Inside Ivy: Exploring the New Angular Compiler

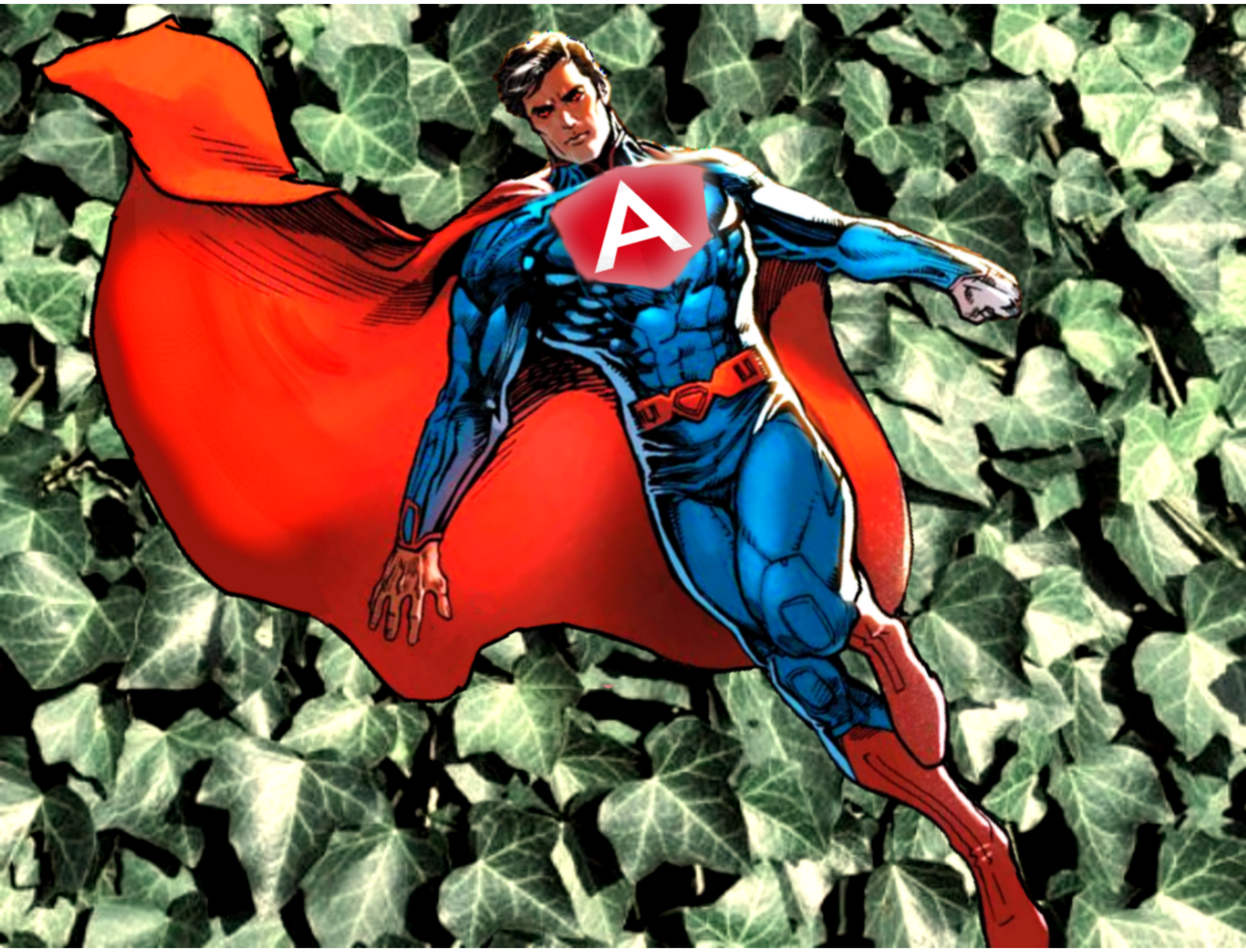
If you had fun with the “Deep, Deep, Deep” Dive into the Angular Compiler, just wait ‘till you get your hands on Ivy!

Uri Shaked Follow
Aug 8, 2018 · 12 min read

I think compilers are pretty neat. Last year, I wrote “A Deep, Deep, Deep, Deep, Deep Dive into the Angular Compiler,” where I reverse engineered the Angular Compiler and simulated some parts of the compilation process in order to understand how it works behind the scenes.

Thus, when I heard the announcement about a new compiler version, called Ivy, I immediately saw an opportunity to look under the hood and see what has changed.

Ivy, just like the earlier versions of the Angular compiler, takes the templates and components we write in Angular and compiles them into plain-Jane HTML and JavaScript so that Chrome and other browsers can show the world our genius.


As well as our Angular Super Powers!

