Upgrading for performance

Angular is the name for the Angular of today and tomorrow. *AngularJS* is the name for all 1.x versions of Angular.

This guide describes some of the built-in tools for efficiently migrating AngularJS projects over to the Angular platform, one piece at a time. It is very similar to Upgrading from AngularJS with the exception that this one uses the {@link downgradeModule downgradeModule()} helper function instead of the {@link UpgradeModule UpgradeModule} class. This affects how the application is bootstrapped and how change detection is propagated between the two frameworks. It allows you to upgrade incrementally while improving the speed of your hybrid applications and leveraging the latest of Angular in AngularJS applications early in the process of upgrading.

Preparation

Before discussing how you can use `downgradeModule()` to create hybrid apps, there are things that you can do to ease the upgrade process even before you begin upgrading. Because the steps are the same regardless of how you upgrade, refer to the Preparation section of Upgrading from AngularJS.

Upgrading with `ngUpgrade`

With the `ngUpgrade` library in Angular you can upgrade an existing AngularJS application incrementally by building a hybrid app where you can run both frameworks side-by-side. In these hybrid applications you can mix and match AngularJS and Angular components and services and have them interoperate seamlessly. That means you don't have to do the upgrade work all at once as there is a natural coexistence between the two frameworks during the transition period.

How `ngUpgrade` Works

Regardless of whether you choose `downgradeModule()` or `UpgradeModule`, the basic principles of upgrading, the mental model behind hybrid apps, and how you use the {@link upgrade/static upgrade/static} utilities remain the same. For more information, see the How `ngUpgrade` Works section of Upgrading from AngularJS.

The Change Detection section of Upgrading from AngularJS only applies to applications that use `UpgradeModule`. Though you handle change detection differently with `downgradeModule()`, which is the focus of this guide, reading the Change Detection section provides helpful context for what follows.

Change Detection with `downgradeModule()` As mentioned before, one of the key differences between `downgradeModule()` and `UpgradeModule` has to do

with change detection and how it is propagated between the two frameworks.

With `UpgradeModule`, the two change detection systems are tied together more tightly. Whenever something happens in the AngularJS part of the app, change detection is automatically triggered on the Angular part and vice versa. This is convenient as it ensures that neither framework misses an important change. Most of the time, though, these extra change detection runs are unnecessary.

`downgradeModule()`, on the other side, avoids explicitly triggering change detection unless it knows the other part of the application is interested in the changes. For example, if a downgraded component defines an `@Input()`, chances are that the application needs to be aware when that value changes. Thus, `downgradeComponent()` automatically triggers change detection on that component.

In most cases, though, the changes made locally in a particular component are of no interest to the rest of the application. For example, if the user clicks a button that submits a form, the component usually handles the result of this action. That being said, there *are* cases where you want to propagate changes to some other part of the application that may be controlled by the other framework. In such cases, you are responsible for notifying the interested parties by manually triggering change detection.

If you want a particular piece of code to trigger change detection in the AngularJS part of the app, you need to wrap it in [scope.*apply*()](*https* : *//docs.angularjs.org/api/ng/type/*rootScope.Scope#$apply). Similarly, for triggering change detection in Angular you would use {@link NgZone#run ngZone.run()}.

In many cases, a few extra change detection runs may not matter much. However, on larger or change-detection-heavy applications they can have a noticeable impact. By giving you more fine-grained control over the change detection propagation, `downgradeModule()` allows you to achieve better performance for your hybrid applications.

Using `downgradeModule()`

Both AngularJS and Angular have their own concept of modules to help organize an application into cohesive blocks of functionality.

Their details are quite different in architecture and implementation. In AngularJS, you create a module by specifying its name and dependencies with angular.module(). Then you can add assets using its various methods. In Angular, you create a class adorned with an {@link NgModule NgModule} decorator that describes assets in metadata.

In a hybrid application you run both frameworks at the same time. This means that you need at least one module each from both AngularJS and Angular.

For the most part, you specify the modules in the same way you would for a regular application. Then, you use the `upgrade/static` helpers to let the two frameworks know about assets they can use from each other. This is known as "upgrading" and "downgrading".

Definitions:

- *Upgrading*: The act of making an AngularJS asset, such as a component or service, available to the Angular part of the application.
- *Downgrading*: The act of making an Angular asset, such as a component or service, available to the AngularJS part of the application.

