Angular Ivy change detection execution: are you prepared?

Alexey Zuev Follow

May 20, 2018 · 8 min read

Update:

Try Ivy jit mode

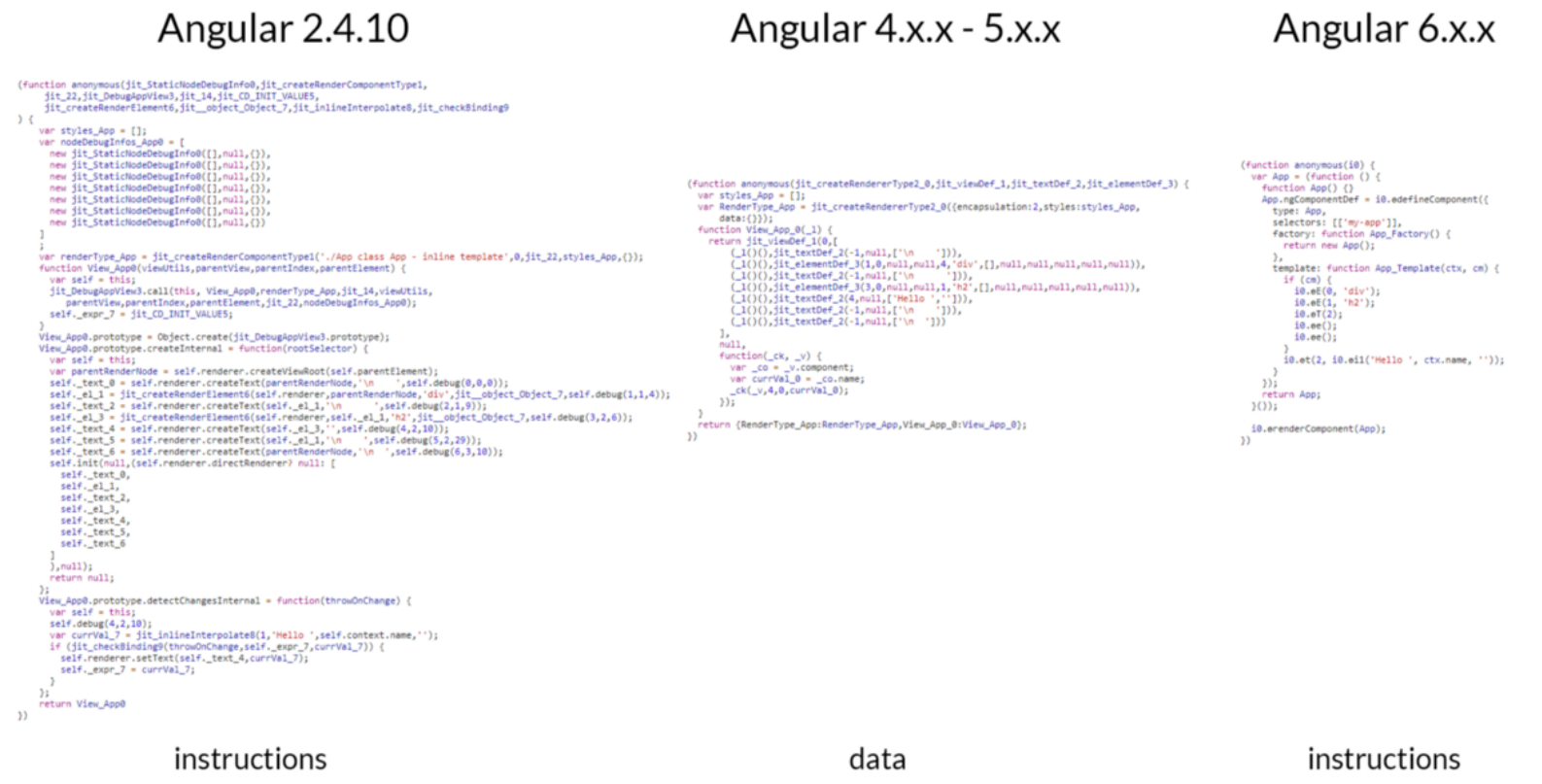
https://alexzuza.github.io/ivy-jit-preview/ 👈

. . .

Let's see what Angular cooks for us

Fan of Angular-In-Depth? Support us on Twitter!

> ***Disclaimer:*** *it is just my learning journey to new Angular renderer*



The Evolution of Angular View Engine

. . .

While new Ivy renderer is not feature completely yet, many people wonder how it will work and what changes it prepares for us.