The cool thing about Ivy versus the older compilers, however, is that it's "tree-shaking friendly," which basically means that it automatically removes unused bits of code (including unused Angular features!), shrinking your bundles. Another upgraded feature is that it compiles each file separately, reducing rebuild times. In short—you get smaller builds, faster rebuild times and simpler generated code.

Understanding how the compiler works is interesting in and of itself (or at least, I think so :-), but it also helps us better understand the internals of Angular, teaching us how to "think" Angular that much better, which in turn makes us better at writing it!

So I am now excited to present: Ivy, the new Angular Compiler, inside out!

Running Ivy

Ivy is still under heavy development, and you can track its status here. While the compiler itself is still not ready for prime time, the run time that will be useful by the compiler, called the “RendererV3,” is already functional and shipping with Angular 6.x.

And even though it’s not 100% ready to go, we can still peek at the compiled code (just like we did with the previous compiler). How do we do it? By creating a new Angular project:

```
ng new ivy-internals
```

And then enabling Ivy by adding the following lines to the `tsconfig.json` file in the new project folder:

```
"angularCompilerOptions": {
  "enableIvy": true
}
```

And finally, we run the compiler by executing `ngc` command inside the newly created project folder:

```
node_modules/.bin/ngc
```

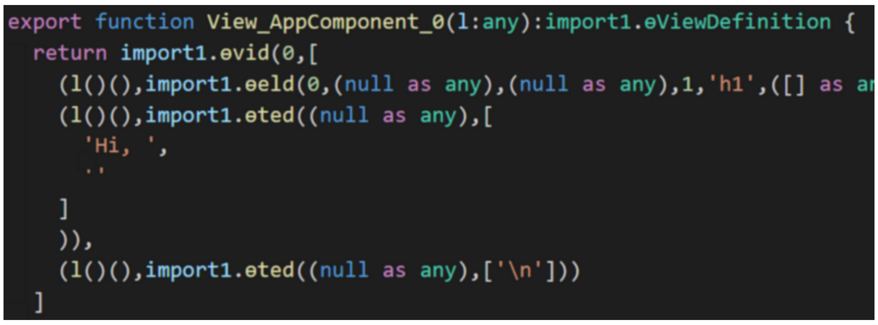
That’s it! We can inspect the generated code inside the `dist/out-tsc` folder. For example, let’s look at the following excerpt from the template of `AppComponent` :

```
<div style="text-align:center">
  <h1>
    Welcome to {{ title }}!
  </h1>
  <img width="300" alt="Angular Logo" src="…">
</div>
<h2>Here are some links to help you start: </h2>
```

We can find the code generated for this template inside by looking inside `dist/out-tsc/src/app/app.component.js` :

```
    i0.ɵE(0, "div", _c0);
    i0.ɵE(1, "h1");
    i0.ɵT(2);
    i0.ɵe();
    i0.ɵE(3, "img", _c1);
    i0.ɵe();
    i0.ɵe();
    i0.ɵE(4, "h2");
    i0.ɵT(5, "Here are some links to help you start: ");
    i0.ɵe();
```

This is actually how the component template is compiled to JavaScript code by Ivy. Contrast this with the way the previous compiler did it:



```
export function View_AppComponent_0(l:any):import1.ɵViewDefinition {
  return import1.ɵvid(0,[
    (l()(),import1.ɵeld(0,(null as any),(null as any),1,'h1',([] as ar
    (l()(),import1.ɵted((null as any),[
      'Hi, ',
      '.'
    ]
    )),
    (l()(),import1.ɵted((null as any),['\n']))
  ]
```

“null as any,” anyone?

Seems like the code Ivy generate is much simpler! You can tinker with the component template (in `src/app/app.component.html`) and run the compiler again, and observe how your modifications are reflected in the generated code.

Understanding the Generated Code

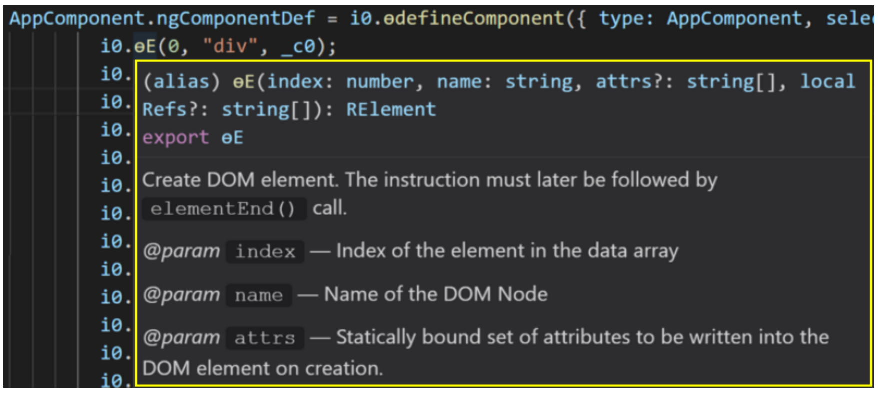
Let’s try to tease apart the generated code and see what it’s doing—for example, what are all these `i0.ɵE` , `i0.ɵT` calls?

