The Angular 7 API should be IDL-based

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

Tracked by: Github Issue #22398
Last Updated: February 23, 2018
Related Paper: Modular View Engine Architecture For Angular

Overview

Note: This is the second of a pair of papers that explore how Angular should evolve. Both proposals include discussion of the use of WebAssembly (and other substrates). The first paper looks at using WebAssembly internally within the Angular Framework, to build a WebAssembly view compiler and view engine. That use of WebAssembly would not be visible to application developers since they do not interact with the view engine.

What we propose in this paper concerns the Angular API that application developers use everyday, and so might directly affect them, in a good way (many more programming languages to choose from). It will have no effect on the continued use of the existing TypeScript-based Angular stable API.

Both proposals are independent and can be considered separately.

Rather than just being the optimum framework for TypeScript, the Angular Project should be much more ambitious and also become the optimum framework for all Web Assembly-based languages (which over time will include all the popular programming languages and many specialist ones too). All these soon-to-be-web-enabled languages will need a powerful web framework for the exact same reason TypeScript needed one that it found in Angular. In addition, the investment in making Angular suitable for use from multiple languages on the web will also pay dividends in making Angular the premier framework for cross-platform native app development.

In this paper we suggest re-specifying the Angular API using some form of IDL (Interface Definition Language), so that it can be used from any language that supports that IDL. Then we explore and contrast different kinds of IDL that might be suitable – firstly Web IDL that the Web Platform already uses to define almost all its interfaces, secondly a new Wasm IDL that utilizes the full power of WebAssembly, thirdly a limited form of TypeScript declaration files, which we will call TypeScript IDL and fourthly a new Semantic IDL that better describes the rich capabilities of modern languages. Each language compiler will need a simple CLI tool to auto-generate a native client library to call the IDL-specified API. For some popular languages, an enhanced client library can be hand-coded on top of this low-level generated code. In TypeScript's case, this enhanced client library should expose an API that exactly match the Angular 6 TypeScript API. Thus existing Angular 6 apps can seamlessly move to Angular 7 and continue to use Angular's stable TypeScript-based API.

This paper examines how to achieve this & recommends a sensible path to a solution.

Current Situation

Currently, the Angular Framework (https://angular.io/api) exposes a TypeScript-based API. This is perfect when used from application code also written in TypeScript, but presents difficulties when applications are written in other languages. If in future Angular is to become a highly competitive offering on diverse substrates (think WebAssembly or native mobile apps, both scenarios which will have a range of runtime 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. With a little more effort, Angular could equally well target many languages.

One language or Many Languages?

According to https://octoverse.github.com, TypeScript is the tenth most popular language in use. TypeScript is an excellent language and a superb choice for what Angular uses it for. However there are plenty of other excellent languages and software engineering teams pick different languages for different projects based on very sensible reasons.

For example, if we were writing client code to interact with TensorFlow, we would use Python. If we were writing complex mathematical algorithms, C++ would be better in conjunction with a library such as Eigen. When trying to fully exploit native (mobile/desktop) substrates, there is often a native language ideal for that (e.g. Swift for iOS, Java/Kotlin for Android, C# for Windows 10/UWP). So there are many languages now and this will continue to be so well into the future.

WebAssembly – An evolving set of W3C specs

The W3C WebAssembly Working Group (https://www.w3.org/wasm) has just published (15th February, 2018) the first working drafts of a set of WebAssembly specifications – these over time will progress to full W3C Recommendations:

- https://www.w3.org/blog/news/archives/6838

All four of the top browser providers – Google, Mozilla, Microsoft and Apple – support WebAssembly in their shipping browsers today. They are cooperating to evolve the WebAssembly set of specifications and there are a number of important enhancements under consideration:

- https://github.com/webassembly

Of particular interest is host bindings:

- https://github.com/WebAssembly/host-bindings/blob/master/proposals/host-bindings/Overview.md

The relationship between WebAssembly, Web IDL, the DOM, an integrated garbage collector (e.g. for Java / C#) and the Anyref proposal (to allow direct handling of host objects) is still being defined at a standards level. Afterwards implementations in web browsers are required. The Web Platform is defined by a set of Web APIs all of which use Web IDL for definitions. Typically the web browsers are written in C++ and application code is written in JavaScript, so Web IDL provides the necessary glue. It is clear that WebAssembly needs to gain the ability to efficiently call Web IDL-based APIs so that languages that compile to WebAssembly can compete with JavaScript.