In this article I am going to visualize Ivy change detection mechanism, show some things I am really excited about and also build simple app based on instructions, similar to angular Ivy instructions, from scratch.

. . .

First, let's introduce the app I'm going to investigate here:

Parent

Child 1

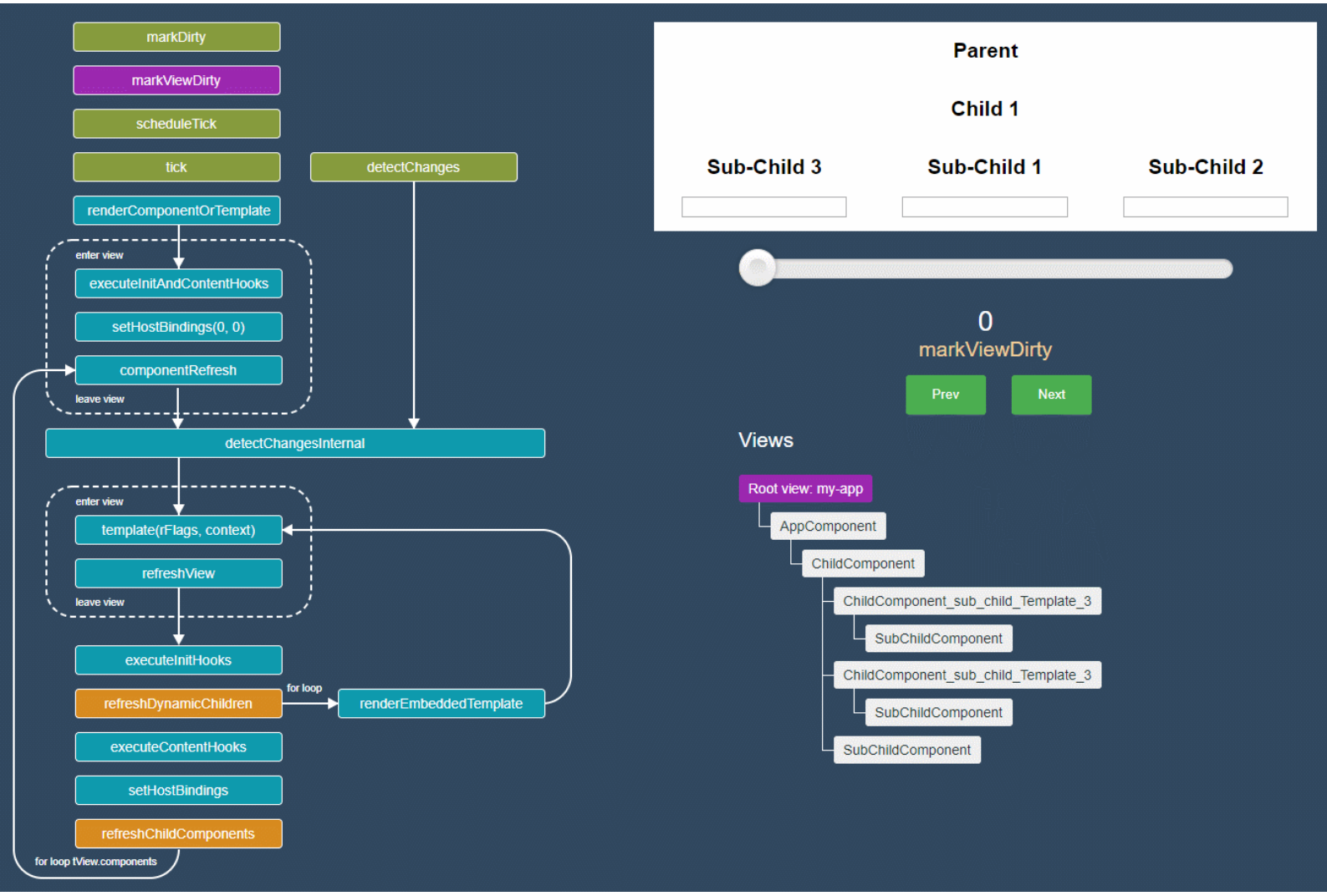
Sub-Child 1 **Sub-Child 2** **Sub-Child 3**

```
@Component({
  selector: 'my-app',
  template: `
   <h2>Parent</h2>
   <child [prop1]="x"></child>
  `
})
export class AppComponent {
  x = 1;
}
@Component({
  selector: 'child',
  template: `
   <h2>Child {{ prop1 }}</h2>
   <sub-child [item]="3"></sub-child>
   <sub-child *ngFor="let item of items" [item]="item"
  `
})
export class ChildComponent {
  @Input() prop1: number;

  items = [1, 2];
}
@Component({
```

I created online demo that I use to understand how it works under the hood:

https://alexzuza.github.io/ivy-cd/ 👈



The demo uses angular 6.0.1 aot compiler. You can click on any lifecycle block to go to the definition.