If we look at the top of the generated file, we find the following statement:

```
var i0 = require("@angular/core");
```

So `i0` is actually just the core Angular module, and these are all functions exported by Angular. The letter ɵ is used by the Angular team to indicate that some method is private to the framework and must not be called directly by the user, as the API for these method is not guaranteed to stay stable between Angular versions (in fact, I would say it's almost guaranteed to break).

So all these methods are private APIs exported by Angular. We can easily figure out what they do if we open the project in Visual Studio Code and look at the tool tips:



```
AppComponent.ngComponentDef = i0.ɵdefineComponent({ type: AppComponent, sele
        i0.ɵE(0, "div", _c0);
        i0.
        i0.
        i0.
        i0.
        i0.
        i0.
        i0.
        i0.
        i0.
        i0.
        i0.
        i0.
```

```
(alias) ɵE(index: number, name: string, attrs?: string[], local
Refs?: string[]): RElement
export ɵE
```

Create DOM element. The instruction must later be followed by `elementEnd()` call.

@param `index` — Index of the element in the data array

@param `name` — Name of the DOM Node

@param `attrs` — Statically bound set of attributes to be written into the DOM element on creation.

Even though this file is a JavaScript file, Visual Studio Code uses the type information from TypeScript to figure out the call signature and documentation for this method. If we Ctrl+Click the method name(Cmd-Click for Mac), we discover that its original name was `elementStart`!

Similarly, `ɵT` stands for `text`, and `ɵe` for `elementEnd`. Armed with this knowledge, we can "translate" the generated code into something more readable—here is a short snippet:

```
var core = require("angular/core");
//...
core.elementStart(0, "div", _c0);
core.elementStart(1, "h1");
core.text(2);
core. ();
core.elementStart(3, "img", _c1);
core.elementEnd();
core.elementEnd();
core.elementStart(4, "h2");
core.text(5, "Here are some links to help you start: ");
core.elementEnd();
```

And as mentioned above, this generated code corresponds to the following text from the html template:

```
<div style="text-align:center">
  <h1>
    Welcome to {{ title }}!
  </h1>
  <img width="300" alt="Angular Logo" src="…">
</div>
<h2>Here are some links to help you start: </h2>
```

We can easily see that:

- Each HTML opening tag maps into an `core.elementStart()` call
- Each closing tag maps into an `core.elementEnd()` call
- And each text node maps into a `core.text()` call

The first argument of `elementStart` and `text` method is a number whose value increments in each call —probably some index into an array where Angular stores references to the created element.

`elementStart` also received a 3rd argument—by looking at the docs above, we learn that it's optional and contains the list of attributes for the DOM node. We can verify it by looking at the value of `_c0` and seeing that it indeed contains the list of attributes and their values for the `div` element:

```
var _c0 = ["style", "text-align:center"];
```

A Quick Note About ngComponentDef

So far, we looked into a part of the generated code that renders the template for the component. This code actually lives in a larger block of code that get assigned to `AppComponent.ngComponentDef` —a static property that contains all the meta-data about the component, such as the CSS selector(s) for the component, its change detection strategy (if specified) and the template. If you feel adventurous, you can try figuring out how this works by yourself now—or keep reading and we will get there shortly.

Do-It-Yourself Ivy

Now that we have a basic understanding of what the generated code looks like, we can try to craft our own component, from scratch, using the very same RendererV3 API the Ivy compiler uses!

The code we are going to create will be similar to the code emitted by the compiler, but we will write it in a more human-readable fashion.

Let's start by writing a simple component and then manually translating it into Ivy-like code:

```
import { Component } from '@angular/core';

@Component({
  selector: 'manual-component',
  template: '<h2>Hello, Component</h2>',
})
```

The compiler takes as an input @Component decorator information, generates instructions and then defines it all as a static property on the component class. So, to simulate what Ivy does, we'll remove the `@Component` decorator and replace it with a static `ngComponentDef` property:

```
import * as core from '@angular/core';

export class ManualComponent {
  static ngComponentDef = core.ɵdefineComponent({
    type: ManualComponent,
    selectors: [['manual-component']],
    factory: () => new ManualComponent(),
    template: (rf: core.ɵRenderFlags, ctx: ManualCompo
      // Compiled template goes here
```

