Angular 1 to Angular 2 Upgrade Strategy

Status: (Draft)

Authors: misko@google.com, iminar@google.com

This document is published to the web as part of the public Angular Design Docs folder

Objective

Describe the strategies which we will use to minimize the porting distance between Angular 1 to Angular 2.

Background

Angular 1 and Angular 2 have different syntax and semantics. Attempting to convert the whole app wholesale is not feasible for most production applications. This document will discuss ways in which an Angular 1 app can be upgraded/refactored piecemeal.

Upgrade Strategies

- Component Directive Interop
- Service Interop
- Angular 1 module.component() Helper
- Angular 1 Annotation Support
- Router Interop
- Automated Translation Tools
- Angular 1 Best Practice / Upgrade Guides

We now have a github repository to capture and collaborate on these ideas at https://github.com/angular/ngUpgrade

Component Directive Interop Strategy

One basic upgrade strategy is to make it easy to bootstrap Angular 2 components inside AngularJS 1 application (and also AngularJS 1 components inside Angular 2 application). This means that an AngularJS 1 application can be upgraded to Angular 2 application piecemeal over many commits one component at a time.

It is important to understand that at any given time any one DOM element can be owned by only one framework at a time. For this reason it is not possible to mix directives between Angular JS 1 and Angular 2. (Example: `<ng2-comp ng1-click="exp">`) Further more the expression evaluation semantics are slightly different between the two frameworks which further prevents mixing on per element basis.

Components provide a natural boundary for mixing AngularJS 1 and Angular 2. Each component has an associated template which can be then be compiled be either Angular JS 1 or Angular 2. Within each component template particular semantics apply and is fully managed by the appropriate framework version.

Transclusion / Projection

Transclusion (AngularJS 1 terminology) and Projection (Angular 2 terminology) is an important part of the Angular components. This proposal allows for a component in Angular 2 to transclude / project elements from Angular JS 1 and vice-versa.

Testability

Both Angular 1 and Angular 2 components can be unit tested using strategies available for each version. The complications arise when there are interdependencies between the components.

For end to end testing, we need protractor to be able to know when both Angular 1 and Angular 2 contexts are stable as well as how to find bindings on the page for a given context. Exposing a single testability service similar to the one available in Angular 1 could be used to make Protractor understand the hybrid application.

Non-component code

This strategy is based on using component boundaries as the boundaries between the Angular 1 and Angular 2 code. However there is a lot of Angular 1 code that is not component based, this code needs to be upgraded using different strategies described in other sections of this document.

Detailed Design

ngUpgrade.js

Angular 2 will ship with a ngUpgrade.js library which can be included with with Angular 2 inside an existing Angular JS 1 application. This means that both of the frameworks are loaded at the same time on the same page. The ng2interop would then provide glue which will allow fluent way of mixing different components on a single page.

Bootstrap

At the root of the application is the AngularJS 1 RootScope and Injector. Because of the way AngularJS 1 wraps services which call $rootScope.$apply(), Angular JS 1 must remain at the root of the application until the application is fully converted to Angular 2, at which point it can be removed.

Adapters

ng2interop provides API which can adapt/wrap Angular 2 component into Angular JS 1 component and vice-versa.

Wrapping: Angular 2 to AngularJS 1 (todo mvc spike)

```
// Example of Angular 2 Component
@Component({
  selector: 'my-settings',
  properties: ['user'],
  events: ['logout']
})
@View({
  templateUrl: '...'
})
class MySettings {
  user: User;
  logout = new EventEmitter();
}


// AngularJS 1 module declaration:
myAppModule
  .directive('mySettings', ng2upgrade.ng1from2(MySettings));
```

```
// It can be used in Angular1 template

<div ng-controller="AppController as appCtl">

  <my-settings user="appCtl.user" logout="appCtl.logout()">

  </my-settings>

</div>
```

```
// Pseudo-Code Implementation of ng1from2

ng2upgrade.ng1from2 = function(type: Type) {

  return function(ng2) {

    var compAno = Reflector.getAnnotation(Component);

    var protoView = ng2context.compile(type);

    return { // DDO

      scope: true,

      transclude: true,
```

```
      link: function(scope, element, attrs, comps, transclude) {
        var cRef = ng2.createComponent(protoView, element, scope,
                                       transclude);

        // watch all values and marshal them to ng2 component
        compAno.properties.forEach((name) => {
          scope.$watch(attrs[name], (v) => cRef[name] = v);
        });

        // set up listeners on ng2 component
        compAno.events.forEach((name) => {
          cRef[name].observe((e) => {
            return scope.$eval(attrs[name], {$event: e});
          });
        });

        // bridge the change detection
        scope.watch(() => cRef.changeDetector.detectChanges();
      }
    };
  };
}
```

