A Modular View Engine Architecture For Angular 6
A Proposal to allow alternative view engines

Eamon O'Tuathail
mailto:eamon.otuathail@clipcode.com
http://www.clipcode.net

Licensing: Creative Commons CC BY 4.0
Last Updated: February 14, 2018
Tracked by Github Issue #22183

Overview

This proposal describes an architecture that makes the view engine used by Angular 6 modular and replaceable. This will allow different approaches to view engine implementation be supported now and in the future, by the core Angular team and by third parties. This should be achieved without requiring changes to other parts of the Angular Framework or in Angular application code.

Angular is just one of many projects engaged in the extremely competitive environment of web UI frameworks. To continue to succeed into the future Angular must be able to respond quickly to emerging threats and efficiently work on alternative substrates as they grow in popularity. There is a need for a view engine architecture in Angular that provides enough flexibility for such evolution.

Both Render2 and Render3 already support the idea of custom renderers, whereby the view engine calls into the configured renderer to complete certain tasks (e.g. create an element or a text node). We see with the internals of platform-webworker and platform-server what can be achieved with a custom renderer. Creating an alternative view engine is a major piece of engineering work, so where possible, a custom renderer should be implemented rather than a custom view engine. Assuming creating a custom renderer has been carefully examined but there is a goal to take a substantially different approach to how views are processed, only then should developing an alternative view engine be considered.

Current Situation

Two view engines exist for Angular 6 - the stable TypeScript-based single-threaded Render2 and the new faster/smaller TypeScript-based single-threaded Render3 (still in development). For a detailed description of how Render3 in Angular 6 currently works, see the appendix at the end of this source tour:
  http://www.clipcode.net/training/clipcode-source-tour.pdf
It is likely by the time Angular 9 or Angular 10 arrives, there will be a Render4. The view engine is a major sub-system within Angular, and as Angular evolves, so too will the view engine implementation(s). Because Angular application developers do not directly interact with the view engine (many don't even know it exists), it is quite practical to allow multiple substantially differing approaches to view engine implementation to be provided (using custom view compilers). However, if you wish to use the Compiler-CLI in the current Angular 6 beta, you have to use either Render2 or Render3 (based on the `enableIvy` flag) and their use is essentially hard-coded.

Motivation

A view engine implementation needs to be all three of fast, secure and robust. The reason alternative view engines interest us is that compared to Render3 they can be better at one or more of these characteristics, or that they offer additional features.

Each view engine implementation requires a different kind of code generation so for each custom view engine there needs to be a custom view compiler. In addition to the view engines that already exist, there are a number of other sensible ways an Angular view engine could be implemented - such as:

- taking a multi-threaded approach to rendering
- generating WebAssembly code
- generating SPIR-V code to exploit the work of the important W3C gpuweb working group
- generating Verilog for a FPGA (which are already available on servers in the cloud and in future will likely become a feature on some client computing devices)
- Deeper integration with native platforms (Android, iOS, Windows 10/UWP)

Approaches to supporting Modular View Engines

There are three possible approaches built on Angular 6 to facilitate modular view engines. They can be summed up as *do nothing*, *do a little* or *do a lot*.

Do nothing literally means that – no changes are need to be carried out to the existing Angular 6 beta code and still it is possible to support a custom view engine. Since it is the view compiler that writes the generated view processing code, a custom compiler command-line tool needs to be written and this, rather than ngc, needs to be called during application build. It would operate similar to Compiler-CLI, but its output would be in whatever language that was needed for the custom view engine.

Do a little means making small changes to Compiler-CLI so that the shared functionality between the different view compilers can be defined once, inside Compiler-CLI, and view engine specific code generation can be offloaded to the configured integrated or external view compiler. This can be achieved by changing Compiler-CLI's current `enableIvy` boolean flag to become a string parameter, which either says "render2" or "render3" (or in future "render4") if one of the integrated view engines is to be used, or it could state the name of an external dynamically loaded module and a class within that module (e.g. "my-module#my-view-compiler") which indicates the custom view compiler to be used. The actual code changes needed for this are limited and are listed in a single page at the end of this document.