In order to run change detection process just type something in one of those inputs that are below Sub-Child.

View

Of course, the view is the main low-level abstraction in Angular.

For our example we will get something like:

```
Root view
   |
   |___ AppComponent view
          |
          |__ ChildComponent view
                 |
```

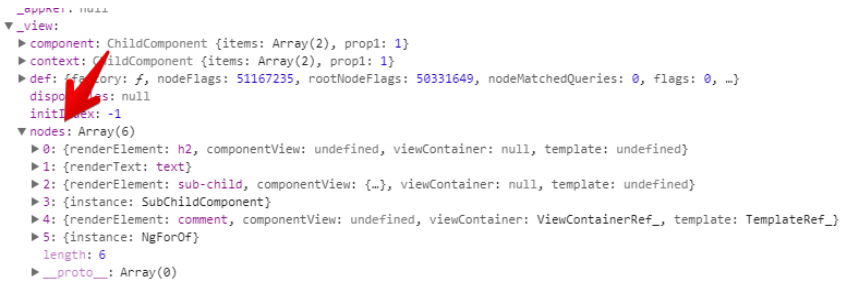
```
                    |_ Embedded view
                    |        |
                    |        |_ SubChildComponent view
                    |
                    |_ Embedded view
                    |        |
                    |        |_ SubChildComponent view
                    |
                    |_ SubChildComponent view
```

View should describe template so it contains some data that will reflect structure of that template.

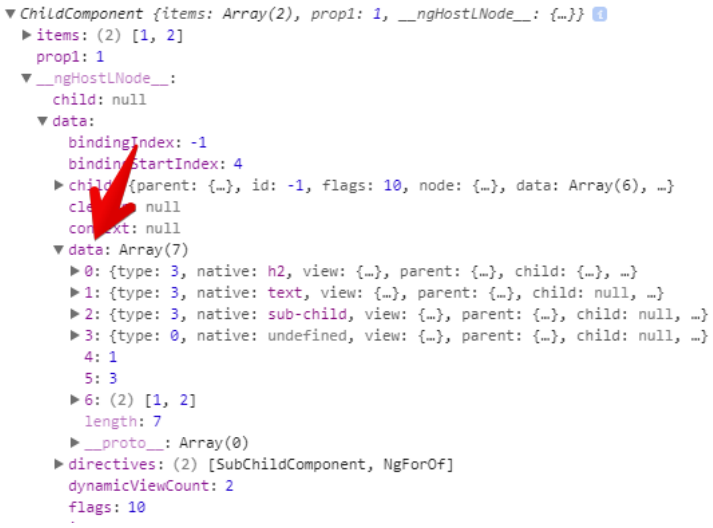
Let's look at `ChildComponent` view. It has the following template:

```
1  <h2>Child {{ prop1 }}</h2>
2  <sub-child [item]="3"></sub-child>
3  <sub-child *ngFor="let item of items" [item]="item"></s
```

Whereas **current view engine creates nodes** from view definition factory and stores them in **nodes** array of view definition



Ivy creates LNodes from instructions, that are written in `ngComponentDef.template` function, and **stores** them in **data** array:



Besides nodes, new view also contains bindings in **data** array(see `data[4]` , `data[5]` , `data[6]` in the picture above). All bindings for a given view are stored in the order in which they appear in the template, starting with `bindingStartIndex`

> *Note how I get view instance from the ChildComponent.* ***ComponentInstance.__ngHostLNode__*** *contains reference to the component host node. (Another way is to inject ChangeDetectorRef)*

