Angular 2 Imports and Bundle Structure

*Status: (Draft)*

*Authors: misko@google.com*

*This document is published to the web as part of the public* [*Angular Design Docs*](https://drive.google.com/#folders/0BxgtL8yFJbacUnUxc3l5aTZrbVk) *folder*

# Objective

Propose a strategy for importing and bundling Angular which will support different platforms.

# Background

Our current strategy is to create angular2/angular2 bundle and then import from it in the user code. The current strategy has two issues:

1. Imports are not portable even if the code is
2. There is no way to express the platform bootstrap instructions.

### Imports are not portable even if the code is:

Imagine a developer creating a fancy button like so:

|  |
| --- |
| import {Component} from 'angular2/angular2';  @Component({  selector: 'fancy-button',  template: ...  })  class FancyButton {  } |

The fancy-button is well behaved in sense that it does not use any of browser API (such as DOM) so it should be possible to run it in the web-worker. However it can't because angular2/angular2 bundle contains DOM renderer which prevents the bundle from being loaded into web-worker. We could cheat, and have two different kinds of bundles which both masquarade as angular2/angular2, but that presents a problem, since we would break components which import DOM specific APIs from DOM angular2/angular2 which would not be present in the web-worker angular2/angular2 bundle.

As it stands now in order to be able to use the fancy-button component in web-worker mode, it will have to change the import from angular2/angular2 to angular2/web\_worker/worker, which prevents reusable components.

### There is no way to express the platform bootstrap instructions

Imagine we have a user details pane such as:

|  |
| --- |
| import {bootstrap} from 'angular2/angular2';  bootstrap(MyApp); |

The above works for DOM renderer. But it already does not work for Dart because due to dart:mirror reasons the bootstrap has to be imported from a library other than angular2/angular2. This means that each platform needs to have a slightly different bootstrap import.

|  |
| --- |
| import 'package:angular2/bootstrap';  main() {  bootstrap(MyApp);  } |

The trouble is that number of platforms in combinatorial. Chose any one from each line:

* Language: JavaScript; Dart
* Marshaling: Browser / WebWorker / NodeJS
* Rendering: DOM, Native

For this reason dart's bootstrap always assumes Browser and DOM, and it can not be used with WebWorker or native. Expecting to have separate bootstrap method for each platform combination is not reasonable.

# Proposal

1. .
2. Create bundles which contain multiple imports, but are tailored for specific usage (such as browser, or web-workers).
3. Deprecate the bootstrap() methBreak up angular2/angular2 import into smaller imports which each deal with specific capabilityod in favor of longer platform().application().component() syntax.

# Detailed Design

## Break up angular2/angular2

No more imports from angular2/angular2. Instead import from specific usage:

* angular2/core: OnInit, Linker,
* angular2/di: Injector
* angular2/metadata: Component, Injectable, Query
* angular2/directive: NgIf, NgFor, NgSwitch
* angular2/pipe: Async
* angular2/forms:
* angular2/compiler: For online or offline compilation of templates.
* angular2/render/dom: DOM renderer implementation
* angular2/render/serializer: Renderer which serialization into a stream
* angular2/platform/\*: Discussed later.

This will make the above example portable between browser and web-worker.

|  |
| --- |
| import {Component} from 'angular2/metadata';  @Component({  selector: 'fancy-button',  template: ...  })  class FancyButton {  } |

Because it is known that angular2/metadata only contains things which are renderer agnostic we can now move the widget into web-worker without having to worry it if will work or change any of its imports. We know that any API which is Browser specific will be imported from angular2/render/dom. This will allow us to easily introspect the code to see if it is web-worker compatible.

## Create bundles based on usage

If we were to create a separate bundle for each import, we would have too many bundles to load in most common cases. For this reason bundles should be tailored around expected usage. Keep in mind that a single bundle can have more than one import inside of it. ([spreadsheet](https://docs.google.com/spreadsheets/d/1XO0WjhRF8JR3NcxwevlsG1Qiunu9LP28rgbrDNCKYQc/edit#gid=0))