Do a lot means realizing that supporting multiple view engines is going to be really important in the future and investing now the necessary engineering effort to structure ongoing development. It is common practice among compilation toolkits that target multiple backends (think LLVM) to have an intermediate representation (IR) – a target-independent description of the code; and to have multiple backend compilers, one per target, that transforms from IR to target. Multiple kinds of transforms can be written – for the IR and for each target. We need a similar architecture for Angular.

ViewIR

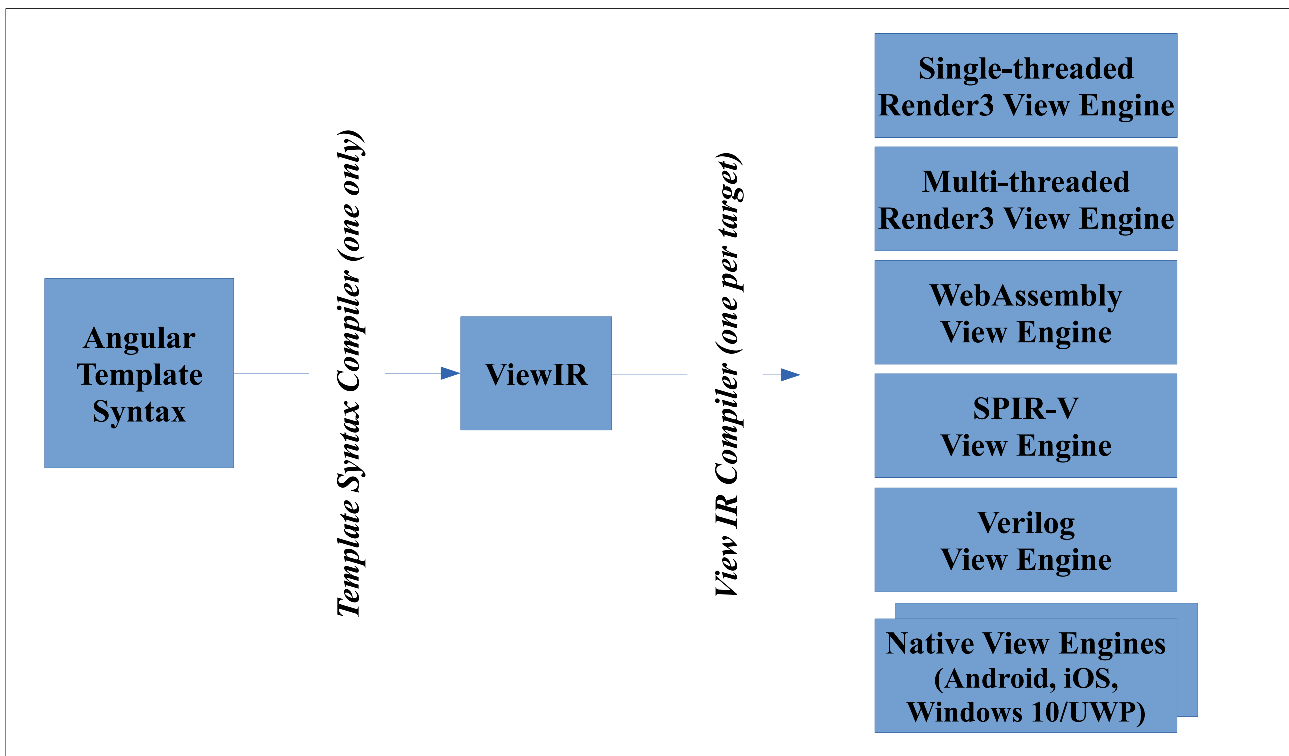
Let's call the Angular 6's intermediate representation ViewIR. The folks who are writing Render3 have essentially come up with the instruction set we need for ViewIR: https://github.com/angular/angular/blob/master/packages/core/src/render3/index.ts

These are the instructions required in order to deliver a view engine compatible with Angular Template Syntax. Perhaps some small changes/additions will be required, but in general this is a very suitable instruction set for ViewIR. The representation for ViewIR in memory should be an array of instructions, with each instruction beginning with an opcode and additional fields for the instruction's parameters. This would be easier to work with from external components (e.g. loop over the array) rather than trying to understand generated TypeScript code.