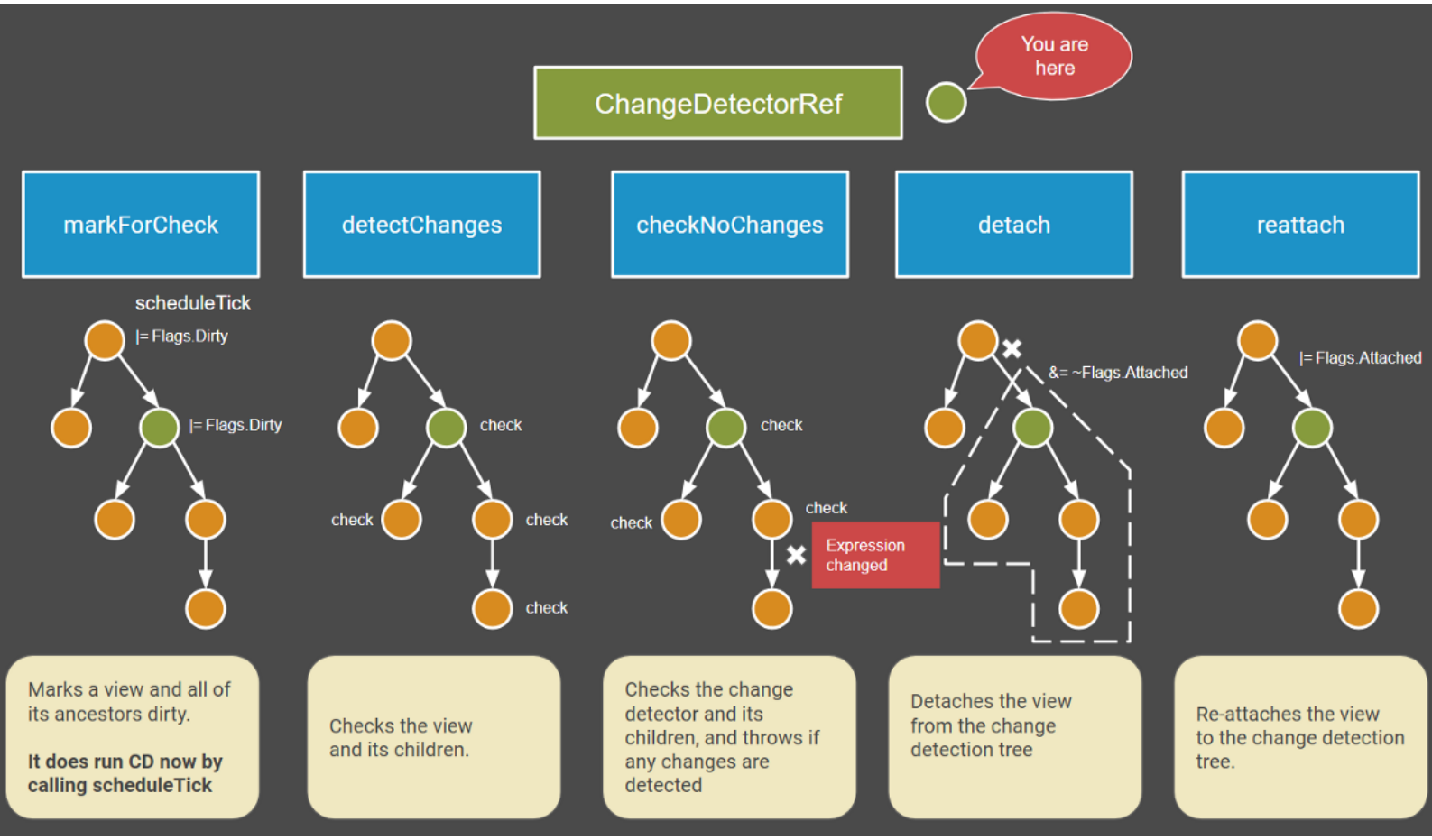
This way angular first creates root view and locate host element at index 0 in `data` array

```
RootView
   data: [LNode]
             native: root component selector
```

and then goes through all components and fills **data** array for each view.

