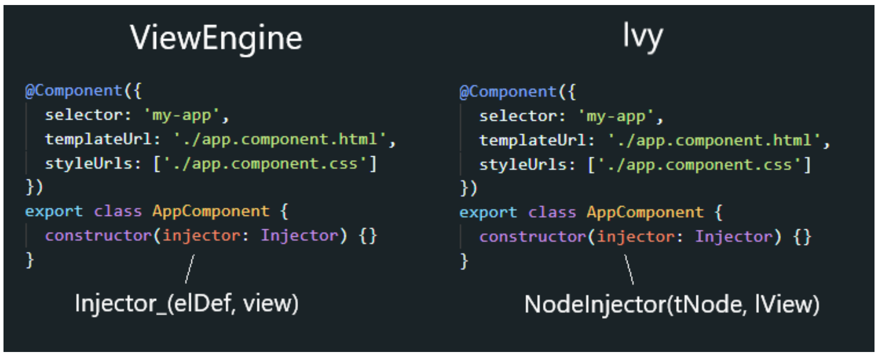
In this article, we're going to examine a new Angular NodeInjector which heavily uses a **bloom filter** to retrieve a token. We'll take a look at:

- How the NodeInjector looks like
- How Angular builds bloom filter for NodeInjector and when we can catch false positive values
- What's the resolution algorithm for resolving dependencies in NodeInjector

· · ·

Introduction

The NodeInjector is one of the two new types(another one is R3Injector) of Angular injectors introduced by the Ivy renderer. We will be switched to them as soon as Ivy renderer has landed.



```
@Component({
  selector: 'my-app',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  constructor(injector: Injector) {}
}
```

```
@Component({
  selector: 'my-app',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  constructor(injector: Injector) {}
}
```

The NodeInjector is going to replace the current Element injector(**Injector_** in the picture above) and reduce memory pressure in an Angular application by using bloom filter.

Let's first take a look at a simple application I'll use along the way:

```
@Component({
  selector: 'my-app',
  template: `
   <div dirA>
     <div dirB>Hello Ivy</div>
   </div>
  `
})
export class AppComponent {}

@Directive({ selector: '[dirA]' })
export class DirA {}
```

The app is pretty simple. We have a root AppComponent that contains two nested `div` elements. Also, there are two directives which are applied to those elements.

What we're going to understand is **how Angular will be able to get the root AppComponent instance in DirB directive**.

Now that we have our goal defined, let's get started.

View as a template representation

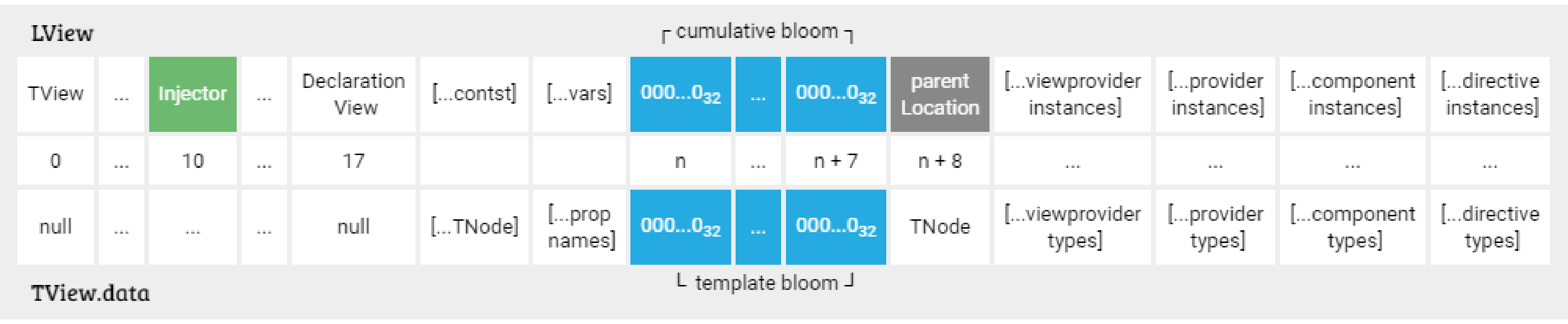
I think you're familiar with the concept of the **view** object in Angular. In simple words, it is some internal object that represents an Angular template.

