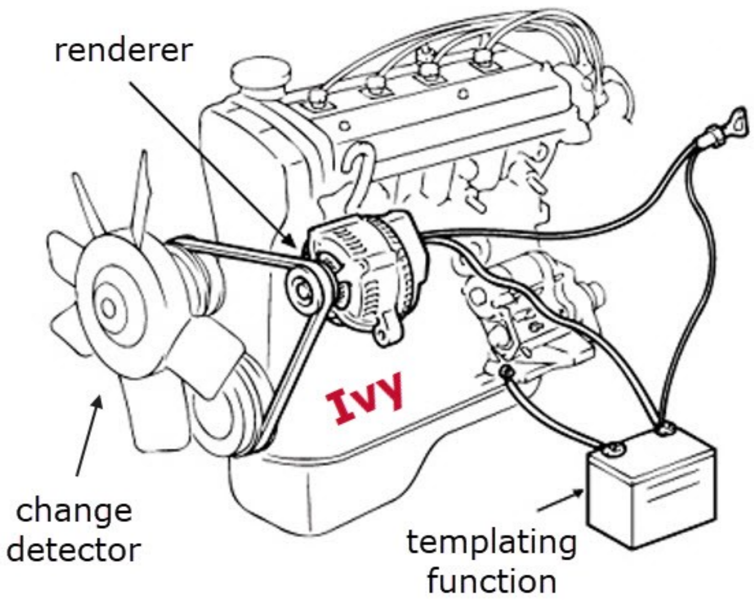
Ivy engine in Angular: first in-depth look at compilation, runtime and change detection

Max Koretskyi aka Wizard Follow
May 15, 2018 · 12 min read



> **We're organizing the first Angular conference in Kiev, Ukraine as part of Angular In Depth community. Read more about it here.** I'll be happy to see you at the conference. We'll have experts from Angular team and the community and we made tickets affordable for most developers (around $150), even if you pay out of your own pocket.

I usually finish my talks with the philosophical phrase that *nothing stays the same*. And as you probably know it's more then true with Angular. The current rendering engine is being rewritten with the new much enhanced version called Ivy. The current status of Ivy can be tracked here. Today, we're going to take a look under the hood of this marvel of complex framework engineering.

I'm going to be honest with you. When I first heard about the new rendering engine I got slightly upset. This is because I'd spent a great deal of time reverse-engineering the current implementation and it now meant that the knowledge I had acquired could potentially become obsolete. I've also written a lot of articles explaining the internals. You might have read some of them and are now wondering just as I was if the knowledge you have is still relevant.

But no worries! I partially reverse-engineered the new implementation again and I have good news for you. **Although most of the underlying implementation has completely changed, the main concepts like a component view is still there**. Discovering this was quite comforting because most of my articles on DOM manipulation and change detection are based on the concept of a view. It looks different now and is used differently, but it's still there. And even the change detection operations are still the same, although the order is slightly changed. So don't panic and stay cool 😎 .

In this article I'll shed some light on the new runtime, compilation and change detection processes. But before we get to the implementation details, I want to spend a few paragraphs answering the question that always pops up in my head when I come across changes. It's this eternal question: *Why?*

> I work as a developer advocate at **ag-Grid**. If you're curious to learn about data grids or looking for the ultimate Angular data grid solution, give it a try with the guide "**Get started with Angular grid in 5 minutes**". I'm happy to answer any questions you may have. **And follow me to stay tuned!**

The Why

During the keynote presentation at NgConf 2018 Misko and Kara mentioned two main ideas that played a major role when engineering Ivy—locality and tree shaking. This is something that the current implementation lacks. Also, it wasn't mentioned, but it's clear that the Angular Elements project significantly contributed to the new Ivy design.

Locality

The idea behind locality is that a compiler is only allowed to use information defined by a component decorator and its class. In contrast, the current implementation requires a global static analysis of the entire application. This has a number of downsides.

First, it makes it difficult to combine compilation outputs from separate projects. For example, you will need to have really advanced knowledge of a compiler to integrate an AOT compiled library into a JIT compiled application. Restricting a compiler to only use the information provided by a component decorator enables shipping AOT code to NPM as a standalone library and significantly simplifies integration of JIT and AOT packages.

Also, the current compilation process produces some artifacts in the form of extra files `metadata.json` and `component.factory.js` that require elaborate handling. The new Ivy engine compiler will not produce them.

And lastly, the new implementation should also make the process of creating dynamic components on the fly easier than the current approach.