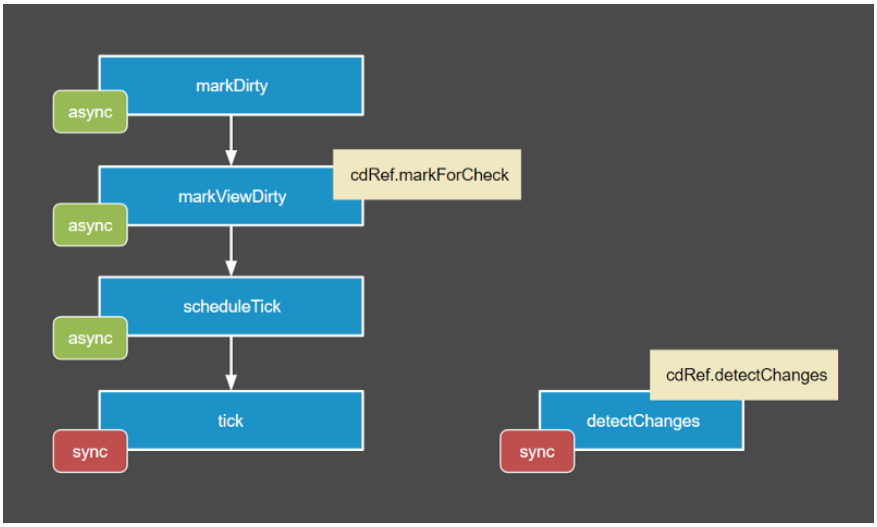
Change detection

Well known ChangeDetectorRef is simply abstract class with abstract methods like `detectChanges` , `markForCheck` , etc.



When we ask this dependency in component constructor we actually gets **ViewRef** instance that extends ChangeDetectorRef class.

Now, let's examine internal methods that are used to run change detection in Ivy. Some of them are available as public api(`markViewDirty` and `detectChanges`) but I am unsure about others.



detectChanges

Synchronously performs change detection on a component (and possibly its sub-components).

> *This function triggers change detection in a synchronous way on a component. There should be very little reason to call this function directly since a preferred way to do change detection is to **use markDirty**(see below) and wait for the scheduler to call this method at some future point in time. This is because a single user action often results in many components being invalidated and calling change detection on each component synchronously would be inefficient. It is better to wait until all components are marked as dirty and then perform single change detection across all of the components*

```
export function detectChanges<T>(component: T): void {
  const hostNode = _getComponentHostLElementNode(compor
  ngDevMode && assertNotNull(hostNode.data, 'Component
  const componentIndex = hostNode.tNode !.flags >> TNoc
  const def = hostNode.view.tView.directives ![componer
  detectChangesInternal(hostNode.data as LView, hostNoc
```

tick

Used to perform change detection on the whole application.

> *This is equivalent to `detectChanges`, but invoked on root component. Additionally, `tick` executes lifecycle hooks and conditionally checks components based on their `ChangeDetectionStrategy` and dirtiness.*

```
export function tick<T>(component: T): void {
  const rootView = getRootView(component);
  const rootComponent = (rootView.context as RootContex
  const hostNode = _getComponentHostLElementNode(rootCo

  ngDevMode && assertNotNull(hostNode.data, 'Component
```

scheduleTick

Used to schedule change detection on the whole application. Unlike `tick`, `scheduleTick` coalesces multiple calls into one change

detection run. It is usually called indirectly by calling `markDirty` when the view needs to be re-rendered.

```
export function scheduleTick<T>(rootContext: RootConte
  if (rootContext.clean == _CLEAN_PROMISE) {
    let res: null|((val: null) => void);
    rootContext.clean = new Promise<null>((r) => res =
    rootContext.scheduler(() => {
      tick(rootContext.component);
      res !(null);
      rootContext.clean = _CLEAN_PROMISE;
```

markViewDirty(markForCheck)

Marks current view and all ancestors dirty.

Whereas early in Angular 5 it only iterated upwards and enabled checks for all parent views, now **please note that markForCheck does trigger change detection cycle in Ivy!!!** 😮 😮 😮

```
export function markViewDirty(view: LView): void {
  let currentView: LView|null = view;

  while (currentView.parent != null) {
    currentView.flags |= LViewFlags.Dirty;
    currentView = currentView.parent;
  }
  currentView.flags |= LViewFlags.Dirty;

```

markDirty

Mark the component as dirty (needing change detection).

> *Marking a component dirty will schedule a change detection on this component at some point in the future. Marking an already dirty component as dirty is a noop. Only one outstanding change detection can be scheduled per component tree. (Two components bootstrapped with separate `renderComponent` will have separate schedulers)*

```
export function markDirty<T>(component: T) {
  ngDevMode && assertNotNull(component, 'component');
  const lElementNode = _getComponentHostLElementNode(cc
  markViewDirty(lElementNode.view);
```

checkNoChanges

Nothing new:)

. . .

When I was debugging new change detection mechanism I noticed that **I forgot to intall zone.js**. And as you have already guessed it worked perfectly without that dependency and without `cdRef.detectChanges` or `tick` . Why?

As you probably know by design Angular triggers change detection for onPush component only if (see my answer on stackoverflow).