We define the metadata for the compiled component by calling `ɵdefineComponent` . The metadata includes the type of the component (used later for dependency injection), the CSS selector(s) that will invoke this component (`manual-component` in our case—that's the name of the component in the HTML template of other components), a factory that returns a new instance of the component and then the function that defines the template for the component. This template renders the view for component and updates it whenever any of the properties of the components change. We will use the methods we found above, `ɵE` , `ɵe` and `ɵT` to write this template:

```
    template: (rf: core.ɵRenderFlags, ctx: ManualCompor
      core.ɵE(0, 'h2');                  // Open h2 elem
      core.ɵT(1, 'Hello, Component');    // Add text
      core.ɵe();                         // Close h2 ele
```

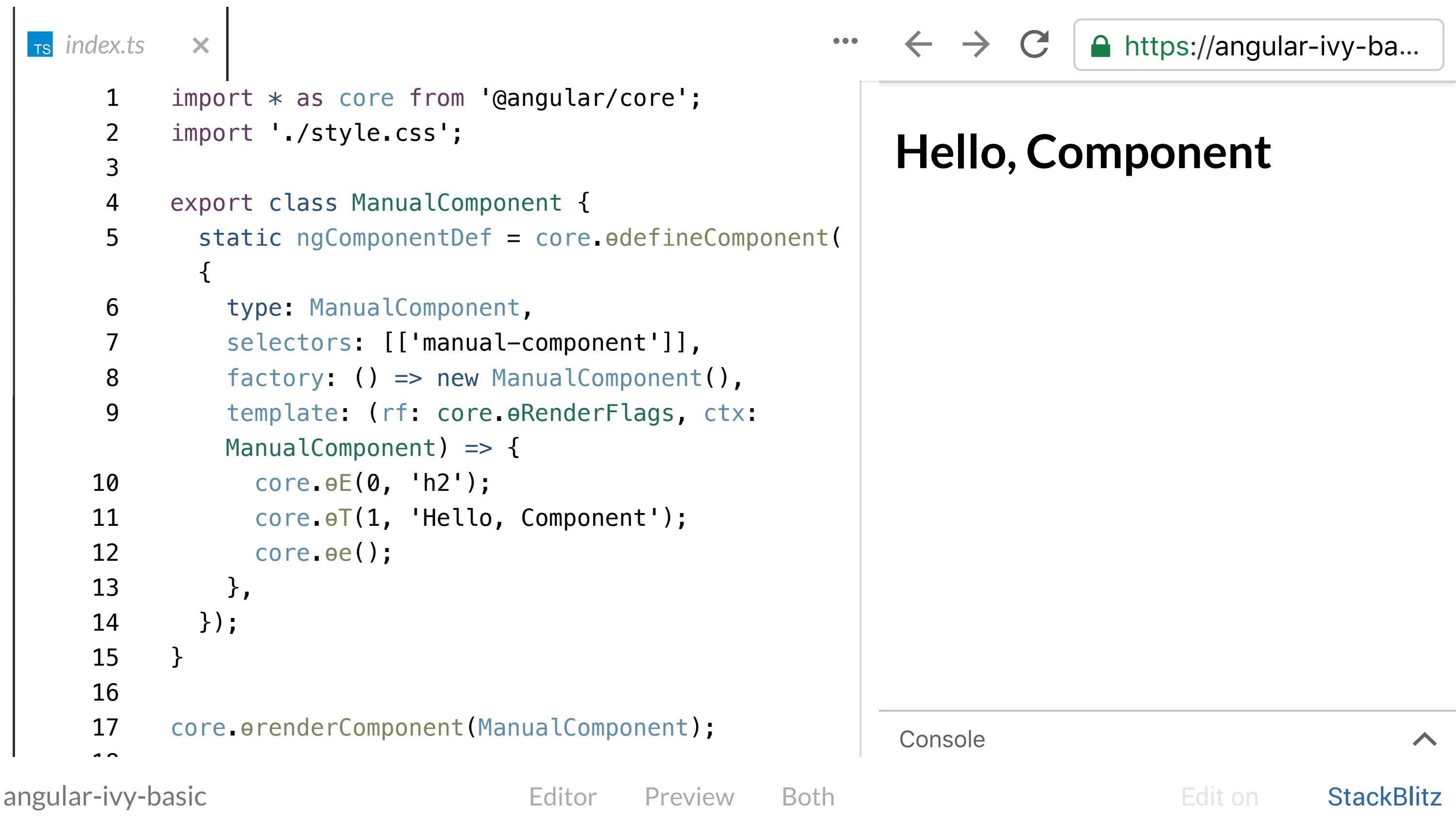
At this point, we are not using the `rf` or `ctx` parameters provided to our template function—we'll get back to them soon. But first, let's see how easy it is to render our first hand-crafted component to screen.