Tree shaking

The other concept that laid the foundation for Ivy is tree shaking. It simply means that unused code is not included in a bundle during the build process. Probably, we are all familiar with tree shaking when it comes to our application code. But Angular developers went further and wondered if some of the framework could be tree shaken. For example, if you don't use view queries, you don't need to ship the Angular code that updates these queries to a browser. Don't need content projection? No problem, it won't be included in a bundle as well. In fact, that's where the significant bundle size reduction comes from. You no longer ship the entire framework code, you only bundle pieces of the framework functionality that you use! And certainly smaller bundles have a lot of benefits, for example faster startup time.

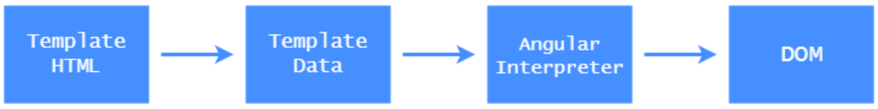
It all becomes possible due to the new instruction based approach used inside a compiled version of a component. And besides making bundles smaller, this approach also brings other benefits like simplified debugging.

Runtime

The new runtime engine is based on the concept of **Incremental DOM**. It's a way to express and apply updates to DOM trees using instructions. DOM updates is the main part of change detection in Angular so this concept can be conveniently applied in the framework. I encourage you to read more about it in this article which explains the reasoning behind the concept and contrasts it with **Virtual DOM** in React. Incremental DOM also happens to be a library, but the new Ivy engine doesn't use it and instead implements its own version.

As Kara explained in the keynote talk, the logic that instantiates components, creates DOM nodes, and runs change detection is implemented as an atomic unit she referred to as Angular Interpreter. A compiler simply produces metadata about a component and elements defined in its template. It's the interpreter that does the main job. It uses this data to instantiate components and run change detection.

This diagram illustrates the current execution model:



In the illustration above the Template Data is a view definition produced by the compiler. The definition holds metadata relevant to a component and is used as a blueprint when creating a component view. In its current form the view definition (template data) is not particularly human readable. Here is, for example, a definition generated for a component with the template `<span>My name is {{name}}</span>` :

```
viewDef(0,[
    elementDef(0,null,null,1,'span',...),
    textDef(null,['My name is ',...])
]
```

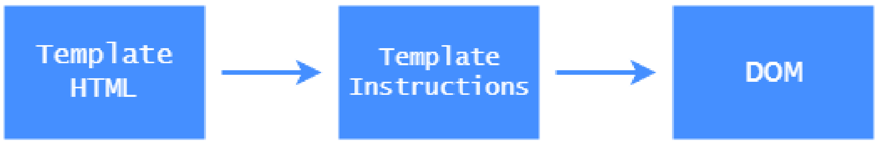
It defines two view nodes—a `span` element and a text node with the static text `My name is` . This information is then interpreted at runtime to create two DOM nodes and provide information for the text binding operation during change detection. Compare that with the current implementation in Ivy:

```
// create mode
if (rf & RenderFlags.Create) {
    elementStart(0, 'span');
    text(1);
    elementEnd();
}
// update mode
if (rf & RenderFlags.Update) {
```

Here, the `elementStart` instruction doesn't define any metadata but instead directly creates a DOM node. If you follow the link, you'll notice that Ivy still uses Renderer: a concept I talked elaborately about at NgVikings. So again, familiar concepts are still here 😄 .

And the same goes for the `text` instruction that creates a text node. And we also have the `textBinding` instruction that actually implements one of the change detection operations—bindings update on a text element. That is an *incremental* update of one DOM node at a time hence the name Incremental DOM.

Here is a diagram that depicts the new execution model in Ivy:



So template instructions is where the logic that instantiates components, creates DOM nodes, and runs change detection lives now. It's been moved from the monolith interpreter into individual instructions.

There are instructions that create standard DOM nodes. They are executed as part of the change detection process in the creation mode. This means that DOM nodes are created as part of change detection too which differs from the current implementation. Interestingly, some creational instructions also setup context for change detection—like `elementEnd` that queues lifecycle hooks and adds entries to a query list. There's also a bunch of instructions that create logical view nodes specific to Angular like directives, view containers and queries.

Another category is comprised of instructions that are executed during the change detection process in the update mode. Among them is `textBinding` that updates a text node, and `elementProperty` that updates a property on an element.

