Code Lazy Loading

*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

Describe the lazy loading strategy of Angular 2. This document will be Dart centric, because Dart lazy loading constraints are stronger than that of JavaScript. JavaScript will follow the same strategy with JavaScript equivalent idioms.

# Background

Lazy loading of code involves these steps:

1. Using a promise based API to load code asynchronously.
2. (optionally, not needed for reflection mode) initializing the reflector with lazy loaded specific stubs.
3. Creating a child injector with configuration loaded from the lazy loaded code.

# Prior Art

* <http://requirejs.org/>
* <http://webpack.github.io/>
* <https://github.com/systemjs/systemjs>

# Detailed Design

The basic lazy loading primitive of Angular is a Component. Here is an example:

|  |
| --- |
| import 'package:library\_a.dart' deferred as lib\_a;  someFn() {  lib\_a.loadLibrary().then((\_) {  // Needed to load reflector stubs from deferred library.  lib\_a.initReflector();  // return the root Component  return lib\_a.SomeComponent;  });  } |

Once the Component is retrieved lazily it can be used with any of the Angular's component loading API (either with router or dynamic component loading).

## Tree Shaking for Code Transforms

It is important that code transformation tools (such as Angular's Dart stub transform) can correctly identify the root Component and tree shake the dependencies appropriately.

The algorithm always starts with a root component. The root component can come in two flavors, either root application component or lazy loaded component.

1. Root application component is identified with: bootstrap(RootComponent);
2. The lazy loaded component is identified by a call to loadLibrary() which returns a component. (see code example above).

In both cases the reflector stub generation starts with the root component and includes all statically reached stubs. (See: [Dart M0: Transform](https://github.com/angular/angular/issues?q=is%3Aissue+milestone%3A%22Dart%3A+M0+Transformers%22+) milestone) All of the reflector stubs are then kept with the library which contains the component.

It is important that after lazy loading of the code the appropriate initReflector() method gets called. As of right now, that is the responsibility of application author, but it could be the responsibility of the code transform.

The initReflector() method is generated by the code transform and contains all of the stubs which are needed for the current library.

## Lazy Loading with Router

Here is the most common example of lazy loading with a router. Note that the code transformer is unaware of the router semantics, it is only interested in the loadLibrary() method call.

app.dart:

|  |
| --- |
| import 'package:library\_a.dart' deferred as lib\_a;  import 'package:library\_b.dart' deferred as lib\_b;  main() {  bootstrap(App); // Code Transform detects as root component  }  @Component({  selector: 'app',  componentServices: [  // Code transform follows this link  bind(RouterConfiguration).toClass(AppRouterConfiguration)  ]  })  class App {}  // Code transform analyzes this class.  class AppRouterConfiguration() {  AppRouterConfiguration(Router router) {  router.config([  {'path': '/routeA', component: () {  return lib\_a.loadLibrary().then(() {  lib\_a.initReflector();  // Code Transform detects ComponentA  // This is recognized as lazy and we start  // from this component as root.  return lib\_a.ComponentA;  });  }}  {'path': '/routeB', component: () {  return lib\_b.loadLibrary().then(() {  lib\_b.initReflector();  // Code Transform detects as root component  return lib\_b.ComponentA;  }}  ]);  }  } |

library\_a.dart:

|  |
| --- |
| initReflector() {  // This function is left blank by the author.  // The function content is filled in by the code transformer.  // It will look something like this:  reflector.registerType(ComponentA, {  "factory": () => new ComponentA(),  "parameters": [],  "annotations" : [  new Component(...),  new Template(...)]  });  }  @Component(...)  @Template(...)  class ComponentA {  } |

# 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.