These rules are also applied to the Ivy:

- **One of the Inputs changes**
  https://github.com/angular/angular/blob/43d62029f0e2da015 0ba6f09fd8989ca6391a355/packages/core/src/render3/instruc tions.ts#L890
- **A bound event triggered from component or its children**
  https://github.com/angular/angular/blob/43d62029f0e2da015 0ba6f09fd8989ca6391a355/packages/core/src/render3/instruc tions.ts#L1743
- **Manually call markForCheck** (markViewDirty function is responsible for that now(see below))

I have `(input)` output binding in SubChildComponent. The second rule will result in calling **markForCheck.** Since we have already learned that this method actually calls change detection it should be clear now how it works without zonejs.

What about Expression has change after it was checked?

Don't worry, it is still here:)

Change detection order

Since Ivy was announced Angular team has been doing hard work to ensure that the new engine correctly handles all lifecycle hooks in the correct order. That means that the order of operations should be similar.

Max NgWizard K wrote in his great article(strongly suggest reading it):

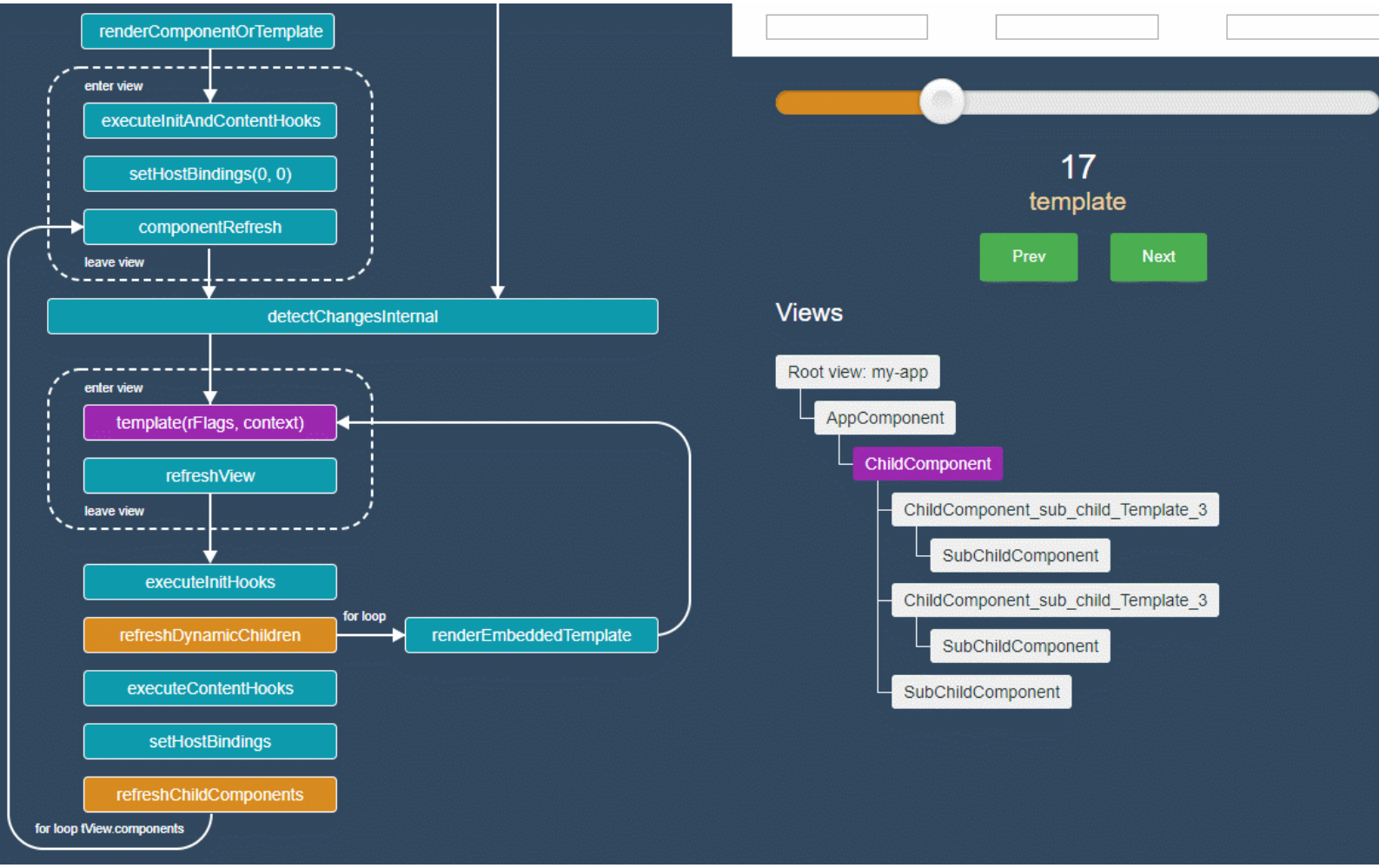
> *As you can see, all the familiar operations are still here. But the order of operations appears to have changed. For example, it seems that now Angular first checks the child components and only then the embedded views. Since at the moment there's no compiler to produce output suitable to test my assumptions, I can't know for sure.*