Popularity

TypeScript is a fine language, but it is just one of many. Re-imagining the Angular API using an IDL will involve a non-trivial amount of work, so we need to convince ourselves that there is a much, much larger developer community out there using many programming languages that is worth paying attention to. We need to prove that there are vastly more developers out there not using TypeScript compared to those who do. We believe in data, and to determine what is actually used by enterprises worldwide, we believe the very best approach is to measure open job vacancies. Other people think we should measure interest in a technology by counting the lines of code in open source repositories, or number of mentions in blog postings. However, if companies are spending substantial amounts of money to hire developers for a particular technology, then this conclusively proves they are deeply interested in that technology. In such a scenario, the technology is not something to be "evaluated in future" – no, by the time developers are being hired, then a competent argument has to be made to the company accountants that this technology is important to the company and real money needs to be spent. So it makes sense to focus on job vacancies as an unbiased measure of what is actually important in the real world. The data never lies.

Here we are particularly interested in programming language popularity, but to first validate our chosen selection criteria (job vacancies), let's start by looking at vue.js vs. React vs. Angular, three competing web frameworks. Let's look at job sites in the USA, Ireland and Singapore to get a global perspective. Ireland is Europe's largest exporter of software (and second largest in the world, behind the USA). Singapore is where many international technology firms select for their APAC regional HQ.

Framework	dice.com (USA)	ie.indeed.com	www.monster.com.sg
Angular	3,626	487	61
React	2,200	494	25
Vue	125	9	0

We see that Angular is significantly more popular that React in the USA and Singapore, but in Ireland they are about equal. Vue is nowhere to be seen. If you are a Vue developer, don't bother looking for a job in Singapore. There aren't too many in the USA either, about two and a half jobs per state. Perhaps you should learn Angular.

Now let's repeat this exercise for a selection of programming languages:

Language	dice.com (USA)	ie.indeed.com	www.monster.com.sg
TypeScript	487	29	6
C#	5,767	263	51
Swift	647	43	29
Java	13,772	337	693
Scala	1,185	38	14
Kotlin	98	5	6

We see that among enterprises recruiting in the USA, C# is about 12 times more popular that TypeScript, and Java is about 28 times more popular. In other countries there are significant differences too. (Aside: although not relevant to our current discussion, we were also very surprised to see how poorly Kotlin fares – Scala is 11 times more popular and Java is 130 times more popular than Kotlin, a big difference).

Hopefully by now we all agree that the number of non-TypeScript developers out there is massive compared to the number of TypeScript developers. When WebAssembly matures somewhat more and gets proper support for Web IDL, we will see every compiler team on the planet at least consider WebAssembly as an important target. Many will dedicate resources to building highly competitive compiler toolchain offerings. Shortly after the language compilers arrive for WebAssembly, then many application developers who currently use those languages on other platforms will start building web apps – and they all need a powerful web framework. An IDL-enabled Angular 7 would be ideal.

Sample Usage of Other Languages with Angular 7+

There are a number of envisaged usage scenarios with Angular 7+ for languages other than TypeScript:

- Write the application completely or partially in one these other languages
- Write one or more event handlers in another language
- Implement (domain) services in other languages

It is just not practical to re-write in TypeScript some of the massive software libraries enterprises around the world are using – it is much more sensible to facilitate running those libraries with little or no modifications in WebAssembly. If you and your application development team have 1,000,000 lines of server code in production in a particular language (that happens not to be TypeScript) and the boss says "we need this on the web, fast", then writing a small amount of extra code in the same language to call an IDL-enabled Angular 7 seems the optimum path to choose. Client machines are getting much more powerful - with multiple cores and faster cores – so pushing extra workloads towards them makes sense (and is free compute capability).

