https://clipcodeapp.appspot.com

Clipcode Source Tour For Angular

A detailed guided tour to the source trees of Angular and related projects

Written by Eamon O'Tuathail

DRAFT

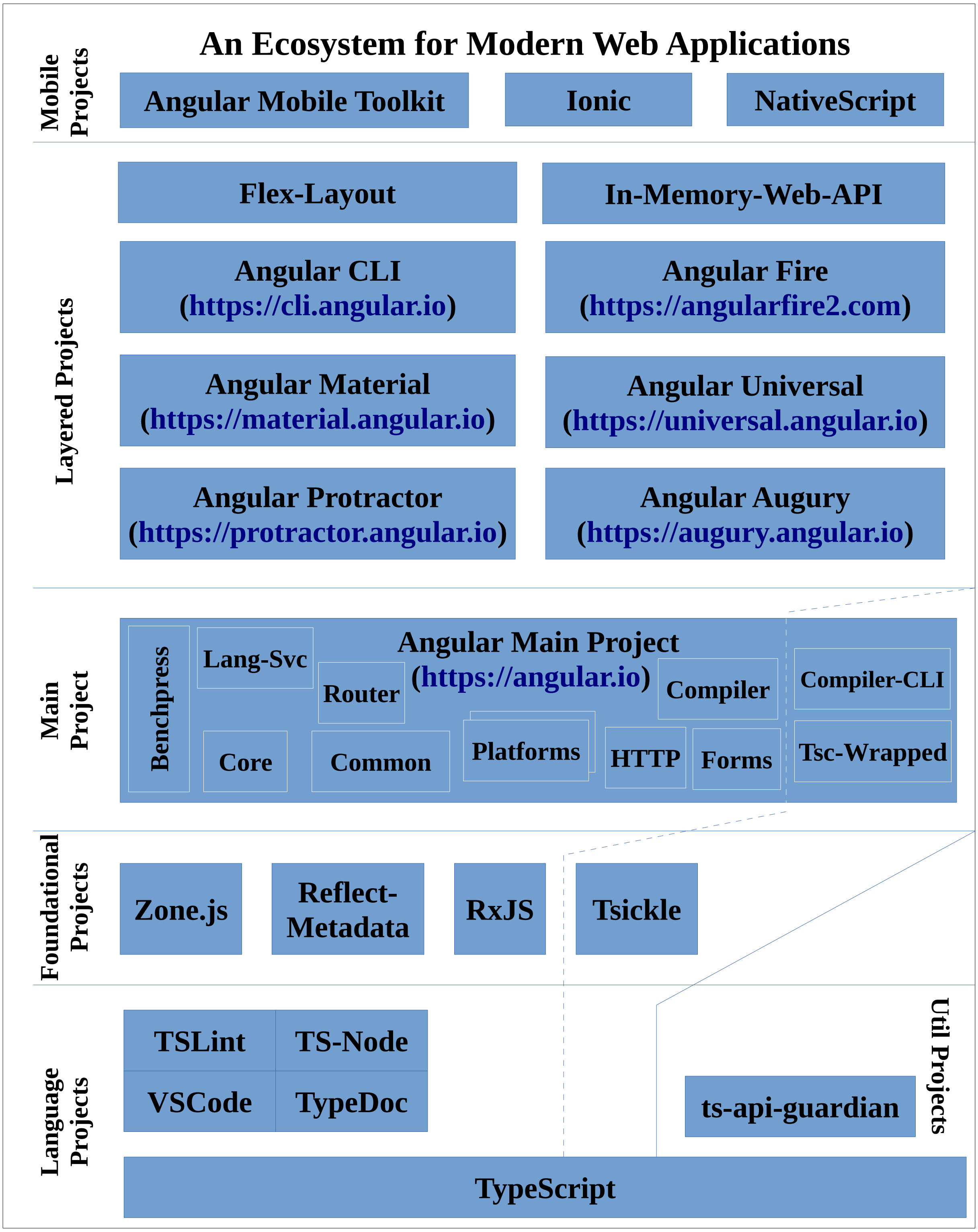
Last Updated: Wed 8 February 2017

Table of Contents

Preface

An Ecosystem for Modern Web Applications

The Angular ecosystem consists of multiple projects that work together to provide a comprehensive foundation for developing modern web applications.



All these projects are dependent on the TypeScript package (even the TypeScript compiler is itself written in TypeScript). The dependency on other language projects varies. None of the projects are dependent on Visual Studio Code. These projects only need a code editor that can work with the TypeScript tsc compiler – and there are many (see middle of main page at http://typescriptlang.org). Visual Studio Code is one such editor, that is particularly good, open source, freely available for macOS, Linux and Windows, and it too is written in TypeScript. But you may choose to use any code editor to investigate the source trees of these projects.

Viewing Markdown Files

In addition to source in TypeScript, all these projects also contain documentation files in markdown format (*.md). Markdown is a domain specific language (DSL) for represent HTML documents. It is easier / quicker to manually author compared to HTML. When transformed to HTML it provides professional documentation. One question you might have is how to view them (as HTML, rather that as .md text). One easy way to see them as html is just to view the .md files on Github, e.g.:

- https://github.com/angular/angular

Alternatively, on your own machine, most text editors have either direct markdown support or it is available through extensions. When examining a source tree with .md files, it is often useful when your code editor can also open markdown files and transform them to HTML when requested. StackOverflow covers this issue here:

- http://stackoverflow.com/questions/9843609/view-markdown-files-offline

For example, Visual Studio Code supports Markdown natively and if you open a .md file and select CTRL+SHIFT+V, you get to see nice HTML:

- https://code.visualstudio.com/docs/languages/markdown

Finally, if you want to learn markdown, try here:

- http://www.markdowntutorial.com

Benefits of Understanding The Source

There are many good reasons for intermediate- to advanced-developers to become familiar with the source trees of the projects that provide the foundation for their daily application development.

Enhanced Understanding - As in common with many software packages, descriptions of software concepts and API documentation may or may not be present, and if present may not be as comprehensive as required, and what is present may or may not accurately reflect what the code currently does - the doc for a particular concept may well once have been up-to-date, but code changes, the doc not necessarily so, and certainly not in lock-step, and this applies to any fast evolving framework.

Advanced Debugging – When things go wrong, as they must assuredly will (and usually just become an important presentation to potential customers), application developers scrambling to fix problems will be much quicker when they have better insight via knowing the source.

Optimization – for large-scale production applications with high data throughput, knowing how your application’s substrate actually works can be really useful in deciding how / what to optimize (for example, when application developers really understand how Angular's `NgZone` works and is intertwined with Angular's change detection, then they can place CPU-intensive but non-UI code in a different zone within the same thread, and this can result in much better performance).

Productivity - Familiarity with the source is important for maximum productivity with any framework and is one part of a developer accumulating a broader understanding of the substrate upon which their applications execute.

Good practices – Studying large codebases that are well tested and subjected to detailed code reviews is a great way for "regular" developers to pick up good coding habits and use in their own application source trees.

Target Audience

This guided tour of the Angular source tree is aimed at intermediate to advanced application developers who wish to develop a deeper understanding of how Angular actually works. To really appreciate the functioning of Angular, an application developer should read the source. Yes, Angular come with developer documentation at https://angular.io/docs/ts/latest (which is mostly quite good) but it is best studied in conjunction with the source code.

Accessing the Source

To study the source, you can browse it online, get a copy of the repo via git (usual) or download a zip. Some packages may provide extra detail about getting the source – for example, for the Angular project, read "Getting the Sources" here:

- https://github.com/angular/angular/blob/master/DEVELOPER.md

We first need to decide which branch to use. For master, we use this:

- https://github.com/angular/angular/tree/master

Specifically for the Angular main project, an additional way to access the source is in the Angular API Reference (https://angular.io/docs/ts/latest/api), the API page for each Angular type has a hyperlink at the bottom of the page to the relevant source file (this resolves to the latest stable version, which may or may not be the same source as master).

1: Zone.js

Overview

The Zone.js project provides multiple asynchronous execution contexts running within a single JavaScript thread (i.e. within the main browser UI thread or within a single webworker). Zones are a way of sub-dividing a single thread using the JavaScript event loop (a single zone does not cross thread boundaries). A nice way to think about zones is that they sub-divide the stack within a JavaScript thread into multiple mini-stacks, and sub-divide the JavaScript event loop into multiple mini event loops.

When your app loads Zone.js, it monkey-patches certain asynchronous calls (e.g. `setTimer`, `addEventListener`), to implement zone functionality. Zone.js adds wrappers to the callbacks the application supplies, and when a timeout occurs or an event is detected, it runs the wrapper first, and then the application callback. Chunks of executing application code form tasks and each task executes in the context of a zone.

Zones are arranged in a hierarchy and provide useful features in areas such as error handling, performance measuring and executing configured work items upon entering and leaving a zone (all of which might be of great interest to implementors of change detection in a modern web framework!).

Zone.js is mostly transparent to application code. Zone.js runs in the background and for the most part "just works". Application code can make zone calls if needed and become more actively involved in zone management. Angular uses Zone.js and Angular application code usually runs inside a zone (although advanced application developers can take certain steps to move some code outside of the Angular zone – using the `NgZone` class).

Using external libraries can occasionally cause problems with zones – so if you detect strange errors do verify it is not a zone-related problem. A specific example is with Stripe payments API (a regular JavaScript library) and Angular – the problem and the simple solution are explained here:

- http://stackoverflow.com/questions/36258252/stripe-json-circular-reference

Project Information

The homepage and root of the source tree for Zone.js is at:

- https://github.com/angular/zone.js

Below we assume you have got the Zone.js source tree downloaded locally under a directory we will call <ZONE> and any pathnames we use will be relative to that.

Zone.js is written in TypeScript. It has no package dependencies (its package.json has this entry: `"dependencies": {}`), though it has many `devDependencies`. It is quite a small source tree, whose size (uncompressed) is less that 700KB.

This primer document will be of interest to developers learning Zone.js:

- https://docs.google.com/document/d/1F5Ug0jcrm031vhSMJEOgp1l-Is-Vf0UCNDY-LsQtAIY/edit

Loading Zone.js

To use Zone.js in your applications, you need to load it. Your package.json file will need:

```
"dependencies": {
    ..
    "zone.js": "<version>"
  },
```

You should load Zone.js just after loading core.js (if using that), and before everything else. For example, in an Angular application based on the QuickStart layout, your index.html file will contain:

```
<script src="node_modules/core-js/client/shim.min.js"></script>
<script src="node_modules/zone.js/dist/zone.js"></script>
<script src="node_modules/reflect-metadata/Reflect.js"></script>
<script src="node_modules/systemjs/dist/system.src.js"></script>
```

If using an Angular application generated via Angular CLI (as most production apps will be), Angular CLI will generate a file called <project-name>/src/polyfills.ts and it will contain:

```
// This file includes polyfills needed by Angular and is loaded before
// the app. You can add your own extra polyfills to this file.
import 'core-js/es6/symbol';
..
import 'zone.js/dist/zone';
```

Angular CLI also generates a main.ts file, and its first line is:

```
import './polyfills.ts';
```

If writing your application in TypeScript (recommended), you also need to get access to the ambient declarations. These define the Zone.js API and are supplied in:

- <ZONE>/dist/zone.js.d.ts

(This file is particularly well documented and well worth some careful study by those learning Zone.js). Unlike declarations for most other libraries, zone.js.d.ts does not use `import` or `export` at all (those constructs do not appear even once in that file). That means application code wishing to use zones cannot simply import its .d.ts file, as is normally the case. Instead, the `///reference` construct needs to be used. This includes the referenced file at the site of the `///reference` in the containing file. The benefit of this approach is that the containing file itself does not have to (but may) use `import`, and thus may be a script, rather than having to be a module. The use of zones is not forcing the application to use modules (however, most larger applications, including all Angular applications - will). How this works is best examined with an example, so lets look at how Angular includes zone.d.ts. Angular contains a file, types.d.ts under its modules directory:

- <ANGULAR-MASTER>/modules/types.d.ts

and it has the following contents:

```
// This file contains all ambient imports needed to compile the modules/
source code
```

```
/// <reference path="../node_modules/zone.js/dist/zone.js.d.ts" />
/// <reference path="../node_modules/@types/hammerjs/index.d.ts" />
/// <reference path="../node_modules/@types/jasmine/index.d.ts" />
/// <reference path="../node_modules/@types/node/index.d.ts" />
/// <reference path="../node_modules/@types/selenium-webdriver/index.d.ts" />
/// <reference path="./es6-subset.d.ts" />
/// <reference path="./system.d.ts" />
```

A similar file is supplied under <ANGULAR-MASTER>/tools for tooling though it has fewer entries:

```
// This file contains all ambient imports needed to compile the tools source
code

/// <reference path="../node_modules/@types/jasmine/index.d.ts" />
/// <reference path="../node_modules/@types/node/index.d.ts" />
/// <reference path="../node_modules/zone.js/dist/zone.js.d.ts" />
```

When building each Angular component, the tsconfig.json file, located in:

- <ANGULAR-MASTER>/modules/@angular/<package>

contains:

```
  "files": [
    "index.ts",
    "../../../node_modules/zone.js/dist/zone.js.d.ts",
    "../../system.d.ts"
  ],
```

Use Within Angular

When writing Angular applications, all your application code runs within a zone, unless you take specific steps to ensure some of it does not. Also, most of the Angular framework code itself runs in a zone. When beginning Angular application development, you can get by simply ignoring zones, since they are set up correctly by default for you and applications do not have to do anything in particular to take advantage of them.

Zones are how Angular initiates change detection – when the zone's mini-stack is empty, change detection occurs. Also, zones are how Angular configures global exception handlers. When an error occurs in a task, its zone's configured error handler is called. A default implementation is provided and applications can supply a custom implementation via dependency injection. For details, see here:

- https://angular.io/docs/ts/latest/api/core/index/ErrorHandler-class.html

On that page note the code sample about setting up your own error handler:

```
class MyErrorHandler implements ErrorHandler {
  handleError(error) {
    // do something with the exception
  }
}
@NgModule({
  providers: [{provide: ErrorHandler, useClass: MyErrorHandler}]
})
class MyModule {}
```

Angular provide a class, NgZone, which builds on zones:

- https://angular.io/docs/ts/latest/api/core/index/NgZone-class.html

As you begin to create more advanced Angular applications, specifically those involving computationally intensive code that does not change the UI midway through the computation (but may at the end), you will see it is desirable to place such CPU-intensive work in a separate zone, and you would use a custom NgZone for that.

We will be looking in detail at NgZone and the use of zones within Angular in general when we explore the source tree for the main Angular project later, but for now, note the source for NgZone is in:

- <ANGULAR-MASTER>/modules/@angular/core/src/zone

and the zone setup during bootstrap for an application is in:

- <ANGULAR-MASTER>/modules/@angular/core/src/application_ref.ts

When we bootstrap our Angular applications, we either use bootstrapModule<M> (using the dynamic compiler) or bootstrapModuleFactory<M> (using the offline compiler) and both these ultimately result in a call to _bootstrapModuleFactoryWithZone<M> (all these functions are in application_ref.ts):

```
bootstrapModuleFactory<M>(
  moduleFactory: NgModuleFactory<M>): Promise<NgModuleRef<M>> {
    return this._bootstrapModuleFactoryWithZone(moduleFactory, null);
  }

private _bootstrapModuleFactoryWithZone<M>(
 moduleFactory: NgModuleFactory<M>, ngZone: NgZone): Promise<NgModuleRef<M>>{
    // Note: We need to create the NgZone _before_ we instantiate the module,
    // as instantiating the module creates some providers eagerly.
    // So we create a mini parent injector that just contains the new NgZone
    // and pass that as parent to the NgModuleFactory.
1   if (!ngZone) ngZone = new NgZone({enableLongStackTrace: isDevMode()});
    // Attention: Don't use ApplicationRef.run here,
    // as we want to be sure that all possible constructor calls are
    // inside `ngZone.run`!
    return ngZone.run(() => {
      const ngZoneInjector =
          ReflectiveInjector.resolveAndCreate([{provide: NgZone,
                                         useValue: ngZone}], this.injector);
      const moduleRef =
               <NgModuleInjector<M>>moduleFactory.create(ngZoneInjector);
2     const exceptionHandler: ErrorHandler =
                 moduleRef.injector.get(ErrorHandler, null);
      if (!exceptionHandler) {
        throw new Error(
          'No ErrorHandler. Is platform module (BrowserModule) included?');
      }
      moduleRef.onDestroy(() =>
                 ListWrapper.remove(this._modules, moduleRef));
3     ngZone.onError.subscribe(
        {next: (error: any) => { exceptionHandler.handleError(error); }});
      return _callAndReportToErrorHandler(exceptionHandler, () => {
        const initStatus: ApplicationInitStatus =
```

```
            moduleRef.injector.get(ApplicationInitStatus);
        return initStatus.donePromise.then(() => {
4         this._moduleDoBootstrap(moduleRef);
          return moduleRef;
        });
      });
```

At 1 we see a new `NgZone` being created and its `run()` method being called, at 2 we see an error handler implementation being requested from dependency injection (a default implementation will be returned unless the application supplies a custom one) and at 3, we see that error handler being used to configure error handling for the newly created `NgZone`. Finally at 4, we see the call to the actual bootstrapping.

So in summary, as Angular application developers, we should clearly learn about zones, since that is the execution context within which our application code will run.

Zone.js API

Zone.js exposes an API for applications to use in the <ZONE>/dist/zone.js.d.ts file:

The two main types it offers are for tasks and zones, along with some helper types. A zone is a (usually named) asynchronous execution context; a task is a block of functionality (may also be named). Tasks run in the context of a zone. Zone.js also supplies a const value, also called `Zone`, of type `ZoneType`:

```
interface ZoneType {
    current: Zone;
    currentTask: Task; }
declare const Zone: ZoneType;
```

Recall that TypeScript has distinct declaration spaces for values and types, so the `Zone` value is distinct from the `Zone` type. For further details, see the TypeScript Language Specification – Section 2.3 – Declarations:

- https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md#2.3

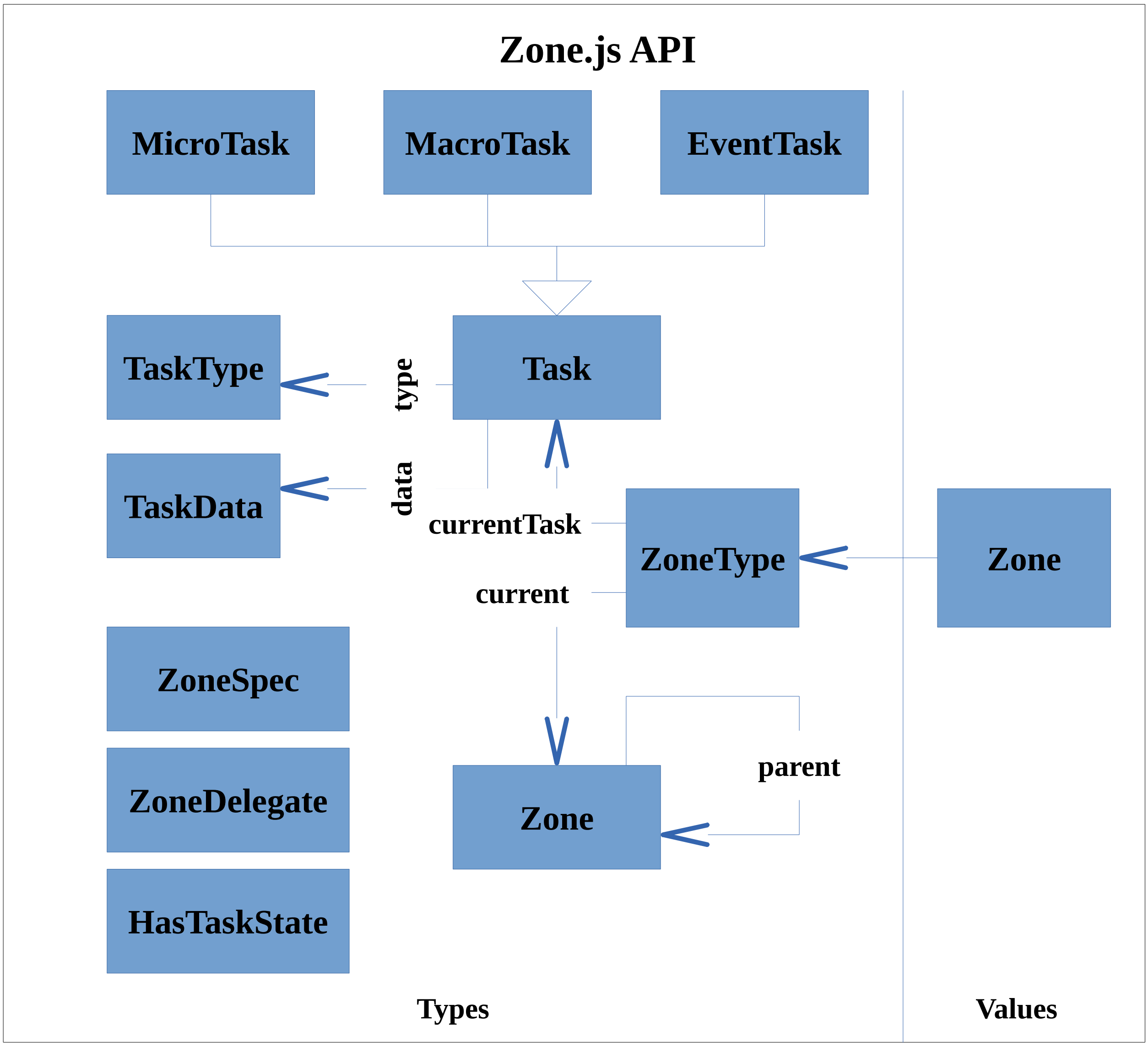
Apart from being used to define the `Zone` value, `ZoneType` is not used further.

When your application code wishes to find out the current zone it simply uses `Zone.current`, and when it wants to discover the current task within that zone, it uses `Zone.currentTask`. If you need to figure out whether Zone.js is available to your application (it will be for Angular applications), then just make sure `Zone` is not undefined. If we examine:

- <ANGULAR-MASTER>/modules/@angular/core/src/zone/ng_zone.ts

– we see that is exactly what Angular's NgZone.ts does:

```
constructor({enableLongStackTrace = false}) {
    if (typeof Zone == 'undefined') {
      throw new Error('Angular requires Zone.js polyfill.');
    }
```



Two simple helper types used to define tasks are `TaskType` and `TaskData`. `TaskType` is just a human-friendly string to associate with a task. It is usually set to one of the three task types as noted in the comment:

```
// Task type: `microTask`, `macroTask`, `eventTask`.
declare type TaskType = string;
```

`TaskData` contains a boolean (is this task periodic, i.e. is to be repeated) and two numbers - delay before executing this task and a handler id from `setTimout`.

```
interface TaskData {
    /**
    * A periodic [MacroTask] is such which get
    * automatically rescheduled after it is executed.
    */
    isPeriodic?: boolean;
    /**
     * Delay in milliseconds when the Task will run.
     */
```

```
        delay?: number;
        /**
         * identifier returned by the native setTimeout.
         */
        handleId?: number;
    }
```

A task is an interface declared as:

```
    interface Task {
        type: TaskType;
        source: string; // for debugging: wh scheduled this task
        invoke: Function; // VM calls this when it enters a task
        callback: Function; / tasks calls this when Zone.currentTask has been set
        data: TaskData; // data passed to scheduleFn
        scheduleFn: (task: Task) => void; // the scheduling
        cancelFn: (task: Task) => void; // to un-schedule a task
        zone: Zone; // this task is in this zone
        runCount: number; // how often called (simple tracking information)
    }
```

There are three empty marker interfaces derived from `Task`:

```
    interface MicroTask extends Task { }
    interface MacroTask extends Task { }
    interface EventTask extends Task { }
```

There are three helper types used to define `Zone`. `HasTaskState` just contains booleans for each of the task types and a string:

```
    declare type HasTaskState = {
        microTask: boolean;
        macroTask: boolean;
        eventTask: boolean;
        change: TaskType;
    };
```

`ZoneDelegate` is used when one zone wishes to delegate to another how certain operations should be performed. So for forcking (creating new tasks), scheduling, intercepting, invoking and error handling, the delegate may be called upon to carry out the action.

```
    interface ZoneDelegate {
        zone: Zone;
        fork(targetZone: Zone, zoneSpec: ZoneSpec): Zone;
        intercept(targetZone: Zone, callback: Function, source: string):
                                                                Function;
        invoke(targetZone: Zone, callback: Function, applyThis: any,
                                    applyArgs: any[], source: string): any;
        handleError(targetZone: Zone, error: any): boolean;
        scheduleTask(targetZone: Zone, task: Task): Task;
        invokeTask(targetZone: Zone, task: Task, applyThis: any,
                                                  applyArgs: any): any;
        cancelTask(targetZone: Zone, task: Task): any;
        hasTask(targetZone: Zone, isEmpty: HasTaskState): void;
    }
```

`ZoneSpec` is an interface that allows implementations to state what should have when certain actions are needed. It uses `ZoneDelegate` and the current zone:

```
interface ZoneSpec {
    name: string;
    properties?: { [key: string]: any; };
    onFork?: (parentZoneDelegate: ZoneDelegate, currentZone: Zone,
              targetZone: Zone, zoneSpec: ZoneSpec) => Zone;
    onIntercept?: (parentZoneDelegate: ZoneDelegate, currentZone: Zone,
           targetZone: Zone, delegate: Function, source: string) => Function;
    onInvoke?: (parentZoneDelegate: ZoneDelegate, currentZone: Zone,
         targetZone: Zone, delegate: Function, applyThis: any,
         applyArgs: any[], source: string) => any;
    onHandleError?: (parentZoneDelegate: ZoneDelegate, currentZone: Zone,
         targetZone: Zone, error: any) => boolean;
    onScheduleTask?: (parentZoneDelegate: ZoneDelegate, currentZone: Zone,
         targetZone: Zone, task: Task) => Task;
    onInvokeTask?: (parentZoneDelegate: ZoneDelegate, currentZone: Zone,
         targetZone: Zone, task: Task, applyThis: any, applyArgs: any)=>any;
    onCancelTask?: (parentZoneDelegate: ZoneDelegate, currentZone: Zone,
         targetZone: Zone, task: Task) => any;
    onHasTask?: (delegate: ZoneDelegate, current: Zone, target: Zone,
         hasTaskState: HasTaskState) => void;
}
```

The definition of the `Zone` type is:

```
interface Zone {
    parent: Zone;
    name: string;
    get(key: string): any;
    getZoneWith(key: string): Zone;
    fork(zoneSpec: ZoneSpec): Zone;
    wrap(callback: Function, source: string): Function;
    run<T>(callback: Function, applyThis?: any,
              applyArgs?: any[], source?: string): T;
    runGuarded<T>(callback: Function, applyThis?: any,
              applyArgs?: any[], source?: string): T;
    runTask(task: Task, applyThis?: any, applyArgs?: any): any;
    scheduleMicroTask(source: string, callback: Function, data?: TaskData,
              customSchedule?: (task: Task) => void): MicroTask;
    scheduleMacroTask(source: string, callback: Function, data: TaskData,
              customSchedule: (task: Task) => void,
              customCancel: (task: Task) => void): MacroTask;
    scheduleEventTask(source: string, callback: Function, data: TaskData,
              customSchedule: (task: Task) => void,
              customCancel: (task: Task) => void): EventTask;
    cancelTask(task: Task): any;
}
```

Relationship between Zone/ZoneSpec/ZoneDelegate interfaces

Think of `ZoneSpec` as the processing engine that controls how a zone works. It is a required parameter to the `Zone.fork()` method:

```
// Used to create a child zone.
// @param zoneSpec A set of rules which the child zone should follow.
// @returns {Zone} A new child zone.
```

```
fork(zoneSpec: ZoneSpec): Zone;
```

Often when a zone needs to perform an action, it uses the supplied `ZoneSpec`. Do you want to record a long stack trace, keep track of tasks, work with WTF (discussed later) or run async test well? For each a these a different `ZoneSpec` is supplied, and each offers different features and comes with different processing costs. Zone.js supplies one implementation of the `Zone` interface, and multiple implementations of the `ZoneSpec` interface (in <ZONE>/lib/zone-spec). Application code with specialist needs could create a custom ZoneSpec.

An application can build up a hierarchy of zones and sometimes a zone needs to make a call into another zone further up the hierarchy, and for this a `ZoneDelegate` is used.

Source Tree Layout

The Zone.js source tree consists of a root directory with a number of files and the following immediate sub-directories:

- dist
- example
- lib
- scripts
- tests

The main source is in lib.

During compilation the source gets built into a newly created build directory.

Root directory

The root directory contains these markdown documentation files:

- DEVELOPER.md
- README.md

DEVELOPER.md is a short document contains:

```
Submitting Changes
------------------
Do NOT submit changes to the built files in the `dist` folder. These are
generated before releases.
To run tests
------------
Make sure your environment is set up with: `npm install`
In a separate process, run the WebSockets server: `npm run ws-server`
Run the browser tests using Karma: `npm test`
Run the node.js tests: `npm run test-node`
```

There were significant changes to the Zone.js API between v.0.5 and v.0.6, and much of README.md describes the old v0.5 with the old API, so perhaps is best ignored. It contains one important comment:

```
# NEW Zone.js POST-v0.6.0
See the new API [here](./dist/zone.js.d.ts).
```

When we examine <ZONE>/dist/zone.js.d.ts we see it is actually very well documented and contains plenty of detail to get us up and running writing applications that use Zone.js. From the DEVELOPER.md document we see the contents of dist is auto-generated (we need to explore how).

The root directory contains these JSON files:

- tsconfig.json
- typings.json
- package.json

tsconfig.json is:

```
"compilerOptions": {
    "module": "commonjs",
    "target": "es5",
    "noImplicitAny": false,
    "outDir": "build",
    "inlineSourceMap": false,
    "inlineSources": false,
    "declaration": true,
    "noEmitOnError": false,
    "stripInternal": true
  },
  "exclude": [
    "node_modules",
    "typings/main",
    "typings/main.d.ts",
    "build",
    "dist"
  ]
```

The typings.json file contains ambient dependencies for Jasmine (unit testing), es6-promise and node.

The package.json file contains metadata (including main and browser, which provide alternative entry points depending on whether this package 1 is loaded into a server [node] or a 2 browser app):

```
{
  "name": "zone.js",
  "version": "0.6.17",
  "description": "Zones for JavaScript",
1 "main": "dist/zone-node.js",
2 "browser": "dist/zone.js",
  "typings": "dist/zone.js.d.ts",
```

and a list of scripts:

```
"scripts": {
  "prepublish": "./node_modules/.bin/typings install &&
                              ./node_modules/.bin/tsc && gulp build",
  "typings": "./node_modules/.bin/typings install",
  "ws-server": "node ./test/ws-server.js",
  "tsc": "./node_modules/.bin/tsc",
  "tsc:w": "./node_modules/.bin/tsc -w",
  "test": "karma start karma.conf.js",
  "test-node": "./node_modules/.bin/gulp test/node",
  "serve": "python -m SimpleHTTPServer 8000"
},
```

it has no dependencies:

```
"dependencies": {},
```

It has many devDependencies, including those related to gulp (task runner), jasmine (unit testing) and karma (unit test runner), and then these:

```
  "devDependencies": {
    ...
    "es6-promise": "^3.0.2",
    "nodejs-websocket": "^1.2.0",
    "ts-loader": "^0.6.0",
    "typescript": "^1.8.0",
    "typings": "^0.7.12",
    "webpack": "^1.12.2"
  }
}
```

Webpack is quite a popular bundler and ts-loader is a TypeScript loader for webpack. Details on both projects can be found here:

```
https://webpack.github.io/
https://github.com/TypeStrong/ts-loader
```

The root directory also contains the MIT license in a file called LICENSE, along with the same within a comment in a file called LICENSE.wrapped.

It contains the following files concerning unit testing and continuous integration testing:

- .travis.yml
- karma.conf.js
- karma-sauce.conf.js
- sause.conf.js

It contains this file concerning bundling:

- webpack.config.js

This has the following content:

```
module.exports = {
  entry: './test/source_map_test.ts',
  output: {
    path: __dirname + '/build',
    filename: 'source_map_test_webpack.js'
  },
  devtool: 'inline-source-map',
  module: {
    loaders: [
      {test: /\.ts/, loaders: ['ts-loader'], exclude: /node_modules/}
    ]
  },
  resolve: {
    extensions: ["", ".js", ".ts"]
  }
}
```

the root directory contains this file related to GIT:

- .gitignore

It contains this task runner configuration:

- gulpfile.js

It supplies a gulp task called “test/node” to run tests against the node version of Zone.js, and a gulp task “compile” which runs the TypeScript tsc compiler in a child process. It supplies many gulp tasks to build individual components:

[1] The ambient declarations

```
 'build/zone.js.d.ts',
```

[2] environments for browser, server and jasmine

```
'build/zone.js',
'build/zone.min.js',
'build/zone-node.js',
'build/jasmine-patch.js',
'build/jasmine-patch.min.js',
```

[3] different ZoneSpecs (so they can be loaded individually, whichever (if any) are required):

```
'build/long-stack-trace-zone.js',
'build/long-stack-trace-zone.min.js',
'build/proxy-zone.js',
'build/proxy-zone.min.js',
'build/task-tracking.js',
'build/task-tracking.min.js',
'build/wtf.js',
'build/wtf.min.js',
'build/async-test.js',
'build/fake-async-test.js',
'build/sync-test.js'
```

[4] and a combined gulp task to build all of them.

All of these tasks result in a call to a local method `generateBrowserScript` which minifies (if required) and calls webpack and places the result in the dist sub-directory.

dist

This single directory contains all the output from the build tasks. The zone.d.ts file is the ambient declarations, which TypeScript application developers will want to use. This is surprisingly well documented, so a new application developer getting up to speed with zone.js should give it a careful read.

Three implementations of Zone are provided, for the browser, for the server and for Jasmine testing:

- zone.js / zone.min.js
- zone-node.js
- jasmine-patch.js / jasmine-patch.min.js

Minified versions are supplied for the browser and jasmine builds, but not node. If you are using Angular in the web browser, then zone.js (or zone.min.js) is all you need.

The remaining files in the dist directory are builds of different zone specs, which for specialist reasons you may wish to include. These are:

- async-test.js
- fake-async-test.js
- long-stack-trace-zone.js / long-stack-trace-zone.min.js
- proxy.js / proxy.min.js
- task-tracking.js / task-tracking.min.js
- wtf.js / wtf.min.js

We will look in detail at what each of these does later when examining the <ZONE>/lib/zone-spec source directory.

example

The example directory contains a range of simple examples showing how to use Zone.js.

scripts

The script directory contains three scripts:

- grab-blink-idl.sh
- sauce/sauce_connect_setup.sh
- sauce/sauce_connect_block.sh

Source

The main source for zone.js is in:

- <ZONE>/lib

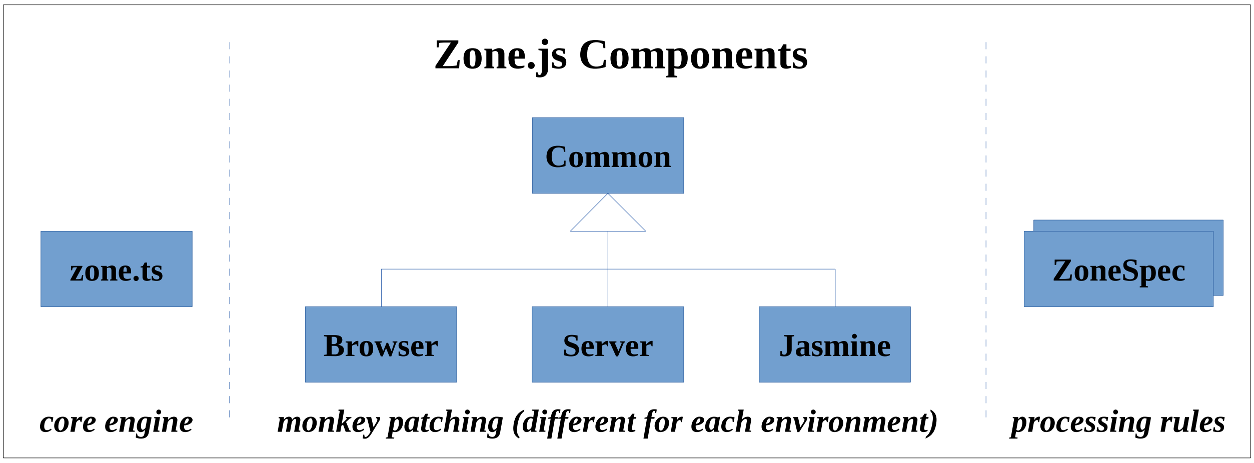
It contains five directories:

- common
- browser
- server
- jasmine
- zone-spec

along with one source file:

- zone.ts

It is best to think of them arranged as follows:



To enable Zone.js to function, any JavaScript APIs related to asynchronous code execution must be patched – a Zone.js specific implementation of the API is needed. So for calls such as `setTimeout` or `addEventListener` and similar, Zone.js needs its own handling, so that when timeouts and events and promises get triggered, zone code runs first.

There are the three environments supported by Zone.js that need monkey patching – the browser, the server (node) and Jasmine and each of these has a sub-direcory with patching code. The common patching code reside in the common directory. The core implementation of the Zone.js API (excluding `ZoneSpec`) is in zone.ts file. The additional directory is for zone specs, which are the configurable logic one can add to a zone to change its behavior. There are multiple implementations of these, and applications could create their own.

zone.ts

The first five hundred lines of the zone.ts file is the well-commented definition of the Zone.js API, that will end up in zone.d.ts. The slightly larger remainder of the file is an implementation of the `Zone` const:

```
const Zone: ZoneType = (function(global: any) {
...
  return global.Zone = Zone;
}) (typeof window === 'undefined' ? global : window);
```

Internally, it implements a `ZoneAwarePromise` class and swaps out the global promise for it.

```
class ZoneAwarePromise<R> implements Promise<R> { .. }
global.Promise = ZoneAwarePromise;
```

It defines these variables:

```
  let _currentZone: Zone = new Zone(null, null);
  let _currentTask: Task = null;
  let _microTaskQueue: Task[] = [];
  let _isDrainingMicrotaskQueue: boolean = false;
  let _numberOfNestedTaskFrames = 0;
```

_microTaskQueue is an array of microtasks, that must be executed before we give up our VM turn. _isDrainingMicrotaskQueue is a boolean that tracks if we are in the process of emptying the microtask queue. When a task is run within an existing task, they are nested and _numberOfNestedTaskFrames, is incremented, and when finised executing, decremented. Draining of the microtask queue only occurs when we are in the root task.

It also implements three classes:

- Zone
- ZoneDelegate
- ZoneTask

There are no implementations of ZoneSpec in this file. They are in the separate zone-spec sub-directory.

ZoneTask is the simplest of these classes:

```
class ZoneTask implements Task {
    public type: TaskType;
    public source: string;
    public invoke: Function;
    public callback: Function;
    public data: TaskData;
    public scheduleFn: (task: Task) => void;
    public cancelFn: (task: Task) => void;
    public zone: Zone;
    public runCount: number = 0;
```

The constructor just records the supplied parameters and sets up invoke:

```
    constructor(type: TaskType, zone: Zone, source: string,
        callback: Function, options: TaskData,
        scheduleFn: (task: Task) => void, cancelFn:(task: Task) => void){
      this.type = type;
      this.zone = zone;
      this.source = source;
      this.data = options;
      this.scheduleFn = scheduleFn;
      this.cancelFn = cancelFn;
      this.callback = callback;
      const self = this;
```

The interesting activity in here is setting up the invoke function. It increments the _numberOfNestedTaskFrames counter, calls zone.runTask(), and in a finally block, checks if _numberOfNestedTaskFrames is 1, and if so, calls the standalone function drainMicroTaskQueue(), and then decrements _numberOfNestedTaskFrames.

```
      this.invoke = function () {
        _numberOfNestedTaskFrames++;
        try {
          return zone.runTask(self, this, <any>arguments);
        } finally {
          if (_numberOfNestedTaskFrames == 1) {
            drainMicroTaskQueue();
          }
```

```
          _numberOfNestedTaskFrames--;
        }
      };
    }
```

A custom toString() implementation returns data.handleId (if available) or else the object's toString() result:

```
    public toString() {
      if (this.data && typeof this.data.handleId !== 'undefined') {
        return this.data.handleId;
      } else {
        return this.toString();
      }
    }
```

If we exclude error handling, drainMicroTaskQueue() is defined as:

```
  function drainMicroTaskQueue() {
      if (!_isDrainingMicrotaskQueue) {
        _isDrainingMicrotaskQueue = true;
        while(_microTaskQueue.length) {
          const queue = _microTaskQueue;
          _microTaskQueue = [];
          for (let i = 0; i < queue.length; i++) {
            const task = queue[i];
            task.zone.runTask(task, null, null);
          }
        }
        _isDrainingMicrotaskQueue = false;
      }
    }
```

The _microTaskQueue gets populated via a call to scheduleMicroTask:

```
  function scheduleMicroTask(task: MicroTask) {
      scheduleQueueDrain();
      _microTaskQueue.push(task);
    }
```

The scheduleQueueDrain() function calls setTimeout with timeout set to 0, to enqueue a request to drain the microtask queue. Even though the timeout is 0, this does not mean that the drainMicroTaskQueue() call will execute immediately. Instead, this puts an event in the JavaScript's event queue, which after the already scheduled events have been handled (there may be one or move already in the queue), will itself be handled. The currently executing function will first run to completion before any event is removed from the event queue. Hence in the above code, where scheduleQueueDrain() is called before _microTaskQueue.push(), is not a problem. _microTaskQueue.push() will execute first, and then sometime in future, the drainMicroTaskQueue() function will be called via the timeout.

```
  function scheduleQueueDrain() {
  // if we are not running in any task, and there has not been anything
  // scheduled we must bootstrap the initial task creation by manually
  // scheduling the drain
      if (_numberOfNestedTaskFrames == 0 && _microTaskQueue.length == 0) {
        // We are not running in Task, so we need to
```

```
        // kickstart the microtask queue.
        if (global[symbolPromise]) {
          global[symbolPromise].resolve(0)[symbolThen](drainMicroTaskQueue);
        } else {
          global[symbolSetTimeout](drainMicroTaskQueue, 0);
        }
      }
    }
```

The ZoneDelegate class has to handle eight scenarios:

- fork
- intercept
- invoke
- handleError
- scheduleTask
- invokeTask
- cancelTask
- hasTask

It defines variables to store values for a ZoneDelegate and ZoneSpec for each of these, which are initialized in the constructor. ZoneDelegate also declares three variables, to store the delegates zone and parent delegate, and to represent task counts (for each kind of task):

```
  public zone: Zone;
  private _taskCounts:
     {microTask: number, macroTask: number, eventTask: number}
      = {microTask: 0, macroTask: 0, eventTask: 0};
  private _parentDelegate: ZoneDelegate;
```

In ZoneDelegate's constructor, the zone and parentDelegate fields are initialized to the supplied parameters, and the ZoneDelegate and ZoneSpec fields for the eight scenarios are set (using TypeScript type guards), either to the supplied ZoneSpec (if not null), or the parent delegate's:

```
  constructor(zone: Zone, parentDelegate: ZoneDelegate, zoneSpec:ZoneSpec) {
        this.zone = zone;
        this._parentDelegate = parentDelegate;
…
       this._scheduleTaskZS =
            zoneSpec && (zoneSpec.onScheduleTask ?
                          zoneSpec : parentDelegate._scheduleTaskZS);
        this._scheduleTaskDlgt =
            zoneSpec && (zoneSpec.onScheduleTask ?
                          parentDelegate : parentDelegate._scheduleTaskDlgt);
```

The ZoneDelegate methods for the eight scenarios just forward the calls to the selected ZoneSpec (pr parent delegate) and does some house keeping. For example, the invoke method checks if _invokeZS is defined, and if so, calls its onInvoke, otherwise it calls the supplied callback directly:

```
    invoke(targetZone: Zone, callback: Function, applyThis: any,
                            applyArgs: any[], source: string): any {
        return this._invokeZS
            ? this._invokeZS.onInvoke(this._invokeDlgt, this.zone, targetZone,
```

```
                                                callback, applyThis, applyArgs, source)
            : callback.apply(applyThis, applyArgs);
      }
```

The `scheduleTask` method is a bit different, in that it first **1** tries to use the `_scheduleTaskZS` (if defined), otherwise **2** tries to use the supplied task's `scheduleFn` (if defined), otherwise **3** if a microtask calls `scheduleMicroTask()`, otherwise **4** it is an error:

```
scheduleTask(targetZone: Zone, task: Task): Task {
   try {
1    if (this._scheduleTaskZS) {
       return this._scheduleTaskZS.onScheduleTask(
            this._scheduleTaskDlgt, this.zone, targetZone, task);
2    } else if (task.scheduleFn) {
       task.scheduleFn(task)
3    } else if (task.type == 'microTask') {
       scheduleMicroTask(<MicroTask>task);
     } else {
4      throw new Error('Task is missing scheduleFn.');
     }
     return task;
   } finally {
     if (targetZone == this.zone) {
       this._updateTaskCount(task.type, 1);
     }
   }
 }
```

The fork method is where new zones get created. If `_forkZS` is defined, it is used, otherwise a new zone is created with the supplied `targetZone` and `zoneSpec`:

```
fork(targetZone: Zone, zoneSpec: ZoneSpec): AmbientZone {
  return this._forkZS
      ? this._forkZS.onFork(
                       this._forkDlgt, this.zone, targetZone, zoneSpec)
      : new Zone(targetZone, zoneSpec);
}
```

The internal variable _currentZone is initialized to the root zone(_currentTask to null):

```
let _currentZone: Zone = new Zone(null, null);
let _currentTask: Task = null;
```

2: Overview Of The Angular Main Project

Angular - A Family of Projects

Angular is a family of projects revolving around creating a rich programming framework for modern web applications. In this source tour we are going to focus on the latest Angular, whose project homepage is located here:

- https://angular.io

We do not cover the previous version of Angular, called AngularJS.

Source code for a variety of Angular-related projects is available on github:

- https://github.com/angular

The source tree for the Angular Main project is here:

- https://github.com/angular/angular

Source code for the layered projects include:

- Angular Mobile Toolkit – manifest and service worker tooling (https://github.com/angular/mobile-toolkit)
- Universal Angular – prerendering on the server (https://github.com/angular/universal)
- Angular CLI – a command line interface for creating Angular artifacts (https://github.com/angular/angular-cli)
- Angular Material 2 – a set of components built on top of Angular implementing the material design (https://github.com/angular/material2)
- Angular Protractor – end-to-end testing framework for Angular applications (https://github.com/angular/protractor)

There are small websites (microsites) dedicated to each of those additional projects:

- https://mobile.angular.io
- https://universal.angular.io
- https://cli.angular.io
- https://material.angular.io
- https://protractor.angular.io

The source code for the main Angular.io website and the microsites is also available:

- https://github.com/angular/angular.io
- https://github.com/angular/microsites

Keeping Up to Date

The Angular project is evolving rapidly and it is important for application developers using it to keep up to date with progress.

A good place to start is with the notes from the Angular Weekly Meeting:

- http://g.co/ng/weekly-notes

When updates to the codebase are released, the latest changes are recorded in the change Log (CHANGELOG.md) in the root folder and app developers should review the contents of this file from time to time.

For general news on Angular, keep an eye on:

- https://angular.io/news.html

The next version will be Angular 4, expected in March, 2017. During the development of Angular, the router went through a number of significant iterations and had its own independent version number – which currently stands at version 3 (within Angular). To bring versioning of Angular and its router back in sync, the next version of Angular will be set to 4 and not 3). Angular 4 is expected to contain smaller evolutionary improvements (certainly nothing on the scale of change between AngularJS and Angular). When Angular 4 arrives, expect more use of just "Angular" to describe the framework.

Select Your Language

Application developers creating Angular-based application can use one of three languages – TypeScript (recommended), JavaScript or DART.

The DART version of Angular is a separate project:

With TypeScript, during a compilation stage TypeScript gets transpiled (converted) into JavaScript, and it is JavaScript code that executes in the browser. While studying the source tree, we will see how this works.

JavaScript is standardized by ECMA as the ECMAScript specification:

- http://www.ecma-international.org/publications/standards/Ecma-262.htm

Recently ECMA changes versioning from simple digit (e.g. 5, 6) to year of publication (e.g. 2015/2016/2017). The most important versions are ECMAScript 5/es5 (available in all browsers today); and ECMAScript 6 / es6 / ECMAScript 2015, only available in modern browsers, and over time will become dominant. ECMAScript 2015 introduced classes (familiar to those from a C++, C# or Java background) which provide the basis for TypeScript classes. There are plans to introduce a new version of the ECMAScript specification each year - there is very little new in ECMAScript 2016:

- http://www.ecma-international.org/ecma-262/7.0/

but ECMAScript 2017 is likely to be much more interesting.

Google's Dart is a wonderful new language focusing on enabling modern programming in web, mobile app and server environments (and with Dartino, even embedded systems are supported). Dart is a different language to JavaScript, whereas TypeScript is an extension to JavaScript.

- https://www.dartlang.org/

The Dart language specification is here:

- https://www.dartlang.org/docs/spec/

There is an active Dart user group here:

- https://groups.google.com/a/dartlang.org/forum/#!forum/misc

Dart is being use by important Google projects such as Google Fiber and AdWords:

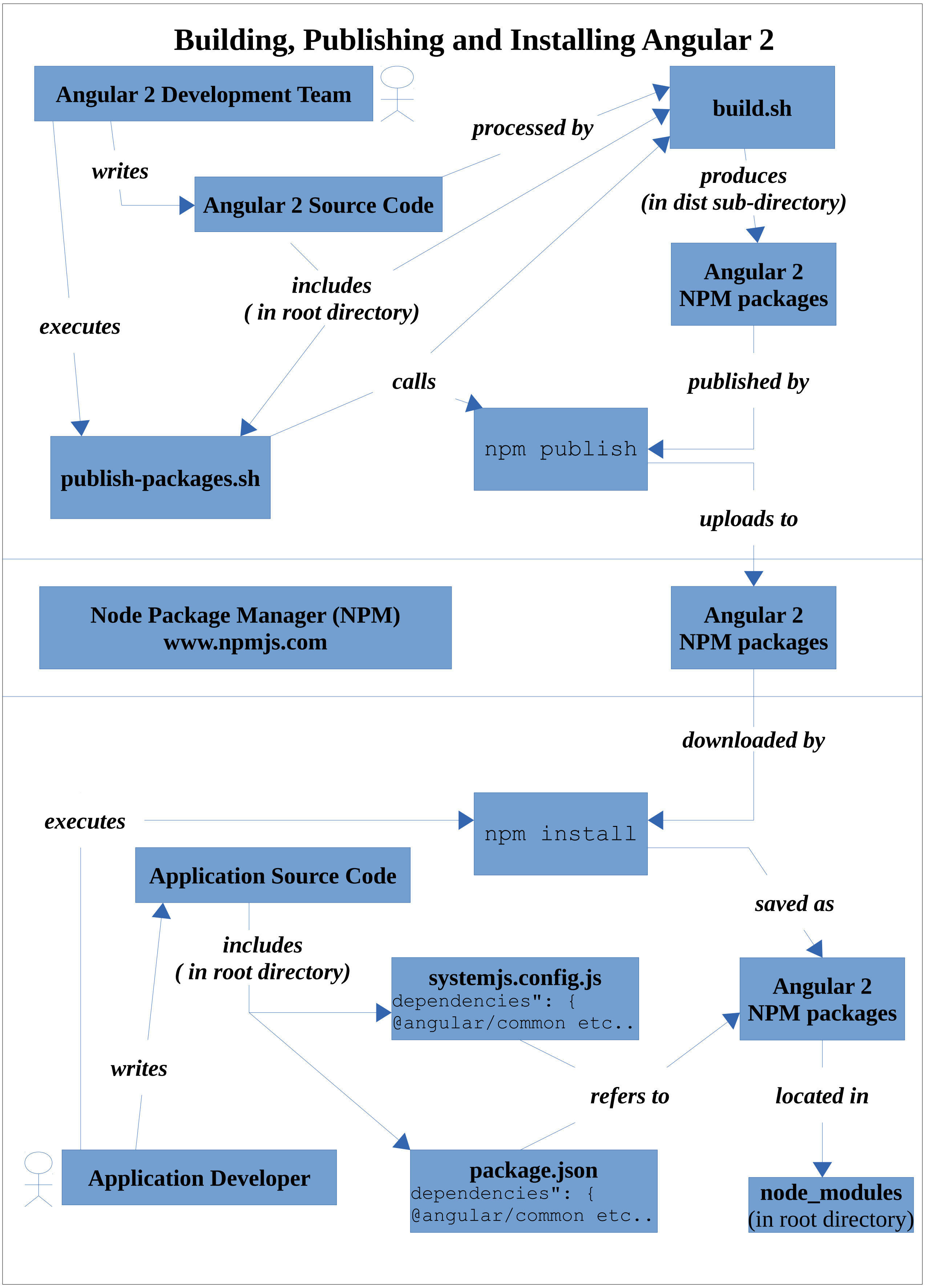
- http://news.dartlang.org/2015/11/how-google-uses-angular-2-with-dart.html

Before Dart, interest in evolving web languages had plateaued – even though the execution speed in web browsers of JavaScript was improving nicely, the JavaScript language itself was not. When Dart was create by a smart team of Googlers from Aarhus, Denmark and elsewhere, it demonstrated what could be achieved in modern web programming and encouraged the competition to improve. Afterward along came TypeScript which had many of the same features as Dart. Dart has two modes of execution – a language-specific VM and transpiling to JavaScript. Initially Google had a plan to embed the Dart VM in Chrome but due to the difficulties of have two simultaneous VMs (Dart and JavaScript) and a combined garbage collector (which Google called project "Oilpan"), this effort was scrapped, so now for web development, Dart code is transpiled to JavaScript for execution (just like TypeScript).

So, after briefly considering all languages for Angular, which language should we choose? We like Dart, but for the moment it is not fully supported in Angular and the Angular / Dart combination does not have much industry backing. We are fans of a class-based approach to programming, so if picking ECMAScript, we would go for the class-based ECMAScript 2015. To get ECMAScript 2015 working in all browsers, we need to transpile it to es5 (just as TypeScript does). However, in addition to classes we also like generics/interfaces/and many other features that TypeScript layers on top of ECMAScript 2015, so therefore we think it is best to select TypeScript for our Angular application development. Over time we expect some (but probably not all) TypeScript features to percolate up into new ECMAScript specifications, so even though TypeScript is not a formal standard like ECMAScript, we think much of what we like about it will be in future standards (which developer would not like to use language features now from a standard language from three years in the future?). Further reasons for this selection is of all the supported languages, Angular application development in TypeScript has the best documentation, best examples and most industry support, so that is our choice. Finally, and very relevant to the purpose of this source tour, if as application developers we wish to read the Angular source code, which is almost all written in TypeScript, we have to learn TypeScript.

The Build Process

TypeScript application code gets transpiled to JavaScript, so there is no distinct set of TypeScript packages – instead TypeScript application developers should use the npm packages.


Building, Publishing and Installing Angular 2
Angular 2 Development Team
processed by
build.sh
writes
Angular 2 Source Code
produces
(in dist sub-directory)
Angular 2
NPM packages
includes
(in root directory)
executes
calls
published by
npm publish
publish-packages.sh
uploads to
Node Package Manager (NPM)
www.npmjs.com
Angular 2
NPM packages
downloaded by
executes
npm install
Application Source Code
saved as
includes
(in root directory)
systemjs.config.js
dependencies": {
@angular/common etc..
Angular 2
NPM packages
writes
refers to
located in
package.json
dependencies": {
@angular/common etc..
node_modules
(in root directory)
Application Developer

3: Tsc-Wrapped

Overview

Tsc-wrapped is a wrapper for the TypeScript tsc compiler that allows extensions to be added. The Angular ahead-of-time (AOT) compiler, Compiler-CLI, calls tsc-wrapped, which in turn calls tsickle (which we will examine in a later chapter).