Our first manually-crafted app

To render this component to screen, Angular exports a method called `ɵrenderComponent` . All we have to do is to make sure that we have a HTML tag matching the element selector, `<manual-component>` , in our index.html file, and then add the following line to the end of our source file:

```
core.ɵrenderComponent(ManualComponent);
```

And that's it! We have a bare-minimum hand-crafted Angular application in just 16 lines of code. You can tinker with the complete example on StackBlitz:


index.ts

```
import * as core from '@angular/core';
import './style.css';

export class ManualComponent {
  static ngComponentDef = core.ɵdefineComponent(
  {
    type: ManualComponent,
    selectors: [['manual-component']],
    factory: () => new ManualComponent(),
    template: (rf: core.ɵRenderFlags, ctx:
    ManualComponent) => {
      core.ɵE(0, 'h2');
      core.ɵT(1, 'Hello, Component');
      core.ɵe();
    },
  });
}

core.ɵrenderComponent(ManualComponent);
```

https://angular-ivy-ba...

Hello, Component

Adding Change Detection

So we got a working example, but can we make it more interactive? How about adding some of Angular's magic sauce—Change Detection?

Let's change the above component so that it lets the user customize the greeting, so instead of just static "Hello, Component," the user will be able to customize the word the appears after "Hello."

We'll start by adding a new `name` property to the component class, and a method to update the value of `name` :

```
export class ManualComponent {
  name = 'Component';

  updateName(newName: string) {
    this.name = newName;
  }

```

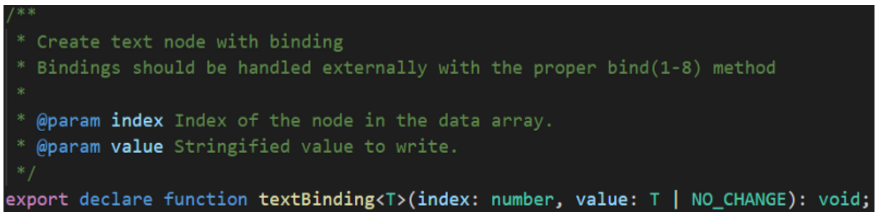
(Nothing too exciting so far... but wait!)

Next, we will update the template function to display the contents of `name` instead of just static text:

```
1   template: (rf: core.ɵRenderFlags, ctx: ManualComponent
2     if (rf & 1) {   // Create: This runs only on first r
3       core.ɵE(0, 'h2');
4       core.ɵT(1, 'Hello, ');
5       core.ɵT(2);   // <-- Placeholder for the name
6       core.ɵe();
7     }
8     if (rf & 2) {   // Update: This runs on every change
```

You probably noticed that we wrapped the template instructions with an if statement that checks the value of `rf`. This parameter is used by Angular to indicate whether we are creating the component for the first time (the least-significant bit will be set), or whether we just need to update the dynamic content during a Change Detection cycle (that's what the 2nd if-statement matches).

So when the component is initially rendered, we create all the different elements, and then whenever Change Detection kicks in, we only update the parts that might have changed. This is taken care of by the internal `ɵt` method (notice the lower-case t), which eventually maps to the `textBinding` function exported by Angular:



```
/**
 * Create text node with binding
 * Bindings should be handled externally with the proper bind(1-8) method
 *
 * @param index Index of the node in the data array.
 * @param value Stringified value to write.
 */
export declare function textBinding<T>(index: number, value: T | NO_CHANGE): void;
```

So the first parameter is the index of the element to update, the second is the value. In this case, we created an empty text element whose index was 2 on line 5, as a placeholder for the name, and then we update it on line 9 whenever change detection fires.

At this point, we should still be seeing "Hello, Component" on screen, although we can change the value of the `name` property and the name on screen will change.

To make this truly interactive, we will also add an input element, with an event listener that calls the `updateName()` method of our component:

```
1  template: (rf: core.ɵRenderFlags, ctx: ManualComponent
2    if (rf & 1) {
3      core.ɵE(0, 'h2');
4      core.ɵT(1, 'Hello, ');
5      core.ɵT(2);
6      core.ɵe();
7      core.ɵT(3, 'Your name: ');
8      core.ɵE(4, 'input');
9      core.ɵL('input', $event => ctx.updateName($event.t
```

Line 9 is where the event binding happens: the `ɵL` method sets up an event listener for the most recently defined element. The first argument is the name of the event (in this case, `input`, which is fired whenever the content of the `<input>` element changes), and the second argument is a callback. This callback gets the event data as an argument, and then we extract the current value of the event target—that is, our input element—and pass it to a function on our component.