We know that Angular builds a tree of views which always starts with a fake root View that contains only one root element. This is how it works in View Engine. The same can be applied to the upcoming Ivy engine.

To keep internal data Angular Ivy uses **LView** and **TView.data** arrays. The LView array contains data describing a specific template and in TView.data Angular keeps the information that is shared across templates.

Also, Angular **Ivy renderer stores the injection information for the node** in view data. In other words, it allocates slots in those LView and TView.data arrays. These slots represent two bloom filters: cumulative and template. One view can have as many bloom filters as many injectors are created for nodes which are located on this view.

Here's a visualization of what it looks like:



LView

┌ cumulative bloom ┐

TView	...	Injector	...	Declaration View	[...contst]	[...vars]	$000...0_{32}$	...	$000...0_{32}$	parent Location	[...viewprovider instances]	[...provider instances]	[...component instances]	[...directive instances]
0	...	10	...	17			n	...	n + 7	n + 8	...	...	...	...
null	...	...	...	null	[...TNode]	[...prop names]	$000...0_{32}$	...	$000...0_{32}$	TNode	[...viewprovider types]	[...provider types]	[...component types]	[...directive types]

└ template bloom ┘

TView.data

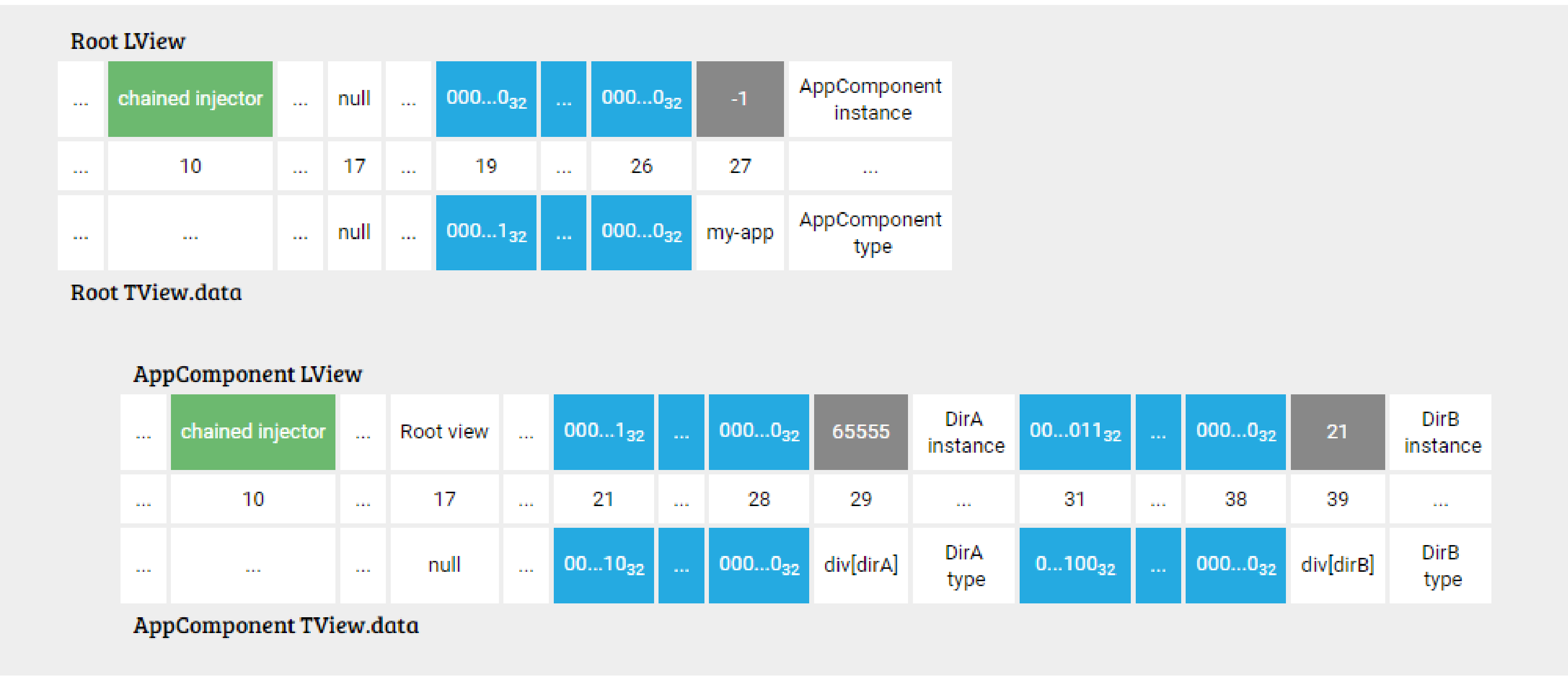
There are a few takeaways from this picture above:

- Ivy view is represented by LView and TView.data arrays that start from the **header**(17 slots). This header contains the reference to the parent **injector** at index **10**. Here's where the Angular resolution algorithm is switched to Module Injector.
- The LView and TView.data arrays can contain lots of bloom filters **8 slots** long ([n, n + 7] indices). Their number is directly proportional to the number of nodes for which the injector is created.
- Each bloom filter has a pointer to the parent bloom filter in the "packed" **parentLocation** slot (n + 8 index).

> By "packed" I meant that this slot's value contains not only a parent injector index but can also contain ViewOffsetShift. The packing and unpacking is a common use of the bitwise operators when we encode several values in one int.

- Angular stores all tokens in TView.data and instances in LView so that we can retrieve all providers looking at the view.

Let's get back to our application and see how many views we have there:



Root LView

...	chained injector	...	null	...	$000...0_{32}$	...	$000...0_{32}$	-1	AppComponent instance
...	10	...	17	...	19	...	26	27	...
...	...	...	null	...	$000...1_{32}$	...	$000...0_{32}$	my-app	AppComponent type

Root TView.data

AppComponent LView

...	chained injector	...	Root view	...	$000...1_{32}$	...	$000...0_{32}$	65555	DirA instance	$00...011_{32}$	...	$000...0_{32}$	21	DirB instance
...	10	...	17	...	21	...	28	29	...	31	...	38	39	...
...	...	...	null	...	$00...10_{32}$	...	$000...0_{32}$	div[dirA]	DirA type	$0...100_{32}$	...	$000...0_{32}$	div[dirB]	DirB type

AppComponent TView.data

That's quite simple. We have root view and one child AppComponent view. In the root view, we see one pair(cumulative in LView and template in TView.data) of bloom filters. The AppComponent view keeps two pairs of bloom filters: one for `div[dirA]` element(index 21) and other for `div[dirB]` element(index 31).

Now, that we have some knowledge of what an Angular Ivy view is, let's move on to bloom filter.

Cumulative and template bloom filters

> *If you are not familiar with bloom filter you might be interested in these great articles: Probabilistic Data structures: Bloom filter and Bloom Filters by Example*

In Angular Ivy we have a quite interesting implementation of bloom filter.

Template bloom filter is a filter that keeps information about the current node's tokens and which can be shared in TView.

Let's see how it is built.

As mentioned earlier "Bloom Filters by Example" article states:

> *The base data structure of a Bloom filter is a* ***Bit Vector***.

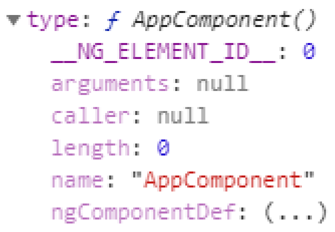
So what’s the bit vector in Angular bloom filter?

Ivy defines bloom size equal to 256 so that we have a vector in 256 bits which are divided into 8 parts.

```
               n                  ..              n + 7
00000000000000000000000000000000 ..
00000000000000000000000000000000
\____________32 bits___________/    \____________32
bits___________/
```

How Angular hashes elements in that filter?