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | angular2-\_\_\_\_\_\_.js | | | | | | | |
| **imports** | all | base | compiler | router | http | browser | worker  \_app | worker  \_render |
| zone.js | X |  |  |  |  |  | X | X |
| es6 shim | X |  |  |  |  |  |  |  |
| rxjs | X |  |  |  |  |  |  |  |
| core | X | X |  |  |  |  |  |  |
| di | X | X |  |  |  |  |  |  |
| ~~metadata~~ | ~~X~~ | ~~X~~ |  |  |  |  |  |  |
| directive | X | X |  |  |  |  |  |  |
| pipe | X | X |  |  |  |  |  |  |
| forms | X | X |  |  |  |  |  |  |
| compiler | X |  | X |  |  |  |  |  |
| router | X |  |  | X |  |  |  |  |
| http | X |  |  |  | X |  |  |  |
| animate | X |  |  |  |  |  |  |  |
| upgrade | X |  |  |  |  |  |  |  |
| instrumentation | X |  |  |  |  |  |  |  |
| testing/core |  |  |  |  |  |  |  |  |
| testing/http |  |  |  |  |  |  |  |  |
| testing/router |  |  |  |  |  |  |  |  |
| testing/animate |  |  |  |  |  |  |  |  |
| render/dom | X |  |  |  |  |  |  | X |
| render/serializer |  |  |  |  |  |  | X |  |
| render/deserializer |  |  |  |  |  |  |  | X |
| platform/browser | X |  |  |  |  | X |  | X |
| platform/worker/  app |  |  |  |  |  |  | X |  |
| platform/worker/  render |  |  |  |  |  |  |  | X |
| platform/dart |  |  |  |  |  |  |  |  |
| platform/node |  |  |  |  |  |  |  |  |

Use cases:

* For most developers they would only have to load angular2-all.js in the browser and would not have to think about bundles any more.
* For advanced developers that would do offline compilation of templates the they would have to load zone.js, angular2-base.js, angular2-browser.js and optionally angular2-router.js, and angular2-http.js. Presumably offline compilation is sufficiently advanced topic that multiple bundles are custom built bundle would not be an issue.
* For web-worker applications:
  + render thread: angular2-worker\_render.js
  + worker thread: angular2-worker\_app.js and optionally angular2-router.js, and angular2-http.js.
* For server side pre-rendering in nodejs load: angular2-base.js, angular2-compiler.js, angular2-node.js and optionally angular2-router.js, and angular2-http.js.

## Rework the bootstrap

The bootstrap() method makes too many assumptions about the platform to be cross-platform, for this reason I think we should abandon it and instead go with the long version which would look as follows:

|  |
| --- |
| import {platform} from 'angular2/core';  import {Component} from 'angular2/metadata';  import {COMPILER} from 'angular2/compiler';  import {ROUTER} from 'angular2/router';  import {BROWSER} from 'angular2/platform/browser';  @Component(...)  class MyApp{}  platform([BROWSER]).application([ROUTER, COMPILER]).component(MyApp); |

In this format it is clear which platform we are using and that the application uses router and the template compilation happens online.

Below are few more examples:

Node JS pre-rendering using universal:

|  |
| --- |
| import {platform} from 'angular2/core';  import {Input} from 'angular2/core/metadata';  import {Component} from 'angular2/metadata';  import {Optional} from 'angular2/di/metadata';  import {ROUTER} from 'angular2/router';  import {HTTP} from 'angular2/http';  import {NODE} from 'angular2/platform/node';  import {UNIVERSAL} from 'angular2/platform/universal';  @Component(...)  class MyApp{  @Input() title: string;  constructor(@Optional() foo:Foo)  }  platform([NODE, UNIVERSAL]).application([ROUTER, HTTP]).component(MyApp); |

WebWorker

|  |
| --- |
| import {platform} from 'angular2/core';  import {WORKER\_RENDER, WORKER, SCRIPTS} from 'angular2/platform/worker\_renderer';  platform([WORKER\_RENDER])  .application([WORKER, provide(SCRIPTS, {useValue: ['my-app.js']})]); |
| // file: my-app.js  import {platform} from 'angular2/core';  import {Component} from 'angualr2/metadata';  import {ROUTER} from 'angular2/router';  import {HTTP} from 'angular2/http';  import {WORKER\_APP} from 'angular2/platform/worker\_app';  @Component(...)  class MyApp{}  platform([WORKER\_APP]).application([ROUTER, HTTP]).component(MyApp); |

Dart with mirrors

|  |
| --- |
| import {platform} from 'angular2/core';  import {Component} from 'angualr2/metadata';  import {ROUTER} from 'angular2/router';  import {HTTP} from 'angular2/http';  import {DART} from 'angular2/platform/dart';  import {MIRRORS} from 'angular2/platform/dart\_mirrors';  @Component(...)  class MyApp{}  platform([DART, MIRRORS]).application([ROUTER, HTTP]).component(MyApp); |

