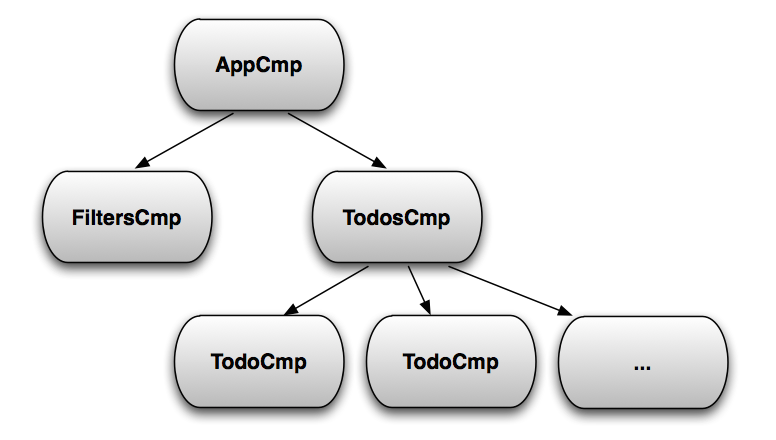
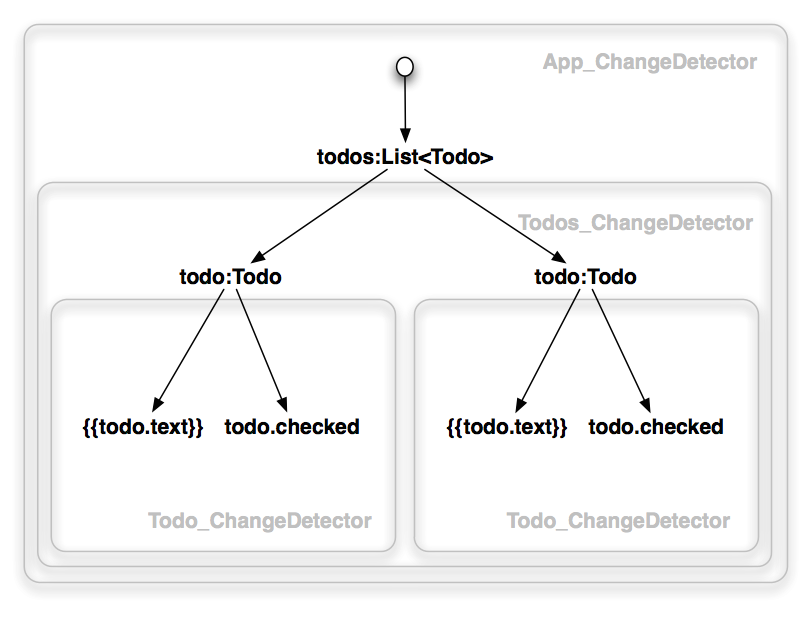
**High-Level Overview**

*An Angular 2 application is a tree of components.*



*An Angular 2 application is a reactive system*, with change detection being the core of it.



* **Every component gets a change detector responsible for checking the bindings defined in its template.** Examples of bindings: {{todo.text}} and [todo]=”t”. Change detectors propagate bindings **from** the root to leaves in the depth first order.
* Angular 2 does not have a generic mechanism implementing two-way data-bindings (But you can still implement the two-way data binding behavior and ng-model. Read more about it [here](http://victorsavkin.com/post/119943127151/angular-2-template-syntax).). That is why the change detection graph is a directed tree and cannot have cycles.
* This makes the system significantly more **performant**. And what is more important we gain guarantees that **make the system more predictable and easier debug.**

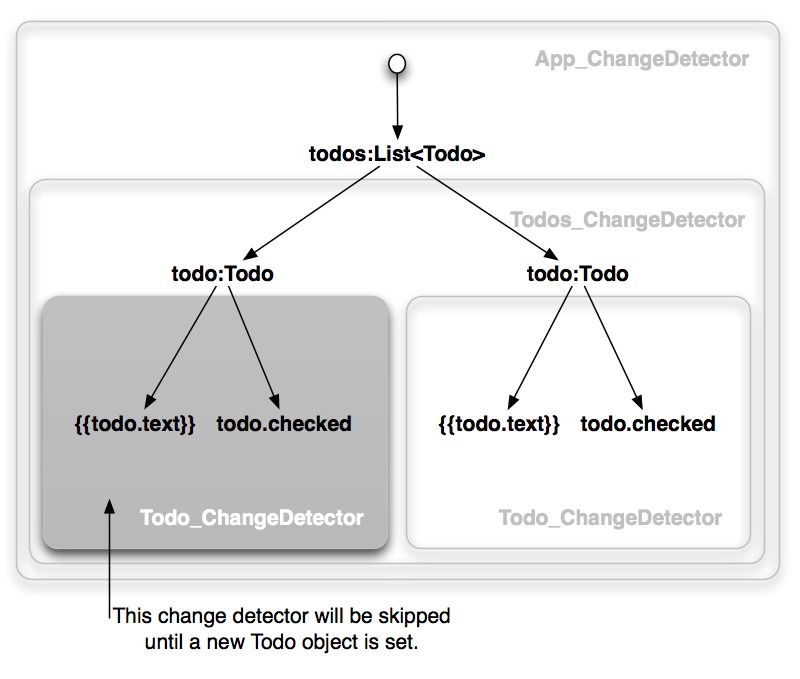
#### How Fast Is It?