First, Angular generates(*if it is not defined yet*) a unique ID for token via incrementing integer value and puts it to static **__NG_ELEMENT_ID__** property:



```
▼type: ƒ AppComponent()
   __NG_ELEMENT_ID__: 0
   arguments: null
   caller: null
   length: 0
   name: "AppComponent"
   ngComponentDef: (...)
```

> ***Note:*** *our AppComponent got 0 id since the token generation starts with 0 and AppComponent is the first directive which Angular adds to the injection system.*

Then, it takes that number and fits it to the bloom size through bitwise AND(&) operator so that the result is always between 0–255.

```
const BLOOM_SIZE = 256;
const BLOOM_MASK = BLOOM_SIZE - 1; // 255


/* it's like a remainder operator
*  so that all unique ids are modulo-ed
*  into a number between 0-255
*/
const bloomBit = id & BLOOM_MASK;
```

```
0 & 255    // 0
1 & 255    // 1
255 & 255  // 255
256 & 255  // 0
257 & 255  // 1
1000 & 255 // 232
```

Finally, Ivy creates a mask using that bloomBit:

```
const mask = 1 << bloomBit;
```

and sets that mask in one of 8 buckets depending on bloomBit:

```
// Use the raw bloomBit number to determine which bloc
// e.g: bf0 = [0 - 31], bf1 = [32 - 63], bf2 = [64 - 9
const b7 = bloomBit & 0x80;
const b6 = bloomBit & 0x40;
const b5 = bloomBit & 0x20;
const tData = tView.data as number[];

if (b7) {
  b6 ? (b5 ? (tData[injectorIndex + 7] |= mask) : (tDa
       (b5 ? (tData[injectorIndex + 5] |= mask) : (tDa
```

So, for all possible Ids, that are modulo-ed into a number between 0–255, we always get the following structure of bloom filter:

```
                                  Ids 0-31
 1 bucket                 00000000000000000000000000000000
                          \____________32 bits___________/

                                  Ids 32-63
 2 bucket                 00000000000000000000000000000000
...

                                 Ids 224 - 255
 8 bucket                 00000000000000000000000000000000
```

Okay, a lot is going here. Let me rephrase.

Having the directive like:

```
@Directive({...})
class MyDirective {
  static __NG_ELEMENT_ID__ = 1;
}
```

will result in the following bloom filter:

```
const bloomBit = 1 % 255 // 1

const mask = 1 << bloomBit;
             1 << 1 // 2

2..toString(2) // 10

1 bucket          00000000000000000000000000000010
....
8 bucket          00000000000000000000000000000000
```

How does Ivy check whether a given Id is in the set or not?

Ivy creates the same mask that targets the specific bit and simple checks that mask with dedicated bucket.

```
const bloomBit = 1 % 255 // 1

const mask = 1 << bloomBit;
             1 << 1 // 2

2..toString(2) // 10

               1 bucket
2 & 0b00000000000000000000000000000010

0b00000000000000000000000000000010
              &
0b00000000000000000000000000000010
              ||
0b00000000000000000000000000000010 = 2 = true
```

Now, let’s talk about cumulative bloom.

Cumulative bloom filter is a filter that stores information about the current node's tokens as well as the tokens of its ancestor nodes.

So, basically, it merges the parent's bloom filter and its own cumulative bloom.

Using this filter we can give a quick answer if there is a token in parent injectors without the need for traversing all parent injectors.

What's the NodeInjector?

In simple words, it's an injector which belongs to a node.

You can think of it the way you would think of any other injector. It is like a container that is used to retrieve object instances as defined by the provider. But I would say it's a special kind of containers.

Where does this container keep those providers?

First, I would look at its definition in source code:

```
export class NodeInjector implements Injector {
  constructor(
      private _tNode: TElementNode|TContainerNode|TElem
      private _lView: LView) {}

  get(token: any, notFoundValue?: any): any {
    return getOrCreateInjectable(this._tNode, this._lVi
```

Comparing it to R3Injector:

```
export class R3Injector {
  private records = new Map<Type<any>|InjectionToken<ar
  ...
}
```