Here the transformer would have to rewrite the code to

|  |
| --- |
| import {platform} from 'angular2/core';  import {Component} from 'angular2/metadata';  import {ROUTER} from 'angular2/router';  import {HTTP} from 'angular2/http';  import {DART} from 'angular2/platform/dart';  **import {GENERATED\_STUBS} from './generated\_code\_stubs';**  @Component(...)  class MyApp{}  platform([DART, **GENERATED\_STUBS**]).application([ROUTER, HTTP]).component(MyApp); |

## Ambient Namespaces

All of the above examples assume ES6 or some sort of module loader, but some of our customers, (ngUpgrade specifically) will need SFX bundles which do not rely on loaders, but rather expects the code to be loaded in global namespace. In Angular 2 this namespace is ng. This means that each of the imports will patch more references onto the ng namespace as needed.

At some point we were thinking about having router and http have separate ngRouter and ngHttp namespace but it is not consistent with the angular2/\*.js prefix, se we should stick to just ng namespace.

# Caveats

## Making Getting Started Simple

The drawback of the above proposal is that it is more verbose. To combat that we could have a bootstrap() and an angular2 (all) import provide the existing behavior. While this would make getting started simple it would make portability of code difficult, since code which imports angular2 could not be run in node or in web-workers. We would have to clearly document this. This would be extra important for third party widget libraries.

|  |
| --- |
| import {bootstrap, Component} from 'angular2';  @Component(...)  class MyApp{}  bootstrap(MyApp); |

## Loading Default Directives / Pipes into @View/@Component

[This proposal is contentious, please see alternatives [here](#h.6pb2txdejnyn).]

Currently the @View annotation does not list any Directives/Pipes. This is done so that reusable components are truly hermetic. It does poses a problem that developers have to import these every time they need to use them in the template.

We had an alternate proposal where we would have @BaseView, and would allow subclassing which would add the platform specific directives/pipes to the @View annotation. There are two issues with the proposal:

1. It is really awkward to have BaseView and View. It also makes imports in Angular complicated, because they have to be packed in a separate bundle, so that different environments can define their own.
2. It does not take into account platforms. ie different platforms may have different definitions of these core directives.

An alternative solution which I am proposing here is that we allow the platform bundle to load a set of ambient directives/pipes into the @View annotation. (Hence no need for @BaseView, and no need for complicated imports.) It does mean that components will not be 100% hermetic, but I believe this is a reasonable trade off.

|  |
| --- |
| import {platform, Component, BROWSER} from 'angular2';  @Component({  selector: 'my-app',  template: '<ul><li \*ng-for="#item in items">{{item}}</li></ul>'  // notice no need for NgFor in directives  // directives: [NgFor]  })  class MyApp{  items: string[] = ['one', 'two', 'three'];  }  platform(BROWSER).application().component(MyApp);  // Here the BROWSER defines a set of Directives/Pipes which can be thought of as  // part of the platform and hence always there. |

An important goal for Angular was that it would be possible to design an application which does not use the built in Angular directives. This can be done by declaring a new platform bundle, and loading different items into it.

Here BROWSER would be a list of providers which would be loaded into the platform injector. One could define their own platform bundle by either extending or replacing the BROWSER (see Ionic example below).

#### For example assume the Ionic Platform:

|  |
| --- |
| import {platform, Component, provide} from 'angular2';  @Component({  selector: 'my-app',  template: '<ion-view \*ng-for="#item in items">{{item}}</ion-view>'  // notice no need for NgFor or IonView in directives  // directives: [NgFor, IonView]  })  class MyApp{  items: string[] = ['one', 'two', 'three'];  }  var IONIC = [  BROWSER,  provide(CORE\_DIRECTIVES, {useClass: IonView, multi: true});  provide(CORE\_DIRECTIVES, {useClass: IonToolbar, multi: true});  ]  platform(IONIC).application().component(MyApp);  // Here the BROWSER defines a set of Directives/Pipes which can be thought of as  // part of the platform and hence always there. |

## Documentation Needs

Angular2 has 300+ symbols which it exports. It is too much for single list. Instead it has to be broken down by groups such as core, compiler, di, and so on. But even then the core import has 100+ symbols and it needs to be further subdivided. Current suggestion is as follows.