Let's come back to ChildComponent in my simple app

```
1  <h2>Child {{ prop1 }}</h2>
2  <sub-child [item]="3"></sub-child>
3  <sub-child *ngFor="let item of items" [item]="item"></s
```

It was intended from my side to write one `sub-child` as regular component before others that are inside embedded view.

Now it's time to see it in action:



As we can angular first checks embedded view and then regular component. So there is no changes here from previous engine.

Anyway, there is optional “run Angular compiler” button in my demo and we can test other cases.

https://alexzuza.github.io/ivy-cd/

One-time string initialization

Imagine we wrote component that can receive color as string input value. And now we want to pass that input as constant string that will never be changed:

```
1  <comp color="#efefef"></comp>
```

It's so called one-time string initialization and angular documentation states:

> *Angular sets it and forgets about it.*

As for me, it means that angular won't do any additional checks for this binding. But what we actually see in angular5 is that it is checked per every change detection cycle during `updateDirectives` call.

```
function updateDirectives(_ck,_v) {
   var currVal_0 = '#efefef';
   _ck(_v,1,0,currVal_0);
```

> *See also great article "Getting to Know the @Attribute Decorator in Angular" about this issue by Netanel Basal*

Now let's look at how it is supposed to be in new engine:

```
var _c0 = ["color", "#efefef"];
AppComponent.ngComponentDef = i0.ɵdefineComponent({
  type: AppComponent,
  selectors: [["my-app"]],
  ...
  template: function AppComponent_Template(rf, ctx) {
    // create mode
      if (rf & 1) {
        i0.ɵE(0, "child", _c0); <========== used only
        i0.ɵe();
      }
```

As we can see angular compiler stores our constant outside of the code that is responsible for creating and updating component and **only uses this value in create mode.**

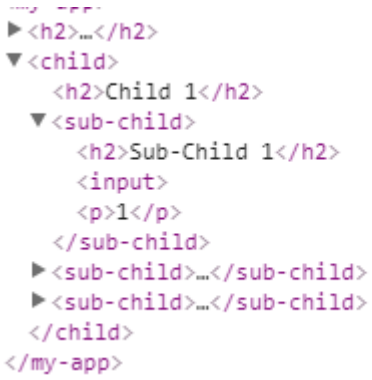
Angular no longer creates text nodes for containers

> ***Update:*** *https://github.com/angular/angular/pull/24346*

Even if you don't know how angular ViewContainer works under the hood you may noticed the following picture when opening devtools:

> *In production mode we see only* `<!---->` .

And here's the Ivy output:



I can't be sure 100% but seems we will have such result once Ivy gets stable.

As a result the `query` in the code below

```
@Component({
  ...,
  template: '<ng-template #foo></ng-template>'
})
class SomeComponent {
  @ViewChild('foo', {read: ElementRef}) query:
```

will return `null` since angular

> *should no longer read ElementRef with a native element pointing to comment DOM node from containers*

Incremental DOM(IDOM) from scratch

A long time ago Google announced so-called incremental DOM library.

The library focuses on building DOM trees and allowing dynamic updates. It wasn't intended to be used directly but as a compilation target for template engines. And seems **the IVy has something in common with incremental DOM library**.

Let's build simple app from scratch that will help us to understand how IDOM render works. **Demo**

Our app will have counter and also print user name that we will by typing in input element.

Assume we already have `<input>` and `<button>` element on the page:

```
<input type="text" value="Alexey">
<button>Increment</button>
```

Existing html page

And all we need to do is to render dynamic html that will look like:

```
<h1>Hello, Alexey</h1>
<ul>
  <li>
    Counter: <span>1</span>
  </li>
```

In order to render this let's write **elementOpen**, **elementClose** and **text** "instructions" (I call it this way because Angular uses such names as IVy can be considered as special kind of virtual CPU).