Wrapping: AngularJS 1 to Angular 2

A component is often composed from other components. For this reason it is also desirable to use existing AngularJS 1 components in Angular 2.

```
// Assuming we have some component defined per AngularJS 1
myAppModule.directive('zippy', ...);

// Create ng2 component from ng1
var Ng2Zippy = ng2upgrade.ng2from1(myAppModule, 'ng2-zippy');


// Then use the ng1 component from ng2 component.
@Component({
  selector: 'some-component'
})
@View({
  template: '<zippy [title]="myTitle" (close)="onClose()"></zippy>',
  directives: [Ng2Zippy]
})
class SomeComponent {
  myTytle = "Some text";
  onClose() {
    // do something
  }
}
```

```
// Pseudo-code implementation for ng2from1
ng2upgrade.ng1from2 = function(module, name): Type {
  var properties: string[] = [];
  var events: string[] = [];
  var viewLinkFn = null;

  module.run(function($inject, $compile) {
    var ddo = $inject.get(name + 'Directive');
    ddo.scope.forEach((key, value) => {

      if (key == '=' || key == '@') {

        properties.push(value);

      } else if (key == '&') {

        events.push(value);

      }

    });
```

```
    viewLinkFn = $compile(ddo.template);

  });

  @Component({
    selector: selector,
    properties: properties,
    events: events
  })
  @View({
    template: ''
  })
  class Ng2Component {
    constructor(scope: Scope, eRef: ElementRef) {
      this.scope = scope.createScope(true);

      // create event hooks
      events.forEach((name) => {
        this[name] = new EventEmitter());
        // pass callback functions to ng1
        this.scope[name] = (e) => this[name].next(e);
      }
      // create view and attach it as a child of this element.
      this.elements = viewLinkFn(scope);
      this.elements.appendTo(eRef.native);
    }

    onChange(changes: StringMap<string, any>): void {
      changes.forEach((key, value) => {
        // marshall all property changes to ng1
        this.scope[key] = value;
      });
    }

    onDestroy() {
      this.elements.remove();
      this.scope.$destroy()
    }
  }
  return Ng2Component;
}
```

Transclusion / Reprojection

It is important that a component's child elements can be transcluded / reprojected into the component destination.

Transclusion from ng1 into ng2 component

- The AngularJS 1 component created from Angular 2 component will have `transclude: true` set of the DDO
- The transclude function generated by AngularJS 1 will be inserted into the Injector of Angular 2 component
- The `ngTransclude` will retrieve the transclude function from the injector.

```
// AngularJS 1 template
<div>
  <ng2-component>
    {{interpolated}} content with <ng1-component></ng1-component>
  </ng2-component>
</div>
```

```
@Component({
  selector: 'ng2-component'
})
@View({
  template: '<div><ng-transclude></ng-transclude></div>`,
  directives: [ng2interop.ngTransclude]
})
class Ng2Component {
  ...
}
```

```
// Pseudo Implementation of Angular2 ngTransclude
@Directive({
  selector: 'ng-transclude'
})
class NgTransclude {
  constructor(@Inject('transcludeFn') transcludeFn,
              eRef: ElementRef) {
    this.element = eRef.native;
    this.transcludeElements = transcludeFn();
    this.transcludeElements.appendTo(this.element);
```

```
  }

  onDestroy() {
    this.transcludeElements.remove()
  }
}
```