Tsc-wrapped is located in:

- <ANGULAR-MASTER>/tools/@angular/tsc-wrapped

and is the only entry there under tools/@angular. In contrast, there are many projects under modules/@angular, where most of the Angular source lives.

Source Tree Layout

The source tree root directory contains two sub-directories:

- src
- test

It also directly contains these files:

- index.ts
- package.json
- readme.md

The readme.md file contains an explanation for the purpose of tsc-wrapped and four specific use cases:

```
This package is an internal dependency used by @angular/compiler-cli. Please
use that instead.

This is a wrapper around TypeScript's `tsc` program that allows us to hook in
extra extensions. TypeScript will eventually have an extensibility model for
arbitrary extensions. We don't want to constrain their design with baggage
from a legacy implementation, so this wrapper only supports specific
extensions developed by the Angular team:

- tsickle down-levels Decorators into Annotations so they can be tree-shaken
- tsickle can also optionally produce Closure Compiler-friendly code
- ./collector.ts emits an extra `.metadata.json` file for every `.d.ts` file
  written, which retains metadata about decorators that is lost in the TS emit
- @angular/compiler-cli extends this library to additionally generate
  template code

## TypeScript Decorator metadata collector

The `.d.ts` format does not preserve information about the Decorators applied
to symbols. Some tools, such as Angular template compiler, need access to
statically analyzable information about Decorators, so this library allows
programs to produce a `foo.metadata.json` to accompany a `foo.d.ts` file, and
preserves the information that was lost in the declaration emit.
```

The index.ts file exports various types from the src sub-directory:

```
export {DecoratorDownlevelCompilerHost, MetadataWriterHost}
   from './src/compiler_host';
export {CodegenExtension, main} from './src/main';
export {default as AngularCompilerOptions} from './src/options';
export * from './src/cli_options';
export * from './src/collector';
export * from './src/schema';
```

The package.json file lists tsickle as a dependency:

```
{
  "name": "@angular/tsc-wrapped",
  "description": "Wraps the tsc CLI, allowing extensions.",
  ...
    "dependencies": {
    "tsickle": "^0.2"
  },
  ...
}
```

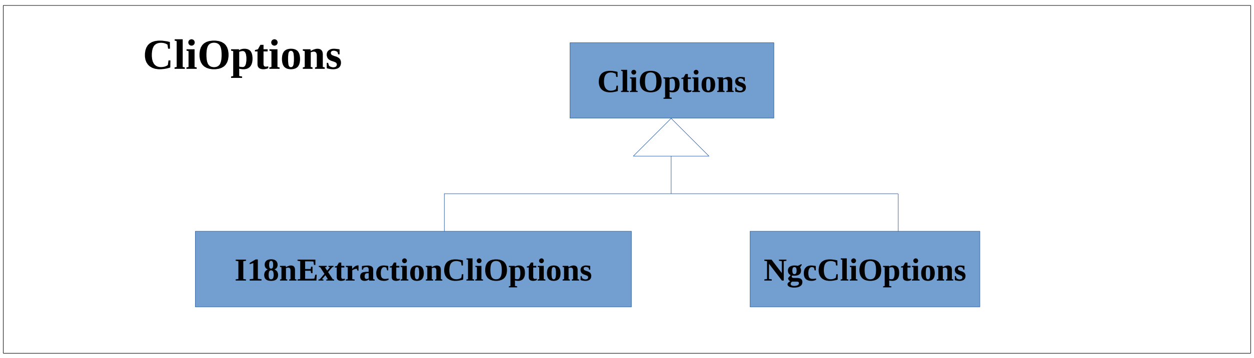
The src directory

The src sub-directory contains these files:

- cli_options.ts
- collector.ts
- compiler_host.ts
- evaluator.ts
- main.ts
- options.ts
- schema.ts
- symbols.ts
- tsc.ts

The CLI entry point is at the bottom of main.ts, and sets the project name to `args.p` or `args.project` (or '.' by default) and creates a `CliOptions` instance with the other arguments, and passes both to the main function.

`CliOptions` is defined in the cli_options.ts file and has this small hierarchy:



Different options are supplied for the internationalization functionality (i18n) and the compilation functionality (ngc). The constructors in each case initialize the properties:

```
export class CliOptions {
  public basePath: string;
  ..
}
export class I18nExtractionCliOptions extends CliOptions {
  public i18nFormat: string;
  ..
}
export class NgcCliOptions extends CliOptions {
  public i18nFormat: string;
  public i18nFile: string;
  public locale: string;
  ..
}
```

The options.ts file defines an interface, `Options`, that extends TypeScript's `CompilerOptions`. It adds the following properties:

```
// Absolute path to a directory where generated file structure is written.
  // If unspecified, generated files will be written alongside sources.
  genDir?: string;
  // Path to the directory containing the tsconfig.json file.
  basePath?: string;
  // Don't produce .metadata.json files
  //   (they don't work for bundled emit with --out)
  skipMetadataEmit?: boolean;
  // Produce an error if the metadata written for a class would
  // produce an error if used.
  strictMetadataEmit?: boolean;
  // Don't produce .ngfactory.ts or .css.shim.ts files
  skipTemplateCodegen?: boolean;
  // Whether to generate code for library code.
  // If true, produce .ngfactory.ts and .css.shim.ts files for .d.ts inputs.
  // Default is true.
  generateCodeForLibraries?: boolean;
  // Insert JSDoc type annotations needed by Closure Compiler
  annotateForClosureCompiler?: boolean;
  // Modify how angular annotations are emitted to improve tree-shaking.
  // Default is static fields. decorators: Leave the Decorators in-place.
  // This makes compilation faster.
  //     TypeScript will emit calls to the __decorate helper.
  //     `--emitDecoratorMetadata` can be used for runtime reflection.
  //     However, the resulting code will not properly tree-shake.
  //     static fields: Replace decorators with a static field in the class.
  //     Allows advanced tree-shakers like Closure Compiler to remove
  //     unused classes.
  annotationsAs?: 'decorators'|'static fields';
  // Print extra information while running the compiler
  trace?: boolean;
  // Whether to embed debug information in the compiled templates
  debug?: boolean;
```

The TypeScript tsc compiler uses a supplied compiler host instance to interact with the world outside the compiler (e.g. loading source files from the file system). Tsc-wrapped uses custom compiler hosts to implement varying functionality.

The compiler_host.ts source file contains four (one abstract) such implementations:

```
// Implementation of CompilerHost that forwards all methods to another
// instance. Useful for partial implementations to override only methods
// they care about.
export abstract class DelegatingHost implements ts.CompilerHost {..}
export class DecoratorDownlevelCompilerHost extends DelegatingHost {..}
export class TsickleCompilerHost extends DelegatingHost {..}
export class MetadataWriterHost extends DelegatingHost {..}
```

`DelegatingHost` allows subclasses to replace default compile host implementation with custom functionality, and to limit additional code only to that which is different. `DecoratorDownlevelCompilerHost` and `TsickleCompilerHost` replace the `getSourceFile` method, whereas `MetadataWriterHost` replaces the `writeFile` callback.

`DecoratorDownlevelCompilerHost.GetSourceFile()` calls `tsickle.convertDecorators()` and `tsickleCompilerHost.GetSourceFile()` calls `tsickle.annotate()`. `MetadataWriterHost.writeFile()` uses the `MetadataCollector` which is defined locally in tsc-wrapped.

Back to main.ts. We see that `CodegenExtension` is a type is defined as :

```
import NgOptions from './options';
export type CodegenExtension =
    (ngOptions: NgOptions, cliOptions: CliOptions,
     program: ts.Program, host: ts.CompilerHost) => Promise<void>;
```

The main function is defined as :

```
export function main(project: string, cliOptions: CliOptions,
                     codegen?: CodegenExtension): Promise<any> {
    ...
    // read the configuration options from wherever you store them
    const {parsed, ngOptions} = tsc.readConfiguration(project, basePath);
    ngOptions.basePath = basePath;
    const createProgram = (host: ts.CompilerHost, oldProgram?: ts.Program) =>
        ts.createProgram(parsed.fileNames, parsed.options, host, oldProgram);
    const host = ts.createCompilerHost(parsed.options, true);
    ...
    const program = createProgram(host);
    ...
   return codegen(ngOptions, cliOptions, program, host).then(() => {
     if (diagnostics) console.timeEnd('NG codegen');
     let definitionsHost = host;
     if (!ngOptions.skipMetadataEmit) {
1      definitionsHost = new MetadataWriterHost(host, ngOptions);
     }
     // Create a new program since codegen files were created
     // after making the old program
     let programWithCodegen = createProgram(definitionsHost, program);
     tsc.typeCheck(host, programWithCodegen);
     let preprocessHost = host;
     let programForJsEmit = programWithCodegen;
     if (ngOptions.annotationsAs !== 'decorators') {
       if (diagnostics) console.time('NG downlevel');
       const downlevelHost =
2      new DecoratorDownlevelCompilerHost(preprocessHost, programForJsEmit);
       // A program can be re-used only once; save the programWithCodegen
```

```
        // to be reused by metadataWriter
        programForJsEmit = createProgram(downlevelHost);
        check(downlevelHost.diagnostics);
        preprocessHost = downlevelHost;
        if (diagnostics) console.timeEnd('NG downlevel');
      }

      if (ngOptions.annotateForClosureCompiler) {
        if (diagnostics) console.time('NG JSDoc');
        const tsickleHost =
3         new TsickleCompilerHost(preprocessHost,programForJsEmit,ngOptions);
        programForJsEmit = createProgram(tsickleHost);
        check(tsickleHost.diagnostics);
        if (diagnostics) console.timeEnd('NG JSDoc');
      }

      // Emit *.js and *.js.map
      tsc.emit(programForJsEmit);

      // Emit *.d.ts and maybe *.metadata.json
      // Not in the same emit pass with above, because tsickle erases
      // decorators which we want to read or document.
      // Do this emit second since TypeScript will create missing
      // directories for us
      // in the standard emit.
      tsc.emit(programWithCodegen);
  ..
}
```

We see the different compiler hosts (1, 2, 3) being used to implement varying functionality.

4: @Angular/Facade

Overview

Facade is a collection of wrapper and utility-style types that give more control over access to low-level functionality. Unlike all the other @angular packages, it has no index.ts in its root directory. Most of its functionality is used internally by the other packages and is imported directly by them.

Unlike all the directories in modules/@angular, Facade is not a package. It does not have a package.json file. It you look up package.json of the application source tree generated by Angular CLI, Facade is not listed among the dependencies, unlike say, @angular/common or @angular/core, hence when you run `npm install` to download all needed packages, and later look in your application's node_modules directory, there is no @angular/facade in there.

Each of the @angular packages needs to contain a symbolic link in its src directory to the facade directory. For Windows, you need to call the create-symlinks.sh script (in <ANGULAR2>/scripts) to create these links – refer to our earlier discussion of this issue when we explored the scripts directory.

@Angular/Facade API

A single Facade type, `EventEmitter`, is exported, and this is done via core's index.ts,

- <ANGULAR2>/modules/@angular/core/index.ts

with this line:

```
export {EventEmitter} from './src/facade/async';
```

An event emitter is an RxJS observable for a stream of events and is widely used within Angular.

Source Tree Layout

The root directory of facade is here:

- <ANGULAR2>/modules/@angular/facade

It directly contains no files. It has a src and a test sub-directories.

Source

The files in facade's src sub-directory:

- <ANGULAR2>/modules/@angular/facade/src

are as follows:

- async.ts
- browser.ts
- collection.ts
- errors.ts
- intl.ts
- lang.ts

Two classes, `BaseError` and `WrappedError`, are defined in errors.ts.

`BaseError` extends the standard `Error` interface from lib.es5.d.ts:

```
interface Error {
    name:    string;
    message: string;
    stack?:  string;
}
```

In its constructor, `BaseError` takes in an error message and passes in to the constructor of its `Error` superclass,and stores the constructed error object in a field called `_nativeError`. All the `BaseError` methods simply call equivalent methods in `_nativeError`. In `BaseError`'s get stack and set stack accessors note the cast to `any`. These add and return stack information to an error object.

```
export class BaseError extends Error {
  _nativeError: Error;
  constructor(message: string) {
     var nativeError = super(message) as any as Error;
    this._nativeError = nativeError;
  }
  get message()          { return this._nativeError.message; }
  set message(message)  { this._nativeError.message = message; }
  get name()             { return this._nativeError.name; }
  get stack()            { return (this._nativeError as any).stack; }
  set stack(value)       { (this._nativeError as any).stack = value; }
  toString()             { return this._nativeError.toString(); }
}
```

`WrappedError` extends `BaseError` and in its constructor takes in an error to wrap and stores this in a field called `_originalError`. The one change `WrappedError` makes to the `BaseError` API is to re-implement `get stack`. It either returns the recorded stack of `_originalError` (if an instance of `Error`) or otherwise `_nativeError`:

```
export class WrappedError extends BaseError {
  originalError: any;
  _nativeError: Error;
  constructor(message: string, error: any) {
    super(
  `${message} caused by: ${error instanceof Error ? error.message: error }`);
    this.originalError = error;
  }
  get stack() {
    return (
     (this.originalError instanceof Error
              ? this.originalError
              : this._nativeError)
     as any).stack;
  }
}
```

The lang.ts file supplies a range of small language-related utilitiy functions, often only one or a few lines long. Here is a small selection:

```
export function hasConstructor(value: Object, type: any): boolean {
  return value.constructor === type;
```

```
}

export function isBoolean(obj: any): boolean {
  return typeof obj === 'boolean';
}
export function getTypeNameForDebugging(type: any): string {
  if (type['name']) {
    return type['name'];
  }
  return typeof type;
}
```

It also adds wrappers for primitives, with useful extra functionality. For example, `StringWrapper` has this method to remove characters on the left of a string:

```
  static stripLeft(s: string, charVal: string): string {
    if (s && s.length) {
      var pos = 0;
      for (var i = 0; i < s.length; i++) {
        if (s[i] != charVal) break;
        pos++;
      }
      s = s.substring(pos);
    }
    return s;
  }
```

A single class, `EventEmitter`, is defined in the async.ts file. It is widely used in the Angular packages, including: by directives, components, `NgZone`, location, async pipe, metadata, forms, http, the message bus for platform browser's webworker communication and more. In other words, it is used everywhere within Angular, and shows the importance of RxJS to Angular development.

`EventEmitter` extends RxJS's `Subject`:

```
export class EventEmitter<T> extends Subject<T> {...}
```

A subject is both an observer and an observable. RxJS declares `Subject` as:

```
export declare class Subject<T>
                extends Observable<T>
                implements Observer<T>, Isubscription
```

`EventEmitter` manages a single field, which is initialized in the constructor:

```
  __isAsync: boolean;
  constructor(isAsync: boolean = false) {
    super();
    this.__isAsync = isAsync;
  }
```

When client code wishes this event emitter to emit a value, it calls the `emit()` method, which forwards the value to the base class observable's `next()` method:

```
emit(value?: T) { super.next(value); }
```

The last method is `subscribe`. It takes three parameters, `generatorOrNext`, `error` and `complete`, all of type `any` and all optional.

```
subscribe(generatorOrNext?: any, error?: any, complete?: any): any {
```

It declares three local variables, and after being set in this method, the last line passes them to the base class observerable's `subscribe()` method, and its result is returned as the result of this method:

```
let schedulerFn: any;
let errorFn = (err: any): any => null;
let completeFn = (): any => null;
…
return super.subscribe(schedulerFn, errorFn, completeFn);
```

The `generatorOrNext` parameter should either be an object with a `next`, `error` and `complete` methods, or a function that can be called directly, with a value parameter.

The body of `EventEmitter.subscribe` has an if / else statement that checks `generatorOrNext`. (When we slightly reformat the layout of the code to better line up the conditional statements), the if side looks like:

```
if (generatorOrNext && typeof generatorOrNext === 'object') {

  schedulerFn =
    this.__isAsync
    ? (value: any) => {setTimeout(() => generatorOrNext.next(value));}
    : (value: any) => { generatorOrNext.next(value); };

  if (generatorOrNext.error) {
    errorFn =
      this.__isAsync
      ? (err) => { setTimeout(() => generatorOrNext.error(err)); }
      : (err) => { generatorOrNext.error(err); };
  }

  if (generatorOrNext.complete) {
    completeFn =
      this.__isAsync
      ? () => { setTimeout(() => generatorOrNext.complete()); }
      : () => { generatorOrNext.complete(); };
  }
}
```

So if `__isAsync` is set, we set `schedulerFn` to call `setTimeout` with a handler that results in `generatorOrNext.next()` being called, and since this `setTimeout` has no additional `timeout` parameter, it default to 0. Hence a request will be added to the event queue to execute this handler, which will happen as the event queue is drained. Very importantly, it is not called immediately. Code that is currently running on the JavaScript stack will run to completion, and only then will the next event be extracted from the event queue. If `__isAsync` is not set, then `schedulerFn` is set to a direct call to `generatorOrNext.next()`. `ErrorFn` and `completeFn` are similarly configured.

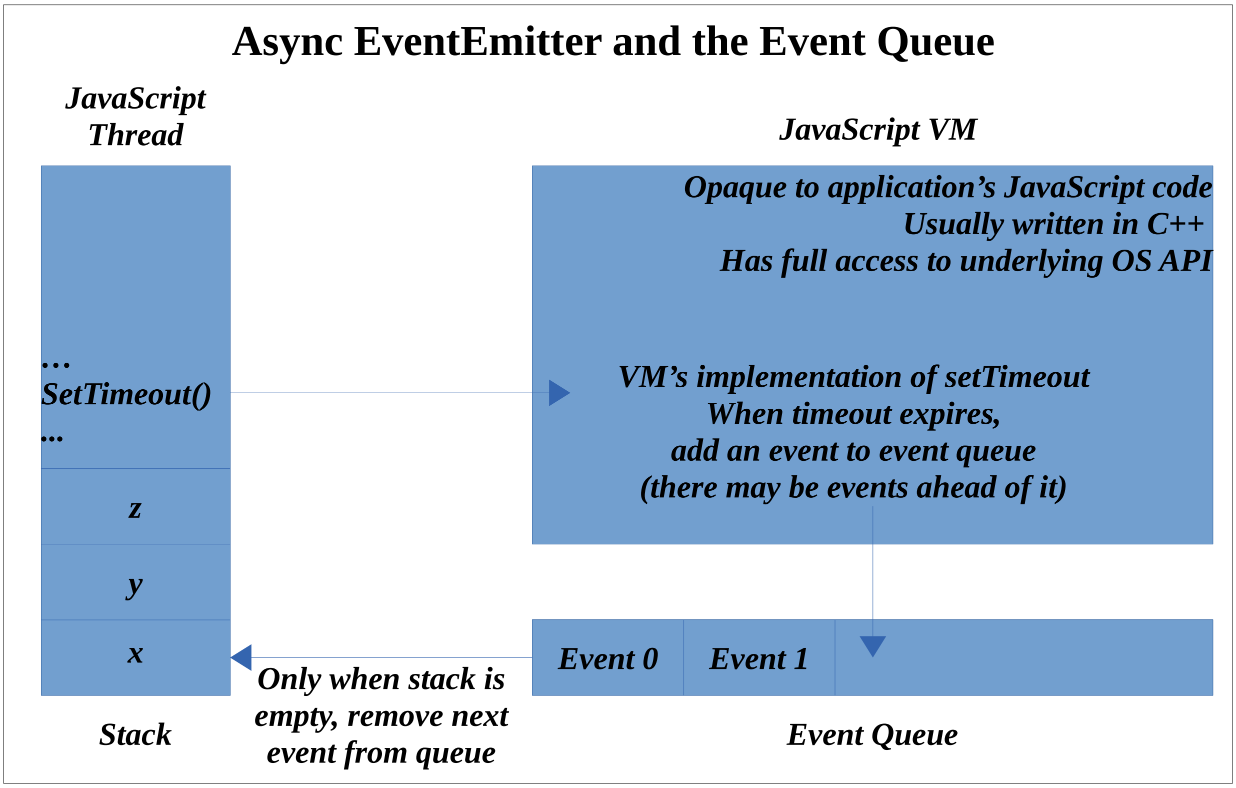
The else side of this if/else is defined as:

```
else {
  schedulerFn =
    this.__isAsync
      ? (value: any ) => { setTimeout(() => generatorOrNext(value)); }
```

```
        : (value: any ) => { generatorOrNext(value); };
    if (error) {
      errorFn =
        this.__isAsync
        ? (err) => { setTimeout(() => error(err)); }
        : (err) => { error(err); };
    }

    if (complete) {
      completeFn =
        this.__isAsync
        ? () => { setTimeout(() => complete()); }
        : () => { complete(); };
    }
```

The `schdulerFn`, `errorFn` and `completeFn` set to handlers that call the passed in function parameters.

How `EventEmitter.subscribe()` works when `__isAsync` is true is best described with the following diagram:



The collection.ts file contains wrappers for collection types, such as maps and lists. Note the methods of these wrappers are static, so the actual collection data needs to be passed in a parameter (usually the first).

For example, the `ListWrapper` class exposes these static methods:

```
export class ListWrapper {
  static createFixedSize(size: number): any[] { }
  static createGrowableSize(size: number): any[] { }
  static clone<T>(array: T[]): T[] { }
```

```
  static forEachWithIndex<T>(array: T[], fn: (t: T, n: number) => void) {}
  static first<T>(array: T[]): T {}
  static last<T>(array: T[]): T {}
  static indexOf<T>(array: T[], value: T, startIndex: number = 0): number {}
  static contains<T>(list: T[], el: T): boolean { }
  static reversed<T>(array: T[]): T[] {}
  static concat(a: any[], b: any[]): any[] { }
  static insert<T>(list: T[], index: number, value: T) { }
  static removeAt<T>(list: T[], index: number): T {}
  static removeAll<T>(list: T[], items: T[]) {}
  static remove<T>(list: T[], el: T): boolean {}
  static clear(list: any[]) {}
  static isEmpty(list: any[]): boolean {  }
  static fill(list: any[], value:any,start: number = 0, end: number = null){}
  static equals(a: any[], b: any[]): boolean {}
  static slice<T>(l: T[], from: number = 0, to: number = null): T[] {}
  static splice<T>(l: T[], from: number, length: number): T[] { }
  static sort<T>(l: T[], compareFn?: (a: T, b: T) => number) {}
  static toString<T>(l: T[]): string { }
  static toJSON<T>(l: T[]): string { }
  static maximum<T>(list: T[], predicate: (t: T) => number): T {}
  static flatten<T>(list: Array<T|T[]>): T[] {}
  static addAll<T>(list: Array<T>, source: Array<T>): void {}
}
```

The browser.ts file defines a small number of browser-related exports:

```
/**
 * JS version of browser APIs. This library can only run in the browser.
 */
var win = typeof window !== 'undefined' && window || <any>{};
export {win as window};
export var document = win.document;
export var location = win.location;
export var gc = win['gc'] ? () => win['gc']() : (): any => null;
export var performance = win['performance'] ? win['performance'] : null;
export const Event = win['Event'];
export const MouseEvent = win['MouseEvent'];
export const KeyboardEvent = win['KeyboardEvent'];
export const EventTarget = win['EventTarget'];
export const History = win['History'];
export const Location = win['Location'];
export const EventListener = win['EventListener'];
```

The intl.ts file contains small helper classes and functions for internationalization. It deals with regional-specific data such as dates, time format and number formatting.

An example class is `NumberFormatter`, which has a single static method, format, that returns an international number format via a call to the standard `Intl.NumberFormat` interface in lib.es6.d.ts:

```
export class NumberFormatter {
  static format(
      num: number, locale: string, style: NumberFormatStyle,
      {minimumIntegerDigits, minimumFractionDigits,
           maximumFractionDigits, currency, currencyAsSymbol = false}: {
        minimumIntegerDigits?: number,
        minimumFractionDigits?: number,
```

```
        maximumFractionDigits?: number,
        currency?: string,
        currencyAsSymbol?: boolean
      } = {}): string {
    let options: Intl.NumberFormatOptions = {
      minimumIntegerDigits,
      minimumFractionDigits,
      maximumFractionDigits,
      style: NumberFormatStyle[style].toLowerCase()
    };
    if (style == NumberFormatStyle.Currency) {
      options.currency = currency;
      options.currencyDisplay = currencyAsSymbol ? 'symbol' : 'code';
    }
    return new Intl.NumberFormat(locale, options).format(num);
  }
}
```

Test Directory

The files in facade’s test sub-directory:

- <ANGULAR2>/modules/@angular/facade/test

are as follows:

- async_spec.ts
- collection-spec.ts
- lang_spec.ts

these contain jasmine unit tests for `EventEmitter`, collections and lang.

5: @Angular/Core

Overview

Core is the foundational package upon which all other packages are based. It supplies a wide range of functionaliy, in areas such as metadata, the template linker, the Ng module system, application initialization, dependency injection, i18n, animation, WTF, and foundational types such as NgZone, Sanitizer and SecurityContext.

@Angular/Core API

The index.ts file in Core's root directory just exports src/core/ts:

- <ANGULAR-MASTER>/modules/@angular/core/src/core.ts

which is where Core's API is exported:

```
export * from './metadata';
export * from './version';
export * from './util';
export * from './di';
export {createPlatform, assertPlatform, destroyPlatform, getPlatform,
  PlatformRef, ApplicationRef, enableProdMode, isDevMode,
  createPlatformFactory, NgProbeToken} from './application_ref';
export {APP_ID, PACKAGE_ROOT_URL, PLATFORM_INITIALIZER,
  APP_BOOTSTRAP_LISTENER} from './application_tokens';
export {APP_INITIALIZER, ApplicationInitStatus} from './application_init';
export * from './zone';
export * from './render';
export * from './linker';
export {DebugElement, DebugNode, asNativeElements, getDebugNode}
  from './debug/debug_node';
export {GetTestability, Testability, TestabilityRegistry,
  setTestabilityGetter} from './testability/testability';
export * from './change_detection';
export * from './platform_core_providers';
export {TRANSLATIONS, TRANSLATIONS_FORMAT, LOCALE_ID} from './i18n/tokens';
export {ApplicationModule} from './application_module';
export {wtfCreateScope, wtfLeave, wtfStartTimeRange,
  wtfEndTimeRange, WtfScopeFn} from './profile/profile';
export {Type} from './type';
export {EventEmitter} from './facade/async';
export {ErrorHandler} from './error_handler';
export * from './core_private_export';
export * from './animation/metadata';
export {AnimationTransitionEvent}
  from './animation/animation_transition_event';
export {AnimationPlayer} from './animation/animation_player';
export {Sanitizer, SecurityContext} from './security';
```

Source Tree Layout

The Core source tree is at:

- <ANGULAR-MASTER>/modules/@angular/core

The source tree for the Core package contains these directories:

- src
- test (unit tests in Jasmine)
- testing (test tooling)

and these files:

- index.ts
- package.json
- rollup.config.js
- rollup-testing.config.js
- tsconfig-build.json
- tsconfig-testing.json

Source

core/src

The core/src directory directly contains many files, which we will group into three categories. Firstly, a number of files just export types from equivalently named sub-directories. Files that fall into this category include:

- change_detection.ts
- core.ts
- core_private_export.ts
- di.ts
- metadata.ts
- linker.ts
- render.ts
- zone.ts

For example, the renderer.ts file is a one-liner that just exports from renderer/api.ts:

```
export {RenderComponentType, Renderer, RootRenderer} from './render/api';
```

and zone.ts file is a one-liner that just exports from zone/ng_zone.ts:

```
export {NgZone } from './zone/ng_zone';
```

Secondly, are files containing what we might call utility functionality:

- console.ts
- error_handler.ts
- platform_core_providers.ts
- security.ts
- types.ts
- util.ts
- version.ts

console.ts contains an injectable service used to write to the console:

```
@Injectable()
export class Console {
  log(message: string): void { print(message); }
//Note: for reporting errors use `DOM.logError()` as it is platform specific
```

```
    warn(message: string): void { warn(message); }
  }
```

It is listed as an entry in `_CORE_PLATFORM_PROVIDERS` in platform_core_providers.ts, which is used to create platforms.

error_handler.ts defines he default error handler and also, in comments, describes how you could implement your own.

platform_core_providers.ts defines `_CORE_PLATFORM_PROVIDERS` which lists the core providers for dependency injection:

```
const _CORE_PLATFORM_PROVIDERS: Provider[] = [
  PlatformRef_,
  {provide: PlatformRef, useExisting: PlatformRef_},
  {provide: Reflector, useFactory: _reflector, deps: []},
  {provide: ReflectorReader, useExisting: Reflector},
  TestabilityRegistry,
  Console,
];
```

It also defines the `platformCore` const, used when creating platforms:

```
export const platformCore =
    createPlatformFactory(null, 'core', _CORE_PLATFORM_PROVIDERS);
```

security.ts defines the `SecurityContext` enum:

```
export enum SecurityContext {
  NONE,
  HTML,
  STYLE,
  SCRIPT,
  URL,
  RESOURCE_URL,
}
```

and the `Sanitizer` abstract class:

```
export abstract class Sanitizer {
  abstract sanitize(context: SecurityContext, value: string): string;
}
```

They are used in @angular/compiler/src/schema and @angular/platform-browser/src/security to enfore security restrictions on potentially dangerous constructs.

The third category of source files directly in @angular/core/src are files related to platform- and application-initialization. These include:

- application_init.ts
- application_module.ts
- application_tokens.ts
- application_ref.ts

The first three of these are small (50-100 lines of code), whereas application_ref.ts is larger at over 500 lines. Let's start with application_tokens.ts. It contains one provider definition and a set of opaque tokens for various uses. `PLATFORM_INITIALIZER` is an opaque token that Angular itself and application code can use to register supplied functions that will be executed when a platform is initialized. An example usage within Angular is in @angular/platform-browser (src/browser.ts) where it is used to have `initDomAdapter` function called upon platform initialization (note use of `multi` – which means multiple such initializer functions can be registered):

```
export const INTERNAL_BROWSER_PLATFORM_PROVIDERS: Provider[] = [
  {provide: PLATFORM_INITIALIZER, useValue: initDomAdapter, multi: true},
  {provide: PlatformLocation, useClass: BrowserPlatformLocation}
];
```

`INTERNAL_BROWSER_PLATFORM_PROVIDERS` is used a few lines later in browser.ts to create the browser platform (and so is used by most Angular applications):

```
export const platformBrowser: (extraProviders?: Provider[]) =>
                PlatformRef = createPlatformFactory(platformCore, 'browser',
                INTERNAL_BROWSER_PLATFORM_PROVIDERS);
```

Another opaque token in application_tokens.ts is `PACKAGE_ROOT_URL` - used to discover the application's root directory. The provider definition identified by `APP_ID` supplies a function that generates a unique string that can be used as an application identifier. A default implementation is supplied, that uses Math.random:

```
function _randomChar(): string {
  return String.fromCharCode(97 + Math.floor(Math.random() * 25));
}
export function _appIdRandomProviderFactory() {
  return `${_randomChar()}${_randomChar()}${_randomChar()}`;
}
export const APP_ID_RANDOM_PROVIDER = {
  provide: APP_ID,
  useFactory: _appIdRandomProviderFactory,
  deps: <any[]>[],
};
```

application_init.ts defines one opaque token and one injectable service. The opaque token is:

```
export const APP_INITIALIZER: any =
                   new OpaqueToken('Application Initializer');
```

`APP_INITIALIZER` Its role is similar to `PLATFORM_INITIALIZER`, except it is called when an application is initialized. The injectable service is `ApplicationInitStatus`, which returns the status of executing app initializers:

```
@Injectable()
export class ApplicationInitStatus {
  private _donePromise: Promise<any>;
  private _done = false;
  constructor(
    @Inject(APP_INITIALIZER) @Optional() appInits: (() => any)[]) {..}
  get done(): boolean { return this._done; }
  get donePromise(): Promise<any> { return this._donePromise; }
}
```

We will soon see how it is used in application_ref.ts.

application_module.ts defines the ApplicationModule class:

```
// This module includes the providers of @angular/core that are needed
// to bootstrap components via `ApplicationRef`.
@NgModule({
  providers: [
    ApplicationRef_,
    {provide: ApplicationRef, useExisting: ApplicationRef_},
    ApplicationInitStatus,
    Compiler,
    APP_ID_RANDOM_PROVIDER,
    ViewUtils,
    AnimationQueue,
    {provide: IterableDiffers, useFactory: _iterableDiffersFactory},
    {provide: KeyValueDiffers, useFactory: _keyValueDiffersFactory},
    {provide: LOCALE_ID, useValue: 'en-US'},
  ]
})
export class ApplicationModule { }
```

Providers are supplied to Angular's dependency injection system. Note the default locale is set to "en-US" - globalized applications may wish to override this for international markets. The IterableDiffers and KeyValueDiffers provides related to change detection. ViewUtils is defined in the src/linker sub-directory and contains utility-style code related to rendering.

Often in the Angular source tree we see an abstract class being defined, along with a concrete implementation (same name with an additional '_' at the end). We see this in ApplicationModule, in these two lines (ApplicationRef_ derives from the abstract ApplicationRef):

```
ApplicationRef_,
{provide: ApplicationRef, useExisting: ApplicationRef_},
```

So if client code asks DI for ApplicationRef_, an instance of that will be returned;; and if client code asks for an instance that implements the base ApplicationRef, then the existi instance of ApplicationRef_ will also be returned. It is dones line this so application code can provide custom implmentations of some of these classes (we would normally not recommend providing a custom implementation of ApplicationRef, but for other types it can be useful).

The types in application_ref.ts plays a pivotal role in how the entire Angular infrastructure works. Application developers wishing to learn how Angular really works are strongly encouraged to carefully study the code in application_ref.ts. Let's start our examination by looking at the `createPlatformFactory()` function:

```
export function createPlatformFactory(
1 parentPlaformFactory:(extraProviders?: Provider[]) => PlatformRef,
2 name: string,
3 providers: Provider[] = [])
4   : (extraProviders?: Provider[]) => PlatformRef {
          const marker = new OpaqueToken(`Platform: ${name}`);
5         return (extraProviders: Provider[] = []) => {
            if (!getPlatform()) {
              if (parentPlaformFactory) {
                parentPlaformFactory(
                  providers.concat(extraProviders).concat(
```

```
                                {provide: marker, useValue: true}));
                } else {
6                 createPlatform(ReflectiveInjector.resolveAndCreate(
                    providers.concat(extraProviders).concat(
                                  {provide: marker, useValue: true})));
      }
    }
7   return assertPlatform(marker);
  };
}
```

It takes three parameters – 1 `parentPlatformFactory`, 2 name and an 3 array of providers. It returns 4 a factory function, that when called, will return a `PlatformRef`.

This factory function first creates an opaque token 5 to use for DI lookup based on the supplied name; then it calls `getPlatform()` to see if a platform already exists (only one is permitted at any one time), and if false is returned, it calls 6 `createPlatform()`, passing in the result of a call to `ReflectiveInjector`'s `resolveAndCreate` (supplied with the providers parameter). Then 7 `assertPlatform` is called with the marker and the result of that call becomes the result of the factory function.

`PlatformRef` is defined as:

```
export abstract class PlatformRef {
  bootstrapModuleFactory<M>(
      moduleFactory: NgModuleFactory<M>): Promise<NgModuleRef<M>> {
    throw unimplemented();
  }
  bootstrapModule<M>(
      moduleType: Type<M>,
      compilerOptions: CompilerOptions|CompilerOptions[] = []):
    Promise<NgModuleRef<M>> { throw unimplemented();
  get injector(): Injector { throw unimplemented(); };

  abstract onDestroy(callback: () => void): void;
  abstract destroy(): void;
  get destroyed(): boolean { throw unimplemented(); }
}
```

A platform represent the hosting environment within which one or more applications execute. Different platforms are supported (e.g. browser UI, webworker, server and you can create your own). For a web page, it is how application code interacts with the page (e.g. sets URL). `PlatformRef` represents the platform, and we see its two main features are supplying the root injector and module bootstrapping. The other members are to do with destroying resources when no longer needed.

One implementation of PlatformRef is supplied, called `PlatformRef_`. It manages the root injector passed in to the constructor, an array of `NgModuleRef`s and an array of destroy listeners. In the constructor, it takes in an injector. Note that calling the platform's `destroy()` method will result in all applications that use that platform having their `destroy()` methods called.

```
@Injectable()
export class PlatformRef_ extends PlatformRef {
```

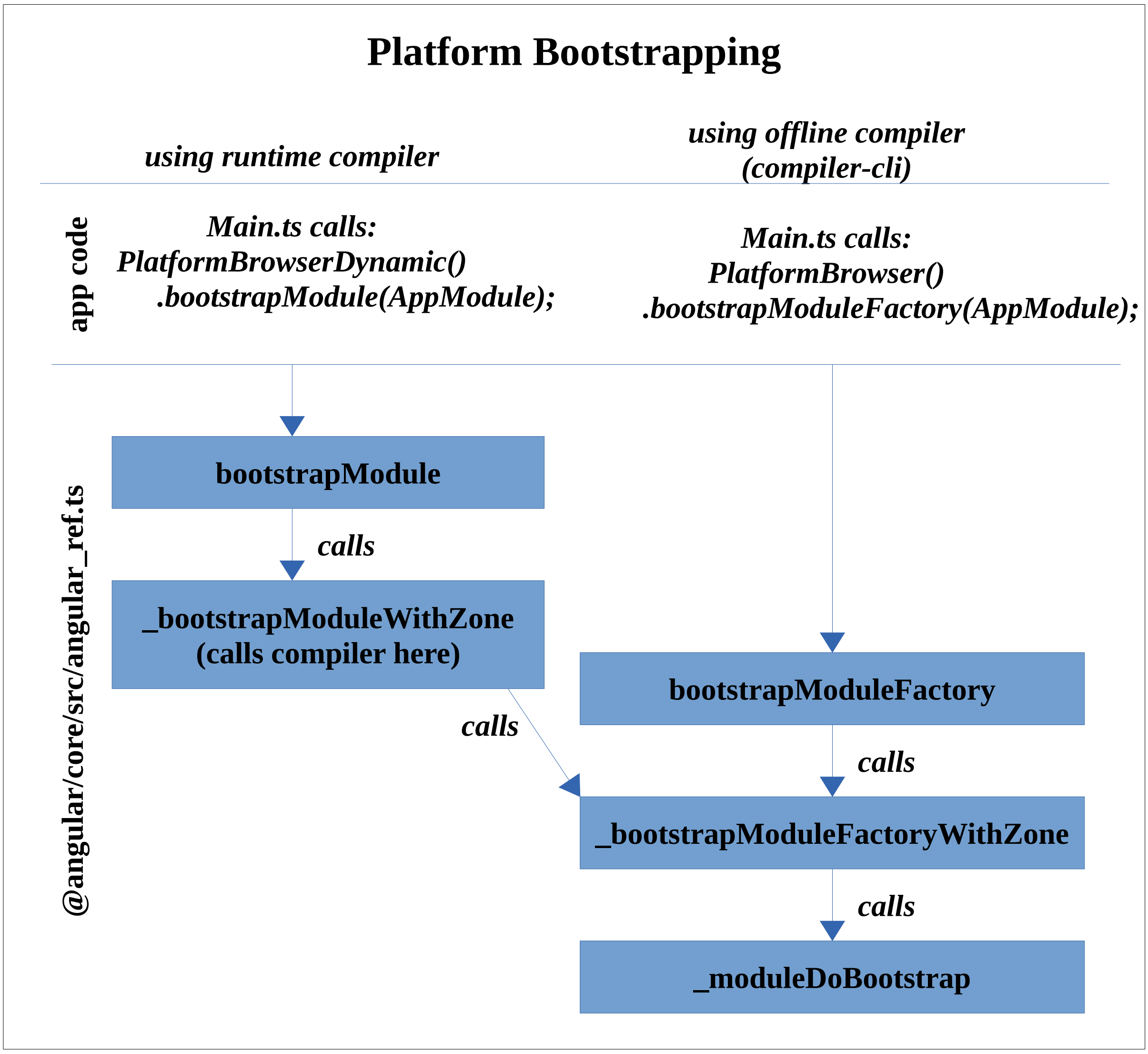
```
private _modules: NgModuleRef<any>[] = [];
private _destroyListeners: Function[] = [];
private _destroyed: boolean = false;

constructor(private _injector: Injector) { super(); }

onDestroy(callback: () => void): void
         { this._destroyListeners.push(callback); }

get injector(): Injector { return this._injector; }

get destroyed() { return this._destroyed; }

destroy() {
  if (this._destroyed) {
    throw new Error('The platform has already been destroyed!');
  }
  this._modules.slice().forEach(module => module.destroy());
  this._destroyListeners.forEach(listener => listener());
  this._destroyed = true;
}
```

The two bootstrapping methods are bootstrapModule and bootstrapModuleFactory. An important decision for any Angular application team is to decide when to use the runtime compilation and when to use offline compilation. Runtime compilation is simpler to use and is demonstrated in the Quickstart on Angular.io. Runtime compilation makes the application bigger (the template compiler needs to run in the browser) and is slower (template compilation is required before the template can be used). Applications that use runtime compilation need to call `bootstrapModule`. Offline compilation involves extra build time configuration and so is a little more complex to set up, but due to its performance advantages is likely to be used for large production applications. Applications that use offline compilation need to call `bootstrapModuleFactory()`.

`PlatformRef_.bootstrapModule()` calls `_bootstrapModuleWithZone` and returns a promise of a `NgModuleRef`. `_bootstrapModuleWithZone` runs the template compiler (`compiler.compileModuleAsync()`) and then returns the result of a call to `_bootstrapModuleFactoryWithZone()`. (so after the extra runtime compilation step, it follows the same code path as that used with offline compilation).

```
bootstrapModule<M>(
  moduleType: Type<M>,
  compilerOptions: CompilerOptions|CompilerOptions[] = [])
 :Promise<NgModuleRef<M>> {
   return this._bootstrapModuleWithZone(moduleType, compilerOptions, null);
 }

 private _bootstrapModuleWithZone<M>(
     moduleType: Type<M>,
     compilerOptions: CompilerOptions|CompilerOptions[] = [],
     ngZone: NgZone,
     componentFactoryCallback?: any): Promise<NgModuleRef<M>> {

   const compilerFactory: CompilerFactory =
                              this.injector.get(CompilerFactory);
```

```
    const compiler = compilerFactory.createCompiler(
        Array.isArray(compilerOptions) ? compilerOptions
                                           : [compilerOptions]);
    if (componentFactoryCallback) {
      return compiler.compileModuleAndAllComponentsAsync(moduleType)
          .then(({ngModuleFactory, componentFactories}) => {
            componentFactoryCallback(componentFactories);
            return this._bootstrapModuleFactoryWithZone(
                                              ngModuleFactory, ngZone);
          });
    }
    return compiler.compileModuleAsync(moduleType)
        .then((moduleFactory) =>
          this._bootstrapModuleFactoryWithZone(moduleFactory, ngZone));
  }
```



PlatformRef_. BootstrapModuleFactory() and BootstrapModuleFactoryWithZone looks like this if we remove comments and error handling:

```
    bootstrapModuleFactory<M>(moduleFactory: NgModuleFactory<M>):
  Promise<NgModuleRef<M>> {
      return this._bootstrapModuleFactoryWithZone(moduleFactory, null);
    }

    private _bootstrapModuleFactoryWithZone<M>(
    moduleFactory: NgModuleFactory<M>, ngZone: NgZone):Promise<NgModuleRef<M>>{
      if (!ngZone)
       ngZone = new NgZone({enableLongStackTrace: isDevMode()});
     return ngZone.run(() => {
        const ngZoneInjector =
            ReflectiveInjector.resolveAndCreate(
                    [{provide: NgZone, useValue: ngZone}], this.injector);
        const moduleRef =
                 <NgModuleInjector<M>>moduleFactory.create(ngZoneInjector);
        const exceptionHandler: ErrorHandler =
                 moduleRef.injector.get(ErrorHandler, null);
        if (!exceptionHandler) {
          throw new Error(
           'No ErrorHandler. Is platform module (BrowserModule) included?');
        }
        moduleRef.onDestroy(()
            => ListWrapper.remove(this._modules, moduleRef));
        ngZone.onError.subscribe(
            {next: (error: any) => { exceptionHandler.handleError(error); }});
        return _callAndReportToErrorHandler(exceptionHandler,
          () => {
          const initStatus: ApplicationInitStatus =
               moduleRef.injector.get(ApplicationInitStatus);
          return initStatus.donePromise.then(() => {
1           this._moduleDoBootstrap(moduleRef);
            return moduleRef;
          });
        });
      });
    }
```

It takes in a NgModuleFactory as a method parameter. It is ensuring the platform code runs in a new NgZone with tracing set depending on the isDevMode setting. It calls the NgModuleFactory's create() method. Then it first runs APP_INITIALIZERs for the application followed by a call 1 to its internal method _moduleDoBootstrap(), which is defined as:

```
    private _moduleDoBootstrap(moduleRef: NgModuleInjector<any>) {
      const appRef = moduleRef.injector.get(ApplicationRef);
      if (moduleRef.bootstrapFactories.length > 0) {
        moduleRef.bootstrapFactories.forEach(
1         (compFactory) => appRef.bootstrap(compFactory));
      } else if (moduleRef.instance.ngDoBootstrap) {
        moduleRef.instance.ngDoBootstrap(appRef);
      } else {
        throw new Error(
            `The module ${stringify(moduleRef.instance.constructor)} was
```

```
            bootstrapped, but it does not declare "@NgModule.bootstrap"
            components nor a "ngDoBootstrap" method. ` +
           `Please define one of these.`);
      }
    }
```

The important line here is the call **1** to `ApplicationRef.bootstrap()`, which we will cover when looking at `ApplicationRef` shortly, but first let's finish off looking at platform-related code.

There are a few simple platform-related functions in core/src/application_ref.ts. `createPlatform()` creates a platform ref instance, or more accurately, as highlighted in the code, asks the injector for a platform ref:

```
export function createPlatform(injector: Injector): PlatformRef {
  if (_platform && !_platform.destroyed) {
    throw new Error(
       'There can be only one platform.
       Destroy the previous one to create a new one.');
  }
  _platform = injector.get(PlatformRef);
  const inits: Function[] =
             <Function[]>injector.get(PLATFORM_INITIALIZER, null);
  if (inits) inits.forEach(init => init());
  return _platform;
}
```

Only a single platform may be active at any one time. `_platform` is defined as:

```
let _platform: PlatformRef;;
```

The `getPlatform()` function is simply defined as:

```
export function getPlatform(): PlatformRef {
  return _platform && !_platform.destroyed ? _platform : null;
}
```

The `assertPlatform()` function ensures two things, and if either false, throws a `BaseException`. Firstly it ensures that a platform exists, and secondly that its injector has a provider for the token specified as a parameter.

```
export function assertPlatform(requiredToken: any): PlatformRef {
  const platform = getPlatform();

  if (!platform) {
    throw new Error('No platform exists!');
  }

  if (!platform.injector.get(requiredToken, null)) {
    throw new Error(
        'A platform with a different configuration has been created.
         Please destroy it first.');
  }

  return platform;
}
```

The `destroyPlatform()` function

```
export function destroyPlatform(): void {
  if (_platform && !_platform.destroyed) {
    _platform.destroy();
  }
}
```

The run mode specifies whether the platform is is production mode or developer mode. By default, it is in developer mode:

```
let _devMode: boolean = true;
let _runModeLocked: boolean = false;
```

This can be set by calling `enableProdMode()`:

```
export function enableProdMode(): void {
  if (_runModeLocked) {
    throw new Error('Cannot enable prod mode after platform setup.');
  }
  _devMode = false;
}
```

To determine which mode is active, call `isDevMode()`. This always returns the same value. In other words, whatever mode is active when this is first call, that is the mode that is always active.

```
export function isDevMode(): boolean {
  _runModeLocked = true;
  return _devMode;
}
```

`ApplicationRef` is defined as:

```
export abstract class ApplicationRef {
  abstract bootstrap<C>(
       componentFactory: ComponentFactory<C>|Type<C>): ComponentRef<C>;
  abstract tick(): void;
  get componentTypes(): Type<any>[] { return <Type<any>[]>unimplemented(); };
  get components(): ComponentRef<any>[] {
      return <ComponentRef<any>[]>unimplemented(); };

  attachView(view: ViewRef): void { unimplemented(); }
  detachView(view: ViewRef): void { unimplemented(); }
  get viewCount() { return unimplemented(); }
}
```

Its has getters for component types and components. It main method is bootstrap(), which is a generic method with a C type parameter - which attaches the component to DOM elements and sets up the application for execution. Note that bootstrap's parameter is a union type, it represents either a `ComponentFactory` or a `Type`, both of which take C as a type parameter.

Attached views are those that are not attached to a view container and are subject to dirty checking. Such views can be attached and detached, and a count returned.

One implementation of `ApplicationRef` is supplied, called `ApplicationRef_`. This is marked as `Injectable()`. It maintains the following fields:

```
    private _bootstrapListeners: Function[] = [];
    private _rootComponents: ComponentRef<any>[] = [];
    private _rootComponentTypes: Type<any>[] = [];
    private _views: AppView<any>[] = [];
    private _runningTick: boolean = false;
    private _enforceNoNewChanges: boolean = false;
```

Its constructor shows what it needs from an injector:

```
   constructor(
        private _zone: NgZone,
        private _console: Console,
        private _injector: Injector,
        private _exceptionHandler: ErrorHandler,
        private _componentFactoryResolver: ComponentFactoryResolver,
        private _initStatus: ApplicationInitStatus,
        @Optional() private _testabilityRegistry: TestabilityRegistry,
        @Optional() private _testability: Testability) {
      super();
      this._enforceNoNewChanges = isDevMode();

      this._zone.onMicrotaskEmpty.subscribe(
          {next: () => { this._zone.run(() => { this.tick(); }); }});
    }
```

Its `bootstrap()` implementation passes some code to the run function (to run in the zone) and this code calls `componentFactory.create()` to create the component and then `_loadComponent()`.

```
  bootstrap<C>(
         componentOrFactory: ComponentFactory<C>|Type<C>): ComponentRef<C> {
      if (!this._initStatus.done) {
        throw new Error(
            'Cannot bootstrap as there are still asynchronous initializers
               running. Bootstrap components in the `ngDoBootstrap` method
                of the root module.');
      }
      let componentFactory: ComponentFactory<C>;
      if (componentOrFactory instanceof ComponentFactory) {
        componentFactory = componentOrFactory;
      } else {
        componentFactory = this._componentFactoryResolver
                               .resolveComponentFactory(componentOrFactory);
      }
      this._rootComponentTypes.push(componentFactory.componentType);
      const compRef = componentFactory.create(
                          this._injector, [], componentFactory.selector);
      compRef.onDestroy(() => { this._unloadComponent(compRef); });
      const testability = compRef.injector.get(Testability, null);
      if (testability) {
        compRef.injector.get(TestabilityRegistry)
            .registerApplication(compRef.location.nativeElement, testability);
      }
      this._loadComponent(compRef);
      if (isDevMode()) {
        this._console.log(
            `Angular is running in the development mode. Call
               enableProdMode() to enable the production mode.`);
```

```
    }
    return compRef;
  }
```

`_loadComponent()` is defined as:

```
private _loadComponent(componentRef: ComponentRef<any>): void {
    this.attachView(componentRef.hostView);
    this.tick();
    this._rootComponents.push(componentRef);
    // Get the listeners lazily to prevent DI cycles.
    const listeners =
        <((compRef: ComponentRef<any>) => void)[]>
                this._injector.get(APP_BOOTSTRAP_LISTENER, [])
                .concat(this._bootstrapListeners);
    listeners.forEach((listener) => listener(componentRef));
  }
```

core/src/util

The core/src/util directory contains two files:

- decorators.ts
- lang.ts

decorators.ts includes definition of the `TypeDecorator` class, which is the basis for Angular's type decorators. Its `makeDecorator()` function uses reflection to examine annotations and returns a `decoratorFactory` function declared inline. Similar make functions are supplied for parameters and properties.

core/src/di

The core/src/di.ts source file exports a variety of dependency injection types:

```
export * from './di/metadata';
export {forwardRef, resolveForwardRef, ForwardRefFn} from './di/forward_ref';
export {Injector} from './di/injector';
export {ReflectiveInjector} from './di/reflective_injector';
export {Provider, TypeProvider, ValueProvider, ClassProvider,
ExistingProvider, FactoryProvider} from './di/provider';
export {ResolvedReflectiveFactory, ResolvedReflectiveProvider}
  from './di/reflective_provider';
export {ReflectiveKey} from './di/reflective_key';
export {OpaqueToken} from './di/opaque_token';
```

The core/src/di directory contains these files:

- forward_ref.ts
- injector.ts
- metadata.ts
- opaque_token.ts
- provider.ts
- provider_util.ts
- reflective_errors.ts
- reflective_injector.ts
- reflective_key.ts
- reflective_provider.ts

The metadata.ts file defines these interfaces:

```
export interface InjectDecorator {
  (token: any): any;
  new (token: any): Inject;
}

export interface Inject { token: any; }

export interface OptionalDecorator {
  (): any;
  new (): Optional;
}

export interface Optional {}

export interface InjectableDecorator {
  (): any;
  new (): Injectable;
}

export interface Injectable {}

export interface SelfDecorator {
  (): any;
  new (): Self;
}

export interface Self {}

export interface SkipSelfDecorator {
  (): any;
  new (): SkipSelf;
}

export interface SkipSelf {}

export interface HostDecorator {
  (): any;
  new (): Host;
}

export interface Host {}
```

metadata.ts also defines these variables:

```
export const Host: HostDecorator = makeParamDecorator('Host', []);
export const Self: SelfDecorator = makeParamDecorator('Self', []);
export const Injectable: InjectableDecorator =
<InjectableDecorator>makeDecorator('Injectable', []);
export const SkipSelf: SkipSelfDecorator = makeParamDecorator('SkipSelf',
[]);
export const Inject: InjectDecorator = makeParamDecorator('Inject',
[['token', undefined]]);
export const Optional: OptionalDecorator = makeParamDecorator('Optional',
[]);
```

forward refs are placeholders used to faciliate out-of-sequence type declarations. The forward_ref.ts file defines an interface and two functions:

```
export interface ForwardRefFn { (): any; }

export function forwardRef(forwardRefFn: ForwardRefFn): Type<any> {
  (<any>forwardRefFn).__forward_ref__ = forwardRef;
  (<any>forwardRefFn).toString = function() { return stringify(this()); };
  return (<Type<any>><any>forwardRefFn);
}

export function resolveForwardRef(type: any): any {
  if (typeof type === 'function' && type.hasOwnProperty('__forward_ref__') &&
      type.__forward_ref__ === forwardRef) {
    return (<ForwardRefFn>type)();
  } else {
    return type;
  }
}
```

The injector.ts file defines the `Injector` abstract class:

```
export abstract class Injector {
  static THROW_IF_NOT_FOUND = _THROW_IF_NOT_FOUND;
  static NULL: Injector = new _NullInjector();
  get(token: any, notFoundValue?: any): any { return unimplemented(); }
}
```

Application code (and indeed, Angular internal code) passes a token to `Injector.get()`, and the implementation returns the matching instance. Concrete implementations of this class need to override the `get()` method, so it actually works as expected. See reflective_injector.ts for a derived class.

The opaque_token.ts file defines the `OpaqueToken` class, which internally uses a string as the opaque token:

```
@Injectable()  // so that metadata is gathered for this class
export class OpaqueToken {
  constructor(private _desc: string) {}

  toString(): string { return `Token ${this._desc}`; }
}
```

provider.ts defines the `Provider` type:

```
export type Provider =
    TypeProvider | ValueProvider
      | ClassProvider | ExistingProvider | FactoryProvider | any[];
```

core/src/metadata

Think of metadata as little nuggets of information we would like to attach to other things. The core/src/metadata.ts file exports a variety of types from the core/src/metadata sub-directory.

The source files in core/src/metadata are:

- di.ts
- directives.ts
- lifecycle_hooks.ts
- ng_module.ts
- view.ts

The di.ts file defines a range of metadata classes derived from `DependencyMetadata`. First is defines `AttributeMetadata` (uses attribute name) and `QueryMetadata` (uses selector) that derive directly from `DependencyMetadata`; then it defines `ContentChidrenMetadata`, `ContentChildMetadata` and `ViewQueryMetadata` – all three of which derive from `QueryMetadata`, and then it defines `ViewChildrenMetadata` and `ViewChildMetadata`, both of which derive from `ViewQueryMetadata`.