The above is the equivalent of writing the following HTML code in your template:

```
Your name: <input (input)="updateName($event.target.value)"
/>
```

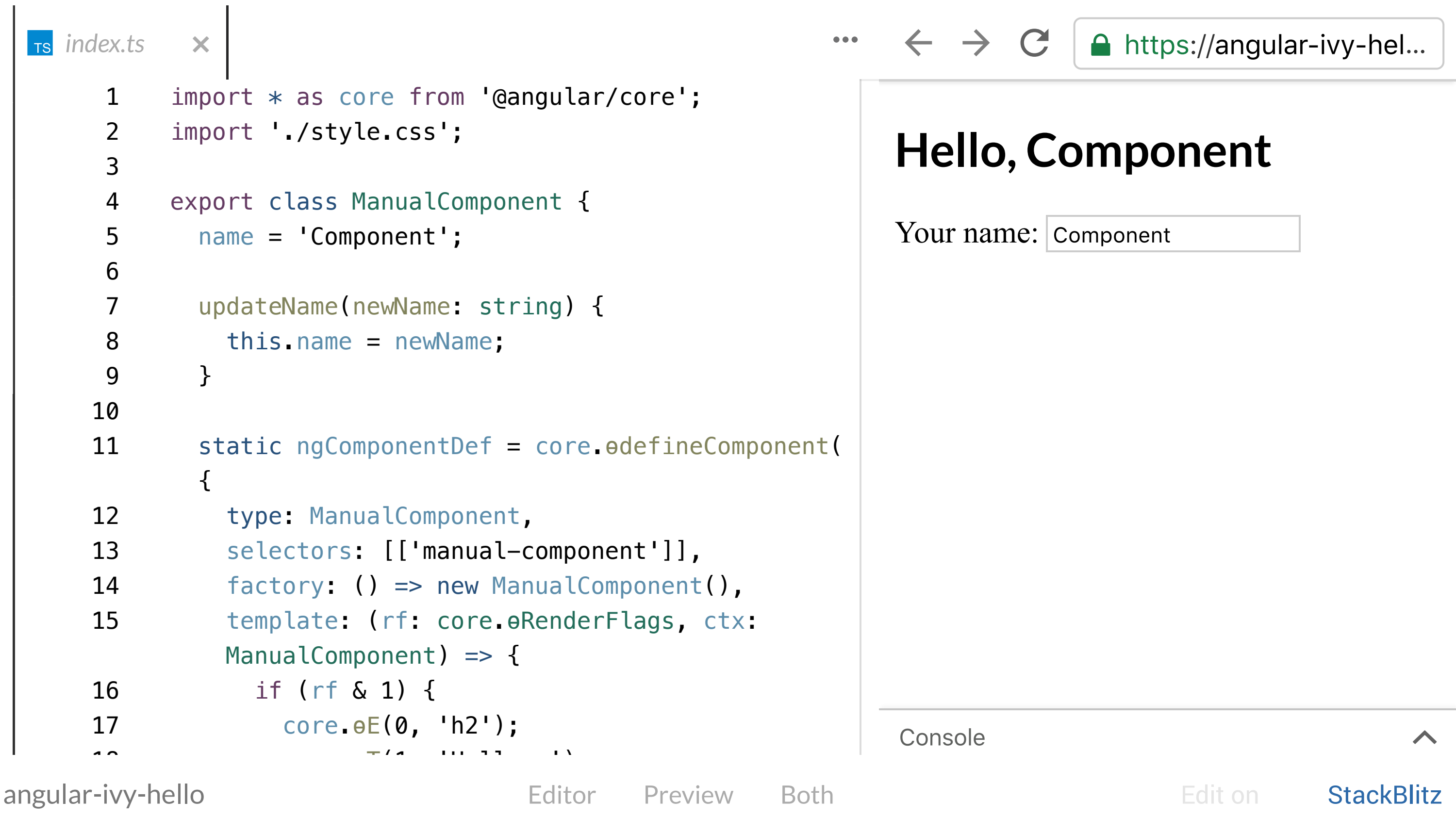
At this point, you can already edit the content of the input element, and you will see the greeting change accordingly. However, the input itself is not populated when the template is loaded—we need to add another instruction to the change detection phase of our template function:

```
template: (rf: core.ɵRenderFlags, ctx: ManualComponent)
  if (rf & 1) { ... }
  if (rf & 2) {
    core.ɵt(2, ctx.name);
    core.ɵp(4, 'value', ctx.name);
  }
```

We do so by introducing another internal rendered method, `ɵp`, which updates a property of an element with the given index. We give it index number 4, which is the index where we defined our `input` element, and instruct it to put the value of `ctx.name` inside the `value` property of this element.

Finally, our example is complete! We implemented two-way data binding, from scratch, using the Ivy rendered API. Pretty cool!