> *Since these are simply functions that can be imported, if you don't use element bindings like `<span [textContent]="value">` in your application, the instruction (runtime code) to process that binding does not need to be imported and hence is not included in a bundle (tree shaking).*

Compilation

Similar to the current implementation, the role of a compiler in Ivy is to take metadata provided by a component decorator and produce a component definition. Here is a very simple component definition generated for the template `<my-app [name]=”name”></my-app>` :

```
const componentDefinition = {
    type: MyApp,
    selectors: [['my-app']],
    template: (rf: RenderFlags, ctx: MyApp) => {
        if (rf & RenderFlags.Create) {
            elementStart(0, 'span');
            elementEnd();
        }
        if (rf & RenderFlags.Update) {
            elementProperty(0, 'name', bind(ctx.name))
```

The definition provides information about the type of a component, the selectors it matches, and a factory function used to instantiate a component class. There are many other properties in a definition and you can get familiar with them using the sources. Of particular interest to us is the `template` property which defines a function executed during each change detection cycle. That function houses the creational and change detection instructions I introduced above. Angular runs the `template` function either in the `create` or `update` mode. Given the instruction in the definition above Angular will create a `span` element in the `create` mode or update its bindings in the `update` mode.

One of the most exciting things about Ivy is that how easy it is to debug change detection. You simply put a breakpoint inside the template function and that's it—you're effectively debugging a change detection run for the current component!

Storing component definition

In the current compiler implementation a component definition (template data) lives in its own file independently of a component class. These are factory files `*.component.factory.ts` that you've probably come across if you used AOT. In the new Ivy compiler a component definition will be attached to a component class through static fields. No separate file will be created during compilation. At the time of this writing, the component definition is stored in `ngComponentDef` static field:

```
export class MyApp {
    name: string;
    static ngComponentDef = defineComponent({
        type: MyApp,
        selectors: [['my-app']],
        template: function() {...},
        factory: () => new MyApp()
```

You can learn that from a document that describes the new compiler architecture. The existence of such documents is a huge aid in the task of reverse-engineering. Kudos to Chuck Jazdzewski for putting it together. Here are some interesting excerpts from the document that can help in understanding the new design.

> *...the Ivy model is that Angular decorators (* `@Injectable` *, etc) are compiled to static properties on the classes (* `ngInjectableDef` *)... Each of the class decorators... creates a corresponding static member on the class that describes to the runtime how to use the class. For example, the* `@Component` *decorator creates an* `ngComponentDef` *static member,* `@Directive` *create an* `ngDirectiveDef` *, etc...*

So, alongside the static field `ngComponentDef` on a component class, we can expect a bunch of other static fields like `ngInjectableDef` and `ngPipeDef` that hold definitions corresponding to providers and pipes used by a component.

> *Each of the class decorators can be thought of as class transformers that take the declared class and transform it...This operation must take place without global program knowledge, and in most cases*

> *only with knowledge of that single decorator… Internally, these class transformers are called a "Compiler"*

What this tells us is that the new Angular compiler will apply a bunch of independent TypeScript class transforms to an AST representing a component class. These class transforms or transformers are implemented as a sort of pure function that takes decorator metadata and adds a definition as a static field to a component class. A few of these class transformers referred to internally as compilers are already implemented. The document also explicitly states that the transformation should take place without global program knowledge which is dictated by the principle of locality.

Change detection

The **Change Detection** process is my main area of interest in Angular. I've studied it and I have written extensively about it in the past. Naturally, I was anxious to learn what has changed and if everything I've learnt so far is even relevant anymore. Yet, similarly to how the concepts that guided the design of AngularJS survived the re-write to Angular, the concepts and operations that constitute the change detection process remained the same. As we've seen they migrated from the Angular interpreter to a component template function. And now they are implemented as individual instructions (functions) as opposed to an atomic unit that can't be tree shaken. The order of executions might be a bit different and the mechanism for lifecycle hooks has changed, but the same mental model I explained in my earlier articles applies as well for Ivy.

Internally the change detection run in Ivy is performed by calling the `detectChanges` function and passing in a component class. This function probably will not be exposed as a public API but rather wrapped inside a familiar abstraction like ChangeDetectorRef.