The view.ts file defines an enum, a var and a class. The `ViewEncapsulation` enum is defined as:

```
export enum ViewEncapsulation {
  Emulated,
  Native,
  None
}
```

These represent how template and style encapsulation should work. `None` means don't use encapsulation, `Native` means use what the renderer offers (specifically the Shadow DOM) and `Emulated` is best explained by the comment:

```
/**
* Emulate `Native` scoping of styles by adding an attribute containing
* surrogate id to the Host Element and pre-processing the style rules
* provided via ViewMetadata#styles or ViewMetadata#stylesUrls, and adding
* the new Host Element attribute to all selectors.
*
* This is the default option.
*/
```

The `VIEW_ENCAPSULATION_VALUES` is an array that just list those values:

```
export var VIEW_ENCAPSULATION_VALUES = [
   ViewEncapsulation.Emulated,
   ViewEncapsulation.Native,
   ViewEncapsulation.None];
```

The `ViewMetadata` class represents metadata for views:

```
export class ViewMetadata {
  templateUrl: string;
  template: string;
```

```
    styleUrls: string[];
    styles: string[];
    directives: Array<Type|any[]>;
    pipes: Array<Type|any[]>;
    encapsulation: ViewEncapsulation;
    animations: AnimationEntryMetadata[];
    interpolation: [string, string];
    constructor(..){} // just initialize above

  }
```

We note the comment:

```
  * You should most likely be using ComponentMetadata instead.
```

The directives.ts file exports classes related to directive metadata. They include:

- DirectiveMetadata (derives from InjectableMetadata)
- ComponentMetadata (derives from DirectiveMetadata)
- PipeMetadata (derives from InjectableMetadata)
- ImputMetadata
- OutputMetadata
- HostBindingMetadata
- HostListenerMetadata

The lifecycle_hooks.ts file defines an enum, an interface, a var and some simple abstract classes. The enum is:

```
  export enum LifecycleHooks {
    OnInit,
    OnDestroy,
    DoCheck,
    OnChanges,
    AfterContentInit,
    AfterContentChecked,
    AfterViewInit,
    AfterViewChecked
  }
```

The interface is:

```
  export interface SimpleChanges { [propName: string]: SimpleChange; }
```

The var is:

```
  export var LIFECYCLE_HOOKS_VALUES = [
    LifecycleHooks.OnInit, LifecycleHooks.OnDestroy, LifecycleHooks.DoCheck,
    LifecycleHooks.OnChanges, LifecycleHooks.AfterContentInit,
    LifecycleHooks.AfterContentChecked, LifecycleHooks.AfterViewInit,
    LifecycleHooks.AfterViewChecked
  ];
```

The abstract classes define the method signatures for handlers that application component interest in the lifecycle hooks must implement:

```
  export abstract class OnChanges {
       abstract ngOnChanges(changes: SimpleChanges): void; }
  export abstract class OnInit { abstract ngOnInit(): void; }
  export abstract class DoCheck { abstract ngDoCheck(): void; }
```

```
export abstract class OnDestroy { abstract ngOnDestroy(): void; }
export abstract class AfterContentInit {
     abstract ngAfterContentInit(): void; }
export abstract class AfterContentChecked {
     abstract ngAfterContentChecked(): void; }
export abstract class AfterViewInit {
     abstract ngAfterViewInit(): void; }
export abstract class AfterViewChecked {
     abstract ngAfterViewChecked(): void; }
```

core/src/animation

TBD

- animation_constants.ts
- animation_group_player.ts
- animation_keyframe.ts
- animation_player.ts
- animation_sequence_player.ts
- animation_style_util.ts
- animation_styles.ts
- metadata.ts
- view_animation_map.ts

core/src/profile

The profiel directory has two files:

- profile.ts
- wtf_init.ts
- wtf_impl.ts

WTF is the Web Tracing Framework:

- http://google.github.io/tracing-framework/

> *"The Web Tracing Framework is a collection of libraries, tools, and scripts aimed at web developers trying to write large, performance-sensitive Javascript applications. It's designed to be used in conjunction with the built-in development tools of browsers but goes far beyond what they usually support at the cost of some user setup time."*
> *from: http://google.github.io/tracing-framework/overview.html*

wtf_init.ts has this noop:

```
export function wtfInit() {}
```

wtf_impl.ts define the following interfaces:

```
export interface WtfScopeFn { (arg0?: any, arg1?: any): any; }
export interface Range {}
export interface Scope { (...args: any[] /** TODO #9100 */): any; }
interface Events {createScope(signature: string, flags: any): Scope;}
interface Trace {
  events: Events;
  leaveScope(scope: Scope, returnValue: any): any /** TODO #9100 */;
```

```
    beginTimeRange(rangeType: string, action: string): Range;
    endTimeRange(range: Range): any /** TODO #9100 */;
  }
  interface WTF { trace: Trace; }
```

It maintains two variables:

```
  var trace: Trace;
  var events: Events;
```

It defines some functions to work with those variables:

```
  export function createScope(signature: string, flags: any = null): any {
    return events.createScope(signature, flags);
  }
  export function leave<T>(scope: Scope, returnValue?: T): T {
    trace.leaveScope(scope, returnValue);
    return returnValue;
  }
  export function startTimeRange(rangeType: string, action: string): Range {
    return trace.beginTimeRange(rangeType, action);
  }
  export function endTimeRange(range: Range): void {
    trace.endTimeRange(range);
  }
```

Finally it has a detectWtf function:

```
  export function detectWTF(): boolean {
    var wtf: WTF = (global as any /** TODO #9100 */)['wtf'];
    if (wtf) {
      trace = wtf['trace'];
      if (trace) {
        events = trace['events'];
        return true;
      }
    }
    return false;
  }
```

The profile.ts file exports WTF-related variables:

```
  export var wtfEnabled = detectWTF();
  export var wtfCreateScope: (signature: string, flags?: any) => WtfScopeFn =
      wtfEnabled ? createScope : (signature: string, flags?: any) => noopScope;
  export var wtfLeave: <T>(scope: any, returnValue?: T) => T =
      wtfEnabled ? leave : (s: any, r?: any) => r;
  export var wtfStartTimeRange: (rangeType: string, action: string) => any =
      wtfEnabled ? startTimeRange : (rangeType: string, action: string)=> null;
  export var wtfEndTimeRange: (range: any) => void = wtfEnabled
                                               ? endTimeRange : (r: any) =>
  null;
```

core/src/reflection

The reflection directory has these files:

- platform_reflection_capabilities.ts
- reflection.ts

- reflection_capabilities.ts
- reflector.ts
- reflector_reader.ts
- types.ts

The reflector_reader.ts file defines the RelectorRader interface:

```
export abstract class ReflectorReader {
  abstract parameters(typeOrFunc: /*Type*/ any): any[][];
  abstract annotations(typeOrFunc: /*Type*/ any): any[];
  abstract propMetadata(typeOrFunc: /*Type*/ any): {[key: string]: any[]};
  abstract importUri(typeOrFunc: /*Type*/ any): string;
}
```

The types.ts file define these:

```
export type SetterFn = (obj: any, value: any) => void;
export type GetterFn = (obj: any) => any;
export type MethodFn = (obj: any, args: any[]) => any;
```

The reflection.ts file has this line:

```
export var reflector = new Reflector(new ReflectionCapabilities());
```

The platform_reflection_capabilities.ts file defines this interface:

```
export interface PlatformReflectionCapabilities {
  isReflectionEnabled(): boolean;
  factory(type: Type): Function;
  interfaces(type: Type): any[];
  hasLifecycleHook(type: any, lcInterface: any, lcProperty: string): boolean;
  parameters(type: any): any[][];
  annotations(type: any): any[];
  propMetadata(typeOrFunc: any): {[key: string]: any[]};
  getter(name: string): GetterFn;
  setter(name: string): SetterFn;
  method(name: string): MethodFn;
  importUri(type: any): string;
}
```

The reflection_capabilities.ts provides an implementation of that interface. We see the main part of reflection in the reflector.ts file, which defines the `Reflector` class, an implementation of `ReflectorReader`.

We see the use of reflection in `PLATFORM_COMMON_PROVIDERS` (core/src/platform_common_providers.ts), which is defined as:

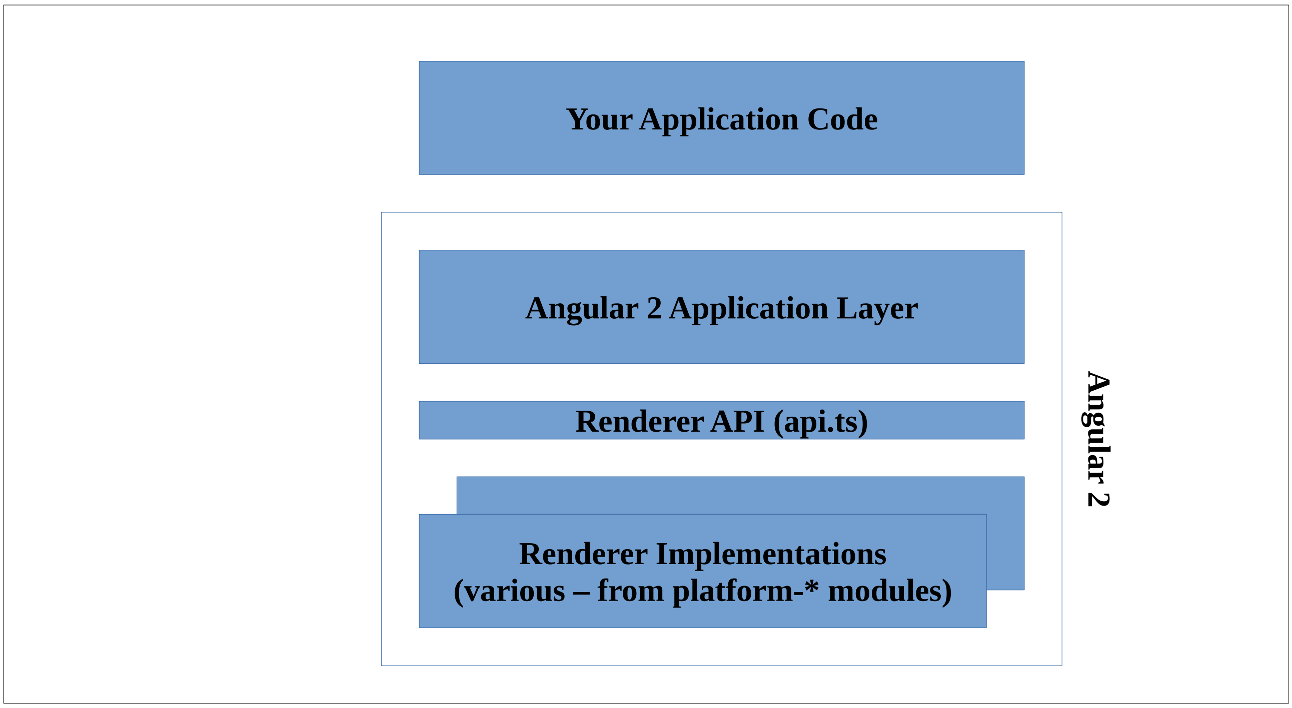
```
export const PLATFORM_COMMON_PROVIDERS: Array<any|Type|Provider|any[]> = [
   ...
   {provide: Reflector, useFactory: _reflector, deps: []},
   {provide: ReflectorReader, useExisting: Reflector},
   ...
];
```

core/src/render

This directory has just one file:

- api.ts

Layering for angular applications involves your application code talking to the Angular framework, which is layered into an application layer and a renderer layer, with a renderer API in between. The core/src/render/api.ts file defines this thin API and nothing else. The API consists of four classes - `RenderDebugInfo`, `RenderComponentType`, `Renderer` and `RootRenderer`. Implementation of this API is not part of core (though core/src/debug/debug_renderer.ts provides a debugging wrapper for it). Instead, the various platform modules need to provide the actual implementations for different scenarios.



Scenarios with diverse rendering requirements include:

- UI web apps in regular browser
- webworker apps
- server apps
- native apps for mobile devices
- testing

The Renderer API is defined in terms of elements – and provides functionality e.g. to create elements, set their properties and listen for their events. The Renderer API is not defined in terms of a DOM. Indeed, the term "DOM" is not part of any of the method names in this API (though it is mentioned in some comments). In that way, how rendering is provided is an internal implementation detail, easily replaced in different scenarios if needed. Obviously, for a web app running in the main UI thread in a regular browser, the platform used for that needs to implement the Renderer API in terms of the browser's DOM (and platform-browser does). But take a webworker as an alternative, where there simply is no browser DOM – a different platform needs to provide an alternative rendering solution. We will be examining in detail how rendering implementations work when we cover platforms later.

A notable characteristic of the Renderer API is that, even though it is defined in terms of elements, it does not list anywhere what those elements are. Elements are identified in terms of string names, but what are valid names is not part of the renderer. Instead, there is an element schema registry defined in the template compiler (@angular/compiler/src/schema) and we will examine it further when looking at the compiler.

Now we can move on to looking at the renderer API. RenderDebugInfo is used to provide debugging context information and is defined as:

```
export abstract class RenderDebugInfo {
  get injector(): Injector { return unimplemented(); }
  get component(): any { return unimplemented(); }
  get providerTokens(): any[] { return unimplemented(); }
  get references(): {[key: string]: any} { return unimplemented(); }
  get context(): any { return unimplemented(); }
  get source(): string { return unimplemented(); }
}
```

RenderComponentType is used to identify component types for which rendering is needed, and is defined as:

```
export class RenderComponentType {
  constructor(
     public id: string,
     public templateUrl: string,
     public slotCount: number,
     public encapsulation: ViewEncapsulation,
     public styles: Array<string|any[]>,
     public animations: {[key: string]: Function}) {}
}
```

The RootRenderer class is used to register a provider with dependency injection and is defined as:

```
export abstract class RootRenderer {
  abstract renderComponent(componentType: RenderComponentType): Renderer;
}
```

Essentially, its single method, renderComponent(), is used to request the root renderer to answer this - “for a given component type, please give me back a renderer that I can use”. This RootRenderer is key to wiring up flexible rendering of components via dependency injection. Two sample uses are in @angular/platform-browser/src/browser.ts, where browserModule is defined as:

```
@NgModule({
  providers: [
    {provide: RootRenderer, useExisting: DomRootRenderer},
    ... ],
  exports: [CommonModule, ApplicationModule]
}) export class BrowserModule { }
```

and in @angular/platform-browser/src/worker_app.ts, where WorkerAppModule is defined as:

```
@NgModule({
  providers: [
```

```
    {provide: RootRenderer, useExisting: WebWorkerRootRenderer},
    ...
  ], exports: [CommonModule, ApplicationModule]
}) export class WorkerAppModule { }
```

The result of this is that different root renderers can be supplied via dependency injection for differing scenarios, and client code using the renderer API can use a suitable implementation. If that is how the `RootRenderer` gets into dependency injection system, then of course the next question is, how does it get out? Surprisingly, there is no call to `injector.get(RootRenderer)` in the codebase. Instead, the `ViewUtils` class (in @angular/core/src/linker/view-utils.ts) is also registered with dependency injection and it takes a `RootRenderer` as a constructor parameter. `ViewUtils` is marked as `@injectable`, so when a request is made to the injector for a `ViewUtils`, then when it is constructed (in @angular/core/src/linker/component_factory.ts) it is supplied with a `RootRenderer` automatically by the injector:

```
export class ComponentFactory<C> {
  ...
  create(
      injector: Injector, projectableNodes: any[][] = null,
      rootSelectorOrNode: string|any = null): ComponentRef<C> {
    var vu: ViewUtils = injector.get(ViewUtils);
    if (isBlank(projectableNodes)) {
      projectableNodes = [];
    }
    // Note: Host views don't need a declarationAppElement!
    var hostView = this._viewFactory(vu, injector, null);
    var hostElement = hostView.create(
              EMPTY_CONTEXT, projectableNodes, rootSelectorOrNode);
    return new ComponentRef_<C>(hostElement, this._componentType);
  }
}
```

ViewUtils has this method:

```
renderComponent(renderComponentType: RenderComponentType): Renderer {
    return this._renderer.renderComponent(renderComponentType); }
```

To complete the picture, note the AppView<T> class (core/src/linker/view.ts) takes in a `ViewUtils` as a constructor parameter and initializes a `Renderer` variable using:

```
this.renderer = viewUtils.renderComponent(componentType);
```

When we examine code generated by the Angular template compiler, it defines a class derived from `AppView` and has many calls to the renderer. Here is a sample of Angular generated code from https://github.com/thelgevold/angular2-offline-compiler:

```
const parentRenderNode:any =
  this.renderer.createViewRoot(this.declarationAppElement.nativeElement);
this._el_0 = this.renderer.createElement(parentRenderNode,'h1',null);
this._text_1 = this.renderer.createText(this._el_0,'',null);
this._text_2 = this.renderer.createText(parentRenderNode,'\n\n',null);
this._el_3=this.renderer.createElement(parentRenderNode,'treeview',null);
```

We'll talk more about the actual use of the rendering API when examining view-related code in core/src/linker.

Now we'll move on to the principal class in the Renderer API, `Renderer`, which is abstract and declares the following methods:

selectRootElement createElement createViewRoot createTemplateAnchor createText projectNodes attachViewAfter	detachView destroyView listen listenGlobal setElementProperty setElementAttribute	setBindingDebugInfo setElementClass setElementStyle invokeElementMethod setText animate

`Renderer` in full is as follows:

```
export abstract class Renderer {
  abstract selectRootElement(
    selectorOrNode: string|any, debugInfo?: RenderDebugInfo): any;
  abstract createElement(
    parentElement: any, name: string, debugInfo?: RenderDebugInfo): any;
  abstract createViewRoot(hostElement: any): any;
  abstract createTemplateAnchor(
    parentElement: any, debugInfo?: RenderDebugInfo): any;
  abstract createText(
    parentElement: any, value: string, debugInfo?: RenderDebugInfo): any;
  abstract projectNodes(parentElement: any, nodes: any[]): void;
  abstract attachViewAfter(node: any, viewRootNodes: any[]): void;
  abstract detachView(viewRootNodes: any[]): void;
  abstract destroyView(hostElement: any, viewAllNodes: any[]): void;
  abstract listen(
    renderElement: any, name: string, callback: Function): Function;
  abstract listenGlobal(
    target: string, name: string, callback: Function): Function;
  abstract setElementProperty(
    renderElement: any, propertyName: string, propertyValue: any): void;
  abstract setElementAttribute(renderElement: any,
    attributeName: string, attributeValue: string): void;
  abstract setBindingDebugInfo(renderElement: any,
    propertyName: string, propertyValue: string):void;
  abstract setElementClass(
    renderElement: any, className: string, isAdd: boolean): void;
  abstract setElementStyle(
    renderElement: any, styleName: string, styleValue: string): void;
  abstract invokeElementMethod(
    renderElement: any, methodName: string, args?: any[]): void;
  abstract setText(renderNode: any, text: string): void;
  abstract animate(element: any, startingStyles: AnimationStyles,
    keyframes: AnimationKeyframe[], duration: number,
    delay: number, easing: string): AnimationPlayer;
}
```

Here only the interface is being defined – for actual implementation, refer to the various platform renderers in the different platform modules. The renderer is a simple abstraction, quite suitable for a variety of rendering engines. Let's look at four methods:

```
abstract createElement(
     parentElement: any, name: string, debugInfo?: RenderDebugInfo): any;
abstract setElementProperty(
     renderElement: any, propertyName: string, propertyValue: any): void;
abstract listen(
     renderElement: any, name: string, callback: Function): Function;
abstract invokeElementMethod(
     renderElement: any, methodName: string, args?: any[]): void;
```

The elements, their properties and their methods are identified by name (strings). The list of arguments to `invokeElementMethod()` is of type `any`, the `propertyValue` for `setElementProperty()` is any. The `parentElement` for `createElement()` is any. The `listen()` function attaches a callback to an event associated with an element.

core/src/debug

This directory contains two files:

- debug_node.ts
- debug_renderer.ts

The debug_node.ts file implements `EventListener`, `DebugNode` and `DebugElement` classes along with some helper functions. `EventListener` stores a name and a function, to be called after events are detected:

```
export class EventListener {
     constructor(public name: string, public callback: Function){}; }
```

The `DebugNode` class represents a node in a tree:

```
export class DebugNode {
  nativeNode: any;
  listeners: EventListener[];
  parent: DebugElement;
  constructor(nativeNode: any, parent: DebugNode,
                        private _debugInfo: RenderDebugInfo) {
    this.nativeNode = nativeNode;
    if (isPresent(parent) && parent instanceof DebugElement) {
      parent.addChild(this);
    } else {
      this.parent = null;
    }
    this.listeners = [];
  } … }
```

The debug node at attached as a child to the parent DebugNode. The `nativeNode` to which this `DebugNode` refers to is recorded. A private field, `_debugInfo` records the `RenderDebugInfo` (defined in core/src/render/api.ts) to be stored in this `DebugNode`.

The `DebugElement` class extends `DebugNode` and supplies a debugging representation of an element.

```
export class DebugElement extends DebugNode {
  name: string;
```

```
    properties: {[key: string]: any};
    attributes: {[key: string]: string};
    classes: {[key: string]: boolean};
    styles: {[key: string]: string};
    childNodes: DebugNode[];
    nativeElement: any;

    constructor(nativeNode: any, parent: any, _debugInfo: RenderDebugInfo) {
      super(nativeNode, parent, _debugInfo);
      this.properties = {};
      this.attributes = {};
      this.classes = {};
      this.styles = {};
      this.childNodes = [];
      this.nativeElement = nativeNode;
    }
```

It includes these for adding and removing children:

```
    addChild(child: DebugNode) {
      if (isPresent(child)) {
        this.childNodes.push(child);
        child.parent = this;
      }
    }
    removeChild(child: DebugNode) {
      var childIndex = this.childNodes.indexOf(child);
      if (childIndex !== -1) {
        child.parent = null;
        this.childNodes.splice(childIndex, 1);
      }
    }
```

It includes this for events:

```
  triggerEventHandler(eventName: string, eventObj: any) {
      this.listeners.forEach((listener) => {
        if (listener.name == eventName) { listener.callback(eventObj); }
      });
    }
```

The functions manage a map:

```
  // Need to keep the nodes in a global Map so that multiple
  // angular apps are supported.
  var _nativeNodeToDebugNode = new Map<any, DebugNode>();
```

This is used to add a node:

```
  export function indexDebugNode(node: DebugNode) {
    _nativeNodeToDebugNode.set(node.nativeNode, node);
  }
```

The debug_renderer.ts file contains two classes, `DebugDomRootRender` and `DebugDomRenderer`. `DebugDomRootRender` returns a debugging renderer for a component:

```
  export class DebugDomRootRenderer implements RootRenderer {
    constructor(private _delegate: RootRenderer) {}
```

```
  renderComponent(componentProto: RenderComponentType): Renderer {
    return new DebugDomRenderer(
                   this._delegate.renderComponent(componentProto));
  }
}
```

The constructor of `DebugDomRenderer` takes in a delegate renderer as a parameter, and in its implementation of the other `Renderer` methods, it ultimately passes on the renderer request to that delegate:

```
export class DebugDomRenderer implements Renderer {
  constructor(private _delegate: Renderer) {}
```

So `DebugDomRenderer`'s `createElement()` method will call the delegate's `createElement()` method, and also create a `DebugElement`:

```
createElement(parentElement:any,name:string,debugInfo?:RenderDebugInfo):any {
    var nativeEl = this._delegate.createElement(
                                       parentElement, name, debugInfo);
    var debugEl = new DebugElement(nativeEl,
                                       getDebugNode(parentElement),
debugInfo);
    debugEl.name = name;
    indexDebugNode(debugEl);
    return nativeEl;
  }
```

Invocations of an element's method is performed by:

```
invokeElementMethod(renderElement: any, methodName: string, args?: any[]) {
    this._delegate.invokeElementMethod(renderElement, methodName, args);
}
```

core/src/change_detection

The change_detection directory has these files:

- change_detection.ts
- change_detection_util.ts
- change_detector_ref.ts
- constants.ts
- pipe_transform.ts

The pipe_transform.ts file defines the PipeTransform interface, needed for pipes:

```
export interface PipeTransform{ transform(value: any, ...args: any[]): any; }
```

The constants.ts file defines two enums for change detection:

```
export enum ChangeDetectionStrategy {
  OnPush,
  Default,
}
export enum ChangeDetectorStatus {
  CheckOnce,
  Checked,
  CheckAlways,
  Detached,
```

```
  Errored,
  Destroyed,
}
```

It also defines these variables:

```
export var CHANGE_DETECTION_STRATEGY_VALUES = [
  ChangeDetectionStrategy.OnPush,
  ChangeDetectionStrategy.Default,
];
export var CHANGE_DETECTOR_STATUS_VALUES = [
  ChangeDetectorStatus.CheckOnce,
  ChangeDetectorStatus.Checked,
  ChangeDetectorStatus.CheckAlways,
  ChangeDetectorStatus.Detached,
  ChangeDetectorStatus.Errored,
  ChangeDetectorStatus.Destroyed,
];
```

Definitions in change_detection_utils.ts include:

```
export function devModeEqual(a: any, b: any): boolean {..}
export class WrappedValue {..}
export class ValueUnwrapper {..}
export class SimpleChange {..}
```

The change_detector_ref.ts file defines the ChangeDetectorRef class:

```
export abstract class ChangeDetectorRef {
  abstract markForCheck(): void;
  abstract detach(): void;
  abstract detectChanges(): void;
  abstract checkNoChanges(): void;
  abstract reattach(): void;
}
```

The change_detection.ts file defines:

```
export const keyValDiff: KeyValueDifferFactory[] =
                                          [new
DefaultKeyValueDifferFactory()];
export const iterableDiff: IterableDifferFactory[] =
                                          [new
DefaultIterableDifferFactory()];
export const defaultIterableDiffers = new IterableDiffers(iterableDiff);
export const defaultKeyValueDiffers = new KeyValueDiffers(keyValDiff);
```

core/src/change_detection/differs

- default_iterable_differ.ts
- default_keyvalue_differ.ts
- iterable_differs.ts
- keyvalue_differs.ts

TBD

core/src/zone

This directory contains these files:

- ng_zone.ts
- ng_zone_impl.dart
- ng_zone_impl.ts

The small zone.js library (https://github.com/angular/zone.js) provides a way of managing execution context when using asynchronous tasks. Angular has a dependency on it, and we see its use in core/src/zone.

The ng_zone_impl.ts file implements the NgzoneImpl class. It has inner and outer Zone fields:

```
private outer: Zone;
private inner: Zone;
```

It has a static method:

```
static isInAngularZone(): boolean {
                        return Zone.current.get('isAngularZone') === true; }
```

It has three run methods:

```
runInner(fn: () => any): any { return this.inner.run(fn); };
runInnerGuarded(fn: () => any): any { return this.inner.runGuarded(fn); };
runOuter(fn: () => any): any { return this.outer.run(fn); };
```

It’s constructor is when the zones are configured. It starts by setting both outer and inner zones to the current zone.

```
this.outer = this.inner = Zone.current;
```

If wtfZoneSpec or longStackTraceZoneSpec zones are required, it forks as needed:

```
if ((Zone as any)['wtfZoneSpec']) {
  this.inner = this.inner.fork((Zone as any)['wtfZoneSpec']);
}
if (trace && (Zone as any)['longStackTraceZoneSpec']) {
  this.inner = this.inner.fork((Zone as any)['longStackTraceZoneSpec']);
}
```

Regardless of the result of above, it always forks once more, passing parameters:

- name: ‘angular’
- properties: <any>{'isAngularZone': true},
- handlers for: onInvokeTask, onInvoke, onHasTask, onHandleError

The ng_zone.ts file defines the NgZone class. This provides simple implementations of the following properties:

```
get onUnstable(): EventEmitter<any> { return this._onUnstable; }
get onMicrotaskEmpty(): EventEmitter<any> { return this._onMicrotaskEmpty; }
get onStable(): EventEmitter<any> { return this._onStable; }
get onError(): EventEmitter<any> { return this._onErrorEvents; }
get isStable(): boolean { return this._isStable; }
get hasPendingMicrotasks(): boolean { return this._hasPendingMicrotasks; }
get hasPendingMacrotasks(): boolean { return this._hasPendingMacrotasks; }
```

It also provides these methods:

```
run(fn: () => any): any { return this._zoneImpl.runInner(fn); }
runGuarded(fn: () => any): any { return this._zoneImpl.runInnerGuarded(fn); }
runOutsideAngular(fn: () => any): any { return this._zoneImpl.runOuter(fn); }
```

core/src/testability

The testability directory contains one file:

- testability.ts

Its provides functionality for testing Angular components, and includes use of NgZone.

It exports two injectable classes, TestabilityRegistry and Testability. TestabilityRegistry maintains a map from any element to an instance of a testability. It provides a registerApplication() method which allows an entry to be added to this map, and a getTestability() method that is a lookup.

```
export class TestabilityRegistry {
  /** @internal */
  _applications = new Map<any, Testability>();
  constructor() { _testabilityGetter.addToWindow(this); }
  registerApplication(token: any, testability: Testability) {
    this._applications.set(token, testability);
  }
  getTestability(elem: any): Testability {
          return this._applications.get(elem); }
  getAllTestabilities(): Testability[] {
          return MapWrapper.values(this._applications); }
  getAllRootElements(): any[] { return MapWrapper.keys(this._applications); }
  findTestabilityInTree(
       elem: Node, findInAncestors: boolean = true): Testability {
    return _testabilityGetter.findTestabilityInTree(
       this, elem, findInAncestors);
  }
}
```

The Testability class is structured as follows:

```
@Injectable()
export class Testability
constructor(private _ngZone: NgZone) { this._watchAngularEvents(); }
_watchAngularEvents(): void {}
increasePendingRequestCount(): number {..}
decreasePendingRequestCount(): number {..}
isStable(): boolean {..}
_runCallbacksIfReady(): void {..}
whenStable(callback: Function): void {..}
  getPendingRequestCount(): number {..}
  findBindings(using: any, provider: string, exactMatch: boolean): any[] {..}
  findProviders(using: any, provider: string, exactMatch: boolean): any[]
{..}
}
```

To get a testabllity, this getter interface is supplied:

```
export interface GetTestability {
  addToWindow(registry: TestabilityRegistry): void;
```

```
    findTestabilityInTree(registry: TestabilityRegistry,
                 elem: any, findInAncestors: boolean): Testability;
  }
```

It maintains a variable as an instance of this:

```
var _testabilityGetter: GetTestability
```

Which is set

```
export function setTestabilityGetter(getter: GetTestability): void {
  _testabilityGetter = getter;
}
```

Now let's look at how testabillity is tied in with the rest of the framework. Recall that `PLATFORM_COMMON_PROVIDERS` (core/src/platform_common_providers.ts) is defined as:

```
export const PLATFORM_COMMON_PROVIDERS: Array<any|Type|Provider|any[]> = [
  ..
  TestabilityRegistry,
  ..
];
```

Also recall that the constructor for `ApplicationRef` is:

```
constructor(
      ..
      @Optional() private _testabilityRegistry: TestabilityRegistry,
      @Optional() private _testability: Testability, ..) {}
```

There is a class `BrowserGetTestability` (platform-browser/src/browser/testability.ts) that implements `GetTestability` with a static `init()` method that calls `setTestabilityGetter()`:

```
export class BrowserGetTestability implements GetTestability {
  static init() { setTestabilityGetter(new BrowserGetTestability()); }
  ..
}
```

core/src/linker

The linker is uses to define an API for the template compiler and is instrumental in how compiled components work together. The core/index.ts file has this line:

```
export * from './src/linker';
```

The core/src/linker.ts file lists the exports:

```
export {AppModuleFactory, AppModuleRef} from './linker/app_module_factory';
export {AppModuleFactoryLoader} from './linker/app_module_factory_loader';
export {Compiler,CompilerFactory,CompilerOptions, ComponentStillLoadingError}
            from './linker/compiler';
export {ComponentFactory, ComponentRef} from './linker/component_factory';
export {ComponentFactoryResolver, NoComponentFactoryError}
            from './linker/component_factory_resolver';
export {ComponentResolver} from './linker/component_resolver';
export {DynamicComponentLoader} from './linker/dynamic_component_loader';
export {ElementRef} from './linker/element_ref';
export {ExpressionChangedAfterItHasBeenCheckedException}
            from './linker/exceptions';
```

```
export {QueryList} from './linker/query_list';
export {SystemJsAppModuleLoader}
            from './linker/system_js_app_module_factory_loader';
export {SystemJsCmpFactoryResolver, SystemJsComponentResolver}
            from './linker/systemjs_component_resolver';
export {TemplateRef} from './linker/template_ref';
export {ViewContainerRef} from './linker/view_container_ref';
export {EmbeddedViewRef, ViewRef} from './linker/view_ref';
```

The source files in core/src/linker implement these exports:

- app_module_factory.ts
- app_module_factory_loader.ts
- compiler.ts
- component_factory.ts
- component_factory_resolver.ts
- component_resolver.ts
- debug_context.ts
- dynamic_component_loader.ts
- element.ts
- element_injector.ts
- element_ref.ts
- exceptions.ts
- query_list.ts
- system_js_app_module_factory_loader.ts
- systemjs_component_resolver.ts
- template_ref.ts
- view.ts
- view_container_ref.ts
- view_ref.ts
- view_type.ts
- view_utils.ts

`CompilerOptions` provides a list of configuration options (all are marked as optional) for a compiler:

```
export type CompilerOptions = {
  useDebug?: boolean,
  useJit?: boolean,
  defaultEncapsulation?: ViewEncapsulation,
  providers?: any[],
  deprecatedAppProviders?: any[]
}
```

`CompilerFactory` is an abstract class used to construct a compiler, via its `createCompiler()` abstract method:

```
abstract createCompiler(options?: CompilerOptions): Compiler;
```

Its `mergeOptions()` method takes two parameters, `defaultOptions` and `newOptions`, both of type `CompilerOptions`, and returns a `CompilerOptions` instance which merges the two inputs.

```
static mergeOptions(defaultOptions: CompilerOptions = {},
                    newOptions: CompilerOptions = {}): CompilerOptions {
  return {
    useDebug: _firstDefined(newOptions.useDebug, defaultOptions.useDebug),
    useJit: _firstDefined(newOptions.useJit, defaultOptions.useJit),
    defaultEncapsulation:
        _firstDefined(newOptions.defaultEncapsulation,
                              defaultOptions.defaultEncapsulation),
    providers: _mergeArrays(defaultOptions.providers,
                              newOptions.providers),
    deprecatedAppProviders:
        _mergeArrays(defaultOptions.deprecatedAppProviders,
                              newOptions.deprecatedAppProviders)
  };
}
```

It also has a `withDefaults()` method, which is used later from `bootstrap()`:

```
withDefaults(options: CompilerOptions = {}): CompilerFactory {
    return new _DefaultApplyingCompilerFactory(this, options);
  }
```

This uses the simple `_DefaultApplyingCompilerFactory` class to allow use fo default options:

```
class _DefaultApplyingCompilerFactory extends CompilerFactory {
  constructor(private _delegate: CompilerFactory,
              private _options: CompilerOptions) { super(); }
  createCompiler(options: CompilerOptions = {}): Compiler {
    return this._delegate.createCompiler(
       CompilerFactory.mergeOptions(this._options, options));
  }
}
```

The `ComponentStillLoadingError` exception has a clear message:

```
export class ComponentStillLoadingError extends BaseException {
  constructor(public compType: Type) {
    super(`Can't compile synchronously as ${stringify(compType)}
                                                  is still being loaded!`);
  }
}
```

The `Compiler` class is the base class for compilers, and it is these derived classes where the actual template compilation occurs.

```
export class Compiler {
  get injector(): Injector {
    throw new BaseException(`Runtime compiler is not loaded.
                                                  Tried to read the
injector.`);
  }
  compileComponentAsync<T>(component: ConcreteType<T>)
                              : Promise<ComponentFactory<T>> {
    throw new BaseException(
        `Runtime compiler is not loaded. Tried to compile
                                                  ${stringify(component)}`);
  }
  compileComponentSync<T>(component: ConcreteType<T>): ComponentFactory<T> {
```

```
      throw new BaseException(
          `Runtime compiler is not loaded. Tried to compile
                                                   ${stringify(component)}`);
    }
    compileAppModuleSync<T>(moduleType: ConcreteType<T>,
                            metadata: AppModuleMetadata = null):
        AppModuleFactory<T> {
      throw new BaseException(
          "Runtime compiler is not loaded. Tried to compile
                                                   ${stringify(moduleType)}");
    }
    compileAppModuleAsync<T>(moduleType: ConcreteType<T>,
                            metadata: AppModuleMetadata = null):
        Promise<AppModuleFactory<T>> {
      throw new BaseException(
          `Runtime compiler is not loaded. Tried to compile
                                                   ${stringify(moduleType)}`);
    }
    clearCache(): void {}
    clearCacheFor(type: Type) {}
  }
```

The four compile generic methods either synchronously or asynchronously compile an individual component, or an entire ngmodule. All take a `ConcreteType` as a parameter. The async versions returns the promise of a factory, whereas the sync versions return the actual factory. The core module does not contain an actual compiler implementation – you will find it in the compiler module. Let's figure out how this is called from within your application code when you call `bootstrap()`. We will start in compiler/src/compiler.ts:

```
@Injectable()
export class _RuntimeCompilerFactory extends CompilerFactory
```

We will examine that class in detail when covering the compiler module. That is the actual compiler that our code uses. The same file has:

```
export const RUNTIME_COMPILER_FACTORY = new _RuntimeCompilerFactory();
```

When covering the platform-browser-dynamic module, we will see this being used to define `BROWSER_DYNAMIC_COMPILER_FACTORY`:

```
export const BROWSER_DYNAMIC_COMPILER_FACTORY =
                        RUNTIME_COMPILER_FACTORY.withDefaults(
                          {providers: [{provide: XHR, useClass: XHRImpl}]});
```

We will cover that XHR parameter later when examining Platform-browser-dynamic/src/xhr, but all we need to know now is a comment from its source file:

```
An interface for retrieving documents by URL that the compiler uses to load
templates
```

`BROWSER_DYNAMIC_COMPILER_FACTORY` in turn is used in `BROWSER_DYNAMIC_PLATFORM_PROVIDERS`:

```
export const BROWSER_DYNAMIC_PLATFORM_PROVIDERS: Array<any> = [
  BROWSER_PLATFORM_PROVIDERS,
  {provide: CompilerFactory, useValue: BROWSER_DYNAMIC_COMPILER_FACTORY},
  {provide: PLATFORM_INITIALIZER, useValue: initReflector, multi: true},
```

```
];
```

Which in turn is passed to `browserDynamicPlatform()`:

```
export const browserDynamicPlatform = createPlatformFactory(
                'browserDynamic', BROWSER_DYNAMIC_PLATFORM_PROVIDERS);
```

Which in turn is used inside the `bootstrap()` method in a call to the `bootstrapModule()` method:

```
export function bootstrap<C>{
  ...
  return bootstrapModule(.., browserDynamicPlatform(),..});
}
```

Application code calls `bootstrap()` to launch.

<application code>
bootstrap (platform-browser-dynamic)
bootstrapModule (platform-browser-dynamic)
platformBrowserDynamic (platform-browser-dynamic)
INTERNAL_BROWSER_DYNAMIC_PLATFORM_PROVIDERS (platform-browser-dynamic)
BROWSER_DYNAMIC_COMPILER_FACTORY (platform-browser-dynamic)
RUNTIME_COMPILER_FACTORY (compiler)
_RuntimeCompilerFactory (compiler) ← The factory for the actual compiler
CompilerFactory (core) ← The API

The exceptions.ts file defines exceptions used by the template compilation process:

```
export class ExpressionChangedAfterItHasBeenCheckedException
                                                    extends BaseException {
  constructor(oldValue: any, currValue: any, context: any) {
    let msg =
        `Expression has changed after it was checked. Previous value:
                   '${oldValue}'. Current value: '${currValue}'.`;
    if (oldValue === UNINITIALIZED) {
      msg += ` It seems like the view has been created after its parent ` +
             ` and its children have been dirty checked.` +
             ` Has it been created in a change detection hook ?`;
    }
    super(msg);
```

```
    }
  }
```

Two other exceptions are `ViewWrappedException` and `ViewDestroyedException`.

The query-list.ts file defines the generic `QueryList<M>` class, which provides controlled access to a map.

The `ElementRef` class is used as a reference to the actual rendered element, what that is depends on the renderer. In general, application developers are advised not to work directly with `ElementRef`s, as it makes their code renderer-specific, and may introduce security issues.

```
export class ElementRef {
  public nativeElement: any;
  constructor(nativeElement: any) { this.nativeElement = nativeElement; }
}
```

The view_type.ts file defines the `ViewType` enum:

```
export enum ViewType {
  // A view that contains the host element with bound component directive.
  // Contains a COMPONENT view
  HOST,
  // The view of the component
  // Can contain 0 to n EMBEDDED views
  COMPONENT,
  // A view that is embedded into another View via a <template> element
  // inside of a COMPONENT view
  EMBEDDED
}
```

The view_ref.ts file defines the `ViewRef` and `EmbeddedViewRef` abstract classes, and defines the `ViewRef_` concrete class.

ViewRef is defined as:

```
export abstract class ViewRef {
  get destroyed(): boolean { return <boolean>unimplemented(); }

  abstract onDestroy(callback: Function): any /** TODO #9100 */;
}
```

`EmbeddedViewRef` is defined as:

```
export abstract class EmbeddedViewRef<C> extends ViewRef {
  get context(): C { return unimplemented(); }
  get rootNodes(): any[] { return <any[]>unimplemented(); };
  abstract destroy(): void;
}
```

`ViewRef_` is defined as:

```
export class ViewRef_<C> implements EmbeddedViewRef<C>, ChangeDetectorRef {
  /** @internal */
  _originalMode: ChangeDetectorStatus;

  constructor(private _view: AppView<C>) {
    this._view = _view;
```

```
    this._originalMode = this._view.cdMode;
  }

  get internalView(): AppView<C> { return this._view; }

  get rootNodes(): any[] { return this._view.flatRootNodes; }

  get context() { return this._view.context; }

  get destroyed(): boolean { return this._view.destroyed; }

  markForCheck(): void { this._view.markPathToRootAsCheckOnce(); }
  detach(): void { this._view.cdMode = ChangeDetectorStatus.Detached; }
  detectChanges(): void { this._view.detectChanges(false); }
  checkNoChanges(): void { this._view.detectChanges(true); }
  reattach(): void {
    this._view.cdMode = this._originalMode;
    this.markForCheck();
  }

  onDestroy(callback: Function) { this._view.disposables.push(callback); }

  destroy() { this._view.destroy(); }
}
```

The element_injector.ts file

TBD

The debug_context.ts file

TBD

The view_container_ref.ts file defines the abstract ViewContainerRef class and the concrete ViewContainerRef_ class. ViewContainerRef is a container for views. It defines four getters – element (for the anchor element of the container), injector, parentInjector and length (number of views attached to container), that in the default implementation all throw unimplemented exceptions. It defines two important create methods – createEmbeddedView and createComponent, which create the two variants of views supported. Finally, it has a few helper methods – clear, get, insert, indexOf, remove and detach – which work on the views within the container.

```
export abstract class ViewContainerRef {

  get element(): ElementRef { return <ElementRef>unimplemented(); }
  get injector(): Injector { return <Injector>unimplemented(); }
  get parentInjector(): Injector { return <Injector>unimplemented(); }
  get length(): number { return <number>unimplemented(); };

  abstract createEmbeddedView<C>(templateRef: TemplateRef<C>,
       context?: C, index?: number): EmbeddedViewRef<C>;
  abstract createComponent<C>(
       componentFactory: ComponentFactory<C>, index?: number,
       injector?: Injector, projectableNodes?: any[][]): ComponentRef<C>;
```

```
    abstract clear(): void;
    abstract get(index: number): ViewRef;
    abstract insert(viewRef: ViewRef, index?: number): ViewRef;
    abstract indexOf(viewRef: ViewRef): number;
    abstract remove(index?: number): void;
    abstract detach(index?: number): ViewRef;
  }
```

The difference between createEmbeddedView and createComponent is that the former takes a template ref as a parameter and creates an embedded view from it, whereas the latter takes a component factory as a parameter and uses the host view of the newly created component.

ViewContainerRef_ implements ViewContainerRef. It takes an AppElement for its anchor element in its constructor. Its getters use that AppElement.

```
  export class ViewContainerRef_ implements ViewContainerRef {
    constructor(private _element: AppElement) {}
    get(index: number): ViewRef {
       return this._element.nestedViews[index].ref; }
    get length(): number {
      var views = this._element.nestedViews;
      return isPresent(views) ? views.length : 0;
    }
    get element(): ElementRef { return this._element.elementRef; }
    get injector(): Injector { return this._element.injector; }
    get parentInjector(): Injector { return this._element.parentInjector; }
    ..
  }
```

The implementations of insert, indexOf, remove, detach result in use of AppElement's nestedViews and use of its attachView and detachView methods. The two create methods are implemented as follows:

```
    createEmbeddedView<C>(templateRef: TemplateRef<C>, context: C = null,
         index: number = -1): EmbeddedViewRef<C> {
      var viewRef: EmbeddedViewRef<any> =
                               templateRef.createEmbeddedView(context);
      this.insert(viewRef, index);
      return viewRef;
    }

    /** @internal */
    _createComponentInContainerScope: WtfScopeFn =
       wtfCreateScope('ViewContainerRef#createComponent()');

    createComponent<C>(componentFactory: ComponentFactory<C>,
         index: number = -1, injector: Injector = null,
         projectableNodes: any[][] = null): ComponentRef<C> {
      var s = this._createComponentInContainerScope();
      var contextInjector =
           isPresent(injector) ? injector : this._element.parentInjector;
      var componentRef = componentFactory.create(
                                           contextInjector, projectableNodes);
      this.insert(componentRef.hostView, index);
      return wtfLeave(s, componentRef);
    }
```

We see the difference between them – `createEmbeddedView()` calls the `TemplateRef`'s `createEmbeddedView()` method and inserts the result viewRef; whereas `createComponent()` calls the component factory's create method, and with the resulting `ComponentRef`, inserts its `HostView`.

The view.ts file has a var, an abstract class, a concrete class and a function. The var is for the web tracing framework (WTF):

```
var _scope_check: WtfScopeFn = wtfCreateScope(`AppView#check(ascii id)`);
```

The internal function `_findLastRenderNode()` take a node as a parameter. If that node is not an `AppElement`, it is returned as the result. Otherwise, if it has no nested views, the `nativeElement` of the `AppElement` is returned. Otherwise, `_findLastRenderNode()` is called again for each nested view.

```
function _findLastRenderNode(node: any): any {
  var lastNode: any;
  if (node instanceof AppElement) {
    var appEl = <AppElement>node;
    lastNode = appEl.nativeElement;
    if (isPresent(appEl.nestedViews)) {
      // Note: Views might have no root nodes at all!
      for (var i = appEl.nestedViews.length - 1; i >= 0; i--) {
        var nestedView = appEl.nestedViews[i];
        if (nestedView.rootNodesOrAppElements.length > 0) {
          lastNode = _findLastRenderNode(nestedView.rootNodesOrAppElements[
                   nestedView.rootNodesOrAppElements.length - 1]);
        }
      }
    }
  } else {
    lastNode = node;
  }
  return lastNode;
}
```

The abstract class is `AppView`. When the Angular template compiler is generating code for your templates, it creates an outer `AppView` with `ViewType` set to `ViewType.HOST`, and an inner AppView with ViewType set to VieType.COMPONENT.

AppView

ViewRef

AppElement

ViewType (HOST, COMPONENT, EMBEDDED)

ViewUtils

VieContainerRef

It has getters for:

- destroyed
- changeDetectorRef
- flatRootNodes
- lastRootNode

It has methods for:

- cancelActiveAnimation
- queueAnimation
- triggerQueuedAnimations
- create
- createInternal
- init
- selectOrCreateHostElement
- injectorGet
- injectorGetInternal
- injector
- destroy
- _destroyRecurse
- destroyLocal
- destroyInternal
- detachInternal
- detach
- dirtyParentQueriesInternal
- detectChanges
- detectChangesInternal
- detectContentChildrenChanges
- detectViewChildrenChanges
- addToContentChildren
- removeFromContentChildren
- markAsCheckOnce
- markPathToRootAsCheckOnce
- eventHandler
- throwDestroyedError

The concrete class is `DebugAppView`. This wraps calls to the underlying method in a try-catch block. Let's take `detach()` as an example:

```
detach(): void {
    this._resetDebug();
    try {
      super.detach();
    } catch (e) {
      this._rethrowWithContext(e, e.stack);
      throw e;
    }
  }
```

The component_factory.ts file exports three classes – `ComponentRef`, `ComponentRef_` and `ComponentFactory`.

```
export class ComponentFactory<C> {
  constructor(
       public selector: string,
       private _viewFactory: Function,
       private _componentType: Type) {}

  get componentType(): Type { return this._componentType; }

  create(injector: Injector, projectableNodes: any[][] = null,
              rootSelectorOrNode: string|any = null): ComponentRef<C> {
    var vu: ViewUtils = injector.get(ViewUtils);
    if (isBlank(projectableNodes)) {
      projectableNodes = [];
    }
    // Note: Host views don't need a declarationAppElement!
    var hostView = this._viewFactory(vu, injector, null);
    var hostElement = hostView.create(
                EMPTY_CONTEXT, projectableNodes, rootSelectorOrNode);
    return new ComponentRef_<C>(hostElement, this._componentType);
  }
}
```

The `ComponentRef` abstract class is defined as:

```
export abstract class ComponentRef<C> {
  get location(): ElementRef { return unimplemented(); }
  get injector(): Injector { return unimplemented(); }
  get instance(): C { return unimplemented(); };
  get hostView(): ViewRef { return unimplemented(); };
  get changeDetectorRef(): ChangeDetectorRef { return unimplemented(); }
  get componentType(): Type { return unimplemented(); }
  abstract destroy(): void;
  abstract onDestroy(callback: Function): void;
}
```

The `ComponentRef_` concrete class extends `ComponentRef`. Its constructors takes in an `AppElement`, and as we see its methods and getters are implemented with calls to it:

```
export class ComponentRef_<C> extends ComponentRef<C> {
  constructor(private _hostElement: AppElement, private _componentType: Type)
      { super(); }
  get location(): ElementRef { return this._hostElement.elementRef; }
  get injector(): Injector { return this._hostElement.injector; }
  get instance(): C { return this._hostElement.component; };
  get hostView(): ViewRef { return this._hostElement.parentView.ref; };
  get changeDetectorRef(): ChangeDetectorRef {
     return this._hostElement.parentView.ref; };
  get componentType(): Type { return this._componentType; }
  destroy(): void { this._hostElement.parentView.destroy(); }
  onDestroy(callback: Function): void { this.hostView.onDestroy(callback); }
}
```

The template_ref.ts file defines the `TemplateRef` abstract class and the `TemplateRef_` concrete class.

```
export abstract class TemplateRef<C> {
  get elementRef(): ElementRef { return null; }
  abstract createEmbeddedView(context: C): EmbeddedViewRef<C>;
```

```
}
```

The `TemplateRef_` implementation has a constructor that takes a view factroy function, and its `createEmbeddedView()` method calls this to create the app view, and then calls the app view's `create`:

```
export class TemplateRef_<C> extends TemplateRef<C> {
  constructor(
     private _appElement: AppElement,
     private _viewFactory: Function) { super(); }

  createEmbeddedView(context: C): EmbeddedViewRef<C> {
    var view: AppView<C> = this._viewFactory(
        this._appElement.parentView.viewUtils,
          this._appElement.parentInjector, this._appElement);
    if (isBlank(context)) {
      context = <any>EMPTY_CONTEXT;
    }
    view.create(context, null, null);
    return view.ref;
  }

  get elementRef(): ElementRef { return this._appElement.elementRef; }
}
```

The element.ts file defines the `AppElement` class:

```
/**
 * An AppElement is created for elements that have a ViewContainerRef,
 * a nested component or a <template> element to keep data around
 * that is needed for later instantiations.
 */
export class AppElement {
  public nestedViews: AppView<any>[] = null;
  public componentView: AppView<any> = null;
  public component: any;
  public componentConstructorViewQueries: QueryList<any>[];

  constructor(
       public index: number,
       public parentIndex: number,
       public parentView: AppView<any>,
       public nativeElement: any) {}
```

`AppElement` defines methods to attach and detach views:

```
  attachView(view: AppView<any>, viewIndex: number) {
    if (view.type === ViewType.COMPONENT) {
      throw new BaseException(`Component views can't be moved!`);
    }
    var nestedViews = this.nestedViews;
    if (nestedViews == null) {
      nestedViews = [];
      this.nestedViews = nestedViews;
    }
    ListWrapper.insert(nestedViews, viewIndex, view);
    var refRenderNode: any;
    if (viewIndex > 0) {
```

```
        var prevView = nestedViews[viewIndex - 1];
        refRenderNode = prevView.lastRootNode;
      } else {
        refRenderNode = this.nativeElement;
      }
      if (isPresent(refRenderNode)) {
        view.renderer.attachViewAfter(refRenderNode, view.flatRootNodes);
      }
      view.addToContentChildren(this);
    }

    detachView(viewIndex: number): AppView<any> {
      var view = ListWrapper.removeAt(this.nestedViews, viewIndex);
      if (view.type === ViewType.COMPONENT) {
        throw new BaseException(`Component views can't be moved!`);
      }
      view.detach();
      view.removeFromContentChildren(this);
      return view;
    }
```

The component_factory_resolver.ts file defines the `ComponentFactoryResolver` abstract class and `CodegenComponentFactoryResolver` concrete class.

`ComponentFactoryResolver` is defined as:

```
  export abstract class ComponentFactoryResolver {
    static NULL: ComponentFactoryResolver =
                     new _NullComponentFactoryResolver();
    abstract resolveComponentFactory<T>(
                component: ConcreteType<T>): ComponentFactory<T>;
  }
```

`CodegenComponentFactoryResolver` is defined as:

```
  export class CodegenComponentFactoryResolver
                              implements ComponentFactoryResolver {
    private _factories = new Map<any, ComponentFactory<any>>();
    constructor(
       factories: ComponentFactory<any>[],
       private _parent: ComponentFactoryResolver) {
      for (let i = 0; i < factories.length; i++) {
        let factory = factories[i];
        this._factories.set(factory.componentType, factory);
      }
    }
    resolveComponentFactory<T>(
         component: {new (...args: any[]): T}): ComponentFactory<T> {
      let result = this._factories.get(component);
      if (!result) {
        result = this._parent.resolveComponentFactory(component);
      }
      return result;
    }
  }
```

`CodegenComponentFactoryResolver` has a private map field, _factories, and its constructor takes an array, factories. Don’t mix them up!! the constructor iterates over the array (factories) and for each item, adds an entry to the map (_factories) that maps the factory's `componentType` to the factory.

In its `resolveComponentFactory()` method, `CodegenComponentFactoryResolver` looks up the map for a matching factory, and if present, returns it, otherwise returns the result of a call to the parent's `resolveComponentFactory`.

The app_module_factory_loader.ts file defines the `AppModuleFactoryLoader` abstract class as:

```
 export abstract class AppModuleFactoryLoader {
  abstract load(path: string): Promise<AppModuleFactory<any>>;
}
```

It is use for lazy loading. We will see an implementation in system_js_app_module_factory_loader.ts.