You can tinker with the final working code example below:



```
import * as core from '@angular/core';
import './style.css';

export class ManualComponent {
  name = 'Component';

  updateName(newName: string) {
    this.name = newName;
  }

  static ngComponentDef = core.ɵdefineComponent(
  {
    type: ManualComponent,
    selectors: [['manual-component']],
    factory: () => new ManualComponent(),
    template: (rf: core.ɵRenderFlags, ctx:
    ManualComponent) => {
      if (rf & 1) {
        core.ɵE(0, 'h2');
```

Change Detection, Ivy Style

At this point, we're familiar with most of the basic building blocks of the new Ivy compiler: we know how to create elements and text nodes, how to bind to properties and set up event listeners, and how to deal with change detection.

Not bad! But...

One More Thing

Before we part ways, let's go over one more interesting aspect: how does the compiler deal with sub-templates? These are the templates used for `*ngIf` or `*ngFor` blocks, and they have special handling.

We will see how to set up and use `*ngIf` in our hand-crafted template code.

First thing, we'd need to install the `@angular/common` npm package—this is where `*ngIf` is defined. Next, we can import the directive from this package:

```
import { NgIf } from '@angular/common';
```

Then, in order to be able to use `NgIf` in our template, we actually need to add some metadata to it, as the `@angular/common` module was not compiled with Ivy (at least not at the time of writing, it will probably change in the future with the introduction of ngcc).

We are going to use `ɵdefineDirective`, a sister-method of `ɵdefineComponent` we used above, which defines the metadata for directives:

```
(NgIf as any).ngDirectiveDef = core.ɵdefineDirective({
  type: NgIf,
  selectors: [['', 'ngIf', '']],
  factory: () => new NgIf(core.ɵinjectViewContainerRef(
  inputs: {ngIf: 'ngIf', ngIfThen: 'ngIfThen', ngIfElse
```

I actually found this definition in Angular's source code, along with a definition of `ngFor`. Now that we have `NgIf` set up properly for use inside Ivy, we can add it to the list of directives for our Component:

```
static ngComponentDef = core.ɵdefineComponent({
  directives: [NgIf],
  // ...
});
```

Next, we define the sub-template just for the section guarded by `*ngIf`.

Let's say we want to display an image there. We will define a new function for this template inside our template function:

```
function ifTemplate(rf: core.ɵRenderFlags, ctx: ManualC
  if (rf & 1) {
    core.ɵE(0, 'div');
    core.ɵE(1, 'img', ['src', 'https://pbs.twimg.com/tw
    core.ɵe();
  }
```

This template function is no different than what we did before—using the same building blocks for creating an `img` element inside a `div` element.

Finally, we can connect the pieces together by adding the actual `ngIf` directive to our component's template:

```
1  template: (rf: core.ɵRenderFlags, ctx: ManualComponent
2    if (rf & 1) {
3      // ...
4      core.ɵC(5, ifTemplate, null, ['ngIf']);
5    }
6    if (rf & 2) {
7      // ...
8      core.ɵp(5, 'ngIf', (ctx.name === 'Igor'));
9    }
10
```

Line 4 calls a new method, `ɵC`, which declares a new container element—an element that has a template. The first argument is just the index of the element, as we had with all the functions before. The second argument is the sub-template function we just defined, which will be used as the template for the container element. The third one is the tag name for the element, which is not relevant here, and finally, the list of directives and attributes associated with this element—this is where `ngIf` comes into play.

Line 8 updates the condition for this element by binding the `ngIf` attribute to the value of the boolean expression `ctx.name === 'Igor'`, essentially checking if the `name` property of our component equals "Igor."