Transclusion from ng2 into ng1 component

- The `ng2from1` adapter attaches the children elements to the DOM property
- The AngularJS 1 `ngTransclude` directive looks for the attached DOM and moves it to the current location.

```
// AngularJS 2 template
<div>
  <ng1-component>
    {{interpolated}} content with <ng2-component></ng2-component>
  </ng1-component>
</div>
```

```
myAppModule.directive('ng1component', function() {
  return {
    scope: {},
    template: '<div><ng-transclude></ng-transclude></div>',
    controller: …,
  }
});
```

```
var Ng1Component
  = ng2interop.ng2from1(myAppModule, 'ng1component');
```

```
// Pseudo Implementation of AngularJS1 ngTransclude
myAppModule.directive('ngTransclude', function () {
  return {
    link: function(scope, element, attr) {
      // get a hold of the ng2 element.
      var transitonElement = findTransitionElement();

      // get a hold of elements which need to be transcluded
      var transcludeElements = transcludeElement.children();
      // transclude the DOM.
      element.append(transcludeElements);

      // restor DOM on removal.
      element.bind('$remove', function() {
        transitionElement.append(transcludeElements);
      });
    }
  };
});
```

Injectors

- AngularJS 1 application has to be at the root, because it only has one injector.
    - provide mappings of AngularJS 1 services into Angular 2 root injector.
- Angular 2 component can be placed at any location in Angular JS 1 component.

- Angular 2 services declared in the root injector can be exported to AngularJS 1.

```
var myApp = angular.module(...);

ng2interop.bindInjector(
  myApp,
  { // import list from AngularJS 1 to Angular 2
    '$injector': Ng1Injector
  },
  { // exportAs list from Angular 2 to AngularJS 1
    'ng2Injector': Injector,
    'ng2Compiler': Compiler
  })
```

```
// Possible implementation
ng2interop.bindInjector = function (module, imports, exports) {
  var ng2Injector = null;
  var ng1Injector = null;
  var ng2module = [bind('ng1Injector').toFactory(() => ng1Injector)];

  exports.forEach((ng1Id, ng2Type) => {
    module.factory(ng1Id, () => ng2Injector.get(ng2Type));
  });
  imports.forEach((ng1Id, ng2Type) => {

    ng2module.push(

       bind(ng2Type).toFactory(() => ng1Injector.get(ng1Id)));

  });

  module.factory('ng2injector', function() {
    return Injector.fromModule(ng2Module);
  });
  module.run(function($injector) {
    ng1Injector = $injector;
    ng2Injector = $injector.get('ng2injector');
  });
}
```

$apply / Zone

- All digests must start with AngularJS 1 `$rootScope`.
- AngularJS1 scopes and Angular 2 `ChangeDetectors` will form a mixed tree.
- All asynchronous AngularJS 1 services already call `$rootScope.$apply()`.
- All asynchronous Angular 2 services go through `ZoneJS`, but it has to be redirected to `$rootScope.$apply()` instead to `LifeCycle.tick()`.

Service Interop Strategy

When mixing frameworks it is important that the frameworks can inject values from each other injectors. AngularJS 1 uses String tokens while Angular 2 uses types for references. Additionally Angular JS 1 has single injector, while Angular 2 has hierarchical injectors with visibility rules. Because of these differences the developers will have to explicitly list how to map injectables from Angular JS 1 into Angular 2 and vice-versa.

Global State

The browser environment contains several global states. Examples:

- cookies
- location
- global click handlers

In order to prevent conflicts, these states need a single owner during the transitional period. In practical terms this means that there should be only one instance of router, cookie service etc that is shared between Angular 1 and Angular 2 contexts.

Non-component code

Code that isn't a service or a component must be upgraded using different strategies:

- controllers and their templates
- decorator (attribute) directives
- element transclusion directives
- filters
- form validators, parsers and formatters

Controllers and their templates

Controllers and their templates are compatible with the proposed strategy, which allows controller templates to take advantage of Angular 2 components. These controllers can be slowly upgraded over to Angular 2 on file by file basis.

It is however highly recommended that the controllers and templates are upgraded to the controllerAs syntax in order to prevent scoping issues.

Decorator (attribute) directives

These directives can't be safely mixed between the Angular 1 and Angular 2 code and must be reimplemented in both contexts, however it is also important to consider that many attribute directives are not necessary in Angular 2. There are two kinds of directives that will be unnecessary in Angular 2:

- directives that listen for events - use `(submit)` binding instead of `ng-submit` directive
- directives that binding to properties of an element - use `[disabled]` binding instead of `ng-disabled` directive.

Custom structural and element transclusion directives

Custom structural directives (custom repeater, if, switch directives) are very rarely seen in practice. These directives will need to be reimplemented in Angular 2.