The system_js_app_module_factory_loader.ts file defines the `SystemJsAppModuleLoader` class, which loads NgModule factories using SystemJS. Its constructor takes a `_compiler` as an optional parameter. In its load method, if _compiler was provided, it calls `loadAndCompile()`, otherwise it calls `loadFactory()`. It is within `loadAndCompile()` that `_compiler.compileAppModuleAsync()` is called.

```
@Injectable()
export class SystemJsAppModuleLoader implements AppModuleFactoryLoader {
  constructor(@Optional() private _compiler: Compiler) {}

  load(path: string): Promise<AppModuleFactory<any>> {
    return this._compiler ? this.loadAndCompile(path) :
                                                this.loadFactory(path);
  }

  private loadAndCompile(path: string): Promise<AppModuleFactory<any>> {
    let [module, exportName] = path.split(_SEPARATOR);
    if (exportName === undefined) exportName = 'default';

    return (<any>global)
        .System.import(module)
        .then((module: any) => module[exportName])
        .then((type: any) => checkNotEmpty(type, module, exportName))
        .then((type: any) => this._compiler.compileAppModuleAsync(type));
  }

  private loadFactory(path: string): Promise<AppModuleFactory<any>> {
    let [module, exportName] = path.split(_SEPARATOR);
    if (exportName === undefined) exportName = 'default';

    return (<any>global)
        .System.import(module + FACTORY_MODULE_SUFFIX)
        .then((module: any) => module[exportName + FACTORY_CLASS_SUFFIX])
        .then((factory: any) => checkNotEmpty(factory, module, exportName));
  }
}
```

The `SystemJsNgModuleLoader` class is used in the router module, to define the `ROUTER_PROVIDERS` array. The router/src/router_module.ts file has this:

```
export const ROUTER_PROVIDERS: any[] = [
  ..
  {provide: NgModuleFactoryLoader, useClass: SystemJsNgModuleLoader},
  {provide: ROUTER_CONFIGURATION, useValue: {enableTracing: false}}
];
```

The app_module_factory.ts file defines the `AppModuleRef` and `AppModuleInjector` abstract classes and the `AppModuleFactory` concrete class.

`AppModuleRef` is defined as:

```
export abstract class AppModuleRef<T> {
  get injector(): Injector { return unimplemented(); }
  get componentFactoryResolver(): ComponentFactoryResolver {
                                                  return unimplemented(); }
  get instance(): T { return unimplemented(); }
}
```

`AppModuleInjector` can be used as an `ComponentFactoryResolver`, `Injector`, `AppModuleRef`. Note it is abstract – its `createInternal()` and `getInternal()` methods are abstract. It is defined as:

```
export abstract class AppModuleInjector<T> extends
CodegenComponentFactoryResolver implements Injector, AppModuleRef<T> {
  public instance: T;
  constructor(public parent: Injector, factories: ComponentFactory<any>[]) {
    super(factories, parent.get(ComponentFactoryResolver,
                                          ComponentFactoryResolver.NULL));
  }
  create() { this.instance = this.createInternal(); }
  abstract createInternal(): T;
  get(token: any, notFoundValue: any = THROW_IF_NOT_FOUND): any {
    if (token === Injector || token === ComponentFactoryResolver) {
      return this;
    }
    var result = this.getInternal(token, _UNDEFINED);
    return result ===
         _UNDEFINED ? this.parent.get(token, notFoundValue) : result;
  }
  abstract getInternal(token: any, notFoundValue: any): any;
  get injector(): Injector { return this; }
  get componentFactoryResolver(): ComponentFactoryResolver { return this; }
}
```

`AppModuleFactory` takes in an injector class in its constructor and in its create method uses that injector to create the `AppModuleRef`. It is defined as:

```
export class AppModuleFactory<T> {
  constructor(private _injectorClass:
       {new (parentInjector: Injector): AppModuleInjector<T>},
      private _moduleype: ConcreteType<T>) {}
  get moduleType(): ConcreteType<T> { return this._moduleype; }
  create(parentInjector: Injector = null): AppModuleRef<T> {
    if (!parentInjector) { parentInjector = Injector.NULL; }
    var instance = new this._injectorClass(parentInjector);
    instance.create();
    return instance;
  }
```

```
}
```

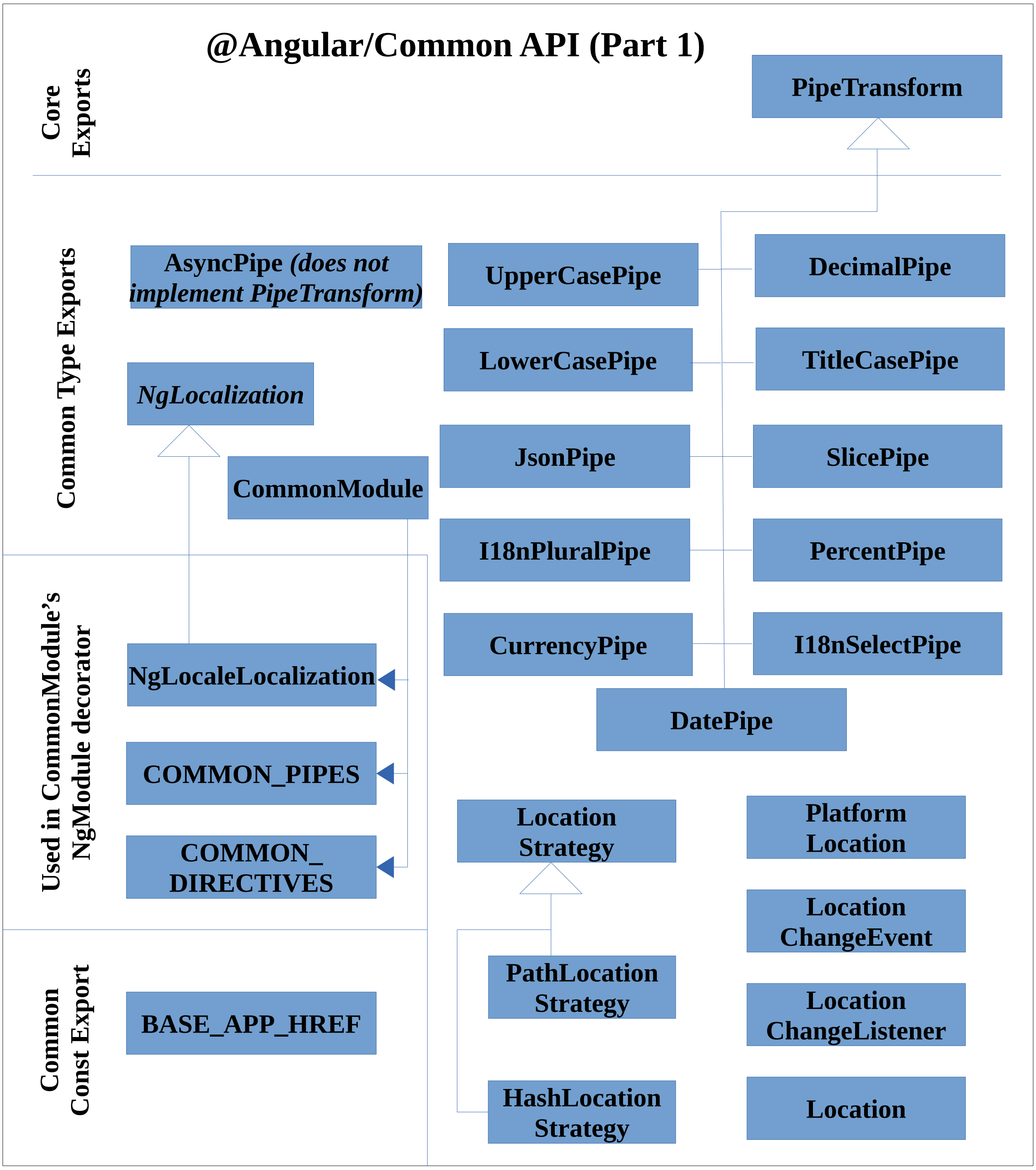
6: @Angular/Common

Overview

The Common package builds on top of the Core package and adds some shared functionality in areas such as directives, location and pipes.

@Angular/Common API

The exported API of the @Angular/Common package can be represented as:



Note that `AsyncPipe`, unlike all the other pipes, does not implement the `PipeTransform` interface. However, it does implement the `transform()` method.

`PipeTransform` is defined in:

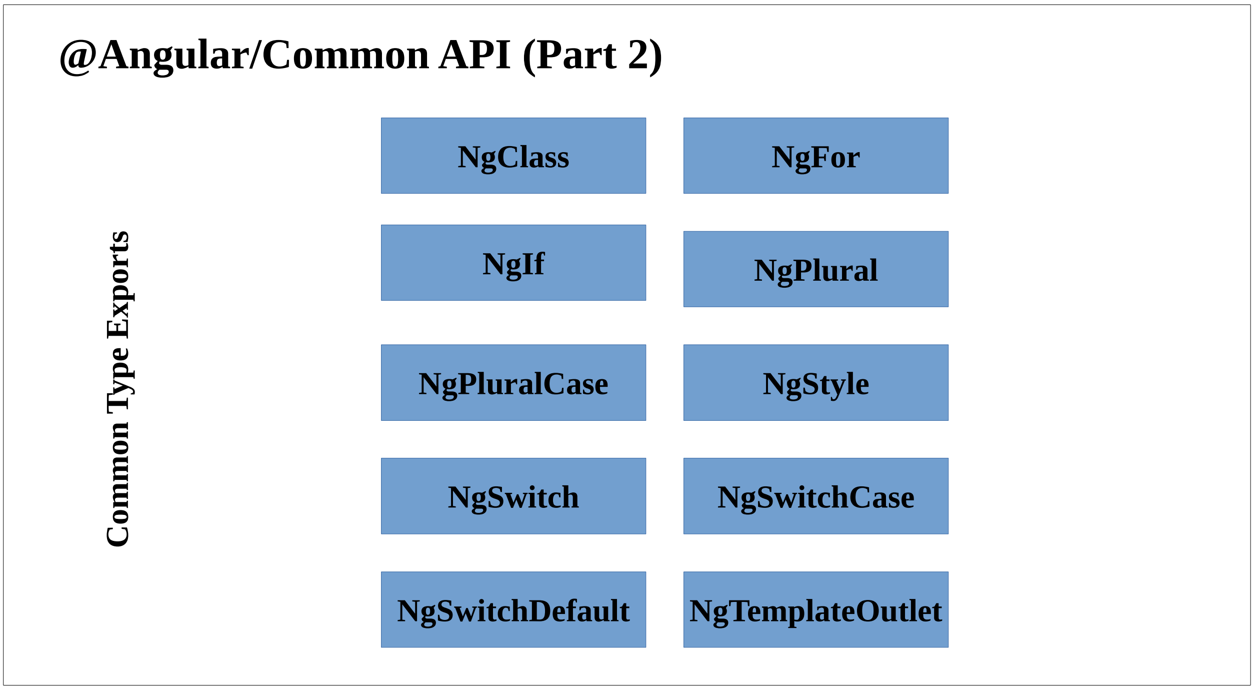
- <ANGULAR-MASTER>/modules/@angular/core/src/change_detection/pipe_transform.ts

as:

```
export
interface PipeTransform { transform(value: any, ...args: any[]): any; }
```

The async_pipe.ts implementation of `transform` is:

```
transform(obj: Observable<any>|Promise<any>|EventEmitter<any>): any {
```



The export API is defined by:

- <ANGULAR-MASTER>/modules/@angular/common/index.ts

simply as:

```
export * from './src/common';
```

The src/common.ts file contains:

```
export * from './location/index';
export {NgLocaleLocalization, NgLocalization} from './localization';
export {CommonModule} from './common_module';
export {NgClass, NgFor, NgIf, NgPlural, NgPluralCase, NgStyle, NgSwitch,
  NgSwitchCase, NgSwitchDefault, NgTemplateOutlet} from './directives/index';
export {AsyncPipe, DatePipe, I18nPluralPipe, I18nSelectPipe, JsonPipe,
  LowerCasePipe, CurrencyPipe, DecimalPipe, PercentPipe, SlicePipe,
```

```
  UpperCasePipe, TitleCasePipe} from './pipes/index';
export {VERSION} from './version';
export {Version} from '@angular/core';
```

This exports the contents of the pipes/directives/location files in src.

The exported location types are listed in:

- <ANGULAR-MASTER>/modules/@angular/common/src/location/index.ts

as follows:

```
export * from './platform_location';
export * from './location_strategy';
export * from './hash_location_strategy';
export * from './path_location_strategy';
export * from './location';
```

platform_location.ts exports the `PlatformLocation` class and the `LocationChangeEvent` and `LocationChangeListener` intefaces. location_strategy.ts exports the `LocationStrategy` class and the `APP_BASE_HREF` const – this is important for routing, for more details refer to the discussion of "*Set the <base href>*" on this page:

- https://angular.io/docs/ts/latest/guide/router.html

The other location files just export a class with the same name as the file.

Source Tree Layout

The source tree for the Common package contains these directories:

- src
- test (unit tests in Jasmine)
- testing (test tooling)

and these files:

- index.ts
- package.json
- rollup.config.js
- rollup-testing.config.js
- testing.ts
- tsconfig-build.json
- tsconfig-testing.json

The tsconfig.json content is:

```
{
  "compilerOptions": {
    "baseUrl": ".",
    "declaration": true,
    "stripInternal": true,
    "experimentalDecorators": true,
    "module": "es2015",
    "moduleResolution": "node",
    "outDir": "../../../dist/packages-dist/common",
    "paths": {
```

```
        "@angular/core": ["../../../dist/packages-dist/core"]
      },
      "rootDir": ".",
      "sourceMap": true,
      "inlineSources": true,
      "target": "es5",
      "skipLibCheck": true,
      "lib": [ "es2015", "dom" ]
    },
    "files": [
      "index.ts",
      "../../../node_modules/zone.js/dist/zone.js.d.ts"
    ],
    "angularCompilerOptions": {
      "annotateForClosureCompiler": true,
      "strictMetadataEmit": true
    }
  }
```

It list as files to build as index.ts and brings in zone.js.d.ts.

Source

common/src

The common/src directory has these source files:

- common.ts
- common_module.ts
- localization.ts
- version.ts

The common_module.ts file defines the CommonModule type, which contains common declarations, exports and providers:

```
   @NgModule({
    declarations: [COMMON_DIRECTIVES, COMMON_PIPES],
    exports: [COMMON_DIRECTIVES, COMMON_PIPES],
    providers: [
      {provide: NgLocalization, useClass: NgLocaleLocalization},
    ],
  })
  export class CommonModule {}
```

The localization.ts file provides localization functionality, mainly in the area of plurals. It defines the `NgLocalization` abstract class, the `NgLocaleLocalization` concrete class and the `getPluralCategory()` function.

```
export abstract class NgLocalization {
  abstract getPluralCategory(value: any): string;
}
```

It declares a single abstract method, `getPluralCategory()`.

The `getPluralCategory()` function calls `NgLocalization.getPluralCategory()` to get the plural category for a value. We note the value passed to the function is of type number, whereas that passed to NgLocalization.getPluralCategory() is of type any:

```
export function getPluralCategory(
    value: number, cases: string[], ngLocalization: NgLocalization): string {
  let key = `=${value}`;

  if (cases.indexOf(key) > -1) {
    return key;
}
```

One implementation of NgLocalization is provided, called NgLocaleLocalization, which takes in a localeId in its constructor:

```
export class NgLocaleLocalization extends NgLocalization {
  constructor(@Inject(LOCALE_ID) protected locale: string) { super(); }
```

As we have just seen with CommonModule, this is what its NgModule decorator uses as the provider for NgLocalization. Its implementation of getPluralCategory() uses CLDR-based code, generated by this script:

- <ANGULAR-MASTER>/script/cldr/gen_plural_rules.js

Cldr

Cldr stands for Unicode's Common Locale Data Repository:

- http://http://cldr.unicode.org/

and is described as:

> *"The Unicode CLDR provides key building blocks for software to support the world's languages, with the largest and most extensive standard repository of locale data available."*

The cldr npm package:

- https://www.npmjs.com/package/cldr

is described as:

"A module that allows you to extract a bunch of locale-specific information from the Unicode CLDR (Common Localization Data Repository), including:

- Date, time, and date-time formats
- Date interval formats
- Number formats, symbols, and digits for all number systems
- Exemplar and ellipsis characters
- Day names, month names, quarter names, era names, and cyclic names
- Patterns for rendering lists of items
- Display names for languages, time zones, territories, scripts and currencies
- Plural rule functions (converted to JavaScript functions)
- Rule-based number formatting functions (converted to JavaScript functions)"

The gen_plural_rules.js script at:

- <ANGULAR-MASTER>/scripts/cldr/cldr.js

uses the cldr npm package to general plural information and places it in:

<ANGULAR-MASTER>/modules/@angular/common/src/localization.ts

common/src/directives

This directory has the following source files:

- index.ts
- ng_class.ts
- ng_for.ts
- ng_if.ts
- ng_plural.ts
- ng_style.ts
- ng_switch.ts
- ng_template_outlet.ts

The indexs.ts file exports the directive types, along with a definition for `COMMON_DIRECTIVES`, which is:

```
export const COMMON_DIRECTIVES: Provider[] = [
  NgClass,
  NgFor,
  NgIf,
  NgTemplateOutlet,
  NgStyle,
  NgSwitch,
  NgSwitchCase,
  NgSwitchDefault,
  NgPlural,
  NgPluralCase,
];
```

We saw its use in the `NgModule` decorator attached to `CommonModule`.

The various ng_ files implement the directives. Lets take a peek at one example, ng_if.ts. It uses a view container to create an embedded view based on a template ref, if the supplied condition is true. We first see in its constructor it records the view container and template ref passed in as parameter:

```
@Directive({selector: '[ngIf]'})
export class NgIf {
  private _context: NgIfContext = new NgIfContext();
  private _thenTemplateRef: TemplateRef<NgIfContext> = null;
  private _elseTemplateRef: TemplateRef<NgIfContext> = null;
  private _thenViewRef: EmbeddedViewRef<NgIfContext> = null;
  private _elseViewRef: EmbeddedViewRef<NgIfContext> = null;

  constructor(
       private _viewContainer: ViewContainerRef,
               templateRef: TemplateRef<NgIfContext>) {
    this._thenTemplateRef = templateRef;
  }
```

we observe that the `NgIf` class does not derive from any other class. It is made into a directive by using the `Directive` decorator.

Then we see it has four setters defined as input properties, for the variations of if:

```
@Input()
set ngIf(condition: any) {
  this._context.$implicit = condition;
  this._updateView();
}

@Input()
set ngIfThen(templateRef: TemplateRef<NgIfContext>) {
  this._thenTemplateRef = templateRef;
  this._thenViewRef = null;  // clear previous view if any.
  this._updateView();
}

@Input()
set ngIfElse(templateRef: TemplateRef<NgIfContext>) {
  this._elseTemplateRef = templateRef;
  this._elseViewRef = null;  // clear previous view if any.
  this._updateView();
}
```

Finally it has an internal method, `_updateView`, where the view is changed as needed:

```
  private _updateView() {
    if (this._context.$implicit) {
      if (!this._thenViewRef) {
        this._viewContainer.clear();
        this._elseViewRef = null;
        if (this._thenTemplateRef) {
          this._thenViewRef =
              this._viewContainer.createEmbeddedView(
                    this._thenTemplateRef, this._context);
        }
      }
    } else {
      if (!this._elseViewRef) {
        this._viewContainer.clear();
        this._thenViewRef = null;
        if (this._elseTemplateRef) {
          this._elseViewRef =
              this._viewContainer.createEmbeddedView(
                   this._elseTemplateRef, this._context);
        }
      }
    }
  }
}
```

The important call here is to this._viewContainer.createEmbeddedView , where the embedded view is created if the NgIf condition is true.

If `NgIf` creates an embedded view zero or once, then we expect `NgFor` to create embedded view zero or more times, depends on the count supplied to `NgFor`. We see this is exactly the case, when we look at ng_for.ts, which implements the `NgFor` class (and two helper classes – `NgForRow` and `RecordViewTuple`). The helper classes are implemented as:

```
export class NgForRow {
  constructor(
     public $implicit: any, public index: number, public count: number) {}
  get first(): boolean { return this.index === 0; }
  get last(): boolean { return this.index === this.count - 1; }
  get even(): boolean { return this.index % 2 === 0; }
  get odd(): boolean { return !this.even; }
}
class RecordViewTuple {
  constructor(public record: any, public view: EmbeddedViewRef<NgForRow>) {}
}
```

The first thing to note about `NgFor`'s implementation is the class implements `DoCheck` and `OnChanges` lifecycle.

```
@Directive({selector: '[ngFor][ngForOf]'})
export class NgFor implements DoCheck, OnChanges { }
```

The `DoCheck` class is a lifecycle hook defined in @angular/core/src/metadata/lifecycle_hooks.ts as:

```
export abstract class DoCheck { abstract ngDoCheck(): void; }
```

`OnChanges` is defined in the same file as:

```
export abstract class OnChanges {
       abstract ngOnChanges(changes: SimpleChanges): void; }
```

Hence we would expect `NgFor` to provide `ngDoCheck` and `ngOnChanges` methods and it does. `ngDoCheck()` calls `_applyChanges`, where for each change operation a call to `viewContainer.createEmbeddedView()` is made.

common/src/location

This directory contains these source files:

- hash_location_strategy.ts
- location.ts
- location_strategy.ts
- path_location_strategy.ts
- platform_location.ts

The location_strategy.ts file defines the `LocationStrategy` class and the `APP_BASE_HREF` opaque token.

```
export abstract class LocationStrategy {
  abstract path(includeHash?: boolean): string;
  abstract prepareExternalUrl(internal: string): string;
  abstract pushState(
     state: any, title: string, url: string, queryParams: string): void;
  abstract replaceState(
     state: any, title: string, url: string, queryParams: string): void;
  abstract forward(): void;
  abstract back(): void;
  abstract onPopState(fn: LocationChangeListener): void;
  abstract getBaseHref(): string;
}
```

The opaque token is defined as:

```
export const APP_BASE_HREF: OpaqueToken = new OpaqueToken('appBaseHref');
```

The two implementations of LocationStrategy are provided in hash_location_strategy.ts and path_location_strategy.ts. Both share the same constructor signature:

```
@Injectable()
export class HashLocationStrategy extends LocationStrategy {
  private _baseHref: string = '';
  constructor(
      private _platformLocation: PlatformLocation,
      @Optional() @Inject(APP_BASE_HREF) _baseHref?: string) {
    super();
    if (isPresent(_baseHref)) {
      this._baseHref = _baseHref;
    }
  }
```

The PlatformLocation parameter is how they access actual location information.

PlatformLocation is an abstract class used to access location (URL) information. Note PlatformLocation does not extend Location - which is a service class used to manage the browser's URL. There have quite distinct purposes.

```
export abstract class PlatformLocation { .. }
```

An important part of the various (browser, server) platform representations is to provide a custom implementation of `PlatformLocation`.

The final file in common/src/location is location.ts, which is where Location is defined. The `Location` class is a service (perhaps it would be better to actually call it `LocationService`) used to interact with URLs. A note in the source is important:

```
* Note: it's better to use {@link Router#navigate} service to trigger route
* changes. Use `Location` only if you need to interact with or create
* normalized URLs outside of routing.
```

The `Location` class does not extend any other class, and its constructor only takes a `LocationStrategy` as as parameter:

```
@Injectable()
export class Location {
  /** @internal */
  _subject: EventEmitter<any> = new EventEmitter();
  /** @internal */
  _baseHref: string;
  /** @internal */
  _platformStrategy: LocationStrategy;

  constructor(platformStrategy: LocationStrategy) {
    this._platformStrategy = platformStrategy;
    const browserBaseHref = this._platformStrategy.getBaseHref();
    this._baseHref =
      Location.stripTrailingSlash(_stripIndexHtml(browserBaseHref));
    this._platformStrategy.onPopState((ev) => {
      this._subject.emit({
```

```
        'url': this.path(true),
        'pop': true,
        'type': ev.type,
      });
    });
  }
```

One useful method is `subscribe()`, which allows your application code to be informed of `popState` events:

```
subscribe(
      onNext: (value: any) => void,
      onThrow: (exception: any) => void = null,
      onReturn: () => void = null): Object {
   return this._subject.subscribe(
      {next: onNext, error: onThrow, complete: onReturn});
 }
```

common/src/pipes

This sub-directory contains these source files:

- async_pipe.ts
- case_conversion_pipes.ts
- date_pipe.ts
- i18n_plural_pipe.ts
- i18n_select_pipe.ts
- invalid_pipe_argument_error.ts
- json_pipe.ts
- number_pipe.ts
- slice_pipe.ts

All the pipe classes are marked with the `Pipe` decorator. Apart from AsyncPipe, all the other pipes implement the PipeTransform interface (and even AsyncPipe implements the transform method). As an example, slice_pipe.ts has the following:

```
@Pipe({name: 'slice', pure: false})
export class SlicePipe implements PipeTransform {
  transform(value: any, start: number, end?: number): any {
    if (isBlank(value)) return value;
    if (!this.supports(value)) {
      throw new InvalidPipeArgumentError(SlicePipe, value);
    }
    return value.slice(start, end);
  }
  private supports(obj: any): boolean {
     return typeof obj === 'string' || Array.isArray(obj); }
}
```

`COMMON_PIPES` (in index.ts) lists the defined pipes and will often be used when creating components.

```
export const COMMON_PIPES = [
  AsyncPipe,
  UpperCasePipe,
  LowerCasePipe,
  JsonPipe,
```

```
    SlicePipe,
    DecimalPipe,
    PercentPipe,
    TitleCasePipe,
    CurrencyPipe,
    DatePipe,
    I18nPluralPipe,
    I18nSelectPipe,
  ];
```

We saw its use in the `NgModule` decorator attached to `CommonModule`.

Finally, `InvalidPipeArgumentError` extends `BaseError`:

```
export class InvalidPipeArgumentError extends BaseError {
  constructor(type: Type<any>, value: Object) {
    super(`Invalid argument '${value}' for pipe '${stringify(type)}'`);
  }
}
```

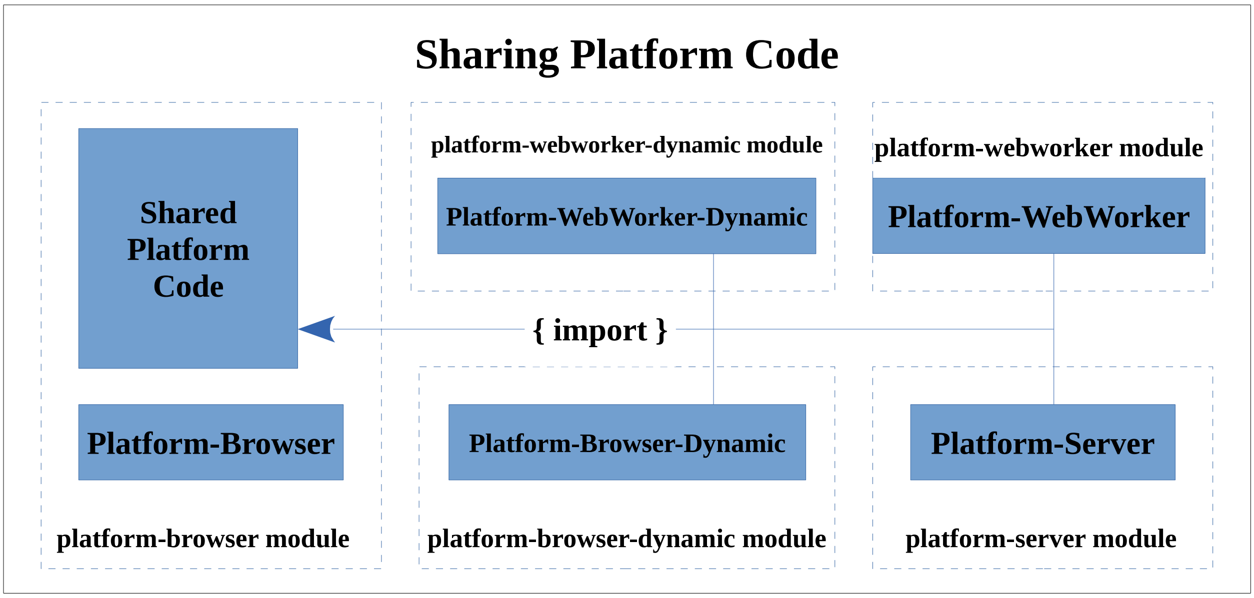
7: @Angular/Platform-Browser

Overview

A platform module is how an application interacts with its hosting environment. Duties of a platform include rendering (deciding what is displayed and how), multitasking (webworkers), security sanitization of URLs/html/style (to detect dangerous constructs) and location management (the URL displayed in the browser).

We have seen how the Core module provides a rendering API, but it includes no implementations of renderers and no mention of the DOM. All other parts of Angular that need to have content rendered talk to this rendering API and rely on an implementation to actually deal with the content to be "displayed" (and what "displayed" means varies depending on the platform). You will find an implementation of renderer in the platform modules – the main one is `DomRenderer`. Note that this renders to a DOM adapter (and multiple of those exist), but Core and all the features sitting on top of Core only know about the rendering API, not the DOM.

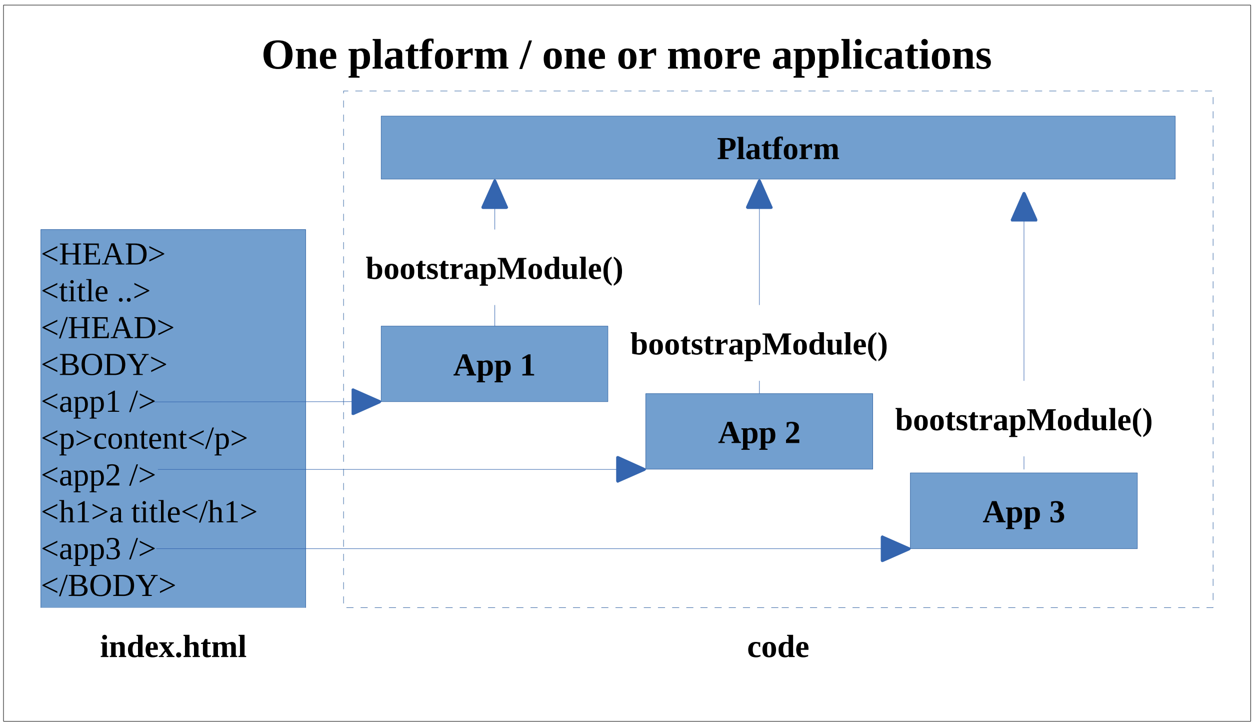
Angular supplies five platform modules: platform-browser (runs in the browser's main UI thread and uses the offline template compiler), platform-browser-dynamic (runs in the browser's main UI thread and uses the runtime template compiler), platform-webworker (runs in a webworker and uses the offline template compiler), platform-webworker-dynamic (runs in a webworker and uses the runtime template compiler) and platform-server (runs in the server and can uses either the offline or runtime template compiler). Shared functionality relating to platforms is in platform-browser and imported by the other platform modules. So platform-browser is a much bigger module compared to the other platform modules.



In this chapter we will explore platform-browser and we will cover the others in the subsequent chapters.

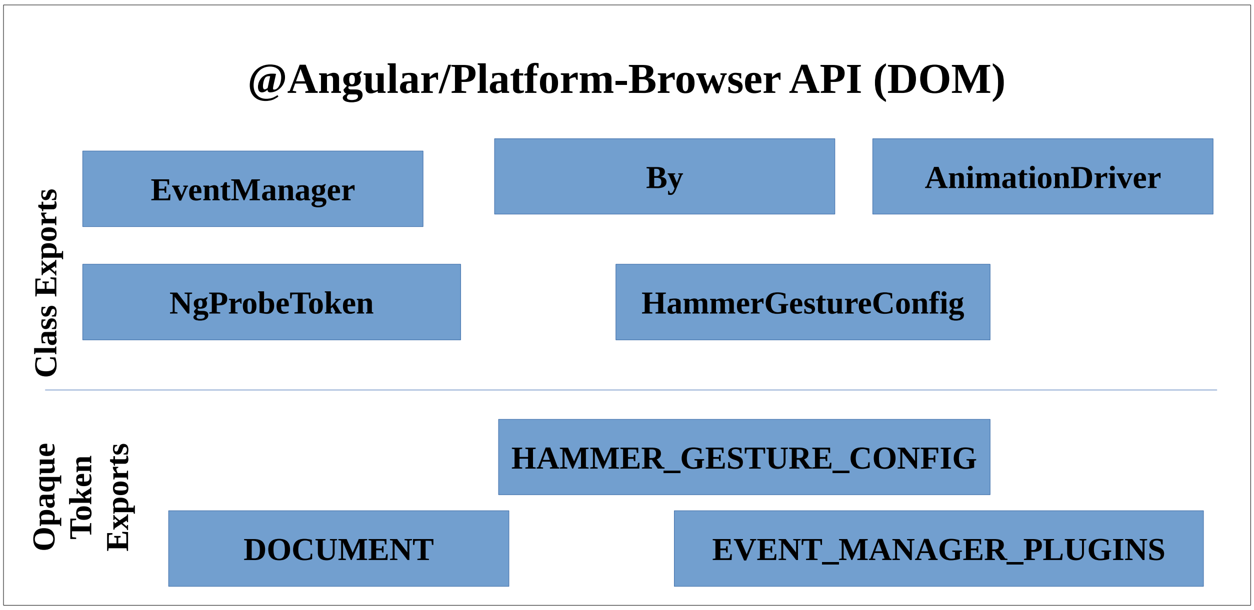
Platform-browser is how application code can interact with the browser when running in the main UI thread and assuming the offline template compiler has been used to pre-generate a module factory. For production use, platform-browser is likely to be the platform of choice, as it results in the fastest display (no in-browser template compilation needed) and smallest download size (the Angular template compiler does not have to be downloaded to the browser).

There is exactly one platform instance per thread (main browser UI thread or webworker). Multiple applications may run in the same thread, and they interact with the same platform instance.



Platform-Browser API

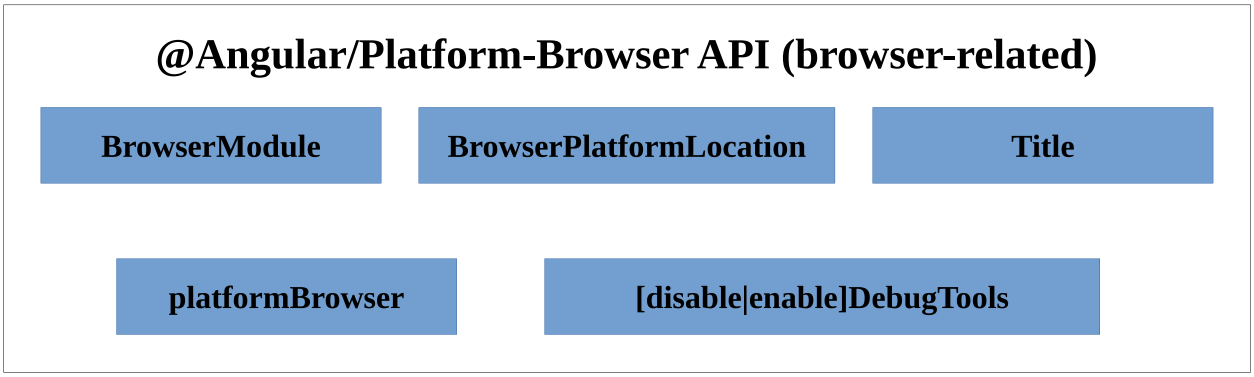
The platform-browser API can be sub-divided into four functional areas: browser-related, DOM, secruity and webworkers. The DOM related API can be represented as:



Hammer is for touch input. Event Manager handles the flow of events. NgProbeToken is used for debugging and is exposed in the browser's developer tools.

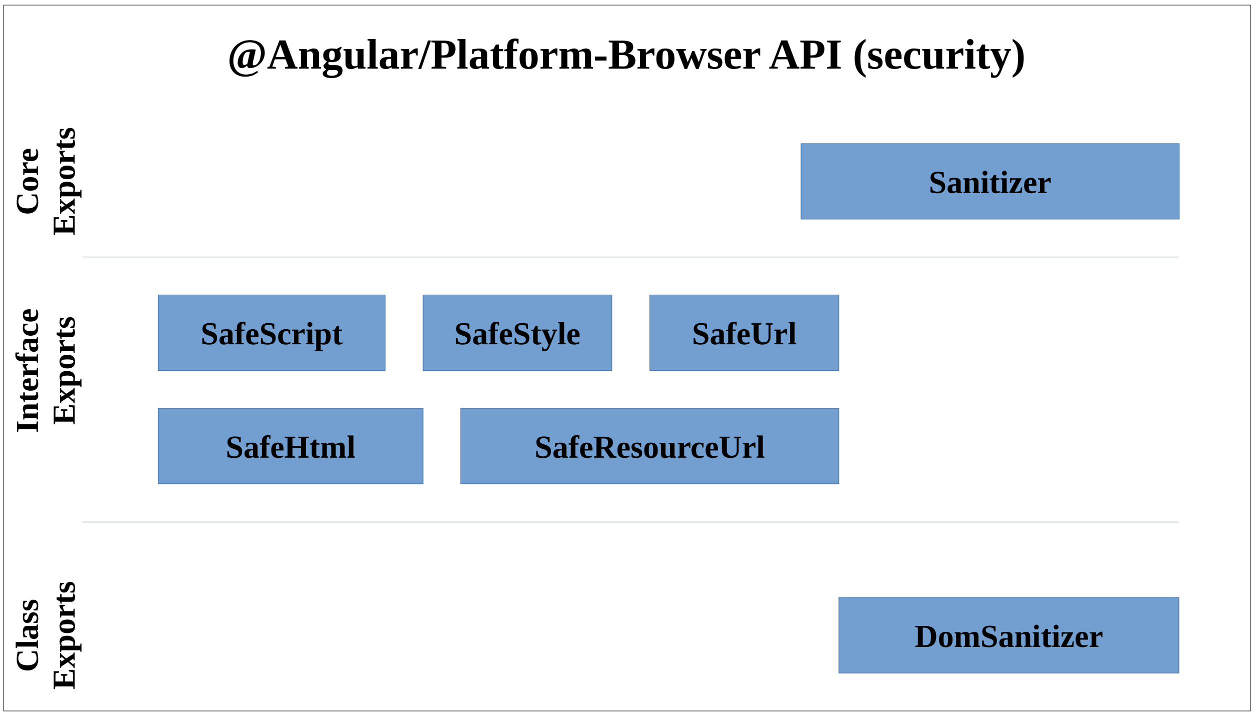
Two important classes - DomRenderer and DomAdapter - are not publically exported. Instead, they are registered with the dependency injector and that is how they are made available to application code (we will shortly examine both in detail).

The browser-related API can be represented as:



This covers the NgModule, the createPlatformFactory function, location information, a title helper class, and functions to enable and disable debug tooling.

The security related API can be represented as:



The root directory of this module has:

- index.ts

which has the single line:

```
export * from './src/platform-browser';
```

If we look at src/platform-browser.ts, we see the exported API of the platform-browser package:

```
export {BrowserModule, platformBrowser} from './browser';
export {Title} from './browser/title';
export {disableDebugTools, enableDebugTools} from './browser/tools/tools';
export {AnimationDriver} from './dom/animation_driver';
export {By} from './dom/debug/by';
export {NgProbeToken} from './dom/debug/ng_probe';
export {DOCUMENT} from './dom/dom_tokens';
export {EVENT_MANAGER_PLUGINS, EventManager} from
'./dom/events/event_manager';
export {HAMMER_GESTURE_CONFIG, HammerGestureConfig} from
'./dom/events/hammer_gestures';
export {DomSanitizer, SafeHtml, SafeResourceUrl, SafeScript, SafeStyle,
SafeUrl} from './security/dom_sanitization_service';

export * from './private_export';
```

Source Tree Layout

The source tree for the Platform-Browser package contains these directories:

- src

- test (unit tests in Jasmine)
- testing (test tooling)

and these files:

- index.ts
- package.json
- rollup.config.js
- rollup-testing.config.js
- testing.ts
- tsconfig.json
- tsconfig-testing.json

The tsconfig.json contains this list of files to compile:

```
{
  "compilerOptions": {
    ..
    "outDir": "../../../dist/packages-dist/platform-browser",
    "paths": {
      "@angular/core": ["../../../dist/packages-dist/core"],
      "@angular/common": ["../../../dist/packages-dist/common"]
    },
    ..
    "lib": ["es2015", "dom"]
  },
  "files": [
    "index.ts",
    "../../../node_modules/@types/hammerjs/index.d.ts",
    "../../../node_modules/zone.js/dist/zone.js.d.ts"
  ],
  "angularCompilerOptions": {
    "strictMetadataEmit": true
  }
}
```

Source

platform-browser/src

This directory contains the following file:

- browser.ts

The browser.ts file defines provider lists and the NgModule-decorated BrowserModule, all important in the bootstrapping of an Angular application.

Sanitization providers mitigate XSS risks:

```
/**
* @security Replacing built-in sanitization providers exposes the
* application to XSS risks. Attacker-controlled data introduced by an
* unsanitized provider could expose your application to XSS risks. For more
* detail, see the [Security Guide](http://g.co/ng/security).
*/
export const BROWSER_SANITIZATION_PROVIDERS: Array<any> = [
  {provide: Sanitizer, useExisting: DomSanitizer},
  {provide: DomSanitizer, useClass: DomSanitizerImpl},
```

```
];
```

It is used in the definition of NgModule:

```
@NgModule({
  providers: [
    BROWSER_SANITIZATION_PROVIDERS, {provide: ErrorHandler,
      useFactory: errorHandler, deps: []},
    {provide: DOCUMENT, useFactory: _document, deps: []},
    {provide: EVENT_MANAGER_PLUGINS, useClass: DomEventsPlugin, multi: true},
    {provide: EVENT_MANAGER_PLUGINS, useClass: KeyEventsPlugin, multi: true},
    {provide: EVENT_MANAGER_PLUGINS, useClass: HammerGesturesPlugin,
      multi: true},
    {provide: HAMMER_GESTURE_CONFIG, useClass: HammerGestureConfig},
    {provide: DomRootRenderer, useClass: DomRootRenderer_},
    {provide: RootRenderer, useExisting: DomRootRenderer},
    {provide: SharedStylesHost, useExisting: DomSharedStylesHost},
    {provide: AnimationDriver, useFactory: _resolveDefaultAnimationDriver},
    DomSharedStylesHost,
    Testability,
    EventManager,
    ELEMENT_PROBE_PROVIDERS
  ],
  exports: [CommonModule, ApplicationModule]
})
export class BrowserModule {
  constructor(@Optional() @SkipSelf() parentModule: BrowserModule) {
    if (parentModule) {
      throw new Error(
          `BrowserModule has already been loaded. If you need access to
           common directives such as NgIf and NgFor from a lazy loaded
           module, import CommonModule instead.`);
    }
  }
}
```

The initDomAdapter function sets the current DOM adapter to be BrowserDomAdapter (covered shortly). It also initializes the Web Tracing Framework via wtfInit(), a method exported by Core.

```
export function initDomAdapter() {
  BrowserDomAdapter.makeCurrent();
  wtfInit();
  BrowserGetTestability.init();
}
```

This function is used in an initializer by the app provider list:

```
export const INTERNAL_BROWSER_PLATFORM_PROVIDERS: Provider[] = [
  {provide: PLATFORM_INITIALIZER, useValue: initDomAdapter, multi: true},
  {provide: PlatformLocation, useClass: BrowserPlatformLocation}
];
```

This in turn is used by the call to createPlatformFactory:

```
export const platformBrowser =
    createPlatformFactory(
       platformCore, 'browser', INTERNAL_BROWSER_PLATFORM_PROVIDERS);
```

Platform-browser/src/dom

We will sub-divide the code in the DOM sub-directory into four categories – adapter/renderer, animation, debug and events.

We have seen how the Core package defines a rendering API and all other parts of the Angular framework and applicaton code uses it to have content rendered. But Core has no implementation. Now it is time to see an implementation, based on the DOM. That is the role of these files:

- dom_tokens.ts
- shared_styles_host.ts
- util.ts
- dom_adapter.ts
- dom_renderer.ts

dom_tokens.ts declares an opaque token:

```
/**
 * A DI Token representing the main rendering context.
 * In a browser this is the DOM Document.
 *
 * Note: Document might not be available in the Application Context when
 * Application and Rendering Contexts are not the same (e.g. when running
 * the application into a WebWorker).
 */
export const DOCUMENT: OpaqueToken = new OpaqueToken('DocumentToken');
```

shared_styles_host.ts manages a set of styles.

Utils.ts contains simple string helpers:

```
export function camelCaseToDashCase(input: string): string {
  return StringWrapper.replaceAllMapped(
     input, CAMEL_CASE_REGEXP,
     (m: any) => { return '-' + m[1].toLowerCase(); });
}

export function dashCaseToCamelCase(input: string): string {
  return StringWrapper.replaceAllMapped(
      input, DASH_CASE_REGEXP, (m: any) => { return m[1].toUpperCase(); });
}
```

The two main files involved in delivering the DomRenderer are dom_adapter.ts and dom_renderer.ts. A DomAdapter is a class that represents an API very close to the HTML DOM that every web developer is familiar with. A DOM renderer is an implementation of Core's Rendering API in terms of a DOM adapter.

The benefit that a DomAdapter brings (compared to hard-coding calls to the actual DOM inside a browser), is that multiple implementations of a DomAdapter can be supplied, including in scenarios where the real DOM is not available (e.g. server-side, or inside webworkers).

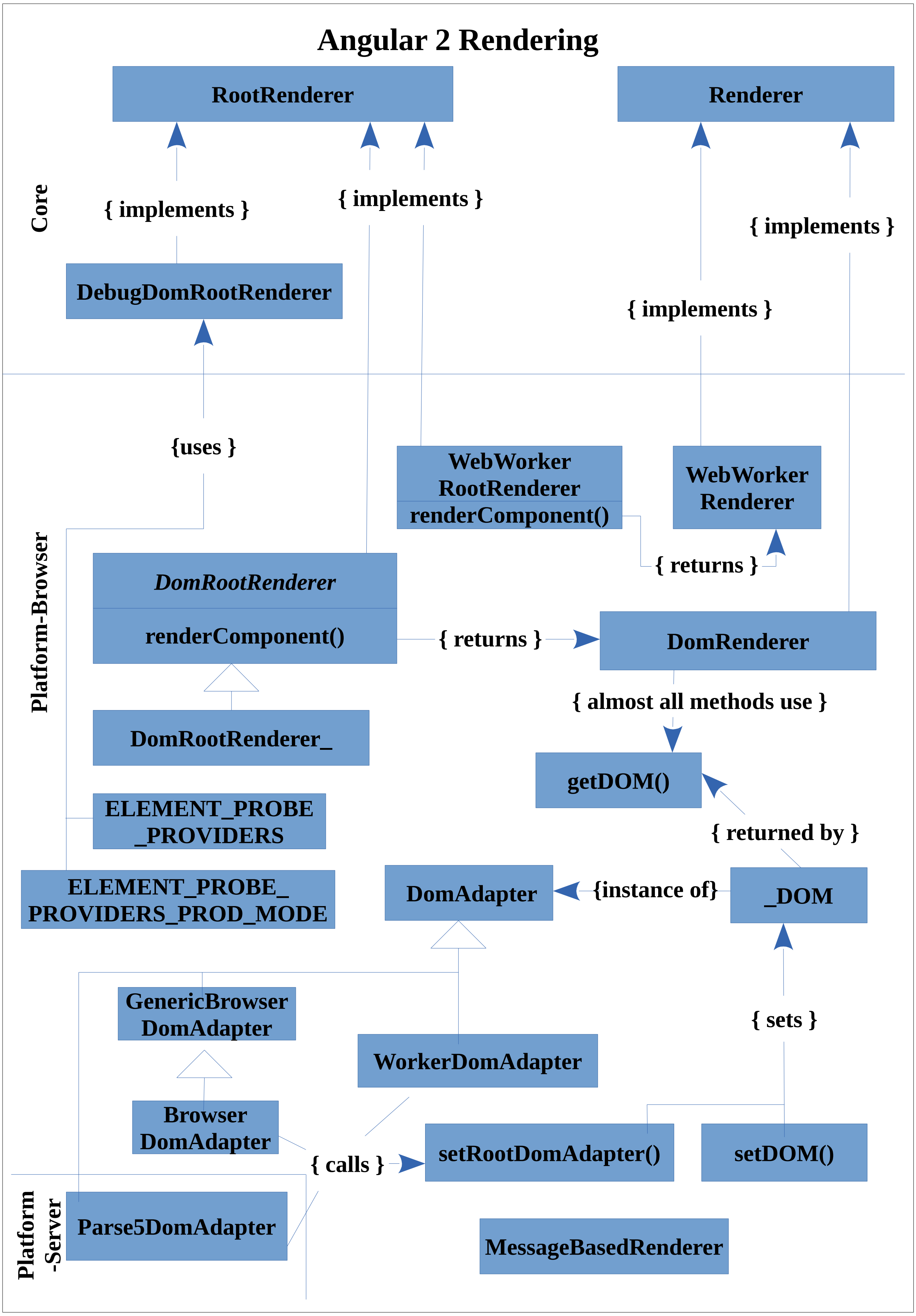
The following diagram shows how Core's Renderer API, renderer implementations and DOM adapters are related. For many applications, the entire application will run in the main browser UI thread and so BrowserDomAdapter will be used alongside DomRenderer.

For server applications with platform-server, a speclaist DOM adapter called parse5DomAdapter will be used alongside DomRenderer, and this results in content being written to an HTML file.

For more advanced browser applications that will use webworkers, things are a little more complicated and four classes are involved: WebWorkerRootRenderer, WorkerRenderer, MessageBasedRender and WorkerDomAdapter. WorkerDomAdapter is used merely for logging and does not place a significant part in rendering from workers. A message broker based on a message bus exchanges messages between the webworker and main browser UI thread. WebWorkerRootRenderer and WorkerRenderer run in the webworker and forward all rendering calls over the message broker to the main browser UI thread, where an instance of MessageBasedRender (which, despite its name, does not implement Core's Renderer interface) receives them and calls the regular DomRenderer. We will shortly examine in detail how rendering works with webworkers.

The dom_adapter.ts class defines the abstract DomAdapter class, the _DOM variable and two functions, getDOM() and setDOM() to get and set it.

```
var _DOM: DomAdapter = null;

export function getDOM() {
  return _DOM;
}

export function setDOM(adapter: DomAdapter) {
  _DOM = adapter;
}

export function setRootDomAdapter(adapter: DomAdapter) {
  if (isBlank(_DOM)) {
    _DOM = adapter;
  }
}
```


Angular 2 Rendering
RootRenderer
Renderer
Core
{ implements }
{ implements }
{ implements }
DebugDomRootRenderer
{ implements }
{uses }
WebWorker RootRenderer
renderComponent()
WebWorker Renderer
Platform-Browser
DomRootRenderer
renderComponent()
{ returns }
{ returns }
DomRenderer
{ almost all methods use }
DomRootRenderer_
getDOM()
ELEMENT_PROBE _PROVIDERS
{ returned by }
ELEMENT_PROBE_ PROVIDERS_PROD_MODE
DomAdapter
{instance of}
_DOM
GenericBrowser DomAdapter
{ sets }
WorkerDomAdapter
Browser DomAdapter
{ calls }
setRootDomAdapter()
setDOM()
Platform -Server
Parse5DomAdapter
MessageBasedRenderer

The DomAdapter class is a very long list of methods similar to what you would find a normal DOM API – here is a sampling of what it contains:

```
/**
 * Provides DOM operations in an environment-agnostic way.
 *
 * @security Tread carefully! Interacting with the DOM directly
 * is dangerous and can introduce XSS risks.
 */
export abstract class DomAdapter {
  abstract createEvent(eventType: string): any;
  abstract getInnerHTML(el: any /** TODO #9100 */): string;
  abstract getOuterHTML(el: any /** TODO #9100 */): string;
  abstract insertBefore(el: any, node: any): any;
  abstract insertAllBefore(el: any , nodes: any ): any;
  abstract insertAfter(el: any, node: any): any;
  abstract setInnerHTML(el: any , value: any): any;
  abstract getText(el: any): string;
  abstract setText(el: any, value: string): any;
  abstract getValue(el: any): string;
  abstract setValue(el: any, value: string): any;
  abstract createElement(tagName: any, doc?: any): HTMLElement;
  abstract createElementNS(ns: string, tagName: string, doc?: any): Elemen
  abstract createShadowRoot(el: any): any;
  abstract getShadowRoot(el: any): any;
  abstract hasAttribute(element: any, attribute: string): boolean;
  abstract getAttribute(element: any, attribute: string): string;
  abstract setAttribute(element: any, name: string, value: string): any;
  abstract getTitle(): string;
  abstract setTitle(newTitle: string): any;
  abstract elementMatches(n: any, selector: string): boolean;
}
```

The dom_renderer.ts file defines the DOM renderer that relies on getDOM() to supply a Dom Adapter. It supplies these classes – DomRootRenderer, DomRootRenderer_ and DomRenderer. We saw earlier how DomRootRenderer_ is used in the NgModule declaration:

```
@NgModule({
  providers: [
    ..
    {provide: DomRootRenderer, useClass: DomRootRenderer_},
    {provide: RootRenderer, useExisting: DomRootRenderer},
    .. })
export class BrowserModule { .. }
```

The DomRootRenderer class manages a map of strings to DomRenders. Its renderComponent() method handles insertions into this map and returns a DoMRender for each component as requested:

```
export abstract class DomRootRenderer implements RootRenderer {
  protected registeredComponents: Map<string, DomRenderer>
                                = new Map<string, DomRenderer>();

  constructor(
      public document: any,
      public eventManager: EventManager,
```

```
        public sharedStylesHost: DomSharedStylesHost,
        public animationDriver: AnimationDriver) {}

    renderComponent(componentProto: RenderComponentType): Renderer {
      var renderer = this.registeredComponents.get(componentProto.id);
      if (isBlank(renderer)) {
        renderer = new DomRenderer(this, componentProto, this.animationDriver);
        this.registeredComponents.set(componentProto.id, renderer);
      }
      return renderer;
    }
  }
```

The DomRootRenderer_ class extends DomRootRenderer. It is decorated with the injectable() and has an @inject(DOCUMENT) decorator for its document parameter:

```
  @Injectable()
  export class DomRootRenderer_ extends DomRootRenderer {
    constructor(
        @Inject(DOCUMENT) _document: any,
        _eventManager: EventManager,
        sharedStylesHost: DomSharedStylesHost,
        animationDriver: AnimationDriver) {
      super(_document, _eventManager, sharedStylesHost, animationDriver);
    }
  }
```

The big class is DomRenderer, which implements Renderer, mostly in terms of calls to a DomAdapter. As an example, let’s look at its `createElement()` method – we see how it uses getDOM() to implement the functionality:

```
    createElement(
         parent: Element,
         name: string,
         debugInfo: RenderDebugInfo): Node {
      var nsAndName = splitNamespace(name);
      var el = isPresent(nsAndName[0]) ?
          getDOM().createElementNS(
               (NAMESPACE_URIS as any)[nsAndName[0]], nsAndName[1]) :
          getDOM().createElement(nsAndName[1]);
      if (isPresent(this._contentAttr)) {
        getDOM().setAttribute(el, this._contentAttr, '');
      }
      if (isPresent(parent)) {
        getDOM().appendChild(parent, el);
      }
      return el;
    }
```

Components have a ViewEncapsulation enum settings – when set to Native, this means the shadow DOM should be used, as can be seen in createViewRoot:

```
    createViewRoot(hostElement: any): any {
      var nodesParent: any;
      if (this.componentProto.encapsulation === ViewEncapsulation.Native) {
        nodesParent = getDOM().createShadowRoot(hostElement);
        this._rootRenderer.sharedStylesHost.addHost(nodesParent);
        for (var i = 0; i < this._styles.length; i++) {
```

```
        getDOM().appendChild(nodesParent,
                             getDOM().createStyleElement(this._styles[i]));
      }
    } else {
      if (isPresent(this._hostAttr)) {
        getDOM().setAttribute(hostElement, this._hostAttr, '');
      }
      nodesParent = hostElement;
    }
    return nodesParent;
  }
```

The Dom animation functionality is supplied by:

- animation_driver.ts
- dom_animate_player.ts
- web_animations_driver.ts
- web_animations_player.ts

animation_driver.ts contains:

```
class _NoOpAnimationDriver implements AnimationDriver {
  animate(element: any, startingStyles: AnimationStyles,
  keyframes: AnimationKeyframe[], duration: number,
  delay: number, easing: string): AnimationPlayer {
    return new NoOpAnimationPlayer();
  }
}
export abstract class AnimationDriver {
  static NOOP: AnimationDriver = new _NoOpAnimationDriver();
  abstract animate(element: any, startingStyles: AnimationStyles,
    keyframes: AnimationKeyframe[], duration: number,
    delay: number, easing: string): AnimationPlayer;
}
```

dom_animate_player.ts contains:

```
export interface DomAnimatePlayer {
  cancel(): void;
  play(): void;
  pause(): void;
  finish(): void;
  onfinish: Function;
  position: number;
  currentTime: number;
}
```

The two other files – web_animations_driver.ts and web_animations-Player.ts – provide implementations of these types.