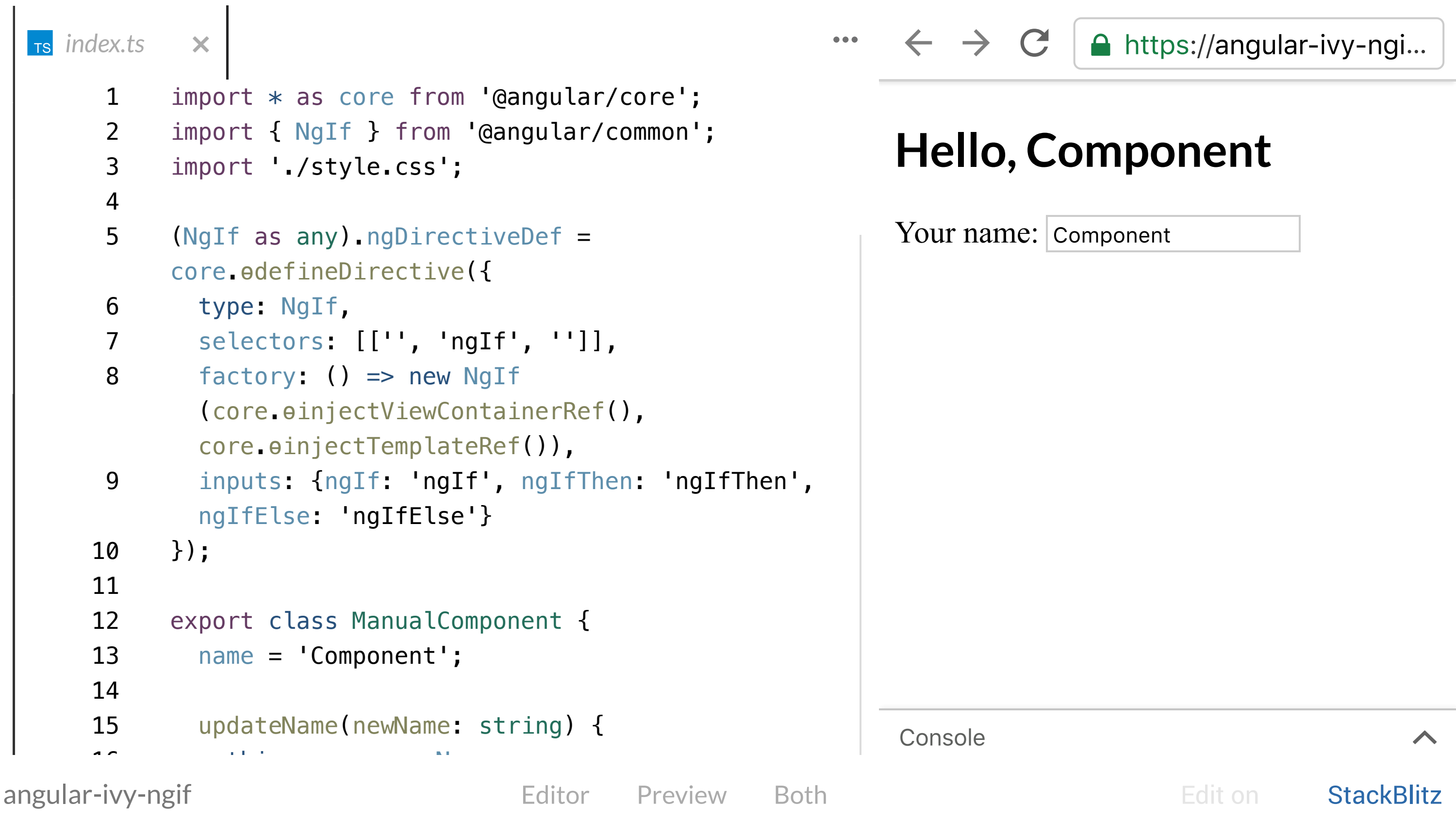
The above code is equivalent to the following HTML code:

```
<div *ngIf="name === 'Igor'">
  <img src="...">
```

```
</div>
```

Yes, this might be a bit more verbose, but still not too bad compared to the code produced by the current Angular compiler.

You can see the final code here (type *Igor* to see the NgIf section in action):



```
import * as core from '@angular/core';
import { NgIf } from '@angular/common';
import './style.css';

(NgIf as any).ngDirectiveDef =
core.ɵdefineDirective({
  type: NgIf,
  selectors: [['', 'ngIf', '']],
  factory: () => new NgIf
  (core.ɵinjectViewContainerRef(),
  core.ɵinjectTemplateRef()),
  inputs: {ngIf: 'ngIf', ngIfThen: 'ngIfThen',
  ngIfElse: 'ngIfElse'}
});

export class ManualComponent {
  name = 'Component';

  updateName(newName: string) {
```

Hello, Component

Your name: Component

It’s just the beginning…

We had quite a journey exploring how Ivy works together, and I hope it has whetted your appetite for learning even more!

By now you should have a setup where you can easily experiment with Ivy and see how it translates your template into JavaScript code, as well as a playground in which you can prototype and call the renderer directly, bypassing the Ivy compiler altogether.

Here are a few more resources to help you learn more about Ivy:

1. Ivy engine in Angular: first in-depth look at compilation, runtime and change detection—A quick overview of Ivy, the compilation process and change detection.
2. Angular Ivy change detection execution: are you prepared?—In-depth explanation about Change Detection in Ivy.
3. Ivy Architecture Design Document—Tons of information about the compilation model, design considerations, backward compatibility and IDE integration.
4. Render3 Source Code—Check out the actual implementation of the methods we discussed here, such as `elementStart()`, `textBinding()`, etc.

A big shout out!

Why is the new compiler called “Ivy”? Because it sounds like “IV”, the forth roman numeral, which stands for the 4th generation of the Angular compiler. It’s amazing to see how the Angular teams keeps experimenting and innovating, rewriting major parts of the framework to be better and always strives to improve and give us, front-end developers, a superior platform to build on. So a big thanks to the Angular teams—keep up the good work!

And if you’re a fan of Angular In-Depth, don’t forget to support us on Twitter!