First we need to write special helpers to traverse nodes tree:

```
1  // The current nodes being processed
2  let currentNode = null;
3  let currentParent = null;
4
5  function enterNode() {
6    currentParent = currentNode;
7    currentNode = null;
8  }
9  function nextNode() {
10   currentNode = currentNode ?
11     currentNode.nextSibling :
12     currentParent.firstChild;
```

Now, let's write instructions:

```
function renderDOM(name) {
  const node = name === '#text' ?
        document.createTextNode('') :
    document.createElement(name);

  currentParent.insertBefore(node, currentNode);

  currentNode = node;

  return node;
}

function elementOpen(name) {
  nextNode();
  const node = renderDOM(name);
  enterNode();

  return currentParent;
}

function elementClose(node) {
  exitNode();
```

Put differently, these functions just walk through DOM nodes and insert node at current position. Also text instruction sets `data` property so that we can see text value the browser.

We want our elements to be capable of keeping some state, so let's introduce `NodeData`:

```
const NODE_DATA_KEY = '__ID_Data__';

class NodeData {
  // key
  // attrs

  constructor(name) {
    this.name = name;
    this.text = null;
  }
}

function getData(node) {
```

Now, let's change our `renderDOM` function so that we won't add new element to the DOM if there is already the same at current position:

```
const matches = function(matchNode, name/*, key */) {
  const data = getData(matchNode);
  return name === data.name // && key === data.key;
};

function renderDOM(name) {
  if (currentNode && matches(currentNode, name/*, key
    return currentNode;
  }
```

Note my comment `/*, key */` . It would be better if our elements have some key to distinguish elements. See also http://google.github.io/incremental-dom/#demos/using-keys

After that let's add logic that will be responsible for text node updates

```
function text(value) {
  nextNode();
  const node = renderDOM('#text');

  // update
  // checks for text updates
  const data = getData(node);
  if (data.text !== value) {
    data.text = (value);
    node.data = value;
  }
```

The same we can do for element nodes.

Then let's write **patch** function that will take **DOM element**, **update function** and some **data** that will be consumed by update function:

```
function patch(node, fn, data) {
  currentNode = node;

  enterNode();
  fn(data);
  exitNode();
```

Finally, let's test our instructions:

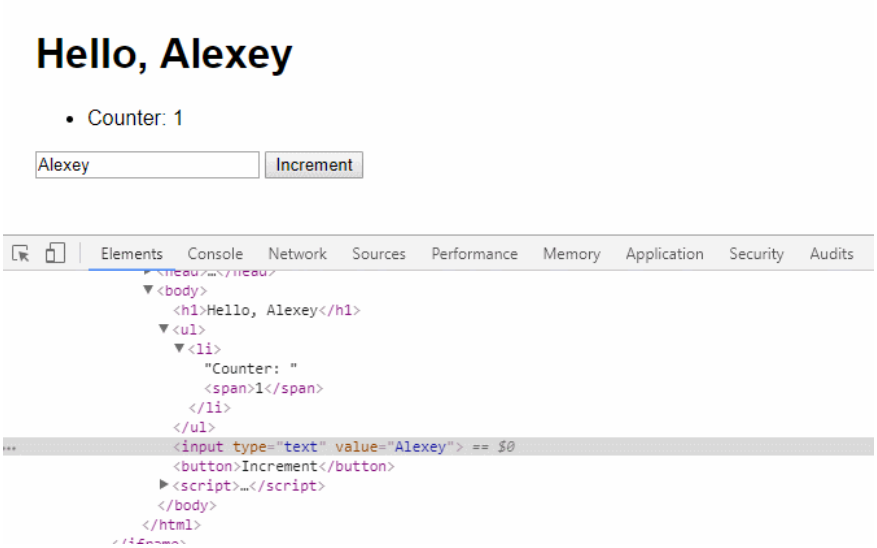
```
function render(data) {
  elementOpen('h1');
  {
    text('Hello, ' + data.user)
  }
  elementClose('h1');
  elementOpen('ul')
  {
    elementOpen('li');
    {
      text('Counter: ')
      elementOpen('span');
      {
        text(data.counter);
      }
      elementClose('span');
    }
    elementClose('li');
  }

  elementClose('ul');
}

document.querySelector('button').addEventListener('cli
   data.counter ++;
```

The result can be found here

You can also verify that the code will only update the text node whose contents has changed by inspecting the with the browser tools:



So the main concept of IDOM is **to just use the real DOM to diff against new trees.**

That's all. Thanks for reading...

• • •