DOM debugging is supported via:

- debug/by.ts
- debug/ng_probe.ts

The By class may be used with Core’s DebugElement to supply predicates for its query functions. It supplies three predicates – all, css and directive:

```
export class By {
```

```
    // Match all elements.
    static all(): Predicate<DebugElement> { return (debugElement) => true; }

    // Match elements by the given CSS selector.
    static css(selector: string): Predicate<DebugElement> {
      return (debugElement) => {
        return isPresent(debugElement.nativeElement) ?
            getDOM().elementMatches(debugElement.nativeElement, selector) :
            false;
      };
    }

    //Match elements that have the given directive present.
    static directive(type: Type<any>): Predicate<DebugElement> {
      return (debugElement) => {
         return debugElement.providerTokens.indexOf(type) !== -1; };
    }
  }
```

DOM events are supported via:

- events/dom_events.ts
- events/event_manager.ts
- events/hammer_common.ts
- events/hammer_gestures.ts
- events/key_events.ts

event_manager.ts provides two classes – EventManager and EventManagerPlugin.

EventManager manages an array of EventManagerPlugins, which is defined as:

```
  export class EventManagerPlugin {
    manager: EventManager;

    // That is equivalent to having supporting $event.target
    supports(eventName: string): boolean { return false; }

    addEventListener(element: HTMLElement, eventName: string, handler:
  Function): Function {
      throw 'not implemented';
    }

    addGlobalEventListener(element: string, eventName: string, handler:
  Function): Function {
      throw 'not implemented';
    }
  }
```

It provides two methods to add event listeners to targets represeneted either by an HTMLElement or a string. EventManager itself is defined as an injectable class:

```
  @Injectable()
  export class EventManager {
    private _plugins: EventManagerPlugin[];

    constructor(
        @Inject(EVENT_MANAGER_PLUGINS) plugins: EventManagerPlugin[],
```

```
      private _zone: NgZone) {
    plugins.forEach(p => p.manager = this);
    this._plugins = ListWrapper.reversed(plugins);
  }

  addEventListener(element: HTMLElement,
                   eventName: string, handler: Function): Function {
    var plugin = this._findPluginFor(eventName);
    return plugin.addEventListener(element, eventName, handler);
  }

  addGlobalEventListener(target: string,
                 eventName: string, handler: Function): Function {
    var plugin = this._findPluginFor(eventName);
    return plugin.addGlobalEventListener(target, eventName, handler);
  }

  getZone(): NgZone { return this._zone; }

  /** @internal */
  _findPluginFor(eventName: string): EventManagerPlugin {
    var plugins = this._plugins;
    for (var i = 0; i < plugins.length; i++) {
      var plugin = plugins[i];
      if (plugin.supports(eventName)) {
        return plugin;
      }
    }
    throw new Error(`No event manager plugin found for event ${eventName}`);
  }
}
```

Its constructor is defiend in such a way as to allow dependency injection to inject an array of event manager plugins. We note this list is reversed in the constructor, which will impact the ordering of finding a plugin for an event type.

The other files in src/dom/events supply event manager plugins.

Note the comment in DomEventsPlugin:

```
@Injectable()
export class DomEventsPlugin extends EventManagerPlugin {
  // This plugin should come last in the list of plugins, because it accepts
all
  // events.
  supports(eventName: string): boolean { return true; }

  addEventListener(element: HTMLElement, eventName: string, handler:
Function): Function {
    var zone = this.manager.getZone();
    var outsideHandler = (event: any /** TODO #9100 */) => zone.runGuarded(()
=> handler(event));
    return this.manager.getZone().runOutsideAngular(
        () => getDOM().onAndCancel(element, eventName, outsideHandler));
  }

  addGlobalEventListener(target: string, eventName: string, handler:
Function): Function {
```

```
      var element = getDOM().getGlobalEventTarget(target);
      var zone = this.manager.getZone();
      var outsideHandler = (event: any /** TODO #9100 */) => zone.runGuarded(()
  => handler(event));
      return this.manager.getZone().runOutsideAngular(
          () => getDOM().onAndCancel(element, eventName, outsideHandler));
    }
  }
```

Touch events via the hammer project are supported via the hammer_common.ts and hammer_gesture.ts files. The list of supported touch events are:

```
var _eventNames = {
  // pan
  'pan': true,
  'panstart': true,
  'panmove': true,
  'panend': true,
  'pancancel': true,
  'panleft': true,
  'panright': true,
  'panup': true,
  'pandown': true,
  // pinch
  'pinch': true,
  'pinchstart': true,
  'pinchmove': true,
  'pinchend': true,
  'pinchcancel': true,
  'pinchin': true,
  'pinchout': true,
  // press
  'press': true,
  'pressup': true,
  // rotate
  'rotate': true,
  'rotatestart': true,
  'rotatemove': true,
  'rotateend': true,
  'rotatecancel': true,
  // swipe
  'swipe': true,
  'swipeleft': true,
  'swiperight': true,
  'swipeup': true,
  'swipedown': true,
  // tap
  'tap': true,
};
```

This is used in HammerGesturesPluginCommon, which is extended by HammerGesturesPlugin.

```
@Injectable()
export class HammerGesturesPlugin extends HammerGesturesPluginCommon {
  constructor(@Inject(HAMMER_GESTURE_CONFIG) private _config:
HammerGestureConfig) { super(); }
```

```
  supports(eventName: string): boolean {
    if (!super.supports(eventName) && !this.isCustomEvent(eventName)) return
false;

    if (!isPresent((window as any /** TODO #???? */)['Hammer'])) {
      throw new Error(`Hammer.js is not loaded, can not bind ${eventName}
event`);
    }

    return true;
  }

  addEventListener(element: HTMLElement, eventName: string, handler:
Function): Function {
    var zone = this.manager.getZone();
    eventName = eventName.toLowerCase();

    return zone.runOutsideAngular(() => {
      // Creating the manager bind events, must be done outside of angular
      var mc = this._config.buildHammer(element);
      var callback = function(eventObj: any /** TODO #???? */) {
        zone.runGuarded(function() { handler(eventObj); });
      };
      mc.on(eventName, callback);
      return () => { mc.off(eventName, callback); };
    });
  }

  isCustomEvent(eventName: string): boolean { return
this._config.events.indexOf(eventName) > -1; }
}
```

Note the use of zone.runOutsideAngular() in addEventListener. Also ntoe it does not implement addGlobalEventListener. Tits constructor expects a HAMMER_GESTURE_CONFIG from dependency injection. The Hammer package is used in the injectable HammerGestureConfig class:

```
@Injectable()
export class HammerGestureConfig {
  events: string[] = [];

  overrides: {[key: string]: Object} = {};

  buildHammer(element: HTMLElement): HammerInstance {
    var mc = new Hammer(element);

    mc.get('pinch').set({enable: true});
    mc.get('rotate').set({enable: true});

    for (let eventName in this.overrides) {
      mc.get(eventName).set(this.overrides[eventName]);
    }

    return mc;
  }
}
```

These event manager plugins need to be set in the NgModule configuration. We see how this is done in BrowserModule:

```
@NgModule({
  providers: [
    ..
    {provide: EVENT_MANAGER_PLUGINS, useClass: DomEventsPlugin, multi: true},
    {provide: EVENT_MANAGER_PLUGINS, useClass: KeyEventsPlugin, multi: true},
    {provide: EVENT_MANAGER_PLUGINS, useClass: HammerGesturesPlugin,
                                                                 multi: true},
    {provide: HAMMER_GESTURE_CONFIG, useClass: HammerGestureConfig},
    ..
  ],
  exports: [CommonModule, ApplicationModule]
})
export class BrowserModule {..}
```

Platform-browser/src/browser

This directory contains these files:

- generic_browser_adapter.ts
- browser_adapter.ts
- estability.ts
- title.ts
- location/browser_platform_location.ts
- location/history.ts
- tools/common_tools.ts
- tools/tools.ts

generic_browser_adapter.ts defines the abstract GenericBrowserDomAdapter class that extends DomAdapter with DOM operations suitable for general browsers:

```
export abstract class GenericBrowserDomAdapter extends DomAdapter{}
```

For example, to check if shadow DOM is supported, it evaluates whether the createShadowRoot function exists:

```
supportsNativeShadowDOM(): boolean {
    return isFunction((<any>this.defaultDoc().body).createShadowRoot);
  }
```

GenericBrowserDomAdapter does not provide the full implementation of DomAdapter and so a derived class is needed also.

browser_adapter.ts supplies the concrete BrowserDomAdapter class, which does come with a full implementation of DomAdapter:

```
/**
 * A `DomAdapter` powered by full browser DOM APIs.
 *
 * @security Tread carefully! Interacting with the DOM directly
 * is dangerous and can introduce XSS risks.
 */
export class BrowserDomAdapter extends GenericBrowserDomAdapter {..}
```

Generally its methods provide implementation based on window or document – here are some samples:

```
  getUserAgent(): string { return window.navigator.userAgent; }
  getHistory(): History { return window.history; }
  getTitle(): string { return document.title; }
  setTitle(newTitle: string) { document.title = newTitle || ''; }
```

title.ts supplies the Title service:

```
/**
 * A service that can be used to get and set the title of a current
 * HTML document.
 *
 * Since an Angular application can't be bootstrapped on the entire
 * HTML document (`<html>` tag) it is not possible to bind to the `text`
 * property of the `HTMLTitleElement` elements (representing the `<title>`
 * tag). Instead, this service can be used to set and get the current
 * title value.
 */
export class Title {
  /**
   * Get the title of the current HTML document.
   * @returns {string}
   */
  getTitle(): string { return getDOM().getTitle(); }

  /**
   * Set the title of the current HTML document.
   * @param newTitle
   */
  setTitle(newTitle: string) { getDOM().setTitle(newTitle); }
}
```

The tools sub-directory contains tools.ts and common_tools.ts, which implement the functions enableDebugTools() and disableDebugTools(), and the AngularTools and AngularProfiler classes. The classes are defined as follows:

```
/**
 * Entry point for all Angular debug tools.
 * This object corresponds to the `ng`
 * global variable accessible in the dev console.
 */
export class AngularTools {
  profiler: AngularProfiler;

  constructor(ref: ComponentRef<any>) {
           this.profiler = new AngularProfiler(ref); }
}

/**
 * Entry point for all Angular profiling-related debug tools. This object
 * corresponds to the `ng.profiler` in the dev console.
 */
export class AngularProfiler {
  appRef: ApplicationRef;
  constructor(ref: ComponentRef<any>) {
          this.appRef = ref.injector.get(ApplicationRef); }
  timeChangeDetection(config: any): ChangeDetectionPerfRecord { .. }
}
```

The functions add 1 and remove 2 a ng property to global and are defined as follows:

```
var context = <any>global;
/**
 * Enabled Angular debug tools that are accessible via your browser's
 * developer console.
 *
 * Usage:
 * 1. Open developer console (e.g. in Chrome Ctrl + Shift + j)
 * 2. Type `ng.` (usually the console will show auto-complete suggestion)
 * 3. Try the change detection profiler `ng.profiler.timeChangeDetection()`
 *    then hit Enter.
 */
export function enableDebugTools<T>(ref: ComponentRef<T>): ComponentRef<T> {
1 context.ng = new AngularTools(ref);
  return ref;
}
/**
 * Disables Angular tools.
 *
 * @experimental All debugging apis are currently experimental.
 */
export function disableDebugTools(): void {
2 delete context.ng;
}
```

The src/browser/location sub-directory contains browser_platform_location.ts and history.ts. browser_platform_location.ts contains the injectable class BrowserPlatformLocation:

```
/**
 * `PlatformLocation` encapsulates all of the direct calls to platform APIs.
 * This class should not be used directly by an application developer.
 *  Instead, use Location.
 */
@Injectable()
export class BrowserPlatformLocation extends PlatformLocation  {..}
```

This manages two private fields, location and history, which are initialized with getDOM().

```
export class BrowserPlatformLocation extends PlatformLocation {
  private _location: Location;
  private _history: History;

  constructor() {
    super();
    this._init();
  }

  _init() {
```

```
    this._location = getDOM().getLocation();
    this._history = getDOM().getHistory();
  }

  get location(): Location { return this._location; }
```

The rest of the class provides functionality to work with location and history:

```
  getBaseHrefFromDOM(): string { return getDOM().getBaseHref(); }

  onPopState(fn: LocationChangeListener): void {
    getDOM().getGlobalEventTarget('window')
                 .addEventListener('popstate', fn, false);
  }

  onHashChange(fn: LocationChangeListener): void {
    getDOM().getGlobalEventTarget('window').
                 addEventListener('hashchange', fn, false);
  }

  get pathname(): string { return this._location.pathname; }
  get search(): string { return this._location.search; }
  get hash(): string { return this._location.hash; }
  set pathname(newPath: string) { this._location.pathname = newPath; }

  pushState(state: any, title: string, url: string): void {
    if (supportsState()) {
      this._history.pushState(state, title, url);
    } else {
      this._location.hash = url;
    }
  }

  replaceState(state: any, title: string, url: string): void {
    if (supportsState()) {
      this._history.replaceState(state, title, url);
    } else {
      this._location.hash = url;
    }
  }

  forward(): void { this._history.forward(); }

  back(): void { this._history.back(); }
}
```

history.ts contains this simple function:

```
export function supportsState(): boolean {
  return !!window.history.pushState;
}
```

Platform-browser/src/security

This directory contains these files:

- dom_sanitization_service.ts
- html_sanitizer.ts
- style_sanitizer.ts

- url_sanitizer.ts

Security sanitizers help prevent the use of dangerous constructs in HTML, CSS styles and URLs. Sanitizers are configured via an NgModule setting. This is the relevant extract from platform-browser/src/browser.ts:

```
/**
 * @security Replacing built-in sanitization providers
 *exposes the application to XSS risks.
 * Attacker-controlled data introduced by an unsanitized provider could
 * expose your application to XSS risks. For more detail, see the
 * [Security Guide](http://g.co/ng/security).
 */
export const BROWSER_SANITIZATION_PROVIDERS: Array<any> = [
  {provide: Sanitizer, useExisting: DomSanitizer},
  {provide: DomSanitizer, useClass: DomSanitizerImpl},
];

@NgModule({
  providers: [
    BROWSER_SANITIZATION_PROVIDERS,
   ..
  ],
  exports: [CommonModule, ApplicationModule]
})
export class BrowserModule {..}
```

The dom_sanitization_service.ts file declares a range of safeXYZ interfaces, implementation classes for them, the DomSanitizer class and the DomSanitizerImpl class. It starts by importing the SecurityContext enum and Sanitizer abstract class from Core. Let's recall they are defiend as:

```
/**
 * A SecurityContext marks a location that has dangerous security
 * implications, e.g. a DOM property like `innerHTML` that could cause
 * Cross Site Scripting (XSS) security bugs when improperly handled.
 */
export enum SecurityContext {
  NONE,
  HTML,
  STYLE,
  SCRIPT,
  URL,
  RESOURCE_URL,
}

// Sanitizer is used by the views to sanitize potentially dangerous values.
export abstract class Sanitizer {
  abstract sanitize(context: SecurityContext, value: string): string;
}
```

The safe marker interfaces are declared as:

```
// Marker interface for a value that's safe to use in a particular context.
export interface SafeValue {}

// Marker interface for a value that's safe to use as HTML.
export interface SafeHtml extends SafeValue {}
```

```
// Marker interface for a value that's safe to use as style (CSS).
export interface SafeStyle extends SafeValue {}

//Marker interface for a value that's safe to use as JavaScript.
export interface SafeScript extends SafeValue {}

// Marker interface for a value that's safe to use as a URL
// linking to a document.
export interface SafeUrl extends SafeValue {}

// Marker interface for a value that's safe to use as a URL to
// load executable code from.
export interface SafeResourceUrl extends SafeValue {}
```

The DomSanitizer abstract class implements Core’s Sanitizer class. Do read the large comment at the beginning – you really do not want to be bypassing security it at all possible.

```
/**
 * DomSanitizer helps preventing Cross Site Scripting Security bugs (XSS) by
 * sanitizing values to be safe to use in the different DOM contexts.
 *
 * For example, when binding a URL in an `<a [href]="someValue">` hyperlink,
 * `someValue` will be sanitized so that an attacker cannot inject e.g. a
 * `javascript:` URL that would execute code on the website.
 *
 * In specific situations, it might be necessary to disable sanitization, for
 * example if the application genuinely needs to produce a `javascript:`
 * style link with a dynamic value in it.
 * Users can bypass security by constructing a value with one of the
 * `bypassSecurityTrust...` methods, and then binding to that value from
 * the template.
 *
 * These situations should be very rare, and extraordinary care must be taken
 * to avoid creating a Cross Site Scripting (XSS) security bug!
 *
 * When using `bypassSecurityTrust...`, make sure to call the method as early
 *  as possible and as close as possible to the source of the value, to make
 * it easy to verify no security bug is created by its use.
 *
 * It is not required (and not recommended) to bypass security if the value
 *  is safe, e.g. a URL that does not start with a suspicious protocol, or an
 * HTML snippet that does not contain dangerous
 * code. The sanitizer leaves safe values intact.
 *
 * @security Calling any of the `bypassSecurityTrust...` APIs disables
 *  Angular's built-in sanitization for the value passed in. Carefully check
 * and audit all values and code paths going into this call. Make sure any
 * user data is appropriately escaped for this security context.
 * For more detail, see the [Security Guide](http://g.co/ng/security).
 */
export abstract class DomSanitizer implements Sanitizer {
  abstract sanitize(context: SecurityContext, value: any): string;
  abstract bypassSecurityTrustHtml(value: string): SafeHtml;
  abstract bypassSecurityTrustStyle(value: string): SafeStyle;
  abstract bypassSecurityTrustScript(value: string): SafeScript;
```

```
    abstract bypassSecurityTrustUrl(value: string): SafeUrl;
    abstract bypassSecurityTrustResourceUrl(value: string): SafeResourceUrl;
  }
```

The DomSanitizerImpl injectable class is what is supplied to NgModule:

```
@Injectable()
export class DomSanitizerImpl extends DomSanitizer { .. }
```

It can be divided into three sections, the checkNotSafeValue method, the sanitize method and the bypassSecurityTrustXYZ methods. The checkNotSafeValue method throws an error is the value parameter is an instance of SafeValueImpl:

```
  private checkNotSafeValue(value: any, expectedType: string) {
    if (value instanceof SafeValueImpl) {
      throw new Error(
          `Required a safe ${expectedType}, got a ${value.getTypeName()} ` +
          `(see http://g.co/ng/security#xss)`);      }    }
```

The sanitize method switches on the securityContext enum paramter, if it is NONE, then value is simply returned, otherwise additional checking is carried out, which varies depending on the security context:

```
  sanitize(ctx: SecurityContext, value: any): string {
    if (value == null) return null;
    switch (ctx) {
      case SecurityContext.NONE:
        return value;
      case SecurityContext.HTML:
        if (value instanceof SafeHtmlImpl)
          return value.changingThisBreaksApplicationSecurity;
        this.checkNotSafeValue(value, 'HTML');
        return sanitizeHtml(String(value));
      case SecurityContext.STYLE:
        if (value instanceof SafeStyleImpl)
          return value.changingThisBreaksApplicationSecurity;
        this.checkNotSafeValue(value, 'Style');
        return sanitizeStyle(value);
      case SecurityContext.SCRIPT:
        if (value instanceof SafeScriptImpl)
          return value.changingThisBreaksApplicationSecurity;
        this.checkNotSafeValue(value, 'Script');
        throw new Error('unsafe value used in a script context');
      case SecurityContext.URL:
        if (value instanceof SafeResourceUrlImpl
                      || value instanceof SafeUrlImpl) {
        // Allow resource URLs in URL contexts,
        // they are strictly more trusted.
          return value.changingThisBreaksApplicationSecurity;
        }
        this.checkNotSafeValue(value, 'URL');
        return sanitizeUrl(String(value));
      case SecurityContext.RESOURCE_URL:
        if (value instanceof SafeResourceUrlImpl) {
          return value.changingThisBreaksApplicationSecurity;
        }
        this.checkNotSafeValue(value, 'ResourceURL');
        throw new Error(
```

```
            'unsafe value used in a resource URL context
             (see http://g.co/ng/security#xss)');
      default:
        throw new Error(
      `Unexpected SecurityContext ${ctx} (see http://g.co/ng/security#xss)`);
    }
  }
```

The bypassSecurityTrust methods returns an appropriate SafeImpl instance:

```
bypassSecurityTrustHtml(value: string): SafeHtml {
   return new SafeHtmlImpl(value); }
bypassSecurityTrustStyle(value: string): SafeStyle {
   return new SafeStyleImpl(value); }
bypassSecurityTrustScript(value: string): SafeScript {
   return new SafeScriptImpl(value); }
bypassSecurityTrustUrl(value: string): SafeUrl {
   return new SafeUrlImpl(value); }
bypassSecurityTrustResourceUrl(value: string): SafeResourceUrl {
  return new SafeResourceUrlImpl(value);
}
```

8: @Angular/Platform-Browser-Dynamic

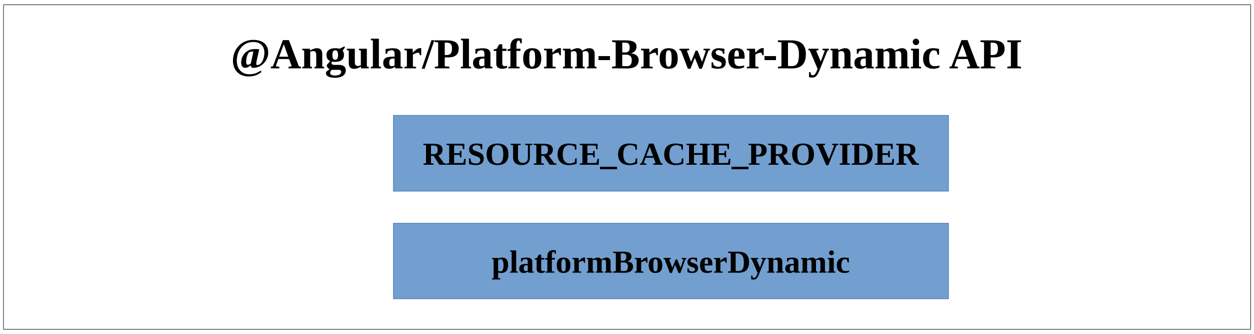
Overview

The term “dynamic” essentially means use the runtime template compiler and its absence means use the offline template compiler.

When Angular applications are bootstrapping they need to supply a platform. Those applications that wish to use runtime template compilation will need to supply a platform from Platform-Browser-Dynamic. If the application is to run in the browser’s main thread the platform to use is `platformBrowserDynamic`. If the application is to run in a webworker, then the platform to use is `platformWorkerAppDynamic` from the @Angular/Platform-WebWorker-Dynamic package.

Platform-Browser-Dynamic API

The exported API of the @Angular/Platform-Browser-Dynamic package can be represented as:



It’s index.ts is defines the following four exports.

```
export const RESOURCE_CACHE_PROVIDER: Provider[] =
    [{provide: ResourceLoader, useClass: CachedResourceLoader}];
```

`RESOURCE_CACHE_PROVIDER` manages a cache of downloaded resources.

```
export const platformBrowserDynamic = createPlatformFactory(
                platformCoreDynamic, 'browserDynamic',
                INTERNAL_BROWSER_DYNAMIC_PLATFORM_PROVIDERS);
```

This is how applications that use the runtime template compiler bootstrap a platform. We will need to see how `INTERNAL_BROWSER_DYNAMIC_PLATFORM_PROVIDERS` registers the compiler with dependency injection (it is defined in src/platform_providers.ts).

```
export function bootstrapWorkerUi(
    workerScriptUri: string,
    customProviders: Provider[] = []): Promise<PlatformRef> {
  // For now, just creates the worker ui platform...
  return Promise.resolve(platformWorkerUi(([{
                                          provide: WORKER_SCRIPT,
                                          useValue: workerScriptUri,
                                        }] as Provider[])
                                            .concat(customProviders)));
}
```

This returns a promise that resolves to `platformWorkerUi`, which runs in the browser's main (UI) thread and handles communication from platform running in a webworker (there is no Dom in the webworker, so to render to the browser's UI, this must be done via the browser's main thread – hence the need for `bootstrapWorkerUi`.

```
export const platformWorkerAppDynamic = createPlatformFactory(
  platformCoreDynamic, 'workerAppDynamic', [{
    provide: COMPILER_OPTIONS,
    useValue:
      {providers: [{provide: ResourceLoader, useClass: ResourceLoaderImpl}]},
      multi: true}]);
```

Creates a platform factory for running the application in a webworker. We note it adds a single provider configurations to platformCoreDynamic – COMPILER_OPTIONS, and this uses a ResourceLoader.

Source Tree Layout

The source tree for the Platform-Browser-Dynamic package contains these directories:

- src
- test (unit tests in Jasmine)
- testing (test tooling)

and these files:

- index.ts
- package.json
- rollup-testing.config.js
- rollup.config.js
- tsconfig.json
- tsconfig-testing.json

The src directory for Platform-Browser-Dynamic also contains these private import/export files:

- private_export.ts
- private-import_core.ts
- private_import_platform_browser.ts

As an example, let's look at private_platform_browser.ts:

```
import {__platform_browser_private__ as _} from '@angular/platform-browser';
export var INTERNAL_BROWSER_PLATFORM_PROVIDERS: typeof
_.INTERNAL_BROWSER_PLATFORM_PROVIDERS =
    _.INTERNAL_BROWSER_PLATFORM_PROVIDERS;
export var getDOM: typeof _.getDOM = _.getDOM;
```

Source

src

Apart from the import/export files, this directory contains one file:

- platform_providers.ts

It contains the definition for a single const:

```
export const INTERNAL_BROWSER_DYNAMIC_PLATFORM_PROVIDERS: Provider[] = [
  INTERNAL_BROWSER_PLATFORM_PROVIDERS,
  {
    provide: COMPILER_OPTIONS,
    useValue: {providers: [{provide: ResourceLoader, useClass:
ResourceLoaderImpl}]},
    multi: true
  },
];
```

This extends `INTERNAL_BROWSER_PLATFORM_PROVIDERS` with a single additional provider configuration, `COMPILER_OPTIONS`, which itself needs a `ResourceLoader` (defined in the Compiler module), that we see is set to use `ResourceLoaderImpl` (defined in this module).

src/resource_loader

This directory has two files:

- resource_loader_cache.ts
- resource_loader_impl.ts

`CachedResourceLoader` is a cached version of a `ResourceLoader`. When the template compiler needs to access documents, it passes the job on to a configured resource loader. `ResourceLoader` is defined in:

- <ANGULAR2>/modules/@angular/compiler/src/resource_loader.ts

as:

```
/**
 * An interface for retrieving documents by URL that the compiler uses
 * to load templates.
 */
export class ResourceLoader {
  get(url: string): Promise<string> { return null; }
}
```

`CachedResourceLoader` uses a promise wrapper to resolve or reject a get request:

```
export class CachedResourceLoader extends ResourceLoader {
  private _cache: {[url: string]: string};

  constructor() {
    super();
    this._cache = (<any>global).$templateCache;
    if (this._cache == null) {
      throw new Error(
    'CachedResourceLoader: Template cache was not found in $templateCache.');
    }
  }

  get(url: string): Promise<string> {
    if (this._cache.hasOwnProperty(url)) {
      return Promise.resolve(this._cache[url]);
    } else {
```

```
        return <Promise<any>>Promise.reject(
            'CachedResourceLoader: Did not find cached template for ' + url);
      }
    }
  }
```

`ResourceLoaderImpl` is a different injectable implementation of `ResourceLoader` that use `XMLHttpRequest()`:

```
  @Injectable()
  export class ResourceLoaderImpl extends ResourceLoader {
    get(url: string): Promise<string> {
      var resolve: (result: any) => void;
      var reject: (error: any) => void;
      const promise = new Promise((res, rej) => {
        resolve = res;
        reject = rej;
      });
      var xhr = new XMLHttpRequest();
      xhr.open('GET', url, true);
      xhr.responseType = 'text';

      xhr.onload = function() { .. };

      xhr.onerror = function() { reject(`Failed to load ${url}`); };

      xhr.send();
      return promise;
    }
  }
```

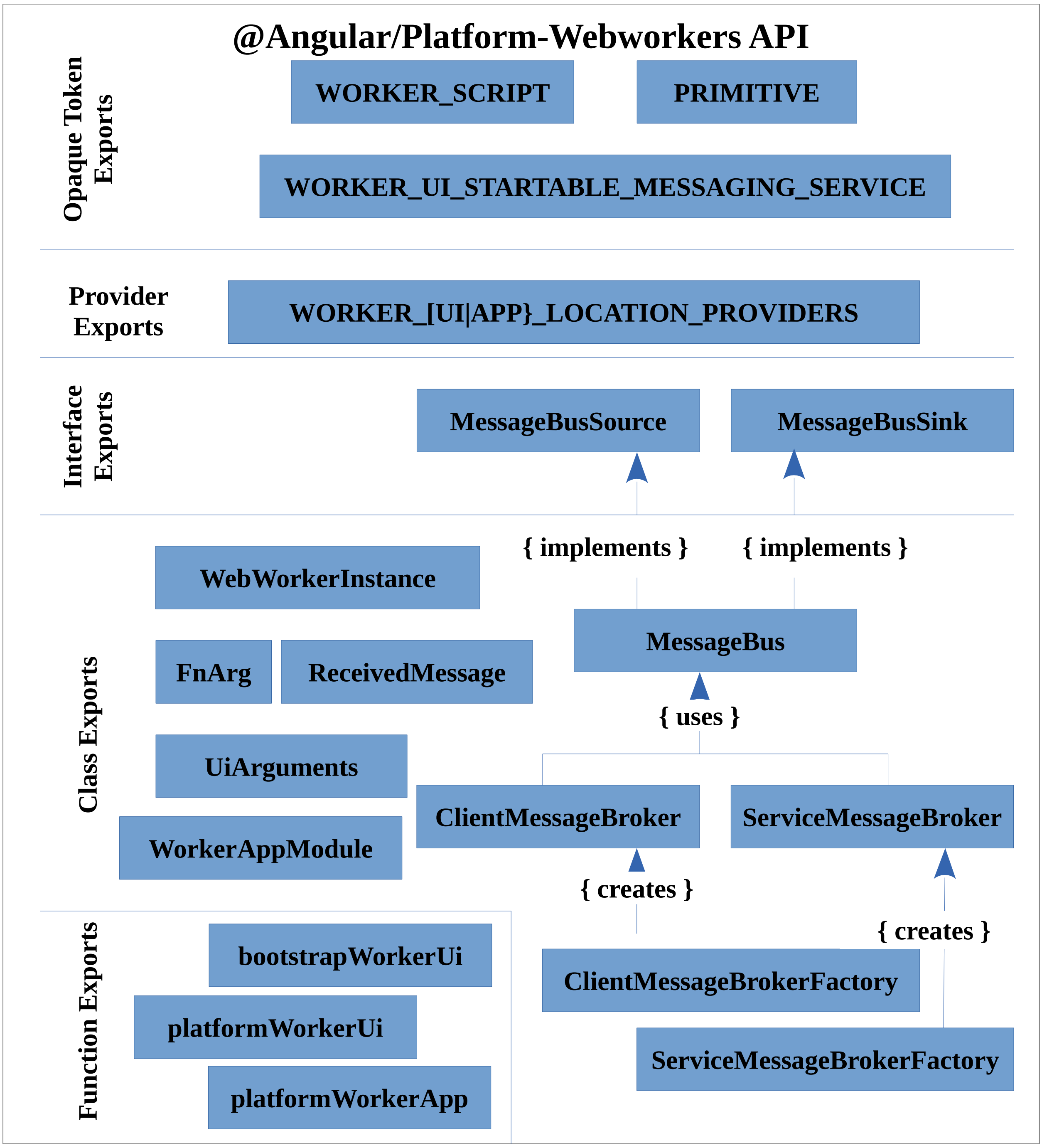
9: @Angular/Platform-WebWorker

Overview

Platform-WebWorkers is used to help applications run inside a webworker and render to the main UI thread via a message bus.

Platform-WebWorker API

The Platform-WebWorker related API is large and can be represented as:



The index.ts file is simply:

```
export * from './src/platform-webworker';
```

The src/platfrom-webworker.ts file lists the various exports and defines the bootstrapWorkerUi function:

```
import {PlatformRef, Provider} from '@angular/core';

import {WORKER_SCRIPT, platformWorkerUi} from './worker_render';

export {ClientMessageBroker, ClientMessageBrokerFactory, FnArg, UiArguments}
from './web_workers/shared/client_message_broker';
export {MessageBus, MessageBusSink, MessageBusSource} from
'./web_workers/shared/message_bus';
export {PRIMITIVE} from './web_workers/shared/serializer';
export {ReceivedMessage, ServiceMessageBroker, ServiceMessageBrokerFactory}
from './web_workers/shared/service_message_broker';
export {WORKER_UI_LOCATION_PROVIDERS} from
'./web_workers/ui/location_providers';
export {WORKER_APP_LOCATION_PROVIDERS} from
'./web_workers/worker/location_providers';
export {WorkerAppModule, platformWorkerApp} from './worker_app';
export {platformWorkerUi} from './worker_render';

//Bootstraps the worker ui.
export function bootstrapWorkerUi(
    workerScriptUri: string, customProviders: Provider[] = []):
Promise<PlatformRef> {
  // For now, just creates the worker ui platform...
  return Promise.resolve(platformWorkerUi(([{
                                            provide: WORKER_SCRIPT,
                                            useValue: workerScriptUri,
                                          }] as Provider[])
                                              .concat(customProviders)));
}
```

Source Tree Layout

The source tree for the Platform-WebWorker package contains these directories:

- src
- test (unit tests in Jasmine)
- testing (test tooling)

and these files:

- index.ts
- package.json
- rollup.config.js
- testing.ts
- tsconfig.json

Source

platform-webworker/src

In addition to the import/export files, this directory contains the following files:

- worker_app.ts
- worker_render.ts

worker_app.ts supplies functionality for an application that runs in a worker and worker_render.ts supplies functionality for the the main UI thread. They communicate via a message broker layered above a simple message bus.

The platformWorkerApp const creates a platform factory for a worker app:

```
export const platformWorkerApp =
                createPlatformFactory(platformCore, 'workerApp');
```

Two important helper functions are supplied. createMessageBus creates the message bus that will supply communications between the main UI thread and the webworker:

```
export function createMessageBus(zone: NgZone): MessageBus {
  let sink = new PostMessageBusSink(_postMessage);
  let source = new PostMessageBusSource();
  let bus = new PostMessageBus(sink, source);
  bus.attachToZone(zone);
  return bus;
}
```

setupWebWorker makes the webworker's DOM adapter implementation as the current DOM adapter. The DOM renderer is used both for apps running in a normal browser UI thread and apps running in webworkers. What is different is the DOM adapter – for a browser UI thread, the DOM adapter just maps to the underlying browser DOM API. There is not underlying DOM API in a webworker. So for an app running in a webworker, the worker DOM adapter needs to forward all DOM actions across the message bus to the browser's main UI thread.

```
export function setupWebWorker(): void {
  WorkerDomAdapter.makeCurrent();
}
```

Finally, the NgModule is defined:

```
@NgModule({
  providers: [
    BROWSER_SANITIZATION_PROVIDERS,
    Serializer,
    {provide: ClientMessageBrokerFactory, useClass:
      ClientMessageBrokerFactory_},
    {provide: ServiceMessageBrokerFactory, useClass:
      ServiceMessageBrokerFactory_},
    WebWorkerRootRenderer,
    {provide: RootRenderer, useExisting: WebWorkerRootRenderer},
    {provide: ON_WEB_WORKER, useValue: true}, RenderStore,
    {provide: ErrorHandler, useFactory: errorHandler, deps: []},
    {provide: MessageBus, useFactory: createMessageBus, deps: [NgZone]},
    {provide: APP_INITIALIZER, useValue: setupWebWorker, multi: true}
  ],
```

```
    exports: [CommonModule, ApplicationModule]
  }) export class WorkerAppModule { }
```

The worker_render.ts file has implementations for the Worker UI. This runs in the main UI thread and uses the BrowserDomAdapter, which writes to the underlying DOM API of the browser. PlatformWorkerUi represents a platform factory:

```
export const platformWorkerUi =
    createPlatformFactory(platformCore, 'workerUi',
      _WORKER_UI_PLATFORM_PROVIDERS);
```

The providers used are as followed:

```
export const _WORKER_UI_PLATFORM_PROVIDERS: Provider[] = [
  {provide: NgZone, useFactory: createNgZone, deps: []},
  MessageBasedRenderer,
  {provide: WORKER_UI_STARTABLE_MESSAGING_SERVICE,
     useExisting: MessageBasedRenderer, multi: true},
  BROWSER_SANITIZATION_PROVIDERS,
  {provide: ErrorHandler, useFactory: _exceptionHandler, deps: []},
  {provide: DOCUMENT, useFactory: _document, deps: []},
  {provide: EVENT_MANAGER_PLUGINS, useClass: DomEventsPlugin, multi: true},
  {provide: EVENT_MANAGER_PLUGINS, useClass: KeyEventsPlugin, multi: true},
  {provide: EVENT_MANAGER_PLUGINS, useClass: HammerGesturesPlugin,
     multi: true},
  {provide: HAMMER_GESTURE_CONFIG, useClass: HammerGestureConfig},
  {provide: DomRootRenderer, useClass: DomRootRenderer_},
  {provide: RootRenderer, useExisting: DomRootRenderer},
  {provide: SharedStylesHost, useExisting: DomSharedStylesHost},
  {provide: ServiceMessageBrokerFactory,
     useClass: ServiceMessageBrokerFactory_},
  {provide: ClientMessageBrokerFactory,
     useClass: ClientMessageBrokerFactory_},
  {provide: AnimationDriver,
     useFactory: _resolveDefaultAnimationDriver, deps: []},
  Serializer,
  {provide: ON_WEB_WORKER, useValue: false},
  RenderStore,
  DomSharedStylesHost,
  Testability,
  EventManager,
  WebWorkerInstance,
  {
    provide: PLATFORM_INITIALIZER,
    useFactory: initWebWorkerRenderPlatform,
    multi: true,
    deps: [Injector]
  },
  {provide: MessageBus, useFactory: messageBusFactory,
     deps: [WebWorkerInstance]}
];
```

Note the provider configuration for WORKER_UI_STARTABLE_MESSAGING_SERVICE is set t multi- thus allowing multiple messaging services to be started. Also note that initWebWorkerRenderPlatform is registered as a PLATFORM_INITIALIZER, so it is going to be called when the platform launches.

WebWorkerInstance is a simple injectable class representing the webworker and its message bus (note the init method just initializes the two fields):

```
@Injectable()
export class WebWorkerInstance {
  public worker: Worker;
  public bus: MessageBus;
  public init(worker: Worker, bus: MessageBus) {
    this.worker = worker;
    this.bus = bus;
  }
}
```

Now let’s look at initWebWorkerRenderPlatform:

```
function initWebWorkerRenderPlatform(injector: Injector): () => void {
  return () => {
    BrowserDomAdapter.makeCurrent(); 1
    wtfInit();
    BrowserGetTestability.init();
    var scriptUri: string;
    try {
      scriptUri = injector.get(WORKER_SCRIPT); 2
    } catch (e) {
      throw new Error(
          'You must provide your WebWorker\'s initialization
           script with the WORKER_SCRIPT token');
    }
    let instance = injector.get(WebWorkerInstance);
    spawnWebWorker(scriptUri, instance); 3
    initializeGenericWorkerRenderer(injector); 4
  };
}
```

It returns a function that makes the browser Dom adatper the current adapter 1; then gets the worker script from di 2; then calls spawnWebWorker 3; and finally calls initializeGenericWorkerRenderer 4. The spawnWebWorker function is defined as:

```
function spawnWebWorker(uri: string, instance: WebWorkerInstance): void {
  var webWorker: Worker = new Worker(uri);  1
  var sink = new PostMessageBusSink(webWorker);  2
  var source = new PostMessageBusSource(webWorker);
  var bus = new PostMessageBus(sink, source);

  instance.init(webWorker, bus);  3
}
```

It first creates a new webworker 1; then it create the message bus with its sinka dn source 2 and finally it calls WebWorkerInstance.init 3 which we have already seen. The initializeGenericWorkerRenderer function is defined as:

```
function initializeGenericWorkerRenderer(injector: Injector) {
  var bus = injector.get(MessageBus);
  let zone = injector.get(NgZone);
  bus.attachToZone(zone);

  // initialize message services after the bus has been created
  let services = injector.get(WORKER_UI_STARTABLE_MESSAGING_SERVICE);
```

```
    zone.runGuarded(() => {
         services.forEach((svc: any) => { svc.start(); }); });
  }
```

It first asks the dependency injector for a message bus and a zone and attached the bus to the zone. Then it also asks the dependency injector for a list of WORKER_UI_STARTABLE_MESSAGING_SERVICE (remember we noted it was configured as a multi provider), and for each service, starts it.

platform-webworker/src/web_workers

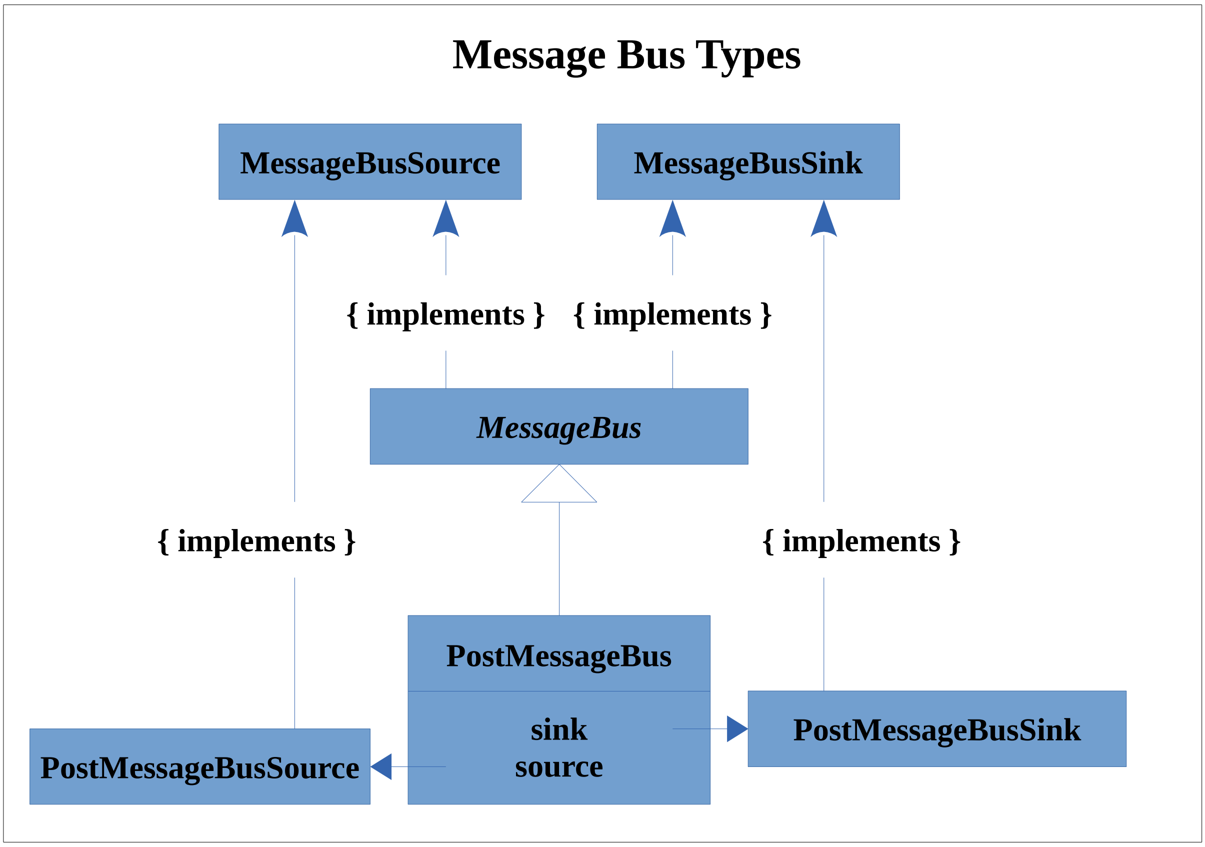
The source files for web_workers are provided in thre sub-directories, one of which has shared messaging functionality use both by the UI side and worker side and the communication, the second refers only to the UI side and the third only to the worker side.

- shared/api.ts
- shared/client_message_broker.ts
- shared/message_bus.ts
- shared/messaging_api.ts
- shared/post_message_bus.ts
- shared/render_store.ts
- shared/serialized_types.ts
- shared/serializer.ts
- shared/service_message_broker.ts
- ui/event_dispatcher.ts
- ui/event_serializer.ts
- ui/location_providers.ts
- ui/platform_location.ts
- ui/renderer.ts
- worker/event_deserializer.ts
- worker/location_providers.ts
- worker/platform_location.ts
- worker/renderer.ts
- worker/worker_adapter.ts

Communication between the UI thread and the webworker is handled by a low-level multi-channel message bus. The shared/messaging_api.ts file defines the names of the three channels used by Angular:

```
/**
 * All channels used by angular's WebWorker components are listed here.
 * You should not use these channels in your application code.
 */
export const RENDERER_CHANNEL = 'ng-Renderer';
export const EVENT_CHANNEL = 'ng-Events';
export const ROUTER_CHANNEL = 'ng-Router';
```

If you are familiar with TCP/IP, the channel name here serves the same purpose as a port number – it is needed to multiplex multiple independent message streams on a single data connection.



The message_bus.ts file defines an abstract MessageBus class and two interfaces, MessageBusSource and MessageBusSink. Let's look at the interfaces first. MessageBusSource is for incoming messages. This interface describes the functionality a message source is expected to supply. It has methods to initialize a channel based on a string name, to attache dto a zone, and to return an RxJS observable (EventEmitter) that can be observed in order to read the incoming messages.

```
export interface MessageBusSource {
  /**
   * Sets up a new channel on the MessageBusSource.
   * MUST be called before calling from on the channel.
   * If runInZone is true then the source will emit events inside
   * the angular zone. if runInZone is false then the source will emit
   * events inside the global zone.
   */
  initChannel(channel: string, runInZone: boolean): void;

  /**
   * Assigns this source to the given zone.
   * Any channels which are initialized with runInZone set to true
   * will emit events that will be executed within the given zone.
   */
  attachToZone(zone: NgZone): void;

  /**
   * Returns an {@link EventEmitter} that emits every time a message
```

```
     * is received on the given channel.
     */
    from(channel: string): EventEmitter<any>;
  }
```

Similarly, MessageBusSink is for outgoing messages. Again it allows a named channel to be initialized, attacing to a zone and returns a RxJS observer (EventEmitter) used to send messages:

```
  export interface MessageBusSink {
    /**
     * Sets up a new channel on the MessageBusSink.
     * MUST be called before calling to on the channel.
     * If runInZone is true the sink will buffer messages and send
     * only once the zone exits. if runInZone is false the sink will send
     * messages immediately.
     */
    initChannel(channel: string, runInZone: boolean): void;

    /**
     * Assigns this sink to the given zone.
     * Any channels which are initialized with runInZone set to true
     * will wait for the given zone to exit before sending messages.
     */
    attachToZone(zone: NgZone): void;

    /**
     * Returns an {@link EventEmitter} for the given channel
     * To publish methods to that channel just call next on
     * the returned emitter
     */
    to(channel: string): EventEmitter<any>;
  }
```

MessageBus is an abstract class that implements both MessageBusSource and MessageBusSink.

```
  /**
   * Message Bus is a low level API used to communicate between the UI
   * and the background.
   * Communication is based on a channel abstraction. Messages published in a
   * given channel to one MessageBusSink are received on the same channel
   * by the corresponding MessageBusSource.
   */
  export abstract class MessageBus implements MessageBusSource, MessageBusSink{
    /**
     * Sets up a new channel on the MessageBus.
     * MUST be called before calling from or to on the channel.
     * If runInZone is true then the source will emit events
     * inside the angular zone
     * and the sink will buffer messages and send only once the zone exits.
     * if runInZone is false then the source will emit events inside
     * the global zone and the sink will send messages immediately.
     */
    abstract initChannel(channel: string, runInZone?: boolean): void;

    /**
     * Assigns this bus to the given zone.
```

```
     * Any callbacks attached to channels where runInZone was set to true on
     *  initialization will be executed in the given zone.
     */
    abstract attachToZone(zone: NgZone): void;
    /**
     * Returns an {@link EventEmitter} that emits every time a message
     * is received on the given channel.
     */
    abstract from(channel: string): EventEmitter<any>;

    /**
     * Returns an {@link EventEmitter} for the given channel
     * To publish methods to that channel just call next on the returned
     *  emitter
     */
    abstract to(channel: string): EventEmitter<any>;
  }
```

So far the message bus description has only specified what the functionality that needs to be supplied. A single implementation is supplied, in post_message_bus.ts, based on the postMessage API. This defines three classes, PostMessageBusSource, PostMessageBusSink and PostMessageBus, that implement the above similarly named types.