|  |  |  |
| --- | --- | --- |
| **Import** | **#** | **Description** |
| compiler | 2 | online/offline template compiler |
| core/base | 6 | Linker, platform, application, component |
| core/contract | 12 | Interfaces which application can implement |
| core/error | 5 | Error conditions |
| core/metadata | 53 | Decorators (all things @ (not DI)) |
| core/model | 12 | Change Detection; differs |
| core/ref | 14 | References which framework hands out to the application |
| core/render | 5 | Renderer interface which platforms can implement |
| di | 12 | All things DI |
| di/error | 7 | Errors thrown by DI |
| di/metadata | 17 | Decorators (all things @ related to DI) |
| directive | 8 | Core directives: NgFor, NgIf, etc... |
| form | 29 | Forms |
| inspect | 9 | Inspector for protractor and tools such as Batarangle |
| pipe | 12 | Core pipes: |
| platform/browser | 5 | bootstrap, Title, Ruler, |

Notice how the core is further broken down to base, contract, error, metadata, model, ref, and render. This kind of granularity makes it easier for the user to find the right symbol, but it also makes imports more complicated. For this reason we should support importing the symbols from core as well.

|  |
| --- |
| // are both supported and they are equivalent.  import {Component} from 'core';  import {Component} from 'core/metadata'; |

# Security Considerations

Should have no impact.

# Performance Considerations / Test Strategy

Should have no impact.

# Work Breakdown

See [#4834](https://github.com/angular/angular/issues/4834).

# APPENDIX A: Listing Ambient Directives

When creating a simple dome and using a core directive such as NgFor it is currently necessary to declare it. This section would like to list all of the pros and cons from the different approaches.

1. Declare directives explicitly
2. Subclass the @BaseComponent annotation to @Component and include the core
3. Include core directives through injection.

## 1. Explicit: (current)

|  |
| --- |
| import {Component, CORE\_DIRECTIVES} from 'angular2';  @Component({  template: '<ul><li \*ng-for="#item of items"></li></ul>',  directives: [CORE\_DIRECTIVES]  })  class MyApp{  items: string[];  } |

|  |  |
| --- | --- |
| **pros** | **cons** |
| * 100% hermetic / Explicit | * Wordy / verbose * People often forget this   + hard to give useful error. Best we can do is to say that there is no property ng-for-of to bind to. |

## 2. Subclass the @BaseComponent annotation to @Component

|  |
| --- |
| import {Component} from 'angular2/metadata';  @Component({  template: '<ul><li \*ng-for="#item of items"></li></ul>'  })  class MyApp{  items: string[];  } |

|  |  |
| --- | --- |
| **pros** | **cons** |
| * 100% hermetic * Simple | * It's not immediately obvious that Component includes core directives. * Very complicated import model since we have to have BaseComponent in core but Component in metadata. * We could easily get Component and View out of sync. * Forces platform providers to create yet another annotations. ie @IonicComponent. * Breaks tooling, since it well be difficult for tools to know about Component subclasses and their extension without executing them. |

## 3. Platform directives through injection

|  |
| --- |
| import {Component} from 'angular2;  @Component({  template: '<ul><li \*ng-for="#item of items"></li></ul>'  })  class MyApp{  items: string[];  } |

|  |  |
| --- | --- |
| **pros** | **cons** |
| * Simple * Allows platform vendors to create their own ambient directives | * Not 100% hermetic * Potential for abuse from app developers who do not want to list directives per component. * breaks tooling and ability to statically analyze components |

## 4. Include directives via template imports

|  |
| --- |
| import {Component} from 'angular2;  @Component({  template: `<import rel="ng-directive" href="angular2/directives">  <ul><li \*ng-for="#item of items"></li></ul>`  })  class MyApp{  items: string[];  } |

* piggy back on import html element or create a custom one
* under the hood we can use System.import() - since the compiler is already async, this will not cause any concurrency issues. after compilation is done, there will be no impact on performance.
* href url is resolved using the same semantics as if it was ES import in the current file
* rely on default exports to fetch the list of directives (CORE\_DIRECTIVES)

|  |  |
| --- | --- |
| **pros** | **cons** |
| * simpler than #1 (no need to define `directives` property) * explicit, "hermetic", toolable * moves import into the template which is where you use the imported directives (no need to jump between files) * allows ionic and other 3rd parties to use the same mechanism for distributing directives/components * can be extended to import individual directives to be 100% hermetic (at the cost of verbosity) | * more verbose than #3 * not 100% hermetic (see my comment for #1) * error messages for missing imports might be tricky (same as #1) * Does not work across platforms   + Dart would require complex matching of strings to Symbols.   + Unclear how it would work with Node or NativeScript * Too many unknowns this close to beta. |