An important part of inter-linking dependencies is linking the two main modules together. This is where `downgradeModule()` comes in. Use it to create an AngularJS module—one that you can use as a dependency in your main AngularJS module—that will bootstrap your main Angular module and kick off the Angular part of the hybrid application. In a sense, it "downgrades" an Angular module to an AngularJS module.

There are a few things to note, though:

1. You don't pass the Angular module directly to `downgradeModule()`. All `downgradeModule()` needs is a "recipe", for example, a factory function, to create an instance for your module.
2. The Angular module is not instantiated until the application actually needs it.

The following is an example of how you can use `downgradeModule()` to link the two modules.

```
// Import `downgradeModule()`.
import { downgradeModule } from '@angular/upgrade/static';

// Use it to downgrade the Angular module to an AngularJS module.
const downgradedModule = downgradeModule(MainAngularModuleFactory);

// Use the downgraded module as a dependency to the main AngularJS module.
angular.module('mainAngularJsModule', [
  downgradedModule
]);
```

Specifying a factory for the Angular module As mentioned earlier, `downgradeModule()` needs to know how to instantiate the Angular module. It needs a recipe. You define that recipe by providing a factory function that can create an instance of the Angular module. `downgradeModule()` accepts two types of factory functions:

1. `NgModuleFactory`
2. `(extraProviders: StaticProvider[]) => Promise<NgModuleRef>`

When you pass an `NgModuleFactory`, `downgradeModule()` uses it to instantiate the module using {@link platformBrowser platformBrowser}'s {@link PlatformRef#bootstrapModuleFactory bootstrapModuleFactory()}, which is compatible with ahead-of-time (AOT) compilation. AOT compilation helps make your applications load faster. For more about AOT and how to create an `NgModuleFactory`, see the Ahead-of-Time Compilation guide.

Alternatively, you can pass a plain function, which is expected to return a promise resolving to an {@link NgModuleRef NgModuleRef} (that is, an instance of your Angular module). The function is called with an array of extra {@link StaticProvider Providers} that are expected to be available on the returned `NgModuleRef`'s {@link Injector Injector}. For example, if you are using {@link platformBrowser platformBrowser} or {@link platformBrowserDynamic platformBrowserDynamic}, you can pass the `extraProviders` array to them:

```
const bootstrapFn = (extraProviders: StaticProvider[]) => {
  const platformRef = platformBrowserDynamic(extraProviders);
  return platformRef.bootstrapModule(MainAngularModule);
};
// or
const bootstrapFn = (extraProviders: StaticProvider[]) => {
  const platformRef = platformBrowser(extraProviders);
  return platformRef.bootstrapModuleFactory(MainAngularModuleFactory);
};
```

Using an `NgModuleFactory` requires less boilerplate and is a good default option as it supports AOT out-of-the-box. Using a custom function requires slightly more code, but gives you greater flexibility.

Instantiating the Angular module on-demand Another key difference between `downgradeModule()` and `UpgradeModule` is that the latter requires you to instantiate both the AngularJS and Angular modules up-front. This means that you have to pay the cost of instantiating the Angular part of the app, even if you don't use any Angular assets until later. `downgradeModule()` is again less aggressive. It will only instantiate the Angular part when it is required for the first time; that is, as soon as it needs to create a downgraded component.

You could go a step further and not even download the code for the Angular part of the application to the user's browser until it is needed. This is especially useful when you use Angular on parts of the hybrid application that are not necessary for the initial rendering or that the user doesn't reach.

A few examples are:

- You use Angular on specific routes only and you don't need it until/if a user visits such a route.
- You use Angular for features that are only visible to specific types of users; for example, logged-in users, administrators, or VIP members. You don't

need to load Angular until a user is authenticated.
- You use Angular for a feature that is not critical for the initial rendering of the application and you can afford a small delay in favor of better initial load performance.

Bootstrapping with `downgradeModule()`

As you might have guessed, you don't need to change anything in the way you bootstrap your existing AngularJS application. Unlike `UpgradeModule`—which requires some extra steps— `downgradeModule()` is able to take care of bootstrapping the Angular module, as long as you provide the recipe.

In order to start using any `upgrade/static` APIs, you still need to load the Angular framework as you would in a normal Angular application. You can see how this can be done with SystemJS by following the instructions in the Upgrade Setup guide, selectively copying code from the QuickStart github repository.

