Angular Universal Design

author: viks@google.com, arick@google.com
last update: 2017-03-03
Status: Implemented

Objective

To move/rebuild the functionality of Universal as part of the core of Angular.

Background

Angular allows different renderers for different platforms - like browser, web worker and server. Angular Universal refers to the API and tools that allow for rendering of an Angular application on the server platform. Developers want to render Angular applications on the server side for different reasons:

- Faster App load
- SEO
- Link Preview / AMP

The current implementation of Angular Universal exists as a separate package outside of the core Angular project. We want to bring Angular Universal into the core Angular project so that server side rendering is better supported by Angular. It allows for a cleaner design of the Angular Universal API. It allows for better support for Universal by making necessary changes to the implementation of the core packages(Ex. Http module, Metatag service). Also, this would allow us to keep Angular Universal in sync with changes in the core Angular platform as it follows the same release process as rest of Angular. Finally it provides a solid platform to develop next set of features in Angular Universal.

This document describes the design of the API for Angular Universal that will be part of core Angular, along with common usage patterns in Universal apps. It also provides the implementation details of the server side platform and renderer that will be part of angular core.

There would be a separate guides that details the support tools in Angular Universal and migration of apps from the prior version of Angular Universal to the one that will be part of core Angular.

Detailed Design

API

Interface

The primary interface to Universal will be a new method on platformServer() called renderModuleFactory(). This method takes a previously compiled factory, bootstraps the Angular application into a virtual DOM tree parsed from the document, and serializes the resulting DOM state to a string.

platformServer() itself will be extended to take in an initial state vector, consisting initially of an HTML document that will be parsed into the initial DOM tree and a URL that will seed the location service. If either one of these is not provided, platformServer()will utilize that option's current default.

The whole operation can be thought of as a transformation from the static index.html contents to an index.html with the Angular application pre-rendered.

```
import {platformServer} from '@angular/platform-server';
import {AppServerModuleNgFactory} from './server.ngfactory';
...
// Per Request to avoid cross-talk.
const platform = platformServer([
  {provide: INITIAL_CONFIG, useValue: {
    url: '...',
    document: '...',
  }}
]);

// Long-form rendering
platform
  .bootstrapModuleFactory(MyAppModuleFactory)
  .then((ref: ApplicationRef) => ref.isStable.filter(v => v).toPromise())
  .then(() => platform.injector.get(PlatformState).renderToString())
  .then(html => ...)
  .then(() => platform.destroy());
```

We will have a JIT mode that will be strictly for dev-mode only since the performance cost of doing JIT per request will be high.

```
platform
  .bootstrapModule(MyAppModule)
  .then((ref: ApplicationRef) => ref.isStable.filter(v => v).first().toPromise())
  .then(() => platform.injector.get(PlatformState).renderToString())
  .then(html => ...)
  .then(() => platform.destroy());
```

There will be helper methods that will provide a shorter way to render to string.

```
Import {renderModuleFactory, renderModule} from '@angular/platform-server'
```

```
// Short-form rendering
renderModuleFactory(MyAppModuleFactory, {url: '...', document: '...', providers:
[..]}).then(html => ...);
```

```
// Short-form with JIT compilation(For dev mode only)
renderModule(MyAppModule, {url: '...', document: '...', providers: [..]});
```

Server → Client Transitions

While not necessary, many applications will want to bootstrap a client-side application "on top of" the server-side rendered app. There are two concerns when doing so: replacement of the server-rendered root component with the client rendered version, and removal of the `<style>`s which are added to the client side app.

The first concern is addressed naturally by the function of the root element in index.html. Contents of the root element are naturally replaced by the client rendered application on bootstrap.

The second concern is more challenging. `<style>` tags added to `<head>` during server rendering need to be removed when the client app is bootstrapped. To do this, Angular needs to be able to associate the bootstrap of server and client applications, so the client can clean up leftover state from the server.

This is accomplished by using `BrowserModule.withServerTransition()` in the client app, which requires the provision of an APP_ID and sets up additional providers to correctly transition server to client apps.

Usage Patterns

We expect most applications will have a specific module for each platform (browser, server) on which they will run. This module will be used to configure the application specifically for this platform. It will install the platform module (BrowserModule or ServerModule), set providers for any services which have platform-specific implementations, and specify the component to be bootstrapped.

Also expected is that most applications will share the majority of their configuration between environments, by "extending" the application module (which imports BrowserModule) by importing it into the application server module.