Angular 2 doesn't have a concept of element transclusion directive. These directives are usually fall into the category of custom structural directives and as such will need to be reimplemented.

Replace directives

Directives that replace their host element (replace: true directives in Angular 1) are not supported in Angular 2. In many cases these directives can be upgraded over to regular component directives.

There are cases when regular component directives are not going to work, in those cases alternative approaches can be used. For example for svg see: https://github.com/mhevery/ng2-svg

Filters

Filters created as pure functions should be trivial to upgrade manually. However this process can be further simplified by providing an adapter for Angular 2 in ng2interop.js.

```
var myFilterFactory = ['myService', function(myService) { … }];


myAppModule.filter(myFilterFactory);
var myFilterAsPipe = ng2upgrade.filterToPipe(myFilterFactory);


// TODO: register pipe with Angular 2, api still in flux
```

Form validators, parsers and formatters

The forms api is still in flux, but we expect that writing an adapter for this Angular 1 code will not be trivial and full compatibility will be hard to achieve.

TODO: update once forms in A2 have more features

Angular 1 module.component() Helper

This Angular 1 helper will make it easier to create "component" directives in Angular 1. The API should map reasonably well to the Angular 2 concept of a component.

See the discussions:

- https://github.com/angular/angular.js/issues/10007
- https://github.com/angular/angular.js/pull/12166

API Considerations

- What are the key use cases that we want to support?
- What are sensible defaults so that on the whole not many options need to be specified?

Implementation Considerations

- Is this change purely syntactic sugar or do we need to make real changes to the compiler to support this?

Angular 1 Annotation Support

There are a number of 3rd party projects with broadly similar goals to this:

- https://github.com/hannahhoward/a1atscript
- https://github.com/pbastowski/angular2-now
- https://github.com/mikeryan52/angular-decorators
- https://github.com/eaze/ng-classy

The aim is to get some convergence here, either into a single project, or at least to a single API specification.

API Considerations

If we are going to converge on a single API then these are the areas where we need agreement.

Angular 2 Shims

These are annotations that map in some way to those used in Angular 2. There is not an exact semantic relationship so we need to discuss whether they create more confusion than good by being nearly but not exactly what will be used in Angular 2. An alternative way of thinking about this is whether they allow us to write code for Angular 1 that will require minimal changes when upgrading to Angular 2.

- @Inject
- @Directive
- @Component

- @View
- @RouteConfig (for ComponentRouter)

Angular 1 Specific Features

These are annotations and functions that have no meaning in Angular 2. In this sense they are purely to improve the development experience in Angular 1 and are not related to upgrading

- @Controller
- @Service
- @Factory
- @Provider
- @Value
- @Constant
- @Filter
- @Animation
- Module (a1atscript + angular-decorators) / SetModule (angular2-now)
- @Module + @AsModule (a1atscript)
- @Config
- @Run
- bootstrap
- @Require (angular-decorators)
- @Transclude (angular-decorators)
- @DirectiveObject (a1atscript) DDO??

In addition some of the libraries offer automatic namespacing of directives and services.

3rd Party Library Features

These are annotations that are related to libraries that are not part of the core Angular project.

- @State (for ui-router)
- @MeteorMethod (for Meteor)

Component Router Interop Strategy

This is a subset of the **Component Interop** strategy above. If that strategy works then we almost get this strategy for free. If not then we might still be able to get this strategy to work in a cut down version of the more general case.

Automated Translation Tools

There are likely to be a set of tasks when upgrading an Angular 1 app to Angular 2 that can be automated. This strategy will look to identify and implement the most useful ones. It is possible that the ideas for some of these tools will come out of the current work to upgrade some internal Google apps.

Angular 1 Best Practice / Upgrade Guides

While interop, syntax support and tools will provide the technology behind an upgrade, we still need to help developers with

- guides to write better Angular 1 code that will be easier to upgrade
- instructions on how to upgrade specific types of Angular 1 code

//

Caveats

You may need to describe what you did not do or why simpler approaches don't work. Mention other things to watch out for (if any).

Security Considerations

How you'll be secure

Performance Considerations / Test Strategy

How you'll be fast.

Work Breakdown

Description of development phases and approximate time estimates.