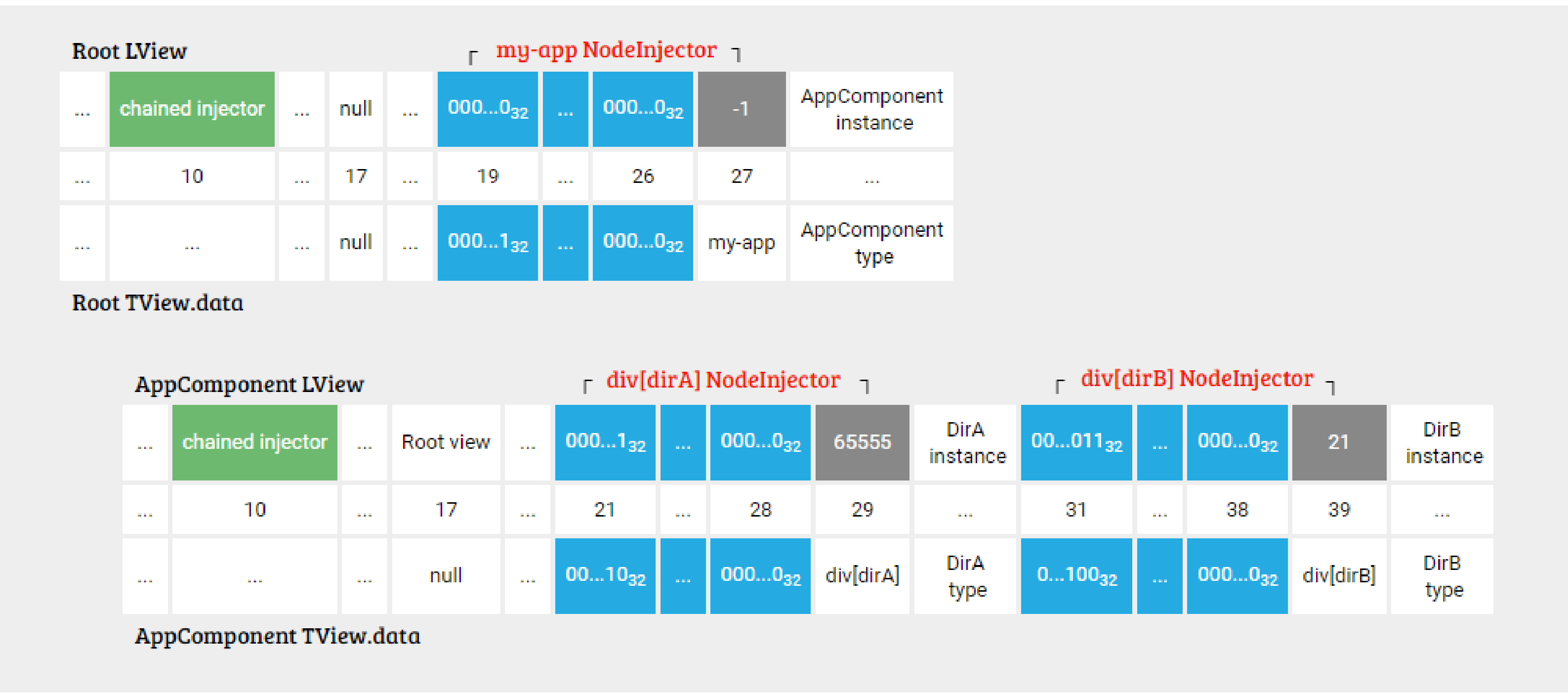
we can say that NodeInjector doesn't have any kind of key-value store which is commonly used to create injector.

The NodeInjector is an object that has references to TNode and LView objects. The TNode might be any kind of object: element, ng-template, ng-container. The NodeInjector gets the required provider by looking at the data contained in TNode and LView objects.

And here is where our bloom filters from the previous chapter come into play. Angular creates an **injectorIndex** property on TNode in order to know where dedicated to this node bloom filter is located.

Also, as we learned before after each bloom filter Angular also stores **parentLocation** pointer in LView array so that we can walk through all parent injectors.

So we can conclude that each NodeInjector is saved in 9 contiguous slots in LView and 9 contiguous slots in TView.data.