The function acts mostly as a wrapper around `detectChangesInternal` that takes a component view and actually runs the check:

```
export function detectChangesInternal(view, hostNode,
    const oldView = enterView(view, hostNode);
    const template = def.template;

    try {
        template(getRenderFlags(view), component);
        refreshDirectives();
        refreshDynamicChildren();
    } finally {
```

But the function itself is pretty slim now. Contrast it to the current implementation referred earlier as runtime "interpreter" that takes a view and executes all change detection operations:

```
export function checkAndUpdateView(view: ViewData) {
    // update child element and components inputs
    Services.updateDirectives(view, CheckType.CheckAnd
    // run change detection for embedded views
    execEmbeddedViewsAction(view, ViewAction.CheckAndU
    // update ContentChild & ContentChildren queries
    execQueriesAction(...);
    // calls AfterContentInit & AfterContentChecked li
    callLifecycleHooksChildrenFirst(...);
    // update bindings
    Services.updateRenderer(...)
    // run change detection for child components
    execComponentViewsAction(...);
```

Now in Ivy most of it is replaced by a call of the `template` function:

```
try {
    // template function defined by a component definit
    template(getRenderFlags(view), component);
    ...
```

And we already know, this template function holds a bunch of instructions that execute change detection related operations, like the text and input bindings update.

Besides calling the `template` function, the `detectChangesInternal` also contains two other function calls: `refreshDirectives` and `refreshDynamicChildren`. The `refreshDirectives` basically triggers change detection for child components. References to child components are stored on a component view similarly to how it's done today. The `refreshDirectives` function is also used to trigger the `NgOnInit` lifecycle hook. The other function `refreshDynamicChildren` runs change detection for embedded views stored inside a view container.

So if we annotate the new `detectChangesInternal` function it will look something like this:

```
export function detectChangesInternal(view, hostNode,
    ...
    try {
        /*
        runs template function that executes instructi
         - updating child elements, directives and com
         - updating text bindings
         - refreshing view and content queries
        */
        template(getRenderFlags(view), component);
        // runs change detection for child components
        // and executes init and content life cycle ho
```

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. I'll wait until Ivy is at least in beta to provide detailed explanation of the new change detection process. Stay tuned!

NgOnChanges

I was also very eager to take a look at the implementation of the `NgOnChanges` hook in Ivy as this translated article claims it's not a real lifecycle hook anymore:

> *So it is notable that* `OnChanges` *in Ivy is not a real lifecycle any more… Put differently, users can extend lifecycle themselves, do as they please… It is no longer right to say Angular is based solely on dirty check*

That's actually a bit of a confusing message. In the existing implementation the hook is called during change detection right after Angular updates input bindings. In Ivy, the hook is called inside the `refreshDirectives` function after Angular has executed the template function and the bindings for child components and the elements have been updated. So the call of `NgOnChanges` is still part of change detection and can be relied upon to be notified when some input bindings have changed.

The mechanics of this particular hook is indeed different from the other hooks. As the article states, it's implemented as a feature—a mechanism to intercept the directive definition and modify it. So basically the `NgOnChangesFeature` that implements the hook adds a wrapper around `ngDoCheck` lifecycle hook that calls `NgOnChanges` first if there's a change, and then the `ngDoCheck` lifecycle hook (named `delegateHook` below):

```
componentDefinition.doCheck = onChangesWrapper(definit
...
function onChangesWrapper(delegateHook: (() => void) |
    return function(this: OnChangesExpando) {
        let simpleChanges = this[PRIVATE_PREFIX];
        if (simpleChanges != null) {
            // calls NgOnChanges hook
            this.ngOnChanges(simpleChanges);
            this[PRIVATE_PREFIX] = null;
        }
```

But again, the `ngOnChanges` hook is called just before the `ngDoCheck` hook, so it's consistent with the existing implementation.

I can presume that by saying "it is no longer right to say Angular is based solely on dirty check", the author meant that any other feature can bind to the `ngDoCheck` hook and call the `NgOnChanges` even without inputs being changed. But that assumes that other parties are allowed to introduce new `features` . And if so, I assume that there's going to be some guards implemented in the framework against custom calls of `NgOnChanges` .

· · ·

Bootstrapping

Ivy provides a simpler API now to bootstrap a component. In the existing implementation we need to use `NgModule` that defines a component to bootstrap an application with:

```
@NgModule({
    ...
    bootstrap: [AppComponent]
})
export class AppModule {}
```

In Ivy the `bootstrap` function takes this bootstrap component directly:

```
renderComponent(AppComponent);
```

· · ·

Thanks for reading! If you liked this article, hit that clap button below 👏. It means a lot to me and it helps other people see the story.

For more insights follow me on Twitter and on Medium.

· · ·