The current Render3 view compiler transforms straight from Angular Template Syntax to TypeScript source code, with calls into the Render3 view engine: https://github.com/angular/angular/blob/master/packages/core/src/render3/instructions.ts

We propose that Render3 view compiler code be modified, so that the frontend become a new template syntax compiler, that generates a ViewIR representation of the application's template syntax and the backend takes ViewIR and generates the TypeScript source code as before (so this becomes the ViewIR compiler for Render3). For other targets, similar ViewIR compilers will be needed.

In addition, the idea of transforms (for optimizers, profilers, lint, etc.) should be supported.



Multithreaded TypeScript Render3

Can we create a view engine that uses more than 3% of a computer's CPU capability?

A modern high-end PC has one (or more) powerful CPU(s) that each run 32 (or more) threads concurrently. Angular 6's Render3 is currently single threaded. When running at full throttle, a large complex Angular 6 app using Render3 will consume just 3% (or less) of the available compute power. Over time support for large number of concurrent threads will become a feature on mainstream desktops, then premium mobile and later general mobile devices. Hence though high thread-count devices are a niche now, they will become much more widely used in future.

The "obvious solution" is to created a multi-threaded TypeScript Render3 view engine. Whether this "obvious solution" will actually be faster than the single-threaded Render3 view engine is an interesting question, that really can only be answered by creating a multi-threaded prototype and then benchmarking against the existing single-threaded approach.

To be clear – what we are discussing here is the use of web workers internally within the view engine architecture. The application developer will not even know of the existence of these web workers. The application still uses platform-browser and application code still runs in a single thread (the main UI thread). What we are discussing here has nothing to do with the use of web workers in platform-webworker, which uses web workers from application code. This is also a good point to bring up the fact that the Angular Template Syntax will include chunks of TypeScript application code and this has to run where the rest of the application code runs. Some parts of view manipulation must be sequential (the parent node in a DOM tree must be created before the child) but it should be possible to work on different node sub-trees concurrently.

We can see three scenarios with differing outcomes to creating a multithreaded TypeScript Render3 view engine. When using web workers and no shared memory buffers (supported everywhere), the view engine work load can be split among several web workers, but writing to the DOM needs to be done from the main UI thread. So a description of what node needs to be created (etc.) has to be encoded in the worker thread, passed over the message bus, and then worked upon in the main UI thread. That message passing would seem time consuming, and the unresolved question is whether enough time is saved by the use of web workers to outweigh this?

Some modern browsers support shared array buffers. Though there currently are security concerns (Spectre) and not all browser vendors support this feature, over time these problems will be solved and this feature will become commonplace. With this approach we can have additional web workers that populate a shared array buffer with a representation of what needs to be created in the DOM, and then this buffer can be used by the main UI thread to populate the DOM.

What would really speed up a multithreaded view engine is that at some point in the future the Web API standards were enhanced to allow web workers write directly to the DOM. That would very significantly improve performance.

WebAssembly

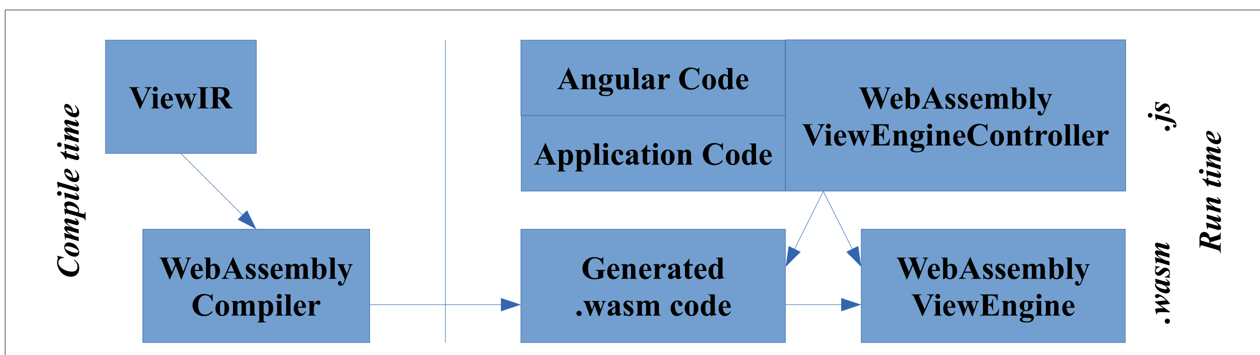
WebAssembly is a binary intermediate representation of code which is passed to an embedder (e.g. the VM inside a web browser; also works inside Node.js) for local compilation and execution. WebAssembly modules are small and binary – they are intended to be extremely fast - fast to download, fast to compile into locally optimized code and fast to run. WebAssembly is supported by all the main browsers (Google, Mozilla, Apple, Microsoft)-http://webassembly.org / “launching the WebAssembly WG”