By default the change detection goes through every node of the tree to see if it changed, and it does it on every browser event. Although it may seem terribly inefficient, **the Angular 2 change detection system can go through hundreds of thousands of simple checks** (the number are platform dependent) in a few milliseconds.

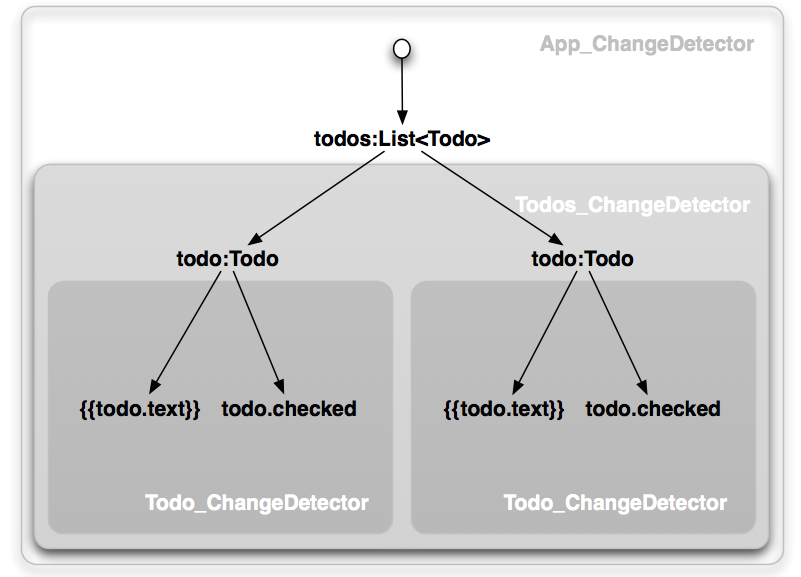
**Angular has to be conservative and run all the checks every single time because the JavaScript language does not give us object mutation guarantees. But we may know that certain properties hold if we, for example, use immutable or observable objects**.

**Immutable Objects**

If a component depends only on its input properties, and they are immutable, then this component can change if and only if one of its input properties changes. Therefore, we can skip the component’s subtree in the change detection tree until such an event occurs. When it happens, we can check the subtree once, and then disable it until the next change (gray boxes indicate disabled change detectors).



If we are aggressive about using immutable objects, a big chunk of the change detection tree will be disabled most of the time.



Implementing this is trivial. Just set the change detection strategy to OnPush.

@Component({changeDetection:ChangeDetectionStrategy.OnPush})

class ImmutableTodoCmp {

todo:Todo;

}

**It is worth noting that a component can still have private mutable state** as long as it changes only due to inputs being updated or an event being fired from within the component’s template. The only thing the OnPush strategy disallows is depending on shared mutable state.

#### Observable Objects

**If a component depends only on its input properties, and they are observable, then this component can change if and only if one of its input properties emits an event.** Therefore, we can skip the component’s subtree in the change detection tree until such an event occurs. When it happens, we can check the subtree once, and then disable it until the next change.

Although it may sound similar to the Immutable Objects case, it is quite different. If you have a tree of components with immutable bindings, a change has to go through all the components starting from the root. This is not the case when dealing with observables.

As you can see, here the Todos component has only a reference to an observable of an array of todos. So it cannot see the changes in individual todos.

The way to handle it is to check the path from the root to the changed Todo component when its todo observable fires an event. The change detection system will make sure this happens.

**Do observables cause cascading updates?**

Observable objects have bad reputation because they can cause cascading updates. Anyone who has experience in building large applications in a framework that relies on observable models knows what I am talking about. One observable object update can cause a bunch of other observable objects trigger updates, which can do the same. Somewhere along the way views get updated. Such systems are very hard to debug.

***Using observable objects in Angular 2 as shown above will not have this problem. An event triggered by an observable just marks a path from the component to the root as to be checked next time. Then the normal change detection process kicks in and goes through the nodes of the tree in the depth first order.*** So the order of updates does not change whether you use observables or not. This is very important. Using an observable object becomes a simple optimization that does not change the way you think about the system.

**Do I have to use observable/immutable objects everywhere to see the benefits?**

No, you don’t have to. You can use observables in one part of your app (e.g., in some gigantic table), and that part will get the performance benefits. Even more, you can compose different types of components and get the benefits of all of them. For instance, an “observable component” can contain an “immutable component”, which can itself contain an “observable component”. Even in such a case the change detection system will minimize the number of checks required to propagate changes.

**Summary**

* An Angular 2 application is a reactive system.
* The change detection system propagates bindings from the root to leaves.
* Unlike Angular 1.x, the change detection graph is a directed tree. **As a result, the system is more performant and predictable.**
* By default, the change detection system walks the whole tree. But if you use immutable objects or observables, you can take advantage of them and check parts of the tree only if they “really change”.
* These optimizations compose and do not break the guarantees the change detection provides.

References:

https://vsavkin.com/change-detection-in-angular-2-4f216b855d4c#.29fw6it6m