...	chained injector	...	null	...	$000...0_{32}$	...	$000...0_{32}$	-1	AppComponent instance
...	10	...	17	...	19	...	26	27	...
...	...	...	null	...	$000...1_{32}$	...	$000...0_{32}$	my-app	AppComponent type

...	chained injector	...	Root view	...	$000...1_{32}$	...	$000...0_{32}$	65555	DirA instance	$00...011_{32}$	...	$000...0_{32}$	21	DirB instance
...	10	...	17	...	21	...	28	29	...	31	...	38	39	...
...	...	...	null	...	$00...10_{32}$	...	$000...0_{32}$	div[dirA]	DirA type	$0...100_{32}$	...	$000...0_{32}$	div[dirB]	DirB type

When Angular creates NodeInjector on a node?

You may think that Angular does it for every node it creates. But it is not true.

There are a few cases when it happens:

- Root component always creates NodeInjector on root view.
- Angular creates NodeInjector on any tag that matches Angular component or any tag on which we're applying a directive.
- Providers and viewproviders defined on Component/Directive also create NodeInjector if it has not been created yet.

So, **every time we have a tag with directive applied(it might be component or directive) we create NodeInjector** on that TNode.

That's why all directives are known by node injector.

Now, its time to learn how NodeInjector resolves dependencies.

Resolution algorithm

As we have already seen earlier, the NodeInjector's `get` method looks like:

```
export class NodeInjector implements Injector {
  constructor(
      private _tNode: TElementNode|TContainerNode|TElem
      private _lView: LView) {}

  get(token: any, notFoundValue?: any): any {
    return getOrCreateInjectable(this._tNode, this._lVi
```

So, the getOrCreateInjectable method is the main entry point. There are lots of code, but let me break it down for you.

Imagine that we're calling `injector.get(SomeClass)`

1. Angular looks for a hash in `SomeClass.__NG_ELEMENT_ID__` static property.
2. If that hash is equal -1 then it is a special case and we'll get NodeInjector instance.
3. If that hash is a factory function then we have another special case where we should initialize object by calling that function.

Angular defines factory function in `__NG_ELEMENT_ID__` static property for the following **special objects**: *ChangeDetectorRef, ElementRef, TemplateRef, ViewContainerRef, Renderer2*.

An interesting thing about these objects is that they are not memoized. For example, `injector.get(ChangeDetectorRef) !== injector.get(ChangeDetectorRef)`

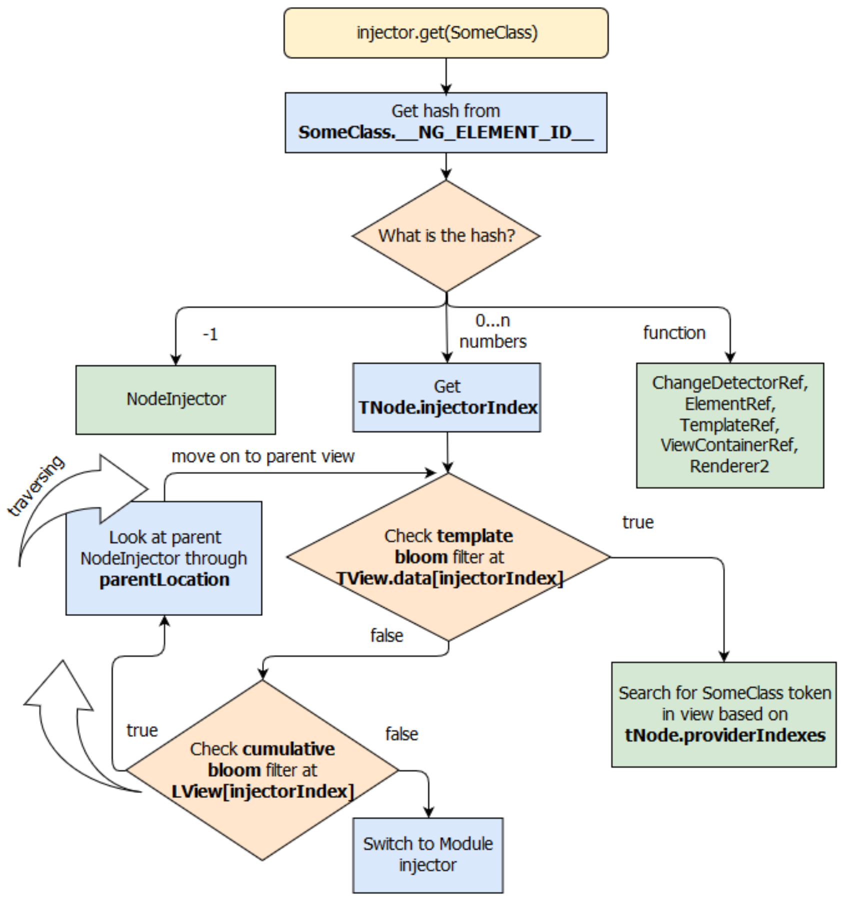
4. If that hash is a number then we're:

- getting injectorIndex from TNode.
- looking at template bloom filter(TView.data[injectorIndex])

If bloom filter gives us **true** then we search for SomeClass token. (We have tNode.providerIndexes so we can find desired token)

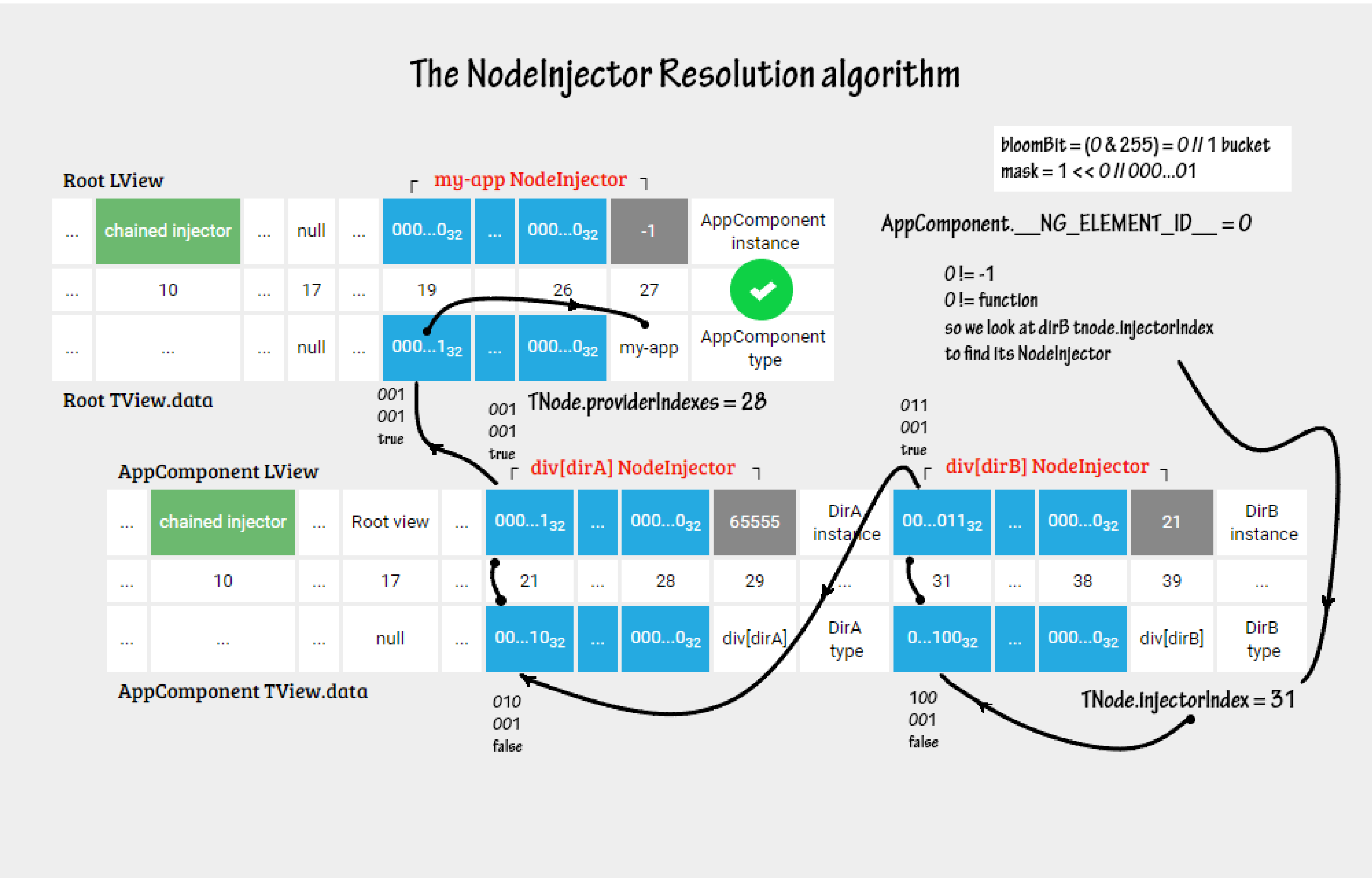
If it gives us **false** then we look at the cumulative bloom.