This is a good paper explaining WebAssembly:
https://github.com/WebAssembly/spec/blob/master/papers/pldi2017.pdf
This developer guide explains how to program in assembly in WebAssembly:
http://www.clipcode.net/training/clipcode-webassembly-devguide.pdf
This set of concept diagrams show how WebAssembly ideas are related:
http://www.clipcode.net/training/clipcode-webassembly-concept-library.pdf

WebAssembly modules and JavaScript (TypeScript) code can interact via tables of function calls or linear memory. It is best to think of a WebAssembly module as a shared library (*.so or *.dll). Currently WebAssembly modules cannot directly call the DOM (even when executing in the browser UI thread). Direct DOM access is an important feature planned for inclusion in a future version. When it arrives, then a WebAssembly-based view engine for Angular would be quite an interesting option. Such an implementation would require three major pieces:

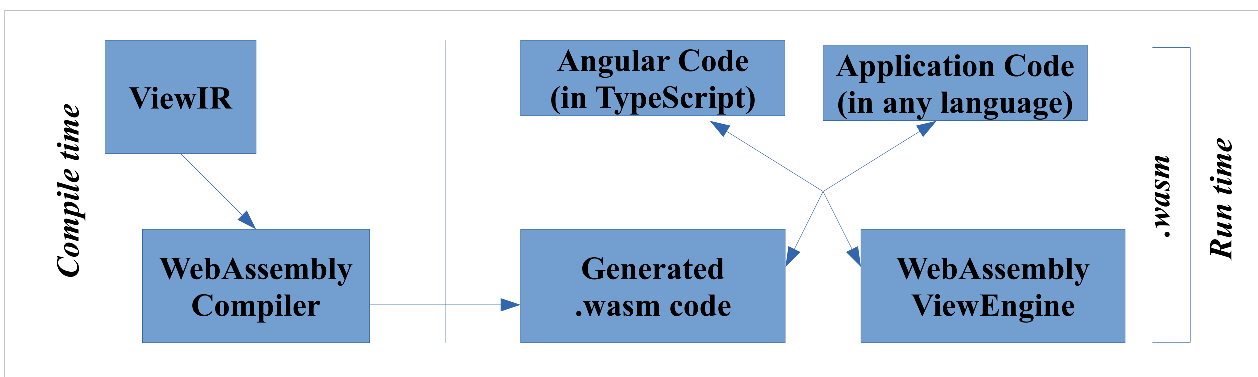
- WebAssemblyCompiler (written in TypeScript) - perform the equivalent of _emitRender3 from: https://github.com/angular/angular/blob/master/packages/compiler-cli/src/transformers/program.ts
  to generate application code based on web assembly (so rather than "./dist/out-tsc", we have "./dist/out-wast").
- WebAssemblyViewEngine - Perform the equivalent of: https://github.com/angular/angular/blob/master/packages/core/src/render3/instructions.ts (written in in WebAssembly assembly)
- WebAssemblyViewEngineController (written in TypeScript) - act as a bridge between the WebAssemblyViewEngine and the rest of the Angular Framework.

The data array that instructions.ts above manages for the TypeScript-based Render3 would now need to be (WebAssembly) linear memory, so TypeScript-based application code and WASM-based WebAssemblyViewEngine code can both access it efficiently.



There will be an alternative architecture to be considered in future. When WebAssembly gains support for a garbage collector and for WebIDL access, then it will become a popular target for compiler writers. Imagine there are a host of mainstream languages (such as Java, C#, Python) and interesting specialist languages (such as OCaml or Coq proof assistant) that can efficiently compile directly to WebAssembly and talk to each other and the DOM via WebIDL-defined interfaces. How does a future Angular play in such a world? Before exploring that, we need to explore what happens to TypeScript/JavaScript in this new world. They too could get compilers that directly target WebAssembly. However, a huge amount of engineering effort (and millions of dollars) has already been spent optimizing the VM for JavaScript, so compiling to WebAssembly probably will not speed up JavaScript, especially initially. Over time as WebAssembly becomes a target output for pretty much every programming language, it too will get lots of attention from VM writers and so will be very fast. Hence over time we expect a WebAssembly target for JavaScript to become popular and very efficient. When that happens, it will greatly simplify the Angular/WebAssembly world.