A useful private class is supplied called _Channel that keeps track of two pieces of data:

```
  /**
   * Helper class that wraps a channel's {@link EventEmitter} and
   * keeps track of if it should run in the zone.
   */
  class _Channel {
    constructor(
       public emitter: EventEmitter<any>,
       public runInZone: boolean) {}
  }
```

We see its use of PostMessageBusSource initChannel:

```
  initChannel(channel: string, runInZone: boolean = true) {
      if (StringMapWrapper.contains(this._channels, channel)) {
        throw new Error(`${channel} has already been initialized`);
      }
      var emitter = new EventEmitter(false);
      var channelInfo = new _Channel(emitter, runInZone);
      this._channels[channel] = channelInfo;
    }
```

So we see a channel is nothing more that a name that maps to a _Channel, which is just an EventEmitter and a boolean (runInZone). PostMessageBusSource manages a map of these called _channels:

```
  export class PostMessageBusSource implements MessageBusSource {
    private _zone: NgZone;
    private _channels: {[key: string]: _Channel}
```

The from() method just returns the appropriate channel emitter:

```
from(channel: string): EventEmitter<any> {
    if (StringMapWrapper.contains(this._channels, channel)) {
      return this._channels[channel].emitter;
    } else {..}
  }
```

The constructor calls addEventListener to add an event listener:

```
constructor(eventTarget?: EventTarget) {
    if (eventTarget) {
      eventTarget.addEventListener('message',
                         (ev: MessageEvent) => this._handleMessages(ev));
    } else {
      // if no eventTarget is given we assume we're in a
      // WebWorker and listen on the global scope
      const workerScope = <EventTarget>self;
      workerScope.addEventListener('message',
                         (ev: MessageEvent) => this._handleMessages(ev));
    }
  }
```

The _handleMessages passes on each message to _handleMessage, which emits it on the approapriate channel:

```
private _handleMessage(data: any): void {
    var channel = data.channel;
    if (StringMapWrapper.contains(this._channels, channel)) {
      var channelInfo = this._channels[channel];
      if (channelInfo.runInZone) {
        this._zone.run(() => { channelInfo.emitter.emit(data.message); });
      } else {
        channelInfo.emitter.emit(data.message);
      }
    }
  }
```

The PostMessageBusSink implementation is slightly different because it needs to use postMessageTarget to post messages. Its constructor creates a field based on the supplied PostMessageTarget parameter:

```
export class PostMessageBusSink implements MessageBusSink {
  private _zone: NgZone;
  private _channels: {[key: string]: _Channel} = StringMapWrapper.create();
  private _messageBuffer: Array<Object> = [];

  constructor(private _postMessageTarget: PostMessageTarget) {}
```

The initChannel method subscribes to the emitter with a next handler that either (when runnign inside the Angular zone) adds the message to the messageBuffer where its sending is deferred, or (if running outside the Angualr zone), calls _sendMessages, to immediately send the message:

```
initChannel(channel: string, runInZone: boolean = true): void {
    if (StringMapWrapper.contains(this._channels, channel)) {
      throw new Error(`${channel} has already been initialized`);
    }
```

```
      var emitter = new EventEmitter(false);
      var channelInfo = new _Channel(emitter, runInZone);
      this._channels[channel] = channelInfo;
      emitter.subscribe((data: Object) => {
        var message = {channel: channel, message: data};
        if (runInZone) {
          this._messageBuffer.push(message);
        } else {
          this._sendMessages([message]);
        }
      });
    }
```

_sendMessages() just sends the message array via PostMessageTarget:

```
   private _sendMessages(messages: Array<Object>) {
            this._postMessageTarget.postMessage(messages); }
```

So we saw with initChannel that the subscription to the emitter either calls _sendMessages immediately or parks the message in a message buffer, for later transmission. So two questions arise – what triggers that transmission and how does it work. Well, to answer the second question first, _sendMessages() is also called for the bulk transmission, from inside the _handleOnEventDone() message:

```
  private _handleOnEventDone() {
      if (this._messageBuffer.length > 0) {
        this._sendMessages(this._messageBuffer);
        this._messageBuffer = [];
      }
    }
```

So, what calls handleOnEventDone()? Let’s digress to look at the NgZone class in

- <ANGULAR2>/modules/@angular/core/src/zone/ngzone.ts

which has this getter:

```
  /**
   * Notifies when the last `onMicrotaskEmpty` has run and there are no more
   * microtasks, which implies we are about to relinquish VM turn.
   * This event gets called just once.
   */
   get onStable(): EventEmitter<any> { return this._onStable; }
```

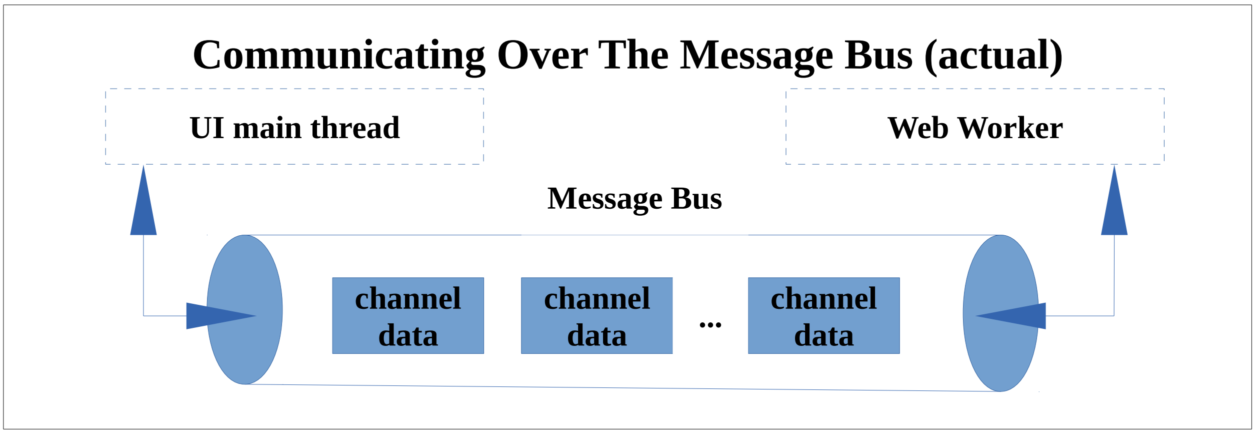
So when the zone has no more work to immedaitely carry out, it emits a message via onStable. Back to PostMessageBusSink – which has this code, that subscribes to the onStable event emitter:

```
  attachToZone(zone: NgZone): void {
      this._zone = zone;
      this._zone.runOutsideAngular(
          () => { this._zone.onStable.subscribe(
                                {next: () => { this._handleOnEventDone(); }}); });
    }
```

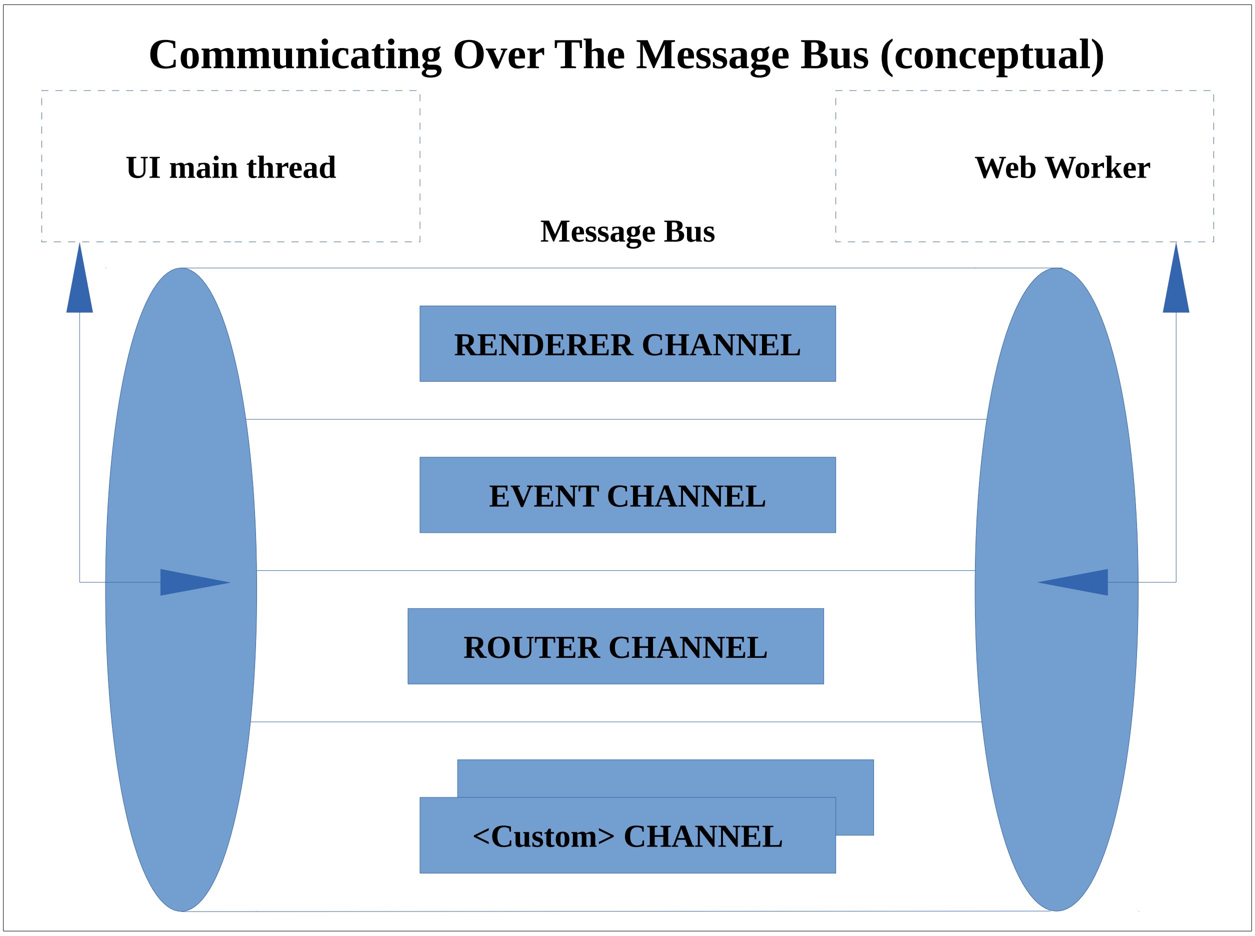
With all the hard work done in PostMessageBusSource and PostMessageBusSink, the implementation of PostMessageBus is quite simple:

```
  @Injectable()
```

```
export class PostMessageBus implements MessageBus {
  constructor(
     public sink: PostMessageBusSink,
     public source: PostMessageBusSource) {}

  attachToZone(zone: NgZone): void {
    this.source.attachToZone(zone);
    this.sink.attachToZone(zone);
  }

  initChannel(channel: string, runInZone: boolean = true): void {
    this.source.initChannel(channel, runInZone);
    this.sink.initChannel(channel, runInZone);
  }

  from(channel: string): EventEmitter<any> {
     return this.source.from(channel); }

  to(channel: string): EventEmitter<any> { return this.sink.to(channel); }
}
```



A different way of looking at the message bus is as a set of independent channels:



In summary, the message bus in Angular is a simple efficient messaging passing mechanism with webworkers. It is based on a single connection, with opaque messages consisting of a channel name (string) and message data:

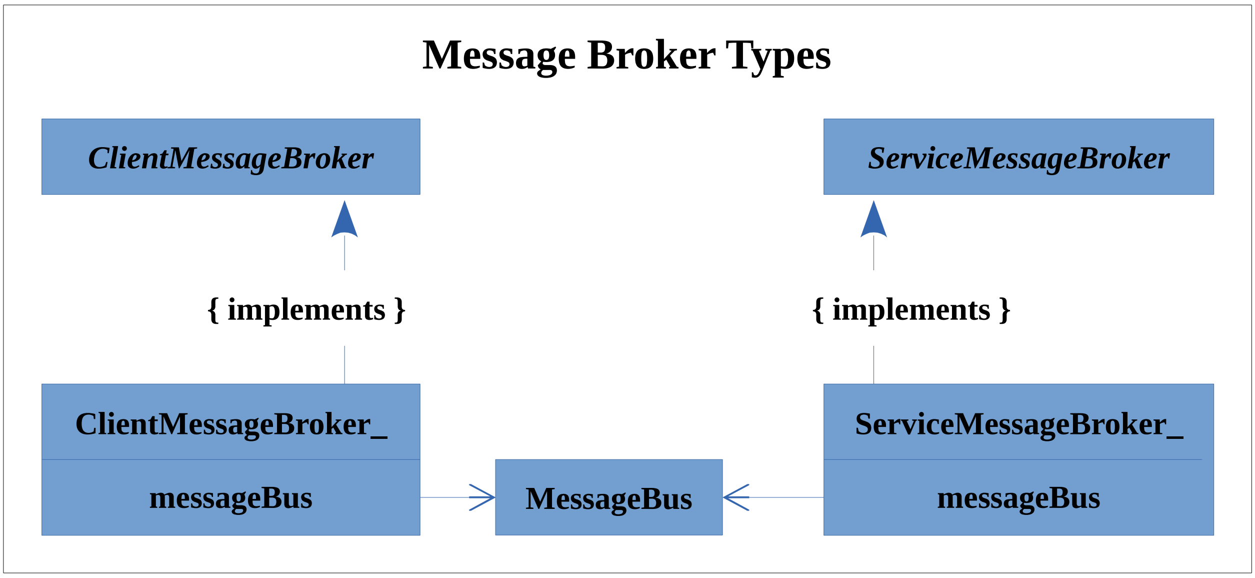
```
var msg = {channel: channel, message: data};
```

Angular also supplies a richer message broker layered above this simple message bus. The files client_message_broker.ts and service_message_broker.ts along with a number of helper files for serialization implement the message broker.

The service_message_broker.ts file defines the ReceivedMessage class that represents a message:

```
export class ReceivedMessage {
  method: string;
  args: any[];
  id: string;
  type: string;

  constructor(data: {[key: string]: any}) {
    this.method = data['method'];
    this.args = data['args'];
    this.id = data['id'];
```

```
    this.type = data['type'];
  }
}
```



The abstract class ServiceMessageBrokerFactory and its implementation ServiceMessageBrokerFactory_ provide a factory for the service message broker:

```
 export abstract class ServiceMessageBrokerFactory {
  /**
   * Initializes the given channel and attaches a new
   * {@link ServiceMessageBroker} to it.
   */
  abstract createMessageBroker(
       channel: string, runInZone?: boolean): ServiceMessageBroker;
}


@Injectable()
export class ServiceMessageBrokerFactory_ extends ServiceMessageBrokerFactory
{
  /** @internal */
  _serializer: Serializer;

  constructor(private _messageBus: MessageBus, _serializer: Serializer) {
    super();
    this._serializer = _serializer;
  }

  createMessageBroker(
     channel: string, runInZone: boolean = true): ServiceMessageBroker {
    this._messageBus.initChannel(channel, runInZone);
    return new ServiceMessageBroker_(
                     this._messageBus, this._serializer, channel);
  }
}
```

The service message broker is created based on the supplied message bus, serializer and channel name. The abstract ServiceMessageBroker class contains just one abstract class declaration, registerMethod():

```
export abstract class ServiceMessageBroker {
  abstract registerMethod(
       methodName: string,
       signature: Type<any>[],
       method: Function, returnType?: Type<any>): void;
}
```

ServiceMessageBroker_ provides an implementation for it:

```
export class ServiceMessageBroker_ extends ServiceMessageBroker {
  private _sink: EventEmitter<any>;
  private _methods: Map<string, Function> = new Map<string, Function>();

  constructor(
       messageBus: MessageBus,
       private _serializer: Serializer,
       public channel: any) {
    super();
    this._sink = messageBus.to(channel);
    var source = messageBus.from(channel);
    source.subscribe({next: (message: any) => this._handleMessage(message)});
  }
```

It has two private fields, an event emitter _sink and a map from string to function _methods. In its constructor it subscribes its internal method _ handleMessage to messageBus.from (this handles incoming messages), and set _sink to messageBus.to (this will be used to send messages).

_handleMessage() message creates a ReceivedMessage based on the map parameter, and then if message.method is listed as a supported message in _methods, looks up _methods for the appropriate function to execute, to handle the message:

```
  private _handleMessage(map: {[key: string]: any}): void {
    var message = new ReceivedMessage(map);
    if (this._methods.has(message.method)) {
      this._methods.get(message.method)(message);
    }
  }
```

The next question is how methods get registered in _methods. That is the job of registerMethod(), whose implementation adds an entry to the _methods map based on the supplied parameters:

```
  registerMethod(
      methodName: string,
      signature: Type<any>[],
      method: (..._: any[]) => Promise<any>| void,
      returnType?: Type<any>): void {
    this._methods.set(methodName, (message: ReceivedMessage) => { .. });
```

Its key is the methodName supplied as a parameter and its value is an anonymous method, with a single parameter, message, of type ReceivedMessage, which is defined as:

```
(message: ReceivedMessage) => {
      var serializedArgs = message.args;
      let numArgs = signature === null ? 0 : signature.length;
      var deserializedArgs: any[] = ListWrapper.createFixedSize(numArgs);
      for (var i = 0; i < numArgs; i++) {
        var serializedArg = serializedArgs[i];
  1     deserializedArgs[i] =
            this._serializer.deserialize(serializedArg, signature[i]);
      }

  2   var promise = FunctionWrapper.apply(method, deserializedArgs);
      if (isPresent(returnType) && isPresent(promise)) {
        this._wrapWebWorkerPromise(message.id, promise, returnType);
      }
```

We see at its 1 deserialization occuring with the help of the serializer, and at 2 method is called with the deserialized arguments, and at 3 we see if a returnType is needed, _ wrapWebWorkerPromise is called, to handle the then of the promise, which emits the result to the sink:

```
    private _wrapWebWorkerPromise(
         id: string, promise: Promise<any>, type: Type<any>): void {
      promise.then((result: any) => {
        this._sink.emit(
         {'type' : 'result',
          'value': this._serializer.serialize(result, type), 'id': id});
      });
    }
}
```

The client side of this messaging relationship is handled by client_message_broker.ts. It defines some simple helper types:

```
interface PromiseCompleter {
  resolve: (result: any) => void;
  reject: (err: any) => void;
}
class MessageData {
  type: string;
  value: any;
  id: string;
  constructor(data: {[key: string]: any}) { .. }
}
export class FnArg {
  constructor(public value: any, public type: Type<any>) {}
}
export class UiArguments {
  constructor(public method: string, public args?: FnArg[]) {}
}
```

To construct a client emssage borker it declares the abstract ClientMessageBrokerFactory with a single method:

```
export abstract class ClientMessageBrokerFactory {
```

```
  abstract createMessageBroker(
         channel: string, runInZone?: boolean): ClientMessageBroker; }
```

This is implemented by ClientMessageBrokerFactory_, which, in its createMessageBroker, method, calls initChannel on the message bus and returns a new ClientMessageBroker_:

```
@Injectable()
export class ClientMessageBrokerFactory_ extends ClientMessageBrokerFactory {
  /** @internal */
  _serializer: Serializer;
  constructor(private _messageBus: MessageBus, _serializer: Serializer) {
    super();
    this._serializer = _serializer;
  }
  //Initialize the given channel and attaches a new ClientMessageBroker to it
  createMessageBroker(
      channel: string, runInZone: boolean = true): ClientMessageBroker {
    this._messageBus.initChannel(channel, runInZone);
    return new ClientMessageBroker_(
                    this._messageBus, this._serializer, channel);
  }
}
```

ClientMessageBroker has a single method, run on service, which calls a method with the supplied UiArguments on the remote service side and returns a promise with the return value (if any)

```
export abstract class ClientMessageBroker {
  abstract runOnService(
                   args: UiArguments, returnType: Type<any>): Promise<any>;
}
```

ClientMessageBroker_ implements this as follows:

```
export class ClientMessageBroker_ extends ClientMessageBroker {
  private _pending: Map<string, PromiseCompleter> =
                                     new Map<string, PromiseCompleter>();
  private _sink: EventEmitter<any>;
  /** @internal */
  public _serializer: Serializer;

  constructor(
      messageBus: MessageBus, _serializer: Serializer, public channel: any){
    super();
    this._sink = messageBus.to(channel);
    this._serializer = _serializer;
    var source = messageBus.from(channel);

    source.subscribe(
    {next: (message: {[key: string]: any}) => this._handleMessage(message)});
  }
```

It has three fields, _pending, _sink and _serializer. _pending is a map from string to PromiseCompleter. It is used to keep track of method calls that require a return value and are outstanding – the message has been set to the service, and the result is awaited. In its constructor _sink is set to the messageBus.to and serializer set to the Serializer parameter. Also a source subscription is set for _handleMessage.

Messages for which a return value is expected have a message id generated for them via:

```
private _generateMessageId(name: string): string {
  var time: string = stringify(DateWrapper.toMillis(DateWrapper.now()));
  var iteration: number = 0;
  var id: string = name + time + stringify(iteration);
  while (isPresent((this as any)._pending[id])) {
    id = `${name}${time}${iteration}`;
    iteration++;
  }
  return id;
}
```

It is this string that is the key into the _pending map. We will now see how it is set up in runOnService and used in _handleMessage. RunOnService is what the client calls when it wants the service to execute a method. It returns a promise, which is a return value is required, it is completed when the service returns it.

Let's first examine runOnService when returnType is null. This creates an array of serialized arguments in fnArgs 1, sets up an object literal called message with properties "method" and "args" 2, and then calls _sink.emit(message) 3:

```
  runOnService(args: UiArguments, returnType: Type<any>): Promise<any> {
    var fnArgs: any[] /** TODO #9100 */ = [];
    if (isPresent(args.args)) {
      args.args.forEach(argument => {
        if (argument.type != null) {
1         fnArgs.push(
                this._serializer.serialize(argument.value, argument.type));
        } else {
          fnArgs.push(argument.value);
        }
      });
    }
    var promise: Promise<any>;
    var id: string = null;
    if (returnType != null) { .. }
2   var message = {'method': args.method, 'args': fnArgs};
    ..
3   this._sink.emit(message);
    return promise;
  }
```

Things are a little more complex when a return value is required. A promise and a promise completer are created 1; _generateMessageId() is called 2 to generate a unique message id for this message; an entry is made into _pending 3, whose key is the id and whose value is the promise completer. The then of the promise 4 returns the result 5a (deserialized if needed 5b). Before the message is sent via sink.emit, the generated message id is attached to the message 6.

```
if (returnType != null) {
1   let completer: PromiseCompleter;
    promise =
      new Promise((resolve, reject) => { completer = {resolve, reject}; });
2   id = this._generateMessageId(args.method);
```

```
3    this._pending.set(id, completer);
     promise.catch((err) => {
       print(err);
       completer.reject(err);
     });

4    promise = promise.then((value: any) => {
       if (this._serializer == null) {
5a        return value;
       } else {
5b        return this._serializer.deserialize(value, returnType);
       }
     });
   } else {
     promise = null;
   }
var message = {'method': args.method, 'args': fnArgs};
   if (id != null) {
6    (message as any)['id'] = id;
   }
   this._sink.emit(message);
```

The _handleMessage method creates a MessageData instance 1 based on the supplied message. It extracts the message id, which it uses to look up _pending 2 for the promise. If message data has a result field, this is used in a call to the promise.resolve 3, otherwise promise.reject 4 is called:

```
  private _handleMessage(message: {[key: string]: any}): void {
1   var data = new MessageData(message);
    if (StringWrapper.equals(data.type, 'result') ||
                        StringWrapper.equals(data.type, 'error')) {
      var id = data.id;
2     if (this._pending.has(id)) {
        if (StringWrapper.equals(data.type, 'result')) {
3         this._pending.get(id).resolve(data.value);
        } else {
4         this._pending.get(id).reject(data.value);
        }
        this._pending.delete(id);
      }
    }
  }
```

Three helper files are involved with serialization – render_store.ts, serializer.ts and serialized_types.ts. The RenderStore class, define in render_store.ts, maps two maps – the first, _lookupById, from number to any, and the second, lookupByObject, from any to number. It supplies methods to store and remove objects and serialize and de-serialize them. The serialized_types.ts file simply has:

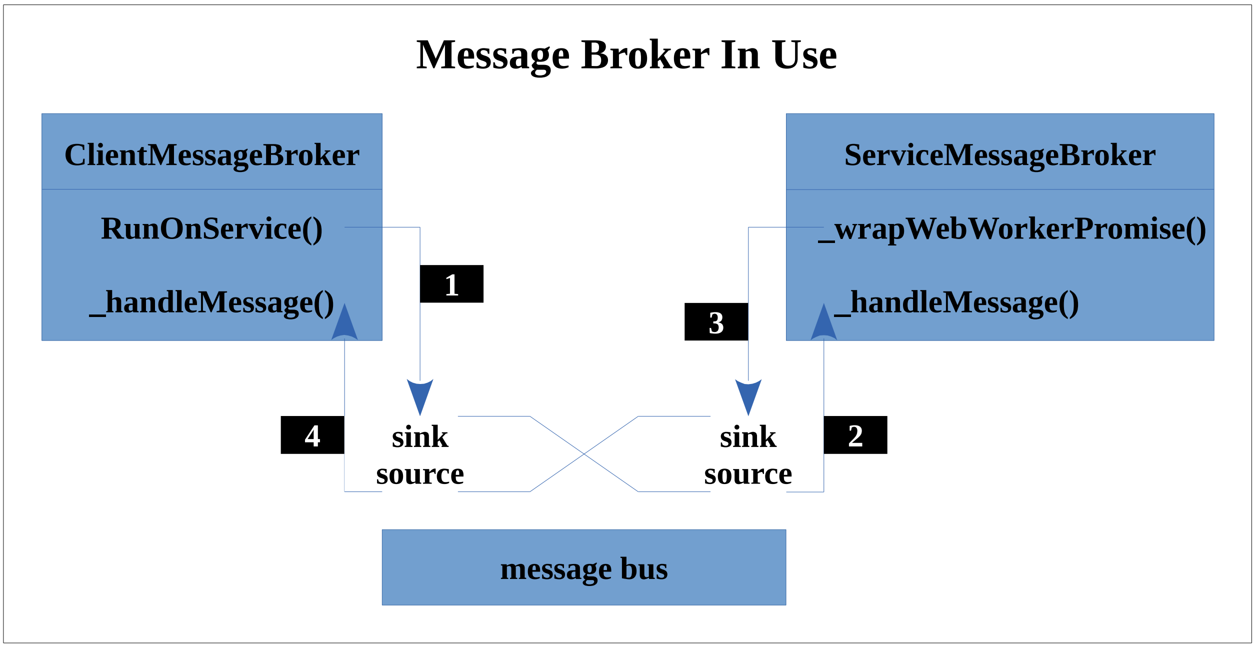
```
// This file contains interface versions of browser types
// that can be serialized to Plain Old JavaScript Objects
export class LocationType {
  constructor(
     public href: string,
     public protocol: string,
     public host: string,
     public hostname: string,
```

```
      public port: string,
      public pathname: string,
      public search: string,
      public hash: string,
      public origin: string) {}
  }
```

The Serializer class is defined in serializer.ts and has methods to serialize and deserialize. The serialize method is:

```
@Injectable()
export class Serializer {
  constructor(private _renderStore: RenderStore) {}

  serialize(obj: any, type: any): Object {
    if (!isPresent(obj)) {
      return null;
    }
    if (isArray(obj)) {
      return (<any[]>obj).map(v => this.serialize(v, type));
    }
    if (type == PRIMITIVE) {
      return obj;
    }
    if (type == RenderStoreObject) {
      return this._renderStore.serialize(obj);
    } else if (type === RenderComponentType) {
      return this._serializeRenderComponentType(obj);
    } else if (type === ViewEncapsulation) {
      return serializeEnum(obj);
    } else if (type === LocationType) {
      return this._serializeLocation(obj);
    } else {
      throw new Error('No serializer for ' + type.toString());
    }
  }
```

Based on the type an appropriate serialization helper method is called.



The last file in the shared directory is api.ts, which has this one line:

```
export const ON_WEB_WORKER = new OpaqueToken('WebWorker.onWebWorker');
```

It is used to store a boolean value with dependency injection, stating whether the current code is running in a webworker or not. At this point it would be helpful to review how shared functionality is actually configured with dependency injection. On the worker side:

- <ANGULAR2>/modules/@angular/platfrom-browser/src/worker_app.ts

has this:

```
@NgModule({
providers: [
Serializer,
{provide: ClientMessageBrokerFactory, useClass: ClientMessageBrokerFactory_},
{provide: ServiceMessageBrokerFactory, useClass: ServiceMessageBrokerFactory_},
{provide: ON_WEB_WORKER, useValue: true},
RenderStore,
{provide: MessageBus, useFactory: createMessageBus, deps: [NgZone]},
..
],
  exports: [CommonModule, ApplicationModule]
})
export class WorkerAppModule { }
```

Note ON_WEB_WORKER is set to true.

On the ui main thread side:

- <ANGULAR2>/modules/@angular/platfrom-browser/src/worker_render.ts

has this:

```
export const _WORKER_UI_PLATFORM_PROVIDERS: Provider[] = [
```

```
  {provide: WORKER_UI_STARTABLE_MESSAGING_SERVICE, useExisting:
MessageBasedRenderer, multi: true},
  {provide: ServiceMessageBrokerFactory, useClass:
ServiceMessageBrokerFactory_},
  {provide: ClientMessageBrokerFactory, useClass:
ClientMessageBrokerFactory_},
  Serializer,
  {provide: ON_WEB_WORKER, useValue: false},
  RenderStore,
  {provide: MessageBus, useFactory: messageBusFactory, deps:
[WebWorkerInstance]}
  ..
];

export const platformWorkerUi =
    createPlatformFactory(
              platformCore, 'workerUi', _WORKER_UI_PLATFORM_PROVIDERS);
```

Note ON_WEB_WORKER is set to false.

Now we will move on to looking at the worker-specific code in:

- <ANGULAR2>/modules/@angular/platfrom-browser/src/web_workers/worker

In worker_adapter.ts, WorkerDomAdapter extends WorkerDomAdapter but only implements the logging functionality – for everything else an exception is raised. WorkerDomAdapter is not how workers render, instead is just logs data to the console.

The renderer.ts file supplies the WebWorkerRootRenderer and WebWorkerRenderer classes and this is where worker-based rendering is managed.

WebWorkerRenderer implements the Renderer API and forwards all calls to runOnService from the root renderer. WebWorkerRenderer is runnining the webworkerr where there is no DOM, so any rendering needs to be forwarded to the main UI thread, and that is what runOnService is doing here. This is a sampling of the calls:

```
export class WebWorkerRenderer implements Renderer, RenderStoreObject {
  constructor(
      private _rootRenderer: WebWorkerRootRenderer,
      private _componentType: RenderComponentType) {}

  private _runOnService(fnName: string, fnArgs: FnArg[]) {
    var fnArgsWithRenderer =
     [new FnArg(this, RenderStoreObject)].concat(fnArgs);
    this._rootRenderer.runOnService(fnName, fnArgsWithRenderer);
  }

  selectRootElement(selectorOrNode: string, debugInfo?:RenderDebugInfo):any {
    var node = this._rootRenderer.allocateNode();
    this._runOnService(
        'selectRootElement', [new FnArg(selectorOrNode, null),
        new FnArg(node, RenderStoreObject)]);
    return node;
  }
```

```
createElement(parentElement: any,
     name: string,
     debugInfo?: RenderDebugInfo): any {
  var node = this._rootRenderer.allocateNode();
  this._runOnService('createElement', [
    new FnArg(parentElement, RenderStoreObject), new FnArg(name, null),
    new FnArg(node, RenderStoreObject)
  ]);
  return node;
}

createViewRoot(hostElement: any): any {
  var viewRoot = this._componentType.encapsulation ===
                    ViewEncapsulation.Native
                    ? this._rootRenderer.allocateNode()
                    : hostElement;
  this._runOnService(
      'createViewRoot',
      [new FnArg(hostElement, RenderStoreObject),
                    new FnArg(viewRoot, RenderStoreObject)]);
  return viewRoot;
}
```

WebWorkerRootRenderer implements RootRenderer by setting up the client message broker factory.

```
@Injectable()
export class WebWorkerRootRenderer implements RootRenderer {
  private _messageBroker: any /** TODO #9100 */;
  public globalEvents: NamedEventEmitter = new NamedEventEmitter();
  private _componentRenderers: Map<string, WebWorkerRenderer> =
      new Map<string, WebWorkerRenderer>();
  constructor(
      messageBrokerFactory: ClientMessageBrokerFactory, bus: MessageBus,
      private _serializer: Serializer, private _renderStore: RenderStore) {
    this._messageBroker =
                messageBrokerFactory.createMessageBroker(RENDERER_CHANNEL);
1   bus.initChannel(EVENT_CHANNEL);
2   var source = bus.from(EVENT_CHANNEL);
3   source.subscribe({next: (message: any) => this._dispatchEvent(message)});
  }
```

An additional and very important role of WebWorkerRootRenderer is to configure event handling. We see at 1 initChannel being called for the EVENT_CHANNEL, at 2 the message source being accessed, and at 3 a subscription being set up with the _dispatchEvent() method, which is implemented as:

```
private _dispatchEvent(message: {[key: string]: any}): void {
  var eventName = message['eventName'];
  var target    = message['eventTarget'];
  var event = deserializeGenericEvent(message['event']);
  if (isPresent(target)) {
    this.globalEvents.dispatchEvent(
            eventNameWithTarget(target, eventName), event);
  } else {
    var element = <WebWorkerRenderNode>this._serializer.deserialize(
                                    message['element'], RenderStoreObject);
```

```
        element.events.dispatchEvent(eventName, event);   } }
```

The ui specific code is in:

- <ANGULAR2>/modules/@angular/platfrom-browser/src/web_workers/ui

The renderer.ts file implements the MessageBasedRenderer class. Note this class, despite its name, does not implement any types from Core's Renderering API. Instead, it accepts a RootRenderer instance as a constructor parameter, and forwards all rendering requests to it. Its constructor also has a few other parameters which it uses as fields and it also defines one extra, an EventDispatcher.

```
@Injectable()
export class MessageBasedRenderer {
  private _eventDispatcher: EventDispatcher;

  constructor(
      private _brokerFactory: ServiceMessageBrokerFactory,
      private _bus: MessageBus,
      private _serializer: Serializer,
      private _renderStore: RenderStore,
      private _rootRenderer: RootRenderer) {}
```

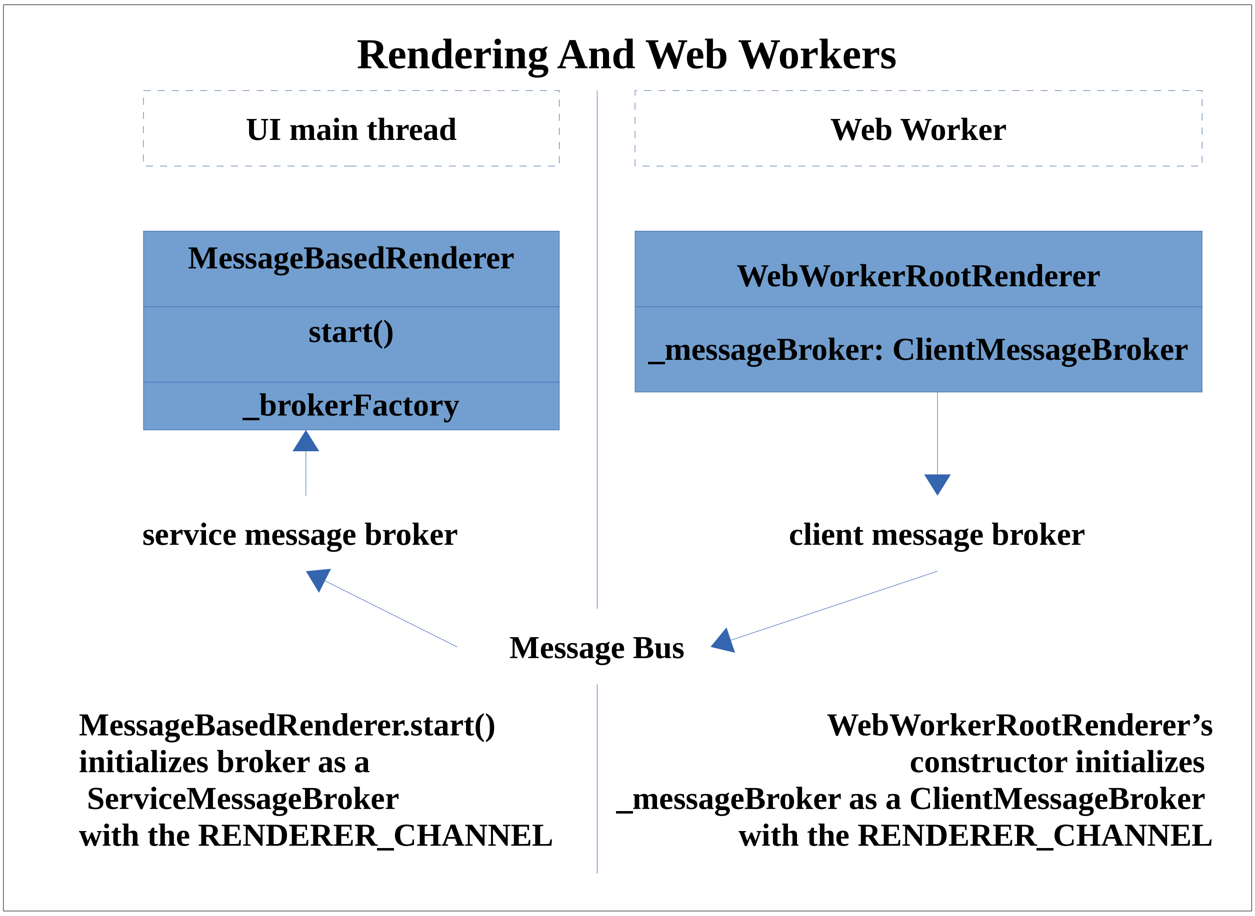
Its start() method initializes the EVENT_CHANNEL, creates a new EventDispatcher, and then has a very long list of calls to registerMethod – when such messages are received, then the configured local method is called. Here is a short selection of the registerMethod calls:

```
  start(): void {
    var broker = this._brokerFactory.createMessageBroker(RENDERER_CHANNEL);
    this._bus.initChannel(EVENT_CHANNEL);
    this._eventDispatcher =
         new EventDispatcher(this._bus.to(EVENT_CHANNEL), this._serializer);

    broker.registerMethod(
        'renderComponent',
        [RenderComponentType, PRIMITIVE],
        FunctionWrapper.bind(this._renderComponent, this));
    broker.registerMethod(
        'selectRootElement',
        [RenderStoreObject, PRIMITIVE, PRIMITIVE],
        FunctionWrapper.bind(this._selectRootElement, this));
    broker.registerMethod(
        'createElement',
        [RenderStoreObject, RenderStoreObject, PRIMITIVE, PRIMITIVE],
        FunctionWrapper.bind(this._createElement, this));
    broker.registerMethod(
        'createViewRoot',
       [RenderStoreObject, RenderStoreObject, PRIMITIVE],
        FunctionWrapper.bind(this._createViewRoot, this));
```

The local methods usually call the qequivalent method in the configured renderer, and often stores the result in the RenderStore. Here is _createElement():

```
private _createElement(
       renderer: Renderer, parentElement: any, name: string, elId: number) {
    this._renderStore.store(
       renderer.createElement(parentElement, name, null), elId);
```

```
}
```



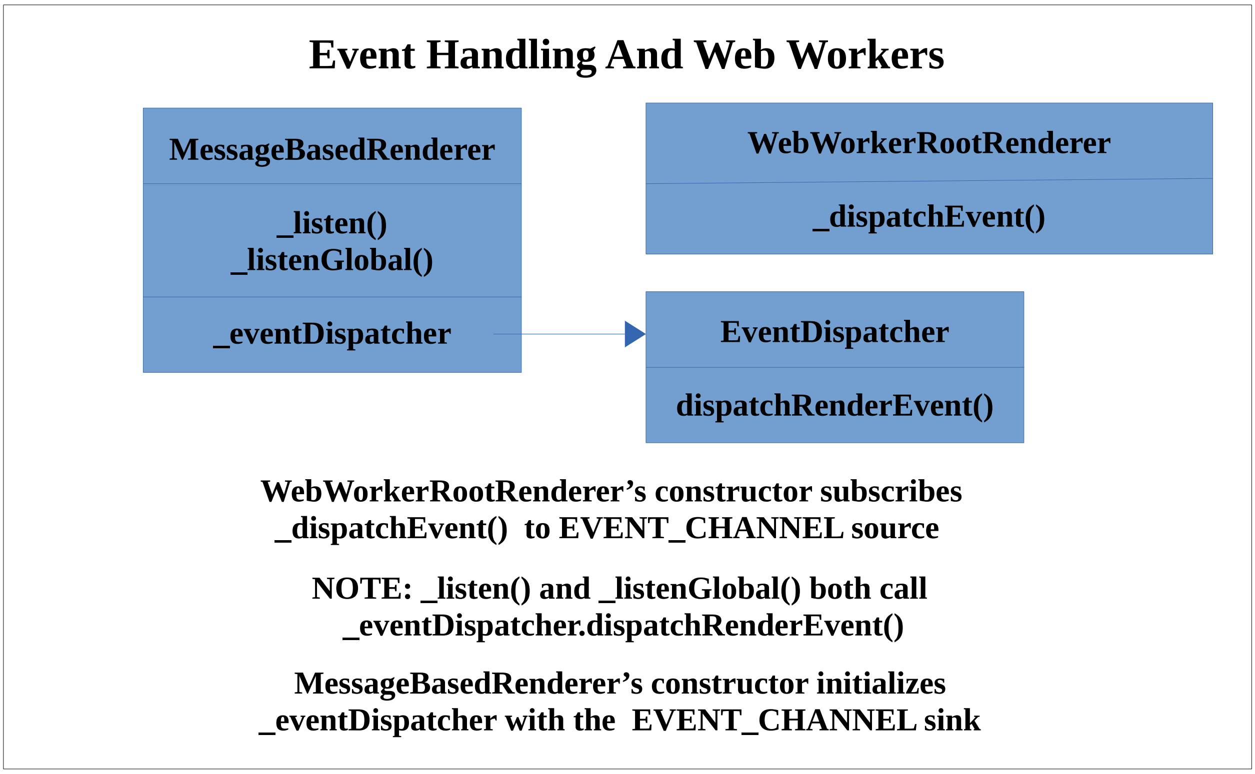
For event listening, the event dispatcher is used:

```
private _listen(renderer: Renderer, renderElement: any,
      eventName: string, unlistenId: number) {
    var unregisterCallback = renderer.listen(
        renderElement, eventName,
        (event: any) =>
            this._eventDispatcher.dispatchRenderEvent(
              renderElement, null, eventName, event));
    this._renderStore.store(unregisterCallback, unlistenId);
}

private _listenGlobal(renderer: Renderer, eventTarget:
      string, eventName: string, unlistenId: number) {
    var unregisterCallback = renderer.listenGlobal(
        eventTarget, eventName,
        (event: any) =>
            this._eventDispatcher.dispatchRenderEvent(
              null, eventTarget, eventName, event));
    this._renderStore.store(unregisterCallback, unlistenId);
}
```

The EventDispatcher class is defined in event_dispatcher.ts and has a constructor and one methods, dispatchRenderEvent.

```
export class EventDispatcher {
  constructor(
     private _sink: EventEmitter<any>,
     private _serializer: Serializer) {}

  dispatchRenderEvent(element: any, eventTarget: string,
       eventName: string, event: any): boolean {
    var serializedEvent: any;

    switch (event.type) {
       ...
       case 'keydown':
       case 'keypress':
       case 'keyup':
         serializedEvent = serializeKeyboardEvent(event);
         break;
       ...
    }
    this._sink.emit({
      'element': this._serializer.serialize(element, RenderStoreObject),
      'eventName': eventName,
      'eventTarget': eventTarget,
      'event': serializedEvent
    });

   return false;
  }
}
```

The switch in the middle is a long list of all the supported events and appropriate calls to an event serializer. Above we show what is has for keyboard events.


Event Handling And Web Workers
MessageBasedRenderer
_listen()
_listenGlobal()
_eventDispatcher
WebWorkerRootRenderer
_dispatchEvent()
EventDispatcher
dispatchRenderEvent()
WebWorkerRootRenderer's constructor subscribes _dispatchEvent() to EVENT_CHANNEL source
NOTE: _listen() and _listenGlobal() both call _eventDispatcher.dispatchRenderEvent()
MessageBasedRenderer's constructor initializes _eventDispatcher with the EVENT_CHANNEL sink

10: @Angular/Platform-WebWorker-Dynamic

Overview

Platform-WebWorkers-Dynamic is the smallest of all the Angular packages. It define one const, `platformWorkerAppDynamic`, which is a call to `createPlatformFactory()`.

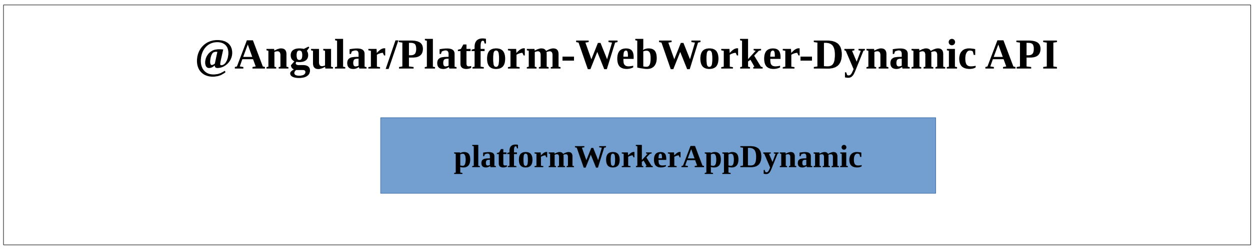
The reason to manage this as a separate package is this one line from package.json:

```
"peerDependencies": { .."@angular/compiler": "0.0.0-PLACEHOLDER", .. },
```

It brings in the runtime compiler, which is quite large. We wish to avoid this if it not used (it is not needed in the non-dynamic packages, which use the offline compiler).

Platform-WebWorker-Dynamic API

The exported API of the @Angular/Platform-WebWorker-Dynamic package can be represented as:



Source Tree Layout

The source tree for the Platform-WebWorker-Dynamic package contains these directories:

- src

Currently there are no test nor testing sub-directories. It also has these files:

- index.ts
- package.json
- rollup.config.js
- tsconfig.json

The index.ts file is simply:

```
export * from './src/platform-webworker-dynamic';
```

Source

Platform-webworker-dynamic just has this code:

```
export const platformWorkerAppDynamic = createPlatformFactory(
    platformCoreDynamic, 'workerAppDynamic', [{
      provide: COMPILER_OPTIONS,
      useValue: {providers:
                   [{provide: ResourceLoader, useClass: ResourceLoaderImpl}]},
      multi: true
    }]);
```

11: @Angular/Platform-Server

Overview

Platform-Server represents a platform when the application is running on a server.

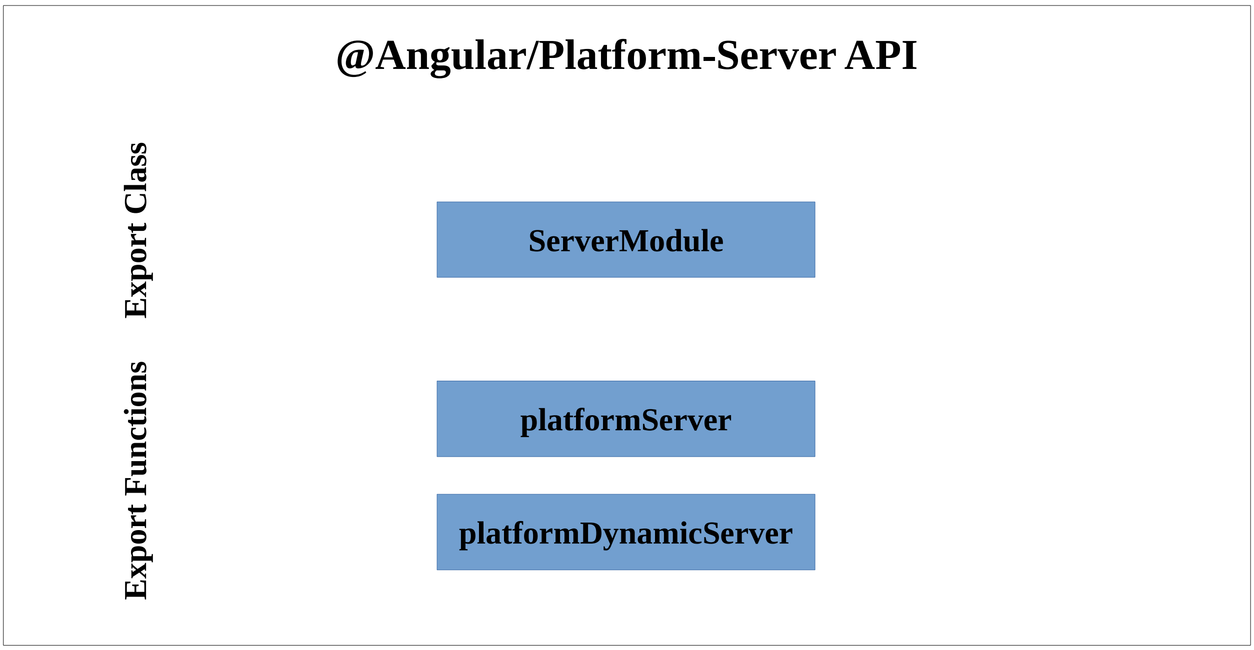
Most of the time Angular applications run in the browser and use either Platform-Browser (with the offline template compiler) or Platform-Browser-Dynamic (with the runtime template compiler). Running Angular applications on the server is a more specialist deployment. It can be of interest for search engine optimization and for pre-rendering output, which is later downloaded to browsers (this can speed up initial display of content to users for some types of applications; and also ease development with backend databases that might not be exposed via REST API to browser code). Platform-Server is used by Angular Universal as its platform module.

When considering Platform-Server, there are two questions the curious software engineer might ponder. Firstly, we wonder, if for the browser there are dynamic and non-dynamic versions of the platform, why not for the server? The answer to this is that for the browser, one wishes to support low-end devices serviced by poor network connections, so reducing the burden on the browser is of great interest (hence using the offline template compiler only); but one assumes the server has ample disk space and RAM and a small amount of extra code is not an issue, so bundling both kinds of server platforms (one that uses the offline template compiler, the other - "dynamic" - that uses the runtime template compiler) in the same module simplifies matters.

The second question is how rendering works on the server (where there is no DOM)? The answer is the rendered output is written via the Angular renderer API to an HTML file, which then can be downloaded to a browser or made available to a search engine - we need to explore how this rendering works.

Platform-Server API

The exported API of the @Angular/Platform-Server package can be represented as:



It’s index.ts is simply:

```
export {
   ServerModule, platformDynamicServer, platformServer} from './src/server';
```

`ServerModule` represents the `NgModule` for this module.

`platfromServer()` uses the offline compiler. `plaformDynamicServer()` uses the runtime compiler. Both are calls to the Core Module’s `createPlatformFactory()`.

Source Tree Layout

The source tree for the Platform-Server package contains these directories:

- src
- test (unit tests in Jasmine)
- testing (test tooling)

and these files:

- index.ts
- package.json
- rollup.config.js
- testing.ts
- tsconfig.json

Package.json has dependencies listed as:

```
"peerDependencies": {
  "@angular/core": "0.0.0-PLACEHOLDER",
  "@angular/common": "0.0.0-PLACEHOLDER",
  "@angular/compiler": "0.0.0-PLACEHOLDER",
  "@angular/platform-browser": "0.0.0-PLACEHOLDER"
},
"dependencies": {
  "parse5": "1.3.2"
},
```

The main directory for Platform-Server also contains these private import/export files:

- compiler_private.ts
- core_private.ts
- platform_browser_dynamic_testing_private.ts
- platform_browser_private.ts

As an example, let’s look at core_private.ts:

```
import {__core_private__ as r} from '@angular/core';

export var reflector: typeof r.reflector = r.reflector;
export var ReflectionCapabilities:
             typeof r.ReflectionCapabilities = r.ReflectionCapabilities;
export var wtfInit: typeof r.wtfInit = r.wtfInit;
export var Console: typeof r.Console = r.Console;
```

Platfrom-Server needs access to additional Core functionailty that is not part of the exported Core API and core-private.ts exports the additional types.

Source

platform-server/src

These source files are present:

- server.ts
- parse5_adapter.ts

There are no sub-directoies beneath src.

The server.ts file declares this const:

```
export const INTERNAL_SERVER_PLATFORM_PROVIDERS: Array<any> = [
  {provide: PLATFORM_INITIALIZER, useValue: initParse5Adapter, multi: true},
  {provide: PlatformLocation, useClass: ServerPlatformLocation},
];
```

The factory functions platformServer and platformDynamicServer create server platforms tha use the offline template compiler and the runtime template compiler respectively.

It adds two additional provider configurations. Firstly, PLATFORM_INITIALIZER, which is an initializer function called before bootstrapping. Here we see it initializes the parse5 adapter, in a call to the local function `initParse5Adapter()` which calls `makeCurrent()` for the `Parse5Adapter` and then calls `wtfInit()`.

```
function initParse5Adapter() {
  Parse5DomAdapter.makeCurrent();
  wtfInit();
}
```

The `wtfInit()` function comes from core_private and works with the WTF code in core/src/profile.

Secondly, it adds PlatformLocation, which is used by applications to interact with location (URL) information. It is set to a local class, ServerPlatformLocation, which just throws exceptions:

```
class ServerPlatformLocation extends PlatformLocation {
  getBaseHrefFromDOM(): string { throw notSupported('getBaseHrefFromDOM'); };
  onPopState(fn: any): void { notSupported('onPopState'); };
  onHashChange(fn: any): void { notSupported('onHashChange'); };
  get pathname(): string { throw notSupported('pathname'); }
  get search(): string { throw notSupported('search'); }
  get hash(): string { throw notSupported('hash'); }
  replaceState(state: any, title: string, url: string):
    void { notSupported('replaceState'); };
  pushState(state: any, title: string, url: string):
    void { notSupported('pushState'); };
  forward(): void { notSupported('forward'); };
  back(): void { notSupported('back'); };
}
```

server.ts declares two exported functions:

```
export const platformServer = createPlatformFactory(
                platformCore, 'server', INTERNAL_SERVER_PLATFORM_PROVIDERS);
```

```
export const platformDynamicServer = createPlatformFactory(
platformCoreDynamic, 'serverDynamic', INTERNAL_SERVER_PLATFORM_PROVIDERS);
```

We saw earlier that `platformCore` is defined in:

- <ANGULAR2>/modules/@angular/core/src/platform_core_providers.ts:

as:

```
const _CORE_PLATFORM_PROVIDERS: Provider[] = [
  PlatformRef_,
  {provide: PlatformRef, useExisting: PlatformRef_},
  {provide: Reflector, useFactory: _reflector, deps: []},
  {provide: ReflectorReader, useExisting: Reflector}, TestabilityRegistry,
  Console
];
export const platformCore = createPlatformFactory(
                              null, 'core', _CORE_PLATFORM_PROVIDERS);
```

`platformCoreDynamic` adds additional provider config (for the dynamic compiler) to `platformCore` and is defined in:

- <ANGULAR2>/modules/@angular/compiler/src/compiler.ts:

as:

```
export const platformCoreDynamic = createPlatformFactory(
                                platformCore, 'coreDynamic', [
  {provide: COMPILER_OPTIONS, useValue: {}, multi: true},
  {provide: CompilerFactory, useClass: RuntimeCompilerFactory},
  {provide: PLATFORM_INITIALIZER, useValue: _initReflector, multi: true},
]);
```

`createPlatformFactory()` is defined in:

- <ANGULAR2>/modules/@angular/core/src/application_ref.ts

it calls Core’s `createPlatform()` with the supplied parameters, resulting in a new platform being constructed.

The last part of Platform-Server’s server.ts file is the definition of the `NgModule`:

```
@NgModule({imports: [BrowserModule]})
export class ServerModule { }
```

We note it imports `BrowserModule`. That is where shared code related to platforms is supplied.

The second file in the @angular/platform-server/src directory, parse5_adapter.ts, create a DOM adapter for parse5. The parse5 library is a parsing and serialization engine for HTML5:

> *“WHATWG HTML5 specification-compliant, fast and ready for production HTML parsing/serialization toolset for Node.js parse5 provides nearly everything you may need when dealing with HTML. It's the fastest spec-compliant HTML parser for Node to date. It parses HTML the way the latest version of your browser does. It has proven itself reliable in such*

projects as jsdom, Angular2, Polymer and many more."
from: https://www.npmjs.com/package/parse5

The parse5_adapter.ts file provides an adapter class, `Parse5DomAdapter`, based on Parse 5's serialization functionality that implements a DomAdapter suitable for use in server environments.

The parse5_adapter.ts file has three variables:

```
var parser: any = null;
var serializer: any = null;
var treeAdapter: any = null;
```

Parse5DomAdapter simply extends DomAdapter:

```
/**
 * A `DomAdapter` powered by the `parse5` NodeJS module.
 *
 * @security Tread carefully! Interacting with the DOM directly is
 * dangerous and can introduce XSS risks.
 */
export class Parse5DomAdapter extends DomAdapter {
```

Its static `makeCurrent()` method, that we saw Universal Angular uses for server-side rendering, initializes those three variables and then calls `setRootDomAdapter()`:

```
  static makeCurrent() {
    parser = new parse5.Parser(parse5.TreeAdapters.htmlparser2);
    serializer = new parse5.Serializer(parse5.TreeAdapters.htmlparser2);
    treeAdapter = parser.treeAdapter;
    setRootDomAdapter(new Parse5DomAdapter());
  }
```

Recall that `setRootDomAdapter()` is defined in @angular/platform-browser/src/dom/dom_adapter.ts as:

```
var _DOM: DomAdapter = null;
export function setRootDomAdapter(adapter: DomAdapter) {
  if (isBlank(_DOM)) {
    _DOM = adapter;
  }
}
```

and that `getDOM()` is used by the `DOMRenderer`. Hence our Parse5DomAdapter gets wired into the DOM renderer.