You also need to install the `@angular/upgrade` package using `npm install @angular/upgrade --save` and add a mapping for the `@angular/upgrade/static` package:

Next, create an `app.module.ts` file and add the following `NgModule` class:

import { NgModule } from '@angular/core'; import { BrowserModule } from '@angular/platform-browser';

@NgModule({ imports: [BrowserModule] }) export class MainAngularModule { // Empty placeholder method to satisfy the `Compiler`. ngDoBootstrap() {} }

This bare minimum `NgModule` imports `BrowserModule`, the module every Angular browser-based app must have. It also defines an empty `ngDoBootstrap()` method, to prevent the {@link Compiler Compiler} from returning errors. This is necessary because the module will not have a `bootstrap` declaration on its `NgModule` decorator.

You do not add a `bootstrap` declaration to the `NgModule` decorator since AngularJS owns the root template of the application and `ngUpgrade` bootstraps the necessary components.

You can now link the AngularJS and Angular modules together using `downgradeModule()`.

import { platformBrowserDynamic } from '@angular/platform-browser-dynamic'; import { downgradeModule } from '@angular/upgrade/static';

const bootstrapFn = (extraProviders: StaticProvider[]) => { const platformRef = platformBrowserDynamic(extraProviders); return platformRef.bootstrapModule(MainAngularModule); }; const downgradedModule = downgradeModule(bootstrapFn);

angular.module('mainAngularJsModule', [downgradedModule]);

The existing AngularJS code works as before *and* you are ready to start adding Angular code.

Using Components and Injectables

The differences between `downgradeModule()` and `UpgradeModule` end here. The rest of the `upgrade/static` APIs and concepts work in the exact same way for both types of hybrid applications. See Upgrading from AngularJS to learn about:

- Using Angular Components from AngularJS Code. *NOTE: If you are downgrading multiple modules, you need to specify the name of the downgraded module each component belongs to, when calling `downgradeComponent()`.*
- Using AngularJS Component Directives from Angular Code.
- Projecting AngularJS Content into Angular Components.
- Transcluding Angular Content into AngularJS Component Directives.
- Making AngularJS Dependencies Injectable to Angular.
- Making Angular Dependencies Injectable to AngularJS. *NOTE: If you are downgrading multiple modules, you need to specify the name of the downgraded module each injectable belongs to, when calling `downgradeInjectable()`.*

While it is possible to downgrade injectables, downgraded injectables will not be available until the Angular module that provides them is instantiated. In order to be safe, you need to ensure that the downgraded injectables are not used anywhere *outside* the part of the application where it is guaranteed that their module has been instantiated.

For example, it is *OK* to use a downgraded service in an upgraded component that is only used from a downgraded Angular component provided by the same Angular module as the injectable, but it is *not OK* to use it in an AngularJS component that may be used independently of Angular or use it in a downgraded Angular component from a different module.

Using ahead-of-time compilation with hybrid apps

You can take advantage of ahead-of-time (AOT) compilation in hybrid applications just like in any other Angular application. The setup for a hybrid application is mostly the same as described in the Ahead-of-Time Compilation guide save for differences in `index.html` and `main-aot.ts`.

AOT needs to load any AngularJS files that are in the `<script>` tags in the AngularJS `index.html`. An easy way to copy them is to add each to the `copy-dist-files.js` file.

You also need to pass the generated `MainAngularModuleFactory` to `downgradeModule()` instead of the custom bootstrap function:

import { downgradeModule } from '@angular/upgrade/static'; import { MainAngularModuleNgFactory } from '../aot/app/app.module.ngfactory';

const downgradedModule = downgradeModule(MainAngularModuleNgFactory);

angular.module('mainAngularJsModule', [downgradedModule]);

And that is all you need to do to get the full benefit of AOT for hybrid Angular applications.

Conclusion

This page covered how to use the {@link upgrade/static upgrade/static} package to incrementally upgrade existing AngularJS applications at your own pace and without impeding further development of the app for the duration of the upgrade process.

Specifically, this guide showed how you can achieve better performance and greater flexibility in your hybrid applications by using {@link downgradeModule downgradeModule()} instead of {@link UpgradeModule UpgradeModule}.

To summarize, the key differentiating factors of `downgradeModule()` are:

1. It allows instantiating or even loading the Angular part lazily, which improves the initial loading time. In some cases this may waive the cost of running a second framework altogether.
2. It improves performance by avoiding unnecessary change detection runs while giving the developer greater ability to customize.
3. It does not require you to change how you bootstrap your AngularJS application.

Using `downgradeModule()` is a good option for hybrid applications when you want to keep the AngularJS and Angular parts less coupled. You can still mix and match components and services from both frameworks, but you might need to manually propagate change detection. In return, `downgradeModule()` offers more control and better performance.