If it gives us true we continue on traversing otherwise we are switched to ModuleInjector.


injector.get(SomeClass)
Get hash from SomeClass.__NG_ELEMENT_ID__
What is the hash?
-1
0...n numbers
function
NodeInjector
Get TNode.injectorIndex
ChangeDetectorRef, ElementRef, TemplateRef, ViewContainerRef, Renderer2
traversing
move on to parent view
Look at parent NodeInjector through parentLocation
Check template bloom filter at TView.data[injectorIndex]
true
false
Search for SomeClass token in view based on tNode.providerIndexes
true
Check cumulative bloom filter at LView[injectorIndex]
false
Switch to Module injector

NodeInjector resolution algorithm

It’s time to take a closer look at how we’re getting root AppComponent in our simple application:


The NodeInjector Resolution algorithm
bloomBit = (0 & 255) = 0 // 1 bucket
mask = 1 << 0 // 000...01
Root LView
my-app NodeInjector
... | chained injector | ... | null | ... | 000...0₃₂ | ... | 000...0₃₂ | -1 | AppComponent instance
... | 10 | ... | 17 | ... | 19 | | 26 | 27
... | ... | ... | null | ... | 000...1₃₂ | ... | 000...0₃₂ | my-app | AppComponent type
Root TView.data
001 001 true
001 001 true
TNode.providerIndexes = 28
AppComponent.__NG_ELEMENT_ID__ = 0
0 != -1
0 != function
so we look at dirB tnode.injectorIndex
to find its NodeInjector
011 001 true
AppComponent LView
div[dirA] NodeInjector
div[dirB] NodeInjector
... | chained injector | ... | Root view | ... | 000...1₃₂ | ... | 000...0₃₂ | 65555 | DirA instance | 00...011₃₂ | ... | 000...0₃₂ | 21 | DirB instance
... | 10 | ... | 17 | ... | 21 | ... | 28 | 29 | ... | 31 | ... | 38 | 39 | ...
... | ... | ... | null | ... | 00...10₃₂ | ... | 000...0₃₂ | div[dirA] | DirA type | 0...100₃₂ | ... | 000...0₃₂ | div[dirB] | DirB type
AppComponent TView.data
010 001 false
100 001 false
TNode.injectorIndex = 31

False positive

You might be interested in cases when we can get an erroneous result from bloom filter, aka **false positive**.

Bloom has a size of 256 bits. Once we crossed 255 ids we can catch false positive.

Let’s see it on the example:

```
@Component({
  selector: 'my-app',
  template: `
   <div>
     <div dirB>Hello Ivy</div>
   </div>
  `
})
export class AppComponent {}

@Directive({ selector: '[dirA]' })
export class DirA {
  static __NG_ELEMENT_ID__ = 256;
```

AppComponent has `__NG_ELEMENT_ID__` defined to 0. We also manually defined id 256 for DirA. 0 and 256 will give us the mask at the same index so we get positive false.

But it is safe to have such a false positive result since after searching the token directly on a view we will get null.

Summary: the more directives and services on those directives we have, the more false positive values we can get.

Conclusion

Finally, we have come to end.

I hope now you have some basic understanding of how the Ivy NodeInjector works. If you haven't then I suggest you reading Angular source code. And for sure you will discover lots of patterns and best practices there.

Additional Resources

What you always wanted to know about Angular Dependency Injection tree

If you didn't dive deep into angular dependency injection mechanism, your…

blog.angularindepth.com