The Angular Framework can continue to be written in TypeScript but application code can be written in any WebIDL-supporting programming language that targets WebAssembly. (we discuss later the Angular API and WebIDL). There is no need for a TypeScript-based WebAssembly-ViewEngineController mentioned earlier, since the WebAssemblyViewEngine can make direct calls into the Angular Framework code and application code.



gpuweb

The mission of the W3C GPU FOR THE WEB community group (https://www.w3.org/community/gpu) is "to provide an interface between the Web Platform and modern 3D graphics and computation capabilities present in native system platforms". Compared to the existing WebGL, gpuweb is intended to offer programmatic access to more modern GPU features and be much faster. All the main browser vendors (Google, Mozilla, Apple, Microsoft) are members of this group. The new spec is likely to be based on Vulkan's SPIR-V intermediate representation (https://www.khronos.org/spir). gpuweb standardization is at the initial stages and will likely take a good while before it becomes a stable feature of shipping web browsers. For Angular applications, usage on VR/AR headsets is in future likely to approach if not match usage on desktop and mobile devices. There will be multiple technical approaches to bringing web apps to the world of 3D - but being able to efficiently produce gpuweb-based output is likely to be one very important approach.

This requires creating a gpuweb-based view engine implementation for Angular so that the view engine itself and the Angular app's view processing code run directly on the GPU and not the CPU. Right now, without having to wait for the gpuweb community, SPIR-V is supported for desktop apps on Windows, Linux and Android:
https://developer.android.com/ndk/guides/graphics/index.html
So today's Angular apps intended for desktop and Android usage could benefit from a SPIR-V based custom view engine.

FPGA

A FPGA - Field Programmable Gated Array - is programmable hardware:
https://en.wikipedia.org/wiki/Field-programmable_gate_array
They are typically programmed using the SystemVerilog language:
https://en.wikipedia.org/wiki/SystemVerilog

Depending on the application type and what marketing you read (and are willing to believe), using an FPGA can be 20 to 100 times faster than using a CPU. Currently we see FPGA being used in data centers (Xilinx, Intel/Altera) and over time as costs drop we are likely to see them appearing firstly on high-end desktops/workstations, then regular desktops/laptops and then mobile devices. Sometime in future many client devices will have embedded FPGA components, and at that stage there is likely to be a WebFPGA group to allow secure access to this functionality from a web application.

Imagine a future WebFPGA committee has come up with a good standard and web browsers vendors have implemented it. At that stage, the Angular view engine and an Angular app's view processing code should run directly on the FPGA and not the CPU. Right now, without having to wait for a future WebFPGA committee, FPGAs are available in some data centers. So Angular apps using Angular Universal could benefit from a Verilog-based custom view engine.

Native Platforms (Android, iOS, Windows 10/UWP)

There is great opportunity to extend the reach of Angular into mobile and desktop app development. Web application developers have invested quite a bit of time learning Angular well and they would really like to leverage this skill on multiple substrates. There already exists third party ways of creating native apps with Angular, but it would be much better for native app support to be directly integrated with the main Angular source tree and developed by the core Angular team. Native platforms and the web platform should be considered equally important when evolving Angular in future. There are three approaches to running on native platforms:

- Use WebView
- Write a custom renderer (moderate amount of work)
- Write a custom view engine (lot of work, much most flexible approach)

App development is very very competitive with multiple frameworks and technical approaches vying for developer attention. For Angular to succeed here in a major way we think we will ultimately find a custom view engine is the way to go. Though this will involve the most work, it will result in the most compelling architecture.

Making the Angular Framework API WebIDL-based

> *Note: the following two sections discuss changing the Angular Framework API – which is a major undertaking and probably will not happen for a few versions. Both ideas are important and should at some point be adopted for existing APIs inside existing packages. These ideas could (and should) be adopted immediately for new packages and new APIs in existing packages.*

Currently, the Angular Framework (https://angular.io/api) exposes a TypeScript-based API. This is fine when it is to be used from application code and view engines also written in TypeScript, but presents difficulties when these are written in other languages. If in future Angular is to become a highly competitive offering on diverse substrates (think WebAssembly or a native mobile app) which will have a range of technologies and programming languages, then the Angular Framework API needs to easily interoperate with a more diverse world than it has been used to up to now.