For a hypothetical news application, this structure looks like this:

```
// app/platform/browser.ts:
```

```
@NgModule({
  bootstrap: [NewsApp],
  declarations: [NewsApp],
  imports: [
    BrowserModule.withServerTransition({appId: 'news'}),
    MaterialModule,
    HttpModule,
    NewsAppRouteModule,
  ],
  providers: [
    NewsAggregationService,
    {provide: AuthService, useClass: OauthAuthService},
    {provide: NewsService, useClass: ClientNewsService},
  ],
})
export class NewsAppBrowserModule {}
```

```
// app/platform/server.ts:

@NgModule({
  bootstrap: [NewsApp],
  imports: [
    NewsAppBrowserModule,
    ServerModule,
  ],
  providers: [
    {provide: AuthService, useClass: CookieAuthService},
    {provide: NewsService, useClass: ServerNewsService},
  ],
})
export class NewsAppServerModule {}
```

This design allows customization of the providers and specific modules used for each platform while allowing most of the application configuration to live in a single module.

Implementation

Platform Server

ApplicationRef will have an extra isStable Observable that indicates when the Application is stable. We will reuse the whenStable from the Testability class.

PlatformState will have methods to get the string through renderToString() and getDOM() method to get access to the DOM tree in tests.

DOM & Renderer

We will use the current Angular server renderer which uses parse5 DOM Adapter to render to a string. We will make changes to the renderer as needed to support further Universal use cases.

Http

The changes in @angular/http necessary for Universal support are being reviewed separately, and are covered in the Stable HTTP Design Doc.

Location

Currently platformServer() provides no support for location. ServerPlatformLocation is provided to satisfy dependencies on PlatformLocation, but it throws a not-implemented error when any methods are invoked. @angular/platform-server will need a proper implementation of PlatformLocation on the server side, with emulated push/pop state behavior.

Angular also has a LocationStrategy interface, which provides a higher level location abstraction on top of PlatformLocation. These two interfaces align fairly well. This is important because a MockLocationStrategy class exists already that has most of the logic for pushing/popping state already implemented.

As part of porting Universal, then, ServerPlatformLocation will be rewritten to properly support a platform-level location state on the server, using the code from MockLocationStrategy to accelerate the process.

Metatag Services

We would reuse the metatag service in platform-browser to manipulate meta tags needed to support SEO/link preview.

Zones

Universal is already Zone aware with Zone.js patching the core Node APIs like setTimeout, setInterval and Promises. However Zone doesn't handle async tasks in custom Node library like XHR2 that provides the server XHR implementation for Http Module.

We will add hooks to the implementation of XHR in platformServer that will explicitly call scheduleMacroTask in Zone in order to track outstanding XHR requests.

We will provide guidelines on how to do the same for any other Node libraries that your application might end up using.

Testing

We would have to provide a way to allow component developers to write unit tests that verify that their component does the initial rendering properly in platformServer

environment along with their Server specific AppModule. The unit test can then verify that the DOM from the PlatformState contains expected nodes. (We might have to provide some helper functions to make it easier to write such unit tests similar to the short form render method)

Protractor should not need any changes to work with prerendered Angular applications. End to end tests will just verify that the final client bootstrapped application works as expected.

Caveats

This doc does not address the changes that will be needed to other parts of the current Universal ecosystem, such as the Express middleware.

Security Considerations

In Angular, the platform is the highest level of singleton. Every instance of platformServer() will contain a separate instance of the running Angular application. Re-use of the same platform for a second rendering will throw an error automatically.

Utilized in this way, state from one instance of a server-side rendered Angular app should be prevented from leaking across into another instance of the app.

Performance Considerations / Test Strategy

Other than the unit tests, there will be end to end tests using Protractor that the initial render through Universal works as expected.

We should have performance tests to track the initial render time from a browser (Need to figure what tools we can use for this, external to Google)

We should consider load testing for concurrent requests to render Angular apps and make sure there are no issues here.

Documentation Plan

The new APIs will need to be documented, with code samples.

The docs team will put together a guide for angular.io on use of Universal with an Express-based node server, for the SSR and prerendering cases.

We should also have a Best Practices doc to avoid issues around memory/resource leaks and how developers can load test their server application.

Style Guide Impact

It might be prudent to document the structure of application entrypoint described above for advanced/larger applications that may wish to ensure compatibility with Universal as a best practice.

Public API Surface Impact

Yes, changes to platformServer() are described above. Changes to the Http package as mentioned above.

Developer Experience Impact

Ideally, the developer experience of setting up an application for server-side rendering becomes easier.

Breaking Changes

It should not be a breaking change for users of platformServer() as new APIs will be additive.

The changes to @angular/http may be breaking for users who configure Http for DI manually (for example, if overriding functionality). These APIs are @experimental.

Deprecations

The old Universal(angular2-universal) can be deprecated and removed at will.

Rollout Plan

All API changes will be part of Angular 4.0.

We will scope out the Google rollout after 4.0 (in Q2). We will have a separate design doc for that.

Maintenance Plan

The core team will handle maintenance of the new APIs in platformServer().

Work Breakdown

- Http module refactoring
- Refactor platform independent parts (DOM Adapter/Metatag service) out of platformBrowser
- Implement Location in platform server
- Implement platform server API and ServerModule
- E2E tests