We could also see developers:

- Using C# to create a custom pipe
- Using C++ to create a custom directive
- Storing a Swift object in Angular's dependency injection system
- Using Java code to add routes to the Angular Router service
- Writing an ultra high-performance custom Render3 view engine in hand-coded WebAssembly assembly (*.wast)

We expect TypeScript to continue to be the most popular language used to create Angular apps, but now client application developers have additional options.

TypeScript will also continue to be the main language for Angular implementation, but in this new world it is also quite possible that some new Angular packages are written in a different language. For example, an imaginary future integration between a WebAssembly-enabled build of Python-friendly TensorFlow and Angular could be based on an Angular package written in Python.

Suggested Plan

Delivering an IDL based Angular API will take time and become effective over multiple releases of future Angular updates. Some of the work to make Angular an IDL-specified API can progress independently of the WebAssembly/Web IDL work – what is important is that Angular be the first major web framework that is ready for production use when WebAssembly is updated to supported IDL.

There are a number of kinds of IDL that might be suitable (we will explore them shortly). Through further investigation, prototyping and benchmarking, a decision is needed which one to choose.

After that, here is how we would recommend progress:

- With immediate effect (starting with the brand new Angular Elements package), all new Angular packages only have IDL-based APIs (and have no TypeScript-only APIs)
- As soon as possible, experimental APIs be upgraded to be IDL-based (their TypeScript-only APIs are to be deleted)
- Over time, the stable APIs in existing Angular packages are to be replaced with new IDL-based APIs
- For TypeScript, a new enhanced client library is to be created that calls this IDL-based API and that exactly matches the Angular 6 API (thus Angular 6 apps that use the stable API can transition to future Angular without change)

Using namespaces to sub-divide APIs of larger packages

Altering 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 random 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.

IDL – Interface Definition Language

An IDL is used to precisely describe the boundary between execution environments. This might be a networking boundary between remote network connections (think RPC). This might be separate incompatible languages (imagine client code written in JavaScript and an implementation written in C++). This might be different language engines (think WebAssembly and its embedding host). When we examine multiple programming or data representation languages we see lots in common – classes, interfaces, methods, properties, primitive data types (ints, strings) but we also see each language is different in various ways – a blob of TypeScript code cannot simply be passed to a C++ compiler. The essence of an IDL is to grasp the commonality and allow constructs that might naturally be incompatible to interoperate easily. An IDL is often simpler than any specific programming language, because it is the shared set of concepts that many languages support.

In this paper we are proposing that Angular looks outside the cocoon of the familiar JavaScript VM and Angular 7's API be based on an IDL – so that it becomes usable from 20+ popular languages that are coming to the web soon. But which IDL?

We think there are four competitive possibilities:

- using **Web IDL** familiar to web developers,
- creating a new IDL specific to WebAssembly which we will call **Wasm IDL**,
- using a subset of TypeScript declaration file syntax which we will call **TypeScript IDL** (such files already cleanly define the exported APIs of a library and a CLI tool could auto-generate client code from them in multiple languages)
- creating a new IDL, which we will call **Semantic IDL**, that more richly describes modern programming languages

Though we will now look at each of these options from a WebAssembly perspective, we would emphasize that much of our discussion here is also relevant to future Angular-powered native mobile and desktop applications.

A significant part of the existing Angular API is marked as "stable" and support for these needs to be continued, so existing applications can continue to work in future without change and new TypeScript applications can also use these APIs.

We are assuming there will be 20+ languages that all need to be able to access Angular 7 functionality. Hand-coding a wrapper library for each is not practical or sustainable in the long term. It is much better to select some kind of IDL and auto-generate client libraries from a single Angular 7 API specification correctly and quickly each time.