The various `DomAdapter` methods are defined mostly in terms of the parse5's tree adapter. As an example, `appendChild()` and `insertBefore()` are defined as:

```
appendChild(el: any, node: any) {
    this.remove(node);
    treeAdapter.appendChild(this.templateAwareRoot(el), node);
}
insertBefore(el: any, node: any) {
    this.remove(node);
    treeAdapter.insertBefore(el.parent, node, el);
}
```

Parse5's serializer is used to implement the `get*Html()` methods as:

```
getInnerHTML(el: any): string {
  return serializer.serialize(this.templateAwareRoot(el));
}
getOuterHTML(el: any): string {
  serializer.html = '';
  serializer._serializeElement(el);
  return serializer.html;
}
```

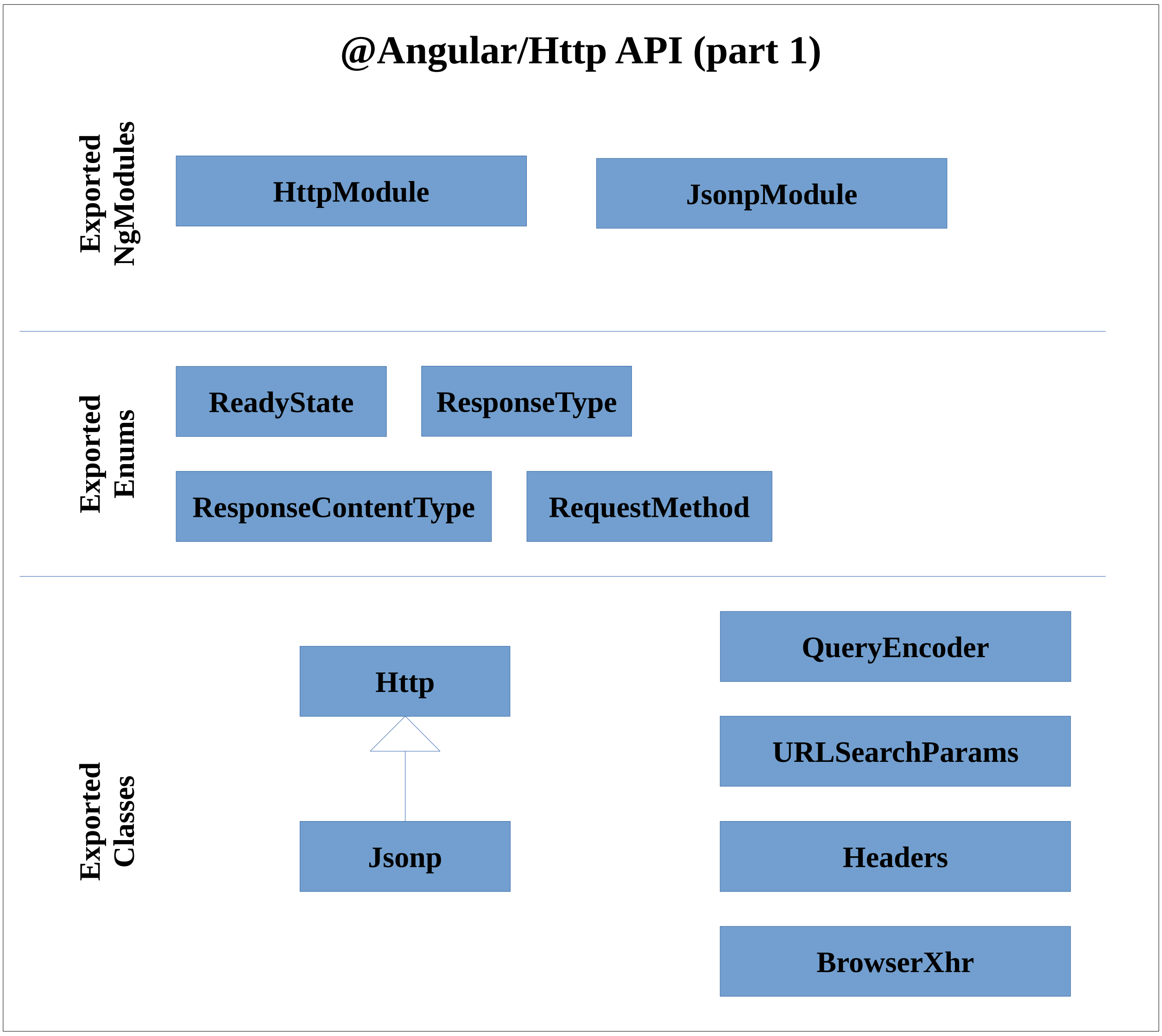
12: @Angular/HTTP

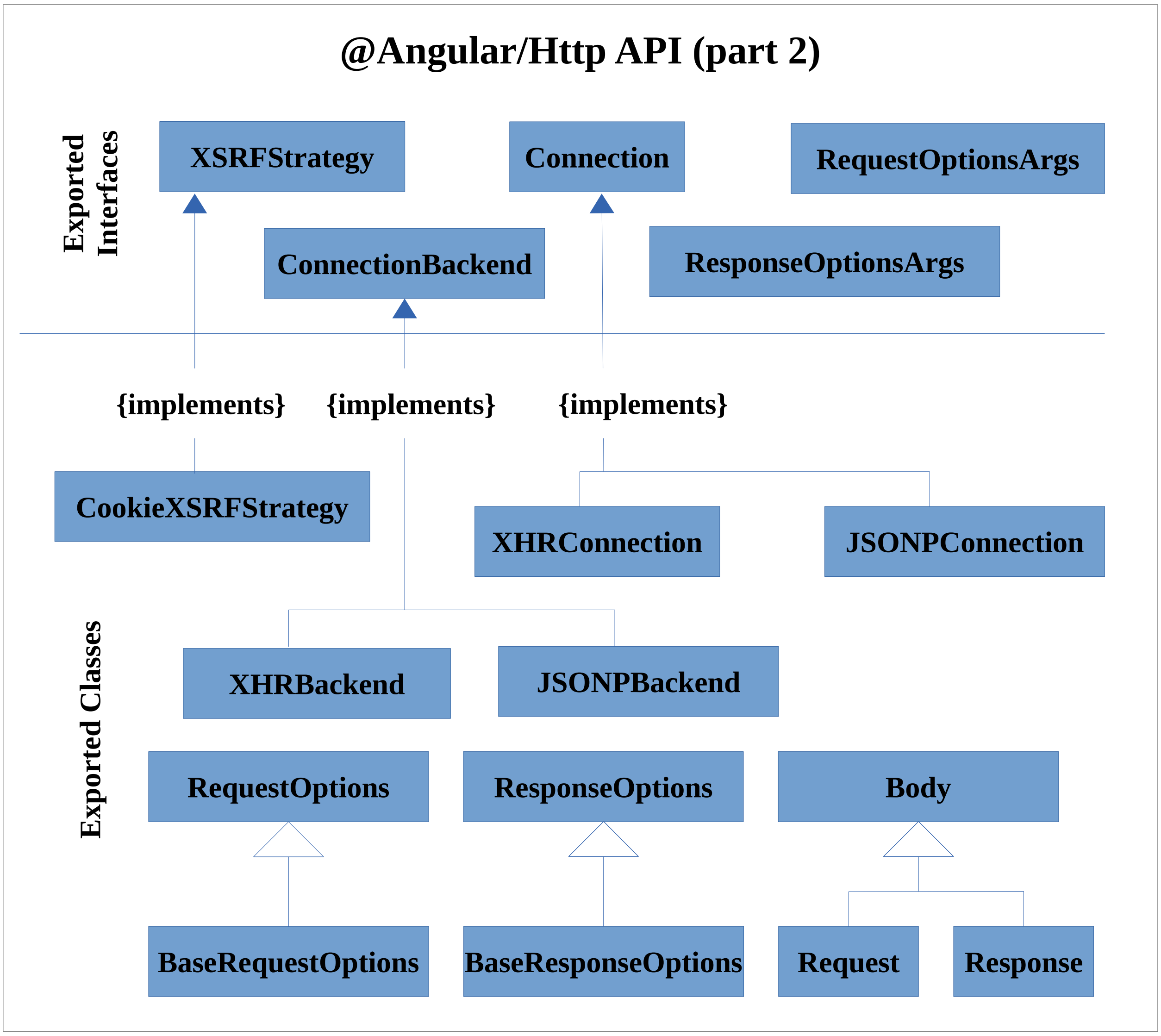
Overview

The Http module provides client-side access to an HTTP stack with configurable HTTP backends (e.g. alternatives for test). For production use, it usually makes calls to the browser's `XmlHttpRequest()` function. An interesting additional project, in-memory-web-api, can be used for a test-friendly implementation.

Http API

The exported API of the @Angular/Http package can be represented as:





The src/index.ts file defines the exports as:

```
export {BrowserXhr} from './backends/browser_xhr';
export {JSONPBackend, JSONPConnection} from './backends/jsonp_backend';
export {CookieXSRFStrategy, XHRBackend, XHRConnection}
  from './backends/xhr_backend';
export {BaseRequestOptions, RequestOptions} from './base_request_options';
export {BaseResponseOptions, ResponseOptions} from './base_response_options';
export {ReadyState, RequestMethod, ResponseContentType, ResponseType}
  from './enums';
export {Headers} from './headers';
export {Http, Jsonp} from './http';
export {HttpModule, JsonpModule} from './http_module';
export {Connection, ConnectionBackend, RequestOptionsArgs,
  ResponseOptionsArgs, XSRFStrategy} from './interfaces';
export {Request} from './static_request';
export {Response} from './static_response';
export {QueryEncoder, URLSearchParams} from './url_search_params';
export {VERSION} from './version';
```

Source Tree Layout

The source tree for the Http package contains these directories:

- src
- test (unit tests in Jasmine)
- testing (test tooling)

and these files:

- index.ts
- package.json
- rollup.config.js
- rollup-testing.config.js
- tsconfig-build.json
- tsconfig-testing.json

http/src

The http/src directory has one sub-directory:

- backends

and the following source files:

- base_request_options.ts
- base_response_options.ts
- body.ts
- enums.ts
- headers.ts
- http.ts
- http_module.ts
- http_utils.ts
- index.ts
- interfaces.ts
- static_request.ts
- static_response.ts
- url_search_params.ts
- version.ts

The http_module.ts file has two NgModule definitions:

```
@NgModule({
  providers: [
    {provide: Http, useFactory: httpFactory,
                                    deps: [XHRBackend, RequestOptions]},
    BrowserXhr,
    {provide: RequestOptions, useClass: BaseRequestOptions},
    {provide: ResponseOptions, useClass: BaseResponseOptions},
    XHRBackend,
    {provide: XSRFStrategy, useFactory: _createDefaultCookieXSRFStrategy},
  ],
})
export class HttpModule {
```

```
}


@NgModule({
  providers: [
   {provide: Jsonp, useFactory: jsonpFactory,
             deps: [JSONPBackend, RequestOptions]},
    BrowserJsonp,
    {provide: RequestOptions, useClass: BaseRequestOptions},
    {provide: ResponseOptions, useClass: BaseResponseOptions},
    {provide: JSONPBackend, useClass: JSONPBackend_},
  ],
})
export class JsonpModule {
}
```

That `XHRBackend` provider for HttpModule may need to be replaced when using an in-memory-web-api for testing, as explained here (search for `InMemoryWebApiModule`):

- https://angular.io/docs/ts/latest/tutorial/toh-pt6.html

http_module.ts also has three simple factory functions:

```
export function _createDefaultCookieXSRFStrategy() {
  return new CookieXSRFStrategy();
}

export function httpFactory(xhrBackend: XHRBackend, requestOptions:
RequestOptions): Http {
  return new Http(xhrBackend, requestOptions);
}

export function jsonpFactory(jsonpBackend: JSONPBackend, requestOptions:
RequestOptions): Jsonp {
  return new Jsonp(jsonpBackend, requestOptions);
}
```

The headers.ts file provides the implementation for the `Headers` class. This is essentially a wrapper around a `Map` data structure. It imports `Map` from ../src/facade/collection. It defines its primary data structure as:

```
_headersMap: Map<string, string[]>;
```

Its offers functionality to view and edit that data structure.

The body.ts file implements the `Body` class, which is used as the basis for requests and responses. `Body` is used internally within the Http module but it is not exported from it. `Body` provides four methods: `json()`, `text()`, `arrayBuffer()` and `blob()`, to return the body contents as each type.

The static_request.ts file implements the `Request` class, which extends `Body`. A request is a request method, a set of headers, and a content type. It can supply body content based on content type. It provides this method to determine content type:

```
detectContentType(): ContentType {
    switch (this.headers.get('content-type')) {
```

```
        case 'application/json':
          return ContentType.JSON;
        case 'application/x-www-form-urlencoded':
          return ContentType.FORM;
        case 'multipart/form-data':
          return ContentType.FORM_DATA;
        case 'text/plain':
        case 'text/html':
          return ContentType.TEXT;
        case 'application/octet-stream':
          return ContentType.BLOB;
        default:
          return this.detectContentTypeFromBody();
      }
    }
```

Similarly, the static_response.ts implements the `Response` class. It manages the following properties for a response:

```
type: ResponseType;
ok: boolean;
url: string;
status: number;
statusText: string;
bytesLoaded: number;
totalBytes: number;
headers: Headers;
```

The enums.ts file lists enums used:

```
export enum RequestMethod {
  Get,
  Post,
  Put,
  Delete,
  Options,
  Head,
  Patch
}

export enum ContentType {
  NONE,
  JSON,
  FORM,
  FORM_DATA,
  TEXT,
  BLOB,
  ARRAY_BUFFER
}
```

```
export enum ReadyState {
  Unsent,
  Open,
  HeadersReceived,
  Loading,
  Done,
  Cancelled
}

export enum
ResponseContentType {
  Text,
  Json,
  ArrayBuffer,
  Blob
}
```

```
export enum ResponseType {
  Basic,
  Cors,
  Default,
  Error,
  Opaque
}
```

Both `Request` and `Response` are low-level classes, usually not called form application code directly. Instead, the `Http` class is usually used, which we'll cover shortly.

The url_search_params.ts file implements the `UrlSearchParams` and `QueryEncoder` classes. There are used to expose a map interface to url search parameters.

The interfaces.ts file provides a number of classes and interfaces for pluggable connection handling. By providing alternative implementations of these, flexible server communication is supported.

```
export abstract class ConnectionBackend {
  abstract createConnection(request: any): Connection; }
export abstract class Connection {
  readyState: ReadyState;
  request: Request;
  response: any;  // TODO: generic of <Response>;
}
export abstract class XSRFStrategy {
   abstract configureRequest(req: Request): void; }

export interface RequestOptionsArgs {
  url?: string;
  method?: string|RequestMethod;
  /** @deprecated from 4.0.0. Use params instead. */
  search?: string|URLSearchParams|{[key: string]: any | any[]};
  params?: string|URLSearchParams|{[key: string]: any | any[]};
  headers?: Headers;
  body?: any;
  withCredentials?: boolean;
  responseType?: ResponseContentType;
}

export interface RequestArgs extends RequestOptionsArgs { url: string; }

export type ResponseOptionsArgs = {
  body?: string | Object | FormData
            | ArrayBuffer | Blob; status?: number; statusText?: string;
  headers?: Headers;
  type?: ResponseType;
  url?: string;
};
```

The http.ts file provides two injectable classes – `Http` and `Jsonp` – and two functions – `httpRequest` and `mergeOptions`. The `Jsonp` class extends the `Http` class.

`http`'s constructor takes in parameters of a backend and request options.

```
@Injectable()
export class Http {
  constructor(
     protected _backend: ConnectionBackend,
     protected _defaultOptions: RequestOptions) {}
```

The backend is important because it means a test-oriented in-memory backend could be switched for the real backend as needed, without further changes to HTTP code. The `Http` class has methods for each HTTP method whose implementations all funnel the request execution to the `httpRequest` function, which in turn delegates responsibility to the configured backend for each operation. The result in an `Observable` of type `Response`.

```
Function request(url: string|Request, options?: RequestOptionsArgs):
   Observable<Response> { .. }
```

Http/backends

This sub-directory has the following source files:

- browser_jsonp.ts
- browser_xhr.ts
- jsonp_backend.ts
- xhr_backend.ts

The browser_xhr.ts file implements the `BrowserXhr` class, which is a wrapper for `XMLHttpRequest` calls:

```
  /**
  * A backend for http that uses the `XMLHttpRequest` browser API.
  * Take care not to evaluate this in non-browser contexts.
  */
@Injectable()
export class BrowserXhr {
  constructor() {}
  build(): any { return <any>(new XMLHttpRequest()); }
}
```

The xhr_backend.ts file implements three classes – `XHRConnection`, `CookieXSRFStrategy` and `XHRBackend`. All of which are implementations of interfaces we saw earlier in @angular/http/src/interfaces.ts.

```
export class XHRConnection implements Connection {..}
export class CookieXSRFStrategy implements XSRFStrategy {..}
@Injectable()
export class XHRBackend implements ConnectionBackend {..}
```

`XHRConnection` has a large constructor which takes in a request, a `browserXHR` instance and options:

```
constructor(req: Request, browserXHR: BrowserXhr,
                              baseResponseOptions?: ResponseOptions) {}
```

`BrowserXHR.build()` is called to access an `XMLHttpRequest` implementation, and this is used to process the request and response.

`XHRBackend` is defined as:

```
@Injectable()
export class XHRBackend implements ConnectionBackend {
  constructor(
      private _browserXHR: BrowserXhr, private _baseResponseOptions:
ResponseOptions,
      private _xsrfStrategy: XSRFStrategy) {}

  createConnection(request: Request): XHRConnection {
    this._xsrfStrategy.configureRequest(request);
    return new XHRConnection(request, this._browserXHR,
this._baseResponseOptions);
  }
}
```

Its job is to return an instance of `XHRConnection` to be used for connecting.

The `CookieXSRFStrategy` class is for Cross Site Request Forgery (XSRF) protection. In its constructor it takes in a cookieName and a headerName. In its `configureRequest()` method, which takes in a request parameter, it checks if the xsrfToken is defined as a cookie in the DOM (with name set to `cookieName`), and if yes, and a header with `headerName` is not already defined for this request, it adds such a header, setting its value to the cookie. It is defined as follows:

```
export class CookieXSRFStrategy implements XSRFStrategy {
  constructor(
      private _cookieName:
      string = 'XSRF-TOKEN', private _headerName: string = 'X-XSRF-TOKEN') {}

  configureRequest(req: Request): void {
    const xsrfToken =
       __platform_browser_private__.getDOM().getCookie(this._cookieName);
    if (xsrfToken) {
      req.headers.set(this._headerName, xsrfToken);
    }
  }
}
```

13: @Angular/Forms

Overview

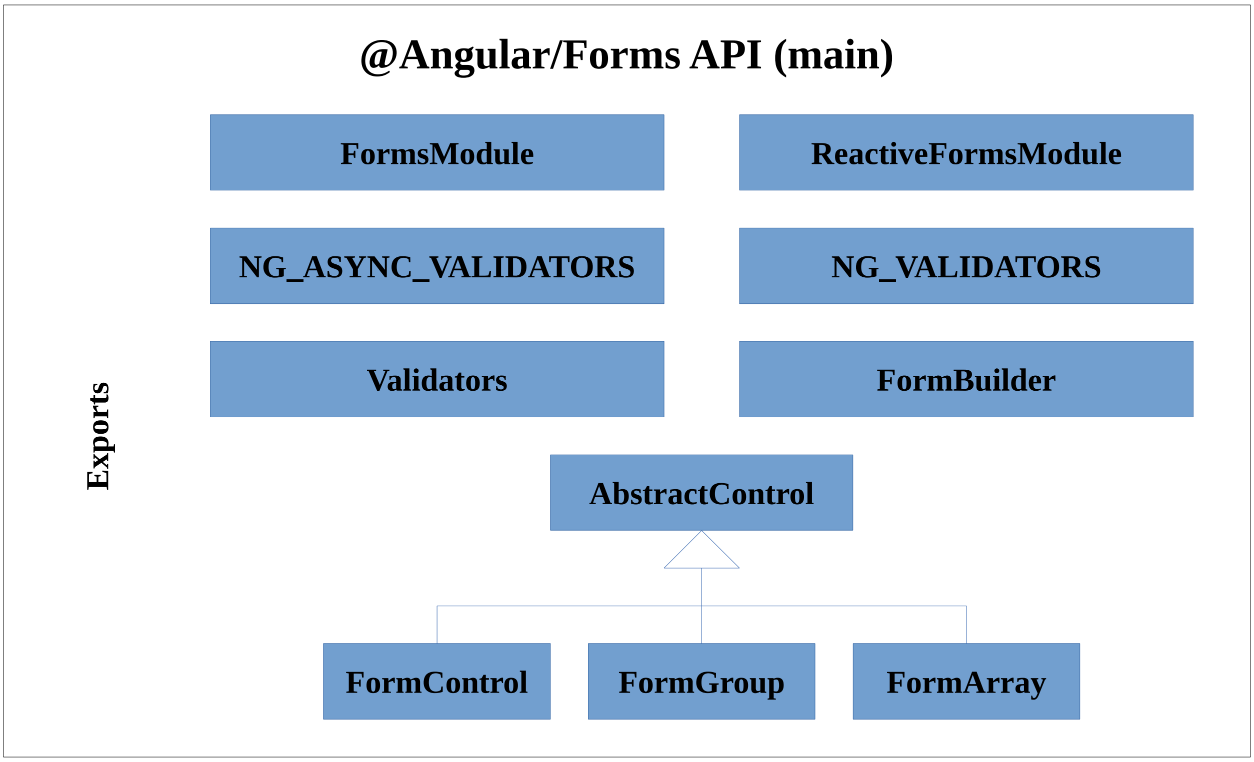
The Forms module provides capabilities to manage web forms.

It supplies functionality in areas such as validation, submit, data model, additional directives and a builder for dynamic forms.

Forms API

The exported API of the @Angular/Forms package can be sub-divided into four groups – main, directives, accessors and validator directives.

The main group can be represented as:



Its index.ts file contains this one line:

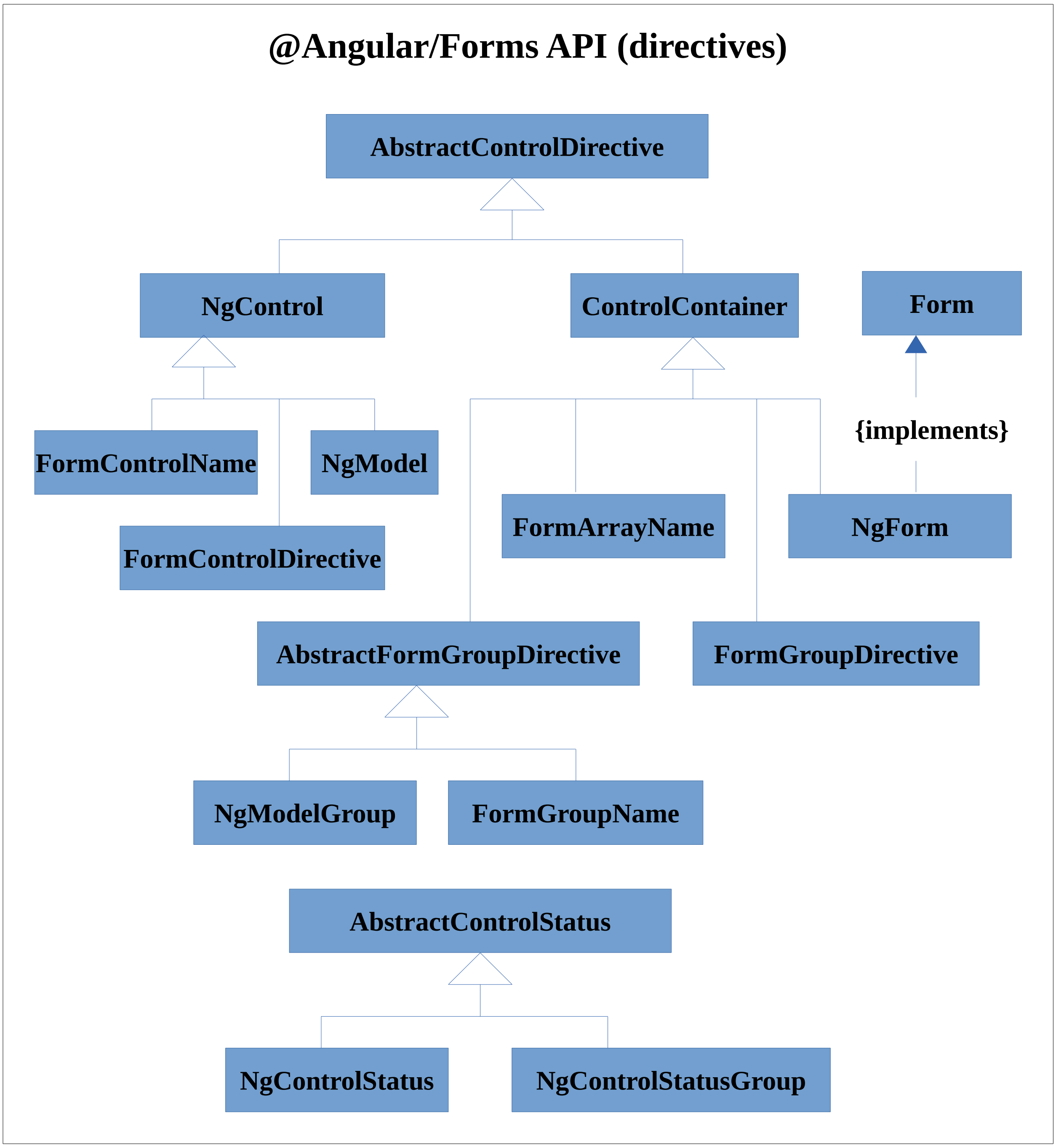
```
export * from './src/forms';
```

The parts of forms.ts related to exporting the above are:

```
export {FormBuilder} from './form_builder';
export {AbstractControl, FormArray, FormControl, FormGroup} from './model';
export {NG_ASYNC_VALIDATORS, NG_VALIDATORS, Validators} from './validators';
export * from './form_providers';
```

Two NgModules are supplied, one for normal forms, FormsModule, and the other for reactive forms, ReactiveFormsModule. The control hierarchy starts with a root, and has FormControl for actual controls,and two combinations of controls, one for group (of fixed size) and one for array (of dynamic size).

A large directive hierarchy is supplied for both types of forms:

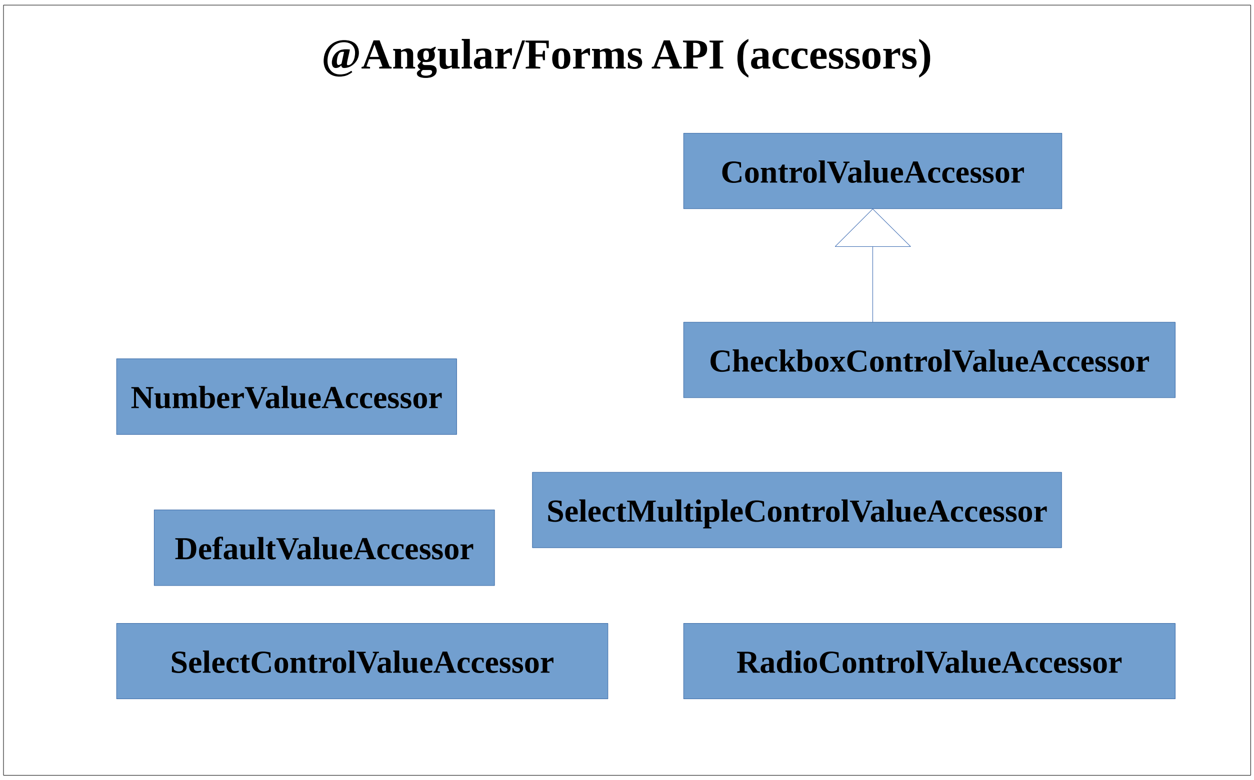


The parts of forms.ts related to exporting directives are:

```
export {AbstractControlDirective}
  from './directives/abstract_control_directive';
export {AbstractFormGroupDirective}
  from './directives/abstract_form_group_directive';
export {ControlContainer} from './directives/control_container';
export {Form} from './directives/form_interface';
export {NgControl} from './directives/ng_control';
export {NgControlStatus, NgControlStatusGroup}
```

```
  from './directives/ng_control_status';
export {NgForm} from './directives/ng_form';
export {NgModel} from './directives/ng_model';
export {NgModelGroup} from './directives/ng_model_group';
export {FormControlDirective}
  from './directives/reactive_directives/form_control_directive';
export {FormControlName}
  from './directives/reactive_directives/form_control_name';
export {FormGroupDirective}
  from './directives/reactive_directives/form_group_directive';
export {FormArrayName}
  from './directives/reactive_directives/form_group_name';
export {FormGroupName}
  from './directives/reactive_directives/form_group_name';
```

The accessor hierarchy can be represented as:



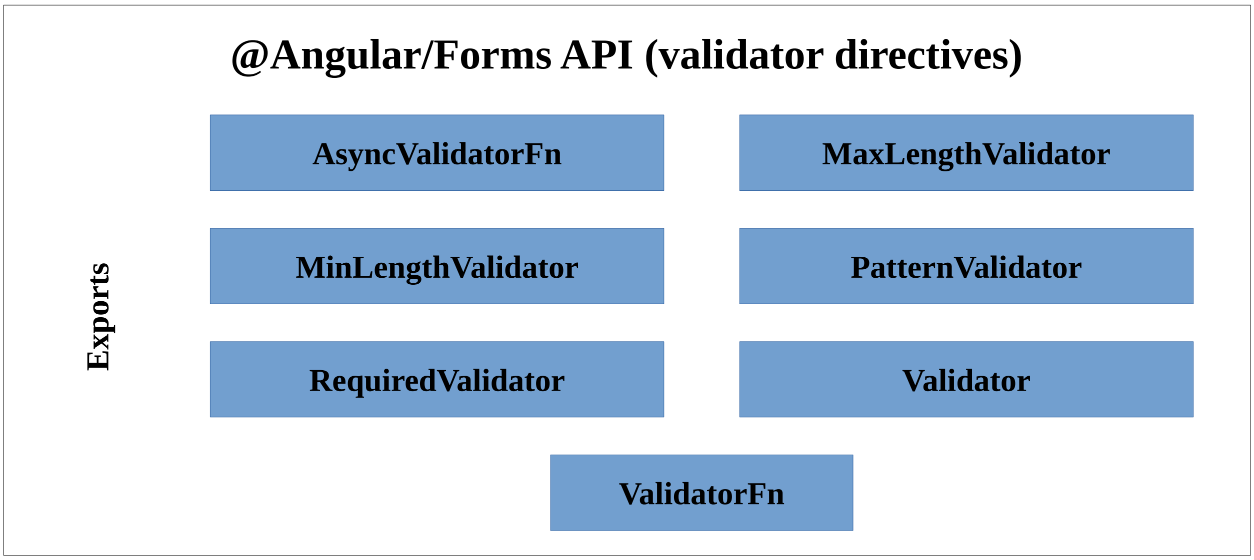
The parts of forms.ts related to exporting accessors are:

```
export {CheckboxControlValueAccessor} from
'./directives/checkbox_value_accessor';
export {ControlValueAccessor, NG_VALUE_ACCESSOR} from
'./directives/control_value_accessor';
export {DefaultValueAccessor} from './directives/default_value_accessor';
export {NgSelectOption, SelectControlValueAccessor} from
'./directives/select_control_value_accessor';
export {SelectMultipleControlValueAccessor} from
'./directives/select_multiple_control_value_accessor';
```

The part of forms.ts related to exporting validator directives is:

```
export {AsyncValidatorFn, MaxLengthValidator, MinLengthValidator,
```

```
    PatternValidator, RequiredValidator, Validator, ValidatorFn}
```




```
    from './directives/validators';
```

Source Tree Layout

The source tree for the Forms package contains these directories:

- src
- test (unit tests in Jasmine)

Note that unlike most other @Angular modules, Forms has no testing directory.

Forms main directory contains these files:

- index.ts
- package.json
- rollup.config.js
- tsconfig.json

Source

forms/src

The @angular/forms/src directory contains these files:

- directives.ts
- form_builder.ts
- form_providers.ts
- forms.ts
- model.ts
- validators.ts

forms_providers.ts defines the `FormsModule` and `ReactiveFormsModule` NgModules as follows:

```
  @NgModule({
    declarations: TEMPLATE_DRIVEN_DIRECTIVES,
```

```
    providers: [RadioControlRegistry],
    exports: [InternalFormsSharedModule, TEMPLATE_DRIVEN_DIRECTIVES]
  })
  export class FormsModule { }
  @NgModule({
    declarations: [REACTIVE_DRIVEN_DIRECTIVES],
    providers: [FormBuilder, RadioControlRegistry],
    exports: [InternalFormsSharedModule, REACTIVE_DRIVEN_DIRECTIVES]
  })
  export class ReactiveFormsModule { }
```

We note the two difference between `FormsModule` and `ReactiveFormsModule` are that `ReactiveFormsModule` has an additional `FormBuilder` provider configuration, and the export from `FormModule` includes `TEMPLATE_DRIVEN_DIRECTIVES` whereas the export from `ReactiveFormsModule` includes `REACTIVE_DRIVEN_DIRECTIVES`.

directives.ts defines these:

```
export const SHARED_FORM_DIRECTIVES: Type<any>[] = [
  NgSelectOption, NgSelectMultipleOption, DefaultValueAccessor,
  NumberValueAccessor, CheckboxControlValueAccessor,
  SelectControlValueAccessor, SelectMultipleControlValueAccessor,
  RadioControlValueAccessor, NgControlStatus, NgControlStatusGroup,
  RequiredValidator, MinLengthValidator, MaxLengthValidator, PatternValidator
];
export const TEMPLATE_DRIVEN_DIRECTIVES: Type<any>[] = [NgModel,
NgModelGroup, NgForm];
export const REACTIVE_DRIVEN_DIRECTIVES: Type<any>[] =
    [FormControlDirective, FormGroupDirective, FormControlName,
FormGroupName, FormArrayName];
export const FORM_DIRECTIVES: Type<any>[][] = [TEMPLATE_DRIVEN_DIRECTIVES,
SHARED_FORM_DIRECTIVES];
export const REACTIVE_FORM_DIRECTIVES: Type<any>[][] =
    [REACTIVE_DRIVEN_DIRECTIVES, SHARED_FORM_DIRECTIVES];

@NgModule(
  {declarations: SHARED_FORM_DIRECTIVES, exports: SHARED_FORM_DIRECTIVES})
export class InternalFormsSharedModule { }
```

A nice discussion of how to create dynamic forms using `REACTIVE_FORM_DIRECTIVES` is here:

- https://angular.io/docs/ts/latest/cookbook/dynamic-form.html

The form_builder.ts file defines the injectable `FormsBuilder` class, which can dynamically construct a `FormGroup`, `FormArray` or `FormControl` via its `group()`, `array()` or `control()` methods. They are defined as:

```
group(controlsConfig:
{[key: string]: any}, extra: {[key: string]: any} = null): FormGroup {
  const controls = this._reduceControls(controlsConfig);
  const validator: ValidatorFn =
   isPresent(extra) ? StringMapWrapper.get(extra, 'validator') : null;
  const asyncValidator: AsyncValidatorFn =
   isPresent(extra) ? StringMapWrapper.get(extra, 'asyncValidator') : null;
  return new FormGroup(controls, validator, asyncValidator);
}
```

```
  control(
      formState: Object, validator: ValidatorFn|ValidatorFn[] = null,
      asyncValidator: AsyncValidatorFn|AsyncValidatorFn[]=null):FormControl {
    return new FormControl(formState, validator, asyncValidator);
  }

  array(
      controlsConfig: any[], validator: ValidatorFn = null,
      asyncValidator: AsyncValidatorFn = null): FormArray {
    var controls = controlsConfig.map(c => this._createControl(c));
    return new FormArray(controls, validator, asyncValidator);
  }
```

The validators.ts file first declares two opaque tokens for dependency injection:

```
export const NG_VALIDATORS: OpaqueToken = new OpaqueToken('NgValidators');
export const NG_ASYNC_VALIDATORS: OpaqueToken =
                                     new OpaqueToken('NgAsyncValidators');
```

It also defines the Validators class:

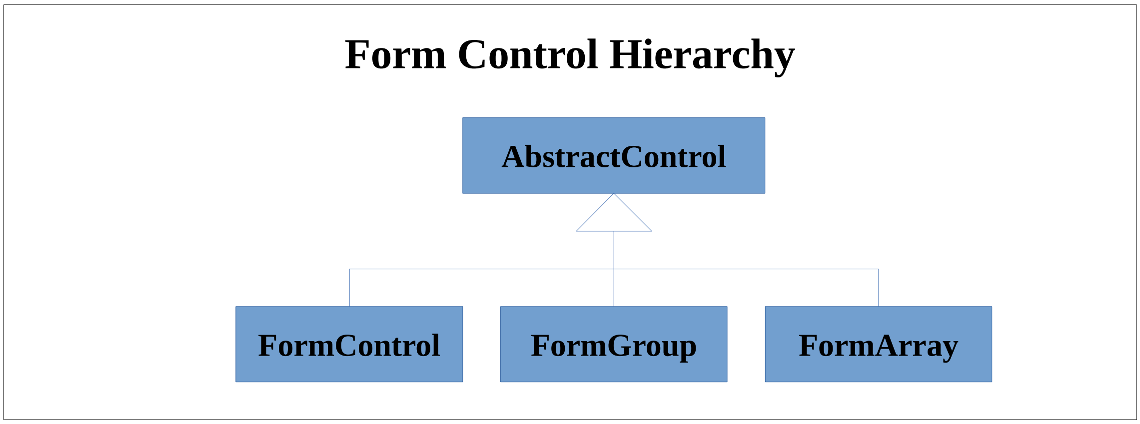
```
// A validator is a function that processes a FormControl or
// collection of controls and returns a map of errors.
//
// A null map means that validation has passed.

export class Validators {
  static required(control: AbstractControl): {[key: string]: boolean} { }
  static minLength(minLength: number): ValidatorFn { }
  static maxLength(maxLength: number): ValidatorFn { }
  static pattern(pattern: string): ValidatorFn { }
}
```

A sample implementation of one of the validators is:

```
static required(control: AbstractControl): {[key: string]: boolean} {  1
 return
 2 isBlank(control.value) || (isString(control.value) && control.value == '')
 3 ? {'required': true}
 4 : null;
}
```

The return value 1 is a string to boolean map. If the first line 2 is true, then {'required': true} is returned 3, otherwise null 4 is returned.

The model.ts file is large and defines the form control hierarchy:



The status of a form control is one of:

```
// Indicates that a FormControl is valid,
// i.e. that no errors exist in the input value
export const VALID = 'VALID';


// Indicates that a FormControl is invalid,
// i.e. that an error exists in the input value.
export const INVALID = 'INVALID';

// Indicates that a FormControl is pending, i.e. that async validation is
// occurring and errors are not yet available for the input value.
export const PENDING = 'PENDING';

// Indicates that a FormControl is disabled, i.e. that the control is
// exempt from ancestor calculations of validity or value.
export const DISABLED = 'DISABLED';
```

The `AbstractControl` class defines a constructor, that takes in validator and async validator functions. This class also defines a bunch of getters which map to private fields. The value field refers to data we wish to strore within the control:

```
get value(): any { return this._value; }
```

The `_status` field refers to the validator checking:

```
get status(): string { return this._status; }
get valid(): boolean { return this._status === VALID; }
get invalid(): boolean { return this._status === INVALID; }
get pending(): boolean { return this._status == PENDING; }
```

The `_error` field returns a map of errors (if any):

```
get errors(): {[key: string]: any} { return this._errors; }
```

The `_pristine` field refers to whether the control's data has been changed – `pristine()` is true if unchanged, and `dirty()` is true if changed:

```
get pristine(): boolean { return this._pristine; }
get dirty(): boolean { return !this.pristine; }
```

The `_touched` field refers to whether the user has visited the control (if does not mean that the control's value has been changed):

```
get touched(): boolean { return this._touched; }
get untouched(): boolean { return !this._touched; }
```

There are also two `xxChanges()` getters, for value changes and status changes, that return observables:

```
get valueChanges(): Observable<any> { return this._valueChanges; }
get statusChanges(): Observable<any> { return this._statusChanges; }
```

These are initialized to event emitters via:

```
_initObservables() {
    this._valueChanges = new EventEmitter();
    this._statusChanges = new EventEmitter();
  }
```

`AbstractControl` also declares a function:

```
abstract _anyControls(condition: Function): boolean;
```

which executes the condition function over the control and its children and return a boolean. This `_anyControls` function is used in many helper methods to determine information about the control, e.g.:

```
_anyControlsHaveStatus(status: string): boolean {
    return this._anyControls(
       (control: AbstractControl) => control.status == status);
}
```

It has a parent field:

```
private _parent: FormGroup|FormArray;
```

which is used when the state of the control is being updated.

```
markAsDirty({onlySelf}: {onlySelf?: boolean} = {}): void {
    onlySelf = normalizeBool(onlySelf);
    this._pristine = false;
    if (isPresent(this._parent) && !onlySelf) {
      this._parent.markAsDirty({onlySelf: onlySelf});
    }
 }
```

It is set via:

```
setParent(parent: FormGroup|FormArray): void { this._parent = parent; }
```

Its is hierarchical and this is supplied to find the root:

```
get root(): AbstractControl {
  let x: AbstractControl = this;
  while (isPresent(x._parent)) { x = x._parent; }
  return x;
}
```

The FormControl class is supplied for atomic controls (that do not contain any child controls).

```
// By default, a `FormControl` is created for every `<input>` or
// other form component.
export class FormControl extends AbstractControl {
```

```
    _onChange: Function[] = [];

    // Register a listener for change events.
    registerOnChange(fn: Function): void { this._onChange.push(fn); }
    ..
  }
```

Its `_value` field is set via `setValue()` method which reacts depending on the four optional booleans supplied:

```
    setValue(value: any, {onlySelf, emitEvent, emitModelToViewChange,
  emitViewToModelChange}: {
      onlySelf?: boolean,
      emitEvent?: boolean,
      emitModelToViewChange?: boolean,
      emitViewToModelChange?: boolean
    } = {}): void {
      emitModelToViewChange = isPresent(emitModelToViewChange) ?
  emitModelToViewChange : true;
      emitViewToModelChange = isPresent(emitViewToModelChange) ?
  emitViewToModelChange : true;

      this._value = value;
      if (this._onChange.length && emitModelToViewChange) {
        this._onChange.forEach((changeFn) => changeFn(this._value,
  emitViewToModelChange));
      }
      this.updateValueAndValidity({onlySelf: onlySelf, emitEvent: emitEvent});
    }
```

It has a `reset()` method to reset control data:

```
  reset(formState: any = null, {onlySelf}: {onlySelf?: boolean} = {}): void {
      this._applyFormState(formState);
      this.markAsPristine({onlySelf});
      this.markAsUntouched({onlySelf});
      this.setValue(this._value, {onlySelf});
  }
```

The FormGroup class extends AbstractControl:

```
  export class FormGroup extends AbstractControl {
```

Its constructor's first parameter defines a controls associative map (in constrast to FormArray):

```
  constructor(
        public controls: {[key: string]: AbstractControl},
        validator: ValidatorFn = null,
        asyncValidator: AsyncValidatorFn = null) {
      super(validator, asyncValidator);
      this._initObservables();
      this._setParentForControls();
      this.updateValueAndValidity({onlySelf: true, emitEvent: false});
    }
```

Controls can be registered with w FormGroup via:

```
  // Register a control with the group's list of controls.
  registerControl(name: string, control: AbstractControl): AbstractControl {
    if (this.controls[name]) return this.controls[name];
    this.controls[name] = control;
    control.setParent(this);
    return control;
  }
```

The values of all controls in the group may be set via:

```
setValue(
  value: {[key: string]: any}, {onlySelf}: {onlySelf?: boolean} = {}): void {
    this._checkAllValuesPresent(value);
    StringMapWrapper.forEach(value, (newValue: any, name: string) => {
      this._throwIfControlMissing(name); 1
      this.controls[name].setValue(newValue, {onlySelf: true});
    });
    this.updateValueAndValidity({onlySelf: onlySelf});
  }
```

Note it throws an exception is any of the controls are missing **1**.

The FormArray class extends AbstractControl:

```
export class FormArray extends AbstractControl {
```

Its constructor’s first parameter is simply an array:

```
constructor(
      public controls: AbstractControl[],
      validator: ValidatorFn = null,
      asyncValidator: AsyncValidatorFn = null) {
    super(validator, asyncValidator);
    this._initObservables();
    this._setParentForControls();
    this.updateValueAndValidity({onlySelf: true, emitEvent: false});
  }
```

It allows you to insert at the end of the array or at a given location, and to remove:

```
  // Insert a new {@link AbstractControl} at the end of the array.
  push(control: AbstractControl): void {
    this.controls.push(control);
    control.setParent(this);
    this.updateValueAndValidity();
  }

  // Insert a new {@link AbstractControl} at the given `index` in the array.
  insert(index: number, control: AbstractControl): void {
    ListWrapper.insert(this.controls, index, control);
    control.setParent(this);
    this.updateValueAndValidity();
  }

  // Remove the control at the given `index` in the array.
  removeAt(index: number): void {
    ListWrapper.removeAt(this.controls, index);
    this.updateValueAndValidity();
  }
```

14: @Angular/Router

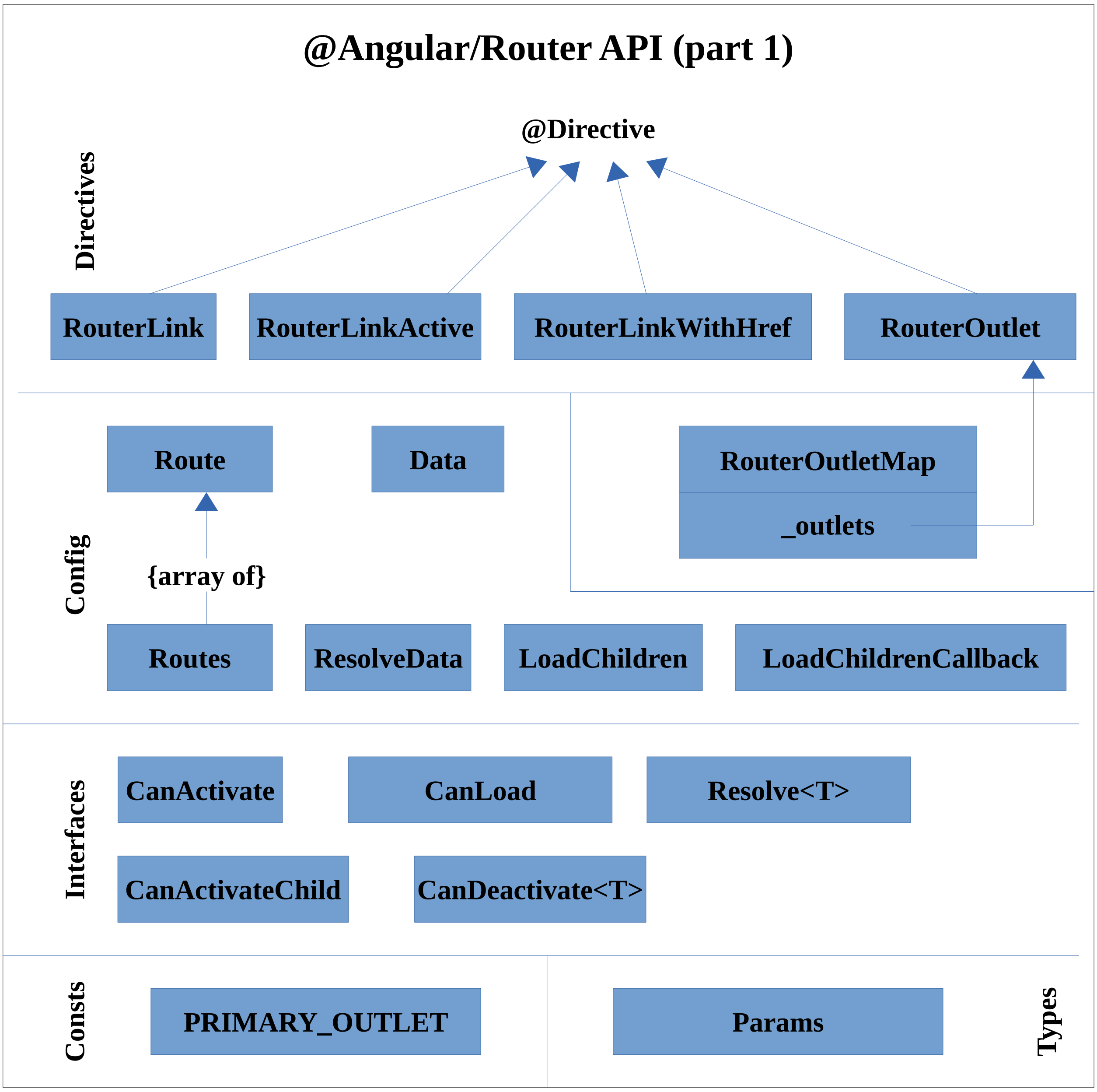
Overview

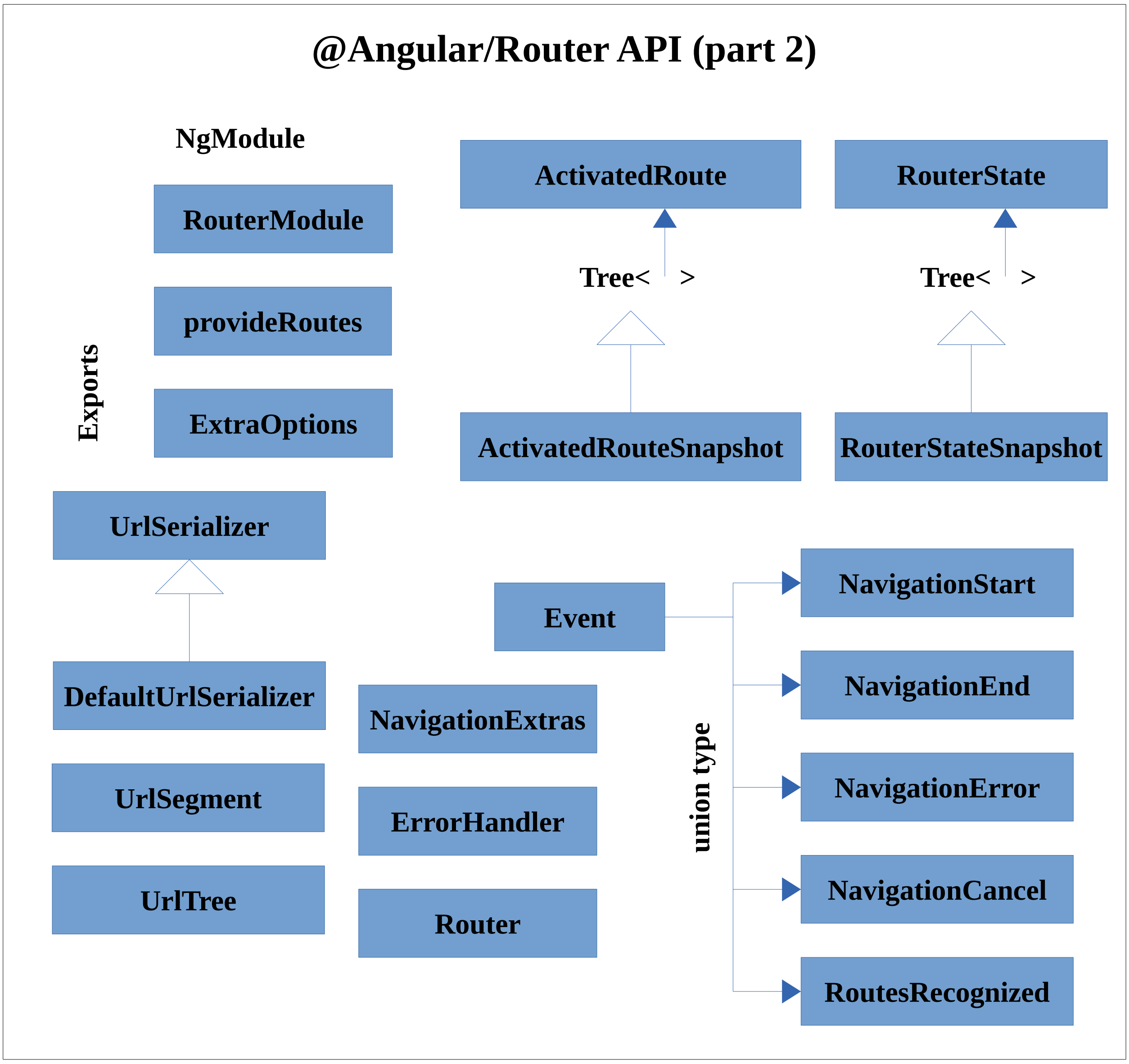
The Angular Router provides functionality to:

- manage application state
- manage state transitions
- reflect state in the URL that the user sees in the browser
- dynamically load compoments as needed

Router API

The exported API of the @Angular/Router package can be represented as:





The <ANGULAR2>/modules/@angular/router directory contains:

- index.ts

This lists the exports and gives an initial impression of the size of the router package:

```
export {Data, LoadChildren, LoadChildrenCallback, ResolveData, Route, Routes}
  from './src/config';
export {RouterLink, RouterLinkWithHref} from './src/directives/router_link';
export {RouterLinkActive} from './src/directives/router_link_active';
export {RouterOutlet} from './src/directives/router_outlet';
export {CanActivate, CanActivateChild, CanDeactivate, CanLoad, Resolve}
  from './src/interfaces';
export {ErrorHandler, Event, NavigationCancel, NavigationEnd,
NavigationError, NavigationExtras, NavigationStart, Router, RoutesRecognized}
  from './src/router';
export {ExtraOptions, RouterModule, provideRoutes}
```

```
  from './src/router_module';
export {RouterOutletMap} from './src/router_outlet_map';
export {ActivatedRoute, ActivatedRouteSnapshot, RouterState,
  RouterStateSnapshot} from './src/router_state';
export {PRIMARY_OUTLET, Params} from './src/shared';
export {DefaultUrlSerializer, UrlSegment, UrlSerializer, UrlTree}
  from './src/url_tree';
```

Source Tree Layout

The source tree for the Router package contains these directories:

- src
- test (unit tests in Jasmine)
- testing (testing tools)
- scripts

The Router main directory contains these files:

- index.ts
- ng_probe_token.ts
- CHANGELOG.md
- karma-test-shim.ts
- karma.config.js
- README.md
- package.json
- rollup.config.js
- tsconfig.json
- tsconfig.json

The ng_probe_token.ts file is:

```
import {NgProbeToken} from '@angular/platform-browser';
import {Router} from './src/router';

export const ROUTER_NG_PROBE_PROVIDER = {
  provide: NgProbeToken,
  multi: true,
  useValue: new NgProbeToken('router', Router)
};
```

It contains an extra provider for the NgProbeToken for the router – helpful for debugging.