The interrelationship between WebAssembly, WebIDL, the DOM, an integrated garbage collector (e.g. for Java / C#) is still being defined, but what is clear is that all WebAssembly-targetting languages need to be able to efficiently call WebIDL-based APIs. Hence that would be a good target for interoperability for Angular in general. Therefore we propose that the Angular Framework API itself become a WebIDL-based API, so that it is easily callable from a whole host of languages. This would also allow future Angular Framework packages themselves to be written in other languages (think Python-friendly TensorFlow).

Using namespaces to sub-divide APIs of larger packages

Making changes to the Angular Framework API would also allow work to be done on the general layout of the API to make it more discoverable. A desirable characteristic of any API design is discoverability – a measure of how an application developer new to an API can simply discover what is available in order to complete some task. In particular, the flat exported APIs of Core and Common packages are getting too large. They are daunting for new Angular developers. These pages explain modules and namespaces in TypeScript:

- http://www.typescriptlang.org/docs/handbook/namespaces.html
- http://www.typescriptlang.org/docs/handbook/namespaces-and-modules.html

As a simple test, ask a developer new to Angular to look at the Core section of https://angular.io/api and pick out types relevant to, let's pick an example, say dependency injection. They may well pick out `Inject`, `Injector` and `InjectionToken`, but they will miss many other DI types. If a future Angular version used multiple namespaces to sub-divide Core APIs, it would have a Dependency Injection namespace, and the documentation page would list its contents as a unified list. At that stage, if we again asked our new Angular developer to select the DI types from the documentation page – this time there will much improved discoverability and all the DI exported types will be correctly identified – and there are many:

- https://github.com/angular/angular/blob/master/packages/core/src/di.ts

Hence moving to a namespace-based layout for larger packages would be desirable.

Change needed to add configurable view compilers to Angular 6

Here are limited changes needed to support custom view compilers in Compiler-CLI.

1) A configurable view compiler is defined as:

```
abtract class ConfigurableViewCompiler {
  emit(
      {emitFlags = EmitFlags.Default, cancellationToken, customTransformers,
       emitCallback = defaultEmitCallback}: {
        emitFlags?: EmitFlags,
        cancellationToken?: ts.CancellationToken,
        customTransformers?: CustomTransformers,
        emitCallback?: TsEmitCallback
      } = {}): ts.EmitResult;
}
// note the emit signature is the same as _emitRender3 and _emitRender2.
```

2) A factory function creates an instance of it:

```
export function viewCompilerFactory() : ConfigurableViewCompiler;
```

3) In https://github.com/angular/angular/blob/master/packages/compiler-cli/src/transformers/api.ts the boolean enableIvy option is renamed to viewCompiler (to identify which view compiler to use) and becomes a string;

```
/*
 * Selects the view compiler to use.
 * This can either be a well-known name of a view compiler that the
 * Angular Framework itself implements, such as "render2" or "render3", or
 * can be the module name and factory function name (separated by the '#'
 *  symbol) to an external view compiler. (e.g. module-name#function-name).
 * (using the same naming layout that lazily-loaded routes use)
 * If not specified, currently defaults to "render2".
 */
export interface CompilerOptions extends ts.CompilerOptions {
..
  viewCompiler?: string;
}
```

4) In https://github.com/angular/angular/blob/master/packages/compiler-cli/src/transformers/program.ts, in the emit function, this line:

```
   return this.options.enableIvy === true ? this._emitRender3(parameters) :
                                            this._emitRender2(parameters);
```

needs to be changed to:

```
if (this.options.viewCompiler == "render2"||this.options.viewCompiler==null)
  return this._emitRender2(parameters)
else if (this.options.viewCompiler == "render3")
  return  this._emitRender3(parameters)
else
  return loadViewCompilerFactory(this.options.viewCompiler).emit(parameters);
```

5) A new function needs to be written to load the view compiler and return the result of calling the view compiler factory function:

```
   loadViewCompilerFactory(compiler:string)
                          : ConfigurableViewCompiler { ... }
```