Web IDL

Web IDL is a W3C Recommendation that it used very extensively throughout web standards to specify the API that the Web Platform exposes to applications. In practice, the Web Platform is implemented in C++, and applications are written in JavaScript or a language that transpiles to it, such as TypeScript. However, Web IDL is specified in a language-independent manner (though it does comes with an ECMAScript binding). There is nothing stopping anyone writing bindings for other

languages. Note Web IDL is spelled with a space between "Web" and "IDL" (important if you are googling!). The latest draft of the Web IDL specification is here:

- https://heycam.github.io/webidl

Though not a feature of the current WebAssembly specifications that all modern web browsers already support, we expect over time that support for directly using Web IDL APIs (e.g. using the DOM or calling Web APIs such as `XMLHttpRequest` or `Worker`) will be added to WebAssembly. That will result in compilers being created for WebAssembly for many languages, including popular ones such as Java/C#/Swift.

What is clear is that all programming languages that target WebAssembly need to be able to efficiently call Web IDL-based APIs. Hence that would be a good target for interoperability for Angular in general. Were the Angular Framework API itself to become a Web IDL-based API, then it will be easily callable from a whole host of languages. In this way all these languages will be treated as first class citizens and equal attention will be paid to them as valid web targets. Unlike with most Web IDL usage, with an Angular Web IDL-based API, the implementation will be written in JavaScript (TypeScript) and the client in a whole host of other languages (including C++). In theory, this should not be a problem, but a prototype might be useful to verify.

One clear advantage of selecting Web IDL is that regardless of what the Angular team does, the 20+ compiler teams that will target WebAssembly all need to provide Web IDL support anyway, as that is how they will be using the DOM. Specifying an API using Web IDL is only part of the effort involved in making Angular 7 a Web IDL-based API. The API itself will need to be simplified, as all the language constructs TypeScript offers that are currently used by the Angular 6 API are not available in Web IDL, so alternative approaches will be needed. Then technical issues will need to be considered – such as how are object lifetimes managed across language boundaries (will some sort of internal object manager be required).

Wasm IDL

Question – would a new IDL specific to WebAssembly – let's call it Wasm IDL - be a better compared to Web IDL? The reason we ask is that WebAssembly has certain coding patterns and Wasm IDL could match these as closely as possible. For example, Wasm IDL data types could precisely match WebAssembly data types. Also Wasm IDL could understand WebAssembly constructs such as linear memory or tables (for indirect functions calls) – neither of which are understood by Web IDL.

Creating Wasm IDL would involve more work than using Web IDL, but it is not specific to Angular. Perhaps it may lead to the best way of allowing language objects from different languages to work together in WebAssembly.

Or perhaps not. One argument against Wasm IDL is that it would be just too low-level. WebAssembly is not an Intermediate Representation (IR) in the sense of LLVM IR or .NET IL or Java bytecode. Instead WebAssembly is a virtual Instruction Set Architecture (ISA), more akin to x86-64 or ARM assembly programming. By the time application code has been transformed to WebAssembly, it is very low level and concepts like classes, interfaces, mixins and inheritance have long disappeared.

TypeScript IDL

The best software engineer is the one who does the least work (is that true?).

What is the quickest way involving the least effort to get Angular 7 up and running as an IDL-based API? How about simply saying it already is. That sure was quick and sure was cheap! Let us explain. Angular defines its API using TypeScript exports. We could just say "that is the IDL-defined interface" and hey presto, we are finished.

Any language that compiles to WebAssembly would need to be able to call this TypeScript declared exported API. The TypeScript compiler can easily read these export declarations and with a little coding to create a tiny CLI tool, one could auto-generate WebAssembly client code in 20+ languages that calls these exports.

This is an interesting approach and though simple, it probably is not that simple. We would expect some TypeScript constructs uses in Angular exported APIs to be too difficult to wrap as Web Assembly client code, so a simplified set of TypeScript constructs might be needed. We would call this simplified subset of TypeScript syntax TypeScript IDL.

Semantic IDL

Question – would a totally new IDL be the best long term approach of all?

When we examine Web IDL we see it does not have all the feature set we would expect to define an API of a modern programming framework. Web IDL was originally defined to describe the layer between JavaScript application client code and C++ implementation of the Web Platform.

When we move to an era when 20+ programming languages can be used with WebAssembly, then perhaps it is time to come up with a new IDL that optimally describes how they can all talk to each other. A new kind of IDL could better capture all the semantics of what a modern API is offering.