Source

router/src

The router/src directory contains:

- apply_redirects.ts
- common_router_providers.ts
- config.ts
- create_router_state.ts
- create_url_tree.ts
- interfaces.ts

- recognize.ts
- resolve.ts
- router.ts
- router_config-loader.ts
- router_module.ts
- router_outlet_map.ts
- router_providers.ts
- router_state.ts
- shared.ts
- url_tree.ts

We'll start by looking at RouterModule (router/src/router_module), which is defined as:

```
// When registered at the root, it should be used as follows:
// bootstrap(AppCmp, {imports: [RouterModule.forRoot(ROUTES)]});
@NgModule({declarations: ROUTER_DIRECTIVES, exports: ROUTER_DIRECTIVES})
export class RouterModule {
  static forRoot(routes: Routes, config?: ExtraOptions):ModuleWithProviders {
    return {
      ngModule: RouterModule,
      providers: [
        ROUTER_PROVIDERS,
        provideRoutes(routes),
        {provide: ROUTER_CONFIGURATION, useValue: config ? config : {}}, {
          provide: LocationStrategy,
          useFactory: provideLocationStrategy,
          deps: [
            PlatformLocation,
            [new Inject(APP_BASE_HREF), new Optional()],
            ROUTER_CONFIGURATION
          ]
        },
        provideRouterInitializer()
      ]
    };
  }
}
```

It uses `ROUTER_PROVIDERS`, which is defined as:

```
export const ROUTER_PROVIDERS: any[] = [
  Location,
  {provide: UrlSerializer, useClass: DefaultUrlSerializer},
  {
    provide: Router,
    useFactory: setupRouter,
    deps: [
      ApplicationRef, ComponentResolver,
      UrlSerializer, RouterOutletMap, Location, Injector,
      NgModuleFactoryLoader, ROUTES, ROUTER_CONFIGURATION ]
  },
  RouterOutletMap,
  {provide: ActivatedRoute, useFactory: rootRoute, deps: [Router]},
  {provide: NgModuleFactoryLoader, useClass: SystemJsNgModuleLoader},
  {provide: ROUTER_CONFIGURATION, useValue: {enableTracing: false}}
```

```
];
```

it also uses `ROUTER_DIRECTIVES`:

```
export const ROUTER_DIRECTIVES =
    [RouterOutlet, RouterLink, RouterLinkWithHref, RouterLinkActive];
```

Interfaces.ts declares a number of useful interfaces.

```
export interface CanActivate {
  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):
      Observable<boolean>|Promise<boolean>|boolean;
}
export interface CanActivateChild {
  canActivateChild(
     childRoute: ActivatedRouteSnapshot,
     state: RouterStateSnapshot):
      Observable<boolean>|Promise<boolean>|boolean;
}
export interface CanDeactivate<T> {
  canDeactivate(component: T, route: ActivatedRouteSnapshot,
   state: RouterStateSnapshot):
      Observable<boolean>|Promise<boolean>|boolean;
}
export interface Resolve<T> {
  resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):
      Observable<any>|Promise<any>|any;
}
export interface CanLoad { canLoad(route: Route):
           Observable<boolean>|Promise<boolean>|boolean; }
```

router_config_loader.ts defines two classes - `LoadedRouterConfig` and `LoadedRouterConfig` – and an opaque token, `ROUTES`. `LoadedRouterConfig` manages three pieces of information, a routes array, an injector and a component factory resolver:

```
export class LoadedRouterConfig {
  constructor(
       public routes: Route[], ]
       public injector: Injector,
       public factoryResolver: ComponentFactoryResolver) {}
}
```

`RouterConfigLoader` loads it:

```
export const ROUTES = new OpaqueToken('ROUTES');

export class RouterConfigLoader {
  constructor(private loader: NgModuleFactoryLoader) {}

  load(parentInjector: Injector, path: string):
                         Observable<LoadedRouterConfig> {
    return fromPromise(this.loader.load(path).then(r => {
      const ref = r.create(parentInjector);
      return new LoadedRouterConfig(
          flatten(ref.injector.get(ROUTES)), ref.injector,
          ref.componentFactoryResolver);
    }));
```

```
    }
  }
```

router_outlet_map.ts defines the `RouterOutletMap` class, a simple wrapper around a map of names to router outlets:

```
export class RouterOutletMap {
  _outlets: {[name: string]: RouterOutlet} = {};
  registerOutlet(name: string, outlet: RouterOutlet): void {
     this._outlets[name] = outlet; }
  removeOutlet(name: string): void { this._outlets[name] = undefined; }
}
```

The router_state.ts file contains these classes (and some helper functions):

- RouterState
- ActivatedRoute
- InheritedResolve
- RouterStateSnapshot

`RouterState` is defined as:

```
export class RouterState extends Tree<ActivatedRoute> {
  constructor(root: TreeNode<ActivatedRoute>,
   public snapshot: RouterStateSnapshot) {
     super(root);
     setRouterStateSnapshot<RouterState, ActivatedRoute>(this, root);
  }
  get fragment(): Observable<string> { return this.root.fragment; }
  toString(): string { return this.snapshot.toString(); }
}
```

`RouterStateSnapshot` is defined as:

```
// The state of the router at a particular moment in time.
export class RouterStateSnapshot extends Tree<ActivatedRouteSnapshot> {
  constructor(
     public url: string,
     root: TreeNode<ActivatedRouteSnapshot>) {
    super(root);
    setRouterStateSnapshot<RouterStateSnapshot, ActivatedRouteSnapshot>(
                                                          this, root);
  }
}
```

The `setRouterStateSnapshot()` function is defined as:

```
function setRouterStateSnapshot<U, T extends{_routerState: U}>(
     state: U, node: TreeNode<T>): void {
  node.value._routerState = state;
  node.children.forEach(c => setRouterStateSnapshot(state, c));
}
```

So its sets the router state for the current node, and then recursively calls `setRouterStateSnapshot()` to set it for all children.

The `ActivatedRoute` class is used byt the router outlet directive to describe the component it has loaded:

```
export class ActivatedRoute {
  _futureSnapshot: ActivatedRouteSnapshot;
  snapshot: ActivatedRouteSnapshot;
  _routerState: RouterState;

  constructor(
       public url: Observable<UrlSegment[]>,
       public params: Observable<Params>,
       public queryParams: Observable<Params>,
       public fragment: Observable<string>,
       public data: Observable<Data>,
       public outlet: string,
       public component: Type|string,
       futureSnapshot: ActivatedRouteSnapshot) {
    this._futureSnapshot = futureSnapshot;
  }
  get routeConfig(): Route { return this._futureSnapshot.routeConfig; }
  get parent(): ActivatedRoute { return this._routerState.parent(this); }
  get firstChild(): ActivatedRoute { return
this._routerState.firstChild(this); }
  get children(): ActivatedRoute[] { return this._routerState.children(this);
}
  get pathFromRoot(): ActivatedRoute[] { return
this._routerState.pathFromRoot(this); }
}
```

router/src/directives

This directory has the following files:

- router_link.ts
- router_link_active.ts
- router_outlet.ts

The router_link.ts file contains the RouterLink directive:

```
@Directive({selector: ':not(a)[routerLink]'})
export class RouterLink {
  private commands: any[] = [];
  @Input() queryParams: {[k: string]: any};
  @Input() fragment: string;
  @Input() preserveQueryParams: boolean;
  @Input() preserveFragment: boolean;
  constructor(
     private router: Router,
     private route: ActivatedRoute,
     private locationStrategy: LocationStrategy) {}
  ..
}
```

The router link commands are set via:

```
 @Input()
  set routerLink(data: any[]|string)
```

When the link is clicked, the `onClick()` method is called:

```
 @HostListener('click', ['$event.button', '$event.ctrlKey',
'$event.metaKey'])
```

```
    onClick(button: number, ctrlKey: boolean, metaKey: boolean): boolean {
      if (button !== 0 || ctrlKey || metaKey) {
        return true;
      }
      this.router.navigateByUrl(this.urlTree);
      return false;
  }
```

The urlTree getter uses `Router.createlUrlTree()`:

```
  get urlTree(): UrlTree {
      return this.router.createUrlTree(this.commands, { .. });
  }
```

The same file also contains the `RouterLinkWithHref` directive:

```
  @Directive({selector: 'a[routerLink]'})
  export class RouterLinkWithHref implements OnChanges, OnDestroy { … }
```

This has a href:

```
   // the url displayed on the anchor element.
    @HostBinding() href: string;
```

and manages the `urlTree` as a field and sets it from the constructor via a call to:

```
  private updateTargetUrlAndHref(): void {
      this.urlTree = this.router.createUrlTree(this.commands, {
        relativeTo: this.route,
        queryParams: this.queryParams,
        fragment: this.fragment,
        preserveQueryParams: toBool(this.preserveQueryParams),
        preserveFragment: toBool(this.preserveFragment)
      });
      if (this.urlTree) {
        this.href = this.locationStrategy.prepareExternalUrl(
                             this.router.serializeUrl(this.urlTree));
      }
  }
```

The router_link_active.ts file contains the `RouterLinkActive` directive:

```
  @Directive({selector: '[routerLinkActive]'})
  export class RouterLinkActive
         implements OnChanges, OnDestroy, AfterContentInit {..}
```

This is used to add a CSS class to an element representing an active route. Its constructor is defiend as:

```
  constructor(private router: Router, private element: ElementRef, private
  renderer: Renderer) {
      this.subscription = router.events.subscribe(s => {
        if (s instanceof NavigationEnd) {
          this.update();
        }
      });
    }
```

Its update method uses the configured renderer to set the element class:

```
    private update(): void {
```

```
    if (!this.links || !this.linksWithHrefs || !this.router.navigated)
      return;

    const isActiveLinks = this.reduceList(this.links);
    const isActiveLinksWithHrefs = this.reduceList(this.linksWithHrefs);
    this.classes.forEach(
        c => this.renderer.setElementClass(
                                this.element.nativeElement,
                                c,
                                isActiveLinks || isActiveLinksWithHrefs));
  }
```

The router_outlet.ts file contains the `RouterOutlet` class:

```
// A router outlet is a placeholder that Angular dynamically
// fills based on the application's route.
@Directive({selector: 'router-outlet'})
export class RouterOutlet implements OnDestroy {
  constructor(
      private parentOutletMap: RouterOutletMap,
      private location: ViewContainerRef,
      private resolver: ComponentFactoryResolver,
      @Attribute('name') private name: string) {
    parentOutletMap.registerOutlet(name ? name : PRIMARY_OUTLET, this);
  }
  ..
}
```

This is where application component whose lifecycle depends on the router live. We note the `ViewContainerRef` and `ComponentFactoryResolver` parameters to the constructor. When its activate method is called, the resolver will be asked to resolve a component factory for the component.

A somewhat simplified version of activate is:

```
activate(
    activatedRoute: ActivatedRoute,
    loadedResolver: ComponentFactoryResolver,
    loadedInjector: Injector,
    providers: ResolvedReflectiveProvider[],
    outletMap: RouterOutletMap): void {

    let factory: ComponentFactory<any> =
                 this.resolver.resolveComponentFactory(component);


    // important:  "location" in the following is ViewContainerRef parameter
    // the passed in to the constructor earlier
    this.activated = this.location.createComponent(
                      factory, this.location.length, inj, []);
    this.activated.changeDetectorRef.detectChanges();
    this.activateEvents.emit(this.activated.instance);
  }
```

There are two uses of the location `ViewContainerRef` field, `createComponent()` and `length`. `ViewContainerRef` is defined in <ANGULAR2>/modules/@angular/core/src/linker/view_container_ref.ts and it has:

```
// Returns the number of Views currently attached to this container.
get length(): number { .. };

// Instantiates a single Component and inserts its Host View into this
// container at the specified `index`.
abstract createComponent<C>(
        componentFactory: ComponentFactory<C>,
        index?: number,
        injector?: Injector,
        projectableNodes?: any[][]): ComponentRef<C>;
```

So the component is appended as the last entry in the `ViewContainer`.

15: @Angular/Compiler-CLI

Overview

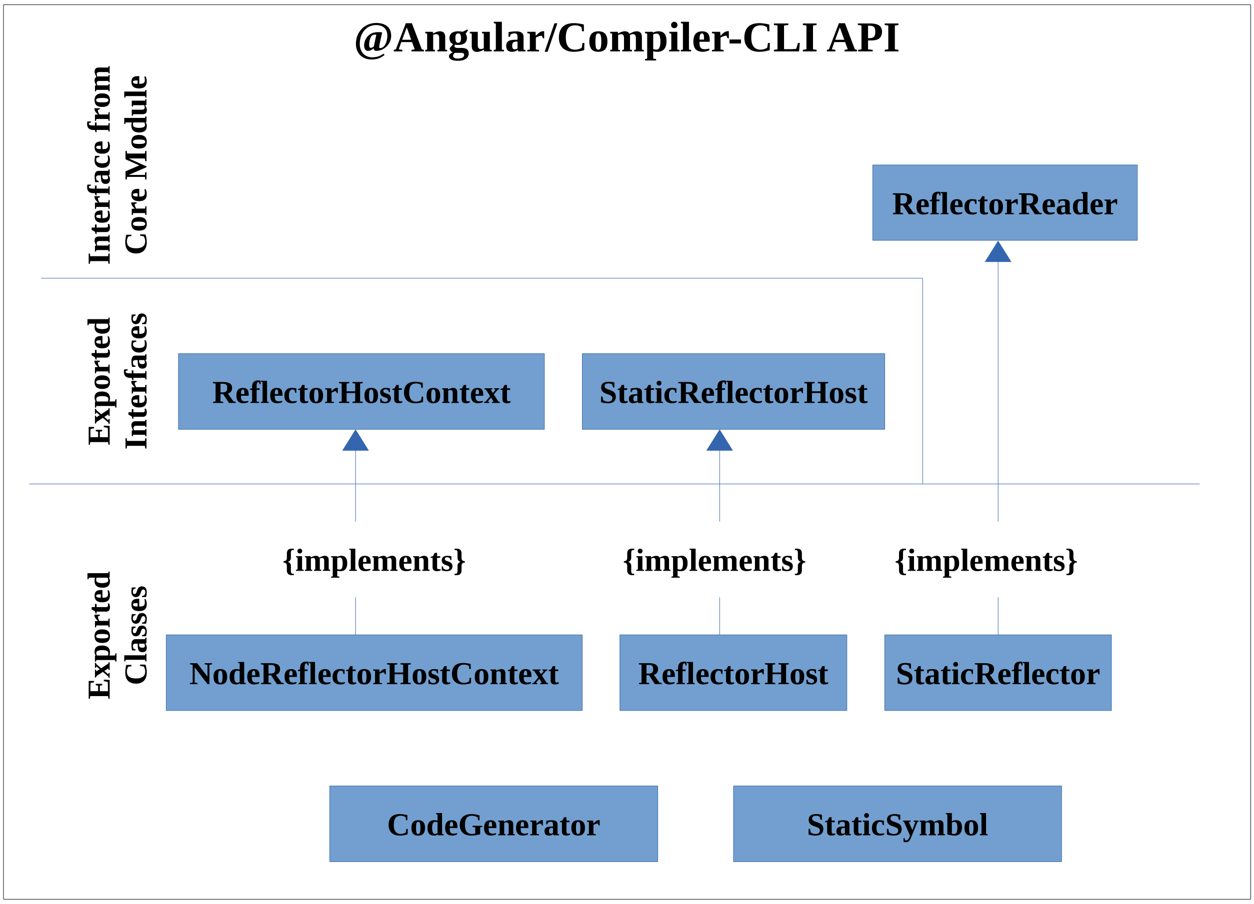
The Compiler-CLI (command line interface) provides two applications – ngc and ng-xi18n - for developers to run as build steps. It also supplies a small API that allows embedding of its functionality within other tools.

Most developers start using Angular with the QuickStart application layout that uses the runtime template compiler. Their beginner applications call `bootstrapModule()` - but as their applications get larger and there is a business demand to deliver as small as possible downloads and as fast as possible launch time, interest grows in the idea of performing template compilation ahead of time, as a build step on the developer's computer.

This is where ngc comes in. A second requirement is to support internationalization and this is where ng-xi18n comes in.

Compiler-CLI API

The exported API of the @AngularCompiler-CLI package can be represented as:



It's index.ts is defines the following exports:

```
export {CodeGenerator} from './src/codegen';
```

```
export {NodeReflectorHostContext, ReflectorHost, ReflectorHostContext}
from './src/reflector_host';

export {StaticReflector, StaticReflectorHost, StaticSymbol} from
'./src/static_reflector';

export * from '@angular/tsc-wrapped';
```

The tsc-wrapped package (from tools/@angular/tsc_wrapped) is a wrapper around the main TypeScript tsc compiler. Its purpose is explained in <ANGULAR2>/tools/@angular/tsc-wrapped/readme.md:

```
"This package is an internal dependency used by @angular/compiler-cli. Please
use that instead. This is a wrapper around TypeScript's `tsc` program that
allows us to hook in extra extensions."
```

Source Tree Layout

The source tree for the Compiler-CLI package contains these directories:

- integrationtest
- src
- test

The Compiler-CLI main directory contains these files:

- index.ts
- package.json
- readme.md
- rollup.config.js
- tsconfig.json

The readme.md is very detailed and explains the purpose (and usage) of compiler-cli. Here is an extract:

```
# Angular Template Compiler

Angular applications are built with templates, which may be `.html` or `.css`
files, or may be inline `template` attributes on Decorators like
`@Component`.

These templates are compiled into executable JS at application runtime
(except in `interpretation` mode). This compilation can occur on the client,
but it results in slower bootstrap time, and also requires that the compiler
be included in the code downloaded to the client.

You can produce smaller, faster applications by running Angular's compiler as
a build step, and then downloading only the executable JS to the client.
```

It is recommended reading the entire readme.md in your favorite markdown viewer (if using Visual studio Code, open the .md file, and select CTRL-SHIFT-V to get nicely formatted text).

The package.json file in the root directory includes:

```
  "name": "@angular/compiler-cli",
  "version": "0.0.0-PLACEHOLDER",
  "description": "Execute angular2 template compiler in nodejs.",
```

```
    "main": "index.js",
    "typings": "index.d.ts",
    "bin": {
      "ngc": "./src/main.js",
      "ng-xi18n": "./src/extract_i18n.js"
    },
    "dependencies": {
      "@angular/tsc-wrapped": "^0.2.2",
      "reflect-metadata": "^0.1.2",
      "parse5": "1.3.2",
      "minimist": "^1.2.0"
    },
    "peerDependencies": {
      "typescript": "^1.9.0-dev",
      "@angular/compiler": "0.0.0-PLACEHOLDER",
      "@angular/platform-server": "0.0.0-PLACEHOLDER",
      "@angular/core": "0.0.0-PLACEHOLDER"
    },
```

Note the two bin entries, which map to two entry points we need to explore, in main.ts and extract_i18n.ts.

Source

The other exported types are defined in <ANGULAR2>/modules/@angular/compiler-cli/src in these source files:

- codegen.ts
- compiler_private.ts
- core_private.ts
- extract_i18n.ts
- main.ts
- path_mapped_reflector.host.ts
- reflector_host.ts
- static_reflection_capabilities.ts
- static_reflector.ts

main.ts contains this code:

```
function codegen(
     ngOptions: tsc.AngularCompilerOptions,
     cliOptions: tsc.NgcCliOptions,
     program: ts.Program,
     host: ts.CompilerHost) {
3 return CodeGenerator.create(
            ngOptions, cliOptions, program, host).codegen();
}

// CLI entry point
if (require.main === module) {
1 const args = require('minimist')(process.argv.slice(2));
  const project = args.p || args.project || '.';
  const cliOptions = new tsc.NgcCliOptions(args);
2 tsc.main(project, cliOptions, codegen).then(exitCode =>
process.exit(exitCode)).catch(e => {
    console.error(e.stack);
```

```
      console.error('Compilation failed');
      process.exit(1);
    });
  }
```

We see main uses the minimist library to access the arguments 1; it then has a call to `tsc.main()` 2 passing in a function named `codegen()`. Earlier in the file we see that `codegen()` function defined – it has a call to `CodeGenerator.create()`3.

The codegen.ts file defines the `CodeGenerator` class. It has four methods:

- `create()` - static that creates an offline compiler and instantiates CodeGenerator, which it returns
- `codegen()` - actual compilation call to offline compiler happens here
- `calculateEmitPath()` - helper used for directory structure
- `readFileMetadata()` - metadata access via static reflector

The static_reflection_capabilities.ts file implements the `StaticAndDynamicReflectionCapabilities` class. This has a static `install()` method which makes `StaticAndDynamicReflectionCapabilities` available to the reflector, which in turn is used by the compilation engine:

```
static install(staticDelegate: StaticReflector) {
    reflector.updateCapabilities(
       new StaticAndDynamicReflectionCapabilities(staticDelegate));
  }
```

The reflector_host.ts file defines the ReflectorHostContext interface, which handles file-related queries:

```
export interface ReflectorHostContext {
  fileExists(fileName: string): boolean;
  directoryExists(directoryName: string): boolean;
  readFile(fileName: string): string;
  assumeFileExists(fileName: string): void;
}
```

One implementation of this is supplied, based on Node:

```
export class NodeReflectorHostContext implements ReflectorHostContext {
  private assumedExists: {[fileName: string]: boolean} = {};
  fileExists(fileName: string): boolean {
    return this.assumedExists[fileName] || fs.existsSync(fileName); }
  directoryExists(directoryName: string): boolean {
    try {
      return fs.statSync(directoryName).isDirectory();
    } catch (e) { return false; } }
  readFile(fileName: string): string {
     return fs.readFileSync(fileName, 'utf8'); }
  assumeFileExists(fileName: string): void {
     this.assumedExists[fileName] = true; }
}
```

This file also defines the ReflectorHost class, which handles the interaction between the reflector and the host, such as making additional import locations available to the reflector and lots more.

The static_reflector.ts file implements the StaticReflectorHost interface and StaticReflector class:

```
/**
* The host of the static resolver is expected to be able to provide module
* metadata in the form of ModuleMetadata. Angular CLI will produce this
* metadata for a module whenever a .d.ts files is produced and the module has
* exported variables or classes with decorators. Module metadata can
* also be produced directly from TypeScript sources by using
* MetadataCollector in tools/metadata.
*/
export interface StaticReflectorHost {
  getMetadataFor(modulePath: string): {[key: string]: any};
  findDeclaration(
     modulePath: string,
     symbolName: string,
     containingFile?: string): StaticSymbol;
  getStaticSymbol(
     declarationFile: string, name: string, members?: string[]): StaticSymbol;
  angularImportLocations(): {
    coreDecorators: string,
    diDecorators: string,
    diMetadata: string,
    diOpaqueToken: string,
    animationMetadata: string,
    provider: string
  };
}
```

This file also defines the StaticReflector class, a large 500-line class whose task is to:

```
/**
* A static reflector implements enough of the Reflector API that
* is necessary to compile templates statically.
 */
export class StaticReflector implements ReflectorReader { .. }
```

Normally with reflection the code to be reflected over is actually running. However, with the StaticReflector (and Compiler-CLI in general), this is not the case, and hence the need to statically (without running the code), access decorators in the code.

The extract_i18n.ts file implements the main and helper functions for internationalization. Its main just calls tsc.main passing in an extract function:

```
tsc.main(project, cliOptions, extract)
```

extract() supportedboth xmb and xliff localization string formats and is implemented as:

```
function extract(
     ngOptions: tsc.AngularCompilerOptions,
     cliOptions: tsc.I18nExtractionCliOptions,
     program: ts.Program, host: ts.CompilerHost) {
  const htmlParser = new compiler.i18n.HtmlParser(new HtmlParser());
  const extractor = Extractor.create(
                ngOptions, cliOptions.i18nFormat, program, host, htmlParser);
  const bundlePromise: Promise<compiler.i18n.MessageBundle> =
                extractor.extract();
```

```
    return (bundlePromise).then(messageBundle => {
      let ext: string;
      let serializer: compiler.i18n.Serializer;
      const format = (cliOptions.i18nFormat || 'xlf').toLowerCase();

      switch (format) {
        case 'xmb':
          ext = 'xmb';
          serializer = new compiler.i18n.Xmb();
          break;
        case 'xliff':
        case 'xlf':
        default:
          ext = 'xlf';
          serializer = new compiler.i18n.Xliff(
                          htmlParser, compiler.DEFAULT_INTERPOLATION_CONFIG);
          break;
      }

      const dstPath = path.join(ngOptions.genDir, `messages.${ext}`);
      host.writeFile(dstPath, messageBundle.write(serializer), false);
    });
  }
```

It uses the static `Extractor.create()` function to construct an `Extractor` instance and then calls `extractor.extract()`. The `create()` method in the `Extractor` class does some initialization and then returns a new instance of Extractor:

```
static create(
  options: tsc.AngularCompilerOptions,
  translationsFormat: string, program: ts.Program,
  compilerHost: ts.CompilerHost, htmlParser: compiler.i18n.HtmlParser,
  reflectorHostContext?: ReflectorHostContext): Extractor {

  const resourceLoader: compiler.ResourceLoader = { .. }
  const urlResolver: compiler.UrlResolver =
        compiler.createOfflineCompileUrlResolver();
  const reflectorHost = new ReflectorHost(
        program, compilerHost, options, reflectorHostContext);
  const staticReflector = new StaticReflector(reflectorHost);
  StaticAndDynamicReflectionCapabilities.install(staticReflector);

  const config = new compiler.CompilerConfig(..);
  const normalizer = new DirectiveNormalizer(
     resourceLoader, urlResolver, htmlParser, config);
  const expressionParser = new Parser(new Lexer());
  const elementSchemaRegistry = new DomElementSchemaRegistry();
  const console = new Console();
  const tmplParser =
      new TemplateParser(
        expressionParser, elementSchemaRegistry, htmlParser, console, []);
  const resolver = new CompileMetadataResolver(
      new compiler.NgModuleResolver(staticReflector),
      new compiler.DirectiveResolver(staticReflector),
      new compiler.PipeResolver(staticReflector),
      elementSchemaRegistry,
```

```
          staticReflector);
      const offlineCompiler = new compiler.OfflineCompiler(
          resolver, normalizer, tmplParser,
          new StyleCompiler(urlResolver),
          new ViewCompiler(config),
          new NgModuleCompiler(),
          new TypeScriptEmitter(reflectorHost), null, null);
      let messageBundle = new compiler.i18n.MessageBundle(htmlParser, [], {});
     return new Extractor(program, compilerHost, staticReflector,
          messageBundle, reflectorHost, resolver, normalizer, offlineCompiler);
    }
  }
```

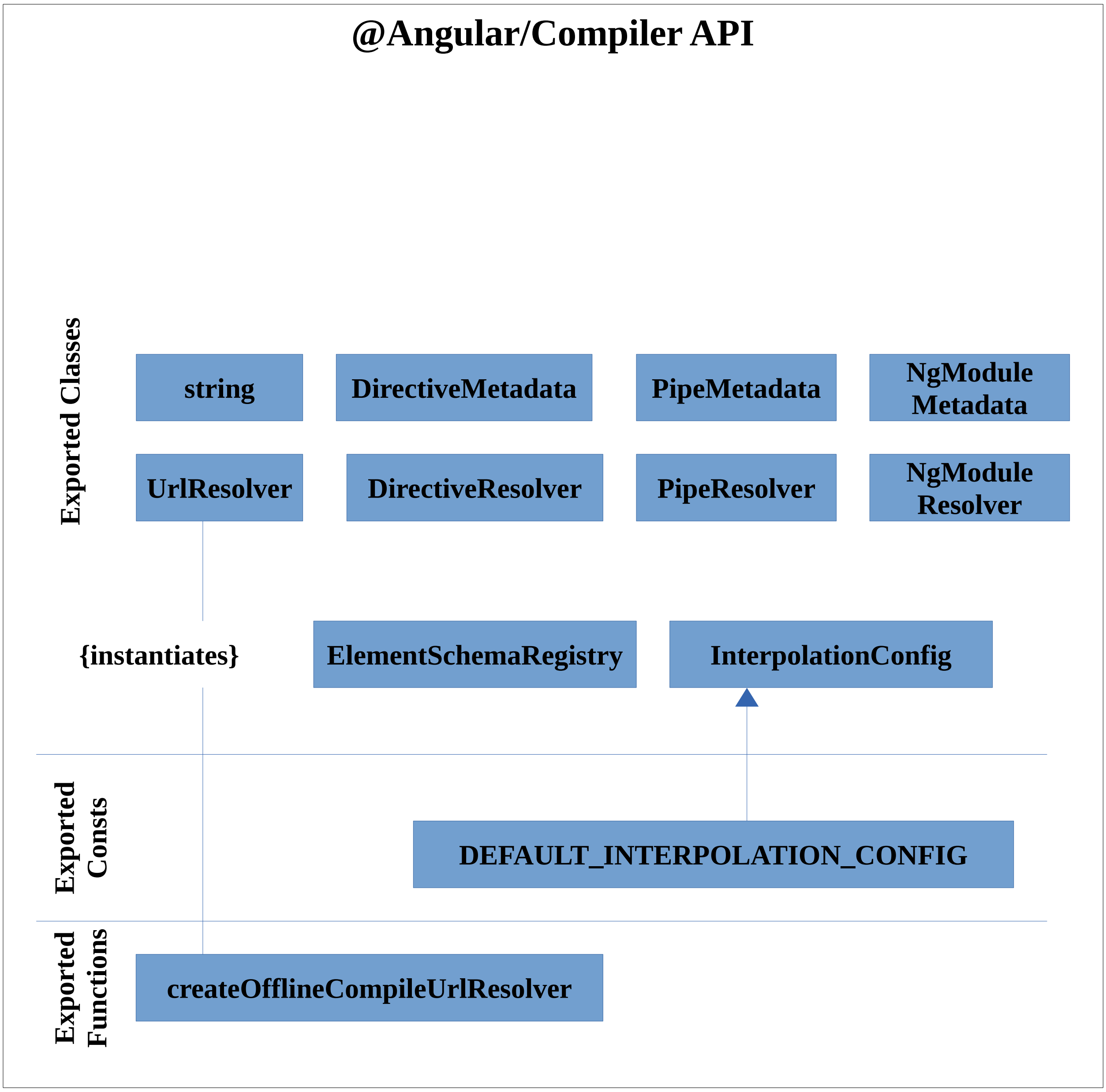
16: @Angular/Compiler

Overview

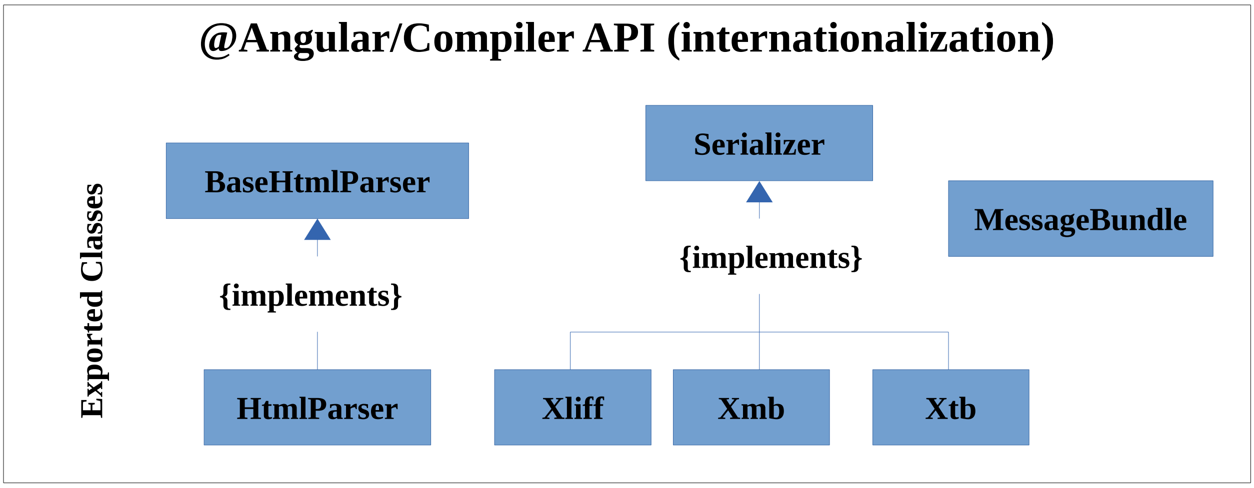
The compiler module provides both runtime and offline template compilation services. It also helps with internationalization (i18n) and it supplies a registry of elements (`ElementSchemaRegistry`) with appropriate security contexts.

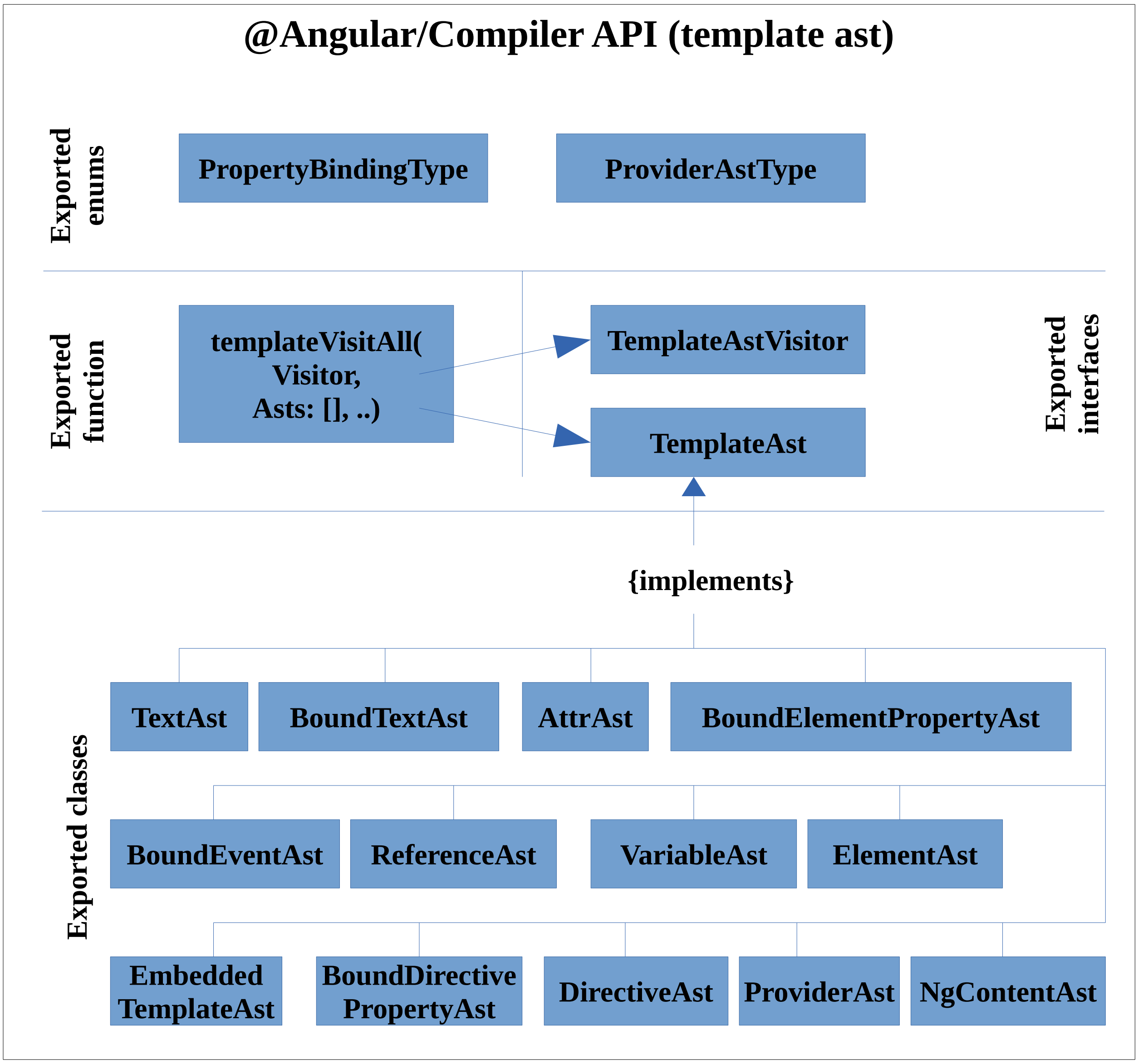
Compiler API

The exported API of the @AngularCompiler module can be sub-divied into a number of areas.



TBD


@Angular/Compiler API (internationalization)
Exported Classes
BaseHtmlParser
{implements}
HtmlParser
Serializer
{implements}
Xliff
Xmb
Xtb
MessageBundle



index.ts lists the public exports which come from five files – compiler.ts (we can divide its exports into metadata1a, general1b and resolvers1c), interpolation_config2, element_schema_registry3, i18n4 and template_ast5:

```
export {
1a
CompileDiDependencyMetadata, CompileDirectiveMetadata,
CompileFactoryMetadata, CompileIdentifierMetadata,
CompileMetadataWithIdentifier, CompilePipeMetadata, CompileProviderMetadata,
CompileQueryMetadata, CompileTemplateMetadata, CompileTokenMetadata,
CompileTypeMetadata,
1b
COMPILER_PROVIDERS, OfflineCompiler, RuntimeCompiler, RenderTypes,
ResourceLoader, CompilerConfig, DEFAULT_PACKAGE_URL_PROVIDER, SourceModule,
TEMPLATE_TRANSFORMS, platformCoreDynamic,
1c
```

```
DirectiveResolver, NgModuleResolver, PipeResolver, UrlResolver,
createOfflineCompileUrlResolver} from './src/compiler';
2
export {DEFAULT_INTERPOLATION_CONFIG, InterpolationConfig}
  from './src/ml_parser/interpolation_config';
3
export {ElementSchemaRegistry} from './src/schema/element_schema_registry';
4
export {i18n};
5
export * from './src/template_parser/template_ast';

export * from './private_export';
```

- Compiler
- Resolver
- Ast
- Lexer
- Parser
- Reflector

TBD

compiler/src/schema

This directory has the following files:

- dom_element_schema_registry,ts
- dom_security_schema.ts
- element_schema_registry.ts

The `SchemaMetadata` interface is declared in Core's src/metadata/ng_module.ts:

```
// Interface for schema definitions in @NgModules.
export interface SchemaMetadata { name: string; }
```

This is imported into element_schema_registry.ts and used to define the `ElementSchemaRegistry` class:

```
import {SchemaMetadata} from '@angular/core';

export abstract class ElementSchemaRegistry {
  abstract hasProperty(
     tagName: string,
     propName: string,
     schemaMetas: SchemaMetadata[]): boolean;
  abstract securityContext(tagName: string, propName: string): any;
  abstract getMappedPropName(propName: string): string;
  abstract getDefaultComponentElementName(): string;
}
```

It contains only abstract methods that a derived class needs to implement to return information about elements.

We have already seen that the SecurityContext enum is defined in Core's src/security.ts class:

```
export enum SecurityContext {
  NONE,
  HTML,
  STYLE,
  SCRIPT,
  URL,
  RESOURCE_URL,
}
```

It is used in compiler/src/schema/dom_security_schema.ts which defines a map from either a tag name or a property name, to a security context. Note the scary warning at the top of the file, followed by the definition of `SECURITY_SCHEMA`:

```
//=========================================================================
//==========================================================================
//=========== S T O P   -  S T O P   -  S T O P   -  S T O P  =============
//==========================================================================
//==========================================================================
//
//           DO NOT EDIT THIS LIST OF SECURITY SENSITIVE PROPERTIES
//           WITHOUT A SECURITY REVIEW!
//           Reach out to mprobst for details
//
//==========================================================================
/** Map from tagName|propertyName SecurityContext.
Properties applying to all tags use '*'. */
export const SECURITY_SCHEMA: {[k: string]: SecurityContext} = {};
```

The `registerContext()` method adds an entry (or a set of entries) to `SECURITY_SCHEMA` for the specified context:

```
function registerContext(ctx: SecurityContext, specs: string[]) {
  for (let spec of specs) SECURITY_SCHEMA[spec.toLowerCase()] = ctx;
}
```

This called four times with lists of items to be asspociated with the security context:

```
registerContext(SecurityContext.HTML, ..);
registerContext(SecurityContext.STYLE, ..);
registerContext(SecurityContext.URL, ..);
registerContext(SecurityContext.RESOURCE_URL, ..);
```

The dom_element_schema_registry.ts file defines the injectable `DomElementSchemaRegistry` class. Again, this file has a warning not to edit without a detailed security review. Application developers may beenfit from reading the file to learn more about security handling, but in general should not edit the file.

```
@Injectable()
export class DomElementSchemaRegistry extends ElementSchemaRegistry {
  schema = <{[element: string]: {[property: string]: string}}>{};
```

It manages a schema which maps elements (identified by string names) each to a map of property names to property values (also string-based):

```
schema = <{[element: string]: {[property: string]: string}}>{};
```

This schema is populated in the constructor.

The class also has a `securityContext()` method that uses `SECURITY_SCHEMA` we have just seen to return a context if specifially configured 1, or if not there by th eproperty name is define diwth the '*' wildcard the use it 2, or else it simply returns `SecurityContext.NONE` 3:

```
/**
 * securityContext returns the security context for the given property on
 * the given DOM tag.
 *
 * Tag and property name are statically known and cannot change at runtime,
 * i.e. it is not possible to bind a value into a changing attribute or tag
 * name.
 *
 * The filtering is white list based. All attributes in the schema above
 * are assumed to have the 'NONE' security context, i.e. that they are safe
 * inert string values. Only specific well known attack vectors are assigned
 * their appropriate context.
 */
securityContext(tagName: string, propName: string): SecurityContext {
   // Make sure comparisons are case insensitive,
   // so that case differences between attribute and
   // property names do not have a security impact.
   tagName = tagName.toLowerCase();
   propName = propName.toLowerCase();
   let ctx = SECURITY_SCHEMA[tagName + '|' + propName];
1  if (ctx !== undefined) return ctx;
2  ctx = SECURITY_SCHEMA['*|' + propName];
  return ctx !== undefined ? ctx : 3 SecurityContext.NONE;
 }
```

17: Tsickle

Overview

Tsickle is a small utility used to transpile from TypeScript to JavaScript and adds annotations (in the form of JSDoc comments) for the Google Closure Compiler to further optimize the generated JavaScript code. To learn more about Closure, visit:

- https://github.com/google/closure-compiler/

Tsickle is used by tsc-wrapped, the compilation utility used to build Angular - located in the main Angular project under *tools/@angular* (in contrast, in the main Angular project, most of the source is located under *modules/@angular*).

Source Tree Layout

The Tsickle source tree has these sub-directories:

- src
- test
- test_files
- third_party

The main directory has these important files:

- readme.md
- package.json
- gulpfile.js
- tsconfig.json

The readme.md contains useful information about the project, including this important guidance about the use of tsconfig.json:

> *Tsickle works by wrapping `tsc`. To use it, you must set up your project such that it builds correctly when you run `tsc` from the command line, by configuring the settings in `tsconfig.json`.*
>
> *If you have complicated tsc command lines and flags in a build file (like a gulpfile etc.) Tsickle won't know about it. Another reason it's nice to put everything in `tsconfig.json` is so your editor inherits all these settings as well.*

The package.json file contains:

```
"main": "build/src/tsickle.js",
"bin": "build/src/main.js",
```

The gulpfile.js file contains the following Gulp tasks:

- gulp watch
- gulp test.e2e (end-to-end tests)
- gulp test.check-format (formatting tests)
- gulp test.unit (unit tests)
- gulp test (runs all three of above)

The src Sub-Directory

The src sub-directory contains the following source files:

- cli_support.ts
- decorator_annotator.ts
- es5processor.ts
- jsdoc.ts
- main.ts
- rewriter.ts
- tsickle.ts
- type-translator.ts
- util.ts

main.ts is where the call to tsickle starts executing and tsickle.ts is where the core logic is – the other files are helpers.

The entry point at the bottom of main.ts calls the main function passing in the argument list as an array of strings.

```
function main(args: string[]): number {
1 let {settings, tscArgs} = loadSettingsFromArgs(args);
  let diagnostics: ts.Diagnostic[] = [];
2 let config = loadTscConfig(tscArgs, diagnostics);
  ..
  // Run tsickle+TSC to convert inputs to Closure JS files.
3 let closure = toClosureJS(
       config.options, config.fileNames, settings, diagnostics);
  ..
4 for (let fileName of toArray(closure.jsFiles.keys())) {
    mkdirp.sync(path.dirname(fileName));
    fs.writeFileSync(fileName, closure.jsFiles.get(fileName));
  }

5 if (settings.externsPath) {
    mkdirp.sync(path.dirname(settings.externsPath));
    fs.writeFileSync(settings.externsPath, closure.externs);
  }
  return 0;
}
```

The main function first loads the settings 1 from the args and 2 the tsc config. Then it calls the `toClosureJs()` function 3, and outputs to a file 4 each resulting JavaScript file. If `externsPath` is set in settings, they too are written out to files 5.

The `loadSettingsfromArgs()` function handles the command-line arguments, which can be a mix of tsickle-specific arguments and regular tsc arguments. The tsickle-specific arguments are –externs (generate externs file) and –untyped (every TypeScript type becomes a Closure {?} type).

The toClosureJs() function is where the transformation occurs. It returns 1 a map of transformed file contents, optionally with externs information, it so configured.

```
function toClosureJS(
    options: ts.CompilerOptions, fileNames: string[], settings: Settings,
```

```
    allDiagnostics: ts.Diagnostic[]):
1     {jsFiles: Map<string, string>, externs: string}|null {
  // Parse and load the program without tsickle processing.
  // This is so:
  // - error messages point at the original source text
  // - tsickle can use the result of typechecking for annotation
2 let program = ts.createProgram(fileNames, options);
  {
    let diagnostics = ts.getPreEmitDiagnostics(program);
    if (diagnostics.length > 0) {
      allDiagnostics.push(...diagnostics);
      return null;
    }
  }
  ..
  // Process each input file with tsickle and save the output.
  const tsickleOutput = new Map<string, string>();
  let tsickleExterns = '';
  for (let fileName of fileNames) {
    let {output, externs, diagnostics} =
3       tsickle.annotate(program, program.getSourceFile(fileName),
                         tsickleOptions);
    ..
4   tsickleOutput.set(ts.sys.resolvePath(fileName), output);
    if (externs) { tsickleExterns += externs; }
  }

  // Reparse and reload the program, inserting the tsickle output in
  // place of the original source.
5 let host = createSourceReplacingCompilerHost(
                    tsickleOutput, ts.createCompilerHost(options));
6 program = ts.createProgram(fileNames, options, host);

..

  return {jsFiles, externs: tsickleExterns};
}
```

It calls 2 TypeScript's `createProgram` method with the original program source to ensure it is syntatically correct and any error messages refer the original source, not the modified source.

Then it calls 3 `tsickle.annotate()` to annotate the source and adds the result to the map that will be returned as the result of the function call. Then it calls 5 the `createSourceReplacingCompilerHost()` function to construct an alternative Compiler Host, which is then passed to the second call 6 to TypeScript's `createProgram` method.

The TypeScript compiler uses a compiler host for all interaction with the hosting environment (such as locating source files). The `createSourceReplacingCompilerHost()` function creates a TypeScript compiler host that is almost a copy of a passed in compile host (e.g. the default), with the difference that a map of source file names t overlay source text called substituteSource is used in order to locate source code. This is used 1 in the local function `getSourceFile()` to find the source.

```
/**
 * Constructs a new ts.CompilerHost that overlays sources in substituteSource
 * over another ts.CompilerHost.
 *
 * @param substituteSource A map of source file name -> overlay source text.
 */
function createSourceReplacingCompilerHost(
    substituteSource: Map<string, string>, delegate: ts.CompilerHost):
ts.CompilerHost {
  return {
1   getSourceFile,
    getCancellationToken: delegate.getCancellationToken,
    getDefaultLibFileName: delegate.getDefaultLibFileName,
    writeFile: delegate.writeFile,
    getCurrentDirectory: delegate.getCurrentDirectory,
    getCanonicalFileName: delegate.getCanonicalFileName,
    useCaseSensitiveFileNames: delegate.useCaseSensitiveFileNames,
    getNewLine: delegate.getNewLine,
    fileExists: delegate.fileExists,
    readFile: delegate.readFile,
    directoryExists: delegate.directoryExists,
    getDirectories: delegate.getDirectories,
  };
  function getSourceFile(
      fileName: string, languageVersion: ts.ScriptTarget,
      onError?: (message: string) => void): ts.SourceFile {
    let path: string = ts.sys.resolvePath(fileName);
    let sourceText = substituteSource.get(path);
    if (sourceText) {
      return ts.createSourceFile(path, sourceText, languageVersion);
    }
    return delegate.getSourceFile(path, languageVersion, onError);
  }
}
```

We have seen that the annotate function from the tsickle source file is called from `toClosureJS()`. It is a simple function:

```
export function annotate(
program: ts.Program, file: ts.SourceFile, options: Options = {}): Output {
  assertTypeChecked(file);
  return new Annotator(program, file, options).annotate();
}
```

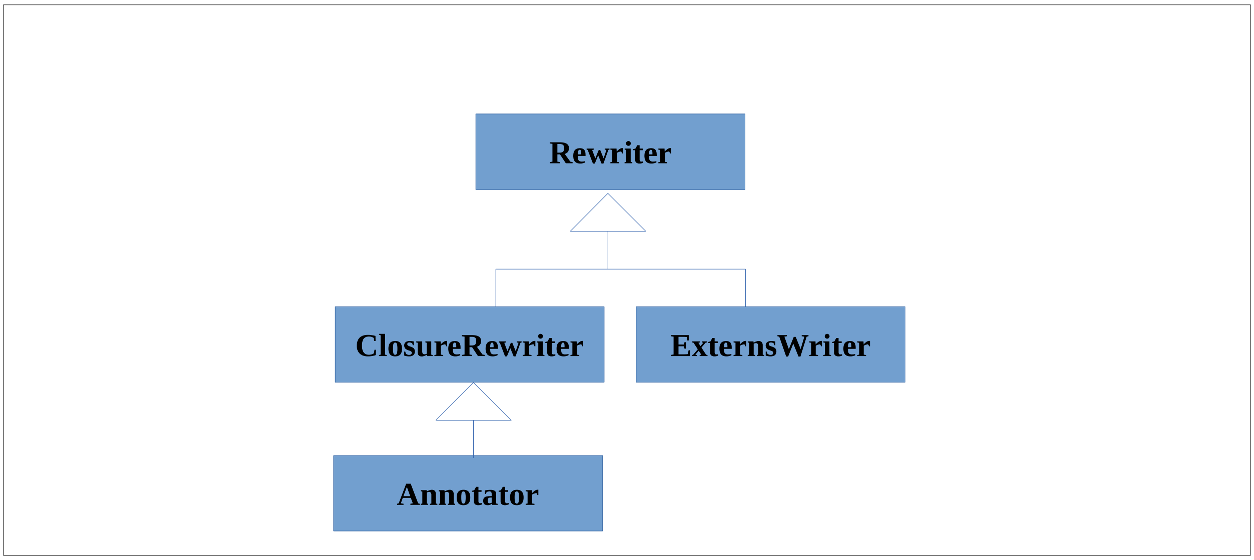
So it uses the `Annotator` class and returns an `Output` instance. `Output` is an interface defined as:

```
export interface Output {
  /** The TypeScript source with Closure annotations inserted. */
  output: string;
  /** Generated externs declarations, if any. */
  externs: string|null;
  /** Error messages, if any. */
  diagnostics: ts.Diagnostic[];
  /** A source map mapping back into the original sources. */
  sourceMap: SourceMapGenerator;
}
```

Classes called rewriters are used to rewrite the source. The rewriter.ts file has the rewriter abstract class. An important method is `maybeProcess()`.

```
/**
 * A Rewriter manages iterating through a ts.SourceFile, copying input
 * to output while letting the subclass potentially alter some nodes
 * along the way by implementing maybeProcess().
 */
export abstract class Rewriter {
..
/**
* maybeProcess lets subclasses optionally processes a node.
*
* @return True if the node has been handled and doesn't need to be traversed;
*         false to have the node written and its children recursively visited.
*/
  protected maybeProcess(node: ts.Node): boolean {
    return false;
  }
}
```

tsickle.ts has some classes that derive from `Rewriter`, according to this hierarchy:



`Annotator.maybeProcess()` is where the actual rewriting occurs.

18: TS-API-Guardian

Overview

Ts-api-guardian is a small tool that tracks a package's public API.

It is used in the Angular build to check for changes to the Angular public API and to ensure that inadvertent changes to the public API are detected. Specifically, it you examine gulpfile.ts in the main Angular project:

- <ANGULAR-MASTER>/gulpfile.js

and look at two tasks named 'public-api:enforce' and 'public-api:update' we see how ts-api-guardian is used, to generate a "golden file" representing the API, and to ensure it has not been unexpectedly changed:

```
// Enforce that the public API matches the golden files
// Note that these two commands work on built d.ts files instead of the source
gulp.task('public-api:enforce', (done) => {
  const childProcess = require('child_process');

  childProcess
      .spawn(
          path.join(__dirname, platformScriptPath(`/node_modules/.bin/ts-api-guardian`)),
          ['--verifyDir', publicApiDir].concat(publicApiArgs), {stdio: 'inherit'})
      .on('close', (errorCode) => {
        if (errorCode !== 0) {
          done(new Error(
              'Public API differs from golden file. Please run `gulp public-api:update`.'));
        } else {
          done();
        }
      });
});

// Generate the public API golden files
gulp.task('public-api:update', ['build.sh'], (done) => {
  const childProcess = require('child_process');

  childProcess
      .spawn(
          path.join(__dirname, platformScriptPath(`/node_modules/.bin/ts-api-guardian`)),
          ['--outDir', publicApiDir].concat(publicApiArgs), {stdio: 'inherit'})
      .on('close', done);
});
```

Source Tree

The ts-api-guardian source tree contains three top-level directories:

- bin
- lib
- test

The main directory of ts-api-guardian contains:

- gulpfile.js
- package.json

- tsconfig.json
- tsd.json

The primary tasks in the gulpfile are:

- compile
- test.compile
- test.unit
- watch

The unit tests are based on mocha (unlike most of the rest of Angular, which uses jasmine).

The package.json has the following dependencies:

```
"dependencies": {
    "chalk": "^1.1.3",
    "diff": "^2.2.3",
    "minimist": "^1.2.0",
    "typescript": "1.7.3"
  },
```

The most important of these is the diff package, which is used to determine differences between blocks of text (https://www.npmjs.com/package/diff).

Package.json also list the single callable program inside ts-api-guardian:

```
"bin": {
    "ts-api-guardian": "./bin/ts-api-guardian"
},
```

bin

This is the single file inside the bin sub-directory, which contains just a single line:

```
require('../build/lib/cli').startCli();
```

lib

The lib sub-directory contains these files:

- cli.ts – command-line interface, processes argument list and invokes commands
- main.ts – main logic for generating and verifying golden files
- serializer.ts – code to serialize an API (to create the contents of a golden file)

A golden file is a textual representation of an API and the two key tasks of ts-api-guardian is to either create or verify golden files based on supplied command line arguments.

Cli.ts starts with some useful comments about how to call ts-api-guardian:

```
// # Generate one declaration file
// ts-api-guardian --out api_guard.d.ts index.d.ts
//
// # Generate multiple declaration files
// # (output location like typescript)
// ts-api-guardian --outDir api_guard [--rootDir .]
                 core/index.d.ts core/testing.d.ts
```

```
//
// # Print usage
// ts-api-guardian --help
//
// # Check against one declaration file
// ts-api-guardian --verify api_guard.d.ts index.d.ts
//
// # Check against multiple declaration files
// ts-api-guardian --verifyDir api_guard [--rootDir .]
                  core/index.d.ts core/testing.d.ts
```

cli.ts accepts the following command line options:

```
--help                          Show this usage message
--out <file>                    Write golden output to file
--outDir <dir>                  Write golden file structure to directory
--verify <file>                 Read golden input from file
--verifyDir <dir>               Read golden file structure from directory
--rootDir <dir>                 Specify the root directory of input files
--stripExportPattern <regexp>   Do not output exports matching the pattern
--allowModuleIdentifiers <id>   Whitelist identifier for "* as foo" imports
--onStabilityMissing <warn|error|none> Warn or error if an export has no
                                       stability annotation`);
```

The Angular API allows annotations to be attached to each API indicating whether it is stable, deprecated or experiemental. The `onStabilityMissing` option indicates what action is required if such an annotation is missing. The `startCli()` function parses the command line and initializes an instance of `SerializationOptions`, and then for generation mode calls `generateGoldenFile()` or for verification mode calls `verifyAgainstGoldenFile()` - both are in main.ts and are actually quite short functions:

```
export function generateGoldenFile(
     entrypoint: string,
     outFile: string, options: SerializationOptions = {}): void {
  const output = publicApi(entrypoint, options);
  ensureDirectory(path.dirname(outFile));
  fs.writeFileSync(outFile, output);  }
```

generateGoldenFile calls publicApi (from Serializer.ts) to generate the contents of the golden file and then writes it to a file. VerifyAgainstGoldenFile() also calls publicApi and saves the result in a string called actual, and then loads the existing golden file data into a string called expected, and then compares then. If then are different, it calls createPatch (from the diff package), to create a representation of the differences between the actual and expected golden files.

```
export function verifyAgainstGoldenFile(
     entrypoint: string, goldenFile: string,
     options: SerializationOptions = {}): string {
  const actual = publicApi(entrypoint, options);
  const expected = fs.readFileSync(goldenFile).toString();

  if (actual === expected) {
    return '';
  } else {
    const patch = createPatch(
```

```
            goldenFile, expected, actual, 'Golden file', 'Generated API');

      // Remove the header of the patch
      const start = patch.indexOf('\n', patch.indexOf('\n') + 1) + 1;

      return patch.substring(start);
    }
  }
```

serializer.ts defines `SerializationOptions` which has three optional properties:

```
export interface SerializationOptions {
  /**
   * Removes all exports matching the regular expression.
   */
  stripExportPattern?: RegExp;
  /**
   * Whitelists these identifiers as modules in the output. For example,
   * ```
   * import * as angular from './angularjs';
   *
   * export class Foo extends angular.Bar {}
   * ```
   * will produce `export class Foo extends angular.Bar {}` and requires
   * whitelisting angular.
   */
  allowModuleIdentifiers?: string[];
  /**
   * Warns or errors if stability annotations are missing on an export.
   * Supports experimental, stable and deprecated.
   */
  onStabilityMissing?: string;  // 'warn' | 'error' | 'none'
}
```

Serializer.ts defines a public API function which just calls `publicApiInternal()`, which in turn calls `ResolvedDeclarationEmitter()`, which is a 200-line class where the actual work is performed. It has three methods which perform the serialization:

- emit(): string
- private getResolvedSymbols(sourceFile: ts.SourceFile): ts.Symbol[]
- emitNode(node: ts.Node)